REASONING INTO CODE

250 deliberate katas for coding interviews

State. Invariants. Protocols. Working Python.

Arunank Divya Sharan

First edition | September 2026

Contents

Module 01 / Execution Grammar and State Mutation **14**

001. Running Total . . . 15
002. Running Minimum and Maximum . . . 18
003. Maximum Value and First Index . . . 21
004. Count Positives, Negatives, and Zeros . . . 24
005. First Cumulative Threshold Crossing . . . 27
006. Longest Positive Streak . . . 30
007. Longest Alternating-Parity Streak . . . 33
008. Run-Length Encoding . . . 36
009. Split on Sentinels . . . 39
010. Peak Active Balance . . . 42

Module 02 / Hash Maps, Sets, and Frequency State **45**

011. Two-Sum Indices . . . 46
012. First Repeated Value . . . 49
013. Valid Anagram . . . 52
014. First Unique Character . . . 56
015. Group Anagrams . . . 60
016. Most Frequent with Earliest Tie . . . 63
017. Isomorphic Strings . . . 66
018. Multiset Intersection . . . 69
019. Longest Consecutive Sequence . . . 72
020. Count Equal Pairs . . . 75

Module 03 / Sorting and Ordered Scans **78**

021. Merge Two Sorted Arrays . . . 79
022. Deduplicate a Sorted Array . . . 82
023. Three-Way Sort 0/1/2 . . . 85
024. Rank Transform . . . 88
025. Merge Overlapping Intervals . . . 91
026. Insert One Interval . . . 94
027. Intersect Two Interval Lists . . . 97
028. Closest Pair to Target . . . 101

029. Minimum-Difference Pairs . . . 105
030. Sort Records by Multiple Keys . . . 108

Module 04 / Two Pointers and In-Place Read/Write State 111

031. Reverse an Array In Place . . . 112
032. Normalized Palindrome . . . 115
033. Move Zeros Stably . . . 119
034. Remove a Target In Place . . . 122
035. Two Sum in Sorted Array . . . 125
036. Sorted Squares . . . 128
037. Three Sum to Zero . . . 131
038. Container With Most Water . . . 135
039. Trapping Rain Water . . . 138
040. Partition Evens Before Odds . . . 141

Module 05 / Sliding Windows 144

041. Maximum Fixed-Window Sum . . . 145
042. Count Fixed Windows Above Threshold . . . 148
043. Moving Average Stream . . . 151
044. Longest Substring Without Repeats . . . 154
045. Longest Substring with At Most K Distinct . . . 158
046. Longest Ones After K Flips . . . 161
047. Permutation Inclusion . . . 165
048. Shortest Positive Window Reaching Target . . . 169
049. Minimum Covering Substring . . . 173
050. Count Subarrays with Exactly K Distinct . . . 177

Module 06 / Prefix Sums, Prefix State, and Difference Arrays 181

051. Build a Prefix-Sum Array . . . 182
052. Answer Static Range-Sum Queries . . . 185
053. Pivot Index . . . 188
054. Count Subarrays Summing to K . . . 191
055. Longest Subarray Summing to K . . . 195
056. Equal Zeros and Ones . . . 199
057. Product Except Self . . . 203
058. Range Addition with Difference Array . . . 207
059. Flight Bookings . . . 211

060. Two-Dimensional Prefix Queries 215

Module 07 / Stacks and Expression State **219**

061. Balanced Brackets 220
062. Evaluate Postfix Notation 224
063. Remove Adjacent Equal Pairs 228
064. Simplify an Absolute Unix Path 232
065. Validate Stack Push/Pop Sequences 236
066. Design Min Stack 240
067. Decode Nested Repetition 243
068. Asteroid Collision 247
069. Basic Calculator with Parentheses 250
070. Score of Parentheses 254

Module 08 / Monotonic Stacks and Monotonic Deques **258**

071. Next Greater Element 259
072. Next Smaller Element 263
073. Days Until Warmer Temperature 267
074. Online Stock Span 271
075. Largest Rectangle in Histogram 275
076. Maximal Rectangle in Binary Matrix 279
077. Sum of Subarray Minimums 282
078. Sliding Window Maximum 286
079. Shortest Subarray Sum At Least K with Negatives 290
080. Remove K Digits 294

Module 09 / Queues, Heaps, and Streaming Frontiers **298**

081. Queue Using Two Stacks 299
082. Stack Using One Queue 303
083. Recent Counter 306
084. Largest K Values from a Stream 309
085. Smallest K Values from a Stream 313
086. Kth Largest Stream Class 317
087. Top K Frequent Values 321
088. Merge K Sorted Arrays 325
089. K Closest Points to Origin 329
090. Running Median 333

Module 10 / Intervals and Sweep-Line Events **337**

091. Maximum Concurrent Intervals . . . 338
092. Minimum Meeting Rooms . . . 342
093. Car Pooling Capacity . . . 346
094. Total Covered Length . . . 350
095. Employee Free Time . . . 353
096. Calendar Booking I . . . 357
097. Calendar Maximum Overlap . . . 360
098. Minimum Points to Cover Intervals . . . 364
099. Skyline Key Points . . . 368
100. Active Interval Count Queries . . . 372

Module 11 / Linked Lists and Pointer Mutation **376**

101. Build and Serialize a Singly Linked List . . . 377
102. Reverse a Linked List . . . 381
103. Merge Two Sorted Linked Lists . . . 384
104. Find the Middle Node . . . 388
105. Detect a Cycle . . . 391
106. Find Cycle Entry . . . 394
107. Remove Nth Node from End . . . 397
108. Reverse a Sublist . . . 401
109. Reorder List . . . 406
110. Copy List with Random Pointer . . . 411

Module 12 / Binary Search and Monotone Answer Spaces **415**

111. Exact Binary Search . . . 416
112. Lower Bound . . . 419
113. Upper Bound . . . 422
114. First and Last Target Position . . . 425
115. Integer Square Root . . . 428
116. Search Rotated Sorted Array . . . 431
117. Find Minimum in Rotated Sorted Array . . . 436
118. Search a Row-Major Sorted Matrix . . . 439
119. Minimum Eating Speed . . . 443
120. Minimum Shipping Capacity . . . 446

Module 13 / Matrix Traversal and Simulation **450**

121. Row and Column Totals . . . 451
122. Transpose a Matrix . . . 455
123. Rotate a Square Matrix Clockwise . . . 458
124. Spiral Traversal . . . 461
125. Set Matrix Zeroes . . . 465
126. Diagonal Zigzag Traversal . . . 469
127. Game of Life Step . . . 473
128. Matrix Reshape . . . 477
129. Sparse Matrix Multiplication . . . 480
130. Validate Sudoku Board . . . 484

Module 14 / Grid DFS, BFS, and Multi-Source Search **489**

131. Flood Fill with DFS . . . 490
132. Flood Fill with BFS . . . 494
133. Count Islands . . . 498
134. Maximum Island Area . . . 502
135. Surrounded Regions . . . 506
136. Rotting Oranges . . . 510
137. Shortest Binary-Matrix Path . . . 514
138. Nearest-Zero Distances . . . 518
139. Walls and Gates . . . 522
140. Pacific Atlantic Water Flow . . . 526

Module 15 / Recursion and Divide-and-Conquer **530**

141. Factorial with Call Trace . . . 531
142. Sum a Nested List . . . 534
143. Flatten a Nested List . . . 538
144. Reverse a String Recursively . . . 541
145. Fast Exponentiation . . . 544
146. Merge Sort . . . 548
147. Quicksort Partition . . . 551
148. Quickselect Kth Statistic . . . 555
149. Tower of Hanoi . . . 559
150. Merge K Sorted Arrays by Divide-and-Conquer . . . 563

Module 16 / Backtracking and Combinatorial Search **567**

151. Generate Binary Strings . . . 568
152. Generate All Subsets . . . 572
153. Choose K Combinations . . . 575
154. Generate Permutations . . . 579
155. Generate Unique Permutations . . . 583
156. Combination Sum . . . 587
157. Generate Balanced Parentheses . . . 591
158. Word Search in Grid . . . 595
159. Palindrome Partitioning . . . 599
160. N-Queens . . . 603

Module 17 / Binary Trees: Recursive DFS and Structural Contracts 607

161. Build and Serialize a Binary Tree . . . 608
162. Maximum Depth . . . 612
163. Recursive Preorder Traversal . . . 615
164. Recursive Inorder Traversal . . . 618
165. Recursive Postorder Traversal . . . 622
166. Same Tree . . . 625
167. Invert Binary Tree . . . 628
168. Diameter of Binary Tree . . . 631
169. Balanced Binary Tree . . . 634
170. Root-to-Leaf Target Sum . . . 638

Module 18 / Tree BFS, Binary Search Trees, and Tries 641

171. Binary Tree Level Order . . . 642
172. Minimum Tree Depth with BFS . . . 646
173. Right Side View . . . 650
174. Zigzag Level Order . . . 654
175. Validate a BST . . . 657
176. Kth Smallest in BST . . . 661
177. Lowest Common Ancestor in BST . . . 665
178. Sorted Array to Balanced BST . . . 668
179. Implement Trie . . . 671
180. Wildcard Word Dictionary . . . 675

Module 19 / Graph Traversal and Connected Components 679

181. Build an Adjacency List . . . 680

182. DFS Reachability . . . 683
183. BFS Shortest Hops . . . 687
184. Count Graph Components . . . 691
185. Clone a Graph . . . 694
186. Validate an Undirected Tree . . . 697
187. Bipartite Graph Check . . . 701
188. All Paths from Source to Target in DAG . . . 705
189. Number of Provinces . . . 709
190. Word Ladder Length . . . 713

Module 20 / Directed Graphs, Cycles, and Topological Order 717

191. Detect a Directed Cycle with Three States . . . 718
192. Course Schedule Feasibility . . . 722
193. Return One Course Order . . . 726
194. Minimum Semesters . . . 730
195. Eventual Safe Nodes . . . 734
196. Longest Path in a DAG . . . 738
197. Prerequisite Reachability Queries . . . 742
198. Alien Dictionary Order . . . 746
199. Reconstruct Itinerary . . . 750
200. Dependency Impact Counts . . . 754

Module 21 / Union-Find, Weighted Graphs, and Minimum Spanning Trees 758

201. Implement Disjoint-Set Union . . . 759
202. Count Components with Union-Find . . . 763
203. Redundant Connection . . . 766
204. Accounts Merge . . . 769
205. Kruskal Minimum Spanning Tree . . . 773
206. Prim Minimum Spanning Tree . . . 777
207. Dijkstra Shortest Paths . . . 781
208. Network Delay Time . . . 785
209. Cheapest Flight Within K Stops . . . 789
210. Zero-One BFS . . . 793

Module 22 / Greedy Algorithms and Kadane-Style State 797

211. Best Single Stock Transaction . . . 798
212. Maximum Subarray Sum . . . 801

213. Maximum Product Subarray . . . 805
214. Maximum Circular Subarray . . . 809
215. Jump Game Reachability . . . 813
216. Minimum Jumps to End . . . 816
217. Gas Station Circuit . . . 819
218. Maximum Number of Non-overlapping Intervals . . . 823
219. Partition Labels . . . 827
220. Task Scheduler with Cooldown . . . 831

Module 23 / Dynamic Programming I: One-Dimensional State 835

221. Fibonacci with Rolling State . . . 836
222. Climbing Stairs . . . 839
223. Minimum Cost Climbing Stairs . . . 842
224. House Robber . . . 845
225. House Robber II . . . 848
226. Decode Ways . . . 852
227. Word Break . . . 855
228. Minimum Coin Change . . . 859
229. Coin-Change Combination Count . . . 863
230. Longest Increasing Subsequence . . . 867

Module 24 / Dynamic Programming II: Grids, Knapsack, and Strings 872

231. Unique Grid Paths . . . 873
232. Minimum Path Sum . . . 877
233. Unique Paths with Obstacles . . . 881
234. Zero-One Knapsack . . . 885
235. Partition Equal Subset Sum . . . 889
236. Target Sum Ways . . . 893
237. Longest Common Subsequence . . . 897
238. Edit Distance . . . 901
239. Interleaving String . . . 905
240. Longest Palindromic Subsequence . . . 909

Module 25 / Bit Manipulation, Mathematics, and Designed Data Structures 913

241. Single Number with XOR . . . 914
242. Missing Number with XOR . . . 917
243. Count Set Bits . . . 920

244. Counting Bits from 0 to N 923
245. Reverse 32 Bits 926
246. Greatest Common Divisor and LCM 930
247. Sieve of Eratosthenes 933
248. Design Randomized Set 936
249. Design LRU Cache 940
250. Time-Based Key-Value Store 945

A. Diagnose the implementation gap **949**

B. A repeatable interview rehearsal **950**

C. Retrieval map **951**

D. Glossary of working terms **952**

E. Running the examples and keeping editions aligned **953**

F. Shared Python structures **954**

A book about the implementation gap

Recognizing a pattern and implementing it are different skills. You can know that a problem needs a sliding window and still move the left boundary too early, forget a frequency decrement, or commit an invalid answer. This book gives those small decisions names and makes them visible.

The 250 exercises are deliberately sequenced. The early scans teach ingest, update and commit. Later chapters reuse those moves with maps, windows, stacks, pointers, recursion, graph frontiers and dynamic-programming states. A short exercise can be foundational even when its final implementation has only six lines.

Use this book with a blank editor. Read the problem and examples, write a state contract, then implement before opening the worked solution. The website keeps solutions behind a disclosure control for that reason. Knowing what a printed solution looks like is not the same as being able to construct it.

How to work through one kata

First, restate the exact input and output. Decide what empty input, duplicates, ties and invalid candidates mean. Second, name the state and write one sentence saying what each variable represents at a particular point in execution. Third, explain why the next update preserves that meaning.

Write a protocol in ordinary language before Python. Then translate one protocol step at a time. Run the main example and a deliberately awkward boundary example. Finally, reconstruct the implementation without looking and attempt the transfer question, which changes one part of the original contract.

A useful review interval is the next day and again several days later. Advance when you can explain the decisions and rebuild the code, not merely when one familiar sample passes. After a module, choose a related interview problem without a visible pattern label and identify the mechanism yourself.

The pre-code worksheet

For an iterative solution, write: Driver; State; Invariant; Incoming event; Update; Possible violation or qualification; Repair or transition; Safe commit point; Stop condition.

For recursion, replace loop mechanics with: Call contract; Base case; State owned by this call; Child calls or choices; Mutation before the call; Returned value or committed result; Restoration afterward. State ownership matters: a local number is different from a shared path list that must be restored.

The short reasoning card at the end of each lesson is a retrieval aid. It summarizes the state and decision after you have studied their derivation. It is not a substitute for the

proof or the protocol.

Conventions that prevent accidental bugs

Unless a problem explicitly says otherwise, indices are zero-based, strings are case-sensitive, and input preconditions are promises rather than a request for exhaustive validation. An in-place problem explicitly identifies the meaningful output region and whether it returns the original object or only a logical length.

Binary-search candidate intervals use [left,right): left is included and right is an exclusive boundary. Two-pointer problems sometimes name the final candidate index right instead; those lessons state that meaning explicitly. Never copy an update rule without also copying its boundary contract.

Interval endpoints are specified locally. Closed intervals can intersect at one point; half-open intervals do not overlap merely because one ends where another starts. Some compressed source exercises left this ambiguous. The editorial register records the chosen conventions without changing the curriculum order.

Complexities use the usual unit-cost interview model for bounded-size arithmetic and expected hash-table operations. Python arbitrary-precision integer costs can grow with bit length; bit and counting exercises call out relevant limits. Output space and recursion depth are included where they matter. Deep recursive inputs can exceed Python's recursion limit even when the algorithm is mathematically correct.

Reading the recorded traces

The examples have expected results written alongside them and are executed against the reference implementation. The worked trace captures local state after statements in that implementation. A line number refers to the downloadable solution file, whose line numbering starts at one. The code display does not alter execution.

N1, N2 and similar labels identify distinct node objects within one example, even when their values match. Queue, set and tuple displays retain their data-structure notation.

Long traces show an initial segment and a final segment with an explicit omitted-step count. Long values are abbreviated for legibility; full outputs remain in the examples and verification record. A trace illustrates an execution, while the invariant and correctness argument explain all inputs covered by the contract.

The illustrations are original text diagrams: array regions, candidate boundaries, call relationships, frontiers or state transitions. They remain selectable and readable in both print and the website. The exercises' solution files can be run locally using the Python standard library.

Edition and verification

First edition, September 2026. Curriculum and learning emphasis: Arunank Divya Sharan. Editorial development, implementation and validation were prepared with AI assistance. The structure follows the supplied DSA 250 Reboot sequence; this is an independently titled work with original explanations and illustrations.

The package includes a verification report distinguishing executed examples from independent randomized or exhaustive checks. Automated checks provide evidence, not a claim of external peer review or a guarantee that no error remains. Report corrections with the kata number, input, observed result and expected reasoning so revisions stay precise.

The editable Markdown chapters, runnable Python files, static website and PDF are generated from one lesson record. Corrections should be made in that source and rebuilt across formats to keep the editions synchronized.

Module 01
Execution Grammar and State Mutation

Make loop order, branch selection, counters, streaks, boundaries, and commit points automatic.

Begin with execution, not pattern names. These scans teach you to distinguish the incoming value, the state update and the moment an answer becomes valid. Trace every assignment. A running total, a streak and a run encoder use the same loop driver but have different commit and reset rules.

The ten-step progression

001. Running Total

002. Running Minimum and Maximum

003. Maximum Value and First Index

004. Count Positives, Negatives, and Zeros

005. First Cumulative Threshold Crossing

006. Longest Positive Streak

007. Longest Alternating-Parity Streak

008. Run-Length Encoding

009. Split on Sentinels

010. Peak Active Balance

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

001. Running Total

Foundation | Module 01

Problem

Given a mutable list of integers nums, replace nums[i] by the sum of the original values at indices 0 through i, inclusive. Return the same list object. Negative values and an empty list are allowed. Earlier writes must not change which original value is consumed next.

Examples

Example 1. The negative second value decreases the total.

```
running_total([3, -2, 5, 1])
=> [3, 1, 6, 7]
```

Example 2. An empty input needs no writes.

```
running_total([])
=> []
```

Example 3. Running totals need not increase.

```
running_total([-2, -3])
=> [-2, -5]
```

Why this mechanism works

Recomputing a sum for every prefix repeats almost all the work. Adjacent prefixes differ by exactly one incoming value. Keep that difference in one accumulator. Read the current cell before overwriting it; the unread suffix still contains original values. This is the smallest example of an ingest-update-commit loop.

State and boundaries

i is the current input index. total is the sum through i after the addition. nums[:i] contains already committed prefix sums before that iteration.

Protocol

1. Set total to zero.
2. Visit original values from left to right.
3. Add the current value to total.
4. Write total into the current cell.
5. Return nums after the scan.

Visualization

```
original:  [3, -2, 5, 1]
consumed:   3  + -2  + 5  + 1
committed: [3,  1, 6, 7]
```

Python implementation

```
def running_total(nums):
    total = 0
    for i, value in enumerate(nums):
        total += value
        nums[i] = total
    return nums
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
running_total([3, -2, 5, 1])

L2 running_total: total = 0
nums=[3, -2, 5, 1]; total=0

L3 running_total: for i, value in enumerate(nums):
nums=[3, -2, 5, 1]; total=0; i=0; value=3

L4 running_total: total += value
nums=[3, -2, 5, 1]; total=3; i=0; value=3

L3 running_total: for i, value in enumerate(nums):
nums=[3, -2, 5, 1]; total=3; i=1; value=-2

L4 running_total: total += value
nums=[3, -2, 5, 1]; total=1; i=1; value=-2

L5 running_total: nums[i] = total
nums=[3, 1, 5, 1]; total=1; i=1; value=-2

L3 running_total: for i, value in enumerate(nums):
nums=[3, 1, 5, 1]; total=1; i=2; value=5

L4 running_total: total += value
nums=[3, 1, 5, 1]; total=6; i=2; value=5

L5 running_total: nums[i] = total
nums=[3, 1, 6, 1]; total=6; i=2; value=5

L3 running_total: for i, value in enumerate(nums):
nums=[3, 1, 6, 1]; total=6; i=3; value=1

L4 running_total: total += value
nums=[3, 1, 6, 1]; total=7; i=3; value=1

L5 running_total: nums[i] = total
nums=[3, 1, 6, 7]; total=7; i=3; value=1
```

Result: [3, 1, 6, 7]. The negative second value decreases the total.

```
running_total([])

L2 running_total: total = 0
nums=[]; total=0
```

Result: []. An empty input needs no writes.

Why the solution is correct

Initially the empty prefix sums to zero. Adding the next original value extends the represented prefix by exactly one element; writing that sum finalizes its answer without touching unread cells. At termination every position has been finalized.

Time and space

O(n) time and O(1) auxiliary space; the supplied list stores the output.

Common mistakes

Writing before adding loses the incoming value. Summing nums[:i+1] after earlier mutations sums prefix sums rather than the original data.

Reasoning card

State: i is the current input index. total is the sum through i after the addition. nums[:i] contains already committed prefix sums before that iteration.

Invariant: total equals the sum of the processed prefix.

Decision to rehearse: Ingest the current value, update state, then commit it.

Finish: Return nums after the scan

Transfer exercise

Return sums of the values strictly before each position. Which comes first now: committing or adding?

002. Running Minimum and Maximum

Foundation | Module 01

Problem

For each position in a list of integers, report the smallest and largest values in the prefix ending there. Return (minimums, maximums), two newly allocated lists in input order. Do not mutate nums. For empty input return two empty lists.

Examples

Example 1. Each column describes the same prefix.

```
running_extrema([4, 2, 5, 1])
=> ([4, 2, 2, 1], [4, 4, 5, 5])
```

Example 2. Both output lists are empty.

```
running_extrema([])
=> ([], [])
```

Example 3. Zero is not an artificial maximum.

```
running_extrema([-4, -2])
=> ([-4, -4], [-4, -2])
```

Why this mechanism works

A new value can change the minimum, the maximum, both on the first step, or neither. These are independent reductions over the same prefix, so one scan can maintain both. Initializing from the first actual value avoids assuming that zero belongs to the data.

State and boundaries

low and high summarize the processed prefix. minimums and maximums contain one committed result per consumed value.

Protocol

1. Return two empty lists when there is no first value.
2. Initialize both extrema from nums[0].
3. Extend both extrema with each incoming value.
4. Append both answers after updating.

Visualization

```
value:  4  2  5  1
low:    4  2  2  1
high:   4  4  5  5
```

Python implementation

```
def running_extrema(nums):
    if not nums:
        return [], []
    low = high = nums[0]
    minimums, maximums = [], []
    for value in nums:
        low = min(low, value)
        high = max(high, value)
        minimums.append(low)
        maximums.append(high)
    return minimums, maximums
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
running_extrema([4, 2, 5, 1])

L2 running_extrema: if not nums:
nums=[4, 2, 5, 1]

L4 running_extrema: low = high = nums[0]
nums=[4, 2, 5, 1]; low=4; high=4

L5 running_extrema: minimums, maximums = [], []
nums=[4, 2, 5, 1]; low=4; high=4; minimums=[]; maximums=[]

L6 running_extrema: for value in nums:
nums=[4, 2, 5, 1]; low=4; high=4; minimums=[]; maximums=[]; value=4

L9 running_extrema: minimums.append(low)
nums=[4, 2, 5, 1]; low=4; high=4; minimums=[4]; maximums=[]; value=4

L10 running_extrema: maximums.append(high)
nums=[4, 2, 5, 1]; low=4; high=4; minimums=[4]; maximums=[4]; value=4

L6 running_extrema: for value in nums:
nums=[4, 2, 5, 1]; low=4; high=4; minimums=[4]; maximums=[4]; value=2

L7 running_extrema: low = min(low, value)
nums=[4, 2, 5, 1]; low=2; high=4; minimums=[4]; maximums=[4]; value=2

... 6 intermediate state changes omitted ...

L6 running_extrema: for value in nums:
nums=[4, 2, 5, 1]; low=2; high=5; minimums=[4, 2, 2]; maximums=[4, 4, 5]; value=1

L7 running_extrema: low = min(low, value)
nums=[4, 2, 5, 1]; low=1; high=5; minimums=[4, 2, 2]; maximums=[4, 4, 5]; value=1

L9 running_extrema: minimums.append(low)
nums=[4, 2, 5, 1]; low=1; high=5; minimums=[4, 2, 2, 1]; maximums=[4, 4, 5]; value=1
```

```
L10 running_extrema: maximums.append(high)
nums=[4, 2, 5, 1]; low=1; high=5; minimums=[4, 2, 2, 1]; maximums=[4, 4, 5, 5]; value=1
```

Result: ([4, 2, 2, 1], [4, 4, 5, 5]). Each column describes the same prefix.

```
running_extrema([])
```

```
L2 running_extrema: if not nums:
nums=[]
```

Result: ([], []). Both output lists are empty.

Why the solution is correct

The first value is both extrema of a singleton prefix. min(low, value) and max(high, value) preserve the definitions when the prefix grows. Appending only after both updates keeps the two output positions aligned.

Time and space

O(n) time; O(1) working state plus O(n) returned storage.

Common mistakes

Initializing low to zero fails on all-positive input. Using elif between independent extrema updates can mishandle initialization or future variants.

Reasoning card

State: low and high summarize the processed prefix. minimums and maximums contain one committed result per consumed value.

Invariant: Both running values describe exactly the processed prefix.

Decision to rehearse: Coordinate two independent accumulators in one pass.

Finish: Append both answers after updating

Transfer exercise

Also return the running range high-low, without rescanning any prefix.

003. Maximum Value and First Index

Foundation | Module 01

Problem

Return (maximum_value, first_index) for a list of integers. Indices are zero-based. If the list is empty, return None. Equal later maxima must not replace the index of the first occurrence. Leave the input unchanged.

Examples

Example 1. The second 9 does not win the tie.

```
maximum_first([4, 9, 2, 9])
=> (9, 1)
```

Example 2. No maximum exists.

```
maximum_first([])
=> None
```

Example 3. The rule also works with negative maxima.

```
maximum_first([-5, -1, -1])
=> (-1, 1)
```

Why this mechanism works

A scan needs only the best value seen and its position. The tie rule is part of correctness, not cosmetic output formatting. A strict improvement replaces the candidate; equality provides no reason to discard an earlier valid index. This distinction will reappear in stable ranking and shortest-path tie policies.

State and boundaries

best_value is the prefix maximum. best_index is its earliest position in that prefix.

Protocol

1. Handle empty input.
2. Seed the candidate from the first element.
3. Compare each later value to best_value.
4. Replace both fields only on a strictly larger value.
5. Return the pair.

Visualization

```
value:      4   9   2   9
best index: 0   1   1   1
                        equal: retain 1
```

Python implementation

```
def maximum_first(nums):
    if not nums:
        return None
    best_value, best_index = nums[0], 0
    for i in range(1, len(nums)):
        if nums[i] > best_value:
            best_value, best_index = nums[i], i
    return best_value, best_index
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
maximum_first([4, 9, 2, 9])

L2 maximum_first: if not nums:
nums=[4, 9, 2, 9]

L4 maximum_first: best_value, best_index = nums[0], 0
nums=[4, 9, 2, 9]; best_value=4; best_index=0

L5 maximum_first: for i in range(1, len(nums)):
nums=[4, 9, 2, 9]; best_value=4; best_index=0; i=1

L7 maximum_first: best_value, best_index = nums[i], i
nums=[4, 9, 2, 9]; best_value=9; best_index=1; i=1

L5 maximum_first: for i in range(1, len(nums)):
nums=[4, 9, 2, 9]; best_value=9; best_index=1; i=2

L5 maximum_first: for i in range(1, len(nums)):
nums=[4, 9, 2, 9]; best_value=9; best_index=1; i=3
```

Result: (9, 1). The second 9 does not win the tie.

```
maximum_first([])

L2 maximum_first: if not nums:
nums=[]
```

Result: None. No maximum exists.

Why the solution is correct

The seeded singleton satisfies both maximum and earliest-index requirements. A larger value invalidates the old maximum and becomes its first occurrence so far. Equal or smaller values leave the existing candidate valid. Thus the final pair meets both requirements.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Using >= returns the last maximum instead. Returning nums.index(max(nums)) works but hides the one-pass state mechanism being practiced.

Reasoning card

State: best_value is the prefix maximum. best_index is its earliest position in that prefix.

Invariant: best_index remains the first index of best_value.

Decision to rehearse: Use strict versus non-strict comparison deliberately.

Finish: Return the pair

Transfer exercise

Change the contract to the last occurrence and identify the single comparison that changes.

004. Count Positives, Negatives, and Zeros

Foundation | Module 01

Problem

Count how many elements of an integer list are positive, negative and zero. Return a tuple in exactly that order. Each occurrence counts, including repeated values. The empty list returns (0, 0, 0), and the input is not changed.

Examples

Example 1. The tuple order is positive, negative, zero.

```
sign_counts([-2, 0, 4, -1, 0])
=> (1, 2, 2)
```

Example 2. No category has an occurrence.

```
sign_counts([])
=> (0, 0, 0)
```

Example 3. Repeated zeros each contribute.

```
sign_counts([0, 0])
=> (0, 0, 2)
```

Why this mechanism works

The three categories partition the input: every integer belongs to one category and cannot belong to two. Express that exclusivity in the branch structure. The sum of the counters provides a cheap consistency check after every processed prefix.

State and boundaries

positive, negative and zero count their respective categories among consumed values.

Protocol

1. Initialize all counters to zero.
2. For each value choose exactly one sign branch.
3. Increment only that branch's counter.
4. Return the counters in the specified order.

Visualization

```
incoming value
  > 0 -> positive
  < 0 -> negative
  else -> zero
```

Python implementation

```
def sign_counts(nums):
    positive = negative = zero = 0
    for value in nums:
        if value > 0:
            positive += 1
        elif value < 0:
            negative += 1
        else:
            zero += 1
    return positive, negative, zero
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
sign_counts([-2, 0, 4, -1, 0])

L2 sign_counts: positive = negative = zero = 0
nums=[-2, 0, 4, -1, 0]; positive=0; negative=0; zero=0

L3 sign_counts: for value in nums:
nums=[-2, 0, 4, -1, 0]; positive=0; negative=0; zero=0; value=-2

L7 sign_counts: negative += 1
nums=[-2, 0, 4, -1, 0]; positive=0; negative=1; zero=0; value=-2

L3 sign_counts: for value in nums:
nums=[-2, 0, 4, -1, 0]; positive=0; negative=1; zero=0; value=0

L9 sign_counts: zero += 1
nums=[-2, 0, 4, -1, 0]; positive=0; negative=1; zero=1; value=0

L3 sign_counts: for value in nums:
nums=[-2, 0, 4, -1, 0]; positive=0; negative=1; zero=1; value=4

L5 sign_counts: positive += 1
nums=[-2, 0, 4, -1, 0]; positive=1; negative=1; zero=1; value=4

L3 sign_counts: for value in nums:
nums=[-2, 0, 4, -1, 0]; positive=1; negative=1; zero=1; value=-1

L7 sign_counts: negative += 1
nums=[-2, 0, 4, -1, 0]; positive=1; negative=2; zero=1; value=-1

L3 sign_counts: for value in nums:
nums=[-2, 0, 4, -1, 0]; positive=1; negative=2; zero=1; value=0

L9 sign_counts: zero += 1
nums=[-2, 0, 4, -1, 0]; positive=1; negative=2; zero=2; value=0
```

Result: (1, 2, 2). The tuple order is positive, negative, zero.

```
sign_counts([])

L2 sign_counts: positive = negative = zero = 0
nums=[]; positive=0; negative=0; zero=0
```

Result: (0, 0, 0). No category has an occurrence.

Why the solution is correct

Initially all three counts match the empty prefix. Exactly one increment accounts for each new integer, so no occurrence is missed or counted twice. At the end the counters cover the entire list and sum to its length.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Three unrelated if statements are easy to make overlapping. A truthiness check classifies nonzero values together and loses their signs.

Reasoning card

State: positive, negative and zero count their respective categories among consumed values.

Invariant: Each processed value contributes to exactly one counter.

Decision to rehearse: Use mutually exclusive if/elif/else branches.

Finish: Return the counters in the specified order

Transfer exercise

Classify values into disjoint numeric bands, and prove that the boundary values belong to exactly one band.

005. First Cumulative Threshold Crossing

Bridge | Module 01

Problem

Given integers nums and target, return the first zero-based index i for which sum(nums[:i+1]) >= target. Return -1 if no nonempty prefix qualifies. Values may be negative, so later prefixes can fall below the threshold again. Even when target <= 0, test actual nonempty prefixes rather than returning an imaginary index.

Examples

Example 1. Nine reaches or exceeds eight.

```
first_threshold([2, 4, 3], 8)
=> 2
```

Example 2. The contract excludes the empty prefix.

```
first_threshold([], 0)
=> -1
```

Example 3. A later decline cannot erase the first crossing.

```
first_threshold([5, -10, 8], 4)
=> 0
```

Why this mechanism works

Only the earliest qualifying prefix matters. Update the cumulative sum before testing, because the question includes the current element. A left-to-right scan establishes earliestness automatically. Negative numbers prevent treating this predicate as globally monotone, but do not invalidate the scan.

State and boundaries

total includes the current value at the check. All earlier nonempty prefixes have failed if execution has not returned.

Protocol

1. Start total at zero.

2. Add nums[i].

3. Test total >= target.

4. Return i immediately on success.

5. Return -1 after all candidates fail.

Visualization

```
prefix sums: 2 -> 6 -> 9
target:                  8
first qualifying index:  2
```

Python implementation

```
def first_threshold(nums, target):
    total = 0
    for i, value in enumerate(nums):
        total += value
        if total >= target:
            return i
    return -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
first_threshold([2, 4, 3], 8)

L2 first_threshold: total = 0
nums=[2, 4, 3]; target=8; total=0

L3 first_threshold: for i, value in enumerate(nums):
nums=[2, 4, 3]; target=8; total=0; i=0; value=2

L4 first_threshold: total += value
nums=[2, 4, 3]; target=8; total=2; i=0; value=2

L3 first_threshold: for i, value in enumerate(nums):
nums=[2, 4, 3]; target=8; total=2; i=1; value=4

L4 first_threshold: total += value
nums=[2, 4, 3]; target=8; total=6; i=1; value=4

L3 first_threshold: for i, value in enumerate(nums):
nums=[2, 4, 3]; target=8; total=6; i=2; value=3

L4 first_threshold: total += value
nums=[2, 4, 3]; target=8; total=9; i=2; value=3
```

Result: 2. Nine reaches or exceeds eight.

```
first_threshold([], 0)

L2 first_threshold: total = 0
nums=[]; target=0; total=0
```

Result: -1. The contract excludes the empty prefix.

Why the solution is correct

The accumulator is correct by prefix extension. Every smaller index has already been tested and rejected when an index is returned, so the answer is earliest. If the scan ends, every admissible prefix has failed.

Time and space

O(n) worst-case time and O(1) auxiliary space; early success may use fewer iterations.

Common mistakes

Testing before adding shifts the reported index. Checking equality instead of >= misses a prefix that jumps over the target.

Reasoning card

State: total includes the current value at the check. All earlier nonempty prefixes have failed if execution has not returned.

Invariant: Before each check, total includes the current value.

Decision to rehearse: Update before checking and return early.

Finish: Return -1 after all candidates fail

Transfer exercise

Return all crossing indices instead of the first. Does a crossing mean qualifying, or moving from below to above? Specify that distinction first.

006. Longest Positive Streak

Bridge | Module 01

Problem

Return the length of the longest contiguous run of strictly positive integers. Zero and negative values break a run. Empty input and an input with no positive value return zero. A run consists of adjacent positions, not a subsequence chosen by skipping failures.

Examples

Example 1. The final run is shorter than the earlier run.

```
positive_streak([-1, 2, 3, -2, 4])
=> 2
```

Example 2. There is no nonempty run.

```
positive_streak([])
=> 0
```

Example 3. Zero breaks the first run.

```
positive_streak([1, 0, 2, 3])
=> 2
```

Why this mechanism works

Separate the run that is still alive from the best completed or ongoing run. The current run has only two possible transitions: extend on a positive event, or reset on a break. The global best is updated after that transition so a run ending at the final element is included.

State and boundaries

current is the positive suffix length of the processed prefix. best is the largest such suffix observed so far.

Protocol

1. Initialize current and best to zero.
2. Extend current for a positive value.
3. Otherwise reset current to zero.
4. Update best from current after the branch.
5. Return best.

Visualization

```
values:  -1  2  3 -2  4
current:  0  1  2  0  1
best:     0  1  2  2  2
```

Python implementation

```
def positive_streak(nums):
    current = best = 0
    for value in nums:
        current = current + 1 if value > 0 else 0
        best = max(best, current)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
positive_streak([-1, 2, 3, -2, 4])

L2 positive_streak: current = best = 0
nums=[-1, 2, 3, -2, 4]; current=0; best=0

L3 positive_streak: for value in nums:
nums=[-1, 2, 3, -2, 4]; current=0; best=0; value=-1

L3 positive_streak: for value in nums:
nums=[-1, 2, 3, -2, 4]; current=0; best=0; value=2

L4 positive_streak: current = current + 1 if value > 0 else 0
nums=[-1, 2, 3, -2, 4]; current=1; best=0; value=2

L5 positive_streak: best = max(best, current)
nums=[-1, 2, 3, -2, 4]; current=1; best=1; value=2

L3 positive_streak: for value in nums:
nums=[-1, 2, 3, -2, 4]; current=1; best=1; value=3

L4 positive_streak: current = current + 1 if value > 0 else 0
nums=[-1, 2, 3, -2, 4]; current=2; best=1; value=3

L5 positive_streak: best = max(best, current)
nums=[-1, 2, 3, -2, 4]; current=2; best=2; value=3

L3 positive_streak: for value in nums:
nums=[-1, 2, 3, -2, 4]; current=2; best=2; value=-2

L4 positive_streak: current = current + 1 if value > 0 else 0
nums=[-1, 2, 3, -2, 4]; current=0; best=2; value=-2

L3 positive_streak: for value in nums:
nums=[-1, 2, 3, -2, 4]; current=0; best=2; value=4

L4 positive_streak: current = current + 1 if value > 0 else 0
nums=[-1, 2, 3, -2, 4]; current=1; best=2; value=4
```

Result: 2. The final run is shorter than the earlier run.

```
positive_streak([])

L2 positive_streak: current = best = 0
nums=[]; current=0; best=0
```

Result: 0. There is no nonempty run.

Why the solution is correct

A positive value extends exactly the preceding positive suffix; a nonpositive value leaves no positive suffix. Every maximal positive run appears as current at its final index. Taking the maximum of these suffix lengths finds the longest run.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Resetting best destroys earlier successful runs. Counting all positive values ignores contiguity.

Reasoning card

State: current is the positive suffix length of the processed prefix. best is the largest such suffix observed so far.

Invariant: current is the positive streak ending at the current value.

Decision to rehearse: Extend or reset, then update the global best.

Finish: Return best

Transfer exercise

Return the first longest run's half-open interval as well as its length.

007. Longest Alternating-Parity Streak

Bridge | Module 01

Problem

Return the longest contiguous run in which every adjacent pair has different parity. Odd and even are defined by value % 2, including negative integers. A singleton is a valid run of length one. Empty input returns zero.

Examples

Example 1. The final even-even edge breaks the run.

```
alternating_parity([1, 2, 5, 8, 10])
=> 4
```

Example 2. No singleton exists.

```
alternating_parity([])
=> 0
```

Example 3. Negative values have usable parity too.

```
alternating_parity([-3, -2, -1])
=> 3
```

Why this mechanism works

The condition belongs to an edge between neighboring values. When an edge fails, the current element remains a valid singleton even though it cannot extend the previous run. That is why a failed edge restarts the state at one rather than zero.

State and boundaries

current is the alternating suffix ending at the current index. best is the maximum suffix length seen.

Protocol

1. Handle the empty list.
2. Seed a singleton run.
3. Compare each value's parity with its predecessor.
4. Extend on different parity, otherwise restart at one.
5. Update the best length.

Visualization

```
values:  1  2  5  8 10
parity:  O  E  O  E  E
length:  1  2  3  4  1
```

Python implementation

```
def alternating_parity(nums):
    if not nums:
        return 0
    current = best = 1
    for i in range(1, len(nums)):
        if nums[i] % 2 != nums[i - 1] % 2:
            current += 1
        else:
            current = 1
        best = max(best, current)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
alternating_parity([1, 2, 5, 8, 10])

L2 alternating_parity: if not nums:
nums=[1, 2, 5, 8, 10]

L4 alternating_parity: current = best = 1
nums=[1, 2, 5, 8, 10]; current=1; best=1

L5 alternating_parity: for i in range(1, len(nums)):
nums=[1, 2, 5, 8, 10]; current=1; best=1; i=1

L7 alternating_parity: current += 1
nums=[1, 2, 5, 8, 10]; current=2; best=1; i=1

L10 alternating_parity: best = max(best, current)
nums=[1, 2, 5, 8, 10]; current=2; best=2; i=1

L5 alternating_parity: for i in range(1, len(nums)):
nums=[1, 2, 5, 8, 10]; current=2; best=2; i=2

L7 alternating_parity: current += 1
nums=[1, 2, 5, 8, 10]; current=3; best=2; i=2

L10 alternating_parity: best = max(best, current)
nums=[1, 2, 5, 8, 10]; current=3; best=3; i=2

... 1 intermediate state changes omitted ...

L7 alternating_parity: current += 1
nums=[1, 2, 5, 8, 10]; current=4; best=3; i=3

L10 alternating_parity: best = max(best, current)
nums=[1, 2, 5, 8, 10]; current=4; best=4; i=3

L5 alternating_parity: for i in range(1, len(nums)):
nums=[1, 2, 5, 8, 10]; current=4; best=4; i=4
```

```
L9 alternating_parity: current = 1
nums=[1, 2, 5, 8, 10]; current=1; best=4; i=4
```

Result: 4. The final even-even edge breaks the run.

```
alternating_parity([])
```

```
L2 alternating_parity: if not nums:
nums=[]
```

Result: 0. No singleton exists.

Why the solution is correct

The first singleton satisfies the contract. A differing parity creates a valid extension of the longest alternating suffix; matching parity makes every longer suffix invalid while leaving the singleton valid. Every candidate run is considered at its endpoint.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Checking signs instead of parity changes the problem. A reset to zero undercounts a new run beginning at the failed edge's right endpoint.

Reasoning card

State: current is the alternating suffix ending at the current index. best is the maximum suffix length seen.

Invariant: current is the valid streak ending at the current element.

Decision to rehearse: A broken streak restarts at length one, not zero.

Finish: Update the best length

Transfer exercise

Replace alternating parity with a fixed adjacent difference. Which state and reset rule survive unchanged?

008. Run-Length Encoding

Interview | Module 01

Problem

Compress a string into a list of (character, count) tuples for its maximal consecutive runs. Preserve run order and character case. Separate runs of the same character remain separate. Empty input returns an empty list; no encoded text format needs to be invented.

Examples

Example 1. Three maximal runs are emitted.

```
run_length_encode("aaabbc")
=> [('a', 3), ('b', 2), ('c', 1)]
```

Example 2. There is no run to seed.

```
run_length_encode("")
=> []
```

Example 3. The two a runs must remain distinct.

```
run_length_encode("aba")
=> [('a', 1), ('b', 1), ('a', 1)]
```

Why this mechanism works

A run's final count becomes known when a different character arrives. Keep one uncommitted run and append it at that boundary. The final run has no following character to trigger the boundary, so termination must explicitly flush it. This is a commit-point problem rather than a frequency-counting problem.

State and boundaries

symbol and count describe the active run. output contains only completed runs, in order.

Protocol

1. Handle empty input.
2. Start the first run.
3. Extend while the incoming character matches.
4. On change append the old run and start the new one.
5. Append the final run after the loop.

Visualization

```
aaa | bb | c
(a,3) (b,2) (c,1)
change commits; end flushes
```

Python implementation

```
def run_length_encode(text):
    if not text:
        return []
    output = []
    symbol, count = text[0], 1
    for i in range(1, len(text)):
        char = text[i]
        if char == symbol:
            count += 1
        else:
            output.append((symbol, count))
            symbol, count = char, 1
    output.append((symbol, count))
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_length_encode("aaabbc")

L2 run_length_encode: if not text:
text='aaabbc'

L4 run_length_encode: output = []
text='aaabbc'; output=[]

L5 run_length_encode: symbol, count = text[0], 1
text='aaabbc'; output=[]; symbol='a'; count=1

L6 run_length_encode: for i in range(1, len(text)):
text='aaabbc'; output=[]; symbol='a'; count=1; i=1

L7 run_length_encode: char = text[i]
text='aaabbc'; output=[]; symbol='a'; count=1; i=1; char='a'

L9 run_length_encode: count += 1
text='aaabbc'; output=[]; symbol='a'; count=2; i=1; char='a'

L6 run_length_encode: for i in range(1, len(text)):
text='aaabbc'; output=[]; symbol='a'; count=2; i=2; char='a'

L9 run_length_encode: count += 1
text='aaabbc'; output=[]; symbol='a'; count=3; i=2; char='a'

... 7 intermediate state changes omitted ...

L7 run_length_encode: char = text[i]
text='aaabbc'; output=[('a', 3)]; symbol='b'; count=2; i=5; char='c'

L11 run_length_encode: output.append((symbol, count))
text='aaabbc'; output=[('a', 3), ('b', 2)]; symbol='b'; count=2; i=5; char='c'
```

```
L12 run_length_encode: symbol, count = char, 1
text='aaabbc'; output=[('a', 3), ('b', 2)]; symbol='c'; count=1; i=5; char='c'

L13 run_length_encode: output.append((symbol, count))
text='aaabbc'; output=[('a', 3), ('b', 2), ('c', 1)]; symbol='c'; count=1; i=5; char='c'
```

Result: [('a', 3), ('b', 2), ('c', 1)]. Three maximal runs are emitted.

```
run_length_encode("")

L2 run_length_encode: if not text:
text=''
```

Result: []. There is no run to seed.

Why the solution is correct

Each matching event stays within the active maximal run. A change proves the old run is complete and starts the next run at count one. The final flush accounts for the only run that lacks a following boundary. Every input character belongs to one output tuple.

Time and space

O(n) time; O(r) output space for r runs and O(1) other state.

Common mistakes

A frequency map merges nonadjacent runs. Forgetting the final append loses the entire result for a one-run string.

Reasoning card

State: symbol and count describe the active run. output contains only completed runs, in order.

Invariant: count describes the active run only.

Decision to rehearse: Commit a group on change and flush the final group.

Finish: Append the final run after the loop

Transfer exercise

Decode the tuples back to the original string and use that identity as a property test.

009. Split on Sentinels

Interview | Module 01

Problem

Split a list of integers and None sentinels into nonempty consecutive groups. None is a separator, not output data. Ignore leading, trailing and repeated separators. Preserve the order and multiplicity of all integers; zero is ordinary data.

Examples

Example 1. Repeated separators do not create empty groups.

```
split_sentinels([1, 2, None, 3, None, None, 4])
=> [[1, 2], [3], [4]]
```

Example 2. No data means no groups.

```
split_sentinels([None, None])
=> []
```

Example 3. Zero is retained as data.

```
split_sentinels([0, None, 0])
=> [[0], [0]]
```

Why this mechanism works

The active group is not final until a separator arrives. Empty groups should never be committed. Allocating a fresh active list after appending also prevents later mutation from changing an already committed group through aliasing.

State and boundaries

current contains the uncommitted integer run. groups contains completed nonempty runs.

Protocol

1. Start an empty current group.
2. Append integers to it.
3. On None, commit only if current is nonempty.
4. Replace current with a fresh list.
5. Flush a nonempty final group.

Visualization

```
1 2 | 3 | | 4
[1,2] [3]  [4]
empty interval: no group
```

Python implementation

```
def split_sentinels(values):
    groups, current = [], []
    for value in values:
        if value is None:
            if current:
                groups.append(current)
                current = []
        else:
            current.append(value)
    if current:
        groups.append(current)
    return groups
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
split_sentinels([1, 2, None, 3, None, None, 4])

L2 split_sentinels: groups, current = [], []
values=[1, 2, None, 3, None, None, 4]; groups=[]; current=[]

L3 split_sentinels: for value in values:
values=[1, 2, None, 3, None, None, 4]; groups=[]; current=[]; value=1

L9 split_sentinels: current.append(value)
values=[1, 2, None, 3, None, None, 4]; groups=[]; current=[1]; value=1

L3 split_sentinels: for value in values:
values=[1, 2, None, 3, None, None, 4]; groups=[]; current=[1]; value=2

L9 split_sentinels: current.append(value)
values=[1, 2, None, 3, None, None, 4]; groups=[]; current=[1, 2]; value=2

L3 split_sentinels: for value in values:
values=[1, 2, None, 3, None, None, 4]; groups=[]; current=[1, 2]; value=None

L6 split_sentinels: groups.append(current)
values=[1, 2, None, 3, None, None, 4]; groups=[[1, 2]]; current=[1, 2]; value=None

L7 split_sentinels: current = []
values=[1, 2, None, 3, None, None, 4]; groups=[[1, 2]]; current=[]; value=None

... 4 intermediate state changes omitted ...

L7 split_sentinels: current = []
values=[1, 2, None, 3, None, None, 4]; groups=[[1, 2], [3]]; current=[]; value=None

L3 split_sentinels: for value in values:
values=[1, 2, None, 3, None, None, 4]; groups=[[1, 2], [3]]; current=[]; value=4

L9 split_sentinels: current.append(value)
```

```
values=[1, 2, None, 3, None, None, 4]; groups=[[1, 2], [3]]; current=[4]; value=4

L11 split_sentinels: groups.append(current)
values=[1, 2, None, 3, None, None, 4]; groups=[[1, 2], [3], [4]]; current=[4]; value=4
```

Result: [[1, 2], [3], [4]]. Repeated separators do not create empty groups.

```
split_sentinels([None, None])

L2 split_sentinels: groups, current = [], []
values=[None, None]; groups=[]; current=[]

L3 split_sentinels: for value in values:
values=[None, None]; groups=[]; current=[]; value=None
```

Result: []. No data means no groups.

Why the solution is correct

Each integer is appended to the only group compatible with its location between sentinels. A separator closes that group, while repeated separators add nothing. The final flush completes the last segment, preserving all data without empty groups.

Time and space

O(n) time and O(n) returned storage; all non-sentinel values are copied once.

Common mistakes

Using if value treats zero as a separator. Clearing the same list after appending it would also empty the stored group.

Reasoning card

State: current contains the uncommitted integer run. groups contains completed nonempty runs.

Invariant: current contains only the uncommitted group.

Decision to rehearse: Commit on a boundary; ignore repeated boundaries; final flush.

Finish: Flush a nonempty final group

Transfer exercise

Generalize the separator to a predicate while preserving the no-empty-groups contract.

010. Peak Active Balance

Interview | Module 01

Problem

Given an ordered list of +1 and -1 events, return the maximum cumulative balance, including the initial balance zero. Events are applied in the given order. Negative intermediate balances are permitted in this arithmetic kata; it does not validate a physical resource log.

Examples

Example 1. The final balance is one but the peak is two.

```
peak_balance([1, 1, -1, 1, -1])
=> 2
```

Example 2. The initial state is included.

```
peak_balance([])
=> 0
```

Example 3. Negative prefixes do not exceed the initial zero.

```
peak_balance([-1, -1])
=> 0
```

Why this mechanism works

An event is a delta rather than a complete state. Apply it to the current balance and then compare the resulting state with the maximum. Including the initial state makes an empty or entirely negative sequence return zero. Later sweep-line problems use this same event-to-state transformation.

State and boundaries

active is the balance after consumed events. peak is the maximum balance among the initial state and all consumed prefixes.

Protocol

1. Initialize active and peak to zero.
2. Apply each delta to active.
3. Update peak after applying the event.
4. Return peak.

Visualization

```
event:    +1 +1 -1 +1 -1
balance:   1  2  1  2  1
peak:      1  2  2  2  2
```

Python implementation

```
def peak_balance(events):
    active = peak = 0
    for delta in events:
        active += delta
        peak = max(peak, active)
    return peak
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
peak_balance([1, 1, -1, 1, -1])

L2 peak_balance: active = peak = 0
events=[1, 1, -1, 1, -1]; active=0; peak=0

L3 peak_balance: for delta in events:
events=[1, 1, -1, 1, -1]; active=0; peak=0; delta=1

L4 peak_balance: active += delta
events=[1, 1, -1, 1, -1]; active=1; peak=0; delta=1

L5 peak_balance: peak = max(peak, active)
events=[1, 1, -1, 1, -1]; active=1; peak=1; delta=1

L4 peak_balance: active += delta
events=[1, 1, -1, 1, -1]; active=2; peak=1; delta=1

L5 peak_balance: peak = max(peak, active)
events=[1, 1, -1, 1, -1]; active=2; peak=2; delta=1

L3 peak_balance: for delta in events:
events=[1, 1, -1, 1, -1]; active=2; peak=2; delta=-1

L4 peak_balance: active += delta
events=[1, 1, -1, 1, -1]; active=1; peak=2; delta=-1

L3 peak_balance: for delta in events:
events=[1, 1, -1, 1, -1]; active=1; peak=2; delta=1

L4 peak_balance: active += delta
events=[1, 1, -1, 1, -1]; active=2; peak=2; delta=1

L3 peak_balance: for delta in events:
events=[1, 1, -1, 1, -1]; active=2; peak=2; delta=-1

L4 peak_balance: active += delta
events=[1, 1, -1, 1, -1]; active=1; peak=2; delta=-1
```

Result: 2. The final balance is one but the peak is two.

```
peak_balance([])

L2 peak_balance: active = peak = 0
events=[]; active=0; peak=0
```

Result: 0. The initial state is included.

Why the solution is correct

The running addition gives the exact balance after each event. Updating the maximum considers every event boundary once, while initialization includes the boundary before any event. Therefore no candidate balance is omitted.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Returning the final balance answers a different question. Checking the peak before updating misses a maximum created by the last event.

Reasoning card

State: active is the balance after consumed events. peak is the maximum balance among the initial state and all consumed prefixes.

Invariant: active is the balance after all processed events.

Decision to rehearse: Apply an event delta, then compare with the best.

Finish: Return peak

Transfer exercise

Return the earliest event index at which the peak occurs, explicitly deciding how to represent an initial-state maximum.

Module 02
Hash Maps, Sets, and Frequency State

Use remembered information about the past to answer the current event in constant expected time.

The first module summarized a prefix with a few numbers. A map or set lets the prefix remember information by value: membership, frequency, identity relationships or a previous occurrence. Before every lookup, decide whether the current element belongs in the stored history yet. That timing separates a valid pair from using one index twice.

The ten-step progression

011. Two-Sum Indices

012. First Repeated Value

013. Valid Anagram

014. First Unique Character

015. Group Anagrams

016. Most Frequent with Earliest Tie

017. Isomorphic Strings

018. Multiset Intersection

019. Longest Consecutive Sequence

020. Count Equal Pairs

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

011. Two-Sum Indices

Foundation | Module 02

Problem

Return two distinct zero-based indices whose integer values sum to target, or [] if no pair exists. Scan the second index from left to right; if several earlier complements exist, use their earliest index. Do not sort or mutate the input.

Examples

Example 1. The second value finds its earlier complement.

```
two_sum([2, 7, 11, 15], 9)
=> [0, 1]
```

Example 2. One position cannot be reused.

```
two_sum([3], 6)
=> []
```

Example 3. Equal values at distinct positions are legal.

```
two_sum([3, 3], 6)
=> [0, 1]
```

Why this mechanism works

For an incoming value x, a successful earlier partner must equal target-x. Remember earlier values in a dictionary instead of scanning them again. Query before inserting x: otherwise one position could incorrectly supply both terms. Preserving the first stored index makes the tie policy deterministic.

State and boundaries

seen maps a previously consumed value to its first index; j is the proposed second index.

Protocol

1. Start with an empty dictionary.
2. Compute target-value.
3. Return its saved index and j if present.
4. Otherwise remember the current value only if new.
5. Return [] on exhaustion.

Visualization

```
past: {2: 0} | incoming: 7
needed: 9 - 7 = 2
answer: [0, 1]
```

Python implementation

```
def two_sum(nums, target):
    seen = {}
    for j, value in enumerate(nums):
        needed = target - value
        if needed in seen:
            return [seen[needed], j]
        seen.setdefault(value, j)
    return []
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
two_sum([2, 7, 11, 15], 9)

L2 two_sum: seen = {}
nums=[2, 7, 11, 15]; target=9; seen={}

L3 two_sum: for j, value in enumerate(nums):
nums=[2, 7, 11, 15]; target=9; seen={}; j=0; value=2

L4 two_sum: needed = target - value
nums=[2, 7, 11, 15]; target=9; seen={}; j=0; value=2; needed=7

L7 two_sum: seen.setdefault(value, j)
nums=[2, 7, 11, 15]; target=9; seen={2:0}; j=0; value=2; needed=7

L3 two_sum: for j, value in enumerate(nums):
nums=[2, 7, 11, 15]; target=9; seen={2:0}; j=1; value=7; needed=7

L4 two_sum: needed = target - value
nums=[2, 7, 11, 15]; target=9; seen={2:0}; j=1; value=7; needed=2
```

Result: [0, 1]. The second value finds its earlier complement.

```
two_sum([3], 6)

L2 two_sum: seen = {}
nums=[3]; target=6; seen={}

L3 two_sum: for j, value in enumerate(nums):
nums=[3]; target=6; seen={}; j=0; value=3

L4 two_sum: needed = target - value
nums=[3]; target=6; seen={}; j=0; value=3; needed=3

L7 two_sum: seen.setdefault(value, j)
nums=[3]; target=6; seen={3:0}; j=0; value=3; needed=3
```

Result: []. One position cannot be reused.

Why the solution is correct

Before the query, seen contains only earlier positions, so a returned pair is distinct and has the required sum. Every possible second index is queried in order, and its complement is found exactly when a partner exists. Thus exhaustion proves no pair exists.

Time and space

O(n) expected time and O(n) auxiliary space under ordinary hash-table assumptions.

Common mistakes

Insert-before-query falsely solves [3] with target 6. Storing values instead of indices cannot satisfy the output contract.

Reasoning card

State: seen maps a previously consumed value to its first index; j is the proposed second index.

Invariant: seen maps earlier values to usable indices.

Decision to rehearse: Query the complement before inserting the current value.

Finish: Return [] on exhaustion

Transfer exercise

Count all index pairs instead of returning one; replace each stored index by an earlier-occurrence count.

012. First Repeated Value

Foundation | Module 02

Problem

Return the first integer encountered for a second time while scanning left to right. Return None if all values are distinct. This asks for the earliest repeated encounter, not the earliest first occurrence of any eventually duplicated value.

Examples

Example 1. Although 3 appears earlier initially, 1 repeats sooner.

```
first_repeat([3, 1, 4, 1, 5, 3])
=> 1
```

Example 2. No encounter exists.

```
first_repeat([])
=> None
```

Example 3. Zero is a valid repeated value, distinct from None.

```
first_repeat([0, 0])
=> 0
```

Why this mechanism works

The decision depends only on whether the incoming value has appeared, not on its count or position. A set is therefore enough. Once a repeated encounter is found, later input cannot improve its encounter order.

State and boundaries

seen is the set of all values strictly before the current position.

Protocol

1. Start an empty set.
2. Query membership before insertion.
3. Return the value on a hit.
4. Add a new value and continue.
5. Return None if the scan ends.

Visualization

```
3, 1, 4 | 1, 5, 3
seen={3,1,4}
           ^ first repeated encounter
```

Python implementation

```
def first_repeat(nums):
    seen = set()
    for value in nums:
        if value in seen:
            return value
        seen.add(value)
    return None
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
first_repeat([3, 1, 4, 1, 5, 3])

L2 first_repeat: seen = set()
nums=[3, 1, 4, 1, 5, 3]; seen=set()

L3 first_repeat: for value in nums:
nums=[3, 1, 4, 1, 5, 3]; seen=set(); value=3

L6 first_repeat: seen.add(value)
nums=[3, 1, 4, 1, 5, 3]; seen={3}; value=3

L3 first_repeat: for value in nums:
nums=[3, 1, 4, 1, 5, 3]; seen={3}; value=1

L6 first_repeat: seen.add(value)
nums=[3, 1, 4, 1, 5, 3]; seen={1, 3}; value=1

L3 first_repeat: for value in nums:
nums=[3, 1, 4, 1, 5, 3]; seen={1, 3}; value=4

L6 first_repeat: seen.add(value)
nums=[3, 1, 4, 1, 5, 3]; seen={1, 3, 4}; value=4

L3 first_repeat: for value in nums:
nums=[3, 1, 4, 1, 5, 3]; seen={1, 3, 4}; value=1
```

Result: 1. Although 3 appears earlier initially, 1 repeats sooner.

```
first_repeat([])

L2 first_repeat: seen = set()
nums=[]; seen=set()
```

Result: None. No encounter exists.

Why the solution is correct

The set is initially correct for an empty prefix. Inserting each new value preserves its meaning. A hit proves an earlier occurrence, and all earlier encounters were misses, establishing firstness.

Time and space

O(n) expected time and O(n) auxiliary space.

Common mistakes

Sorting loses encounter order. Testing after insertion makes every value look repeated.

Reasoning card

State: seen is the set of all values strictly before the current position.

Invariant: seen contains exactly the previously processed values.

Decision to rehearse: Check membership before adding.

Finish: Return None if the scan ends

Transfer exercise

Return the earliest first occurrence among duplicated values. Explain why this requires information about the future.

013. Valid Anagram

Foundation | Module 02

Problem

Return whether strings a and b contain exactly the same character multiplicities. Case, spaces and punctuation are significant. The empty string is an anagram of itself. Characters are Python string elements; this kata does not normalize Unicode spellings or grapheme clusters.

Examples

Example 1. Order changes but multiplicities match.

```
is_anagram("anagram", "nagaram")
=> True
```

Example 2. The sets match but the counts do not.

```
is_anagram("aab", "abb")
=> False
```

Example 3. Both multisets are empty.

```
is_anagram("", "")
=> True
```

Why this mechanism works

A set loses multiplicity: aab and abb have the same set but are not anagrams. Build a supply count from a, then consume it with b. A missing supply rejects immediately. Equal lengths ensure that successful consumption cannot leave unmatched characters.

State and boundaries

counts stores unused occurrences supplied by a. Each accepted character of b consumes one occurrence.

Protocol

1. Reject unequal lengths.
2. Count every character in a.
3. For each character in b reject an exhausted supply.
4. Decrement its count.
5. Accept after all of b is consumed.

Visualization

```
supply aab: a=2, b=1
consume aba: a -> b -> a
remaining: a=0, b=0
```

Python implementation

```
def is_anagram(a, b):
    if len(a) != len(b):
        return False
    counts = {}
    for char in a:
        counts[char] = counts.get(char, 0) + 1
    for char in b:
        if counts.get(char, 0) == 0:
            return False
        counts[char] -= 1
    return True
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
is_anagram("anagram", "nagaram")

L2 is_anagram: if len(a) != len(b):
a='anagram'; b='nagaram'

L4 is_anagram: counts = {}
a='anagram'; b='nagaram'; counts={}

L5 is_anagram: for char in a:
a='anagram'; b='nagaram'; counts={}; char='a'

L6 is_anagram: counts[char] = counts.get(char, 0) + 1
a='anagram'; b='nagaram'; counts={'a':1}; char='a'

L5 is_anagram: for char in a:
a='anagram'; b='nagaram'; counts={'a':1}; char='n'

L6 is_anagram: counts[char] = counts.get(char, 0) + 1
a='anagram'; b='nagaram'; counts={'a':1, 'n':1}; char='n'

L5 is_anagram: for char in a:
a='anagram'; b='nagaram'; counts={'a':1, 'n':1}; char='a'

L6 is_anagram: counts[char] = counts.get(char, 0) + 1
a='anagram'; b='nagaram'; counts={'a':2, 'n':1}; char='a'

... 18 intermediate state changes omitted ...

L7 is_anagram: for char in b:
a='anagram'; b='nagaram'; counts={'a':1, 'n':0, 'g':0, 'r':0, 'm':1}; char='a'

L10 is_anagram: counts[char] -= 1
a='anagram'; b='nagaram'; counts={'a':0, 'n':0, 'g':0, 'r':0, 'm':1}; char='a'

L7 is_anagram: for char in b:
a='anagram'; b='nagaram'; counts={'a':0, 'n':0, 'g':0, 'r':0, 'm':1}; char='m'
```

```
L10 is_anagram: counts[char] -= 1
a='anagram'; b='nagaram'; counts={'a':0, 'n':0, 'g':0, 'r':0, 'm':0}; char='m'
```

Result: True. Order changes but multiplicities match.

```
is_anagram("aab", "abb")

L2 is_anagram: if len(a) != len(b):
a='aab'; b='abb'

L4 is_anagram: counts = {}
a='aab'; b='abb'; counts={}

L5 is_anagram: for char in a:
a='aab'; b='abb'; counts={}; char='a'

L6 is_anagram: counts[char] = counts.get(char, 0) + 1
a='aab'; b='abb'; counts={'a':1}; char='a'

L6 is_anagram: counts[char] = counts.get(char, 0) + 1
a='aab'; b='abb'; counts={'a':2}; char='a'

L5 is_anagram: for char in a:
a='aab'; b='abb'; counts={'a':2}; char='b'

L6 is_anagram: counts[char] = counts.get(char, 0) + 1
a='aab'; b='abb'; counts={'a':2, 'b':1}; char='b'

L7 is_anagram: for char in b:
a='aab'; b='abb'; counts={'a':2, 'b':1}; char='a'

L10 is_anagram: counts[char] -= 1
a='aab'; b='abb'; counts={'a':1, 'b':1}; char='a'

L7 is_anagram: for char in b:
a='aab'; b='abb'; counts={'a':1, 'b':1}; char='b'

L10 is_anagram: counts[char] -= 1
a='aab'; b='abb'; counts={'a':1, 'b':0}; char='b'
```

Result: False. The sets match but the counts do not.

Why the solution is correct

Counts begin as the exact multiset of a. Every decrement matches one distinct occurrence. Failure proves a multiplicity mismatch; success after equal-length consumption matches every supplied occurrence exactly once.

Time and space

O(n+m) expected time and O(u) space for u distinct characters.

Common mistakes

Comparing sets ignores repeated letters. Accidentally lowercasing changes the stated contract.

Reasoning card

State: counts stores unused occurrences supplied by a. Each accepted character of b consumes one occurrence.

Invariant: Net counts are zero exactly when multiplicities match.

Decision to rehearse: Build and cancel frequencies without losing zero-count logic.

Finish: Accept after all of b is consumed

Transfer exercise

Return the minimum deletions across both strings needed to make their character multisets equal.

014. First Unique Character

Foundation | Module 02

Problem

Return the smallest zero-based index whose character occurs exactly once in the entire string. Return -1 if there is no such character. Comparisons are case-sensitive, and the input is unchanged.

Examples

Example 1. The initial l is globally unique.

```
first_unique("leetcode")
=> 0
```

Example 2. Every character repeats.

```
first_unique("aabb")
=> -1
```

Example 3. There is no index to return.

```
first_unique("")
=> -1
```

Why this mechanism works

A character that looks unique now may recur later. Finish counting before committing to the earliest index. The second pass retains input order, while the dictionary answers the global frequency question.

State and boundaries

counts is complete before selection. i is the next candidate index in original order.

Protocol

1. Count all characters.
2. Scan indices from the beginning.
3. Return the first index with count one.
4. Return -1 if none qualifies.

Visualization

```
leetcode
counts: l=1, e=3, t=1, c=1, o=1, d=1
scan from left -> index 0
```

Python implementation

```
def first_unique(text):
    counts = {}
    for char in text:
        counts[char] = counts.get(char, 0) + 1
    for i, char in enumerate(text):
        if counts[char] == 1:
            return i
    return -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
first_unique("leetcode")

L2 first_unique: counts = {}
text='leetcode'; counts={}

L3 first_unique: for char in text:
text='leetcode'; counts={}; char='l'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='leetcode'; counts={'l':1}; char='l'

L3 first_unique: for char in text:
text='leetcode'; counts={'l':1}; char='e'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='leetcode'; counts={'l':1, 'e':1}; char='e'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='leetcode'; counts={'l':1, 'e':2}; char='e'

L3 first_unique: for char in text:
text='leetcode'; counts={'l':1, 'e':2}; char='t'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='leetcode'; counts={'l':1, 'e':2, 't':1}; char='t'

... 5 intermediate state changes omitted ...

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='leetcode'; counts={'l':1, 'e':2, 't':1, 'c':1, 'o':1, ...}; char='d'

L3 first_unique: for char in text:
text='leetcode'; counts={'l':1, 'e':2, 't':1, 'c':1, 'o':1, ...}; char='e'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='leetcode'; counts={'l':1, 'e':3, 't':1, 'c':1, 'o':1, ...}; char='e'

L5 first_unique: for i, char in enumerate(text):
text='leetcode'; counts={'l':1, 'e':3, 't':1, 'c':1, 'o':1, ...}; char='l'; i=0
```

Result: 0. The initial l is globally unique.

```
first_unique("aabb")

L2 first_unique: counts = {}
text='aabb'; counts={}
```

```
L3 first_unique: for char in text:
text='aabb'; counts={}; char='a'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='aabb'; counts={'a':1}; char='a'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='aabb'; counts={'a':2}; char='a'

L3 first_unique: for char in text:
text='aabb'; counts={'a':2}; char='b'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='aabb'; counts={'a':2, 'b':1}; char='b'

L4 first_unique: counts[char] = counts.get(char, 0) + 1
text='aabb'; counts={'a':2, 'b':2}; char='b'

L5 first_unique: for i, char in enumerate(text):
text='aabb'; counts={'a':2, 'b':2}; char='a'; i=0

L5 first_unique: for i, char in enumerate(text):
text='aabb'; counts={'a':2, 'b':2}; char='a'; i=1

L5 first_unique: for i, char in enumerate(text):
text='aabb'; counts={'a':2, 'b':2}; char='b'; i=2

L5 first_unique: for i, char in enumerate(text):
text='aabb'; counts={'a':2, 'b':2}; char='b'; i=3
```

Result: -1. Every character repeats.

Why the solution is correct

The counting pass determines final multiplicities. The selection pass tests candidates in increasing index order, so its first qualifying index is both globally unique and earliest. Exhaustion means every character is repeated.

Time and space

O(n) expected time and O(u) auxiliary space.

Common mistakes

Returning during the counting pass commits before future occurrences are known. Returning a dictionary key does not provide the requested index.

Reasoning card

State: counts is complete before selection. i is the next candidate index in original order.

Invariant: The frequency map is complete before choosing the answer.

Decision to rehearse: Separate the counting pass from the ordered selection pass.

Finish: Return -1 if none qualifies

Transfer exercise

Maintain the first unique character after every stream insertion; identify the extra order state a streaming solution needs.

015. Group Anagrams

Bridge | Module 02

Problem

Group strings that have the same character multiset. Preserve input order within each group and order groups by the first occurrence of their signature. Duplicate words remain duplicated. Characters are case-sensitive; return [] for an empty word list.

Examples

Example 1. Group and member order follow first encounters.

```
group_anagrams(["eat", "tea", "tan", "ate", "nat", "bat"])
=> [['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']]
```

Example 2. There are no groups.

```
group_anagrams([])
=> []
```

Example 3. Empty words share the empty signature.

```
group_anagrams(["", "", "a"])
=> [['', ''], ['a']]
```

Why this mechanism works

Convert an equivalence relation into a canonical key. Sorting a word's characters yields the same tuple exactly when its multiplicities match another word's. A dictionary then gathers each equivalence class without comparing every pair of words.

State and boundaries

groups maps a sorted-character tuple to the words seen with that signature, in encounter order.

Protocol

1. Start an empty dictionary.
2. Sort each word's characters into a tuple.
3. Create its group if absent.
4. Append the original word.
5. Return dictionary groups in insertion order.

Visualization

```
eat, tea, ate -> ('a','e','t')
tan, nat      -> ('a','n','t')
bat           -> ('a','b','t')
```

Python implementation

```
def group_anagrams(words):
    groups = {}
    for word in words:
        key = tuple(sorted(word))
        groups.setdefault(key, []).append(word)
    return list(groups.values())
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
group_anagrams(["eat", "tea", "tan", "ate", "nat", "bat"])

L2 group_anagrams: groups = {}
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={}

L3 group_anagrams: for word in words:
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={}; word='eat'

L4 group_anagrams: key = tuple(sorted(word))
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={}; word='eat'; key=('a', 'e', 't')

L5 group_anagrams: groups.setdefault(key, []).append(word)
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat']};
    word='eat'; key=('a', 'e', 't')

L3 group_anagrams: for word in words:
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat']};
    word='tea'; key=('a', 'e', 't')

L5 group_anagrams: groups.setdefault(key, []).append(word)
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat', 'tea']};
    word='tea'; key=('a', 'e', 't')

L3 group_anagrams: for word in words:
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat', 'tea']};
    word='tan'; key=('a', 'e', 't')

L4 group_anagrams: key = tuple(sorted(word))
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat', 'tea']};
    word='tan'; key=('a', 'n', 't')

... 6 intermediate state changes omitted ...

L5 group_anagrams: groups.setdefault(key, []).append(word)
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat', 'tea',
    'ate'], ('a', 'n', 't'):['tan', 'nat']}; word='nat'; key=('a', 'n', 't')

L3 group_anagrams: for word in words:
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat', 'tea',
    'ate'], ('a', 'n', 't'):['tan', 'nat']}; word='bat'; key=('a', 'n', 't')
```

```
L4 group_anagrams: key = tuple(sorted(word))
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat', 'tea',
    'ate'], ('a', 'n', 't'):['tan', 'nat']}; word='bat'; key=('a', 'b', 't')

L5 group_anagrams: groups.setdefault(key, []).append(word)
words=['eat', 'tea', 'tan', 'ate', 'nat', 'bat']; groups={('a', 'e', 't'):['eat', 'tea',
    'ate'], ('a', 'n', 't'):['tan', 'nat'], ('a', 'b', 't'):['ba...; word='bat'; key=('a',
    'b', 't')
```

Result: [['eat', 'tea', 'ate'], ['tan', 'nat'], ['bat']]. Group and member order follow first encounters.

```
group_anagrams([])

L2 group_anagrams: groups = {}
words=[]; groups={}
```

Result: []. There are no groups.

Why the solution is correct

Equal keys imply equal sorted sequences and therefore equal character counts. Anagrams necessarily produce equal keys. Each word is appended once to its unique correct group, and dictionary insertion order enforces the requested group ordering.

Time and space

O(sum(L_i log L_i)) sorting time, plus O(sum L_i) key/output-reference storage in the worst case.

Common mistakes

A list cannot be a dictionary key. Using a set signature incorrectly groups ab with aab.

Reasoning card

State: groups maps a sorted-character tuple to the words seen with that signature, in encounter order.

Invariant: Words sharing a key are equivalent under the chosen signature.

Decision to rehearse: Create a canonical hashable signature.

Finish: Return dictionary groups in insertion order

Transfer exercise

For lowercase English letters only, replace sorting with a 26-count tuple and derive the new cost.

016. Most Frequent with Earliest Tie

Bridge | Module 02

Problem

Return the most frequent integer, breaking ties by its first appearance in nums. Empty input returns None. Frequency is measured over the whole list, not merely the prefix at which a value first becomes a temporary leader.

Examples

Example 1. Two and one tie; two appeared first.

```
most_frequent_earliest([2, 1, 2, 1])
=> 2
```

Example 2. No candidate exists.

```
most_frequent_earliest([])
=> None
```

Example 3. A temporary lead by one must not decide the final tie.

```
most_frequent_earliest([9, 1, 1, 9])
=> 9
```

Why this mechanism works

First build complete frequencies. Then scan the original order and replace the answer only on a strictly larger final frequency. The first candidate at a tied frequency therefore stays ahead. Separating counting and selection makes the tie proof simpler than maintaining a moving leader.

State and boundaries

counts contains final frequencies. best is the earliest encountered value among those with the largest frequency examined so far.

Protocol

1. Handle empty input.
2. Build frequencies.
3. Seed best from the first value.
4. Replace best only for a strictly greater final count.

5. Return best.

Visualization

```
input: 2 1 2 1
final: count(2)=2, count(1)=2
first encounter wins -> 2
```

Python implementation

```
def most_frequent_earliest(nums):
    if not nums:
        return None
    counts = {}
    for value in nums:
        counts[value] = counts.get(value, 0) + 1
    best = nums[0]
    for value in nums:
        if counts[value] > counts[best]:
            best = value
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
most_frequent_earliest([2, 1, 2, 1])

L2 most_frequent_earliest: if not nums:
nums=[2, 1, 2, 1]

L4 most_frequent_earliest: counts = {}
nums=[2, 1, 2, 1]; counts={}

L5 most_frequent_earliest: for value in nums:
nums=[2, 1, 2, 1]; counts={}; value=2

L6 most_frequent_earliest: counts[value] = counts.get(value, 0) + 1
nums=[2, 1, 2, 1]; counts={2:1}; value=2

L5 most_frequent_earliest: for value in nums:
nums=[2, 1, 2, 1]; counts={2:1}; value=1

L6 most_frequent_earliest: counts[value] = counts.get(value, 0) + 1
nums=[2, 1, 2, 1]; counts={2:1, 1:1}; value=1

L5 most_frequent_earliest: for value in nums:
nums=[2, 1, 2, 1]; counts={2:1, 1:1}; value=2

L6 most_frequent_earliest: counts[value] = counts.get(value, 0) + 1
nums=[2, 1, 2, 1]; counts={2:2, 1:1}; value=2

... 3 intermediate state changes omitted ...

L8 most_frequent_earliest: for value in nums:
nums=[2, 1, 2, 1]; counts={2:2, 1:2}; value=2; best=2

L8 most_frequent_earliest: for value in nums:
nums=[2, 1, 2, 1]; counts={2:2, 1:2}; value=1; best=2
```

```
L8 most_frequent_earliest: for value in nums:
nums=[2, 1, 2, 1]; counts={2:2, 1:2}; value=2; best=2

L8 most_frequent_earliest: for value in nums:
nums=[2, 1, 2, 1]; counts={2:2, 1:2}; value=1; best=2
```

Result: 2. Two and one tie; two appeared first.

```
most_frequent_earliest([])

L2 most_frequent_earliest: if not nums:
nums=[]
```

Result: None. No candidate exists.

Why the solution is correct

After counting, all comparisons use final rather than provisional frequencies. Strict replacement picks a larger frequency but never displaces an earlier equal-frequency value. The scan thus optimizes frequency first and occurrence order second.

Time and space

O(n) expected time and O(u) space.

Common mistakes

Using >= lets a later tied value win. A one-pass leader updated only on larger current counts does not automatically implement final-frequency earliest ties.

Reasoning card

State: counts contains final frequencies. best is the earliest encountered value among those with the largest frequency examined so far.

Invariant: The chosen candidate obeys both frequency and tie rules.

Decision to rehearse: Coordinate frequency and first-position state.

Finish: Return best

Transfer exercise

Break ties by the smallest value instead; express the two-part ordering explicitly.

017. Isomorphic Strings

Bridge | Module 02

Problem

Return whether two strings admit a consistent one-to-one character substitution from a to b. Each source character maps to one target character, and two different source characters cannot share a target. Lengths must match. A character may map to itself.

Examples

Example 1. Repeated g consistently maps to d.

```
isomorphic("egg", "add")
=> True
```

Example 2. Two source characters cannot map to one target.

```
isomorphic("ab", "xx")
=> False
```

Example 3. The empty mapping is valid.

```
isomorphic("", "")
=> True
```

Why this mechanism works

A forward map enforces consistency but not injectivity: it would allow both a and b to map to x. A reverse map enforces that the target character has only one source. Check both directions before extending the mapping.

State and boundaries

forward and reverse encode mutually consistent character pairs established by earlier positions.

Protocol

1. Reject unequal lengths.
2. For each aligned character pair check any existing forward mapping.
3. Check the reverse mapping too.
4. Record the pair in both dictionaries.
5. Accept when all pairs pass.

Visualization

```
egg -> add
e <-> a
g <-> d
second g reuses g <-> d
```

Python implementation

```
def isomorphic(a, b):
    if len(a) != len(b):
        return False
    forward, reverse = {}, {}
    for x, y in zip(a, b):
        if x in forward and forward[x] != y:
            return False
        if y in reverse and reverse[y] != x:
            return False
        forward[x], reverse[y] = y, x
    return True
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
isomorphic("egg", "add")

L2 isomorphic: if len(a) != len(b):
a='egg'; b='add'

L4 isomorphic: forward, reverse = {}, {}
a='egg'; b='add'; forward={}; reverse={}

L5 isomorphic: for x, y in zip(a, b):
a='egg'; b='add'; forward={}; reverse={}; x='e'; y='a'

L10 isomorphic: forward[x], reverse[y] = y, x
a='egg'; b='add'; forward={'e':'a'}; reverse={'a':'e'}; x='e'; y='a'

L5 isomorphic: for x, y in zip(a, b):
a='egg'; b='add'; forward={'e':'a'}; reverse={'a':'e'}; x='g'; y='d'

L10 isomorphic: forward[x], reverse[y] = y, x
a='egg'; b='add'; forward={'e':'a', 'g':'d'}; reverse={'a':'e', 'd':'g'}; x='g'; y='d'
```

Result: True. Repeated g consistently maps to d.

```
isomorphic("ab", "xx")

L2 isomorphic: if len(a) != len(b):
a='ab'; b='xx'

L4 isomorphic: forward, reverse = {}, {}
a='ab'; b='xx'; forward={}; reverse={}

L5 isomorphic: for x, y in zip(a, b):
a='ab'; b='xx'; forward={}; reverse={}; x='a'; y='x'

L10 isomorphic: forward[x], reverse[y] = y, x
a='ab'; b='xx'; forward={'a':'x'}; reverse={'x':'a'}; x='a'; y='x'
```

```
L5 isomorphic: for x, y in zip(a, b):
a='ab'; b='xx'; forward={'a':'x'}; reverse={'x':'a'}; x='b'; y='x'
```

Result: False. Two source characters cannot map to one target.

Why the solution is correct

Each accepted pair preserves functional consistency in both directions. The resulting forward relation is therefore one-to-one and explains every aligned position. Any rejected pair contradicts a required earlier mapping, so no valid substitution could exist.

Time and space

O(n) expected time and O(u+v) space for distinct characters in the strings.

Common mistakes

A forward-only map accepts ab -> xx incorrectly. Frequency equality is neither the intended order constraint nor a substitute for checking aligned positions.

Reasoning card

State: forward and reverse encode mutually consistent character pairs established by earlier positions.

Invariant: Every established mapping remains consistent and bijective.

Decision to rehearse: Maintain both forward and reverse mappings.

Finish: Accept when all pairs pass

Transfer exercise

Use the same bijection idea to match a word pattern such as abba against a sentence.

018. Multiset Intersection

Interview | Module 02

Problem

Return common integers with multiplicity: value x appears min(count_a(x), count_b(x)) times. Emit matches in the order they are consumed from b. Do not mutate either list. Empty input on either side yields [].

Examples

Example 1. Both available copies are consumed.

```
multiset_intersection([1, 2, 2, 1], [2, 2])
=> [2, 2]
```

Example 2. There is no supply.

```
multiset_intersection([], [1])
=> []
```

Example 3. One supply occurrence cannot be reused.

```
multiset_intersection([2], [2, 2])
=> [2]
```

Why this mechanism works

Treat a as a finite supply and b as demand. Finding a value is insufficient after all copies have been consumed. Decrementing remaining supply after each committed match ensures that the same occurrence cannot serve several requests.

State and boundaries

supply[x] is the number of unmatched x occurrences from a. output records fulfilled demand from b.

Protocol

1. Count supply from a.
2. Scan b in order.
3. Append only when supply is positive.
4. Decrement immediately after appending.
5. Return output.

Visualization

```
a supply: 1=2, 2=2
b demand: 2, 2, 2
fulfilled: 2, 2; final request rejected
```

Python implementation

```
def multiset_intersection(a, b):
    supply = {}
    for value in a:
        supply[value] = supply.get(value, 0) + 1
    output = []
    for value in b:
        if supply.get(value, 0) > 0:
            output.append(value)
            supply[value] -= 1
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
multiset_intersection([1, 2, 2, 1], [2, 2])

L2 multiset_intersection: supply = {}
a=[1, 2, 2, 1]; b=[2, 2]; supply={}

L3 multiset_intersection: for value in a:
a=[1, 2, 2, 1]; b=[2, 2]; supply={}; value=1

L4 multiset_intersection: supply[value] = supply.get(value, 0) + 1
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:1}; value=1

L3 multiset_intersection: for value in a:
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:1}; value=2

L4 multiset_intersection: supply[value] = supply.get(value, 0) + 1
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:1, 2:1}; value=2

L4 multiset_intersection: supply[value] = supply.get(value, 0) + 1
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:1, 2:2}; value=2

L3 multiset_intersection: for value in a:
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:1, 2:2}; value=1

L4 multiset_intersection: supply[value] = supply.get(value, 0) + 1
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:2, 2:2}; value=1

... 2 intermediate state changes omitted ...

L8 multiset_intersection: output.append(value)
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:2, 2:2}; value=2; output=[2]

L9 multiset_intersection: supply[value] -= 1
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:2, 2:1}; value=2; output=[2]

L8 multiset_intersection: output.append(value)
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:2, 2:1}; value=2; output=[2, 2]
```

```
L9 multiset_intersection: supply[value] -= 1
a=[1, 2, 2, 1]; b=[2, 2]; supply={1:2, 2:0}; value=2; output=[2, 2]
```

Result: [2, 2]. Both available copies are consumed.

```
multiset_intersection([], [1])

L2 multiset_intersection: supply = {}
a=[]; b=[1]; supply={}

L5 multiset_intersection: output = []
a=[]; b=[1]; supply={}; output=[]

L6 multiset_intersection: for value in b:
a=[]; b=[1]; supply={}; output=[]; value=1
```

Result: []. There is no supply.

Why the solution is correct

A match consumes a previously unused occurrence from each list. Demand is fulfilled until one side's multiplicity is exhausted, giving exactly the smaller count. Scanning b directly provides the required output order.

Time and space

O(n+m) expected time; O(u) supply state plus O(min(n,m)) output.

Common mistakes

A set intersection loses multiplicity. Forgetting to decrement can overproduce repeated matches.

Reasoning card

State: supply[x] is the number of unmatched x occurrences from a. output records fulfilled demand from b.

Invariant: Remaining counts represent unmatched supply.

Decision to rehearse: Consume counts as matches are committed.

Finish: Return output

Transfer exercise

When both inputs are sorted, solve the same matching problem with two pointers and no frequency table.

019. Longest Consecutive Sequence

Interview | Module 02

Problem

Return the length of the longest sequence of consecutive integer values present in an unsorted list. Their positions need not be adjacent or increasing. Duplicate occurrences do not extend a sequence. Empty input returns zero.

Examples

Example 1. The value sequence 1,2,3,4 is present.

```
longest_consecutive([100, 4, 200, 1, 3, 2])
=> 4
```

Example 2. No sequence exists.

```
longest_consecutive([])
=> 0
```

Example 3. Duplicates do not add sequence length.

```
longest_consecutive([1, 2, 2, 3])
=> 3
```

Why this mechanism works

A set gives constant expected membership checks, but expanding from every value would repeat long runs. Only expand x when x-1 is absent. That marks x as a sequence's unique start, so the total number of successful extension steps is bounded by the number of distinct values.

State and boundaries

values contains distinct input values. start is a candidate minimum; end is the first unconfirmed value beyond its currently verified consecutive run.

Protocol

1. Build a set.
2. Skip each value whose predecessor exists.
3. From each true start advance end while it exists.
4. Compare end-start with best.

5. Return best.

Visualization

```
{100,4,200,1,3,2}
starts: 100, 200, 1
1 -> 2 -> 3 -> 4 -> missing 5
```

Python implementation

```
def longest_consecutive(nums):
    values = set(nums)
    best = 0
    for start in values:
        if start - 1 in values:
            continue
        end = start
        while end in values:
            end += 1
        best = max(best, end - start)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
longest_consecutive([100, 4, 200, 1, 3, 2])

L2 longest_consecutive: values = set(nums)
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}

L3 longest_consecutive: best = 0
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=0

L4 longest_consecutive: for start in values:
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=0; start=1

L7 longest_consecutive: end = start
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=0; start=1; end=1

L9 longest_consecutive: end += 1
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=0; start=1; end=2

L9 longest_consecutive: end += 1
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=0; start=1; end=3

L9 longest_consecutive: end += 1
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=0; start=1; end=4

L9 longest_consecutive: end += 1
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=0; start=1; end=5

... 6 intermediate state changes omitted ...

L4 longest_consecutive: for start in values:
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=4; start=4; end=101

L4 longest_consecutive: for start in values:
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=4; start=200; end=101
```

```
L7 longest_consecutive: end = start
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=4; start=200; end=200

L9 longest_consecutive: end += 1
nums=[100, 4, 200, 1, 3, 2]; values={1, 100, 2, 200, 3, 4}; best=4; start=200; end=201
```

Result: 4. The value sequence 1,2,3,4 is present.

```
longest_consecutive([])

L2 longest_consecutive: values = set(nums)
nums=[]; values=set()

L3 longest_consecutive: best = 0
nums=[]; values=set(); best=0
```

Result: 0. No sequence exists.

Why the solution is correct

Every maximal consecutive sequence has exactly one value with no predecessor. Expansion from that value discovers its full length and no other start repeats its work. Taking the maximum over these disjoint sequences gives the answer.

Time and space

O(n) expected time and O(u) space. The nested-looking expansions visit each distinct value only once across all starts.

Common mistakes

Expanding from every value can be quadratic. Sorting is a valid O(n log n) alternative but is not the hash-state mechanism practiced here.

Reasoning card

State: values contains distinct input values. start is a candidate minimum; end is the first unconfirmed value beyond its currently verified consecutive run.

Invariant: Each sequence is expanded exactly once from its smallest value.

Decision to rehearse: Start counting only at sequence starts.

Finish: Return best

Transfer exercise

Return the actual sequence, breaking equal-length ties by its smallest starting value.

020. Count Equal Pairs

Interview | Module 02

Problem

Count pairs of indices (i,j) satisfying i<j and nums[i]==nums[j]. Return an integer, not the pairs themselves. Each pair of occurrences counts separately, and an empty or singleton list has zero pairs.

Examples

Example 1. Three pairs of ones and one pair of threes.

```
equal_pairs([1, 2, 3, 1, 1, 3])
=> 4
```

Example 2. No pair has two endpoints.

```
equal_pairs([])
=> 0
```

Example 3. Four occurrences yield 4*3/2 pairs.

```
equal_pairs([7, 7, 7, 7])
=> 6
```

Why this mechanism works

When a new occurrence arrives, it forms one new pair with every earlier equal value. Add the earlier frequency before increasing it. Each pair has one later endpoint, so this charges it exactly once and avoids quadratic pair enumeration.

State and boundaries

frequency[x] counts earlier x occurrences. pairs counts valid pairs entirely inside the processed prefix.

Protocol

1. Initialize empty frequencies and zero pairs.
2. Read the incoming value's earlier count.
3. Add that count to pairs.
4. Increase its frequency.
5. Return pairs.

Visualization

```
occurrences of 1: first, second, third
new pairs:          0       1      2
total for 1: 3
```

Python implementation

```
def equal_pairs(nums):
    frequency = {}
    pairs = 0
    for value in nums:
        earlier = frequency.get(value, 0)
        pairs += earlier
        frequency[value] = earlier + 1
    return pairs
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
equal_pairs([1, 2, 3, 1, 1, 3])

L2 equal_pairs: frequency = {}
nums=[1, 2, 3, 1, 1, 3]; frequency={}

L3 equal_pairs: pairs = 0
nums=[1, 2, 3, 1, 1, 3]; frequency={}; pairs=0

L4 equal_pairs: for value in nums:
nums=[1, 2, 3, 1, 1, 3]; frequency={}; pairs=0; value=1

L5 equal_pairs: earlier = frequency.get(value, 0)
nums=[1, 2, 3, 1, 1, 3]; frequency={}; pairs=0; value=1; earlier=0

L7 equal_pairs: frequency[value] = earlier + 1
nums=[1, 2, 3, 1, 1, 3]; frequency={1:1}; pairs=0; value=1; earlier=0

L4 equal_pairs: for value in nums:
nums=[1, 2, 3, 1, 1, 3]; frequency={1:1}; pairs=0; value=2; earlier=0

L7 equal_pairs: frequency[value] = earlier + 1
nums=[1, 2, 3, 1, 1, 3]; frequency={1:1, 2:1}; pairs=0; value=2; earlier=0

L4 equal_pairs: for value in nums:
nums=[1, 2, 3, 1, 1, 3]; frequency={1:1, 2:1}; pairs=0; value=3; earlier=0

... 8 intermediate state changes omitted ...

L4 equal_pairs: for value in nums:
nums=[1, 2, 3, 1, 1, 3]; frequency={1:3, 2:1, 3:1}; pairs=3; value=3; earlier=2

L5 equal_pairs: earlier = frequency.get(value, 0)
nums=[1, 2, 3, 1, 1, 3]; frequency={1:3, 2:1, 3:1}; pairs=3; value=3; earlier=1

L6 equal_pairs: pairs += earlier
nums=[1, 2, 3, 1, 1, 3]; frequency={1:3, 2:1, 3:1}; pairs=4; value=3; earlier=1
```

```
L7 equal_pairs: frequency[value] = earlier + 1
nums=[1, 2, 3, 1, 1, 3]; frequency={1:3, 2:1, 3:2}; pairs=4; value=3; earlier=1
```

Result: 4. Three pairs of ones and one pair of threes.

```
equal_pairs([])

L2 equal_pairs: frequency = {}
nums=[]; frequency={}

L3 equal_pairs: pairs = 0
nums=[]; frequency={}; pairs=0
```

Result: 0. No pair has two endpoints.

Why the solution is correct

For each new later endpoint j, the earlier equal positions are exactly those counted by frequency. These new pairs are disjoint from pairs already counted and cover all pairs ending at j. Induction over endpoints accounts for every valid pair once.

Time and space

O(n) expected time and O(u) space; Python integers can represent counts exceeding fixed-width ranges.

Common mistakes

Incrementing first counts a value paired with itself. Counting distinct repeated values answers a different question.

Reasoning card

State: frequency[x] counts earlier x occurrences. pairs counts valid pairs entirely inside the processed prefix.

Invariant: frequency[x] is the number of earlier x values.

Decision to rehearse: Each new occurrence pairs with all previous occurrences.

Finish: Return pairs

Transfer exercise

Derive the equivalent final-frequency formula sum(c*(c-1)//2), then explain the streaming version's advantage.

Module 03
Sorting and Ordered Scans

Turn global relationships into local comparisons by imposing useful order.

Sorting deliberately changes the order in which evidence arrives. Equal values become neighbors, intervals can be consumed from left to right, and a current frontier can summarize everything already settled. Identify what ordering buys before paying its cost. Keep the input-order tie policy separate from the sorted processing order.

The ten-step progression

021. Merge Two Sorted Arrays

022. Deduplicate a Sorted Array

023. Three-Way Sort 0/1/2

024. Rank Transform

025. Merge Overlapping Intervals

026. Insert One Interval

027. Intersect Two Interval Lists

028. Closest Pair to Target

029. Minimum-Difference Pairs

030. Sort Records by Multiple Keys

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

021. Merge Two Sorted Arrays

Foundation | Module 03

Problem

Merge two nondecreasing integer lists into a new nondecreasing list containing every occurrence. Neither input is modified. Empty inputs are allowed. When equal values meet, consume from a first; this makes the merge stable with respect to the two source streams.

Examples

Example 1. Each smaller frontier is consumed.

```
merge_sorted([1, 4, 7], [2, 3, 9])
=> [1, 2, 3, 4, 7, 9]
```

Example 2. All remaining values come from b.

```
merge_sorted([], [2, 2])
=> [2, 2]
```

Example 3. Multiplicity is retained.

```
merge_sorted([1, 1], [1])
=> [1, 1, 1]
```

Why this mechanism works

Only the first unread value of each sorted source can be its next contribution. Compare those two frontier values and consume the smaller one. Once a source is exhausted, the other unread suffix is already sorted and can be appended without more comparisons.

State and boundaries

i and j are unread boundaries in a and b. output is the sorted merge of the consumed prefixes.

Protocol

1. Start both boundaries at zero.
2. Compare while both sources have candidates.
3. Append the smaller frontier and advance only its source.
4. Append either remaining suffix.
5. Return output.

Visualization

```
a: [1 | 4,7]
b: [2,3,9]
output: [1] -> compare 4 with 2
```

Python implementation

```
def merge_sorted(a, b):
    i = j = 0
    output = []
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            output.append(a[i])
            i += 1
        else:
            output.append(b[j])
            j += 1
    output.extend(a[i:])
    output.extend(b[j:])
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
merge_sorted([1, 4, 7], [2, 3, 9])

L2 merge_sorted: i = j = 0
a=[1, 4, 7]; b=[2, 3, 9]; i=0; j=0

L3 merge_sorted: output = []
a=[1, 4, 7]; b=[2, 3, 9]; i=0; j=0; output=[]

L6 merge_sorted: output.append(a[i])
a=[1, 4, 7]; b=[2, 3, 9]; i=0; j=0; output=[1]

L7 merge_sorted: i += 1
a=[1, 4, 7]; b=[2, 3, 9]; i=1; j=0; output=[1]

L9 merge_sorted: output.append(b[j])
a=[1, 4, 7]; b=[2, 3, 9]; i=1; j=0; output=[1, 2]

L10 merge_sorted: j += 1
a=[1, 4, 7]; b=[2, 3, 9]; i=1; j=1; output=[1, 2]

L9 merge_sorted: output.append(b[j])
a=[1, 4, 7]; b=[2, 3, 9]; i=1; j=1; output=[1, 2, 3]

L10 merge_sorted: j += 1
a=[1, 4, 7]; b=[2, 3, 9]; i=1; j=2; output=[1, 2, 3]

... 1 intermediate state changes omitted ...

L7 merge_sorted: i += 1
a=[1, 4, 7]; b=[2, 3, 9]; i=2; j=2; output=[1, 2, 3, 4]

L6 merge_sorted: output.append(a[i])
a=[1, 4, 7]; b=[2, 3, 9]; i=2; j=2; output=[1, 2, 3, 4, 7]
```

```
L7 merge_sorted: i += 1
a=[1, 4, 7]; b=[2, 3, 9]; i=3; j=2; output=[1, 2, 3, 4, 7]

L12 merge_sorted: output.extend(b[j:])
a=[1, 4, 7]; b=[2, 3, 9]; i=3; j=2; output=[1, 2, 3, 4, 7, 9]
```

Result: [1, 2, 3, 4, 7, 9]. Each smaller frontier is consumed.

```
merge_sorted([], [2, 2])

L2 merge_sorted: i = j = 0
a=[]; b=[2, 2]; i=0; j=0

L3 merge_sorted: output = []
a=[]; b=[2, 2]; i=0; j=0; output=[]

L12 merge_sorted: output.extend(b[j:])
a=[]; b=[2, 2]; i=0; j=0; output=[2, 2]
```

Result: [2, 2]. All remaining values come from b.

Why the solution is correct

The smaller frontier is no larger than any remaining value in either sorted source, so appending it preserves order. Each advance consumes exactly one occurrence. Tail copying finishes the remaining occurrences without changing order.

Time and space

O(n+m) time and O(n+m) output space; suffix slicing also creates temporary copies of at most that size.

Common mistakes

Advancing both pointers drops an unconsumed value. Forgetting the tail loses everything after the shorter input ends.

Reasoning card

State: i and j are unread boundaries in a and b. output is the sorted merge of the consumed prefixes.

Invariant: Output is sorted and contains all consumed prefixes.

Decision to rehearse: Advance only the source whose item was consumed.

Finish: Return output

Transfer exercise

Merge into spare capacity at the end of a. Explain why writing from the back protects unread input.

022. Deduplicate a Sorted Array

Foundation | Module 03

Problem

Modify a nondecreasing integer list so its first k positions contain the distinct values in sorted order, and return k. Values beyond k are unspecified and the physical list length need not change. Empty input returns zero.

Examples

Example 1. The meaningful prefix has three values.

```
a = [1, 1, 2, 2, 3]
(deduplicate_sorted(a), a[:3])
=> (3, [1, 2, 3])
```

Example 2. There is no meaningful prefix.

```
deduplicate_sorted([])
=> 0
```

Example 3. One duplicate run contributes one value.

```
a = [7, 7, 7]
(deduplicate_sorted(a), a[:1])
=> (1, [7])
```

Why this mechanism works

Sorting places equal values together. Compare each incoming value with the last value already committed, not with a cell whose meaning is vague after mutation. The write frontier never overtakes the read frontier, so compacting accepted values cannot destroy unread data.

State and boundaries

write is the meaningful output length. nums[:write] contains the distinct values of the consumed prefix.

Protocol

1. Start write at zero.
2. Read each value from left to right.
3. Accept when the output is empty or its last value differs.
4. Write the accepted value and advance write.
5. Return write.

Visualization

```
read:  [1,1,2,2,3]
write: [1,2,3 | ?,?]
meaningful interval: [0,3)
```

Python implementation

```
def deduplicate_sorted(nums):
    write = 0
    for value in nums:
        if write == 0 or nums[write - 1] != value:
            nums[write] = value
            write += 1
    return write
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
(deduplicate_sorted(a), a[:3])

L2 deduplicate_sorted: write = 0
nums=[1, 1, 2, 2, 3]; write=0

L3 deduplicate_sorted: for value in nums:
nums=[1, 1, 2, 2, 3]; write=0; value=1

L6 deduplicate_sorted: write += 1
nums=[1, 1, 2, 2, 3]; write=1; value=1

L3 deduplicate_sorted: for value in nums:
nums=[1, 1, 2, 2, 3]; write=1; value=2

L5 deduplicate_sorted: nums[write] = value
nums=[1, 2, 2, 2, 3]; write=1; value=2

L6 deduplicate_sorted: write += 1
nums=[1, 2, 2, 2, 3]; write=2; value=2

L3 deduplicate_sorted: for value in nums:
nums=[1, 2, 2, 2, 3]; write=2; value=3

L5 deduplicate_sorted: nums[write] = value
nums=[1, 2, 3, 2, 3]; write=2; value=3

L6 deduplicate_sorted: write += 1
nums=[1, 2, 3, 2, 3]; write=3; value=3
```

Result: (3, [1, 2, 3]). The meaningful prefix has three values.

```
deduplicate_sorted([])

L2 deduplicate_sorted: write = 0
nums=[]; write=0
```

Result: 0. There is no meaningful prefix.

Why the solution is correct

The first accepted value starts a distinct sorted prefix. Later equal values add no new distinct value; later different values are larger and can be appended safely. Every distinct run contributes exactly once, and unread cells remain available.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Returning the whole physical list exposes unspecified tail cells. Comparing with nums[write] tests an uncommitted location.

Reasoning card

State: write is the meaningful output length. nums[:write] contains the distinct values of the consumed prefix.

Invariant: The prefix before write is unique and final.

Decision to rehearse: Compare incoming values with the last committed output.

Finish: Return write

Transfer exercise

Allow each distinct value at most twice. Which previously committed output cell should the incoming value be compared with?

023. Three-Way Sort 0/1/2

Foundation | Module 03

Problem

Sort a mutable list containing only 0, 1 and 2 in one pass, using constant auxiliary space. Return the same list. Empty input is valid. Values outside these three categories are outside the contract.

Examples

Example 1. Unknown replacements are inspected again.

```
sort_three([2, 0, 2, 1, 1, 0])
=> [0, 0, 1, 1, 2, 2]
```

Example 2. The unknown region is initially empty.

```
sort_three([])
=> []
```

Example 3. Repeated twos expose the no-advance branch.

```
sort_three([2, 2, 0])
=> [0, 2, 2]
```

Why this mechanism works

Maintain a classified prefix of zeros, a following region of ones, an unknown middle and a suffix of twos. Sending a two to the right brings an unknown value back, so inspect that replacement before advancing. Sending a zero left brings back a known one, or swaps the current cell with itself.

State and boundaries

[0,low) is zero; [low,current) is one; [current,high) is unknown; [high,n) is two. high is exclusive.

Protocol

1. Set low=current=0 and high=n.
2. While unknown cells remain inspect nums[current].
3. For zero swap with low and advance both.
4. For one advance current.
5. For two shrink high, swap, and keep current fixed.

Visualization

```
[ zeros | ones | unknown | twos ]
0      low   current   high    n
               2 -> right; inspect replacement
```

Python implementation

```
def sort_three(nums):
    low = current = 0
    high = len(nums)
    while current < high:
        if nums[current] == 0:
            nums[low], nums[current] = nums[current], nums[low]
            low += 1
            current += 1
        elif nums[current] == 1:
            current += 1
        else:
            high -= 1
            nums[current], nums[high] = nums[high], nums[current]
    return nums
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
sort_three([2, 0, 2, 1, 1, 0])

L2 sort_three: low = current = 0
nums=[2, 0, 2, 1, 1, 0]; low=0; current=0

L3 sort_three: high = len(nums)
nums=[2, 0, 2, 1, 1, 0]; low=0; current=0; high=6

L12 sort_three: high -= 1
nums=[2, 0, 2, 1, 1, 0]; low=0; current=0; high=5

L13 sort_three: nums[current], nums[high] = nums[high], nums[current]
nums=[0, 0, 2, 1, 1, 2]; low=0; current=0; high=5

L7 sort_three: low += 1
nums=[0, 0, 2, 1, 1, 2]; low=1; current=0; high=5

L8 sort_three: current += 1
nums=[0, 0, 2, 1, 1, 2]; low=1; current=1; high=5

L7 sort_three: low += 1
nums=[0, 0, 2, 1, 1, 2]; low=2; current=1; high=5

L8 sort_three: current += 1
nums=[0, 0, 2, 1, 1, 2]; low=2; current=2; high=5

L12 sort_three: high -= 1
nums=[0, 0, 2, 1, 1, 2]; low=2; current=2; high=4

L13 sort_three: nums[current], nums[high] = nums[high], nums[current]
nums=[0, 0, 1, 1, 2, 2]; low=2; current=2; high=4

L10 sort_three: current += 1
```

```
nums=[0, 0, 1, 1, 2, 2]; low=2; current=3; high=4

L10 sort_three: current += 1
nums=[0, 0, 1, 1, 2, 2]; low=2; current=4; high=4
```

Result: [0, 0, 1, 1, 2, 2]. Unknown replacements are inspected again.

```
sort_three([])

L2 sort_three: low = current = 0
nums=[]; low=0; current=0

L3 sort_three: high = len(nums)
nums=[]; low=0; current=0; high=0
```

Result: []. The unknown region is initially empty.

Why the solution is correct

Each branch extends a correctly classified region while preserving the others. The unknown interval loses one cell per iteration, proving termination. When it is empty, the three classified regions form the required sorted order.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Advancing current after swapping a two can skip another two or a zero. Mixing inclusive and exclusive high conventions corrupts the region boundaries.

Reasoning card

State: [0,low) is zero; [low,current) is one; [current,high) is unknown; [high,n) is two. high is exclusive.

Invariant: Left is all 0, middle processed region all 1, right all 2.

Decision to rehearse: Maintain three classified regions; one branch must not advance current.

Finish: For two shrink high, swap, and keep current fixed

Transfer exercise

Partition arbitrary values relative to a pivot into smaller, equal and larger regions.

024. Rank Transform

Foundation | Module 03

Problem

Replace each integer by its one-based rank among the distinct values in the input. Equal values receive equal ranks and adjacent distinct ranks have no gaps. Return a new list in original position order; empty input returns [].

Examples

Example 1. Ranks are returned in original order.

```
rank_transform([40, 10, 20, 30])
=> [4, 1, 2, 3]
```

Example 2. Duplicates share a rank.

```
rank_transform([5, 5, 1])
=> [2, 2, 1]
```

Example 3. The rank map is empty.

```
rank_transform([])
=> []
```

Why this mechanism works

The rank belongs to a value's position in the ordered set of distinct values, not to its original index. Build the complete rank map first, then project it back onto the original sequence. This separates ordering information from output order.

State and boundaries

rank maps each distinct value to its one-based position in sorted unique values.

Protocol

1. Deduplicate values with a set.
2. Sort the distinct values.
3. Assign ranks beginning at one.
4. Look up each original value's rank.

Visualization

```
distinct sorted: 10 20 30 40
ranks:            1  2  3  4
40 10 20 30 ->    4  1  2  3
```

Python implementation

```
def rank_transform(nums):
    rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
    return [rank[value] for value in nums]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
rank_transform([40, 10, 20, 30])

L2 rank_transform: rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
nums=[40, 10, 20, 30]; i=0; value=10

L2 rank_transform: rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
nums=[40, 10, 20, 30]; i=1; value=20

L2 rank_transform: rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
nums=[40, 10, 20, 30]; i=2; value=30

L2 rank_transform: rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
nums=[40, 10, 20, 30]; i=3; value=40

L2 rank_transform: rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
nums=[40, 10, 20, 30]; rank={10:1, 20:2, 30:3, 40:4}

L3 rank_transform: return [rank[value] for value in nums]
nums=[40, 10, 20, 30]; rank={10:1, 20:2, 30:3, 40:4}; value=40

L3 rank_transform: return [rank[value] for value in nums]
nums=[40, 10, 20, 30]; rank={10:1, 20:2, 30:3, 40:4}; value=10

L3 rank_transform: return [rank[value] for value in nums]
nums=[40, 10, 20, 30]; rank={10:1, 20:2, 30:3, 40:4}; value=20

L3 rank_transform: return [rank[value] for value in nums]
nums=[40, 10, 20, 30]; rank={10:1, 20:2, 30:3, 40:4}; value=30

L3 rank_transform: return [rank[value] for value in nums]
nums=[40, 10, 20, 30]; rank={10:1, 20:2, 30:3, 40:4}
```

Result: [4, 1, 2, 3]. Ranks are returned in original order.

```
rank_transform([5, 5, 1])

L2 rank_transform: rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
nums=[5, 5, 1]; i=0; value=1

L2 rank_transform: rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
nums=[5, 5, 1]; i=1; value=5

L2 rank_transform: rank = {value: i + 1 for i, value in enumerate(sorted(set(nums)))}
nums=[5, 5, 1]; rank={1:1, 5:2}

L3 rank_transform: return [rank[value] for value in nums]
nums=[5, 5, 1]; rank={1:1, 5:2}; value=5

L3 rank_transform: return [rank[value] for value in nums]
nums=[5, 5, 1]; rank={1:1, 5:2}; value=1
```

```
L3 rank_transform: return [rank[value] for value in nums]
nums=[5, 5, 1]; rank={1:1, 5:2}
```

Result: [2, 2, 1]. Duplicates share a rank.

Why the solution is correct

Sorted distinct values establish a strict total order, and enumeration assigns consecutive increasing ranks. Equal inputs share a dictionary key, so they receive equal ranks. Looking up the original sequence preserves its position order.

Time and space

O(n+u log u) expected time and O(n+u) space including output, where u is the distinct count.

Common mistakes

Ranking the fully sorted list gives gaps or inconsistent duplicate ranks. Returning the sorted ranks loses the original arrangement.

Reasoning card

State: rank maps each distinct value to its one-based position in sorted unique values.

Invariant: Rank mapping is complete before transformation.

Decision to rehearse: Sort unique values, then map back to original order.

Finish: Look up each original value's rank

Transfer exercise

Use coordinate compression to store large sparse coordinates in an array, while preserving the original distances separately.

025. Merge Overlapping Intervals

Bridge | Module 03

Problem

Given closed intervals [start,end] with start<=end, return their union as sorted, pairwise disjoint closed intervals. Sharing an endpoint counts as overlap. Inputs may be unsorted; return fresh two-element lists without mutating the supplied intervals.

Examples

Example 1. The first two intervals overlap.

```
merge_intervals([[1, 3], [2, 6], [8, 10]])
=> [[1, 6], [8, 10]]
```

Example 2. The union of no intervals is empty.

```
merge_intervals([])
=> []
```

Example 3. Containment and endpoint contact both preserve one union interval.

```
merge_intervals([[1, 5], [2, 3], [5, 7]])
=> [[1, 7]]
```

Why this mechanism works

Sorting by start ensures that a new interval cannot reach backward past a completed gap. Keep the last output interval active. If the next start is within it, only its end may need extension. Otherwise a permanent gap has been found and a new interval begins.

State and boundaries

output is a sorted merged prefix. Its last interval remains extendable; earlier intervals are final.

Protocol

1. Sort by start.
2. Start an output interval when there is a gap.
3. Otherwise extend the active end with max.
4. Return the merged intervals.

Visualization

```
[1---3]
  [2------6]     [8--10]
=> [1------6]    [8--10]
```

Python implementation

```
def merge_intervals(intervals):
    output = []
    for start, end in sorted(intervals):
        if not output or start > output[-1][1]:
            output.append([start, end])
        else:
            output[-1][1] = max(output[-1][1], end)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
merge_intervals([[1, 3], [2, 6], [8, 10]])

L2 merge_intervals: output = []
intervals=[[1, 3], [2, 6], [8, 10]]; output=[]

L3 merge_intervals: for start, end in sorted(intervals):
intervals=[[1, 3], [2, 6], [8, 10]]; output=[]; start=1; end=3

L5 merge_intervals: output.append([start, end])
intervals=[[1, 3], [2, 6], [8, 10]]; output=[[1, 3]]; start=1; end=3

L3 merge_intervals: for start, end in sorted(intervals):
intervals=[[1, 3], [2, 6], [8, 10]]; output=[[1, 3]]; start=2; end=6

L7 merge_intervals: output[-1][1] = max(output[-1][1], end)
intervals=[[1, 3], [2, 6], [8, 10]]; output=[[1, 6]]; start=2; end=6

L3 merge_intervals: for start, end in sorted(intervals):
intervals=[[1, 3], [2, 6], [8, 10]]; output=[[1, 6]]; start=8; end=10

L5 merge_intervals: output.append([start, end])
intervals=[[1, 3], [2, 6], [8, 10]]; output=[[1, 6], [8, 10]]; start=8; end=10
```

Result: [[1, 6], [8, 10]]. The first two intervals overlap.

```
merge_intervals([])

L2 merge_intervals: output = []
intervals=[]; output=[]
```

Result: []. The union of no intervals is empty.

Why the solution is correct

With nondecreasing starts, a gap before the next interval can never be filled by a later interval starting even farther right. Overlap can therefore be resolved locally against the active interval. Extension preserves the union, and committing a gap preserves disjointness.

Time and space

O(n log n) time and O(n) storage for sorting and output.

Common mistakes

Replacing the end instead of taking max shrinks a containing interval. For closed intervals, equality at an endpoint must merge.

Reasoning card

State: output is a sorted merged prefix. Its last interval remains extendable; earlier intervals are final.

Invariant: Committed intervals are final and disjoint from the active interval.

Decision to rehearse: Sort by start; extend current interval or commit it.

Finish: Return the merged intervals

Transfer exercise

Change the contract to half-open intervals and decide whether touching intervals should be coalesced as an output representation.

026. Insert One Interval

Bridge | Module 03

Problem

Insert a closed interval into an already sorted list of pairwise disjoint closed intervals, merging any overlaps including shared endpoints. Return new interval lists, leaving both inputs unchanged. The new interval may lie before, after or across the existing intervals.

Examples

Example 1. The insertion extends the first interval only.

```
insert_interval([[1, 3], [6, 9]], [2, 5])
=> [[1, 5], [6, 9]]
```

Example 2. The new interval forms the whole output.

```
insert_interval([], [2, 4])
=> [[2, 4]]
```

Example 3. The insertion bridges all three intervals.

```
insert_interval([[1, 2], [4, 5], [7, 8]], [2, 7])
=> [[1, 8]]
```

Why this mechanism works

The sorted input has three phases relative to the new interval: entirely before, overlapping, and entirely after. Only the middle phase can enlarge the interval being inserted. Its growing end can cause more intervals to enter the overlap phase, so compare against the updated end each time.

State and boundaries

i is the next unread interval. start,end describe the accumulated insertion interval. output holds finalized intervals before it.

Protocol

1. Copy intervals ending before the new start.
2. Merge every next interval starting no later than the growing end.
3. Append the accumulated insertion interval.
4. Copy the remaining suffix.

Visualization

```
before | overlapping block | after
       [ new, growing end ]
commit only after overlap block ends
```

Python implementation

```
def insert_interval(intervals, new):
    start, end = new
    output = []
    i = 0
    while i < len(intervals) and intervals[i][1] < start:
        output.append(list(intervals[i]))
        i += 1
    while i < len(intervals) and intervals[i][0] <= end:
        start = min(start, intervals[i][0])
        end = max(end, intervals[i][1])
        i += 1
    output.append([start, end])
    output.extend(list(pair) for pair in intervals[i:])
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
insert_interval([[1, 3], [6, 9]], [2, 5])

L2 insert_interval: start, end = new
intervals=[[1, 3], [6, 9]]; new=[2, 5]; start=2; end=5

L3 insert_interval: output = []
intervals=[[1, 3], [6, 9]]; new=[2, 5]; start=2; end=5; output=[]

L4 insert_interval: i = 0
intervals=[[1, 3], [6, 9]]; new=[2, 5]; start=2; end=5; output=[]; i=0

L9 insert_interval: start = min(start, intervals[i][0])
intervals=[[1, 3], [6, 9]]; new=[2, 5]; start=1; end=5; output=[]; i=0

L11 insert_interval: i += 1
intervals=[[1, 3], [6, 9]]; new=[2, 5]; start=1; end=5; output=[]; i=1

L12 insert_interval: output.append([start, end])
intervals=[[1, 3], [6, 9]]; new=[2, 5]; start=1; end=5; output=[[1, 5]]; i=1

L13 <genexpr>: output.extend(list(pair) for pair in intervals[i:])
.0=list_iterator; pair=[6, 9]

L13 <genexpr>: output.extend(list(pair) for pair in intervals[i:])
.0=list_iterator; pair=[6, 9]

L13 insert_interval: output.extend(list(pair) for pair in intervals[i:])
intervals=[[1, 3], [6, 9]]; new=[2, 5]; start=1; end=5; output=[[1, 5], [6, 9]]; i=1
```

Result: [[1, 5], [6, 9]]. The insertion extends the first interval only.

```
insert_interval([], [2, 4])

L2 insert_interval: start, end = new
intervals=[]; new=[2, 4]; start=2; end=4

L3 insert_interval: output = []
intervals=[]; new=[2, 4]; start=2; end=4; output=[]

L4 insert_interval: i = 0
intervals=[]; new=[2, 4]; start=2; end=4; output=[]; i=0

L12 insert_interval: output.append([start, end])
intervals=[]; new=[2, 4]; start=2; end=4; output=[[2, 4]]; i=0

L13 <genexpr>: output.extend(list(pair) for pair in intervals[i:])
.0=list_iterator
```

Result: [[2, 4]]. The new interval forms the whole output.

Why the solution is correct

Intervals in the first phase cannot overlap the insertion. The middle phase preserves their union by extending endpoints and continues until a genuine gap appears. Sorted order ensures every remaining interval lies after that gap, so appending it is safe.

Time and space

O(n) time and O(n) output space; the input's existing order avoids another sort.

Common mistakes

Testing overlap only against the original new end can miss a chain of overlaps. Appending the new interval too early commits incomplete state.

Reasoning card

State: i is the next unread interval. start,end describe the accumulated insertion interval. output holds finalized intervals before it.

Invariant: Everything appended before the active interval is final.

Decision to rehearse: Process before, overlapping, and after regions.

Finish: Copy the remaining suffix

Transfer exercise

Explain why arbitrary unsorted inputs require sorting or a different data structure before this three-phase scan.

027. Intersect Two Interval Lists

Bridge | Module 03

Problem

Return nonempty intersections of two sorted lists of pairwise disjoint half-open intervals [start,end), with start<end. Output fresh two-element lists in order. Touching endpoints alone have zero length and are not emitted. This half-open convention resolves the compressed source example's endpoint ambiguity.

Examples

Example 1. Endpoint-only contact at five is excluded.

```
intersect_intervals([[0, 2], [5, 10]], [[1, 5], [8, 12]])
=> [[1, 2], [8, 10]]
```

Example 2. Half-open intervals do not share the endpoint two.

```
intersect_intervals([[1, 2]], [[2, 3]])
=> []
```

Example 3. One empty source prevents all intersections.

```
intersect_intervals([], [[1, 3]])
=> []
```

Why this mechanism works

The overlap starts at the later start and ends at the earlier end. It exists only when start<end. After considering the two current intervals, the one ending first cannot intersect any later interval from the other list; that is the frontier that can be discarded safely.

State and boundaries

i and j are current interval indices. All intersections involving an interval before either frontier have been emitted.

Protocol

1. Compute max of starts and min of ends.
2. Emit if the resulting interval has positive length.
3. Advance the interval that ends first.
4. Stop when either source is exhausted.

Visualization

```
A: [0,2)   [5,10)
B:   [1,5)    [8,12)
=> [1,2), [8,10); no [5,5)
```

Python implementation

```
def intersect_intervals(a, b):
    i = j = 0
    output = []
    while i < len(a) and j < len(b):
        start = max(a[i][0], b[j][0])
        end = min(a[i][1], b[j][1])
        if start < end:
            output.append([start, end])
        if a[i][1] <= b[j][1]:
            i += 1
        else:
            j += 1
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
intersect_intervals([[0, 2], [5, 10]], [[1, 5], [8, 12]])

L2 intersect_intervals: i = j = 0
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=0; j=0

L3 intersect_intervals: output = []
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=0; j=0; output=[]

L5 intersect_intervals: start = max(a[i][0], b[j][0])
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=0; j=0; output=[]; start=1

L6 intersect_intervals: end = min(a[i][1], b[j][1])
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=0; j=0; output=[]; start=1; end=2

L8 intersect_intervals: output.append([start, end])
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=0; j=0; output=[[1, 2]]; start=1; end=2

L10 intersect_intervals: i += 1
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=1; j=0; output=[[1, 2]]; start=1; end=2

L5 intersect_intervals: start = max(a[i][0], b[j][0])
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=1; j=0; output=[[1, 2]]; start=5; end=2

L6 intersect_intervals: end = min(a[i][1], b[j][1])
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=1; j=0; output=[[1, 2]]; start=5; end=5

... 1 intermediate state changes omitted ...

L5 intersect_intervals: start = max(a[i][0], b[j][0])
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=1; j=1; output=[[1, 2]]; start=8; end=5

L6 intersect_intervals: end = min(a[i][1], b[j][1])
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=1; j=1; output=[[1, 2]]; start=8; end=10
```

```
L8 intersect_intervals: output.append([start, end])
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=1; j=1; output=[[1, 2], [8, 10]]; start=8; end=10

L10 intersect_intervals: i += 1
a=[[0, 2], [5, 10]]; b=[[1, 5], [8, 12]]; i=2; j=1; output=[[1, 2], [8, 10]]; start=8; end=10
```

Result: [[1, 2], [8, 10]]. Endpoint-only contact at five is excluded.

```
intersect_intervals([[1, 2]], [[2, 3]])

L2 intersect_intervals: i = j = 0
a=[[1, 2]]; b=[[2, 3]]; i=0; j=0

L3 intersect_intervals: output = []
a=[[1, 2]]; b=[[2, 3]]; i=0; j=0; output=[]

L5 intersect_intervals: start = max(a[i][0], b[j][0])
a=[[1, 2]]; b=[[2, 3]]; i=0; j=0; output=[]; start=2

L6 intersect_intervals: end = min(a[i][1], b[j][1])
a=[[1, 2]]; b=[[2, 3]]; i=0; j=0; output=[]; start=2; end=2

L10 intersect_intervals: i += 1
a=[[1, 2]]; b=[[2, 3]]; i=1; j=0; output=[]; start=2; end=2
```

Result: []. Half-open intervals do not share the endpoint two.

Why the solution is correct

The endpoint formula describes exactly the values shared by the current intervals. Disjoint sorted inputs ensure that the earlier-ending interval cannot overlap a later opposing interval after its current partner ends. Every possible positive-length overlap is therefore considered once.

Time and space

O(n+m) time and O(n+m) worst-case output storage; O(1) additional state.

Common mistakes

Using <= in the overlap test emits zero-length contacts. Advancing by the smaller start rather than the smaller end can discard an interval that still has future overlaps.

Reasoning card

State: i and j are current interval indices. All intersections involving an interval before either frontier have been emitted.

Invariant: A discarded interval cannot intersect any future interval.

Decision to rehearse: Advance the interval that ends first.

Finish: Stop when either source is exhausted

Transfer exercise

Convert the solution to closed intervals and include singleton intersections such as [5,5].

028. Closest Pair to Target

Interview | Module 03

Problem

Return a pair of values from distinct positions whose sum has minimum absolute difference from target. Sort a copy; return values in ascending order. If several pairs are equally close, return the first encountered by the specified inward two-pointer scan, starting at the extremes. Return None for fewer than two values.

Examples

Example 1. Four plus ten is closest, with error one.

```
closest_pair([1, 4, 7, 10], 15)
=> (4, 10)
```

Example 2. Two distinct positions are required.

```
closest_pair([5], 10)
=> None
```

Example 3. An exact pair ends the scan.

```
closest_pair([1, 2, 4, 5], 7)
=> (2, 5)
```

Why this mechanism works

If the current sum is too small, any pair using the same left endpoint and a smaller right endpoint is no better. Discard that left endpoint. The symmetric argument discards the right endpoint when the sum is too large. Keep the best pair before discarding either candidate frontier.

State and boundaries

left and right are distinct candidate indices. best and error describe the best visited pair; strict replacement preserves the encounter tie rule.

Protocol

1. Sort a copy and handle fewer than two values.
2. Evaluate the extreme pair.
3. Replace best only on a smaller absolute error.
4. Move left for a small sum or right for a large sum.
5. Finish immediately on an exact sum.

Visualization

```
1 + 10 = 11 < 15 -> move left
4 + 10 = 14       -> best error 1
7 + 10 = 17 > 15 -> move right
```

Python implementation

```
def closest_pair(nums, target):
    values = sorted(nums)
    if len(values) < 2:
        return None
    left, right = 0, len(values) - 1
    best, error = None, float('inf')
    while left < right:
        total = values[left] + values[right]
        difference = abs(total - target)
        if difference < error:
            best = (values[left], values[right])
            error = difference
        if total == target:
            break
        if total < target:
            left += 1
        else:
            right -= 1
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
closest_pair([1, 4, 7, 10], 15)

L2 closest_pair: values = sorted(nums)
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]

L5 closest_pair: left, right = 0, len(values) - 1
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=0; right=3

L6 closest_pair: best, error = None, float('inf')
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=0; right=3; best=None; error=inf

L8 closest_pair: total = values[left] + values[right]
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=0; right=3; best=None; error=inf;
    total=11

L9 closest_pair: difference = abs(total - target)
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=0; right=3; best=None; error=inf;
    total=11; difference=4

L11 closest_pair: best = (values[left], values[right])
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=0; right=3; best=(1, 10); error=inf;
     total=11; difference=4

L12 closest_pair: error = difference
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=0; right=3; best=(1, 10); error=4;
    total=11; difference=4
```

```
L16 closest_pair: left += 1
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=1; right=3; best=(1, 10); error=4;
    total=11; difference=4

... 4 intermediate state changes omitted ...

L16 closest_pair: left += 1
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=2; right=3; best=(4, 10); error=1;
    total=14; difference=1

L8 closest_pair: total = values[left] + values[right]
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=2; right=3; best=(4, 10); error=1;
    total=17; difference=1

L9 closest_pair: difference = abs(total - target)
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=2; right=3; best=(4, 10); error=1;
    total=17; difference=2

L18 closest_pair: right -= 1
nums=[1, 4, 7, 10]; target=15; values=[1, 4, 7, 10]; left=2; right=2; best=(4, 10); error=1;
    total=17; difference=2
```

Result: (4, 10). Four plus ten is closest, with error one.

```
closest_pair([5], 10)

L2 closest_pair: values = sorted(nums)
nums=[5]; target=10; values=[5]
```

Result: None. Two distinct positions are required.

Why the solution is correct

Each pointer move discards only pairs no better than the pair just evaluated on that frontier. Thus an optimal error is never lost without being represented by an equally good visited candidate. Exact error zero is globally optimal.

Time and space

O(n log n) time including sorting and O(n) space for the sorted copy.

Common mistakes

For [1,4,7,10] and target 15, (4,10) differs by one; (7,10) differs by two and is not an equally good answer. Tie behavior must be specified separately from optimal error.

Reasoning card

State: left and right are distinct candidate indices. best and error describe the best visited pair; strict replacement preserves the encounter tie rule.

Invariant: Discarded frontier pairs cannot improve in the required direction.

Decision to rehearse: Use order to decide which frontier must move.

Finish: Finish immediately on an exact sum

Transfer exercise

Require the lexicographically smallest optimal pair instead. Determine whether the existing early-exit and tie rules still suffice.

029. Minimum-Difference Pairs

Interview | Module 03

Problem

Return all adjacent pairs in a sorted copy whose difference equals the minimum adjacent difference. Inputs are integers and may contain duplicates; repeated adjacent pairs are retained. Output tuples in sorted encounter order. With fewer than two values, return [].

Examples

Example 1. All adjacent gaps tie at one.

```
minimum_difference_pairs([4, 2, 1, 3])
=> [(1, 2), (2, 3), (3, 4)]
```

Example 2. No adjacent edge exists.

```
minimum_difference_pairs([1])
=> []
```

Example 3. Two distinct adjacent edges have zero difference.

```
minimum_difference_pairs([2, 2, 2])
=> [(2, 2), (2, 2)]
```

Why this mechanism works

For sorted values a<=b<=c, the nonadjacent difference c-a is at least either intervening difference. Consequently a globally minimum pair difference appears between neighbors. One pass over those neighbor edges can reset the output when a better gap is found or append when the gap ties.

State and boundaries

gap is the smallest edge difference examined. output contains every examined adjacent pair with that difference.

Protocol

1. Sort the values.
2. Examine adjacent pairs.
3. On a smaller gap replace gap and clear old answers.
4. On an equal gap append the pair.
5. Return output.

Visualization

```
sorted: 1--2--3--4
gaps:     1  1  1
all three adjacent pairs qualify
```

Python implementation

```
def minimum_difference_pairs(nums):
    values = sorted(nums)
    gap = float('inf')
    output = []
    for i in range(1, len(values)):
        difference = values[i] - values[i - 1]
        if difference < gap:
            gap = difference
            output = []
        if difference == gap:
            output.append((values[i - 1], values[i]))
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
minimum_difference_pairs([4, 2, 1, 3])

L2 minimum_difference_pairs: values = sorted(nums)
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]

L3 minimum_difference_pairs: gap = float('inf')
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=inf

L4 minimum_difference_pairs: output = []
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=inf; output=[]

L5 minimum_difference_pairs: for i in range(1, len(values)):
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=inf; output=[]; i=1

L6 minimum_difference_pairs: difference = values[i] - values[i - 1]
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=inf; output=[]; i=1; difference=1

L8 minimum_difference_pairs: gap = difference
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=1; output=[]; i=1; difference=1

L11 minimum_difference_pairs: output.append((values[i - 1], values[i]))
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=1; output=[(1, 2)]; i=1; difference=1

L5 minimum_difference_pairs: for i in range(1, len(values)):
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=1; output=[(1, 2)]; i=2; difference=1

L11 minimum_difference_pairs: output.append((values[i - 1], values[i]))
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=1; output=[(1, 2), (2, 3)]; i=2; difference=1

L5 minimum_difference_pairs: for i in range(1, len(values)):
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=1; output=[(1, 2), (2, 3)]; i=3; difference=1
```

```
L11 minimum_difference_pairs: output.append((values[i - 1], values[i]))
nums=[4, 2, 1, 3]; values=[1, 2, 3, 4]; gap=1; output=[(1, 2), (2, 3), (3, 4)]; i=3;
    difference=1
```

Result: [(1, 2), (2, 3), (3, 4)]. All adjacent gaps tie at one.

```
minimum_difference_pairs([1])

L2 minimum_difference_pairs: values = sorted(nums)
nums=[1]; values=[1]

L3 minimum_difference_pairs: gap = float('inf')
nums=[1]; values=[1]; gap=inf

L4 minimum_difference_pairs: output = []
nums=[1]; values=[1]; gap=inf; output=[]
```

Result: []. No adjacent edge exists.

Why the solution is correct

A smaller observed gap invalidates all previous larger-gap answers, while equality adds another valid answer. The neighbor argument ensures that no nonadjacent pair has a smaller difference. The requested adjacent pairs are exactly those retained.

Time and space

O(n log n) time and O(n) space for sorting and output.

Common mistakes

Failing to clear on improvement retains nonminimal pairs. Duplicate values make zero a legitimate minimum difference.

Reasoning card

State: gap is the smallest edge difference examined. output contains every examined adjacent pair with that difference.

Invariant: The best gap seen covers all processed adjacent pairs.

Decision to rehearse: After sorting, only neighbours can realize the minimum gap.

Finish: Return output

Transfer exercise

If the requested output is unique value pairs rather than adjacent occurrences, state how duplicate emission changes.

030. Sort Records by Multiple Keys

Interview | Module 03

Problem

Given records represented as (name, score, timestamp) tuples, return a new list sorted by ascending integer score and descending integer timestamp within equal scores. For ties on both keys, preserve original input order. Names are payload and do not affect ordering.

Examples

Example 1. Score dominates; timestamp resolves the score tie.

```
sort_records([("A", 2, 10), ("B", 1, 5), ("C", 2, 20)])
=> [('B', 1, 5), ('C', 2, 20), ('A', 2, 10)]
```

Example 2. The empty list remains empty.

```
sort_records([])
=> []
```

Example 3. Names do not break complete-key ties.

```
sort_records([("Z", 1, 2), ("A", 1, 2)])
=> [('Z', 1, 2), ('A', 1, 2)]
```

Why this mechanism works

A tuple key expresses priority: Python compares the score first, then the negated timestamp. Negation converts descending numeric order into ascending key order. Python's stable sort handles complete-key ties without adding an unintended name-based tie breaker.

State and boundaries

Each record has key (score,-timestamp). Stability preserves relative order for equal keys.

Protocol

1. Define the two-field key.

2. Sort records by that key.

3. Retain the original records as output payload.

Visualization

```
record (A,2,10) -> key (2,-10)
record (B,1, 5) -> key (1, -5)
record (C,2,20) -> key (2,-20)
order: B, C, A
```

Python implementation

```
def sort_records(records):
    return sorted(records, key=lambda record: (record[1], -record[2]))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
sort_records([("A", 2, 10), ("B", 1, 5), ("C", 2, 20)])

L2 <lambda>: return sorted(records, key=lambda record: (record[1], -record[2]))
record=('A', 2, 10)

L2 <lambda>: return sorted(records, key=lambda record: (record[1], -record[2]))
record=('B', 1, 5)

L2 <lambda>: return sorted(records, key=lambda record: (record[1], -record[2]))
record=('C', 2, 20)

L2 sort_records: return sorted(records, key=lambda record: (record[1], -record[2]))
records=[('A', 2, 10), ('B', 1, 5), ('C', 2, 20)]
```

Result: [('B', 1, 5), ('C', 2, 20), ('A', 2, 10)]. Score dominates; timestamp resolves the score tie.

```
sort_records([])

L2 sort_records: return sorted(records, key=lambda record: (record[1], -record[2]))
records=[]
```

Result: []. The empty list remains empty.

Why the solution is correct

Lexicographic key comparison orders smaller scores first. Equal scores compare negative timestamps, placing larger original timestamps first. Stability leaves complete-key ties in input order, satisfying all three ordering requirements.

Time and space

O(n log n) time and O(n) additional storage for the sorted result and sorting workspace.

Common mistakes

Setting reverse=True reverses the primary score too. Including name in the key silently changes the complete-tie rule.

Reasoning card

State: Each record has key (score,-timestamp). Stability preserves relative order for equal keys.

Invariant: The sort key totally determines required ordering.

Decision to rehearse: Build a complete key whose tuple order matches the specification.

Finish: Retain the original records as output payload

Transfer exercise

Sort timestamps represented by comparable objects that cannot be negated. Use stable sorting passes in the correct order.

Module 04
Two Pointers and In-Place Read/Write State

Use two coordinated frontiers and prove why each pointer movement is safe.

Two boundaries now cooperate within one structure. Sometimes they converge around candidates; sometimes one reads and another writes a compacted prefix. Do not treat these as one interchangeable template. Name the region each pointer owns and prove why moving it cannot destroy an answer or unread input.

The ten-step progression

031. Reverse an Array In Place

032. Normalized Palindrome

033. Move Zeros Stably

034. Remove a Target In Place

035. Two Sum in Sorted Array

036. Sorted Squares

037. Three Sum to Zero

038. Container With Most Water

039. Trapping Rain Water

040. Partition Evens Before Odds

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

031. Reverse an Array In Place

Foundation | Module 04

Problem

Reverse a mutable list in place and return the same list object. Preserve all occurrences. Empty and singleton lists are valid; no second array may be allocated.

Examples

Example 1. Two swaps suffice; the middle stays in place.

```
reverse_array([1, 2, 3, 4, 5])
=> [5, 4, 3, 2, 1]
```

Example 2. There are no unresolved pairs.

```
reverse_array([])
=> []
```

Example 3. The returned object is the supplied list.

```
a = [1, 2]
(reverse_array(a) is a, a)
=> (True, [2, 1])
```

Why this mechanism works

The first and last remaining values have uniquely determined destinations in a reversal. Exchange them, then solve the smaller interior. Values outside the two frontiers are already final and never need to be visited again.

State and boundaries

left and right are inclusive indices of the unresolved region. Positions outside them are in final reversed order.

Protocol

1. Place pointers at both ends.
2. While left<right swap the endpoint values.
3. Advance left and retreat right.
4. Return the list.

Visualization

```
[1,2,3,4,5] -> [5,2,3,4,1]
   L   R    -> [5,4,3,2,1]
                   center stays
```

Python implementation

```
def reverse_array(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1
    return nums
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
reverse_array([1, 2, 3, 4, 5])

L2 reverse_array: left, right = 0, len(nums) - 1
nums=[1, 2, 3, 4, 5]; left=0; right=4

L4 reverse_array: nums[left], nums[right] = nums[right], nums[left]
nums=[5, 2, 3, 4, 1]; left=0; right=4

L5 reverse_array: left += 1
nums=[5, 2, 3, 4, 1]; left=1; right=4

L6 reverse_array: right -= 1
nums=[5, 2, 3, 4, 1]; left=1; right=3

L4 reverse_array: nums[left], nums[right] = nums[right], nums[left]
nums=[5, 4, 3, 2, 1]; left=1; right=3

L5 reverse_array: left += 1
nums=[5, 4, 3, 2, 1]; left=2; right=3

L6 reverse_array: right -= 1
nums=[5, 4, 3, 2, 1]; left=2; right=2
```

Result: [5, 4, 3, 2, 1]. Two swaps suffice; the middle stays in place.

```
reverse_array([])

L2 reverse_array: left, right = 0, len(nums) - 1
nums=[]; left=0; right=-1
```

Result: []. There are no unresolved pairs.

Why the solution is correct

Each exchange places two original values at their required mirrored positions. Shrinking the region protects these final values. When at most one position remains, it is either absent or already its own mirror.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Using nums[::-1] allocates a new list. Running the pointers through each other swaps completed pairs back.

Reasoning card

State: left and right are inclusive indices of the unresolved region. Positions outside them are in final reversed order.

Invariant: Everything outside [left,right] is already final.

Decision to rehearse: Swap first, then move both boundaries inward.

Finish: Return the list

Transfer exercise

Reverse only a specified half-open subrange [start,end), leaving everything else unchanged.

032. Normalized Palindrome

Foundation | Module 04

Problem

For an ASCII string, ignore non-alphanumeric characters and letter case, then decide whether the remaining sequence is a palindrome. Return True when no alphanumeric characters remain. Do not allocate a filtered copy.

Examples

Example 1. Ignoring spaces, punctuation and case leaves matching endpoints.

```
normalized_palindrome("A man, a plan, a canal: Panama")
=> True
```

Example 2. The meaningful sequence is empty.

```
normalized_palindrome("!!!")
=> True
```

Example 3. A meaningful endpoint pair eventually differs.

```
normalized_palindrome("race a car")
=> False
```

Why this mechanism works

Opposite ends can be compared after irrelevant characters are skipped independently. A mismatch at the next meaningful pair disproves a palindrome immediately. The skip loops need bounds checks because the entire remaining region may be punctuation.

State and boundaries

left and right delimit unexamined characters. All meaningful pairs outside them have already matched.

Protocol

1. Start at both ends.
2. Skip non-alphanumeric characters on the left.
3. Skip them on the right.
4. Compare lowercase meaningful characters.
5. Reject a mismatch or move both pointers inward.

Visualization

```
A man, a plan, a canal: Panama
A <----------------------> a
next meaningful pair: m <-> m
```

Python implementation

```
def normalized_palindrome(text):
    left, right = 0, len(text) - 1
    while left < right:
        while left < right and not text[left].isalnum():
            left += 1
        while left < right and not text[right].isalnum():
            right -= 1
        if text[left].lower() != text[right].lower():
            return False
        left += 1
        right -= 1
    return True
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
normalized_palindrome("A man, a plan, a canal: Panama")

L2 normalized_palindrome: left, right = 0, len(text) - 1
text='A man, a plan, a canal: Panama'; left=0; right=29

L10 normalized_palindrome: left += 1
text='A man, a plan, a canal: Panama'; left=1; right=29

L11 normalized_palindrome: right -= 1
text='A man, a plan, a canal: Panama'; left=1; right=28

L5 normalized_palindrome: left += 1
text='A man, a plan, a canal: Panama'; left=2; right=28

L10 normalized_palindrome: left += 1
text='A man, a plan, a canal: Panama'; left=3; right=28

L11 normalized_palindrome: right -= 1
text='A man, a plan, a canal: Panama'; left=3; right=27

L10 normalized_palindrome: left += 1
text='A man, a plan, a canal: Panama'; left=4; right=27

L11 normalized_palindrome: right -= 1
text='A man, a plan, a canal: Panama'; left=4; right=26

... 20 intermediate state changes omitted ...

L11 normalized_palindrome: right -= 1
text='A man, a plan, a canal: Panama'; left=16; right=17

L5 normalized_palindrome: left += 1
text='A man, a plan, a canal: Panama'; left=17; right=17

L10 normalized_palindrome: left += 1
```

```
text='A man, a plan, a canal: Panama'; left=18; right=17

L11 normalized_palindrome: right -= 1
text='A man, a plan, a canal: Panama'; left=18; right=16
```

Result: True. Ignoring spaces, punctuation and case leaves matching endpoints.

```
normalized_palindrome("!!!")

L2 normalized_palindrome: left, right = 0, len(text) - 1
text='!!!'; left=0; right=2

L5 normalized_palindrome: left += 1
text='!!!'; left=1; right=2

L5 normalized_palindrome: left += 1
text='!!!'; left=2; right=2

L10 normalized_palindrome: left += 1
text='!!!'; left=3; right=2

L11 normalized_palindrome: right -= 1
text='!!!'; left=3; right=1
```

Result: True. The meaningful sequence is empty.

Why the solution is correct

Skipping removes only characters the contract ignores. Matching the next meaningful endpoints is necessary for any palindrome, and after a match the interior is the remaining subproblem. Exhausting it proves all required pairs matched.

Time and space

O(n) time and O(1) auxiliary space for ASCII input.

Common mistakes

Skipping with if instead of while fails on several punctuation marks. Unbounded skipping can move a pointer outside the string.

Reasoning card

State: left and right delimit unexamined characters. All meaningful pairs outside them have already matched.

Invariant: Pointers always refer to the next visible characters.

Decision to rehearse: Repeatedly skip invalid positions before comparing.

Finish: Reject a mismatch or move both pointers inward

Transfer exercise

Permit deletion of at most one meaningful character. At the first mismatch, which two subproblems must be considered?

033. Move Zeros Stably

Foundation | Module 04

Problem

Move all zeros to the end of a mutable integer list, preserving the relative order of nonzero values. Return the same list, with its original length. Negative numbers are nonzero and must be retained in order.

Examples

Example 1. Nonzero order is unchanged.

```
move_zeros([0, 1, 0, 3, 12])
=> [1, 3, 12, 0, 0]
```

Example 2. Both phases do no work.

```
move_zeros([])
=> []
```

Example 3. Negative values are ordinary retained data.

```
move_zeros([0, -1, 0, -2])
=> [-1, -2, 0, 0]
```

Why this mechanism works

First compact accepted values into a prefix. Because write never exceeds the read position, each write is safe. After all original values have been read, every remaining position can be filled with zero. The second phase avoids swapping zeros repeatedly through the list.

State and boundaries

write is the count of nonzero values consumed; nums[:write] is their stable sequence.

Protocol

1. Scan original values.
2. Write each nonzero at write and advance write.
3. After the scan fill the remaining suffix with zeros.
4. Return nums.

Visualization

```
[0,1,0,3,12]
compact: [1,3,12 | ?,?]
fill:    [1,3,12 | 0,0]
```

Python implementation

```
def move_zeros(nums):
    write = 0
    for value in nums:
        if value != 0:
            nums[write] = value
            write += 1
    for i in range(write, len(nums)):
        nums[i] = 0
    return nums
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
move_zeros([0, 1, 0, 3, 12])

L2 move_zeros: write = 0
nums=[0, 1, 0, 3, 12]; write=0

L3 move_zeros: for value in nums:
nums=[0, 1, 0, 3, 12]; write=0; value=0

L3 move_zeros: for value in nums:
nums=[0, 1, 0, 3, 12]; write=0; value=1

L5 move_zeros: nums[write] = value
nums=[1, 1, 0, 3, 12]; write=0; value=1

L6 move_zeros: write += 1
nums=[1, 1, 0, 3, 12]; write=1; value=1

L3 move_zeros: for value in nums:
nums=[1, 1, 0, 3, 12]; write=1; value=0

L3 move_zeros: for value in nums:
nums=[1, 1, 0, 3, 12]; write=1; value=3

L5 move_zeros: nums[write] = value
nums=[1, 3, 0, 3, 12]; write=1; value=3

... 4 intermediate state changes omitted ...

L7 move_zeros: for i in range(write, len(nums)):
nums=[1, 3, 12, 3, 12]; write=3; value=12; i=3

L8 move_zeros: nums[i] = 0
nums=[1, 3, 12, 0, 12]; write=3; value=12; i=3

L7 move_zeros: for i in range(write, len(nums)):
nums=[1, 3, 12, 0, 12]; write=3; value=12; i=4
```

```
L8 move_zeros: nums[i] = 0
nums=[1, 3, 12, 0, 0]; write=3; value=12; i=4
```

Result: [1, 3, 12, 0, 0]. Nonzero order is unchanged.

```
move_zeros([])

L2 move_zeros: write = 0
nums=[]; write=0
```

Result: []. Both phases do no work.

Why the solution is correct

The first phase appends nonzero values in encounter order without overwriting unread data. The number of unwritten positions equals the number of zeros. Filling exactly that suffix preserves length and multiplicity.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Swapping arbitrary nonzero endpoints can break stability. Filling zeros during the read scan can overwrite values not yet examined.

Reasoning card

State: write is the count of nonzero values consumed; nums[:write] is their stable sequence.

Invariant: The prefix before write is the final compacted output.

Decision to rehearse: Separate read position from next write position.

Finish: Return nums

Transfer exercise

Generalize zero to a discard predicate and state whether the discarded suffix must preserve order too.

034. Remove a Target In Place

Foundation | Module 04

Problem

Remove every occurrence of target from a mutable integer list stably. Return the number k of retained values; only nums[:k] is meaningful afterward. The physical list length and tail values are unspecified.

Examples

Example 1. Only the first two cells are meaningful.

```
a = [3, 2, 2, 3]
(remove_target(a, 3), a[:2])
=> (2, [2, 2])
```

Example 2. No values survive because none exist.

```
remove_target([], 3)
=> 0
```

Example 3. Every occurrence is rejected.

```
remove_target([1, 1], 1)
=> 0
```

Why this mechanism works

A read frontier examines all original values, while a write frontier advances only for accepted ones. This is stable filtering without allocation. It is the same compaction mechanism as deduplication, but acceptance now depends on a fixed predicate rather than the last committed value.

State and boundaries

write counts accepted values. The prefix before write contains exactly the consumed non-target values in order.

Protocol

1. Set write to zero.
2. Read each value.
3. If it differs from target, commit it at write.
4. Advance write only on a commit.
5. Return write.

Visualization

```
target=3
read:  [3,2,2,3]
keep:  [2,2 | ?,?]
k=2
```

Python implementation

```
def remove_target(nums, target):
    write = 0
    for value in nums:
        if value != target:
            nums[write] = value
            write += 1
    return write
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
(remove_target(a, 3), a[:2])

L2 remove_target: write = 0
nums=[3, 2, 2, 3]; target=3; write=0

L3 remove_target: for value in nums:
nums=[3, 2, 2, 3]; target=3; write=0; value=3

L3 remove_target: for value in nums:
nums=[3, 2, 2, 3]; target=3; write=0; value=2

L5 remove_target: nums[write] = value
nums=[2, 2, 2, 3]; target=3; write=0; value=2

L6 remove_target: write += 1
nums=[2, 2, 2, 3]; target=3; write=1; value=2

L6 remove_target: write += 1
nums=[2, 2, 2, 3]; target=3; write=2; value=2

L3 remove_target: for value in nums:
nums=[2, 2, 2, 3]; target=3; write=2; value=3
```

Result: (2, [2, 2]). Only the first two cells are meaningful.

```
remove_target([], 3)

L2 remove_target: write = 0
nums=[]; target=3; write=0
```

Result: 0. No values survive because none exist.

Why the solution is correct

Each accepted value is written once after all previously accepted values, preserving order. Rejected values contribute nothing. Because write never overtakes reading, the unread suffix stays intact, and the final prefix is exactly the stable filter.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Deleting cells while iterating shifts unread indices. Returning len(nums) ignores the logical output boundary.

Reasoning card

State: write counts accepted values. The prefix before write contains exactly the consumed non-target values in order.

Invariant: write is the next output slot.

Decision to rehearse: Write only values that survive.

Finish: Return write

Transfer exercise

Drop the stability requirement and remove by swapping with the final unread position. Compare write counts.

035. Two Sum in Sorted Array

Bridge | Module 04

Problem

Given a nondecreasing integer list, return one-based indices of two distinct positions summing to target, or [] if none exists. Return the first matching pair found by an inward scan starting at the endpoints. Leave the input unchanged.

Examples

Example 1. The answer uses one-based indices.

```
two_sum_sorted([2, 7, 11, 15], 9)
=> [1, 2]
```

Example 2. A single cell cannot be used twice.

```
two_sum_sorted([3], 6)
=> []
```

Example 3. Negative values do not invalidate sorted-order reasoning.

```
two_sum_sorted([-2, 0, 2, 3], 1)
=> [1, 4]
```

Why this mechanism works

Sorted order makes a too-small sum informative: with the same left endpoint, every smaller right endpoint gives an even smaller sum. Discard left. A too-large sum similarly rules out the current right endpoint. No hash map is needed because ordering provides the exclusion proof.

State and boundaries

left and right are inclusive candidate endpoints, with left<right. Any still-possible pair lies inside this region.

Protocol

1. Evaluate the endpoint sum.
2. Return indices plus one on equality.
3. Advance left if the sum is small.
4. Retreat right if it is large.
5. Return [] when fewer than two candidates remain.

Visualization

```
2 ........ 15 -> sum 17 > 9
2 ..... 11    -> sum 13 > 9
2 .. 7        -> sum  9
```

Python implementation

```
def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        total = nums[left] + nums[right]
        if total == target:
            return [left + 1, right + 1]
        if total < target:
            left += 1
        else:
            right -= 1
    return []
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
two_sum_sorted([2, 7, 11, 15], 9)

L2 two_sum_sorted: left, right = 0, len(nums) - 1
nums=[2, 7, 11, 15]; target=9; left=0; right=3

L4 two_sum_sorted: total = nums[left] + nums[right]
nums=[2, 7, 11, 15]; target=9; left=0; right=3; total=17

L10 two_sum_sorted: right -= 1
nums=[2, 7, 11, 15]; target=9; left=0; right=2; total=17

L4 two_sum_sorted: total = nums[left] + nums[right]
nums=[2, 7, 11, 15]; target=9; left=0; right=2; total=13

L10 two_sum_sorted: right -= 1
nums=[2, 7, 11, 15]; target=9; left=0; right=1; total=13

L4 two_sum_sorted: total = nums[left] + nums[right]
nums=[2, 7, 11, 15]; target=9; left=0; right=1; total=9
```

Result: [1, 2]. The answer uses one-based indices.

```
two_sum_sorted([3], 6)

L2 two_sum_sorted: left, right = 0, len(nums) - 1
nums=[3]; target=6; left=0; right=0
```

Result: []. A single cell cannot be used twice.

Why the solution is correct

Each movement removes only a frontier that cannot participate in a target-sum pair among the remaining values. Equality supplies two distinct valid positions. If the pointers meet, every possible pair has been ruled out.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Returning zero-based indices violates this kata's explicit convention. Moving the larger-valued endpoint on a too-small sum moves in the wrong direction.

Reasoning card

State: left and right are inclusive candidate endpoints, with left<right. Any still-possible pair lies inside this region.

Invariant: All discarded pairs cannot equal target.

Decision to rehearse: Move left when sum is too small and right when too large.

Finish: Return [] when fewer than two candidates remain

Transfer exercise

Count all pairs when duplicates exist; explain why one match may represent a product of two run lengths.

036. Sorted Squares

Bridge | Module 04

Problem

Given a nondecreasing integer list, return a new nondecreasing list of squared values in O(n) time. Negative values and duplicates are allowed. Do not mutate the input.

Examples

Example 1. The largest magnitude is consumed first.

```
sorted_squares([-4, -1, 0, 3, 10])
=> [0, 1, 9, 16, 100]
```

Example 2. No output index is visited.

```
sorted_squares([])
=> []
```

Example 3. An all-negative input is handled by the same frontiers.

```
sorted_squares([-3, -2, -1])
=> [1, 4, 9]
```

Why this mechanism works

Squaring is not monotone across zero, but the largest absolute value of a sorted interval occurs at an endpoint. Compare the two endpoint squares and fill the output from right to left. This turns two monotone magnitude frontiers into a merge in descending consumption order.

State and boundaries

left and right bound unread input. write is the next output index from the right; positions after it are final.

Protocol

1. Allocate an output of the same length.
2. Compare endpoint squares.
3. Write the larger square at write.
4. Consume its source endpoint.
5. Move write left until output is complete.

Visualization

```
input: [-4,-1,0,3,10]
choose max square: 100,16,9,1,0
write from right: [0,1,9,16,100]
```

Python implementation

```
def sorted_squares(nums):
    output = [0] * len(nums)
    left, right = 0, len(nums) - 1
    for write in range(len(nums) - 1, -1, -1):
        if nums[left] ** 2 > nums[right] ** 2:
            output[write] = nums[left] ** 2
            left += 1
        else:
            output[write] = nums[right] ** 2
            right -= 1
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
sorted_squares([-4, -1, 0, 3, 10])

L2 sorted_squares: output = [0] * len(nums)
nums=[-4, -1, 0, 3, 10]; output=[0, 0, 0, 0, 0]

L3 sorted_squares: left, right = 0, len(nums) - 1
nums=[-4, -1, 0, 3, 10]; output=[0, 0, 0, 0, 0]; left=0; right=4

L4 sorted_squares: for write in range(len(nums) - 1, -1, -1):
nums=[-4, -1, 0, 3, 10]; output=[0, 0, 0, 0, 0]; left=0; right=4; write=4

L9 sorted_squares: output[write] = nums[right] ** 2
nums=[-4, -1, 0, 3, 10]; output=[0, 0, 0, 0, 100]; left=0; right=4; write=4

L10 sorted_squares: right -= 1
nums=[-4, -1, 0, 3, 10]; output=[0, 0, 0, 0, 100]; left=0; right=3; write=4

L4 sorted_squares: for write in range(len(nums) - 1, -1, -1):
nums=[-4, -1, 0, 3, 10]; output=[0, 0, 0, 0, 100]; left=0; right=3; write=3

L6 sorted_squares: output[write] = nums[left] ** 2
nums=[-4, -1, 0, 3, 10]; output=[0, 0, 0, 16, 100]; left=0; right=3; write=3

L7 sorted_squares: left += 1
nums=[-4, -1, 0, 3, 10]; output=[0, 0, 0, 16, 100]; left=1; right=3; write=3

... 4 intermediate state changes omitted ...

L6 sorted_squares: output[write] = nums[left] ** 2
nums=[-4, -1, 0, 3, 10]; output=[0, 1, 9, 16, 100]; left=1; right=2; write=1

L7 sorted_squares: left += 1
nums=[-4, -1, 0, 3, 10]; output=[0, 1, 9, 16, 100]; left=2; right=2; write=1

L4 sorted_squares: for write in range(len(nums) - 1, -1, -1):
nums=[-4, -1, 0, 3, 10]; output=[0, 1, 9, 16, 100]; left=2; right=2; write=0
```

```
L10 sorted_squares: right -= 1
nums=[-4, -1, 0, 3, 10]; output=[0, 1, 9, 16, 100]; left=2; right=1; write=0
```

Result: [0, 1, 9, 16, 100]. The largest magnitude is consumed first.

```
sorted_squares([])

L2 sorted_squares: output = [0] * len(nums)
nums=[]; output=[]

L3 sorted_squares: left, right = 0, len(nums) - 1
nums=[]; output=[]; left=0; right=-1
```

Result: []. No output index is visited.

Why the solution is correct

The selected endpoint square is maximal among all unread values, so it belongs at the greatest unfilled output index. Consuming it preserves that claim for the smaller interval. Filling every index gives all squares in nondecreasing order.

Time and space

O(n) time and O(n) returned space; O(1) other state.

Common mistakes

Squaring in place without reordering leaves large negative squares at the front. Filling from the left would require the smallest magnitude, which is not generally at an endpoint.

Reasoning card

State: left and right bound unread input. write is the next output index from the right; positions after it are final.

Invariant: The output suffix contains the largest squares in final order.

Decision to rehearse: Largest magnitude must be at one of the ends.

Finish: Move write left until output is complete

Transfer exercise

Merge squared values outward from the sign boundary instead; compare which direction each source must move.

037. Three Sum to Zero

Bridge | Module 04

Problem

Return every unique value triple summing to zero, using three distinct input positions. Sort values within each triple and return triples in lexicographic order. Duplicate input values may be used when enough occurrences exist. Leave the input unchanged.

Examples

Example 1. Two different value triples exist.

```
three_sum([-1, 0, 1, 2, -1, -4])
=> [[-1, -1, 2], [-1, 0, 1]]
```

Example 2. Many index triples represent one value triple.

```
three_sum([0, 0, 0, 0])
=> [[0, 0, 0]]
```

Example 3. Three positions are required.

```
three_sum([1, 2])
=> []
```

Why this mechanism works

Fix the smallest value of a triple in a sorted copy, then solve a two-sum problem in the suffix. Skipping repeated fixed values prevents repeating entire searches. After a match, skip equal values on both moving frontiers so the same value triple is not emitted again.

State and boundaries

i selects the first value; left and right search indices strictly after i. output contains distinct triples from completed searches.

Protocol

1. Sort a copy.
2. Skip duplicate fixed values.
3. Run inward two pointers for target -values[i].
4. On a match append the triple and advance past equal endpoint runs.
5. Move one frontier according to the sum otherwise.

Visualization

```
fixed -1 | suffix [-1,0,1,2]
           -1 + 2 -> total 0
            0 + 1 -> total 0
```

Python implementation

```
def three_sum(nums):
    values = sorted(nums)
    output = []
    for i in range(len(values) - 2):
        if i and values[i] == values[i - 1]:
            continue
        left, right = i + 1, len(values) - 1
        while left < right:
            total = values[i] + values[left] + values[right]
            if total == 0:
                output.append([values[i], values[left], values[right]])
                left += 1
                right -= 1
                while left < right and values[left] == values[left - 1]:
                    left += 1
                while left < right and values[right] == values[right + 1]:
                    right -= 1
            elif total < 0:
                left += 1
            else:
                right -= 1
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
three_sum([-1, 0, 1, 2, -1, -4])

L2 three_sum: values = sorted(nums)
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]

L3 three_sum: output = []
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[]

L4 three_sum: for i in range(len(values) - 2):
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[]; i=0

L7 three_sum: left, right = i + 1, len(values) - 1
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[]; i=0; left=1; right=5

L9 three_sum: total = values[i] + values[left] + values[right]
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[]; i=0; left=1; right=5;
    total=-3

L19 three_sum: left += 1
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[]; i=0; left=2; right=5;
    total=-3

L19 three_sum: left += 1
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[]; i=0; left=3; right=5;
```

```
    total=-3

L9 three_sum: total = values[i] + values[left] + values[right]
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[]; i=0; left=3; right=5;
    total=-2

... 13 intermediate state changes omitted ...

L4 three_sum: for i in range(len(values) - 2):
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[[-1, -1, 2], [-1, 0, 1]];
    i=3; left=4; right=3; total=0

L7 three_sum: left, right = i + 1, len(values) - 1
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[[-1, -1, 2], [-1, 0, 1]];
    i=3; left=4; right=5; total=0

L9 three_sum: total = values[i] + values[left] + values[right]
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[[-1, -1, 2], [-1, 0, 1]];
    i=3; left=4; right=5; total=3

L21 three_sum: right -= 1
nums=[-1, 0, 1, 2, -1, -4]; values=[-4, -1, -1, 0, 1, 2]; output=[[-1, -1, 2], [-1, 0, 1]];
    i=3; left=4; right=4; total=3
```

Result: [[-1, -1, 2], [-1, 0, 1]]. Two different value triples exist.

`three_sum([0, 0, 0, 0])`

```
L2 three_sum: values = sorted(nums)
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]

L3 three_sum: output = []
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]; output=[]

L4 three_sum: for i in range(len(values) - 2):
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]; output=[]; i=0

L7 three_sum: left, right = i + 1, len(values) - 1
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]; output=[]; i=0; left=1; right=3

L9 three_sum: total = values[i] + values[left] + values[right]
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]; output=[]; i=0; left=1; right=3; total=0

L11 three_sum: output.append([values[i], values[left], values[right]])
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]; output=[[0, 0, 0]]; i=0; left=1; right=3; total=0

L12 three_sum: left += 1
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]; output=[[0, 0, 0]]; i=0; left=2; right=3; total=0

L13 three_sum: right -= 1
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]; output=[[0, 0, 0]]; i=0; left=2; right=2; total=0

L4 three_sum: for i in range(len(values) - 2):
nums=[0, 0, 0, 0]; values=[0, 0, 0, 0]; output=[[0, 0, 0]]; i=1; left=2; right=2; total=0
```

Result: [[0, 0, 0]]. Many index triples represent one value triple.

Why the solution is correct

Every triple has a smallest sorted position that can serve as i, and the two-pointer exclusion proof finds its required suffix pair. Duplicate skipping removes repeated value combinations without eliminating a distinct triple, because equal values have identical sum effects.

Time and space

O(n^2) time; O(n) sorting storage plus the returned triples, which can require O(n^2) space.

Common mistakes

Skipping duplicates before a match without a clear rule can miss valid multiplicities such as [0,0,0]. Indices must remain distinct.

Reasoning card

State: i selects the first value; left and right search indices strictly after i. output contains distinct triples from completed searches.

Invariant: All triples before the current fixed/frontier state are resolved.

Decision to rehearse: Sort, fix one value, run two pointers, skip duplicates.

Finish: Move one frontier according to the sum otherwise

Transfer exercise

Extend the sorted fixed-prefix strategy to four-sum and account for the extra search dimension.

038. Container With Most Water

Interview | Module 04

Problem

Given nonnegative heights at unit-spaced indices, choose two distinct lines maximizing (right-left)*min(height[left],height[right]). Return that area; fewer than two lines return zero. Interior lines do not reduce the container's capacity.

Examples

Example 1. Indices one and eight give width seven and limiting height seven.

```
max_container([1, 8, 6, 2, 5, 4, 8, 3, 7])
=> 49
```

Example 2. Two lines do not exist.

```
max_container([])
=> 0
```

Example 3. Width one times height two.

```
max_container([2, 2])
=> 2
```

Why this mechanism works

The shorter endpoint limits the current water height. Keeping it while narrowing the width cannot improve the area, even if the other line becomes taller. Therefore discard the shorter endpoint after recording the area. This is an elimination argument, not a claim that each next area increases.

State and boundaries

left and right are candidate endpoints. best is the maximum area evaluated before discarding a frontier.

Protocol

1. Evaluate area using width times the shorter height.
2. Update best.
3. Move the shorter endpoint inward.
4. Repeat while two endpoints remain.

Visualization

```
left height 1 ........ right height 7
capacity limited by 1
keep 1 + narrower width cannot improve
```

Python implementation

```
def max_container(heights):
    left, right = 0, len(heights) - 1
    best = 0
    while left < right:
        area = (right - left) * min(heights[left], heights[right])
        best = max(best, area)
        if heights[left] <= heights[right]:
            left += 1
        else:
            right -= 1
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
max_container([1, 8, 6, 2, 5, 4, 8, 3, 7])

L2 max_container: left, right = 0, len(heights) - 1
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=0; right=8

L3 max_container: best = 0
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=0; right=8; best=0

L5 max_container: area = (right - left) * min(heights[left], heights[right])
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=0; right=8; best=0; area=8

L6 max_container: best = max(best, area)
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=0; right=8; best=8; area=8

L8 max_container: left += 1
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=1; right=8; best=8; area=8

L5 max_container: area = (right - left) * min(heights[left], heights[right])
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=1; right=8; best=8; area=49

L6 max_container: best = max(best, area)
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=1; right=8; best=49; area=49

L10 max_container: right -= 1
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=1; right=7; best=49; area=49

... 8 intermediate state changes omitted ...

L5 max_container: area = (right - left) * min(heights[left], heights[right])
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=4; right=6; best=49; area=10

L8 max_container: left += 1
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=5; right=6; best=49; area=10

L5 max_container: area = (right - left) * min(heights[left], heights[right])
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=5; right=6; best=49; area=4
```

```
L8 max_container: left += 1
heights=[1, 8, 6, 2, 5, 4, 8, 3, ...]; left=6; right=6; best=49; area=4
```

Result: 49. Indices one and eight give width seven and limiting height seven.

```
max_container([])

L2 max_container: left, right = 0, len(heights) - 1
heights=[]; left=0; right=-1

L3 max_container: best = 0
heights=[]; left=0; right=-1; best=0
```

Result: 0. Two lines do not exist.

Why the solution is correct

Every pair using the discarded short endpoint with a narrower width has area no larger than the pair just checked. Hence discarding that endpoint cannot hide a better answer. Repeating this safe elimination eventually covers or dominates every pair.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Moving the taller endpoint has no elimination proof. Summing trapped water answers a different problem from choosing one container.

Reasoning card

State: left and right are candidate endpoints. best is the maximum area evaluated before discarding a frontier.

Invariant: Discarded shorter boundaries cannot form a better container.

Decision to rehearse: Move the shorter boundary because the taller one cannot compensate for it.

Finish: Repeat while two endpoints remain

Transfer exercise

Allow nonuniform x-coordinates and replace index distance by coordinate distance in both the formula and proof.

039. Trapping Rain Water

Interview | Module 04

Problem

Return total water trapped above unit-width bars with nonnegative heights. Water at a position is limited by the tallest bar on each side, including the position itself. Empty input returns zero. Do not change the heights.

Examples

Example 1. Contributions across all valleys sum to six.

```
trapped_water([0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1])
=> 6
```

Example 2. There are no bars.

```
trapped_water([])
=> 0
```

Example 3. Interior contributions are three, one and three.

```
trapped_water([3, 0, 2, 0, 4])
=> 7
```

Why this mechanism works

A prefix/suffix maximum solution stores both walls for every cell. Two pointers avoid those arrays: finalize the side whose known wall is lower. The opposite side already supplies a wall at least that high. A taller current bar raises its side's wall and contributes zero rather than negative water.

State and boundaries

left_max and right_max are walls established by consumed bars on their sides. water is final for consumed cells; [left,right] remains unresolved.

Protocol

1. Start both known walls and water at zero.
2. Choose the side with the lower known wall.
3. Update that side's wall from its current bar.
4. Add wall minus bar height.
5. Consume that cell and repeat.

Visualization

```
bars: 3 0 2 0 4
water:0 3 1 3 0
limiting interior wall: min(3,4)=3
```

Python implementation

```
def trapped_water(heights):
    left, right = 0, len(heights) - 1
    left_max = right_max = water = 0
    while left <= right:
        if left_max <= right_max:
            left_max = max(left_max, heights[left])
            water += left_max - heights[left]
            left += 1
        else:
            right_max = max(right_max, heights[right])
            water += right_max - heights[right]
            right -= 1
    return water
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
trapped_water([0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1])

L2 trapped_water: left, right = 0, len(heights) - 1
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=0; right=11

L3 trapped_water: left_max = right_max = water = 0
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=0; right=11; left_max=0; right_max=0; water=0

L8 trapped_water: left += 1
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=1; right=11; left_max=0; right_max=0; water=0

L6 trapped_water: left_max = max(left_max, heights[left])
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=1; right=11; left_max=1; right_max=0; water=0

L8 trapped_water: left += 1
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=2; right=11; left_max=1; right_max=0; water=0

L10 trapped_water: right_max = max(right_max, heights[right])
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=2; right=11; left_max=1; right_max=1; water=0

L12 trapped_water: right -= 1
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=2; right=10; left_max=1; right_max=1; water=0

L7 trapped_water: water += left_max - heights[left]
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=2; right=10; left_max=1; right_max=1; water=1

... 12 intermediate state changes omitted ...

L8 trapped_water: left += 1
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=8; right=9; left_max=3; right_max=2; water=5

L11 trapped_water: water += right_max - heights[right]
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=8; right=9; left_max=3; right_max=2; water=6
```

```
L12 trapped_water: right -= 1
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=8; right=8; left_max=3; right_max=2; water=6

L12 trapped_water: right -= 1
heights=[0, 1, 0, 2, 1, 0, 1, 3, ...]; left=8; right=7; left_max=3; right_max=2; water=6
```

Result: 6. Contributions across all valleys sum to six.

```
trapped_water([])

L2 trapped_water: left, right = 0, len(heights) - 1
heights=[]; left=0; right=-1

L3 trapped_water: left_max = right_max = water = 0
heights=[]; left=0; right=-1; left_max=0; right_max=0; water=0
```

Result: 0. There are no bars.

Why the solution is correct

If the chosen bar is below its known wall, the opposite known wall is high enough to support exactly that limiting level. If the bar exceeds the known wall, it cannot hold water and becomes the new wall. Every cell is finalized once with a nonnegative correct contribution.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Adding a difference before raising the wall may subtract water. Container area and trapped volume have different per-position meanings.

Reasoning card

State: left_max and right_max are walls established by consumed bars on their sides. water is final for consumed cells; [left,right] remains unresolved.

Invariant: Processed positions have their final trapped-water contribution.

Decision to rehearse: Advance the side with the smaller known boundary maximum.

Finish: Consume that cell and repeat

Transfer exercise

Implement the two-array wall method as an independent oracle and compare both methods on short random arrays.

040. Partition Evens Before Odds

Interview | Module 04

Problem

Rearrange an integer list in place so every even value precedes every odd value. Relative order within either category is not required. Return the same list using the deterministic inward scan shown here. Negative values use Python's modulo parity rule.

Examples

Example 1. This scan produces a valid, deliberately unstable partition.

```
evens_before_odds([3, 1, 2, 4])
=> [4, 2, 1, 3]
```

Example 2. The region is empty.

```
evens_before_odds([])
=> []
```

Example 3. Zero and negative even values belong on the left.

```
evens_before_odds([-3, -2, 0])
=> [0, -2, -3]
```

Why this mechanism works

An even value at the left frontier is already on the correct side; an odd value at the right frontier is likewise final. When both are misplaced, swap them. Unlike stable filtering, this contract permits exchanges across the list.

State and boundaries

Positions before left are even; positions after right are odd. The inclusive interior is unclassified.

Protocol

1. Advance left across already-correct even values.
2. Retreat right across already-correct odd values.
3. Swap the two misplaced endpoints if distinct.
4. Shrink both frontiers after a swap.

Visualization

```
[even prefix | unknown | odd suffix]
wrong left: odd <-> even :wrong right
exchange finalizes both endpoints
```

Python implementation

```
def evens_before_odds(nums):
    left, right = 0, len(nums) - 1
    while left < right:
        while left < right and nums[left] % 2 == 0:
            left += 1
        while left < right and nums[right] % 2 != 0:
            right -= 1
        if left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1
    return nums
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
evens_before_odds([3, 1, 2, 4])

L2 evens_before_odds: left, right = 0, len(nums) - 1
nums=[3, 1, 2, 4]; left=0; right=3

L9 evens_before_odds: nums[left], nums[right] = nums[right], nums[left]
nums=[4, 1, 2, 3]; left=0; right=3

L10 evens_before_odds: left += 1
nums=[4, 1, 2, 3]; left=1; right=3

L11 evens_before_odds: right -= 1
nums=[4, 1, 2, 3]; left=1; right=2

L9 evens_before_odds: nums[left], nums[right] = nums[right], nums[left]
nums=[4, 2, 1, 3]; left=1; right=2

L10 evens_before_odds: left += 1
nums=[4, 2, 1, 3]; left=2; right=2

L11 evens_before_odds: right -= 1
nums=[4, 2, 1, 3]; left=2; right=1
```

Result: [4, 2, 1, 3]. This scan produces a valid, deliberately unstable partition.

```
evens_before_odds([])

L2 evens_before_odds: left, right = 0, len(nums) - 1
nums=[]; left=0; right=-1
```

Result: []. The region is empty.

Why the solution is correct

Skipping finalizes correctly classified values. A swap places an even value on the left and an odd value on the right, preserving the region claims. The unclassified region shrinks until it has at most one cell, which lies at the partition boundary.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

A stable expected output is too strict for this contract. Every skip loop still needs a crossing check.

Reasoning card

State: Positions before left are even; positions after right are odd. The inclusive interior is unclassified.

Invariant: Everything left of left is even; everything right of right is odd.

Decision to rehearse: Skip already-correct values before swapping.

Finish: Shrink both frontiers after a swap

Transfer exercise

Require stability. Compare a new-array solution with more complicated constant-space approaches rather than assuming the same swaps remain valid.

Module 05
Sliding Windows

Make fixed windows, longest-valid windows, shortest-satisfying windows, and counting windows feel distinct.

A window is a contiguous region whose contents change at both ends. Fixed-width problems make removal predictable; variable-width problems require a validity condition and a repair loop. Practice ingest, repair and commit in that order, and explain when a counting problem commits several answers rather than just one.

The ten-step progression

041. Maximum Fixed-Window Sum

042. Count Fixed Windows Above Threshold

043. Moving Average Stream

044. Longest Substring Without Repeats

045. Longest Substring with At Most K Distinct

046. Longest Ones After K Flips

047. Permutation Inclusion

048. Shortest Positive Window Reaching Target

049. Minimum Covering Substring

050. Count Subarrays with Exactly K Distinct

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

041. Maximum Fixed-Window Sum

Foundation | Module 05

Problem

Return the maximum sum of a contiguous window containing exactly k integers. Values may be negative. If k<1 or k>len(nums), return None because no such window exists. The input is unchanged.

Examples

Example 1. The window [5,1,3] has the largest sum.

```
max_window_sum([2, 1, 5, 1, 3, 2], 3)
=> 9
```

Example 2. A real negative window seeds best.

```
max_window_sum([-5, -2, -3], 2)
=> -5
```

Example 3. No complete window exists.

```
max_window_sum([], 1)
=> None
```

Why this mechanism works

Consecutive size-k windows differ by one departing and one arriving value. Update the sum with that difference instead of summing k cells again. Seed the best answer from an actual window, because zero may exceed every legal negative window sum.

State and boundaries

total is the current size-k window sum. best is the maximum sum among complete windows processed so far.

Protocol

1. Reject an invalid window size.
2. Sum the first k values.
3. Move the right endpoint forward.
4. Add the incoming value and subtract the departing value.
5. Compare the repaired sum with best.

Visualization

```
[2,1,5] 1 3 2 -> sum 8
2 [1,5,1] 3 2 -> 8 - 2 + 1 = 7
2 1 [5,1,3] 2 -> 7 - 1 + 3 = 9
```

Python implementation

```
def max_window_sum(nums, k):
    if k < 1 or k > len(nums):
        return None
    total = sum(nums[:k])
    best = total
    for right in range(k, len(nums)):
        total += nums[right] - nums[right - k]
        best = max(best, total)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
max_window_sum([2, 1, 5, 1, 3, 2], 3)

L2 max_window_sum: if k < 1 or k > len(nums):
nums=[2, 1, 5, 1, 3, 2]; k=3

L4 max_window_sum: total = sum(nums[:k])
nums=[2, 1, 5, 1, 3, 2]; k=3; total=8

L5 max_window_sum: best = total
nums=[2, 1, 5, 1, 3, 2]; k=3; total=8; best=8

L6 max_window_sum: for right in range(k, len(nums)):
nums=[2, 1, 5, 1, 3, 2]; k=3; total=8; best=8; right=3

L7 max_window_sum: total += nums[right] - nums[right - k]
nums=[2, 1, 5, 1, 3, 2]; k=3; total=7; best=8; right=3

L6 max_window_sum: for right in range(k, len(nums)):
nums=[2, 1, 5, 1, 3, 2]; k=3; total=7; best=8; right=4

L7 max_window_sum: total += nums[right] - nums[right - k]
nums=[2, 1, 5, 1, 3, 2]; k=3; total=9; best=8; right=4

L8 max_window_sum: best = max(best, total)
nums=[2, 1, 5, 1, 3, 2]; k=3; total=9; best=9; right=4

L6 max_window_sum: for right in range(k, len(nums)):
nums=[2, 1, 5, 1, 3, 2]; k=3; total=9; best=9; right=5

L7 max_window_sum: total += nums[right] - nums[right - k]
nums=[2, 1, 5, 1, 3, 2]; k=3; total=6; best=9; right=5
```

Result: 9. The window [5,1,3] has the largest sum.

```
max_window_sum([-5, -2, -3], 2)

L2 max_window_sum: if k < 1 or k > len(nums):
nums=[-5, -2, -3]; k=2
```

```
L4 max_window_sum: total = sum(nums[:k])
nums=[-5, -2, -3]; k=2; total=-7

L5 max_window_sum: best = total
nums=[-5, -2, -3]; k=2; total=-7; best=-7

L6 max_window_sum: for right in range(k, len(nums)):
nums=[-5, -2, -3]; k=2; total=-7; best=-7; right=2

L7 max_window_sum: total += nums[right] - nums[right - k]
nums=[-5, -2, -3]; k=2; total=-5; best=-7; right=2

L8 max_window_sum: best = max(best, total)
nums=[-5, -2, -3]; k=2; total=-5; best=-5; right=2
```

Result: -5. A real negative window seeds best.

Why the solution is correct

The initial sum represents the first window. Each replacement removes exactly the value leaving the window and adds exactly the entering value, preserving its size and sum. Every complete window is visited once, so the maximum is correct.

Time and space

O(n) time; O(k) temporary space from the initial slice in this implementation, with O(1) persistent state.

Common mistakes

Initializing best to zero fails for negative sums. Subtracting nums[right-k+1] removes the wrong departing value.

Reasoning card

State: total is the current size-k window sum. best is the maximum sum among complete windows processed so far.

Invariant: window_sum describes exactly k current values.

Decision to rehearse: Initialize one complete window; add incoming and remove outgoing exactly once.

Finish: Compare the repaired sum with best

Transfer exercise

Remove the initial slice allocation while keeping the same recurrence and O(n) time.

042. Count Fixed Windows Above Threshold

Foundation | Module 05

Problem

Count contiguous windows of exactly k integers whose sum is at least threshold. Values may be negative. Return zero if k<1 or k exceeds the list length. Windows may overlap and are counted by their positions.

Examples

Example 1. Sums eight and nine qualify.

```
count_windows([2, 1, 5, 1, 3, 2], 3, 8)
=> 2
```

Example 2. Equality with the threshold qualifies.

```
count_windows([2, 2], 1, 2)
=> 2
```

Example 3. A qualifying partial sum is not a full window.

```
count_windows([1], 2, 0)
=> 0
```

Why this mechanism works

The sum update is identical to maximum fixed-window sum, but the commit operation is a qualification count rather than a maximum. Only evaluate complete size-k windows; a shorter prefix may satisfy the numeric threshold but is not a legal candidate.

State and boundaries

total describes the current complete window. count is the number of visited complete windows with total>=threshold.

Protocol

1. Handle invalid k.
2. Evaluate the first complete window.
3. Replace one departing value with each incoming value.
4. Add one when the new complete sum qualifies.

5. Return count.

Visualization

```
size 3 sums: 8, 7, 9, 6
threshold 8: Y, N, Y, N
count: 2
```

Python implementation

```
def count_windows(nums, k, threshold):
    if k < 1 or k > len(nums):
        return 0
    total = sum(nums[:k])
    count = int(total >= threshold)
    for right in range(k, len(nums)):
        total += nums[right] - nums[right - k]
        count += int(total >= threshold)
    return count
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
count_windows([2, 1, 5, 1, 3, 2], 3, 8)

L2 count_windows: if k < 1 or k > len(nums):
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8

L4 count_windows: total = sum(nums[:k])
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=8

L5 count_windows: count = int(total >= threshold)
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=8; count=1

L6 count_windows: for right in range(k, len(nums)):
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=8; count=1; right=3

L7 count_windows: total += nums[right] - nums[right - k]
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=7; count=1; right=3

L6 count_windows: for right in range(k, len(nums)):
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=7; count=1; right=4

L7 count_windows: total += nums[right] - nums[right - k]
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=9; count=1; right=4

L8 count_windows: count += int(total >= threshold)
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=9; count=2; right=4

L6 count_windows: for right in range(k, len(nums)):
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=9; count=2; right=5

L7 count_windows: total += nums[right] - nums[right - k]
nums=[2, 1, 5, 1, 3, 2]; k=3; threshold=8; total=6; count=2; right=5
```

Result: 2. Sums eight and nine qualify.

```
count_windows([2, 2], 1, 2)

L2 count_windows: if k < 1 or k > len(nums):
nums=[2, 2]; k=1; threshold=2

L4 count_windows: total = sum(nums[:k])
nums=[2, 2]; k=1; threshold=2; total=2

L5 count_windows: count = int(total >= threshold)
nums=[2, 2]; k=1; threshold=2; total=2; count=1

L6 count_windows: for right in range(k, len(nums)):
nums=[2, 2]; k=1; threshold=2; total=2; count=1; right=1

L8 count_windows: count += int(total >= threshold)
nums=[2, 2]; k=1; threshold=2; total=2; count=2; right=1
```

Result: 2. Equality with the threshold qualifies.

Why the solution is correct

The replacement recurrence enumerates exactly the size-k windows. Each contributes either one or zero according to the required predicate, so the accumulated count neither misses qualifying windows nor includes partial ones.

Time and space

O(n) time; O(k) temporary slice space and O(1) persistent state.

Common mistakes

Counting before a full window exists introduces extra candidates. Using > instead of >= loses threshold-equal windows.

Reasoning card

State: total describes the current complete window. count is the number of visited complete windows with total>=threshold.

Invariant: Every recorded candidate has exactly k elements.

Decision to rehearse: Update the fixed window before evaluating each complete candidate.

Finish: Return count

Transfer exercise

Count windows whose average reaches a threshold using a sum comparison, and explain when multiplication avoids floating-point error.

043. Moving Average Stream

Foundation | Module 05

Problem

For a finite stream of numeric values and a positive integer k, return the moving average after every insertion, using the most recent min(k, values_seen) elements. Empty input returns []. The denominator is the current window size during warm-up.

Examples

Example 1. Warm-up uses the actual number of values.

```
moving_averages([1, 10, 3, 5], 3)
=> [1.0, 5.5, 4.666666666666667, 6.0]
```

Example 2. No insertion means no average.

```
moving_averages([], 2)
=> []
```

Example 3. A size-one window reports each incoming value.

```
moving_averages([2, -3], 1)
=> [2.0, -3.0]
```

Why this mechanism works

A stream cannot subtract the departing value unless it remembers it. A FIFO deque stores the recent values and a running sum avoids repeated summation. Remove the oldest value only when the new insertion exceeds the capacity.

State and boundaries

window contains the latest at-most-k values in arrival order. total is their sum.

Protocol

1. Append the incoming value and add it to total.
2. If size exceeds k, pop the oldest and subtract it.
3. Divide by the repaired window size.
4. Append the average.

Visualization

```
k=3, incoming 1,10,3,5
windows: [1], [1,10], [1,10,3], [10,3,5]
divisors: 1,2,3,3
```

Python implementation

```
from collections import deque

def moving_averages(values, k):
    if k < 1:
        raise ValueError('k must be positive')
    window = deque()
    total = 0
    output = []
    for value in values:
        window.append(value)
        total += value
        if len(window) > k:
            total -= window.popleft()
        output.append(total / len(window))
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
moving_averages([1, 10, 3, 5], 3)

L4 moving_averages: if k < 1:
values=[1, 10, 3, 5]; k=3

L6 moving_averages: window = deque()
values=[1, 10, 3, 5]; k=3; window=deque([])

L7 moving_averages: total = 0
values=[1, 10, 3, 5]; k=3; window=deque([]); total=0

L8 moving_averages: output = []
values=[1, 10, 3, 5]; k=3; window=deque([]); total=0; output=[]

L9 moving_averages: for value in values:
values=[1, 10, 3, 5]; k=3; window=deque([]); total=0; output=[]; value=1

L10 moving_averages: window.append(value)
values=[1, 10, 3, 5]; k=3; window=deque([1]); total=0; output=[]; value=1

L11 moving_averages: total += value
values=[1, 10, 3, 5]; k=3; window=deque([1]); total=1; output=[]; value=1

L14 moving_averages: output.append(total / len(window))
values=[1, 10, 3, 5]; k=3; window=deque([1]); total=1; output=[1.0]; value=1

... 9 intermediate state changes omitted ...

L10 moving_averages: window.append(value)
values=[1, 10, 3, 5]; k=3; window=deque([1, 10, 3, 5]); total=14; output=[1.0, 5.5,
    4.666666666666667]; value=5

L11 moving_averages: total += value
values=[1, 10, 3, 5]; k=3; window=deque([1, 10, 3, 5]); total=19; output=[1.0, 5.5,
    4.666666666666667]; value=5

L13 moving_averages: total -= window.popleft()
values=[1, 10, 3, 5]; k=3; window=deque([10, 3, 5]); total=18; output=[1.0, 5.5,
    4.666666666666667]; value=5
```

```
L14 moving_averages: output.append(total / len(window))
values=[1, 10, 3, 5]; k=3; window=deque([10, 3, 5]); total=18; output=[1.0, 5.5,
    4.666666666666667, 6.0]; value=5
```

Result: [1.0, 5.5, 4.666666666666667, 6.0]. Warm-up uses the actual number of values.

```
moving_averages([], 2)

L4 moving_averages: if k < 1:
values=[]; k=2

L6 moving_averages: window = deque()
values=[]; k=2; window=deque([])

L7 moving_averages: total = 0
values=[]; k=2; window=deque([]); total=0

L8 moving_averages: output = []
values=[]; k=2; window=deque([]); total=0; output=[]
```

Result: []. No insertion means no average.

Why the solution is correct

After insertion there can be only one excess value. Removing the oldest restores exactly the most recent k values, or keeps the entire shorter prefix during warm-up. The sum tracks those same values, making the quotient correct.

Time and space

O(n) time; O(k) queue space plus O(n) output. Floating-point averages have ordinary numeric rounding limits.

Common mistakes

Dividing by k during warm-up biases early averages. A list pop(0) introduces linear shifts per removal.

Reasoning card

State: window contains the latest at-most-k values in arrival order. total is their sum.

Invariant: Deque and total describe exactly the retained values.

Decision to rehearse: Add first; evict only when the window becomes oversized.

Finish: Append the average

Transfer exercise

Turn the function into a class with a next(value) method, preserving the exact same window invariant.

044. Longest Substring Without Repeats

Foundation | Module 05

Problem

Return the length of the longest contiguous substring with no repeated character. Case and punctuation are significant. Empty input returns zero. Length counts Python string elements.

Examples

Example 1. The maximum unique window has length three.

```
longest_unique_substring("abcabcbb")
=> 3
```

Example 2. The final a cannot reopen the old prefix.

```
longest_unique_substring("abba")
=> 2
```

Example 3. There is no nonempty substring.

```
longest_unique_substring("")
=> 0
```

Why this mechanism works

When the incoming character repeats inside the active window, its previous occurrence must be excluded. Store last-seen indices and jump the left boundary directly past that occurrence. Never move left backward when the previous occurrence lies outside the active window.

State and boundaries

[left,right+1) is duplicate-free after repair. last records the most recent index of each character anywhere in the processed prefix.

Protocol

1. Read the incoming character's last position.
2. Raise left to at least one past that position.
3. Record its new position.
4. Commit the repaired window length to best.

Visualization

```
a b b a
0 1 2 3
second b moves left to 2
last a=0 must not move left backward
```

Python implementation

```
def longest_unique_substring(text):
    last = {}
    left = best = 0
    for right, char in enumerate(text):
        if char in last:
            left = max(left, last[char] + 1)
        last[char] = right
        best = max(best, right - left + 1)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
longest_unique_substring("abcabcbb")

L2 longest_unique_substring: last = {}
text='abcabcbb'; last={}

L3 longest_unique_substring: left = best = 0
text='abcabcbb'; last={}; left=0; best=0

L4 longest_unique_substring: for right, char in enumerate(text):
text='abcabcbb'; last={}; left=0; best=0; right=0; char='a'

L7 longest_unique_substring: last[char] = right
text='abcabcbb'; last={'a':0}; left=0; best=0; right=0; char='a'

L8 longest_unique_substring: best = max(best, right - left + 1)
text='abcabcbb'; last={'a':0}; left=0; best=1; right=0; char='a'

L4 longest_unique_substring: for right, char in enumerate(text):
text='abcabcbb'; last={'a':0}; left=0; best=1; right=1; char='b'

L7 longest_unique_substring: last[char] = right
text='abcabcbb'; last={'a':0, 'b':1}; left=0; best=1; right=1; char='b'

L8 longest_unique_substring: best = max(best, right - left + 1)
text='abcabcbb'; last={'a':0, 'b':1}; left=0; best=2; right=1; char='b'

... 14 intermediate state changes omitted ...

L7 longest_unique_substring: last[char] = right
text='abcabcbb'; last={'a':3, 'b':6, 'c':5}; left=5; best=3; right=6; char='b'

L4 longest_unique_substring: for right, char in enumerate(text):
text='abcabcbb'; last={'a':3, 'b':6, 'c':5}; left=5; best=3; right=7; char='b'

L6 longest_unique_substring: left = max(left, last[char] + 1)
text='abcabcbb'; last={'a':3, 'b':6, 'c':5}; left=7; best=3; right=7; char='b'
```

```
L7 longest_unique_substring: last[char] = right
text='abcabcbb'; last={'a':3, 'b':7, 'c':5}; left=7; best=3; right=7; char='b'
```

Result: 3. The maximum unique window has length three.

```
longest_unique_substring("abba")

L2 longest_unique_substring: last = {}
text='abba'; last={}

L3 longest_unique_substring: left = best = 0
text='abba'; last={}; left=0; best=0

L4 longest_unique_substring: for right, char in enumerate(text):
text='abba'; last={}; left=0; best=0; right=0; char='a'

L7 longest_unique_substring: last[char] = right
text='abba'; last={'a':0}; left=0; best=0; right=0; char='a'

L8 longest_unique_substring: best = max(best, right - left + 1)
text='abba'; last={'a':0}; left=0; best=1; right=0; char='a'

L4 longest_unique_substring: for right, char in enumerate(text):
text='abba'; last={'a':0}; left=0; best=1; right=1; char='b'

L7 longest_unique_substring: last[char] = right
text='abba'; last={'a':0, 'b':1}; left=0; best=1; right=1; char='b'

L8 longest_unique_substring: best = max(best, right - left + 1)
text='abba'; last={'a':0, 'b':1}; left=0; best=2; right=1; char='b'

... 1 intermediate state changes omitted ...

L6 longest_unique_substring: left = max(left, last[char] + 1)
text='abba'; last={'a':0, 'b':1}; left=2; best=2; right=2; char='b'

L7 longest_unique_substring: last[char] = right
text='abba'; last={'a':0, 'b':2}; left=2; best=2; right=2; char='b'

L4 longest_unique_substring: for right, char in enumerate(text):
text='abba'; last={'a':0, 'b':2}; left=2; best=2; right=3; char='a'

L7 longest_unique_substring: last[char] = right
text='abba'; last={'a':3, 'b':2}; left=2; best=2; right=3; char='a'
```

Result: 2. The final a cannot reopen the old prefix.

Why the solution is correct

Before insertion the window is unique, so the incoming character is the only possible new duplicate. Removing its earlier occurrence and everything before it restores uniqueness with the longest possible suffix. Maximizing these longest valid suffixes covers the optimal substring.

Time and space

O(n) expected time and O(u) auxiliary space.

Common mistakes

Assigning left=last[char]+1 without max can move backward in abba. Updating last before querying loses the previous index.

Reasoning card

State: [left,right+1) is duplicate-free after repair. last records the most recent index of each character anywhere in the processed prefix.

Invariant: The current window contains no repeated character.

Decision to rehearse: Add right, repair duplicates from left, then commit.

Finish: Commit the repaired window length to best

Transfer exercise

Return the earliest longest substring itself while retaining a strict best-length improvement rule.

045. Longest Substring with At Most K Distinct

Bridge | Module 05

Problem

Return the longest substring length containing at most k distinct characters. k may be zero; nonpositive k returns zero. Repeated occurrences of the same character count as one distinct character. The empty string returns zero.

Examples

Example 1. The prefix ece uses two distinct characters.

```
longest_k_distinct("eceba", 2)
=> 3
```

Example 2. No nonempty window can use zero distinct characters.

```
longest_k_distinct("aaa", 0)
=> 0
```

Example 3. Empty input has length zero.

```
longest_k_distinct("", 3)
=> 0
```

Why this mechanism works

A frequency map represents the current window, not all history. Inserting a new character may violate the distinct-count limit. Remove characters from the left until the map contains at most k keys, deleting keys whose counts reach zero.

State and boundaries

counts contains exactly positive frequencies in the active window. left is repaired before best is updated.

Protocol

1. Add the incoming character.
2. While the distinct count exceeds k, consume the left character.
3. Delete a count that becomes zero.
4. Advance left.
5. Commit the longest valid window ending here.

Visualization

```
e c e b a, k=2
[e c e] valid length 3
add b -> shrink until only e,b remain
```

Python implementation

```
def longest_k_distinct(text, k):
    if k <= 0:
        return 0
    counts = {}
    left = best = 0
    for right, char in enumerate(text):
        counts[char] = counts.get(char, 0) + 1
        while len(counts) > k:
            old = text[left]
            counts[old] -= 1
            if counts[old] == 0:
                del counts[old]
            left += 1
        best = max(best, right - left + 1)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
longest_k_distinct("eceba", 2)

L2 longest_k_distinct: if k <= 0:
text='eceba'; k=2

L4 longest_k_distinct: counts = {}
text='eceba'; k=2; counts={}

L5 longest_k_distinct: left = best = 0
text='eceba'; k=2; counts={}; left=0; best=0

L6 longest_k_distinct: for right, char in enumerate(text):
text='eceba'; k=2; counts={}; left=0; best=0; right=0; char='e'

L7 longest_k_distinct: counts[char] = counts.get(char, 0) + 1
text='eceba'; k=2; counts={'e':1}; left=0; best=0; right=0; char='e'

L14 longest_k_distinct: best = max(best, right - left + 1)
text='eceba'; k=2; counts={'e':1}; left=0; best=1; right=0; char='e'

L6 longest_k_distinct: for right, char in enumerate(text):
text='eceba'; k=2; counts={'e':1}; left=0; best=1; right=1; char='c'

L7 longest_k_distinct: counts[char] = counts.get(char, 0) + 1
text='eceba'; k=2; counts={'e':1, 'c':1}; left=0; best=1; right=1; char='c'

... 15 intermediate state changes omitted ...

L9 longest_k_distinct: old = text[left]
text='eceba'; k=2; counts={'e':1, 'b':1, 'a':1}; left=2; best=3; right=4; char='a'; old='e'

L10 longest_k_distinct: counts[old] -= 1
```

```
text='eceba'; k=2; counts={'e':0, 'b':1, 'a':1}; left=2; best=3; right=4; char='a'; old='e'

L12 longest_k_distinct: del counts[old]
text='eceba'; k=2; counts={'b':1, 'a':1}; left=2; best=3; right=4; char='a'; old='e'

L13 longest_k_distinct: left += 1
text='eceba'; k=2; counts={'b':1, 'a':1}; left=3; best=3; right=4; char='a'; old='e'
```

Result: 3. The prefix ece uses two distinct characters.

```
longest_k_distinct("aaa", 0)

L2 longest_k_distinct: if k <= 0:
text='aaa'; k=0
```

Result: 0. No nonempty window can use zero distinct characters.

Why the solution is correct

Insertion is the only way validity can break. Removing a left prefix eventually restores the key limit, and stopping at the first valid position keeps the longest valid suffix for that endpoint. The maximum over endpoints is the global optimum.

Time and space

O(n) expected time because each endpoint advances at most n times; O(min(n,u)) map space.

Common mistakes

Leaving zero-count keys inflates the distinct count. A single if repair may leave several excess occurrences before a distinct key disappears.

Reasoning card

State: counts contains exactly positive frequencies in the active window. left is repaired before best is updated.

Invariant: After repair, the map describes a valid window with at most k keys.

Decision to rehearse: Maintain counts; delete a key only when its count reaches zero.

Finish: Commit the longest valid window ending here

Transfer exercise

Compare the state needed for at most k distinct characters with at most k total occurrences of one distinguished character.

046. Longest Ones After K Flips

Bridge | Module 05

Problem

Given a binary list and nonnegative k, return the longest contiguous length that can be made all ones by changing at most k zeros to ones. The input is not actually modified. Empty input returns zero.

Examples

Example 1. A six-element window needs exactly two flips.

```
longest_ones([1,1,1,0,0,0,1,1,1,1,0], 2)
=> 6
```

Example 2. No zero can be included without a flip.

```
longest_ones([0, 0], 0)
=> 0
```

Example 3. There is no window.

```
longest_ones([], 3)
=> 0
```

Why this mechanism works

The number of required flips is exactly the number of zeros in the window. Ones have no cost. This reduces the transformation story to an at-most-budget window: expand, repair excessive zero count, then commit the valid length.

State and boundaries

zeros is the count of zeros in [left,right+1). best is updated only after zeros<=k.

Protocol

1. Count the incoming zero if present.
2. While zeros exceeds k, remove the leftmost element's cost.
3. Advance left during repair.
4. Update best with the repaired length.

Visualization

```
window: 1 1 0 1 0 -> cost 2
add 0 -> cost 3
remove through the first 0 to restore budget 2
```

Python implementation

```
def longest_ones(nums, k):
    left = zeros = best = 0
    for right, value in enumerate(nums):
        zeros += int(value == 0)
        while zeros > k:
            zeros -= int(nums[left] == 0)
            left += 1
        best = max(best, right - left + 1)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
longest_ones([1,1,1,0,0,0,1,1,1,1,0], 2)

L2 longest_ones: left = zeros = best = 0
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=0; zeros=0; best=0

L3 longest_ones: for right, value in enumerate(nums):
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=0; zeros=0; best=0; right=0; value=1

L8 longest_ones: best = max(best, right - left + 1)
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=0; zeros=0; best=1; right=0; value=1

L3 longest_ones: for right, value in enumerate(nums):
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=0; zeros=0; best=1; right=1; value=1

L8 longest_ones: best = max(best, right - left + 1)
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=0; zeros=0; best=2; right=1; value=1

L3 longest_ones: for right, value in enumerate(nums):
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=0; zeros=0; best=2; right=2; value=1

L8 longest_ones: best = max(best, right - left + 1)
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=0; zeros=0; best=3; right=2; value=1

L3 longest_ones: for right, value in enumerate(nums):
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=0; zeros=0; best=3; right=3; value=0

... 17 intermediate state changes omitted ...

L3 longest_ones: for right, value in enumerate(nums):
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=4; zeros=2; best=6; right=10; value=0

L4 longest_ones: zeros += int(value == 0)
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=4; zeros=3; best=6; right=10; value=0

L6 longest_ones: zeros -= int(nums[left] == 0)
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=4; zeros=2; best=6; right=10; value=0

L7 longest_ones: left += 1
nums=[1, 1, 1, 0, 0, 0, 1, 1, ...]; k=2; left=5; zeros=2; best=6; right=10; value=0
```

Result: 6. A six-element window needs exactly two flips.

```
longest_ones([0, 0], 0)

L2 longest_ones: left = zeros = best = 0
nums=[0, 0]; k=0; left=0; zeros=0; best=0

L3 longest_ones: for right, value in enumerate(nums):
nums=[0, 0]; k=0; left=0; zeros=0; best=0; right=0; value=0

L4 longest_ones: zeros += int(value == 0)
nums=[0, 0]; k=0; left=0; zeros=1; best=0; right=0; value=0

L6 longest_ones: zeros -= int(nums[left] == 0)
nums=[0, 0]; k=0; left=0; zeros=0; best=0; right=0; value=0

L7 longest_ones: left += 1
nums=[0, 0]; k=0; left=1; zeros=0; best=0; right=0; value=0

L3 longest_ones: for right, value in enumerate(nums):
nums=[0, 0]; k=0; left=1; zeros=0; best=0; right=1; value=0

L4 longest_ones: zeros += int(value == 0)
nums=[0, 0]; k=0; left=1; zeros=1; best=0; right=1; value=0

L6 longest_ones: zeros -= int(nums[left] == 0)
nums=[0, 0]; k=0; left=1; zeros=0; best=0; right=1; value=0

L7 longest_ones: left += 1
nums=[0, 0]; k=0; left=2; zeros=0; best=0; right=1; value=0
```

Result: 0. No zero can be included without a flip.

Why the solution is correct

A window is feasible exactly when its zero count fits the flip budget. The repair loop discards the shortest prefix necessary to restore feasibility, leaving the longest feasible suffix ending at right. Every optimum has an endpoint considered by this scan.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Counting zero runs rather than individual zeros changes the budget. Mutating the input loses the original costs needed during repair.

Reasoning card

State: zeros is the count of zeros in [left,right+1). best is updated only after zeros<=k.

Invariant: Current window contains at most k zeros.

Decision to rehearse: Treat zeros as resource usage; repair when resource exceeds k.

Finish: Update best with the repaired length

Transfer exercise

Give each zero a different nonnegative conversion cost. Replace zero count by window cost and preserve the same repair logic.

047. Permutation Inclusion

Bridge | Module 05

Problem

Return whether text contains a contiguous substring that is a permutation of pattern. Character multiplicities must match exactly. Comparisons are case-sensitive. An empty pattern is present in every text; a longer pattern than text cannot fit.

Examples

Example 1. The substring ba is a permutation.

```
permutation_inclusion("ab", "eidbaooo")
=> True
```

Example 2. The repeated-letter supply does not match.

```
permutation_inclusion("aab", "abb")
=> False
```

Example 3. The empty pattern is always present.

```
permutation_inclusion("", "x")
=> True
```

Why this mechanism works

Every possible answer has length len(pattern). Maintain a fixed window frequency map and compare it with the pattern's frequencies. Deleting zero counts makes dictionary equality reflect true multisets. This intentionally direct version favors a transparent state contract; a mismatch counter can optimize large alphabets later.

State and boundaries

need is the pattern multiset. window is the most recent at-most-m characters, with only positive counts.

Protocol

1. Handle an empty pattern.
2. Add each incoming character.
3. Remove the character more than m positions behind.
4. Compare maps only once a full window exists.
5. Return True on equality, otherwise False at the end.

Visualization

```
pattern ab -> {a:1,b:1}
text eid[ba]ooo
window ba has the same multiset
```

Python implementation

```
from collections import Counter

def permutation_inclusion(pattern, text):
    m = len(pattern)
    if m == 0:
        return True
    need, window = Counter(pattern), {}
    for right, char in enumerate(text):
        window[char] = window.get(char, 0) + 1
        if right >= m:
            old = text[right - m]
            window[old] -= 1
            if window[old] == 0:
                del window[old]
        if right + 1 >= m and window == need:
            return True
    return False
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
permutation_inclusion("ab", "eidbaooo")

L4 permutation_inclusion: m = len(pattern)
pattern='ab'; text='eidbaooo'; m=2

L7 permutation_inclusion: need, window = Counter(pattern), {}
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={}

L8 permutation_inclusion: for right, char in enumerate(text):
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={}; right=0; char='e'

L9 permutation_inclusion: window[char] = window.get(char, 0) + 1
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'e':1}; right=0; char='e'

L8 permutation_inclusion: for right, char in enumerate(text):
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'e':1}; right=1; char='i'

L9 permutation_inclusion: window[char] = window.get(char, 0) + 1
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'e':1, 'i':1}; right=1;
    char='i'

L8 permutation_inclusion: for right, char in enumerate(text):
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'e':1, 'i':1}; right=2;
    char='d'

L9 permutation_inclusion: window[char] = window.get(char, 0) + 1
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'e':1, 'i':1, 'd':1};
    right=2; char='d'
```

```
... 9 intermediate state changes omitted ...

L9 permutation_inclusion: window[char] = window.get(char, 0) + 1
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'d':1, 'b':1, 'a':1};
    right=4; char='a'; old='i'

L11 permutation_inclusion: old = text[right - m]
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'d':1, 'b':1, 'a':1};
    right=4; char='a'; old='d'

L12 permutation_inclusion: window[old] -= 1
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'d':0, 'b':1, 'a':1};
    right=4; char='a'; old='d'

L14 permutation_inclusion: del window[old]
pattern='ab'; text='eidbaooo'; m=2; need={'a':1, 'b':1}; window={'b':1, 'a':1}; right=4;
    char='a'; old='d'
```

Result: True. The substring ba is a permutation.

```
permutation_inclusion("aab", "abb")

L4 permutation_inclusion: m = len(pattern)
pattern='aab'; text='abb'; m=3

L7 permutation_inclusion: need, window = Counter(pattern), {}
pattern='aab'; text='abb'; m=3; need={'a':2, 'b':1}; window={}

L8 permutation_inclusion: for right, char in enumerate(text):
pattern='aab'; text='abb'; m=3; need={'a':2, 'b':1}; window={}; right=0; char='a'

L9 permutation_inclusion: window[char] = window.get(char, 0) + 1
pattern='aab'; text='abb'; m=3; need={'a':2, 'b':1}; window={'a':1}; right=0; char='a'

L8 permutation_inclusion: for right, char in enumerate(text):
pattern='aab'; text='abb'; m=3; need={'a':2, 'b':1}; window={'a':1}; right=1; char='b'

L9 permutation_inclusion: window[char] = window.get(char, 0) + 1
pattern='aab'; text='abb'; m=3; need={'a':2, 'b':1}; window={'a':1, 'b':1}; right=1; char='b'

L8 permutation_inclusion: for right, char in enumerate(text):
pattern='aab'; text='abb'; m=3; need={'a':2, 'b':1}; window={'a':1, 'b':1}; right=2; char='b'

L9 permutation_inclusion: window[char] = window.get(char, 0) + 1
pattern='aab'; text='abb'; m=3; need={'a':2, 'b':1}; window={'a':1, 'b':2}; right=2; char='b'
```

Result: False. The repeated-letter supply does not match.

Why the solution is correct

The fixed-size update enumerates all possible answer intervals. Equal positive frequency maps are equivalent to equal character multiplicities, so a hit is a permutation and every permutation produces a hit.

Time and space

O(n*u) worst-case time for map comparisons, where u is the number of distinct keys compared (at most O(m)); O(m) space. A fixed-size alphabet makes u constant.

Common mistakes

Claiming unconditional O(n) while comparing arbitrary-size dictionaries hides repeated work. Forgetting multiplicities accepts incorrect windows.

Reasoning card

State: need is the pattern multiset. window is the most recent at-most-m characters, with only positive counts.

Invariant: The state describes exactly the latest pattern-length window.

Decision to rehearse: Use a fixed-size frequency window and compare fulfilled requirements.

Finish: Return True on equality, otherwise False at the end

Transfer exercise

Maintain the number of character-frequency mismatches so each window can be checked in expected O(1) time.

048. Shortest Positive Window Reaching Target

Interview | Module 05

Problem

For strictly positive integer values and a positive target, return the minimum length of a nonempty contiguous subarray with sum at least target. Return zero if none exists. Strict positivity is a required input condition for this window method.

Examples

Example 1. The final pair [4,3] reaches seven.

```
shortest_positive_window([2, 3, 1, 2, 4, 3], 7)
=> 2
```

Example 2. No window reaches the target.

```
shortest_positive_window([1, 1], 3)
=> 0
```

Example 3. There is no nonempty candidate.

```
shortest_positive_window([], 1)
=> 0
```

Why this mechanism works

Expansion can only raise the sum, and removing from the left can only lower it. Once a window qualifies, record it before shrinking, then continue shrinking while it still qualifies. This is the opposite commit timing from longest-valid windows: the current satisfying candidate must be saved before repair destroys it.

State and boundaries

total is the active window sum. best is the shortest satisfying window encountered, initialized to an impossible length n+1.

Protocol

1. Add the incoming positive value.

2. While total>=target record the current length.

3. Remove the left value and advance left.

4. Continue expansion after qualification is lost.

5. Return zero if best was never improved.

Visualization

```
[2,3,1,2] sum 8 -> record 4
remove 2 -> sum 6, stop
later [4,3] sum 7 -> record 2
```

Python implementation

```
def shortest_positive_window(nums, target):
    left = total = 0
    best = len(nums) + 1
    for right, value in enumerate(nums):
        total += value
        while total >= target:
            best = min(best, right - left + 1)
            total -= nums[left]
            left += 1
    return 0 if best > len(nums) else best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
shortest_positive_window([2, 3, 1, 2, 4, 3], 7)

L2 shortest_positive_window: left = total = 0
nums=[2, 3, 1, 2, 4, 3]; target=7; left=0; total=0

L3 shortest_positive_window: best = len(nums) + 1
nums=[2, 3, 1, 2, 4, 3]; target=7; left=0; total=0; best=7

L4 shortest_positive_window: for right, value in enumerate(nums):
nums=[2, 3, 1, 2, 4, 3]; target=7; left=0; total=0; best=7; right=0; value=2

L5 shortest_positive_window: total += value
nums=[2, 3, 1, 2, 4, 3]; target=7; left=0; total=2; best=7; right=0; value=2

L4 shortest_positive_window: for right, value in enumerate(nums):
nums=[2, 3, 1, 2, 4, 3]; target=7; left=0; total=2; best=7; right=1; value=3

L5 shortest_positive_window: total += value
nums=[2, 3, 1, 2, 4, 3]; target=7; left=0; total=5; best=7; right=1; value=3

L4 shortest_positive_window: for right, value in enumerate(nums):
nums=[2, 3, 1, 2, 4, 3]; target=7; left=0; total=5; best=7; right=2; value=1

L5 shortest_positive_window: total += value
nums=[2, 3, 1, 2, 4, 3]; target=7; left=0; total=6; best=7; right=2; value=1

... 15 intermediate state changes omitted ...

L9 shortest_positive_window: left += 1
nums=[2, 3, 1, 2, 4, 3]; target=7; left=4; total=7; best=3; right=5; value=3

L7 shortest_positive_window: best = min(best, right - left + 1)
nums=[2, 3, 1, 2, 4, 3]; target=7; left=4; total=7; best=2; right=5; value=3
```

```
L8 shortest_positive_window: total -= nums[left]
nums=[2, 3, 1, 2, 4, 3]; target=7; left=4; total=3; best=2; right=5; value=3

L9 shortest_positive_window: left += 1
nums=[2, 3, 1, 2, 4, 3]; target=7; left=5; total=3; best=2; right=5; value=3
```

Result: 2. The final pair [4,3] reaches seven.

```
shortest_positive_window([1, 1], 3)

L2 shortest_positive_window: left = total = 0
nums=[1, 1]; target=3; left=0; total=0

L3 shortest_positive_window: best = len(nums) + 1
nums=[1, 1]; target=3; left=0; total=0; best=3

L4 shortest_positive_window: for right, value in enumerate(nums):
nums=[1, 1]; target=3; left=0; total=0; best=3; right=0; value=1

L5 shortest_positive_window: total += value
nums=[1, 1]; target=3; left=0; total=1; best=3; right=0; value=1

L4 shortest_positive_window: for right, value in enumerate(nums):
nums=[1, 1]; target=3; left=0; total=1; best=3; right=1; value=1

L5 shortest_positive_window: total += value
nums=[1, 1]; target=3; left=0; total=2; best=3; right=1; value=1
```

Result: 0. No window reaches the target.

Why the solution is correct

For each right endpoint, repeated shrinking visits every relevant satisfying left boundary until the first failure. Further shrinking cannot restore qualification because values are positive. Hence the shortest satisfying suffix at each endpoint is recorded, including the global optimum.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Committing after subtraction can record an invalid window. Negative values invalidate the monotone shrink argument; kata 079 supplies a different structure.

Reasoning card

State: total is the active window sum. best is the shortest satisfying window encountered, initialized to an impossible length n+1.

Invariant: Inside the while-loop, the current window satisfies the target.

Decision to rehearse: While the window is valid, record before shrinking.

Finish: Return zero if best was never improved

Transfer exercise

Construct a counterexample showing why this exact loop is not valid when negative values are allowed.

049. Minimum Covering Substring

Interview | Module 05

Problem

Return the shortest substring of source containing every character occurrence required by target. Extra characters are allowed. Break equal-length ties by earliest start. Return an empty string when target is empty or no cover exists. Matching is case-sensitive.

Examples

Example 1. Every required character is covered in four positions.

```
minimum_cover("ADOBECODEBANC", "ABC")
=> 'BANC'
```

Example 2. One occurrence cannot satisfy two required copies.

```
minimum_cover("a", "aa")
=> ''
```

Example 3. The empty target has the designated empty cover.

```
minimum_cover("abc", "")
=> ''
```

Why this mechanism works

Track outstanding demand rather than merely distinct matches. A negative remaining count represents surplus copies in the window. The total missing occurrence count reaches zero exactly when all demand is covered. Record a valid window before removing its left character, then see whether that removal creates new unmet demand.

State and boundaries

need stores required count minus current-window count. missing counts unfulfilled required occurrences; negative need entries indicate surplus.

Protocol

1. Count target demand and total missing.

2. Add each incoming character, reducing missing only if it was still needed.

3. While missing is zero, record a strict length improvement.

4. Release the left character.

5. Increase missing if its remaining demand becomes positive.

Visualization

```
target ABC: missing 3
ADOBEC -> missing 0, first cover
shrink/expand -> BANC, length 4
```

Python implementation

```
from collections import Counter

def minimum_cover(source, target):
    if not target:
        return ''
    need = Counter(target)
    missing = len(target)
    left = best_start = 0
    best_length = len(source) + 1
    for right, char in enumerate(source):
        if need[char] > 0:
            missing -= 1
        need[char] -= 1
        while missing == 0:
            length = right - left + 1
            if length < best_length:
                best_start, best_length = left, length
            old = source[left]
            need[old] += 1
            if need[old] > 0:
                missing += 1
            left += 1
    if best_length > len(source):
        return ''
    return source[best_start:best_start + best_length]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
minimum_cover("ADOBECODEBANC", "ABC")

L4 minimum_cover: if not target:
source='ADOBECODEBANC'; target='ABC'

L6 minimum_cover: need = Counter(target)
source='ADOBECODEBANC'; target='ABC'; need={'A':1, 'B':1, 'C':1}

L7 minimum_cover: missing = len(target)
source='ADOBECODEBANC'; target='ABC'; need={'A':1, 'B':1, 'C':1}; missing=3

L8 minimum_cover: left = best_start = 0
source='ADOBECODEBANC'; target='ABC'; need={'A':1, 'B':1, 'C':1}; missing=3; left=0;
    best_start=0

L9 minimum_cover: best_length = len(source) + 1
source='ADOBECODEBANC'; target='ABC'; need={'A':1, 'B':1, 'C':1}; missing=3; left=0;
```

```
    best_start=0; best_length=14

L10 minimum_cover: for right, char in enumerate(source):
source='ADOBECODEBANC'; target='ABC'; need={'A':1, 'B':1, 'C':1}; missing=3; left=0;
    best_start=0; best_length=14; right=0; char='A'

L12 minimum_cover: missing -= 1
source='ADOBECODEBANC'; target='ABC'; need={'A':1, 'B':1, 'C':1}; missing=2; left=0;
    best_start=0; best_length=14; right=0; char='A'

L13 minimum_cover: need[char] -= 1
source='ADOBECODEBANC'; target='ABC'; need={'A':0, 'B':1, 'C':1}; missing=2; left=0;
    best_start=0; best_length=14; right=0; char='A'

... 66 intermediate state changes omitted ...

L18 minimum_cover: old = source[left]
source='ADOBECODEBANC'; target='ABC'; need={'A':0, 'B':0, 'C':0, 'D':0, 'O':0, ...};
    missing=0; left=9; best_start=9; best_length=4; right=12; char='C'; length=4; old='B'

L19 minimum_cover: need[old] += 1
source='ADOBECODEBANC'; target='ABC'; need={'A':0, 'B':1, 'C':0, 'D':0, 'O':0, ...};
    missing=0; left=9; best_start=9; best_length=4; right=12; char='C'; length=4; old='B'

L21 minimum_cover: missing += 1
source='ADOBECODEBANC'; target='ABC'; need={'A':0, 'B':1, 'C':0, 'D':0, 'O':0, ...};
    missing=1; left=9; best_start=9; best_length=4; right=12; char='C'; length=4; old='B'

L22 minimum_cover: left += 1
source='ADOBECODEBANC'; target='ABC'; need={'A':0, 'B':1, 'C':0, 'D':0, 'O':0, ...};
    missing=1; left=10; best_start=9; best_length=4; right=12; char='C'; length=4; old='B'
```

Result: 'BANC'. Every required character is covered in four positions.

```
minimum_cover("a", "aa")

L4 minimum_cover: if not target:
source='a'; target='aa'

L6 minimum_cover: need = Counter(target)
source='a'; target='aa'; need={'a':2}

L7 minimum_cover: missing = len(target)
source='a'; target='aa'; need={'a':2}; missing=2

L8 minimum_cover: left = best_start = 0
source='a'; target='aa'; need={'a':2}; missing=2; left=0; best_start=0

L9 minimum_cover: best_length = len(source) + 1
source='a'; target='aa'; need={'a':2}; missing=2; left=0; best_start=0; best_length=2

L10 minimum_cover: for right, char in enumerate(source):
source='a'; target='aa'; need={'a':2}; missing=2; left=0; best_start=0; best_length=2;
    right=0; char='a'

L12 minimum_cover: missing -= 1
source='a'; target='aa'; need={'a':2}; missing=1; left=0; best_start=0; best_length=2;
    right=0; char='a'

L13 minimum_cover: need[char] -= 1
source='a'; target='aa'; need={'a':1}; missing=1; left=0; best_start=0; best_length=2;
    right=0; char='a'
```

Result: ''. One occurrence cannot satisfy two required copies.

Why the solution is correct

Ingestion fulfills one missing occurrence exactly when prior demand is positive. Releasing a character breaks coverage exactly when updated demand becomes positive. The shrink loop therefore records valid candidates through the last satisfying boundary; strict replacement preserves the earliest equal-length answer.

Time and space

O(n+m) expected time and O(u) frequency space, plus the returned substring; u includes encountered source characters in this implementation.

Common mistakes

Counting only distinct characters misses repeated demand such as AABC. Testing validity after releasing a required character can save an invalid cover.

Reasoning card

State: need stores required count minus current-window count. missing counts unfulfilled required occurrences; negative need entries indicate surplus.

Invariant: formed accurately counts satisfied character requirements.

Decision to rehearse: Expand until all requirements are met; shrink while still valid.

Finish: Increase missing if its remaining demand becomes positive

Transfer exercise

Return all shortest covers rather than the earliest one, carefully separating strict improvements from ties.

050. Count Subarrays with Exactly K Distinct

Interview | Module 05

Problem

Count nonempty contiguous subarrays containing exactly k distinct integer values. Repeated values affect frequencies but not the distinct count. Return zero for k<=0 and for empty input. Count intervals by their indices, even when their contents are equal.

Examples

Example 1. Seven index intervals have exactly two distinct values.

```
subarrays_exactly_k([1, 2, 1, 2, 3], 2)
=> 7
```

Example 2. Both singletons and the two-element interval count.

```
subarrays_exactly_k([1, 1], 1)
=> 3
```

Example 3. No nonempty intervals exist.

```
subarrays_exactly_k([], 2)
=> 0
```

Why this mechanism works

Exactly-k validity is awkward to preserve directly, but at-most-k is monotone under removing values. Every interval counted by atMost(k) but not by atMost(k-1) has exactly k distinct values. For each repaired at-most window ending at right, every start from left through right is valid, contributing right-left+1 intervals.

State and boundaries

The helper's counts represent the active window. left is its earliest valid start after repair. total counts all valid intervals by their right endpoint.

Protocol

1. Define an at-most helper.

2. Expand and shrink until the distinct count fits the limit.

3. Add the number of valid suffixes ending here.

4. Compute atMost(k)-atMost(k-1).

Visualization

```
atMost(2) intervals
  minus atMost(1) intervals
  equals exactly two distinct
valid starts for fixed right: left..right
```

Python implementation

```python
def subarrays_exactly_k(nums, k):
    def at_most(limit):
        if limit <= 0:
            return 0
        counts = {}
        left = total = 0
        for right, value in enumerate(nums):
            counts[value] = counts.get(value, 0) + 1
            while len(counts) > limit:
                old = nums[left]
                counts[old] -= 1
                if counts[old] == 0:
                    del counts[old]
                left += 1
            total += right - left + 1
        return total
    return 0 if k <= 0 else at_most(k) - at_most(k - 1)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
subarrays_exactly_k([1, 2, 1, 2, 3], 2)

L2 subarrays_exactly_k: def at_most(limit):
nums=[1, 2, 1, 2, 3]; k=2

L3 at_most: if limit <= 0:
limit=2; nums=[1, 2, 1, 2, 3]

L5 at_most: counts = {}
limit=2; nums=[1, 2, 1, 2, 3]; counts={}

L6 at_most: left = total = 0
limit=2; nums=[1, 2, 1, 2, 3]; counts={}; left=0; total=0

L7 at_most: for right, value in enumerate(nums):
limit=2; nums=[1, 2, 1, 2, 3]; counts={}; left=0; total=0; right=0; value=1

L8 at_most: counts[value] = counts.get(value, 0) + 1
limit=2; nums=[1, 2, 1, 2, 3]; counts={1:1}; left=0; total=0; right=0; value=1

L15 at_most: total += right - left + 1
limit=2; nums=[1, 2, 1, 2, 3]; counts={1:1}; left=0; total=1; right=0; value=1

L7 at_most: for right, value in enumerate(nums):
limit=2; nums=[1, 2, 1, 2, 3]; counts={1:1}; left=0; total=1; right=1; value=2

... 51 intermediate state changes omitted ...
```

```
L11 at_most: counts[old] -= 1
limit=1; nums=[1, 2, 1, 2, 3]; counts={2:0, 3:1}; left=3; total=4; right=4; value=3; old=2

L13 at_most: del counts[old]
limit=1; nums=[1, 2, 1, 2, 3]; counts={3:1}; left=3; total=4; right=4; value=3; old=2

L14 at_most: left += 1
limit=1; nums=[1, 2, 1, 2, 3]; counts={3:1}; left=4; total=4; right=4; value=3; old=2

L15 at_most: total += right - left + 1
limit=1; nums=[1, 2, 1, 2, 3]; counts={3:1}; left=4; total=5; right=4; value=3; old=2
```

Result: 7. Seven index intervals have exactly two distinct values.

```
subarrays_exactly_k([1, 1], 1)

L2 subarrays_exactly_k: def at_most(limit):
nums=[1, 1]; k=1

L3 at_most: if limit <= 0:
limit=1; nums=[1, 1]

L5 at_most: counts = {}
limit=1; nums=[1, 1]; counts={}

L6 at_most: left = total = 0
limit=1; nums=[1, 1]; counts={}; left=0; total=0

L7 at_most: for right, value in enumerate(nums):
limit=1; nums=[1, 1]; counts={}; left=0; total=0; right=0; value=1

L8 at_most: counts[value] = counts.get(value, 0) + 1
limit=1; nums=[1, 1]; counts={1:1}; left=0; total=0; right=0; value=1

L15 at_most: total += right - left + 1
limit=1; nums=[1, 1]; counts={1:1}; left=0; total=1; right=0; value=1

L7 at_most: for right, value in enumerate(nums):
limit=1; nums=[1, 1]; counts={1:1}; left=0; total=1; right=1; value=1

L8 at_most: counts[value] = counts.get(value, 0) + 1
limit=1; nums=[1, 1]; counts={1:2}; left=0; total=1; right=1; value=1

L15 at_most: total += right - left + 1
limit=1; nums=[1, 1]; counts={1:2}; left=0; total=3; right=1; value=1

L3 at_most: if limit <= 0:
limit=0; nums=[1, 1]
```

Result: 3. Both singletons and the two-element interval count.

Why the solution is correct

All suffixes of an at-most window remain valid, while starts before the minimal repaired left boundary are invalid for this endpoint. Thus the helper counts each qualifying interval once. Subtracting the nested sets leaves exactly those with k distinct values.

Time and space

O(n) expected time across two scans and O(u) auxiliary space.

Common mistakes

Adding one per window counts endpoints rather than all valid intervals. The zero or negative limit must return zero without entering a repair loop that cannot become valid.

Reasoning card

State: The helper's counts represent the active window. left is its earliest valid start after repair. total counts all valid intervals by their right endpoint.

Invariant: Exactly-k equals the difference of two monotone counts.

Decision to rehearse: Compute at-most-k minus at-most-(k-1).

Finish: Compute atMost(k)-atMost(k-1)

Transfer exercise

Use the same difference-of-at-most idea to count binary subarrays with an exact number of ones.

Module 06
Prefix Sums, Prefix State, and Difference Arrays

Reason with array boundaries rather than repeatedly recomputing ranges.

Some interval questions can be answered without moving a window at all. Prefix summaries turn interval work into a difference or a relationship between two boundaries. The empty prefix is an actual state that often needs to be seeded. Difference arrays reverse the idea: record changes at boundaries and reconstruct their effect later.

The ten-step progression

051. Build a Prefix-Sum Array

052. Answer Static Range-Sum Queries

053. Pivot Index

054. Count Subarrays Summing to K

055. Longest Subarray Summing to K

056. Equal Zeros and Ones

057. Product Except Self

058. Range Addition with Difference Array

059. Flight Bookings

060. Two-Dimensional Prefix Queries

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

051. Build a Prefix-Sum Array

Foundation | Module 06

Problem

Build a boundary-based prefix-sum list for integers nums. Return n+1 values with prefix[0]=0 and prefix[i] equal to sum(nums[:i]). The extra leading zero represents the empty prefix; do not mutate the input.

Examples

Example 1. The leading zero belongs to boundary zero.

```
prefix_sums([2, -1, 3, 4])
=> [0, 2, 1, 4, 8]
```

Example 2. Even an empty array has one boundary.

```
prefix_sums([])
=> [0]
```

Example 3. A singleton has two boundaries.

```
prefix_sums([5])
=> [0, 5]
```

Why this mechanism works

Attach sums to boundaries between cells rather than to cells themselves. Then every half-open range [left,right) has the uniform expression prefix[right]-prefix[left], including ranges starting at zero. The leading zero removes a special case from every later query.

State and boundaries

prefix has one entry for each processed boundary. Its last value is the sum of all consumed elements.

Protocol

1. Start prefix with zero.

2. For each value append prefix[-1]+value.

3. Return the complete boundary array.

Visualization

```
cells:       2   -1    3    4
boundaries: 0    1    2    3    4
prefix:     0    2    1    4    8
```

Python implementation

```
def prefix_sums(nums):
    prefix = [0]
    for value in nums:
        prefix.append(prefix[-1] + value)
    return prefix
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
prefix_sums([2, -1, 3, 4])

L2 prefix_sums: prefix = [0]
nums=[2, -1, 3, 4]; prefix=[0]

L3 prefix_sums: for value in nums:
nums=[2, -1, 3, 4]; prefix=[0]; value=2

L4 prefix_sums: prefix.append(prefix[-1] + value)
nums=[2, -1, 3, 4]; prefix=[0, 2]; value=2

L3 prefix_sums: for value in nums:
nums=[2, -1, 3, 4]; prefix=[0, 2]; value=-1

L4 prefix_sums: prefix.append(prefix[-1] + value)
nums=[2, -1, 3, 4]; prefix=[0, 2, 1]; value=-1

L3 prefix_sums: for value in nums:
nums=[2, -1, 3, 4]; prefix=[0, 2, 1]; value=3

L4 prefix_sums: prefix.append(prefix[-1] + value)
nums=[2, -1, 3, 4]; prefix=[0, 2, 1, 4]; value=3

L3 prefix_sums: for value in nums:
nums=[2, -1, 3, 4]; prefix=[0, 2, 1, 4]; value=4

L4 prefix_sums: prefix.append(prefix[-1] + value)
nums=[2, -1, 3, 4]; prefix=[0, 2, 1, 4, 8]; value=4
```

Result: [0, 2, 1, 4, 8]. The leading zero belongs to boundary zero.

```
prefix_sums([])

L2 prefix_sums: prefix = [0]
nums=[]; prefix=[0]
```

Result: [0]. Even an empty array has one boundary.

Why the solution is correct

The zero entry correctly represents no elements. Each append extends the preceding prefix by one value, establishing the next boundary sum. Induction gives the required meaning at every index.

Time and space

O(n) time and O(n) returned space.

Common mistakes

Returning n rather than n+1 entries mixes cell-based and boundary-based definitions. Negative numbers do not invalidate prefix subtraction.

Reasoning card

State: prefix has one entry for each processed boundary. Its last value is the sum of all consumed elements.

Invariant: prefix[i] is the sum of the first i elements.

Decision to rehearse: Define whether the current element is included before or after each boundary.

Finish: Return the complete boundary array

Transfer exercise

Build prefix counts of odd values by replacing each incoming integer with its Boolean oddness.

052. Answer Static Range-Sum Queries

Foundation | Module 06

Problem

Given integers nums and valid queries (left,right) satisfying 0<=left<=right<=n, return each sum over nums[left:right]. Queries use half-open boundaries and may be empty. Preprocess once; answer each query in constant time.

Examples

Example 1. The first range contains -1 and 3; the last is empty.

```
range_sums([2, -1, 3, 4], [(1, 3), (0, 4), (2, 2)])
=> [2, 8, 0]
```

Example 2. The empty array still supports its empty range.

```
range_sums([], [(0, 0)])
=> [0]
```

Example 3. No queries require no answers.

```
range_sums([5], [])
=> []
```

Why this mechanism works

The prefix through right contains the desired range plus everything before left. Subtracting the prefix through left cancels that unwanted part, regardless of signs. Reusing one prefix list amortizes preprocessing over all queries.

State and boundaries

prefix[i] is the sum before index i. Each query reads two immutable boundary values.

Protocol

1. Build a leading-zero prefix array.

2. For each query subtract prefix[left] from prefix[right].

3. Preserve query order in the output.

Visualization

```
prefix[right] = before-left + requested range
prefix[left]  = before-left
subtract      = requested range
```

Python implementation

```
def range_sums(nums, queries):
    prefix = [0]
    for value in nums:
        prefix.append(prefix[-1] + value)
    return [prefix[right] - prefix[left] for left, right in queries]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
range_sums([2, -1, 3, 4], [(1, 3), (0, 4), (2, 2)])

L2 range_sums: prefix = [0]
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0]

L3 range_sums: for value in nums:
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0]; value=2

L4 range_sums: prefix.append(prefix[-1] + value)
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2]; value=2

L3 range_sums: for value in nums:
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2]; value=-1

L4 range_sums: prefix.append(prefix[-1] + value)
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2, 1]; value=-1

L3 range_sums: for value in nums:
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2, 1]; value=3

L4 range_sums: prefix.append(prefix[-1] + value)
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2, 1, 4]; value=3

L3 range_sums: for value in nums:
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2, 1, 4]; value=4

... 1 intermediate state changes omitted ...

L5 range_sums: return [prefix[right] - prefix[left] for left, right in queries]
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2, 1, 4, 8]; value=4; left=1;
     right=3

L5 range_sums: return [prefix[right] - prefix[left] for left, right in queries]
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2, 1, 4, 8]; value=4; left=0;
     right=4

L5 range_sums: return [prefix[right] - prefix[left] for left, right in queries]
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2, 1, 4, 8]; value=4; left=2;
     right=2

L5 range_sums: return [prefix[right] - prefix[left] for left, right in queries]
nums=[2, -1, 3, 4]; queries=[(1, 3), (0, 4), (2, 2)]; prefix=[0, 2, 1, 4, 8]; value=4
```

Result: [2, 8, 0]. The first range contains -1 and 3; the last is empty.

```
range_sums([], [(0, 0)])

L2 range_sums: prefix = [0]
nums=[]; queries=[(0, 0)]; prefix=[0]
```

```
L5 range_sums: return [prefix[right] - prefix[left] for left, right in queries]
nums=[]; queries=[(0, 0)]; prefix=[0]; left=0; right=0

L5 range_sums: return [prefix[right] - prefix[left] for left, right in queries]
nums=[]; queries=[(0, 0)]; prefix=[0]
```

Result: [0]. The empty array still supports its empty range.

Why the solution is correct

Both prefix values include exactly the same positions before left, so subtraction cancels them. Positions from left through right-1 occur only in the larger prefix expression and remain once. Equal boundaries therefore yield zero.

Time and space

O(n+q) time and O(n+q) space including q answers; each query is O(1) after preprocessing.

Common mistakes

Using right+1 would reinterpret the query as inclusive. Rebuilding the prefix per query defeats the intended preprocessing benefit.

Reasoning card

State: prefix[i] is the sum before index i. Each query reads two immutable boundary values.

Invariant: prefix[right]-prefix[left] equals the requested range.

Decision to rehearse: Subtract cumulative boundaries.

Finish: Preserve query order in the output

Transfer exercise

Allow point updates between queries. Explain why a fixed prefix list becomes stale and what a Fenwick tree would change.

053. Pivot Index

Foundation | Module 06

Problem

Return the earliest index whose elements strictly to its left and strictly to its right have equal sums. The pivot value is excluded from both sides. Empty input returns -1. Negative values are allowed.

Examples

Example 1. Both strict sides sum to eleven.

```
pivot_index([1, 7, 3, 6, 5, 6])
=> 3
```

Example 2. Both sides of the singleton are empty sums.

```
pivot_index([9])
=> 0
```

Example 3. There is no candidate index.

```
pivot_index([])
=> -1
```

Why this mechanism works

With total sum known, the right sum can be derived from total-left-current. Keep only the sum strictly before the candidate and test before adding the candidate to it. This is a commit-before-ingest variation on the basic scan.

State and boundaries

left is the sum at indices smaller than i. total is the full input sum.

Protocol

1. Compute total.
2. Start left at zero.
3. Test left == total-left-value.
4. Return the first matching index.
5. Add value to left only after rejection.

Visualization

```
[1,7,3] | 6 | [5,6]
left=11        right=11
pivot index=3
```

Python implementation

```
def pivot_index(nums):
    total = sum(nums)
    left = 0
    for i, value in enumerate(nums):
        if left == total - left - value:
            return i
        left += value
    return -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
pivot_index([1, 7, 3, 6, 5, 6])

L2 pivot_index: total = sum(nums)
nums=[1, 7, 3, 6, 5, 6]; total=28

L3 pivot_index: left = 0
nums=[1, 7, 3, 6, 5, 6]; total=28; left=0

L4 pivot_index: for i, value in enumerate(nums):
nums=[1, 7, 3, 6, 5, 6]; total=28; left=0; i=0; value=1

L7 pivot_index: left += value
nums=[1, 7, 3, 6, 5, 6]; total=28; left=1; i=0; value=1

L4 pivot_index: for i, value in enumerate(nums):
nums=[1, 7, 3, 6, 5, 6]; total=28; left=1; i=1; value=7

L7 pivot_index: left += value
nums=[1, 7, 3, 6, 5, 6]; total=28; left=8; i=1; value=7

L4 pivot_index: for i, value in enumerate(nums):
nums=[1, 7, 3, 6, 5, 6]; total=28; left=8; i=2; value=3

L7 pivot_index: left += value
nums=[1, 7, 3, 6, 5, 6]; total=28; left=11; i=2; value=3

L4 pivot_index: for i, value in enumerate(nums):
nums=[1, 7, 3, 6, 5, 6]; total=28; left=11; i=3; value=6
```

Result: 3. Both strict sides sum to eleven.

```
pivot_index([9])

L2 pivot_index: total = sum(nums)
nums=[9]; total=9

L3 pivot_index: left = 0
nums=[9]; total=9; left=0
```

```
L4 pivot_index: for i, value in enumerate(nums):
nums=[9]; total=9; left=0; i=0; value=9
```

Result: 0. Both sides of the singleton are empty sums.

Why the solution is correct

At each test, subtracting the left prefix and current value from total leaves exactly the right suffix. The equality is therefore the pivot condition. Increasing-index tests guarantee the earliest pivot is returned.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Adding the current value before the test includes it on the wrong side. Treating missing left or right sides as invalid wrongly rejects endpoint pivots.

Reasoning card

State: left is the sum at indices smaller than i. total is the full input sum.

Invariant: Before index i, left_sum equals values strictly left of i.

Decision to rehearse: Derive right sum from total and processed prefix.

Finish: Add value to left only after rejection

Transfer exercise

Return every pivot index, preserving the distinction between checking and extending the left prefix.

054. Count Subarrays Summing to K

Foundation | Module 06

Problem

Count nonempty contiguous subarrays whose integer sum equals k. Values may be positive, negative or zero. Count intervals by endpoints, including equal-content intervals at different positions. Empty input returns zero.

Examples

Example 1. Both singleton ones and the whole array qualify.

```
count_sum_k([1, -1, 1], 1)
=> 3
```

Example 2. Two singletons and one pair are nonempty zero-sum intervals.

```
count_sum_k([0, 0], 0)
=> 3
```

Example 3. The seeded empty prefix is not itself an interval.

```
count_sum_k([], 0)
=> 0
```

Why this mechanism works

A range ending at boundary r sums to k exactly when an earlier prefix equals prefix[r]-k. Store frequencies of earlier prefix sums, because several boundaries may have the same sum. Seed the empty prefix with frequency one so ranges starting at index zero are counted.

State and boundaries

total is the current prefix sum. frequency contains only earlier boundary sums when queried. answer counts ranges ending at processed boundaries.

Protocol

1. Seed frequency[0]=1.
2. Extend total with the incoming value.
3. Add frequency[total-k] to answer.
4. Record the current total afterward.

5. Return answer.

Visualization

```
prefix sums for [1,-1,1]: 0,1,0,1
current 1 needs earlier 0 for k=1
there can be more than one earlier match
```

Python implementation

```
def count_sum_k(nums, k):
    frequency = {0: 1}
    total = answer = 0
    for value in nums:
        total += value
        answer += frequency.get(total - k, 0)
        frequency[total] = frequency.get(total, 0) + 1
    return answer
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
count_sum_k([1, -1, 1], 1)

L2 count_sum_k: frequency = {0: 1}
nums=[1, -1, 1]; k=1; frequency={0:1}

L3 count_sum_k: total = answer = 0
nums=[1, -1, 1]; k=1; frequency={0:1}; total=0; answer=0

L4 count_sum_k: for value in nums:
nums=[1, -1, 1]; k=1; frequency={0:1}; total=0; answer=0; value=1

L5 count_sum_k: total += value
nums=[1, -1, 1]; k=1; frequency={0:1}; total=1; answer=0; value=1

L6 count_sum_k: answer += frequency.get(total - k, 0)
nums=[1, -1, 1]; k=1; frequency={0:1}; total=1; answer=1; value=1

L7 count_sum_k: frequency[total] = frequency.get(total, 0) + 1
nums=[1, -1, 1]; k=1; frequency={0:1, 1:1}; total=1; answer=1; value=1

L4 count_sum_k: for value in nums:
nums=[1, -1, 1]; k=1; frequency={0:1, 1:1}; total=1; answer=1; value=-1

L5 count_sum_k: total += value
nums=[1, -1, 1]; k=1; frequency={0:1, 1:1}; total=0; answer=1; value=-1

... 1 intermediate state changes omitted ...

L4 count_sum_k: for value in nums:
nums=[1, -1, 1]; k=1; frequency={0:2, 1:1}; total=0; answer=1; value=1

L5 count_sum_k: total += value
nums=[1, -1, 1]; k=1; frequency={0:2, 1:1}; total=1; answer=1; value=1

L6 count_sum_k: answer += frequency.get(total - k, 0)
nums=[1, -1, 1]; k=1; frequency={0:2, 1:1}; total=1; answer=3; value=1
```

```
L7 count_sum_k: frequency[total] = frequency.get(total, 0) + 1
nums=[1, -1, 1]; k=1; frequency={0:2, 1:2}; total=1; answer=3; value=1
```

Result: 3. Both singleton ones and the whole array qualify.

```
count_sum_k([0, 0], 0)

L2 count_sum_k: frequency = {0: 1}
nums=[0, 0]; k=0; frequency={0:1}

L3 count_sum_k: total = answer = 0
nums=[0, 0]; k=0; frequency={0:1}; total=0; answer=0

L4 count_sum_k: for value in nums:
nums=[0, 0]; k=0; frequency={0:1}; total=0; answer=0; value=0

L6 count_sum_k: answer += frequency.get(total - k, 0)
nums=[0, 0]; k=0; frequency={0:1}; total=0; answer=1; value=0

L7 count_sum_k: frequency[total] = frequency.get(total, 0) + 1
nums=[0, 0]; k=0; frequency={0:2}; total=0; answer=1; value=0

L6 count_sum_k: answer += frequency.get(total - k, 0)
nums=[0, 0]; k=0; frequency={0:2}; total=0; answer=3; value=0

L7 count_sum_k: frequency[total] = frequency.get(total, 0) + 1
nums=[0, 0]; k=0; frequency={0:3}; total=0; answer=3; value=0
```

Result: 3. Two singletons and one pair are nonempty zero-sum intervals.

Why the solution is correct

Each earlier matching prefix identifies exactly one nonempty interval ending here. Querying before insertion prevents using the current boundary as both endpoints. Each interval has one right boundary, so it is counted exactly once.

Time and space

O(n) expected time and O(n) auxiliary space.

Common mistakes

A set loses multiple equal-prefix boundaries. Inserting before querying overcounts empty ranges when k=0.

Reasoning card

State: total is the current prefix sum. frequency contains only earlier boundary sums when queried. answer counts ranges ending at processed boundaries.

Invariant: The frequency map contains only earlier prefix boundaries.

Decision to rehearse: Extend prefix, query earlier prefix-k, then store current prefix.

Finish: Return answer

Transfer exercise

Count sums divisible by a positive modulus by grouping prefix sums with equal remainders.

055. Longest Subarray Summing to K

Bridge | Module 06

Problem

Return the maximum length of a nonempty contiguous subarray summing to k, or zero if none exists. Integers may have either sign. Do not mutate the input.

Examples

Example 1. The first four values sum to three.

```
longest_sum_k([1, -1, 5, -2, 3], 3)
=> 4
```

Example 2. Keeping boundary zero allows the full length.

```
longest_sum_k([0, 0, 0], 0)
=> 3
```

Example 3. No nonempty range exists.

```
longest_sum_k([], 1)
=> 0
```

Why this mechanism works

The same prefix difference identifies valid starts, but the objective changes the remembered information. To maximize right-left for a fixed right boundary, keep the earliest occurrence of the needed prefix sum. Frequencies are unnecessary, and overwriting an early boundary can destroy the best length.

State and boundaries

earliest maps a prefix sum to its first boundary index. right counts consumed elements, beginning at one.

Protocol

1. Seed sum zero at boundary zero.
2. Extend the prefix.
3. If total-k exists, compare right-earliest[total-k] with best.
4. Store total only on its first occurrence.
5. Return best.

Visualization

```
[1,-1,5,-2,3], k=3
prefix at boundary 0: 0
prefix at boundary 4: 3
length=4-0=4
```

Python implementation

```
def longest_sum_k(nums, k):
    earliest = {0: 0}
    total = best = 0
    for right, value in enumerate(nums, 1):
        total += value
        if total - k in earliest:
            best = max(best, right - earliest[total - k])
        earliest.setdefault(total, right)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
longest_sum_k([1, -1, 5, -2, 3], 3)

L2 longest_sum_k: earliest = {0: 0}
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0}

L3 longest_sum_k: total = best = 0
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0}; total=0; best=0

L4 longest_sum_k: for right, value in enumerate(nums, 1):
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0}; total=0; best=0; right=1; value=1

L5 longest_sum_k: total += value
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0}; total=1; best=0; right=1; value=1

L8 longest_sum_k: earliest.setdefault(total, right)
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0, 1:1}; total=1; best=0; right=1; value=1

L4 longest_sum_k: for right, value in enumerate(nums, 1):
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0, 1:1}; total=1; best=0; right=2; value=-1

L5 longest_sum_k: total += value
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0, 1:1}; total=0; best=0; right=2; value=-1

L4 longest_sum_k: for right, value in enumerate(nums, 1):
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0, 1:1}; total=0; best=0; right=3; value=5

... 5 intermediate state changes omitted ...

L8 longest_sum_k: earliest.setdefault(total, right)
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0, 1:1, 5:3, 3:4}; total=3; best=4; right=4; value=-2

L4 longest_sum_k: for right, value in enumerate(nums, 1):
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0, 1:1, 5:3, 3:4}; total=3; best=4; right=5; value=3

L5 longest_sum_k: total += value
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0, 1:1, 5:3, 3:4}; total=6; best=4; right=5; value=3
```

```
L8 longest_sum_k: earliest.setdefault(total, right)
nums=[1, -1, 5, -2, 3]; k=3; earliest={0:0, 1:1, 5:3, 3:4, 6:5}; total=6; best=4; right=5;
    value=3
```

Result: 4. The first four values sum to three.

```
longest_sum_k([0, 0, 0], 0)

L2 longest_sum_k: earliest = {0: 0}
nums=[0, 0, 0]; k=0; earliest={0:0}

L3 longest_sum_k: total = best = 0
nums=[0, 0, 0]; k=0; earliest={0:0}; total=0; best=0

L4 longest_sum_k: for right, value in enumerate(nums, 1):
nums=[0, 0, 0]; k=0; earliest={0:0}; total=0; best=0; right=1; value=0

L7 longest_sum_k: best = max(best, right - earliest[total - k])
nums=[0, 0, 0]; k=0; earliest={0:0}; total=0; best=1; right=1; value=0

L4 longest_sum_k: for right, value in enumerate(nums, 1):
nums=[0, 0, 0]; k=0; earliest={0:0}; total=0; best=1; right=2; value=0

L7 longest_sum_k: best = max(best, right - earliest[total - k])
nums=[0, 0, 0]; k=0; earliest={0:0}; total=0; best=2; right=2; value=0

L4 longest_sum_k: for right, value in enumerate(nums, 1):
nums=[0, 0, 0]; k=0; earliest={0:0}; total=0; best=2; right=3; value=0

L7 longest_sum_k: best = max(best, right - earliest[total - k])
nums=[0, 0, 0]; k=0; earliest={0:0}; total=0; best=3; right=3; value=0
```

Result: 3. Keeping boundary zero allows the full length.

Why the solution is correct

All boundaries with the needed prefix yield valid intervals, and the earliest yields the longest for this endpoint. Retaining it cannot hurt any later endpoint. Considering every endpoint therefore finds the longest qualifying interval.

Time and space

O(n) expected time and O(n) auxiliary space.

Common mistakes

Replacing the earliest boundary with a later one shortens future candidates. Cell indices and boundary indices must not be mixed in the length formula.

Reasoning card

State: earliest maps a prefix sum to its first boundary index. right counts consumed elements, beginning at one.

Invariant: Earliest occurrence maximizes any later matching span.

Decision to rehearse: Store the earliest index of each prefix sum.

Finish: Return best

Transfer exercise

Find the shortest sum-k interval using remembered prefix positions. Which occurrence should be retained now?

056. Equal Zeros and Ones

Bridge | Module 06

Problem

For a binary list, return (longest_length, count) over all nonempty subarrays containing equally many zeros and ones. Empty input returns (0,0). This expanded kata includes both objectives suggested by the compressed source.

Examples

Example 1. Two length-two intervals are balanced.

```
equal_binary_subarrays([0, 1, 0])
=> (2, 2)
```

Example 2. No interval contains a one.

```
equal_binary_subarrays([0, 0])
=> (0, 0)
```

Example 3. Both objectives have empty-input neutral answers.

```
equal_binary_subarrays([])
=> (0, 0)
```

Why this mechanism works

Map a zero to -1 and a one to +1. Equal counts then mean zero transformed sum, so two equal prefix balances delimit a valid interval. Keep both the earliest boundary for maximum length and the frequency of boundaries for the count.

State and boundaries

balance is ones minus zeros. earliest supports longest intervals; frequency supports counting all intervals.

Protocol

1. Seed balance zero at boundary zero with frequency one.
2. Apply +1 or -1 per bit.
3. Count earlier equal balances.
4. Compare against the earliest equal boundary.
5. Preserve earliest and increment frequency.

Visualization

```
bits:     0  1  0
balance:0 -1  0 -1
equal boundaries -> intervals [0,2), [1,3)
```

Python implementation

```
def equal_binary_subarrays(nums):
    earliest, frequency = {0: 0}, {0: 1}
    balance = longest = count = 0
    for right, bit in enumerate(nums, 1):
        balance += 1 if bit == 1 else -1
        count += frequency.get(balance, 0)
        if balance in earliest:
            longest = max(longest, right - earliest[balance])
        earliest.setdefault(balance, right)
        frequency[balance] = frequency.get(balance, 0) + 1
    return longest, count
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
equal_binary_subarrays([0, 1, 0])

L2 equal_binary_subarrays: earliest, frequency = {0: 0}, {0: 1}
nums=[0, 1, 0]; earliest={0:0}; frequency={0:1}

L3 equal_binary_subarrays: balance = longest = count = 0
nums=[0, 1, 0]; earliest={0:0}; frequency={0:1}; balance=0; longest=0; count=0

L4 equal_binary_subarrays: for right, bit in enumerate(nums, 1):
nums=[0, 1, 0]; earliest={0:0}; frequency={0:1}; balance=0; longest=0; count=0; right=1; bit=0

L5 equal_binary_subarrays: balance += 1 if bit == 1 else -1
nums=[0, 1, 0]; earliest={0:0}; frequency={0:1}; balance=-1; longest=0; count=0; right=1;
    bit=0

L9 equal_binary_subarrays: earliest.setdefault(balance, right)
nums=[0, 1, 0]; earliest={0:0, -1:1}; frequency={0:1}; balance=-1; longest=0; count=0;
    right=1; bit=0

L10 equal_binary_subarrays: frequency[balance] = frequency.get(balance, 0) + 1
nums=[0, 1, 0]; earliest={0:0, -1:1}; frequency={0:1, -1:1}; balance=-1; longest=0; count=0;
    right=1; bit=0

L4 equal_binary_subarrays: for right, bit in enumerate(nums, 1):
nums=[0, 1, 0]; earliest={0:0, -1:1}; frequency={0:1, -1:1}; balance=-1; longest=0; count=0;
    right=2; bit=1

L5 equal_binary_subarrays: balance += 1 if bit == 1 else -1
nums=[0, 1, 0]; earliest={0:0, -1:1}; frequency={0:1, -1:1}; balance=0; longest=0; count=0;
    right=2; bit=1

... 3 intermediate state changes omitted ...

L4 equal_binary_subarrays: for right, bit in enumerate(nums, 1):
nums=[0, 1, 0]; earliest={0:0, -1:1}; frequency={0:2, -1:1}; balance=0; longest=2; count=1;
    right=3; bit=0
```

```
L5 equal_binary_subarrays: balance += 1 if bit == 1 else -1
nums=[0, 1, 0]; earliest={0:0, -1:1}; frequency={0:2, -1:1}; balance=-1; longest=2; count=1;
    right=3; bit=0

L6 equal_binary_subarrays: count += frequency.get(balance, 0)
nums=[0, 1, 0]; earliest={0:0, -1:1}; frequency={0:2, -1:1}; balance=-1; longest=2; count=2;
    right=3; bit=0

L10 equal_binary_subarrays: frequency[balance] = frequency.get(balance, 0) + 1
nums=[0, 1, 0]; earliest={0:0, -1:1}; frequency={0:2, -1:2}; balance=-1; longest=2; count=2;
    right=3; bit=0
```

Result: (2, 2). Two length-two intervals are balanced.

```
equal_binary_subarrays([0, 0])

L2 equal_binary_subarrays: earliest, frequency = {0: 0}, {0: 1}
nums=[0, 0]; earliest={0:0}; frequency={0:1}

L3 equal_binary_subarrays: balance = longest = count = 0
nums=[0, 0]; earliest={0:0}; frequency={0:1}; balance=0; longest=0; count=0

L4 equal_binary_subarrays: for right, bit in enumerate(nums, 1):
nums=[0, 0]; earliest={0:0}; frequency={0:1}; balance=0; longest=0; count=0; right=1; bit=0

L5 equal_binary_subarrays: balance += 1 if bit == 1 else -1
nums=[0, 0]; earliest={0:0}; frequency={0:1}; balance=-1; longest=0; count=0; right=1; bit=0

L9 equal_binary_subarrays: earliest.setdefault(balance, right)
nums=[0, 0]; earliest={0:0, -1:1}; frequency={0:1}; balance=-1; longest=0; count=0; right=1;
    bit=0

L10 equal_binary_subarrays: frequency[balance] = frequency.get(balance, 0) + 1
nums=[0, 0]; earliest={0:0, -1:1}; frequency={0:1, -1:1}; balance=-1; longest=0; count=0;
    right=1; bit=0

L4 equal_binary_subarrays: for right, bit in enumerate(nums, 1):
nums=[0, 0]; earliest={0:0, -1:1}; frequency={0:1, -1:1}; balance=-1; longest=0; count=0;
    right=2; bit=0

L5 equal_binary_subarrays: balance += 1 if bit == 1 else -1
nums=[0, 0]; earliest={0:0, -1:1}; frequency={0:1, -1:1}; balance=-2; longest=0; count=0;
    right=2; bit=0

L9 equal_binary_subarrays: earliest.setdefault(balance, right)
nums=[0, 0]; earliest={0:0, -1:1, -2:2}; frequency={0:1, -1:1}; balance=-2; longest=0;
    count=0; right=2; bit=0

L10 equal_binary_subarrays: frequency[balance] = frequency.get(balance, 0) + 1
nums=[0, 0]; earliest={0:0, -1:1, -2:2}; frequency={0:1, -1:1, -2:1}; balance=-2; longest=0;
    count=0; right=2; bit=0
```

Result: (0, 0). No interval contains a one.

Why the solution is correct

Equal prefix balances mean the intervening interval has net contribution zero, exactly equal numbers of zeros and ones. Every matching earlier boundary counts once, while the earliest maximizes length. The two summaries answer different objectives over the same valid intervals.

Time and space

O(n) expected time and O(n) auxiliary space.

Common mistakes

Treating zero as contribution zero measures the number of ones instead of balance. Overwriting earliest breaks the length result even if the count stays correct.

Reasoning card

State: balance is ones minus zeros. earliest supports longest intervals; frequency supports counting all intervals.

Invariant: The prefix balance is ones minus zeros through the current point.

Decision to rehearse: Repeated prefix balance means the middle balances to zero.

Finish: Preserve earliest and increment frequency

Transfer exercise

For three symbols, store a pair of count differences and count subarrays with equal counts of all three.

057. Product Except Self

Bridge | Module 06

Problem

Return a list where output[i] is the product of every input value except nums[i], without division. Integers may include zero and negatives. Empty input returns []; a singleton returns [1], the empty product. Do not modify nums.

Examples

Example 1. Each answer combines the two strict sides.

```
product_except_self([1, 2, 3, 4])
=> [24, 12, 8, 6]
```

Example 2. Only the zero position excludes the zero factor.

```
product_except_self([0, 2, 3])
=> [6, 0, 0]
```

Example 3. The product of no remaining values is one.

```
product_except_self([9])
=> [1]
```

Why this mechanism works

Split each answer into the product strictly to its left and strictly to its right. Store left products in the output, then multiply by a running right product in reverse order. Zero requires no special branch because multiplication handles it naturally.

State and boundaries

left is the product before i in the forward pass. right is the product after i in the backward pass.

Protocol

1. Initialize output entries to one.
2. Commit the current left product before multiplying in nums[i].
3. Scan backward.
4. Multiply output[i] by the current right product before extending right.

Visualization

```
nums:   1 2 3 4
left:   1 1 2 6
right: 24 12 4 1
answer:24 12 8 6
```

Python implementation

```
def product_except_self(nums):
    output = [1] * len(nums)
    left = 1
    for i, value in enumerate(nums):
        output[i] = left
        left *= value
    right = 1
    for i in range(len(nums) - 1, -1, -1):
        output[i] *= right
        right *= nums[i]
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
product_except_self([1, 2, 3, 4])

L2 product_except_self: output = [1] * len(nums)
nums=[1, 2, 3, 4]; output=[1, 1, 1, 1]

L3 product_except_self: left = 1
nums=[1, 2, 3, 4]; output=[1, 1, 1, 1]; left=1

L4 product_except_self: for i, value in enumerate(nums):
nums=[1, 2, 3, 4]; output=[1, 1, 1, 1]; left=1; i=0; value=1

L4 product_except_self: for i, value in enumerate(nums):
nums=[1, 2, 3, 4]; output=[1, 1, 1, 1]; left=1; i=1; value=2

L6 product_except_self: left *= value
nums=[1, 2, 3, 4]; output=[1, 1, 1, 1]; left=2; i=1; value=2

L4 product_except_self: for i, value in enumerate(nums):
nums=[1, 2, 3, 4]; output=[1, 1, 1, 1]; left=2; i=2; value=3

L5 product_except_self: output[i] = left
nums=[1, 2, 3, 4]; output=[1, 1, 2, 1]; left=2; i=2; value=3

L6 product_except_self: left *= value
nums=[1, 2, 3, 4]; output=[1, 1, 2, 1]; left=6; i=2; value=3

... 9 intermediate state changes omitted ...

L9 product_except_self: output[i] *= right
nums=[1, 2, 3, 4]; output=[1, 12, 8, 6]; left=24; i=1; value=4; right=12

L10 product_except_self: right *= nums[i]
nums=[1, 2, 3, 4]; output=[1, 12, 8, 6]; left=24; i=1; value=4; right=24

L8 product_except_self: for i in range(len(nums) - 1, -1, -1):
```

```
nums=[1, 2, 3, 4]; output=[1, 12, 8, 6]; left=24; i=0; value=4; right=24

L9 product_except_self: output[i] *= right
nums=[1, 2, 3, 4]; output=[24, 12, 8, 6]; left=24; i=0; value=4; right=24
```

Result: [24, 12, 8, 6]. Each answer combines the two strict sides.

```
product_except_self([0, 2, 3])

L2 product_except_self: output = [1] * len(nums)
nums=[0, 2, 3]; output=[1, 1, 1]

L3 product_except_self: left = 1
nums=[0, 2, 3]; output=[1, 1, 1]; left=1

L4 product_except_self: for i, value in enumerate(nums):
nums=[0, 2, 3]; output=[1, 1, 1]; left=1; i=0; value=0

L6 product_except_self: left *= value
nums=[0, 2, 3]; output=[1, 1, 1]; left=0; i=0; value=0

L4 product_except_self: for i, value in enumerate(nums):
nums=[0, 2, 3]; output=[1, 1, 1]; left=0; i=1; value=2

L5 product_except_self: output[i] = left
nums=[0, 2, 3]; output=[1, 0, 1]; left=0; i=1; value=2

L4 product_except_self: for i, value in enumerate(nums):
nums=[0, 2, 3]; output=[1, 0, 1]; left=0; i=2; value=3

L5 product_except_self: output[i] = left
nums=[0, 2, 3]; output=[1, 0, 0]; left=0; i=2; value=3

... 3 intermediate state changes omitted ...

L10 product_except_self: right *= nums[i]
nums=[0, 2, 3]; output=[1, 0, 0]; left=0; i=1; value=3; right=6

L8 product_except_self: for i in range(len(nums) - 1, -1, -1):
nums=[0, 2, 3]; output=[1, 0, 0]; left=0; i=0; value=3; right=6

L9 product_except_self: output[i] *= right
nums=[0, 2, 3]; output=[6, 0, 0]; left=0; i=0; value=3; right=6

L10 product_except_self: right *= nums[i]
nums=[0, 2, 3]; output=[6, 0, 0]; left=0; i=0; value=3; right=0
```

Result: [6, 0, 0]. Only the zero position excludes the zero factor.

Why the solution is correct

The forward pass writes exactly the factors with indices smaller than i. The backward pass supplies exactly the greater indices. These disjoint factor sets exclude only i, and their product is the required answer even when some factors are zero.

Time and space

O(n) arithmetic operations; O(n) output and O(1) other state. Large integer products may have nonconstant arithmetic cost.

Common mistakes

Updating a side product before committing includes the value that must be excluded. Division is both prohibited and problematic at zero.

Reasoning card

State: left is the product before i in the forward pass. right is the product after i in the backward pass.

Invariant: Before the reverse pass, output[i] equals product strictly left of i.

Decision to rehearse: Commit left products, then multiply by a rolling right product.

Finish: Multiply output[i] by the current right product before extending right

Transfer exercise

Generalize the two-sided aggregation to an associative operation and specify its identity element.

058. Range Addition with Difference Array

Interview | Module 06

Problem

Start with n zeros and apply updates (left,right,delta), adding delta to every index in [left,right). Bounds satisfy 0<=left<=right<=n. Return the final array. Empty updates and empty ranges are valid; n may be zero.

Examples

Example 1. The exclusive right boundary turns the update off.

```
range_additions(5, [(1, 4, 2)])
=> [0, 2, 2, 2, 0]
```

Example 2. Overlapping positive and negative updates add.

```
range_additions(3, [(0, 3, 2), (1, 2, -1)])
=> [2, 1, 2]
```

Example 3. There are boundaries but no cells to emit.

```
range_additions(0, [])
=> []
```

Why this mechanism works

A range addition changes the running adjustment only at its two boundaries: turn delta on at left and off at right. Record these changes in a difference array, then integrate them with a prefix sum. Overlapping updates combine by addition automatically.

State and boundaries

difference stores changes in active adjustment at boundaries. active is the adjustment applying to the current output cell.

Protocol

1. Allocate n+1 boundary differences.
2. Add delta at left and subtract it at right.
3. Scan the first n boundaries with a running sum.
4. Emit active for each cell.

Visualization

```
add 2 on [1,4)
difference: 0,+2,0,0,-2,0
integrate:  0, 2,2,2, 0
```

Python implementation

```
def range_additions(n, updates):
    difference = [0] * (n + 1)
    for left, right, delta in updates:
        difference[left] += delta
        difference[right] -= delta
    output = []
    active = 0
    for i in range(n):
        active += difference[i]
        output.append(active)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
range_additions(5, [(1, 4, 2)])

L2 range_additions: difference = [0] * (n + 1)
n=5; updates=[(1, 4, 2)]; difference=[0, 0, 0, 0, 0, 0]

L3 range_additions: for left, right, delta in updates:
n=5; updates=[(1, 4, 2)]; difference=[0, 0, 0, 0, 0, 0]; left=1; right=4; delta=2

L4 range_additions: difference[left] += delta
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, 0, 0]; left=1; right=4; delta=2

L5 range_additions: difference[right] -= delta
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2

L6 range_additions: output = []
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2; output=[]

L7 range_additions: active = 0
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2; output=[];
     active=0

L8 range_additions: for i in range(n):
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2; output=[];
     active=0; i=0

L10 range_additions: output.append(active)
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2;
    output=[0]; active=0; i=0

... 6 intermediate state changes omitted ...

L10 range_additions: output.append(active)
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2; output=[0,
     2, 2, 2]; active=2; i=3

L8 range_additions: for i in range(n):
```

```
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2; output=[0,
     2, 2, 2]; active=2; i=4

L9 range_additions: active += difference[i]
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2; output=[0,
     2, 2, 2]; active=0; i=4

L10 range_additions: output.append(active)
n=5; updates=[(1, 4, 2)]; difference=[0, 2, 0, 0, -2, 0]; left=1; right=4; delta=2; output=[0,
     2, 2, 2, 0]; active=0; i=4
```

Result: [0, 2, 2, 2, 0]. The exclusive right boundary turns the update off.

```
range_additions(3, [(0, 3, 2), (1, 2, -1)])

L2 range_additions: difference = [0] * (n + 1)
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[0, 0, 0, 0]

L3 range_additions: for left, right, delta in updates:
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[0, 0, 0, 0]; left=0; right=3; delta=2

L4 range_additions: difference[left] += delta
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, 0, 0, 0]; left=0; right=3; delta=2

L5 range_additions: difference[right] -= delta
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, 0, 0, -2]; left=0; right=3; delta=2

L3 range_additions: for left, right, delta in updates:
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, 0, 0, -2]; left=1; right=2; delta=-1

L4 range_additions: difference[left] += delta
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, -1, 0, -2]; left=1; right=2; delta=-1

L5 range_additions: difference[right] -= delta
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, -1, 1, -2]; left=1; right=2; delta=-1

L6 range_additions: output = []
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, -1, 1, -2]; left=1; right=2; delta=-1;
    output=[]

... 6 intermediate state changes omitted ...

L10 range_additions: output.append(active)
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, -1, 1, -2]; left=1; right=2; delta=-1;
    output=[2, 1]; active=1; i=1

L8 range_additions: for i in range(n):
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, -1, 1, -2]; left=1; right=2; delta=-1;
    output=[2, 1]; active=1; i=2

L9 range_additions: active += difference[i]
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, -1, 1, -2]; left=1; right=2; delta=-1;
    output=[2, 1]; active=2; i=2

L10 range_additions: output.append(active)
n=3; updates=[(0, 3, 2), (1, 2, -1)]; difference=[2, -1, 1, -2]; left=1; right=2; delta=-1;
    output=[2, 1, 2]; active=2; i=2
```

Result: [2, 1, 2]. Overlapping positive and negative updates add.

Why the solution is correct

An update contributes delta to the prefix of differences exactly from left through right-1; at right its opposite delta cancels it. Summing contributions over updates therefore gives the required final value at every cell.

Time and space

O(n+q) time and O(n) storage for n cells and q updates.

Common mistakes

Subtracting at right-1 ends the update one cell early. The extra boundary n is needed for updates reaching the array's end.

Reasoning card

State: difference stores changes in active adjustment at boundaries. active is the adjustment applying to the current output cell.

Invariant: Running difference sum equals active range updates.

Decision to rehearse: Record changes at boundaries, then prefix-sum.

Finish: Emit active for each cell

Transfer exercise

Recover a difference array from an existing nonzero array, then apply the same update protocol.

059. Flight Bookings

Interview | Module 06

Problem

Flights are numbered 1 through n. Each booking (first,last,seats) adds seats to every flight from first through last, inclusive. Return n seat totals in flight order. Bookings have valid one-based endpoints and nonnegative seat counts; n may be zero when there are no bookings.

Examples

Example 1. Flight two receives all three bookings.

```
flight_bookings(5, [(1, 2, 10), (2, 3, 20), (2, 5, 25)])
=> [10, 55, 45, 25, 25]
```

Example 2. Both inclusive endpoints name the same flight.

```
flight_bookings(1, [(1, 1, 7)])
=> [7]
```

Example 3. No flights produce no totals.

```
flight_bookings(0, [])
=> []
```

Why this mechanism works

Translate the business indexing into the established difference-array convention before updating state. Inclusive flights first..last correspond to zero-based cells [first-1,last). That makes last the cancellation boundary without another subtraction or addition.

State and boundaries

difference is indexed by zero-based array boundaries. active is the total booked seats on the current flight.

Protocol

1. Allocate n+1 differences.
2. Add seats at first-1.
3. Subtract seats at last.
4. Prefix-sum the first n entries.
5. Return per-flight totals.

Visualization

```
flights 2..4 inclusive
cell indices 1,2,3
boundary range [1,4)
```

Python implementation

```
def flight_bookings(n, bookings):
    difference = [0] * (n + 1)
    for first, last, seats in bookings:
        difference[first - 1] += seats
        difference[last] -= seats
    output = []
    active = 0
    for i in range(n):
        active += difference[i]
        output.append(active)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
flight_bookings(5, [(1, 2, 10), (2, 3, 20), (2, 5, 25)])

L2 flight_bookings: difference = [0] * (n + 1)
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[0, 0, 0, 0, 0, 0]

L3 flight_bookings: for first, last, seats in bookings:
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[0, 0, 0, 0, 0, 0]; first=1;
    last=2; seats=10

L4 flight_bookings: difference[first - 1] += seats
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 0, 0, 0, 0, 0]; first=1;
    last=2; seats=10

L5 flight_bookings: difference[last] -= seats
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 0, -10, 0, 0, 0]; first=1;
     last=2; seats=10

L3 flight_bookings: for first, last, seats in bookings:
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 0, -10, 0, 0, 0]; first=2;
     last=3; seats=20

L4 flight_bookings: difference[first - 1] += seats
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 20, -10, 0, 0, 0];
    first=2; last=3; seats=20

L5 flight_bookings: difference[last] -= seats
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 20, -10, -20, 0, 0];
    first=2; last=3; seats=20

L3 flight_bookings: for first, last, seats in bookings:
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 20, -10, -20, 0, 0];
    first=2; last=5; seats=25

... 14 intermediate state changes omitted ...

L9 flight_bookings: active += difference[i]
```

```
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 45, -10, -20, 0, -25];
    first=2; last=5; seats=25; output=[10, 55, 45]; active=25; i=3

L10 flight_bookings: output.append(active)
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 45, -10, -20, 0, -25];
    first=2; last=5; seats=25; output=[10, 55, 45, 25]; active=25; i=3

L8 flight_bookings: for i in range(n):
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 45, -10, -20, 0, -25];
    first=2; last=5; seats=25; output=[10, 55, 45, 25]; active=25; i=4

L10 flight_bookings: output.append(active)
n=5; bookings=[(1, 2, 10), (2, 3, 20), (2, 5, 25)]; difference=[10, 45, -10, -20, 0, -25];
    first=2; last=5; seats=25; output=[10, 55, 45, 25, 25]; active=25; i=4
```

Result: [10, 55, 45, 25, 25]. Flight two receives all three bookings.

```
flight_bookings(1, [(1, 1, 7)])

L2 flight_bookings: difference = [0] * (n + 1)
n=1; bookings=[(1, 1, 7)]; difference=[0, 0]

L3 flight_bookings: for first, last, seats in bookings:
n=1; bookings=[(1, 1, 7)]; difference=[0, 0]; first=1; last=1; seats=7

L4 flight_bookings: difference[first - 1] += seats
n=1; bookings=[(1, 1, 7)]; difference=[7, 0]; first=1; last=1; seats=7

L5 flight_bookings: difference[last] -= seats
n=1; bookings=[(1, 1, 7)]; difference=[7, -7]; first=1; last=1; seats=7

L6 flight_bookings: output = []
n=1; bookings=[(1, 1, 7)]; difference=[7, -7]; first=1; last=1; seats=7; output=[]

L7 flight_bookings: active = 0
n=1; bookings=[(1, 1, 7)]; difference=[7, -7]; first=1; last=1; seats=7; output=[]; active=0

L8 flight_bookings: for i in range(n):
n=1; bookings=[(1, 1, 7)]; difference=[7, -7]; first=1; last=1; seats=7; output=[]; active=0;
    i=0

L9 flight_bookings: active += difference[i]
n=1; bookings=[(1, 1, 7)]; difference=[7, -7]; first=1; last=1; seats=7; output=[]; active=7;
    i=0

L10 flight_bookings: output.append(active)
n=1; bookings=[(1, 1, 7)]; difference=[7, -7]; first=1; last=1; seats=7; output=[7]; active=7;
     i=0
```

Result: [7]. Both inclusive endpoints name the same flight.

Why the solution is correct

The translated half-open cell range includes exactly the named flights. Each booking activates at the first included cell and cancels just after the last included cell, so integration accumulates precisely the bookings covering each flight.

Time and space

O(n+b) time and O(n) space for b bookings.

Common mistakes

Subtracting at last-1 removes seats from the final included flight. Mixing flight numbers with cell indices causes endpoint errors.

Reasoning card

State: difference is indexed by zero-based array boundaries. active is the total booked seats on the current flight.

Invariant: The running sum is total seats booked on the current flight.

Decision to rehearse: Translate inclusive ranges into start +delta and after-end -delta.

Finish: Return per-flight totals

Transfer exercise

Apply the same endpoint translation to inclusive date ranges after mapping dates to consecutive day indices.

060. Two-Dimensional Prefix Queries

Interview | Module 06

Problem

Preprocess a rectangular integer matrix and answer queries (r1,c1,r2,c2) over rows [r1,r2) and columns [c1,c2). All boundaries are valid. Return sums in query order. Empty-area rectangles sum to zero; an empty matrix supports only empty rectangles.

Examples

Example 1. The first query sums 2,3,5,6; the last has zero height.

```
rectangle_sums([[1,2,3],[4,5,6]], [(0,1,2,3),(1,0,2,2),(0,0,0,3)])
=> [16, 9, 0]
```

Example 2. All boundaries coincide in the empty matrix.

```
rectangle_sums([], [(0,0,0,0)])
=> [0]
```

Example 3. Signed values work with inclusion-exclusion.

```
rectangle_sums([[-2]], [(0,0,1,1)])
=> [-2]
```

Why this mechanism works

A two-dimensional prefix records the rectangle above and left of a boundary. To isolate a query, subtract the unwanted top strip and left strip. Their overlap was subtracted twice, so add it back once. The same inclusion-exclusion principle builds the prefix table.

State and boundaries

prefix[r][c] sums matrix rows before r and columns before c. Its leading row and column are zero boundaries.

Protocol

1. Allocate an extra zero row and column.

2. Build each prefix from top plus left minus shared corner plus current cell.

3. For each query take large prefix minus top minus left plus overlap.

4. Emit the result.

Visualization

```
P(r2,c2) - P(r1,c2)
         - P(r2,c1) + P(r1,c1)
last term restores the doubly removed corner
```

Python implementation

```
def rectangle_sums(matrix, queries):
    rows = len(matrix)
    cols = len(matrix[0]) if rows else 0
    prefix = [[0] * (cols + 1) for _ in range(rows + 1)]
    for r in range(rows):
        for c in range(cols):
            prefix[r + 1][c + 1] = (
                matrix[r][c] + prefix[r][c + 1]
                + prefix[r + 1][c] - prefix[r][c]
            )
    output = []
    for r1, c1, r2, c2 in queries:
        total = (prefix[r2][c2] - prefix[r1][c2]
                 - prefix[r2][c1] + prefix[r1][c1])
        output.append(total)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
rectangle_sums([[1,2,3],[4,5,6]], [(0,1,2,3),(1,0,2,2),(0,0,0,3)])

L2 rectangle_sums: rows = len(matrix)
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2

L3 rectangle_sums: cols = len(matrix[0]) if rows else 0
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3

L4 rectangle_sums: prefix = [[0] * (cols + 1) for _ in range(rows + 1)]
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; _=0

L4 rectangle_sums: prefix = [[0] * (cols + 1) for _ in range(rows + 1)]
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; _=1

L4 rectangle_sums: prefix = [[0] * (cols + 1) for _ in range(rows + 1)]
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; _=2

L4 rectangle_sums: prefix = [[0] * (cols + 1) for _ in range(rows + 1)]
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; prefix=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

L5 rectangle_sums: for r in range(rows):
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; prefix=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]; r=0

L6 rectangle_sums: for c in range(cols):
```

```
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; prefix=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]; r=0; c=0

... 18 intermediate state changes omitted ...

L15 rectangle_sums: output.append(total)
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; prefix=[[0, 0, 0, 0], [0, 1, 3, 6], [0, 5, 12, 21]]; r=1; c=2; output=[16, 9];
    r1=1; c1=0; r2=2; c2=2; total=9

L12 rectangle_sums: for r1, c1, r2, c2 in queries:
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; prefix=[[0, 0, 0, 0], [0, 1, 3, 6], [0, 5, 12, 21]]; r=1; c=2; output=[16, 9];
    r1=0; c1=0; r2=0; c2=3; total=9

L13 rectangle_sums: total = (prefix[r2][c2] - prefix[r1][c2]
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; prefix=[[0, 0, 0, 0], [0, 1, 3, 6], [0, 5, 12, 21]]; r=1; c=2; output=[16, 9];
    r1=0; c1=0; r2=0; c2=3; total=0

L15 rectangle_sums: output.append(total)
matrix=[[1, 2, 3], [4, 5, 6]]; queries=[(0, 1, 2, 3), (1, 0, 2, 2), (0, 0, 0, 3)]; rows=2;
    cols=3; prefix=[[0, 0, 0, 0], [0, 1, 3, 6], [0, 5, 12, 21]]; r=1; c=2; output=[16, 9, 0];
    r1=0; c1=0; r2=0; c2=3; total=0
```

Result: [16, 9, 0]. The first query sums 2,3,5,6; the last has zero height.

```
rectangle_sums([], [(0,0,0,0)])

L2 rectangle_sums: rows = len(matrix)
matrix=[]; queries=[(0, 0, 0, 0)]; rows=0

L3 rectangle_sums: cols = len(matrix[0]) if rows else 0
matrix=[]; queries=[(0, 0, 0, 0)]; rows=0; cols=0

L4 rectangle_sums: prefix = [[0] * (cols + 1) for _ in range(rows + 1)]
matrix=[]; queries=[(0, 0, 0, 0)]; rows=0; cols=0; _=0

L4 rectangle_sums: prefix = [[0] * (cols + 1) for _ in range(rows + 1)]
matrix=[]; queries=[(0, 0, 0, 0)]; rows=0; cols=0; prefix=[[0]]

L11 rectangle_sums: output = []
matrix=[]; queries=[(0, 0, 0, 0)]; rows=0; cols=0; prefix=[[0]]; output=[]

L12 rectangle_sums: for r1, c1, r2, c2 in queries:
matrix=[]; queries=[(0, 0, 0, 0)]; rows=0; cols=0; prefix=[[0]]; output=[]; r1=0; c1=0; r2=0;
    c2=0

L13 rectangle_sums: total = (prefix[r2][c2] - prefix[r1][c2]
matrix=[]; queries=[(0, 0, 0, 0)]; rows=0; cols=0; prefix=[[0]]; output=[]; r1=0; c1=0; r2=0;
    c2=0; total=0

L15 rectangle_sums: output.append(total)
matrix=[]; queries=[(0, 0, 0, 0)]; rows=0; cols=0; prefix=[[0]]; output=[0]; r1=0; c1=0; r2=0;
     c2=0; total=0
```

Result: [0]. All boundaries coincide in the empty matrix.

Why the solution is correct

The construction formula counts each cell in the represented rectangle once. The query formula removes exactly the cells outside the requested row and column bounds, adding back the doubly removed corner. Thus only the requested rectangle remains.

Time and space

O(rows*cols+q) time and O(rows*cols+q) storage including answers; each query is O(1).

Common mistakes

Omitting the added-back corner double-subtracts it. Using matrix coordinates directly in the prefix table forgets the leading boundary row and column.

Reasoning card

State: prefix[r][c] sums matrix rows before r and columns before c. Its leading row and column are zero boundaries.

Invariant: prefix[r][c] summarizes the rectangle above and left of that boundary.

Decision to rehearse: Use inclusion-exclusion over four prefix corners.

Finish: Emit the result

Transfer exercise

Describe the four corner updates needed for adding a value to many rectangles in a two-dimensional difference array.

Module 07
Stacks and Expression State

Use last-in-first-out state for nested structures, unresolved neighbours, and parsing contexts.

A stack records unfinished obligations in last-in, first-out order. Delimiters, operators, deferred strings and colliding objects all have different meanings, but each next event interacts with the most recent unresolved state. Write down what a pop completes before writing the pop itself.

The ten-step progression

061. Balanced Brackets

062. Evaluate Postfix Notation

063. Remove Adjacent Equal Pairs

064. Simplify an Absolute Unix Path

065. Validate Stack Push/Pop Sequences

066. Design Min Stack

067. Decode Nested Repetition

068. Asteroid Collision

069. Basic Calculator with Parentheses

070. Score of Parentheses

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

061. Balanced Brackets

Foundation | Module 07

Problem

Decide whether a string containing only (), [] and {} is correctly nested and matched. Empty input is balanced. Every closing bracket must match the most recent unmatched opening bracket; having equal total counts is not sufficient.

Examples

Example 1. Each closing bracket matches the innermost opening.

```
balanced_brackets("([{}])")
=> True
```

Example 2. Counts match but nesting crosses.

```
balanced_brackets("([)]")
=> False
```

Example 3. There are no unmatched brackets.

```
balanced_brackets("")
=> True
```

Why this mechanism works

Nested structure is last-opened, first-closed. Store unresolved opening brackets on a stack. A closing bracket has only one legal partner: the top. At the end, unmatched openings are as invalid as unmatched closings.

State and boundaries

stack contains opening brackets whose closing partners have not yet appeared, in nesting order.

Protocol

1. Push every opening bracket.
2. For a closing bracket require a nonempty stack with the matching top.
3. Pop that match.
4. Accept only if the stack is empty at the end.

Visualization

```
([{}])
open stack: ( -> ([ -> ([{
close order: } -> ] -> )
```

Python implementation

```
def balanced_brackets(text):
    matching = {')': '(', ']': '[', '}': '{'}
    stack = []
    for char in text:
        if char in '([{':
            stack.append(char)
        else:
            if not stack or stack[-1] != matching[char]:
                return False
            stack.pop()
    return not stack
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
balanced_brackets("([{}])")

L2 balanced_brackets: matching = {')': '(', ']': '[', '}': '{'}
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}

L3 balanced_brackets: stack = []
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=[]

L4 balanced_brackets: for char in text:
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=[]; char='('

L6 balanced_brackets: stack.append(char)
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=['(']; char='('

L4 balanced_brackets: for char in text:
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=['(']; char='['

L6 balanced_brackets: stack.append(char)
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=['(', '[']; char='['

L4 balanced_brackets: for char in text:
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=['(', '[']; char='{'

L6 balanced_brackets: stack.append(char)
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=['(', '[', '{']; char='{'

... 2 intermediate state changes omitted ...

L4 balanced_brackets: for char in text:
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=['(', '[']; char=']'

L10 balanced_brackets: stack.pop()
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=['(']; char=']'

L4 balanced_brackets: for char in text:
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=['(']; char=')'
```

```
L10 balanced_brackets: stack.pop()
text='([{}])'; matching={')':'(', ']':'[', '}':'{'}; stack=[]; char=')'
```

Result: True. Each closing bracket matches the innermost opening.

```
balanced_brackets("([)]")

L2 balanced_brackets: matching = {')': '(', ']': '[', '}': '{'}
text='([)]'; matching={')':'(', ']':'[', '}':'{'}

L3 balanced_brackets: stack = []
text='([)]'; matching={')':'(', ']':'[', '}':'{'}; stack=[]

L4 balanced_brackets: for char in text:
text='([)]'; matching={')':'(', ']':'[', '}':'{'}; stack=[]; char='('

L6 balanced_brackets: stack.append(char)
text='([)]'; matching={')':'(', ']':'[', '}':'{'}; stack=['(']; char='('

L4 balanced_brackets: for char in text:
text='([)]'; matching={')':'(', ']':'[', '}':'{'}; stack=['(']; char='['

L6 balanced_brackets: stack.append(char)
text='([)]'; matching={')':'(', ']':'[', '}':'{'}; stack=['(', '[']; char='['

L4 balanced_brackets: for char in text:
text='([)]'; matching={')':'(', ']':'[', '}':'{'}; stack=['(', '[']; char=')'
```

Result: False. Counts match but nesting crosses.

Why the solution is correct

Pushing records a newly opened inner context. Matching only the top enforces nesting as well as type. Rejecting an impossible close is necessary, and an empty final stack means every recorded opening was matched exactly once.

Time and space

O(n) time and O(d) stack space, where d is maximum nesting depth.

Common mistakes

Counting bracket types accepts crossing pairs such as ([)]. Forgetting the final emptiness check accepts unfinished openings.

Reasoning card

State: stack contains opening brackets whose closing partners have not yet appeared, in nesting order.

Invariant: Stack contains exactly unresolved opening brackets.

Decision to rehearse: Reject empty-stack closes; compare only with the most recent opener.

Finish: Accept only if the stack is empty at the end

Transfer exercise

Return the index of the first impossible closing bracket, distinguishing it from unmatched openings discovered only at the end.

062. Evaluate Postfix Notation

Foundation | Module 07

Problem

Evaluate a valid nonempty, space-separated postfix expression of signed integer literals and binary +, -, * or /. Division truncates toward zero, and divisors are nonzero. Return the integer result. The expression is guaranteed to reduce to one value.

Examples

Example 1. Multiplication combines three and four before addition.

```
evaluate_postfix("2 3 4 * +")
=> 14
```

Example 2. Truncation moves toward zero, not negative infinity.

```
evaluate_postfix("-7 3 /")
=> -2
```

Example 3. The second pop is the left operand.

```
evaluate_postfix("9 4 -")
=> 5
```

Why this mechanism works

A postfix operator arrives after both operand expressions have been evaluated. Their results are the two most recent unresolved values. Pop the right operand first and the left operand second. For exact integer truncation, divide absolute magnitudes and restore the sign instead of converting through floating point.

State and boundaries

stack contains values of completed subexpressions not yet consumed by an operator.

Protocol

1. Push integer literals.
2. On an operator pop right then left.
3. Apply the operator in that order.
4. Push the combined result.
5. Return the sole remaining value.

Visualization

```
2 3 4 * +
[2] -> [2,3] -> [2,3,4]
* -> [2,12]
 -> [14]
```

Python implementation

```
def evaluate_postfix(expression):
    stack = []
    for token in expression.split():
        if token not in ('+', '-', '*', '/'):
            stack.append(int(token))
            continue
        right = stack.pop()
        left = stack.pop()
        if token == '+':
            value = left + right
        elif token == '-':
            value = left - right
        elif token == '*':
            value = left * right
        else:
            value = abs(left) // abs(right)
            if (left < 0) != (right < 0):
                value = -value
        stack.append(value)
    return stack[0]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
evaluate_postfix("2 3 4 * +")

L2 evaluate_postfix: stack = []
expression='2 3 4 * +'; stack=[]

L3 evaluate_postfix: for token in expression.split():
expression='2 3 4 * +'; stack=[]; token='2'

L5 evaluate_postfix: stack.append(int(token))
expression='2 3 4 * +'; stack=[2]; token='2'

L3 evaluate_postfix: for token in expression.split():
expression='2 3 4 * +'; stack=[2]; token='3'

L5 evaluate_postfix: stack.append(int(token))
expression='2 3 4 * +'; stack=[2, 3]; token='3'

L3 evaluate_postfix: for token in expression.split():
expression='2 3 4 * +'; stack=[2, 3]; token='4'

L5 evaluate_postfix: stack.append(int(token))
expression='2 3 4 * +'; stack=[2, 3, 4]; token='4'

L3 evaluate_postfix: for token in expression.split():
expression='2 3 4 * +'; stack=[2, 3, 4]; token='*'
```

```
... 5 intermediate state changes omitted ...

L7 evaluate_postfix: right = stack.pop()
expression='2 3 4 * +'; stack=[2]; token='+'; right=12; left=3; value=12

L8 evaluate_postfix: left = stack.pop()
expression='2 3 4 * +'; stack=[]; token='+'; right=12; left=2; value=12

L10 evaluate_postfix: value = left + right
expression='2 3 4 * +'; stack=[]; token='+'; right=12; left=2; value=14

L19 evaluate_postfix: stack.append(value)
expression='2 3 4 * +'; stack=[14]; token='+'; right=12; left=2; value=14
```

Result: 14. Multiplication combines three and four before addition.

```
evaluate_postfix("-7 3 /")

L2 evaluate_postfix: stack = []
expression='-7 3 /'; stack=[]

L3 evaluate_postfix: for token in expression.split():
expression='-7 3 /'; stack=[]; token='-7'

L5 evaluate_postfix: stack.append(int(token))
expression='-7 3 /'; stack=[-7]; token='-7'

L3 evaluate_postfix: for token in expression.split():
expression='-7 3 /'; stack=[-7]; token='3'

L5 evaluate_postfix: stack.append(int(token))
expression='-7 3 /'; stack=[-7, 3]; token='3'

L3 evaluate_postfix: for token in expression.split():
expression='-7 3 /'; stack=[-7, 3]; token='/'

L7 evaluate_postfix: right = stack.pop()
expression='-7 3 /'; stack=[-7]; token='/'; right=3

L8 evaluate_postfix: left = stack.pop()
expression='-7 3 /'; stack=[]; token='/'; right=3; left=-7

L16 evaluate_postfix: value = abs(left) // abs(right)
expression='-7 3 /'; stack=[]; token='/'; right=3; left=-7; value=2

L18 evaluate_postfix: value = -value
expression='-7 3 /'; stack=[]; token='/'; right=3; left=-7; value=-2

L19 evaluate_postfix: stack.append(value)
expression='-7 3 /'; stack=[-2]; token='/'; right=3; left=-7; value=-2
```

Result: -2. Truncation moves toward zero, not negative infinity.

Why the solution is correct

Each literal is a complete subexpression. An operator replaces its two operand results by their correct combined value, preserving the stack's interpretation. Valid postfix syntax ensures the process finishes with the value of the full expression.

Time and space

O(t) arithmetic operations and O(t) stack space for t tokens; big-integer operation costs depend on operand size.

Common mistakes

Reversing pops breaks subtraction and division. Python // floors negative results rather than truncating toward zero.

Reasoning card

State: stack contains values of completed subexpressions not yet consumed by an operator.

Invariant: Stack holds evaluated subexpressions awaiting use.

Decision to rehearse: Numbers push; operators pop right then left; final stack must contain one value.

Finish: Return the sole remaining value

Transfer exercise

Build an expression tree instead of immediately evaluating, using the same operand-stack protocol.

063. Remove Adjacent Equal Pairs

Foundation | Module 07

Problem

Repeatedly remove adjacent equal character pairs until no such pair remains, and return the remaining string. Characters are case-sensitive. Removing a pair may expose a new pair across its former boundary. Empty input returns an empty string.

Examples

Example 1. Removing bb exposes aa, which also disappears.

```
remove_adjacent_pairs("abbaca")
=> 'ca'
```

Example 2. Two cancellations remove all four characters.

```
remove_adjacent_pairs("aaaa")
=> ''
```

Example 3. The reduced empty prefix is empty.

```
remove_adjacent_pairs("")
=> ''
```

Why this mechanism works

Keep the already reduced prefix on a stack. An incoming character can create a new removable pair only with that prefix's final character. A match pops it; otherwise the new character remains unresolved at the top. This simulates cascading removals without rescanning the whole string.

State and boundaries

stack is the fully reduced form of the consumed prefix and has no adjacent equal pair.

Protocol

1. Compare the incoming character with the stack top.
2. Pop on equality.
3. Otherwise push the character.
4. Join the stack after the scan.

Visualization

```
abbaca
abb -> a
aa -> empty
remaining ca -> ca
```

Python implementation

```
def remove_adjacent_pairs(text):
    stack = []
    for char in text:
        if stack and stack[-1] == char:
            stack.pop()
        else:
            stack.append(char)
    return ''.join(stack)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
remove_adjacent_pairs("abbaca")

L2 remove_adjacent_pairs: stack = []
text='abbaca'; stack=[]

L3 remove_adjacent_pairs: for char in text:
text='abbaca'; stack=[]; char='a'

L7 remove_adjacent_pairs: stack.append(char)
text='abbaca'; stack=['a']; char='a'

L3 remove_adjacent_pairs: for char in text:
text='abbaca'; stack=['a']; char='b'

L7 remove_adjacent_pairs: stack.append(char)
text='abbaca'; stack=['a', 'b']; char='b'

L5 remove_adjacent_pairs: stack.pop()
text='abbaca'; stack=['a']; char='b'

L3 remove_adjacent_pairs: for char in text:
text='abbaca'; stack=['a']; char='a'

L5 remove_adjacent_pairs: stack.pop()
text='abbaca'; stack=[]; char='a'

L3 remove_adjacent_pairs: for char in text:
text='abbaca'; stack=[]; char='c'

L7 remove_adjacent_pairs: stack.append(char)
text='abbaca'; stack=['c']; char='c'

L3 remove_adjacent_pairs: for char in text:
text='abbaca'; stack=['c']; char='a'
```

```
L7 remove_adjacent_pairs: stack.append(char)
text='abbaca'; stack=['c', 'a']; char='a'
```

Result: 'ca'. Removing bb exposes aa, which also disappears.

```
remove_adjacent_pairs("aaaa")

L2 remove_adjacent_pairs: stack = []
text='aaaa'; stack=[]

L3 remove_adjacent_pairs: for char in text:
text='aaaa'; stack=[]; char='a'

L7 remove_adjacent_pairs: stack.append(char)
text='aaaa'; stack=['a']; char='a'

L5 remove_adjacent_pairs: stack.pop()
text='aaaa'; stack=[]; char='a'

L7 remove_adjacent_pairs: stack.append(char)
text='aaaa'; stack=['a']; char='a'

L5 remove_adjacent_pairs: stack.pop()
text='aaaa'; stack=[]; char='a'
```

Result: ''. Two cancellations remove all four characters.

Why the solution is correct

The reduced prefix has no internal removable pair. Appending one character can create a pair only at its boundary, and popping that pair restores the reduced-prefix property. Repeating this transition yields a fully reduced result for the whole input.

Time and space

O(n) time and O(n) stack/output space.

Common mistakes

Removing only pairs present in the original string misses cascades. Popping without checking emptiness fails on the first character.

Reasoning card

State: stack is the fully reduced form of the consumed prefix and has no adjacent equal pair.

Invariant: Stack is the fully reduced processed prefix.

Decision to rehearse: Stack top is the unresolved neighbour of the incoming character.

Finish: Join the stack after the scan

Transfer exercise

Remove runs of exactly k equal characters by storing a character and an active count per stack entry.

064. Simplify an Absolute Unix Path

Foundation | Module 07

Problem

Normalize an absolute Unix-style path lexically. Ignore repeated slashes and '.' components; '..' removes the most recent ordinary component when one exists and stays at root otherwise. Return one leading slash, no trailing slash except at root. Do not inspect a filesystem or resolve symbolic links.

Examples

Example 1. Two parent components undo b and a.

```
simplify_path("/a/./b/../../c/")
=> '/c'
```

Example 2. Parent navigation cannot move above root.

```
simplify_path("/../../")
=> '/'
```

Example 3. Three dots are an ordinary component.

```
simplify_path("/a//.../b/")
=> '/a/.../b'
```

Why this mechanism works

A path component adds a level, while '..' undoes the most recent retained level. A stack represents that reversible history. Dot-like names other than exactly '.' and '..', such as '...', are ordinary names.

State and boundaries

parts is the sequence of retained directory names from root to the current lexical location.

Protocol

1. Split on slash.
2. Ignore empty and dot components.
3. On dot-dot pop a retained component if possible.
4. Otherwise push the ordinary name.

5. Join with one slash and prepend root.

Visualization

```
/a/./b/../../c/
[] -> [a] -> [a,b] -> [a] -> [] -> [c]
result /c
```

Python implementation

```
def simplify_path(path):
    parts = []
    for part in path.split('/'):
        if part in ('', '.'):
            continue
        if part == '..':
            if parts:
                parts.pop()
        else:
            parts.append(part)
    return '/' + '/'.join(parts)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
simplify_path("/a/./b/../../c/")

L2 simplify_path: parts = []
path='/a/./b/../../c/'; parts=[]

L3 simplify_path: for part in path.split('/'):
path='/a/./b/../../c/'; parts=[]; part=''

L3 simplify_path: for part in path.split('/'):
path='/a/./b/../../c/'; parts=[]; part='a'

L10 simplify_path: parts.append(part)
path='/a/./b/../../c/'; parts=['a']; part='a'

L3 simplify_path: for part in path.split('/'):
path='/a/./b/../../c/'; parts=['a']; part='.'

L3 simplify_path: for part in path.split('/'):
path='/a/./b/../../c/'; parts=['a']; part='b'

L10 simplify_path: parts.append(part)
path='/a/./b/../../c/'; parts=['a', 'b']; part='b'

L3 simplify_path: for part in path.split('/'):
path='/a/./b/../../c/'; parts=['a', 'b']; part='..'

... 1 intermediate state changes omitted ...

L8 simplify_path: parts.pop()
path='/a/./b/../../c/'; parts=[]; part='..'

L3 simplify_path: for part in path.split('/'):
path='/a/./b/../../c/'; parts=[]; part='c'
```

```
L10 simplify_path: parts.append(part)
path='/a/./b/../../c/'; parts=['c']; part='c'

L3 simplify_path: for part in path.split('/'):
path='/a/./b/../../c/'; parts=['c']; part=''
```

Result: '/c'. Two parent components undo b and a.

```
simplify_path("/../../")

L2 simplify_path: parts = []
path='/../../'; parts=[]

L3 simplify_path: for part in path.split('/'):
path='/../../'; parts=[]; part=''

L3 simplify_path: for part in path.split('/'):
path='/../../'; parts=[]; part='..'

L3 simplify_path: for part in path.split('/'):
path='/../../'; parts=[]; part=''
```

Result: '/'. Parent navigation cannot move above root.

Why the solution is correct

Ordinary components extend the path; a parent component removes precisely the latest extension when possible. Ignored components do not change location. These rules preserve lexical path meaning, and joining the retained names gives the canonical form.

Time and space

O(n) time and O(n) space in the path length.

Common mistakes

Treating any name containing dots as special corrupts valid names. Lexical simplification is not symbolic-link resolution.

Reasoning card

State: parts is the sequence of retained directory names from root to the current lexical location.

Invariant: Stack contains normalized path components.

Decision to rehearse: Push normal components; pop on .. when possible.

Finish: Join with one slash and prepend root

Transfer exercise

Specify the different behavior needed for a relative path whose leading '..' components cannot be canceled.

065. Validate Stack Push/Pop Sequences

Bridge | Module 07

Problem

Given pushed with distinct values and popped of the same length, decide whether one stack can produce popped while pushing values in pushed order. At any time you may push the next input or pop the current top. Empty sequences are valid.

Examples

Example 1. The required pop chain can be simulated.

```
valid_stack_sequences([1,2,3,4,5], [4,5,3,2,1])
=> True
```

Example 2. Two blocks access to one near the end.

```
valid_stack_sequences([1,2,3,4,5], [4,3,5,1,2])
=> False
```

Example 3. No operations are required.

```
valid_stack_sequences([], [])
=> True
```

Why this mechanism works

Whenever the current top equals the next required output, popping it immediately cannot hurt: leaving it would only block access to values beneath it, and pushing more values is unnecessary for producing that required output. Simulate this forced opportunity after every push.

State and boundaries

j is the next required pop index. stack contains values pushed but not yet matched to the output sequence.

Protocol

1. Push the next input value.
2. While top equals popped[j], pop and advance j.
3. Continue with the next push.
4. Accept only if every requested output was matched.

Visualization

```
push 1,2,3,4; need 4 -> pop
next need 5 -> push 5, pop
then 3,2,1 can unwind
```

Python implementation

```python
def valid_stack_sequences(pushed, popped):
    if len(pushed) != len(popped):
        return False
    stack = []
    j = 0
    for value in pushed:
        stack.append(value)
        while stack and j < len(popped) and stack[-1] == popped[j]:
            stack.pop()
            j += 1
    return j == len(popped)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
valid_stack_sequences([1,2,3,4,5], [4,5,3,2,1])

L2 valid_stack_sequences: if len(pushed) != len(popped):
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]

L4 valid_stack_sequences: stack = []
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[]

L5 valid_stack_sequences: j = 0
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[]; j=0

L6 valid_stack_sequences: for value in pushed:
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[]; j=0; value=1

L7 valid_stack_sequences: stack.append(value)
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[1]; j=0; value=1

L6 valid_stack_sequences: for value in pushed:
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[1]; j=0; value=2

L7 valid_stack_sequences: stack.append(value)
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[1, 2]; j=0; value=2

L6 valid_stack_sequences: for value in pushed:
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[1, 2]; j=0; value=3

... 11 intermediate state changes omitted ...

L9 valid_stack_sequences: stack.pop()
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[1]; j=3; value=5

L10 valid_stack_sequences: j += 1
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[1]; j=4; value=5

L9 valid_stack_sequences: stack.pop()
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[]; j=4; value=5
```

```
L10 valid_stack_sequences: j += 1
pushed=[1, 2, 3, 4, 5]; popped=[4, 5, 3, 2, 1]; stack=[]; j=5; value=5
```

Result: True. The required pop chain can be simulated.

```
valid_stack_sequences([1,2,3,4,5], [4,3,5,1,2])

L2 valid_stack_sequences: if len(pushed) != len(popped):
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]

L4 valid_stack_sequences: stack = []
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[]

L5 valid_stack_sequences: j = 0
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[]; j=0

L6 valid_stack_sequences: for value in pushed:
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[]; j=0; value=1

L7 valid_stack_sequences: stack.append(value)
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[1]; j=0; value=1

L6 valid_stack_sequences: for value in pushed:
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[1]; j=0; value=2

L7 valid_stack_sequences: stack.append(value)
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[1, 2]; j=0; value=2

L6 valid_stack_sequences: for value in pushed:
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[1, 2]; j=0; value=3

... 7 intermediate state changes omitted ...

L6 valid_stack_sequences: for value in pushed:
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[1, 2]; j=2; value=5

L7 valid_stack_sequences: stack.append(value)
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[1, 2, 5]; j=2; value=5

L9 valid_stack_sequences: stack.pop()
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[1, 2]; j=2; value=5

L10 valid_stack_sequences: j += 1
pushed=[1, 2, 3, 4, 5]; popped=[4, 3, 5, 1, 2]; stack=[1, 2]; j=3; value=5
```

Result: False. Two blocks access to one near the end.

Why the solution is correct

Every simulated operation is legal. The next requested value cannot be popped unless it is on top; when it is, taking it cannot invalidate a later operation. Thus this greedy simulation succeeds exactly when the required order is achievable.

Time and space

O(n) time and O(n) space; each value is pushed and popped at most once.

Common mistakes

A single pop check can miss a whole newly exposed chain. Comparing unordered value sets does not validate stack order.

Reasoning card

State: j is the next required pop index. stack contains values pushed but not yet matched to the output sequence.

Invariant: The stack models exactly the values currently present.

Decision to rehearse: Push each input and pop greedily while the next requested value is on top.

Finish: Accept only if every requested output was matched

Transfer exercise

Return one valid push/pop operation sequence as a witness, rather than only a Boolean result.

066. Design Min Stack

Bridge | Module 07

Problem

Implement MinStack with push(value), pop(), top() and get_min(). Each operation must take O(1) time. pop returns the removed integer; push returns None. Reading or popping an empty stack raises IndexError. Duplicate minima must work correctly.

Examples

Example 1. Removing two leaves the minimum and top equal to one.

```
run_operations(MinStack,
    [("push",3),("push",1),("push",2),("get_min",),("pop",),("top",)])
=> [None, None, None, 1, 2, 1]
```

Example 2. An empty operation sequence returns no results.

```
run_operations(MinStack, [])
=> []
```

Example 3. Popping one copy does not lose a duplicate minimum.

```
run_operations(MinStack, [("push",1),("push",1),("pop",),("get_min",)])
=> [None, None, 1, 1]
```

Why this mechanism works

Store the minimum-so-far beside every pushed value. Then removing an entry restores the previous minimum automatically, including repeated minima. This uses extra state at each depth to make all operations independent of stack size.

State and boundaries

entries contains (value, minimum_through_this_depth) pairs. The final pair summarizes the whole current stack.

Protocol

1. On push combine value with the previous stored minimum.
2. Append both value and new minimum.
3. Pop one complete pair.
4. Read value or minimum from the final pair.

Visualization

```
push 3 -> (3,3)
push 1 -> (1,1)
push 2 -> (2,1)
pop 2 -> previous minimum still 1
```

Python implementation

```
from structures import run_operations

class MinStack:
    def __init__(self):
        self.entries = []

    def push(self, value):
        minimum = min(value, self.entries[-1][1]) if self.entries else value
        self.entries.append((value, minimum))

    def pop(self):
        return self.entries.pop()[0]

    def top(self):
        return self.entries[-1][0]

    def get_min(self):
        return self.entries[-1][1]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(MinStack,
    [("push",3),("push",1),("push",2),("get_min",),("pop",),("top",)])

L5 __init__: self.entries = []
self=MinStack{'entries':[]}

L8 push: minimum = min(value, self.entries[-1][1]) if self.entries else value
self=MinStack{'entries':[]}; value=3; minimum=3

L9 push: self.entries.append((value, minimum))
self=MinStack{'entries':[...]}; value=3; minimum=3

L8 push: minimum = min(value, self.entries[-1][1]) if self.entries else value
self=MinStack{'entries':[...]}; value=1; minimum=1

L9 push: self.entries.append((value, minimum))
self=MinStack{'entries':[..., ...]}; value=1; minimum=1

L8 push: minimum = min(value, self.entries[-1][1]) if self.entries else value
self=MinStack{'entries':[..., ...]}; value=2; minimum=1

L9 push: self.entries.append((value, minimum))
self=MinStack{'entries':[..., ..., ...]}; value=2; minimum=1

L18 get_min: return self.entries[-1][1]
self=MinStack{'entries':[..., ..., ...]}
```

```
L12 pop: return self.entries.pop()[0]
self=MinStack{'entries':[..., ...]}

L15 top: return self.entries[-1][0]
self=MinStack{'entries':[..., ...]}
```

Result: [None, None, None, 1, 2, 1]. Removing two leaves the minimum and top equal to one.

```
run_operations(MinStack, [])

L5 __init__: self.entries = []
self=MinStack{'entries':[]}
```

Result: []. An empty operation sequence returns no results.

Why the solution is correct

At depth one the value is its own minimum. Each later stored minimum is the smaller of the prior minimum and the incoming value. Popping removes only the final prefix summary, exposing the correct summary for the remaining prefix.

Time and space

O(1) time per operation and O(n) storage for n current values.

Common mistakes

A single global minimum cannot recover the previous minimum after a pop. Recording only strict new minima requires careful duplicate handling.

Reasoning card

State: entries contains (value, minimum_through_this_depth) pairs. The final pair summarizes the whole current stack.

Invariant: Current minimum is available at the top of auxiliary state.

Decision to rehearse: Store enough historical minimum state to survive pops.

Finish: Read value or minimum from the final pair

Transfer exercise

Store maximum-so-far alongside minimum-so-far and preserve constant-time operations.

067. Decode Nested Repetition

Bridge | Module 07

Problem

Decode a valid expression made from lowercase letters and groups count[expression], where count is a positive decimal integer and groups may nest. Concatenation is implicit. Empty input decodes to an empty string. Return the fully expanded text.

Examples

Example 1. The inner group is completed before the outer repetition.

```
decode_repetition("3[a2[c]]")
=> 'accaccacc'
```

Example 2. Counts may contain multiple digits.

```
decode_repetition("10[a]")
=> 'aaaaaaaaaa'
```

Example 3. The outer context has no fragments.

```
decode_repetition("")
=> ''
```

Why this mechanism works

Opening a group suspends a parent output context and its repetition count. Closing it completes the child text, repeats that text, and appends it to the restored parent. A stack preserves these contexts in nesting order. Multi-digit counts are accumulated numerically before '['.

State and boundaries

pieces holds completed fragments in the active context. stack holds suspended (parent_fragments, repeat_count) pairs. count is the numeric prefix being read.

Protocol

1. Accumulate digits into count.

2. On '[' save parent and count, then start a fresh child.

3. Append letters as fragments.

4. On ']' join and repeat the child, restore its parent and append the result.

5. Join the outer fragments.

Visualization

```
3[a2[c]]
inner 2[c] -> cc
parent a + cc -> acc
outer 3[acc] -> accaccacc
```

Python implementation

```
def decode_repetition(text):
    stack = []
    pieces = []
    count = 0
    for char in text:
        if char.isdigit():
            count = count * 10 + int(char)
        elif char == '[':
            stack.append((pieces, count))
            pieces, count = [], 0
        elif char == ']':
            parent, repeat = stack.pop()
            parent.append(''.join(pieces) * repeat)
            pieces = parent
        else:
            pieces.append(char)
    return ''.join(pieces)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
decode_repetition("3[a2[c]]")

L2 decode_repetition: stack = []
text='3[a2[c]]'; stack=[]

L3 decode_repetition: pieces = []
text='3[a2[c]]'; stack=[]; pieces=[]

L4 decode_repetition: count = 0
text='3[a2[c]]'; stack=[]; pieces=[]; count=0

L5 decode_repetition: for char in text:
text='3[a2[c]]'; stack=[]; pieces=[]; count=0; char='3'

L7 decode_repetition: count = count * 10 + int(char)
text='3[a2[c]]'; stack=[]; pieces=[]; count=3; char='3'

L5 decode_repetition: for char in text:
text='3[a2[c]]'; stack=[]; pieces=[]; count=3; char='['

L9 decode_repetition: stack.append((pieces, count))
text='3[a2[c]]'; stack=[([], 3)]; pieces=[]; count=3; char='['

L10 decode_repetition: pieces, count = [], 0
text='3[a2[c]]'; stack=[([], 3)]; pieces=[]; count=0; char='['

... 12 intermediate state changes omitted ...
```

```
L14 decode_repetition: pieces = parent
text='3[a2[c]]'; stack=[([], 3)]; pieces=['a', 'cc']; count=0; char=']'; parent=['a', 'cc'];
    repeat=2

L12 decode_repetition: parent, repeat = stack.pop()
text='3[a2[c]]'; stack=[]; pieces=['a', 'cc']; count=0; char=']'; parent=[]; repeat=3

L13 decode_repetition: parent.append(''.join(pieces) * repeat)
text='3[a2[c]]'; stack=[]; pieces=['a', 'cc']; count=0; char=']'; parent=['accaccacc'];
    repeat=3

L14 decode_repetition: pieces = parent
text='3[a2[c]]'; stack=[]; pieces=['accaccacc']; count=0; char=']'; parent=['accaccacc'];
    repeat=3
```

Result: 'accaccacc'. The inner group is completed before the outer repetition.

```
decode_repetition("10[a]")

L2 decode_repetition: stack = []
text='10[a]'; stack=[]

L3 decode_repetition: pieces = []
text='10[a]'; stack=[]; pieces=[]

L4 decode_repetition: count = 0
text='10[a]'; stack=[]; pieces=[]; count=0

L5 decode_repetition: for char in text:
text='10[a]'; stack=[]; pieces=[]; count=0; char='1'

L7 decode_repetition: count = count * 10 + int(char)
text='10[a]'; stack=[]; pieces=[]; count=1; char='1'

L5 decode_repetition: for char in text:
text='10[a]'; stack=[]; pieces=[]; count=1; char='0'

L7 decode_repetition: count = count * 10 + int(char)
text='10[a]'; stack=[]; pieces=[]; count=10; char='0'

L5 decode_repetition: for char in text:
text='10[a]'; stack=[]; pieces=[]; count=10; char='['

... 4 intermediate state changes omitted ...

L5 decode_repetition: for char in text:
text='10[a]'; stack=[([], 10)]; pieces=['a']; count=0; char=']'

L12 decode_repetition: parent, repeat = stack.pop()
text='10[a]'; stack=[]; pieces=['a']; count=0; char=']'; parent=[]; repeat=10

L13 decode_repetition: parent.append(''.join(pieces) * repeat)
text='10[a]'; stack=[]; pieces=['a']; count=0; char=']'; parent=['aaaaaaaaaa']; repeat=10

L14 decode_repetition: pieces = parent
text='10[a]'; stack=[]; pieces=['aaaaaaaaaa']; count=0; char=']'; parent=['aaaaaaaaaa'];
    repeat=10
```

Result: 'aaaaaaaaaa'. Counts may contain multiple digits.

Why the solution is correct

A context's fragments decode its completed components in order. Closing a valid group has all child components available, so joining and repeating gives that group's exact meaning. Restoring the enclosing context preserves concatenation order; the final join decodes the outer expression.

Time and space

O(n+L*d) conservative time for nesting depth d and final output length L, because nested expansions may recopy text; O(n+L+d) space. Output can be exponentially larger than encoded input.

Common mistakes

Resetting count too late reuses digits across groups. One shared current string without saved parent context loses text before a nested group.

Reasoning card

State: pieces holds completed fragments in the active context. stack holds suspended (parent_fragments, repeat_count) pairs. count is the numeric prefix being read.

Invariant: Current buffers describe the active nesting context.

Decision to rehearse: Save outer string/count when entering brackets; combine when closing.

Finish: Join the outer fragments

Transfer exercise

Compute only the decoded length, avoiding materializing a potentially enormous string.

068. Asteroid Collision

Interview | Module 07

Problem

Asteroids lie in listed left-to-right order. A positive integer moves right, a negative integer moves left, and magnitude is size; zero is excluded. Equal-speed asteroids collide only when approaching. The smaller disappears; equal sizes both disappear. Return survivors in order.

Examples

Example 1. Ten destroys the smaller incoming asteroid.

```
asteroid_collision([5, 10, -5])
=> [5, 10]
```

Example 2. Equal approaching sizes both disappear.

```
asteroid_collision([8, -8])
=> []
```

Example 3. These asteroids do not approach across a positive-negative boundary.

```
asteroid_collision([-2, -1, 1, 2])
=> [-2, -1, 1, 2]
```

Why this mechanism works

Only a right-moving stack top and a new left-moving asteroid can approach each other. A large incoming asteroid may destroy several earlier candidates, so resolve in a loop. Other sign combinations move apart or in parallel and cannot collide.

State and boundaries

stack is the fully resolved survivor sequence of the consumed prefix. alive indicates whether the incoming asteroid still exists.

Protocol

1. While incoming is left-moving and top is right-moving, compare sizes.
2. Pop a smaller top and continue.
3. Destroy both on equality.
4. Destroy incoming when top is larger.
5. Append incoming only if it survives.

Visualization

```
[10,2] incoming -5
2 disappears -> compare 10 with -5
-5 disappears -> survivor [10]
```

Python implementation

```
def asteroid_collision(asteroids):
    stack = []
    for asteroid in asteroids:
        alive = True
        while alive and asteroid < 0 and stack and stack[-1] > 0:
            if stack[-1] < -asteroid:
                stack.pop()
            elif stack[-1] == -asteroid:
                stack.pop()
                alive = False
            else:
                alive = False
        if alive:
            stack.append(asteroid)
    return stack
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
asteroid_collision([5, 10, -5])

L2 asteroid_collision: stack = []
asteroids=[5, 10, -5]; stack=[]

L3 asteroid_collision: for asteroid in asteroids:
asteroids=[5, 10, -5]; stack=[]; asteroid=5

L4 asteroid_collision: alive = True
asteroids=[5, 10, -5]; stack=[]; asteroid=5; alive=True

L14 asteroid_collision: stack.append(asteroid)
asteroids=[5, 10, -5]; stack=[5]; asteroid=5; alive=True

L3 asteroid_collision: for asteroid in asteroids:
asteroids=[5, 10, -5]; stack=[5]; asteroid=10; alive=True

L14 asteroid_collision: stack.append(asteroid)
asteroids=[5, 10, -5]; stack=[5, 10]; asteroid=10; alive=True

L3 asteroid_collision: for asteroid in asteroids:
asteroids=[5, 10, -5]; stack=[5, 10]; asteroid=-5; alive=True

L12 asteroid_collision: alive = False
asteroids=[5, 10, -5]; stack=[5, 10]; asteroid=-5; alive=False
```

Result: [5, 10]. Ten destroys the smaller incoming asteroid.

```
asteroid_collision([8, -8])

L2 asteroid_collision: stack = []
asteroids=[8, -8]; stack=[]
```

```
L3 asteroid_collision: for asteroid in asteroids:
asteroids=[8, -8]; stack=[]; asteroid=8

L4 asteroid_collision: alive = True
asteroids=[8, -8]; stack=[]; asteroid=8; alive=True

L14 asteroid_collision: stack.append(asteroid)
asteroids=[8, -8]; stack=[8]; asteroid=8; alive=True

L3 asteroid_collision: for asteroid in asteroids:
asteroids=[8, -8]; stack=[8]; asteroid=-8; alive=True

L9 asteroid_collision: stack.pop()
asteroids=[8, -8]; stack=[]; asteroid=-8; alive=True

L10 asteroid_collision: alive = False
asteroids=[8, -8]; stack=[]; asteroid=-8; alive=False
```

Result: []. Equal approaching sizes both disappear.

Why the solution is correct

The previously resolved prefix has no internal future collision. Any new collision must involve the incoming asteroid and the nearest survivor to its left. Resolving that pair either finishes the incoming asteroid or exposes the next possible pair, covering every collision in order.

Time and space

O(n) time and O(n) space; each asteroid enters and leaves the stack at most once.

Common mistakes

Treating every opposite-sign pair as colliding wrongly cancels [-2,1], which moves apart. A single comparison misses chain collisions.

Reasoning card

State: stack is the fully resolved survivor sequence of the consumed prefix. alive indicates whether the incoming asteroid still exists.

Invariant: Stack is the stable outcome of the processed prefix.

Decision to rehearse: Only a positive stack top and negative incoming asteroid can collide.

Finish: Append incoming only if it survives

Transfer exercise

Explain why different speeds would invalidate this stack-only event ordering and require time-aware collision reasoning.

069. Basic Calculator with Parentheses

Interview | Module 07

Problem

Evaluate a valid expression containing nonnegative integer literals, spaces, binary + and -, and parentheses. A single unary + or - is allowed at the beginning of the expression or immediately after '('. Use parentheses for forms such as 1+(-2); consecutive operator chains are outside this grammar. Return an integer.

Examples

Example 1. The innermost subtraction is combined first.

```
calculate("1+(2-(3-4))")
=> 4
```

Example 2. The leading minus applies to the entire group.

```
calculate("-(2+3)+4")
=> -1
```

Example 3. Parentheses make the supported unary placement explicit.

```
calculate("1+(-2)")
=> -1
```

Why this mechanism works

Within one parenthesis context, keep a committed result, a pending sign and the current number. An opening parenthesis saves the parent's committed result and sign, then starts a fresh context. A closing parenthesis turns the completed child result into one signed value in its parent.

State and boundaries

result is the sum of committed terms in the active context. sign applies to number or the next parenthesized group. stack holds suspended (result,sign) contexts.

Protocol

1. Accumulate decimal digits.

2. On + or -, commit the current signed number and set the next sign.

3. On '(' save context and reset.

4. On ')' finish the child and combine with its saved parent.

5. Commit any final number.

Visualization

```
1 + (2 - (3 - 4))
inner: 3-4=-1
middle: 2-(-1)=3
outer: 1+3=4
```

Python implementation

```
def calculate(expression):
    result = number = 0
    sign = 1
    stack = []
    for char in expression:
        if char.isdigit():
            number = number * 10 + int(char)
        elif char in '+-':
            result += sign * number
            number = 0
            sign = 1 if char == '+' else -1
        elif char == '(':
            stack.append((result, sign))
            result, number, sign = 0, 0, 1
        elif char == ')':
            result += sign * number
            parent, parent_sign = stack.pop()
            result = parent + parent_sign * result
            number = 0
    return result + sign * number
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
calculate("1+(2-(3-4))")

L2 calculate: result = number = 0
expression='1+(2-(3-4))'; result=0; number=0

L3 calculate: sign = 1
expression='1+(2-(3-4))'; result=0; number=0; sign=1

L4 calculate: stack = []
expression='1+(2-(3-4))'; result=0; number=0; sign=1; stack=[]

L5 calculate: for char in expression:
expression='1+(2-(3-4))'; result=0; number=0; sign=1; stack=[]; char='1'

L7 calculate: number = number * 10 + int(char)
expression='1+(2-(3-4))'; result=0; number=1; sign=1; stack=[]; char='1'

L5 calculate: for char in expression:
expression='1+(2-(3-4))'; result=0; number=1; sign=1; stack=[]; char='+'

L9 calculate: result += sign * number
expression='1+(2-(3-4))'; result=1; number=1; sign=1; stack=[]; char='+'
```

```
L10 calculate: number = 0
expression='1+(2-(3-4))'; result=1; number=0; sign=1; stack=[]; char='+'

... 23 intermediate state changes omitted ...

L18 calculate: result = parent + parent_sign * result
expression='1+(2-(3-4))'; result=3; number=4; sign=-1; stack=[(1, 1)]; char=')'; parent=2;
    parent_sign=-1

L19 calculate: number = 0
expression='1+(2-(3-4))'; result=3; number=0; sign=-1; stack=[(1, 1)]; char=')'; parent=2;
    parent_sign=-1

L17 calculate: parent, parent_sign = stack.pop()
expression='1+(2-(3-4))'; result=3; number=0; sign=-1; stack=[]; char=')'; parent=1;
    parent_sign=1

L18 calculate: result = parent + parent_sign * result
expression='1+(2-(3-4))'; result=4; number=0; sign=-1; stack=[]; char=')'; parent=1;
    parent_sign=1
```

Result: 4. The innermost subtraction is combined first.

```
calculate("-(2+3)+4")

L2 calculate: result = number = 0
expression='-(2+3)+4'; result=0; number=0

L3 calculate: sign = 1
expression='-(2+3)+4'; result=0; number=0; sign=1

L4 calculate: stack = []
expression='-(2+3)+4'; result=0; number=0; sign=1; stack=[]

L5 calculate: for char in expression:
expression='-(2+3)+4'; result=0; number=0; sign=1; stack=[]; char='-'

L11 calculate: sign = 1 if char == '+' else -1
expression='-(2+3)+4'; result=0; number=0; sign=-1; stack=[]; char='-'

L5 calculate: for char in expression:
expression='-(2+3)+4'; result=0; number=0; sign=-1; stack=[]; char='('

L13 calculate: stack.append((result, sign))
expression='-(2+3)+4'; result=0; number=0; sign=-1; stack=[(0, -1)]; char='('

L14 calculate: result, number, sign = 0, 0, 1
expression='-(2+3)+4'; result=0; number=0; sign=1; stack=[(0, -1)]; char='('

... 11 intermediate state changes omitted ...

L19 calculate: number = 0
expression='-(2+3)+4'; result=-5; number=0; sign=1; stack=[]; char=')'; parent=0;
    parent_sign=-1

L5 calculate: for char in expression:
expression='-(2+3)+4'; result=-5; number=0; sign=1; stack=[]; char='+'; parent=0;
    parent_sign=-1

L5 calculate: for char in expression:
expression='-(2+3)+4'; result=-5; number=0; sign=1; stack=[]; char='4'; parent=0;
    parent_sign=-1
```

```
L7 calculate: number = number * 10 + int(char)
expression='-(2+3)+4'; result=-5; number=4; sign=1; stack=[]; char='4'; parent=0;
    parent_sign=-1
```

Result: -1. The leading minus applies to the entire group.

Why the solution is correct

Within each context, terms are added exactly once with their pending sign. Saving a parent preserves the sign that applies to the whole child expression. Closing evaluates that child before combining it, matching parenthesis semantics recursively without recursive calls.

Time and space

O(n) arithmetic operations and O(d) context space for nesting depth d; integer bit costs are additional.

Common mistakes

Subtracting only the first term of a parenthesized expression ignores the group's sign. This simple parser does not promise arbitrary consecutive unary operators.

Reasoning card

State: result is the sum of committed terms in the active context. sign applies to number or the next parenthesized group. stack holds suspended (result,sign) contexts.

Invariant: Current result represents the active parenthesized expression.

Decision to rehearse: Save outer result/sign when entering a new context.

Finish: Commit any final number

Transfer exercise

Extend the grammar to multiplication and division. Identify why the current single-level sign state cannot express precedence by itself.

070. Score of Parentheses

Interview | Module 07

Problem

Given a balanced parentheses string, compute its score: () has score one, concatenation adds scores, and wrapping a nonempty balanced expression doubles its score. Empty input has score zero. Return an integer.

Examples

Example 1. The outer group doubles the sum one plus two.

```
parentheses_score("(()(()))")
=> 6
```

Example 2. Sibling groups add.

```
parentheses_score("()()")
=> 2
```

Example 3. The outer context remains zero.

```
parentheses_score("")
=> 0
```

Why this mechanism works

Each nesting level accumulates the scores of its completed children. When a level closes, an empty child context represents (), worth one; otherwise its accumulated score is doubled. Add that completed group to its parent context.

State and boundaries

stack holds one running score per open context, beginning with an outer zero context.

Protocol

1. Push a zero score on '('.
2. On ')' pop the child's accumulated score.
3. Convert zero to one, otherwise double it.
4. Add the group score to the parent.
5. Return the outer score.

Visualization

```
(()(()))
children: ()=1 and (())=2
inside score=3
outer wrapping -> 6
```

Python implementation

```
def parentheses_score(text):
    stack = [0]
    for char in text:
        if char == '(':
            stack.append(0)
        else:
            inside = stack.pop()
            stack[-1] += 1 if inside == 0 else 2 * inside
    return stack[0]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
parentheses_score("((()(()))")

L2 parentheses_score: stack = [0]
text='((()(()))'; stack=[0]

L3 parentheses_score: for char in text:
text='((()(()))'; stack=[0]; char='('

L5 parentheses_score: stack.append(0)
text='((()(()))'; stack=[0, 0]; char='('

L5 parentheses_score: stack.append(0)
text='((()(()))'; stack=[0, 0, 0]; char='('

L3 parentheses_score: for char in text:
text='((()(()))'; stack=[0, 0, 0]; char=')'

L7 parentheses_score: inside = stack.pop()
text='((()(()))'; stack=[0, 0]; char=')'; inside=0

L8 parentheses_score: stack[-1] += 1 if inside == 0 else 2 * inside
text='((()(()))'; stack=[0, 1]; char=')'; inside=0

L3 parentheses_score: for char in text:
text='((()(()))'; stack=[0, 1]; char='('; inside=0

... 5 intermediate state changes omitted ...

L7 parentheses_score: inside = stack.pop()
text='((()(()))'; stack=[0, 1]; char=')'; inside=1

L8 parentheses_score: stack[-1] += 1 if inside == 0 else 2 * inside
text='((()(()))'; stack=[0, 3]; char=')'; inside=1

L7 parentheses_score: inside = stack.pop()
text='((()(()))'; stack=[0]; char=')'; inside=3
```

```
L8 parentheses_score: stack[-1] += 1 if inside == 0 else 2 * inside
text='((()()))'; stack=[6]; char=')'; inside=3
```

Result: 6. The outer group doubles the sum one plus two.

```
parentheses_score("()()")

L2 parentheses_score: stack = [0]
text='()()'; stack=[0]

L3 parentheses_score: for char in text:
text='()()'; stack=[0]; char='('

L5 parentheses_score: stack.append(0)
text='()()'; stack=[0, 0]; char='('

L3 parentheses_score: for char in text:
text='()()'; stack=[0, 0]; char=')'

L7 parentheses_score: inside = stack.pop()
text='()()'; stack=[0]; char=')'; inside=0

L8 parentheses_score: stack[-1] += 1 if inside == 0 else 2 * inside
text='()()'; stack=[1]; char=')'; inside=0

L3 parentheses_score: for char in text:
text='()()'; stack=[1]; char='('; inside=0

L5 parentheses_score: stack.append(0)
text='()()'; stack=[1, 0]; char='('; inside=0

L3 parentheses_score: for char in text:
text='()()'; stack=[1, 0]; char=')'; inside=0

L7 parentheses_score: inside = stack.pop()
text='()()'; stack=[1]; char=')'; inside=0

L8 parentheses_score: stack[-1] += 1 if inside == 0 else 2 * inside
text='()()'; stack=[2]; char=')'; inside=0
```

Result: 2. Sibling groups add.

Why the solution is correct

An open context begins with no completed children, hence score zero. Closing applies exactly the primitive or wrapping rule, and adding to the parent applies concatenation. Structural induction over groups gives the full expression's score.

Time and space

O(n) arithmetic operations and O(d) stack entries; large scores require additional integer bits.

Common mistakes

Always doubling would give () a score of zero. Returning the last child rather than adding it loses sibling groups.

Reasoning card

State: stack holds one running score per open context, beginning with an outer zero context.

Invariant: Stack/depth accurately represents unmatched opening contexts.

Decision to rehearse: Use depth or stack contexts; commit score when a primitive pair closes.

Finish: Return the outer score

Transfer exercise

Derive an alternative that adds 2**depth for each primitive () pair, and prove that it matches the nesting rule.

Module 08 Monotonic Stacks and Monotonic Deques

Invert repeated future searches: let each incoming event resolve or dominate earlier pending candidates.

Monotonic structures keep only candidates that can still matter. A new value may resolve old candidates or dominate them permanently. Those are different reasons to remove an entry. Equality policy is especially important: repeated heights or minima must have a consistent owner, and window expiry is separate from value dominance.

The ten-step progression

071. Next Greater Element

072. Next Smaller Element

073. Days Until Warmer Temperature

074. Online Stock Span

075. Largest Rectangle in Histogram

076. Maximal Rectangle in Binary Matrix

077. Sum of Subarray Minimums

078. Sliding Window Maximum

079. Shortest Subarray Sum At Least K with Negatives

080. Remove K Digits

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

071. Next Greater Element

Foundation | Module 08

Problem

For each integer position, return the first strictly greater value to its right, or -1 if none exists. 'First' means smallest later index, not smallest greater magnitude. Equal values do not qualify. Return a list of the same length.

Examples

Example 1. Four resolves both pending twos.

```
next_greater([2, 1, 2, 4, 3])
=> [4, 2, 4, -1, -1]
```

Example 2. Equality is not strictly greater.

```
next_greater([2, 2])
=> [-1, -1]
```

Example 3. There are no pending positions.

```
next_greater([])
=> []
```

Why this mechanism works

Instead of searching the future separately for every cell, let each incoming value resolve earlier pending cells. A stack holds their indices in nonincreasing value order. If the incoming value is greater than the top, it is the first qualifying value for that unresolved position.

State and boundaries

stack contains indices still waiting for a strictly greater value; their values are nonincreasing from bottom to top.

Protocol

1. Initialize answers to -1.
2. For each incoming value pop every strictly smaller pending top.
3. Set each popped answer to the incoming value.
4. Push the current index.

Visualization

```
pending indices for values [2,1,2]
incoming 4 resolves both 2s
next greater is about position, not minimum magnitude
```

Python implementation

```
def next_greater(nums):
    answer = [-1] * len(nums)
    stack = []
    for i, value in enumerate(nums):
        while stack and nums[stack[-1]] < value:
            answer[stack.pop()] = value
        stack.append(i)
    return answer
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
next_greater([2, 1, 2, 4, 3])

L2 next_greater: answer = [-1] * len(nums)
nums=[2, 1, 2, 4, 3]; answer=[-1, -1, -1, -1, -1]

L3 next_greater: stack = []
nums=[2, 1, 2, 4, 3]; answer=[-1, -1, -1, -1, -1]; stack=[]

L4 next_greater: for i, value in enumerate(nums):
nums=[2, 1, 2, 4, 3]; answer=[-1, -1, -1, -1, -1]; stack=[]; i=0; value=2

L7 next_greater: stack.append(i)
nums=[2, 1, 2, 4, 3]; answer=[-1, -1, -1, -1, -1]; stack=[0]; i=0; value=2

L4 next_greater: for i, value in enumerate(nums):
nums=[2, 1, 2, 4, 3]; answer=[-1, -1, -1, -1, -1]; stack=[0]; i=1; value=1

L7 next_greater: stack.append(i)
nums=[2, 1, 2, 4, 3]; answer=[-1, -1, -1, -1, -1]; stack=[0, 1]; i=1; value=1

L4 next_greater: for i, value in enumerate(nums):
nums=[2, 1, 2, 4, 3]; answer=[-1, -1, -1, -1, -1]; stack=[0, 1]; i=2; value=2

L6 next_greater: answer[stack.pop()] = value
nums=[2, 1, 2, 4, 3]; answer=[-1, 2, -1, -1, -1]; stack=[0]; i=2; value=2

... 3 intermediate state changes omitted ...

L6 next_greater: answer[stack.pop()] = value
nums=[2, 1, 2, 4, 3]; answer=[4, 2, 4, -1, -1]; stack=[]; i=3; value=4

L7 next_greater: stack.append(i)
nums=[2, 1, 2, 4, 3]; answer=[4, 2, 4, -1, -1]; stack=[3]; i=3; value=4

L4 next_greater: for i, value in enumerate(nums):
nums=[2, 1, 2, 4, 3]; answer=[4, 2, 4, -1, -1]; stack=[3]; i=4; value=3
```

```
L7 next_greater: stack.append(i)
nums=[2, 1, 2, 4, 3]; answer=[4, 2, 4, -1, -1]; stack=[3, 4]; i=4; value=3
```

Result: [4, 2, 4, -1, -1]. Four resolves both pending twos.

```
next_greater([2, 2])

L2 next_greater: answer = [-1] * len(nums)
nums=[2, 2]; answer=[-1, -1]

L3 next_greater: stack = []
nums=[2, 2]; answer=[-1, -1]; stack=[]

L4 next_greater: for i, value in enumerate(nums):
nums=[2, 2]; answer=[-1, -1]; stack=[]; i=0; value=2

L7 next_greater: stack.append(i)
nums=[2, 2]; answer=[-1, -1]; stack=[0]; i=0; value=2

L4 next_greater: for i, value in enumerate(nums):
nums=[2, 2]; answer=[-1, -1]; stack=[0]; i=1; value=2

L7 next_greater: stack.append(i)
nums=[2, 2]; answer=[-1, -1]; stack=[0, 1]; i=1; value=2
```

Result: [-1, -1]. Equality is not strictly greater.

Why the solution is correct

Any earlier greater value would already have removed a pending index. Therefore the value that finally pops it is its first greater successor. Indices left on the stack have no qualifying successor and retain -1.

Time and space

O(n) time and O(n) space; every index is pushed and popped at most once.

Common mistakes

Using >= when resolving makes equal values qualify incorrectly. Storing only values cannot assign results to repeated positions reliably.

Reasoning card

State: stack contains indices still waiting for a strictly greater value; their values are nonincreasing from bottom to top.

Invariant: Stack values are non-increasing and unresolved.

Decision to rehearse: Current value resolves all smaller waiting indices on top.

Finish: Push the current index

Transfer exercise

Return distances instead of values, or treat the array as circular while ensuring each index receives only its first answer.

072. Next Smaller Element

Foundation | Module 08

Problem

For every position, return the first strictly smaller value to its right, or -1 when no such value exists. Equal values do not qualify. The input is unchanged and the output has the same length.

Examples

Example 1. One resolves several pending larger values.

```
next_smaller([4, 2, 5, 1])
=> [2, 1, 1, -1]
```

Example 2. An equal successor is not smaller.

```
next_smaller([1, 1])
=> [-1, -1]
```

Example 3. An empty input produces an empty output.

```
next_smaller([])
=> []
```

Why this mechanism works

Reverse the comparison from next-greater search, not the traversal direction. Pending values are nondecreasing from bottom to top. An incoming smaller value resolves a suffix of larger pending candidates, which explains why the data structure remains a stack.

State and boundaries

stack stores unresolved indices in nondecreasing value order. answer remains -1 until a first smaller successor arrives.

Protocol

1. Scan left to right.
2. While the top value exceeds the incoming value, pop and answer it.
3. Push the new index.
4. Leave unresolved answers at -1.

Visualization

```
values: 4 2 5 1
2 resolves 4
1 resolves 5 and 2
answers: 2 1 1 -1
```

Python implementation

```
def next_smaller(nums):
    answer = [-1] * len(nums)
    stack = []
    for i, value in enumerate(nums):
        while stack and nums[stack[-1]] > value:
            answer[stack.pop()] = value
        stack.append(i)
    return answer
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
next_smaller([4, 2, 5, 1])

L2 next_smaller: answer = [-1] * len(nums)
nums=[4, 2, 5, 1]; answer=[-1, -1, -1, -1]

L3 next_smaller: stack = []
nums=[4, 2, 5, 1]; answer=[-1, -1, -1, -1]; stack=[]

L4 next_smaller: for i, value in enumerate(nums):
nums=[4, 2, 5, 1]; answer=[-1, -1, -1, -1]; stack=[]; i=0; value=4

L7 next_smaller: stack.append(i)
nums=[4, 2, 5, 1]; answer=[-1, -1, -1, -1]; stack=[0]; i=0; value=4

L4 next_smaller: for i, value in enumerate(nums):
nums=[4, 2, 5, 1]; answer=[-1, -1, -1, -1]; stack=[0]; i=1; value=2

L6 next_smaller: answer[stack.pop()] = value
nums=[4, 2, 5, 1]; answer=[2, -1, -1, -1]; stack=[]; i=1; value=2

L7 next_smaller: stack.append(i)
nums=[4, 2, 5, 1]; answer=[2, -1, -1, -1]; stack=[1]; i=1; value=2

L4 next_smaller: for i, value in enumerate(nums):
nums=[4, 2, 5, 1]; answer=[2, -1, -1, -1]; stack=[1]; i=2; value=5

... 1 intermediate state changes omitted ...

L4 next_smaller: for i, value in enumerate(nums):
nums=[4, 2, 5, 1]; answer=[2, -1, -1, -1]; stack=[1, 2]; i=3; value=1

L6 next_smaller: answer[stack.pop()] = value
nums=[4, 2, 5, 1]; answer=[2, -1, 1, -1]; stack=[1]; i=3; value=1

L6 next_smaller: answer[stack.pop()] = value
nums=[4, 2, 5, 1]; answer=[2, 1, 1, -1]; stack=[]; i=3; value=1
```

```
L7 next_smaller: stack.append(i)
nums=[4, 2, 5, 1]; answer=[2, 1, 1, -1]; stack=[3]; i=3; value=1
```

Result: [2, 1, 1, -1]. One resolves several pending larger values.

```
next_smaller([1, 1])

L2 next_smaller: answer = [-1] * len(nums)
nums=[1, 1]; answer=[-1, -1]

L3 next_smaller: stack = []
nums=[1, 1]; answer=[-1, -1]; stack=[]

L4 next_smaller: for i, value in enumerate(nums):
nums=[1, 1]; answer=[-1, -1]; stack=[]; i=0; value=1

L7 next_smaller: stack.append(i)
nums=[1, 1]; answer=[-1, -1]; stack=[0]; i=0; value=1

L4 next_smaller: for i, value in enumerate(nums):
nums=[1, 1]; answer=[-1, -1]; stack=[0]; i=1; value=1

L7 next_smaller: stack.append(i)
nums=[1, 1]; answer=[-1, -1]; stack=[0, 1]; i=1; value=1
```

Result: [-1, -1]. An equal successor is not smaller.

Why the solution is correct

An unresolved index has encountered no smaller successor yet. The first incoming value that is smaller will reach and remove it after resolving any larger tops above it. The resulting answer is therefore the earliest qualifying successor.

Time and space

O(n) time and O(n) space by one push and at most one pop per index.

Common mistakes

Changing only the output sign rather than the comparison does not change the ordering invariant. Equal values must remain unresolved relative to one another.

Reasoning card

State: stack stores unresolved indices in nondecreasing value order. answer remains -1 until a first smaller successor arrives.

Invariant: Stack values are non-decreasing and unresolved.

Decision to rehearse: Invert the comparison while preserving unresolved indices.

Finish: Leave unresolved answers at -1

Transfer exercise

Return the previous strictly smaller index using a left-to-right stack that discards dominated history before answering the current position.

073. Days Until Warmer Temperature

Foundation | Module 08

Problem

For each daily temperature, return how many days must pass before a strictly warmer day. Return zero where no warmer day exists. Temperatures may repeat; output distances rather than temperature values.

Examples

Example 1. Each answer measures the distance to the first warmer day.

```
warmer_days([73,74,75,71,69,72,76,73])
=> [1, 1, 4, 2, 1, 1, 0, 0]
```

Example 2. The equal middle day does not resolve day zero.

```
warmer_days([70,70,71])
=> [2, 1, 0]
```

Example 3. No days need answers.

```
warmer_days([])
=> []
```

Why this mechanism works

The pending-neighbor mechanism is unchanged from next greater element. Store indices because the required output is a distance. A warmer incoming day resolves every cooler day currently exposed at the stack top.

State and boundaries

stack contains unresolved day indices with nonincreasing temperatures. answer[j] becomes i-j when day i resolves day j.

Protocol

1. Initialize all waits to zero.
2. For each day pop cooler pending days.
3. Store the index difference for each pop.
4. Push the current day.

Visualization

```
day 0:73 waits for day 1:74 -> 1
day 3:71 waits for day 5:72 -> 2
store indices to measure the wait
```

Python implementation

```
def warmer_days(temperatures):
    answer = [0] * len(temperatures)
    stack = []
    for i, value in enumerate(temperatures):
        while stack and temperatures[stack[-1]] < value:
            day = stack.pop()
            answer[day] = i - day
        stack.append(i)
    return answer
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
warmer_days([73,74,75,71,69,72,76,73])

L2 warmer_days: answer = [0] * len(temperatures)
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[0, 0, 0, 0, 0, 0, 0, 0]

L3 warmer_days: stack = []
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[0, 0, 0, 0, 0, 0, 0, 0]; stack=[]

L4 warmer_days: for i, value in enumerate(temperatures):
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[0, 0, 0, 0, 0, 0, 0, 0]; stack=[]; i=0;
     value=73

L8 warmer_days: stack.append(i)
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[0, 0, 0, 0, 0, 0, 0, 0]; stack=[0];
    i=0; value=73

L4 warmer_days: for i, value in enumerate(temperatures):
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[0, 0, 0, 0, 0, 0, 0, 0]; stack=[0];
    i=1; value=74

L6 warmer_days: day = stack.pop()
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[0, 0, 0, 0, 0, 0, 0, 0]; stack=[]; i=1;
     value=74; day=0

L7 warmer_days: answer[day] = i - day
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[1, 0, 0, 0, 0, 0, 0, 0]; stack=[]; i=1;
     value=74; day=0

L8 warmer_days: stack.append(i)
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[1, 0, 0, 0, 0, 0, 0, 0]; stack=[1];
    i=1; value=74; day=0

... 18 intermediate state changes omitted ...

L7 warmer_days: answer[day] = i - day
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[1, 1, 4, 2, 1, 1, 0, 0]; stack=[]; i=6;
     value=76; day=2
```

```
L8 warmer_days: stack.append(i)
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[1, 1, 4, 2, 1, 1, 0, 0]; stack=[6];
    i=6; value=76; day=2

L4 warmer_days: for i, value in enumerate(temperatures):
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[1, 1, 4, 2, 1, 1, 0, 0]; stack=[6];
    i=7; value=73; day=2

L8 warmer_days: stack.append(i)
temperatures=[73, 74, 75, 71, 69, 72, 76, 73]; answer=[1, 1, 4, 2, 1, 1, 0, 0]; stack=[6, 7];
    i=7; value=73; day=2
```

Result: [1, 1, 4, 2, 1, 1, 0, 0]. Each answer measures the distance to the first warmer day.

```
warmer_days([70,70,71])

L2 warmer_days: answer = [0] * len(temperatures)
temperatures=[70, 70, 71]; answer=[0, 0, 0]

L3 warmer_days: stack = []
temperatures=[70, 70, 71]; answer=[0, 0, 0]; stack=[]

L4 warmer_days: for i, value in enumerate(temperatures):
temperatures=[70, 70, 71]; answer=[0, 0, 0]; stack=[]; i=0; value=70

L8 warmer_days: stack.append(i)
temperatures=[70, 70, 71]; answer=[0, 0, 0]; stack=[0]; i=0; value=70

L4 warmer_days: for i, value in enumerate(temperatures):
temperatures=[70, 70, 71]; answer=[0, 0, 0]; stack=[0]; i=1; value=70

L8 warmer_days: stack.append(i)
temperatures=[70, 70, 71]; answer=[0, 0, 0]; stack=[0, 1]; i=1; value=70

L4 warmer_days: for i, value in enumerate(temperatures):
temperatures=[70, 70, 71]; answer=[0, 0, 0]; stack=[0, 1]; i=2; value=71

L6 warmer_days: day = stack.pop()
temperatures=[70, 70, 71]; answer=[0, 0, 0]; stack=[0]; i=2; value=71; day=1

L7 warmer_days: answer[day] = i - day
temperatures=[70, 70, 71]; answer=[0, 1, 0]; stack=[0]; i=2; value=71; day=1

L6 warmer_days: day = stack.pop()
temperatures=[70, 70, 71]; answer=[0, 1, 0]; stack=[]; i=2; value=71; day=0

L7 warmer_days: answer[day] = i - day
temperatures=[70, 70, 71]; answer=[2, 1, 0]; stack=[]; i=2; value=71; day=0

L8 warmer_days: stack.append(i)
temperatures=[70, 70, 71]; answer=[2, 1, 0]; stack=[2]; i=2; value=71; day=0
```

Result: [2, 1, 0]. The equal middle day does not resolve day zero.

Why the solution is correct

Each pending day has seen no warmer successor before the resolving day. Hence that day gives the minimum waiting distance. Remaining pending days never encounter a warmer successor and correctly keep zero.

Time and space

O(n) time and O(n) space.

Common mistakes

Returning the temperature difference answers a different question. Equal temperatures cannot resolve one another.

Reasoning card

State: stack contains unresolved day indices with nonincreasing temperatures. answer[j] becomes i-j when day i resolves day j.

Invariant: Stack indices wait for their first warmer future day.

Decision to rehearse: Write distance rather than the resolving value.

Finish: Push the current day

Transfer exercise

Require a temperature at least delta warmer. Explain why the simple monotonic-stack pop rule may no longer preserve the same candidate ordering.

074. Online Stock Span

Foundation | Module 08

Problem

For every incoming stock price, return its span: the number of consecutive days ending today whose prices are <= today's price. Today always counts. Return spans for the whole stream; an empty stream returns [].

Examples

Example 1. Compressed blocks let the last price absorb six days.

```
stock_spans([100,80,60,70,60,75,85])
=> [1, 1, 1, 2, 1, 4, 6]
```

Example 2. Equality is included in each span.

```
stock_spans([5,5,5])
=> [1, 2, 3]
```

Example 3. There is no incoming price.

```
stock_spans([])
=> []
```

Why this mechanism works

A smaller or equal previous price no longer blocks today's span. Store compressed blocks as (price,span). When a block's price is <= the new price, all days represented by that block can be absorbed at once. Strictly larger prices remain as barriers.

State and boundaries

stack has strictly decreasing block prices. Each block span counts consecutive days it summarizes.

Protocol

1. Start the incoming span at one.
2. Pop and add every block with price<=current.
3. Push the combined block.
4. Emit its span.

Visualization

```
prices: 100 80 60 70 60 75 85
spans:    1  1  1  2  1  4  6
75 absorbs blocks (60,1) and (70,2)
```

Python implementation

```
def stock_spans(prices):
    stack = []
    output = []
    for price in prices:
        span = 1
        while stack and stack[-1][0] <= price:
            span += stack.pop()[1]
        stack.append((price, span))
        output.append(span)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
stock_spans([100,80,60,70,60,75,85])

L2 stock_spans: stack = []
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[]

L3 stock_spans: output = []
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[]; output=[]

L4 stock_spans: for price in prices:
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[]; output=[]; price=100

L5 stock_spans: span = 1
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[]; output=[]; price=100; span=1

L8 stock_spans: stack.append((price, span))
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[(100, 1)]; output=[]; price=100; span=1

L9 stock_spans: output.append(span)
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[(100, 1)]; output=[1]; price=100; span=1

L4 stock_spans: for price in prices:
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[(100, 1)]; output=[1]; price=80; span=1

L8 stock_spans: stack.append((price, span))
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[(100, 1), (80, 1)]; output=[1]; price=80; span=1

... 19 intermediate state changes omitted ...

L7 stock_spans: span += stack.pop()[1]
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[(100, 1), (80, 1)]; output=[1, 1, 1, 2, 1, 4];
    price=85; span=5

L7 stock_spans: span += stack.pop()[1]
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[(100, 1)]; output=[1, 1, 1, 2, 1, 4]; price=85;
    span=6

L8 stock_spans: stack.append((price, span))
```

```
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[(100, 1), (85, 6)]; output=[1, 1, 1, 2, 1, 4];
    price=85; span=6

L9 stock_spans: output.append(span)
prices=[100, 80, 60, 70, 60, 75, 85]; stack=[(100, 1), (85, 6)]; output=[1, 1, 1, 2, 1, 4, 6];
     price=85; span=6
```

Result: [1, 1, 1, 2, 1, 4, 6]. Compressed blocks let the last price absorb six days.

```
stock_spans([5,5,5])

L2 stock_spans: stack = []
prices=[5, 5, 5]; stack=[]

L3 stock_spans: output = []
prices=[5, 5, 5]; stack=[]; output=[]

L4 stock_spans: for price in prices:
prices=[5, 5, 5]; stack=[]; output=[]; price=5

L5 stock_spans: span = 1
prices=[5, 5, 5]; stack=[]; output=[]; price=5; span=1

L8 stock_spans: stack.append((price, span))
prices=[5, 5, 5]; stack=[(5, 1)]; output=[]; price=5; span=1

L9 stock_spans: output.append(span)
prices=[5, 5, 5]; stack=[(5, 1)]; output=[1]; price=5; span=1

L7 stock_spans: span += stack.pop()[1]
prices=[5, 5, 5]; stack=[]; output=[1]; price=5; span=2

L8 stock_spans: stack.append((price, span))
prices=[5, 5, 5]; stack=[(5, 2)]; output=[1]; price=5; span=2

... 1 intermediate state changes omitted ...

L5 stock_spans: span = 1
prices=[5, 5, 5]; stack=[(5, 2)]; output=[1, 2]; price=5; span=1

L7 stock_spans: span += stack.pop()[1]
prices=[5, 5, 5]; stack=[]; output=[1, 2]; price=5; span=3

L8 stock_spans: stack.append((price, span))
prices=[5, 5, 5]; stack=[(5, 3)]; output=[1, 2]; price=5; span=3

L9 stock_spans: output.append(span)
prices=[5, 5, 5]; stack=[(5, 3)]; output=[1, 2, 3]; price=5; span=3
```

Result: [1, 2, 3]. Equality is included in each span.

Why the solution is correct

Every absorbed block contains only prices no greater than its representative price, which is no greater than today. These adjacent blocks therefore all belong to today's span. The first remaining greater price is the exact stopping barrier.

Time and space

O(n) total time and O(n) worst-case space; a single insertion may pop many blocks but each block is removed once.

Common mistakes

Using < instead of <= incorrectly treats equal prices as barriers. Counting popped blocks rather than their stored spans undercounts compressed history.

Reasoning card

State: stack has strictly decreasing block prices. Each block span counts consecutive days it summarizes.

Invariant: Stack contains strictly greater blocking prices with compressed spans.

Decision to rehearse: Pop dominated prior prices while accumulating their spans.

Finish: Emit its span

Transfer exercise

Turn this into an online next(price) class and explain the difference between worst-case and amortized update time.

075. Largest Rectangle in Histogram

Bridge | Module 08

Problem

Given nonnegative unit-width bar heights, return the largest rectangular area entirely under the histogram. A rectangle spans consecutive bars and uses their minimum height. Empty input returns zero. Do not mutate the heights.

Examples

Example 1. The height-five rectangle spans the middle two bars.

```
largest_histogram([2,1,5,6,2,3])
=> 10
```

Example 2. Equal heights can combine into a wider rectangle.

```
largest_histogram([2,2])
=> 4
```

Example 3. No positive area exists.

```
largest_histogram([])
=> 0
```

Why this mechanism works

A bar can define a candidate rectangle until a shorter bar closes its right boundary. Maintain increasing-height indices. When popping a taller bar, the current index is its first shorter boundary to the right and the new stack top limits its left extension. A final zero-height sentinel closes remaining positive-height candidates.

State and boundaries

stack contains indices in nondecreasing height order. Popping index j finalizes a rectangle of height heights[j] across all bars between its boundaries.

Protocol

1. Scan bars followed by a zero-height sentinel.
2. Pop while the current height is smaller.
3. Compute width as current_index-new_top-1, using -1 for no left blocker.
4. Update best area.

5. Push the current index.

Visualization

```
heights: 2 1 5 6 2 3
when 2 arrives: pop 6, then 5
height 5 spans two bars -> area 10
```

Python implementation

```
def largest_histogram(heights):
    stack = []
    best = 0
    for i in range(len(heights) + 1):
        current = heights[i] if i < len(heights) else 0
        while stack and heights[stack[-1]] > current:
            height = heights[stack.pop()]
            left = stack[-1] if stack else -1
            best = max(best, height * (i - left - 1))
        if i < len(heights):
            stack.append(i)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
largest_histogram([2,1,5,6,2,3])

L2 largest_histogram: stack = []
heights=[2, 1, 5, 6, 2, 3]; stack=[]

L3 largest_histogram: best = 0
heights=[2, 1, 5, 6, 2, 3]; stack=[]; best=0

L4 largest_histogram: for i in range(len(heights) + 1):
heights=[2, 1, 5, 6, 2, 3]; stack=[]; best=0; i=0

L5 largest_histogram: current = heights[i] if i < len(heights) else 0
heights=[2, 1, 5, 6, 2, 3]; stack=[]; best=0; i=0; current=2

L11 largest_histogram: stack.append(i)
heights=[2, 1, 5, 6, 2, 3]; stack=[0]; best=0; i=0; current=2

L4 largest_histogram: for i in range(len(heights) + 1):
heights=[2, 1, 5, 6, 2, 3]; stack=[0]; best=0; i=1; current=2

L5 largest_histogram: current = heights[i] if i < len(heights) else 0
heights=[2, 1, 5, 6, 2, 3]; stack=[0]; best=0; i=1; current=1

L7 largest_histogram: height = heights[stack.pop()]
heights=[2, 1, 5, 6, 2, 3]; stack=[]; best=0; i=1; current=1; height=2

... 25 intermediate state changes omitted ...

L7 largest_histogram: height = heights[stack.pop()]
heights=[2, 1, 5, 6, 2, 3]; stack=[1]; best=10; i=6; current=0; height=2; left=4

L8 largest_histogram: left = stack[-1] if stack else -1
```

```
heights=[2, 1, 5, 6, 2, 3]; stack=[1]; best=10; i=6; current=0; height=2; left=1

L7 largest_histogram: height = heights[stack.pop()]
heights=[2, 1, 5, 6, 2, 3]; stack=[]; best=10; i=6; current=0; height=1; left=1

L8 largest_histogram: left = stack[-1] if stack else -1
heights=[2, 1, 5, 6, 2, 3]; stack=[]; best=10; i=6; current=0; height=1; left=-1
```

Result: 10. The height-five rectangle spans the middle two bars.

```
largest_histogram([2,2])

L2 largest_histogram: stack = []
heights=[2, 2]; stack=[]

L3 largest_histogram: best = 0
heights=[2, 2]; stack=[]; best=0

L4 largest_histogram: for i in range(len(heights) + 1):
heights=[2, 2]; stack=[]; best=0; i=0

L5 largest_histogram: current = heights[i] if i < len(heights) else 0
heights=[2, 2]; stack=[]; best=0; i=0; current=2

L11 largest_histogram: stack.append(i)
heights=[2, 2]; stack=[0]; best=0; i=0; current=2

L4 largest_histogram: for i in range(len(heights) + 1):
heights=[2, 2]; stack=[0]; best=0; i=1; current=2

L11 largest_histogram: stack.append(i)
heights=[2, 2]; stack=[0, 1]; best=0; i=1; current=2

L4 largest_histogram: for i in range(len(heights) + 1):
heights=[2, 2]; stack=[0, 1]; best=0; i=2; current=2

... 3 intermediate state changes omitted ...

L9 largest_histogram: best = max(best, height * (i - left - 1))
heights=[2, 2]; stack=[0]; best=2; i=2; current=0; height=2; left=0

L7 largest_histogram: height = heights[stack.pop()]
heights=[2, 2]; stack=[]; best=2; i=2; current=0; height=2; left=0

L8 largest_histogram: left = stack[-1] if stack else -1
heights=[2, 2]; stack=[]; best=2; i=2; current=0; height=2; left=-1

L9 largest_histogram: best = max(best, height * (i - left - 1))
heights=[2, 2]; stack=[]; best=4; i=2; current=0; height=2; left=-1
```

Result: 4. Equal heights can combine into a wider rectangle.

Why the solution is correct

All bars between a popped bar's blockers are at least its height, so the computed rectangle is valid. For any optimal rectangle, one of its minimum-height bars is eventually popped with a span at least as wide, ensuring the optimum is considered. Sentinel processing prevents unfinished candidates.

Time and space

O(n) time and O(n) stack space.

Common mistakes

Width is not simply the number of stack entries. Forgetting the sentinel misses increasing histograms.

Reasoning card

State: stack contains indices in nondecreasing height order. Popping index j finalizes a rectangle of height heights[j] across all bars between its boundaries.

Invariant: Stack indices have increasing heights and unresolved right boundaries.

Decision to rehearse: A lower bar finalizes rectangles whose heights can no longer extend.

Finish: Push the current index

Transfer exercise

Return the rectangle's start, exclusive end and height, and define a deterministic tie policy for equal areas.

076. Maximal Rectangle in Binary Matrix

Bridge | Module 08

Problem

Return the area of the largest all-one axis-aligned rectangle in a rectangular matrix of integer zeros and ones. Empty input or zero columns returns zero. The matrix is unchanged. Reuse the histogram routine from kata 075.

Examples

Example 1. The final two rows form a 2-by-3 rectangle.

```
maximal_rectangle([[1,0,1],[1,1,1],[1,1,1]])
=> 6
```

Example 2. There is no bottom row.

```
maximal_rectangle([])
=> 0
```

Example 3. Every column height repeatedly resets to zero.

```
maximal_rectangle([[0,0],[0,0]])
=> 0
```

Why this mechanism works

At each row, let a column's height count consecutive ones ending at that row. Any all-one rectangle has a bottom row; at that row it appears as a rectangle under these heights. Solving one histogram per bottom row therefore searches all possible rectangles without enumerating every pair of corners.

State and boundaries

heights[c] is the vertical one-run ending at the current row. best is the maximum histogram area over processed bottom rows.

Protocol

1. Initialize column heights to zero.
2. Increment a height for one and reset it for zero.
3. Solve the current histogram.
4. Update the global maximum.

Visualization

```
row 1: 1 0 1 -> heights 1 0 1
row 2: 1 1 1 -> heights 2 1 2
row 3: 1 1 1 -> heights 3 2 3
last histogram supports area 6
```

Python implementation

```
from kata_075 import largest_histogram

def maximal_rectangle(matrix):
    if not matrix or not matrix[0]:
        return 0
    heights = [0] * len(matrix[0])
    best = 0
    for row in matrix:
        for c, value in enumerate(row):
            heights[c] = heights[c] + 1 if value == 1 else 0
        best = max(best, largest_histogram(heights))
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
maximal_rectangle([[1,0,1],[1,1,1],[1,1,1]])

L4 maximal_rectangle: if not matrix or not matrix[0]:
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]

L6 maximal_rectangle: heights = [0] * len(matrix[0])
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[0, 0, 0]

L7 maximal_rectangle: best = 0
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[0, 0, 0]; best=0

L8 maximal_rectangle: for row in matrix:
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[0, 0, 0]; best=0; row=[1, 0, 1]

L9 maximal_rectangle: for c, value in enumerate(row):
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[0, 0, 0]; best=0; row=[1, 0, 1]; c=0;
    value=1

L10 maximal_rectangle: heights[c] = heights[c] + 1 if value == 1 else 0
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[1, 0, 0]; best=0; row=[1, 0, 1]; c=0;
    value=1

L9 maximal_rectangle: for c, value in enumerate(row):
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[1, 0, 0]; best=0; row=[1, 0, 1]; c=1;
    value=0

L9 maximal_rectangle: for c, value in enumerate(row):
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[1, 0, 0]; best=0; row=[1, 0, 1]; c=2;
    value=1

... 13 intermediate state changes omitted ...

L10 maximal_rectangle: heights[c] = heights[c] + 1 if value == 1 else 0
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[3, 2, 2]; best=3; row=[1, 1, 1]; c=1;
```

```
    value=1

L9 maximal_rectangle: for c, value in enumerate(row):
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[3, 2, 2]; best=3; row=[1, 1, 1]; c=2;
    value=1

L10 maximal_rectangle: heights[c] = heights[c] + 1 if value == 1 else 0
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[3, 2, 3]; best=3; row=[1, 1, 1]; c=2;
    value=1

L11 maximal_rectangle: best = max(best, largest_histogram(heights))
matrix=[[1, 0, 1], [1, 1, 1], [1, 1, 1]]; heights=[3, 2, 3]; best=6; row=[1, 1, 1]; c=2;
    value=1
```

Result: 6. The final two rows form a 2-by-3 rectangle.

```
maximal_rectangle([])

L4 maximal_rectangle: if not matrix or not matrix[0]:
matrix=[]
```

Result: 0. There is no bottom row.

Why the solution is correct

The height recurrence exactly measures contiguous upward availability. A histogram rectangle corresponds to an all-one matrix rectangle ending at this row, and every matrix rectangle has such a bottom row. Taking the maximum over rows is therefore complete.

Time and space

O(rows*cols) time and O(cols) working space, including the reused histogram stack.

Common mistakes

Not resetting a height at zero allows rectangles to pass through blocked cells. Summing all column heights gives area without enforcing a shared minimum height.

Reasoning card

State: heights[c] is the vertical one-run ending at the current row. best is the maximum histogram area over processed bottom rows.

Invariant: Heights summarize consecutive ones ending at the current row.

Decision to rehearse: Update column heights, then reuse histogram logic.

Finish: Update the global maximum

Transfer exercise

Find the largest all-zero rectangle by changing only the height-update predicate.

077. Sum of Subarray Minimums

Bridge | Module 08

Problem

Return the exact sum of the minimum value of every nonempty contiguous subarray. Values are integers and may repeat or be negative. No modulo reduction is requested. Empty input returns zero.

Examples

Example 1. All ten subarray minima sum to seventeen.

```
sum_subarray_minimums([3,1,2,4])
=> 17
```

Example 2. Duplicate ownership counts each of three subarrays once.

```
sum_subarray_minimums([2,2])
=> 6
```

Example 3. There are no nonempty subarrays.

```
sum_subarray_minimums([])
=> 0
```

Why this mechanism works

Count how many subarrays choose each position as their minimum. Duplicate minima require a unique owner: choose the rightmost equal minimum. Find the previous strictly smaller position and the next smaller-or-equal position. Their distances give independent choices for the subarray's left and right boundaries.

State and boundaries

previous[i] is the nearest strictly smaller index to the left; following[i] is the nearest <= index to the right.

Protocol

1. Scan left to right, popping >= values to find previous strict blockers.

2. Scan right to left, popping > values to find following non-strict blockers.

3. Add nums[i]*(i-previous[i])*(following[i]-i) for every i.

Visualization

```
[2,2] -> rightmost minimum owns [0,2)
index 0 owns [0,1)
index 1 owns [1,2) and [0,2)
sum=2+2+2=6
```

Python implementation

```
def sum_subarray_minimums(nums):
    n = len(nums)
    previous, following = [-1] * n, [n] * n
    stack = []
    for i in range(n):
        while stack and nums[stack[-1]] >= nums[i]:
            stack.pop()
        previous[i] = stack[-1] if stack else -1
        stack.append(i)
    stack = []
    for i in range(n - 1, -1, -1):
        while stack and nums[stack[-1]] > nums[i]:
            stack.pop()
        following[i] = stack[-1] if stack else n
        stack.append(i)
    return sum(value * (i - previous[i]) * (following[i] - i)
               for i, value in enumerate(nums))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
sum_subarray_minimums([3,1,2,4])

L2 sum_subarray_minimums: n = len(nums)
nums=[3, 1, 2, 4]; n=4

L3 sum_subarray_minimums: previous, following = [-1] * n, [n] * n
nums=[3, 1, 2, 4]; n=4; following=[4, 4, 4, 4]; previous=[-1, -1, -1, -1]

L4 sum_subarray_minimums: stack = []
nums=[3, 1, 2, 4]; n=4; following=[4, 4, 4, 4]; previous=[-1, -1, -1, -1]; stack=[]

L5 sum_subarray_minimums: for i in range(n):
nums=[3, 1, 2, 4]; n=4; following=[4, 4, 4, 4]; previous=[-1, -1, -1, -1]; stack=[]; i=0

L9 sum_subarray_minimums: stack.append(i)
nums=[3, 1, 2, 4]; n=4; following=[4, 4, 4, 4]; previous=[-1, -1, -1, -1]; stack=[0]; i=0

L5 sum_subarray_minimums: for i in range(n):
nums=[3, 1, 2, 4]; n=4; following=[4, 4, 4, 4]; previous=[-1, -1, -1, -1]; stack=[0]; i=1

L7 sum_subarray_minimums: stack.pop()
nums=[3, 1, 2, 4]; n=4; following=[4, 4, 4, 4]; previous=[-1, -1, -1, -1]; stack=[]; i=1

L9 sum_subarray_minimums: stack.append(i)
nums=[3, 1, 2, 4]; n=4; following=[4, 4, 4, 4]; previous=[-1, -1, -1, -1]; stack=[1]; i=1

... 19 intermediate state changes omitted ...
```

```
L17 <genexpr>: for i, value in enumerate(nums))
.0=enumerate; following=[1, 4, 4, 4]; previous=[-1, -1, 1, 2]; i=1; value=1

L17 <genexpr>: for i, value in enumerate(nums))
.0=enumerate; following=[1, 4, 4, 4]; previous=[-1, -1, 1, 2]; i=2; value=2

L17 <genexpr>: for i, value in enumerate(nums))
.0=enumerate; following=[1, 4, 4, 4]; previous=[-1, -1, 1, 2]; i=3; value=4

L17 <genexpr>: for i, value in enumerate(nums))
.0=enumerate; following=[1, 4, 4, 4]; previous=[-1, -1, 1, 2]; i=3; value=4
```

Result: 17. All ten subarray minima sum to seventeen.

```
sum_subarray_minimums([2,2])

L2 sum_subarray_minimums: n = len(nums)
nums=[2, 2]; n=2

L3 sum_subarray_minimums: previous, following = [-1] * n, [n] * n
nums=[2, 2]; n=2; following=[2, 2]; previous=[-1, -1]

L4 sum_subarray_minimums: stack = []
nums=[2, 2]; n=2; following=[2, 2]; previous=[-1, -1]; stack=[]

L5 sum_subarray_minimums: for i in range(n):
nums=[2, 2]; n=2; following=[2, 2]; previous=[-1, -1]; stack=[]; i=0

L9 sum_subarray_minimums: stack.append(i)
nums=[2, 2]; n=2; following=[2, 2]; previous=[-1, -1]; stack=[0]; i=0

L5 sum_subarray_minimums: for i in range(n):
nums=[2, 2]; n=2; following=[2, 2]; previous=[-1, -1]; stack=[0]; i=1

L7 sum_subarray_minimums: stack.pop()
nums=[2, 2]; n=2; following=[2, 2]; previous=[-1, -1]; stack=[]; i=1

L9 sum_subarray_minimums: stack.append(i)
nums=[2, 2]; n=2; following=[2, 2]; previous=[-1, -1]; stack=[1]; i=1

... 5 intermediate state changes omitted ...

L16 <genexpr>: return sum(value * (i - previous[i]) * (following[i] - i)
.0=enumerate; following=[1, 2]; previous=[-1, -1]

L17 <genexpr>: for i, value in enumerate(nums))
.0=enumerate; following=[1, 2]; previous=[-1, -1]; i=0; value=2

L17 <genexpr>: for i, value in enumerate(nums))
.0=enumerate; following=[1, 2]; previous=[-1, -1]; i=1; value=2

L17 <genexpr>: for i, value in enumerate(nums))
.0=enumerate; following=[1, 2]; previous=[-1, -1]; i=1; value=2
```

Result: 6. Duplicate ownership counts each of three subarrays once.

Why the solution is correct

Between these blockers, every value is at least nums[i], and no equal minimum to its right is included. Thus i owns exactly the subarrays choosing a left boundary after previous[i] and an end before following[i]. Every subarray has one rightmost minimum, so contributions partition the set without duplication.

Time and space

O(n) time and O(n) space. Integer multiplication costs depend on magnitude beyond the unit-cost model.

Common mistakes

Using strict comparisons on both sides double-counts tied minima; using non-strict comparisons on both can miss them. Tie ownership is part of the proof.

Reasoning card

State: previous[i] is the nearest strictly smaller index to the left; following[i] is the nearest <= index to the right.

Invariant: Tie rules assign every subarray to exactly one minimum index.

Decision to rehearse: Find how many subarrays choose each element as minimum using previous/next smaller boundaries.

Finish: Add nums[i]*(i-previous[i])*(following[i]-i) for every i

Transfer exercise

Subtract summed minima from summed maxima to obtain the sum of all subarray ranges.

078. Sliding Window Maximum

Interview | Module 08

Problem

Return the maximum of every contiguous window of exactly k integers. If k<1 or k exceeds input length, return []. Windows are reported in left-to-right order, and the input is unchanged.

Examples

Example 1. Expiration and domination maintain each window frontier.

```
window_maximum([1,3,-1,-3,5,3,6,7], 3)
=> [3, 3, 5, 5, 6, 7]
```

Example 2. The newer equal value safely replaces the older one.

```
window_maximum([2,2,2], 2)
=> [2, 2]
```

Example 3. No complete window exists.

```
window_maximum([], 1)
=> []
```

Why this mechanism works

A later value at least as large as an earlier candidate dominates it: the later value is better or equal and expires later. Keep only undominated indices in a deque with decreasing values. The front is the current maximum, while front expiration and back domination perform different jobs.

State and boundaries

candidates has increasing indices and strictly decreasing values. Every retained index belongs to the active window.

Protocol

1. Expire indices at or before right-k.

2. Remove back candidates no larger than the incoming value.

3. Append the incoming index.

4. Once k values have arrived, emit the front value.

Visualization

```
window [1,3,-1] -> deque values [3,-1]
new -3: [3,-1,-3]
new 5 dominates the back until deque [5]
```

Python implementation

```
from collections import deque

def window_maximum(nums, k):
    if k < 1 or k > len(nums):
        return []
    candidates = deque()
    output = []
    for right, value in enumerate(nums):
        while candidates and candidates[0] <= right - k:
            candidates.popleft()
        while candidates and nums[candidates[-1]] <= value:
            candidates.pop()
        candidates.append(right)
        if right + 1 >= k:
            output.append(nums[candidates[0]])
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
window_maximum([1,3,-1,-3,5,3,6,7], 3)

L4 window_maximum: if k < 1 or k > len(nums):
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3

L6 window_maximum: candidates = deque()
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([])

L7 window_maximum: output = []
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([]); output=[]

L8 window_maximum: for right, value in enumerate(nums):
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([]); output=[]; right=0; value=1

L13 window_maximum: candidates.append(right)
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([0]); output=[]; right=0; value=1

L8 window_maximum: for right, value in enumerate(nums):
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([0]); output=[]; right=1; value=3

L12 window_maximum: candidates.pop()
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([]); output=[]; right=1; value=3

L13 window_maximum: candidates.append(right)
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([1]); output=[]; right=1; value=3

... 20 intermediate state changes omitted ...

L8 window_maximum: for right, value in enumerate(nums):
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([6]); output=[3, 3, 5, 5, 6]; right=7;
```

```
    value=7

L12 window_maximum: candidates.pop()
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([]); output=[3, 3, 5, 5, 6]; right=7;
    value=7

L13 window_maximum: candidates.append(right)
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([7]); output=[3, 3, 5, 5, 6]; right=7;
    value=7

L15 window_maximum: output.append(nums[candidates[0]])
nums=[1, 3, -1, -3, 5, 3, 6, 7]; k=3; candidates=deque([7]); output=[3, 3, 5, 5, 6, 7];
    right=7; value=7
```

Result: [3, 3, 5, 5, 6, 7]. Expiration and domination maintain each window frontier.

```
window_maximum([2,2,2], 2)

L4 window_maximum: if k < 1 or k > len(nums):
nums=[2, 2, 2]; k=2

L6 window_maximum: candidates = deque()
nums=[2, 2, 2]; k=2; candidates=deque([])

L7 window_maximum: output = []
nums=[2, 2, 2]; k=2; candidates=deque([]); output=[]

L8 window_maximum: for right, value in enumerate(nums):
nums=[2, 2, 2]; k=2; candidates=deque([]); output=[]; right=0; value=2

L13 window_maximum: candidates.append(right)
nums=[2, 2, 2]; k=2; candidates=deque([0]); output=[]; right=0; value=2

L8 window_maximum: for right, value in enumerate(nums):
nums=[2, 2, 2]; k=2; candidates=deque([0]); output=[]; right=1; value=2

L12 window_maximum: candidates.pop()
nums=[2, 2, 2]; k=2; candidates=deque([]); output=[]; right=1; value=2

L13 window_maximum: candidates.append(right)
nums=[2, 2, 2]; k=2; candidates=deque([1]); output=[]; right=1; value=2

... 1 intermediate state changes omitted ...

L8 window_maximum: for right, value in enumerate(nums):
nums=[2, 2, 2]; k=2; candidates=deque([1]); output=[2]; right=2; value=2

L12 window_maximum: candidates.pop()
nums=[2, 2, 2]; k=2; candidates=deque([]); output=[2]; right=2; value=2

L13 window_maximum: candidates.append(right)
nums=[2, 2, 2]; k=2; candidates=deque([2]); output=[2]; right=2; value=2

L15 window_maximum: output.append(nums[candidates[0]])
nums=[2, 2, 2]; k=2; candidates=deque([2]); output=[2, 2]; right=2; value=2
```

Result: [2, 2]. The newer equal value safely replaces the older one.

Why the solution is correct

Removing an expired index is mandatory. Removing a dominated candidate is safe because its later replacement remains available whenever the older one would be relevant. The largest retained value stays at the front and represents the active window maximum.

Time and space

O(n) time and O(k) auxiliary deque space, plus output.

Common mistakes

Storing values alone makes expiration ambiguous with duplicates. A heap without removing stale entries can report an out-of-window maximum.

Reasoning card

State: candidates has increasing indices and strictly decreasing values. Every retained index belongs to the active window.

Invariant: Deque indices are in-window and values decrease front to back.

Decision to rehearse: Expire out-of-window indices; remove dominated values from the back.

Finish: Once k values have arrived, emit the front value

Transfer exercise

Maintain both minimum and maximum deques to find the longest window whose range is at most a limit.

079. Shortest Subarray Sum At Least K with Negatives

Interview | Module 08

Problem

Find the minimum length of a nonempty contiguous subarray with sum at least k when integers may have either sign. Return -1 if none exists. k may be any integer; the empty interval is never a candidate.

Examples

Example 1. The whole array is the only qualifying interval.

```
shortest_sum_at_least([2,-1,2], 3)
=> 3
```

Example 2. The final three values reach the target.

```
shortest_sum_at_least([84,-37,32,40,95], 167)
=> 3
```

Example 3. Even a nonpositive target does not permit an empty interval.

```
shortest_sum_at_least([], 0)
=> -1
```

Why this mechanism works

Negative values break ordinary window monotonicity, so move to prefix sums. A later prefix no greater than an earlier prefix dominates it as a future start: it yields at least as much sum with a shorter interval. Keep increasing prefix values in a deque. Qualifying starts are removed from the front after recording their shortest opportunity.

State and boundaries

prefix boundaries in candidates increase in both index and prefix value. best is the shortest qualifying interval already recorded.

Protocol

1. Build leading-zero prefix sums.

2. For each boundary first remove qualifying front starts while updating best.

3. Remove back starts with prefix>=current prefix.

4. Append the current boundary only after querying.

5. Return -1 if no candidate was recorded.

Visualization

```
nums [2,-1,2], prefixes 0,2,1,3
prefix 1 dominates earlier prefix 2
current 3 minus start 0 reaches k=3
length 3
```

Python implementation

```
from collections import deque

def shortest_sum_at_least(nums, k):
    prefix = [0]
    for value in nums:
        prefix.append(prefix[-1] + value)
    candidates = deque()
    best = len(nums) + 1
    for right, total in enumerate(prefix):
        while candidates and total - prefix[candidates[0]] >= k:
            best = min(best, right - candidates.popleft())
        while candidates and prefix[candidates[-1]] >= total:
            candidates.pop()
        candidates.append(right)
    return -1 if best > len(nums) else best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
shortest_sum_at_least([2,-1,2], 3)

L4 shortest_sum_at_least: prefix = [0]
nums=[2, -1, 2]; k=3; prefix=[0]

L5 shortest_sum_at_least: for value in nums:
nums=[2, -1, 2]; k=3; prefix=[0]; value=2

L6 shortest_sum_at_least: prefix.append(prefix[-1] + value)
nums=[2, -1, 2]; k=3; prefix=[0, 2]; value=2

L5 shortest_sum_at_least: for value in nums:
nums=[2, -1, 2]; k=3; prefix=[0, 2]; value=-1

L6 shortest_sum_at_least: prefix.append(prefix[-1] + value)
nums=[2, -1, 2]; k=3; prefix=[0, 2, 1]; value=-1

L5 shortest_sum_at_least: for value in nums:
nums=[2, -1, 2]; k=3; prefix=[0, 2, 1]; value=2

L6 shortest_sum_at_least: prefix.append(prefix[-1] + value)
nums=[2, -1, 2]; k=3; prefix=[0, 2, 1, 3]; value=2

L7 shortest_sum_at_least: candidates = deque()
nums=[2, -1, 2]; k=3; prefix=[0, 2, 1, 3]; value=2; candidates=deque([])

... 7 intermediate state changes omitted ...
```

```
L14 shortest_sum_at_least: candidates.append(right)
nums=[2, -1, 2]; k=3; prefix=[0, 2, 1, 3]; value=2; candidates=deque([0, 2]); best=4; right=2;
     total=1

L9 shortest_sum_at_least: for right, total in enumerate(prefix):
nums=[2, -1, 2]; k=3; prefix=[0, 2, 1, 3]; value=2; candidates=deque([0, 2]); best=4; right=3;
     total=3

L11 shortest_sum_at_least: best = min(best, right - candidates.popleft())
nums=[2, -1, 2]; k=3; prefix=[0, 2, 1, 3]; value=2; candidates=deque([2]); best=3; right=3;
    total=3

L14 shortest_sum_at_least: candidates.append(right)
nums=[2, -1, 2]; k=3; prefix=[0, 2, 1, 3]; value=2; candidates=deque([2, 3]); best=3; right=3;
     total=3
```

Result: 3. The whole array is the only qualifying interval.

```
shortest_sum_at_least([84,-37,32,40,95], 167)

L4 shortest_sum_at_least: prefix = [0]
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0]

L5 shortest_sum_at_least: for value in nums:
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0]; value=84

L6 shortest_sum_at_least: prefix.append(prefix[-1] + value)
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84]; value=84

L5 shortest_sum_at_least: for value in nums:
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84]; value=-37

L6 shortest_sum_at_least: prefix.append(prefix[-1] + value)
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84, 47]; value=-37

L5 shortest_sum_at_least: for value in nums:
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84, 47]; value=32

L6 shortest_sum_at_least: prefix.append(prefix[-1] + value)
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84, 47, 79]; value=32

L5 shortest_sum_at_least: for value in nums:
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84, 47, 79]; value=40

... 16 intermediate state changes omitted ...

L9 shortest_sum_at_least: for right, total in enumerate(prefix):
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84, 47, 79, 119, 214]; value=95;
    candidates=deque([0, 2, 3, 4]); best=6; right=5; total=214

L11 shortest_sum_at_least: best = min(best, right - candidates.popleft())
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84, 47, 79, 119, 214]; value=95;
    candidates=deque([2, 3, 4]); best=5; right=5; total=214

L11 shortest_sum_at_least: best = min(best, right - candidates.popleft())
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84, 47, 79, 119, 214]; value=95;
    candidates=deque([3, 4]); best=3; right=5; total=214

L14 shortest_sum_at_least: candidates.append(right)
nums=[84, -37, 32, 40, 95]; k=167; prefix=[0, 84, 47, 79, 119, 214]; value=95;
    candidates=deque([3, 4, 5]); best=3; right=5; total=214
```

Result: 3. The final three values reach the target.

Why the solution is correct

A front start that qualifies cannot yield a shorter interval at a later end, so it is safely finalized. A back start dominated by a later lower prefix can never outperform its replacement at a future end. Querying before appending excludes empty intervals and preserves correctness even for nonpositive k.

Time and space

O(n) time and O(n) space; every prefix boundary is inserted and removed at most once.

Common mistakes

Applying the positive-only shrink rule fails with negative values. Appending the current boundary before testing can admit an empty interval.

Reasoning card

State: prefix boundaries in candidates increase in both index and prefix value. best is the shortest qualifying interval already recorded.

Invariant: Deque contains useful prefix candidates in increasing value order.

Decision to rehearse: Maintain increasing prefix sums in a deque and resolve valid front boundaries.

Finish: Return -1 if no candidate was recorded

Transfer exercise

Explain each deque removal as a distinct proof: completed opportunity at the front and future dominance at the back.

080. Remove K Digits

Interview | Module 08

Problem

Remove exactly k digits from a nonempty string of decimal digits, preserving the order of remaining digits, to minimize the resulting nonnegative number. 0<=k<=len(number). Normalize leading zeros afterward and return '0' if no nonzero representation remains.

Examples

Example 1. Three improving deletions are spent at descents.

```
remove_k_digits("1432219", 3)
=> '1219'
```

Example 2. Removing one leaves 0200, normalized to 200.

```
remove_k_digits("10200", 1)
=> '200'
```

Example 3. Removing all digits uses the designated zero representation.

```
remove_k_digits("10", 2)
=> '0'
```

Why this mechanism works

Earlier digits have greater significance. If a larger retained digit precedes a smaller incoming one and deletion budget remains, removing the larger digit improves the earliest differing position. Keep a nondecreasing stack where budget permits. If budget remains after scanning, remove digits from the right, where they are least useful to retain.

State and boundaries

stack is the best retained prefix under deletions already spent. remaining is the number of required deletions still available.

Protocol

1. Pop larger trailing digits while budget remains.

2. Append the incoming digit.

3. After scanning remove remaining digits from the tail.

4. Strip representation-only leading zeros.

5. Return zero for an empty normalized result.

Visualization

```
1432219, k=3
4 removed by 3; 3 removed by 2
later 2 removed by 1
retained 1219
```

Python implementation

```
def remove_k_digits(number, k):
    stack = []
    remaining = k
    for digit in number:
        while remaining and stack and stack[-1] > digit:
            stack.pop()
            remaining -= 1
        stack.append(digit)
    if remaining:
        del stack[-remaining:]
    return ''.join(stack).lstrip('0') or '0'
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
remove_k_digits("1432219", 3)

L2 remove_k_digits: stack = []
number='1432219'; k=3; stack=[]

L3 remove_k_digits: remaining = k
number='1432219'; k=3; stack=[]; remaining=3

L4 remove_k_digits: for digit in number:
number='1432219'; k=3; stack=[]; remaining=3; digit='1'

L8 remove_k_digits: stack.append(digit)
number='1432219'; k=3; stack=['1']; remaining=3; digit='1'

L4 remove_k_digits: for digit in number:
number='1432219'; k=3; stack=['1']; remaining=3; digit='4'

L8 remove_k_digits: stack.append(digit)
number='1432219'; k=3; stack=['1', '4']; remaining=3; digit='4'

L4 remove_k_digits: for digit in number:
number='1432219'; k=3; stack=['1', '4']; remaining=3; digit='3'

L6 remove_k_digits: stack.pop()
number='1432219'; k=3; stack=['1']; remaining=3; digit='3'

... 9 intermediate state changes omitted ...

L7 remove_k_digits: remaining -= 1
number='1432219'; k=3; stack=['1', '2']; remaining=0; digit='1'

L8 remove_k_digits: stack.append(digit)
```

```
number='1432219'; k=3; stack=['1', '2', '1']; remaining=0; digit='1'

L4 remove_k_digits: for digit in number:
number='1432219'; k=3; stack=['1', '2', '1']; remaining=0; digit='9'

L8 remove_k_digits: stack.append(digit)
number='1432219'; k=3; stack=['1', '2', '1', '9']; remaining=0; digit='9'
```

Result: '1219'. Three improving deletions are spent at descents.

```
remove_k_digits("10200", 1)

L2 remove_k_digits: stack = []
number='10200'; k=1; stack=[]

L3 remove_k_digits: remaining = k
number='10200'; k=1; stack=[]; remaining=1

L4 remove_k_digits: for digit in number:
number='10200'; k=1; stack=[]; remaining=1; digit='1'

L8 remove_k_digits: stack.append(digit)
number='10200'; k=1; stack=['1']; remaining=1; digit='1'

L4 remove_k_digits: for digit in number:
number='10200'; k=1; stack=['1']; remaining=1; digit='0'

L6 remove_k_digits: stack.pop()
number='10200'; k=1; stack=[]; remaining=1; digit='0'

L7 remove_k_digits: remaining -= 1
number='10200'; k=1; stack=[]; remaining=0; digit='0'

L8 remove_k_digits: stack.append(digit)
number='10200'; k=1; stack=['0']; remaining=0; digit='0'

... 1 intermediate state changes omitted ...

L8 remove_k_digits: stack.append(digit)
number='10200'; k=1; stack=['0', '2']; remaining=0; digit='2'

L4 remove_k_digits: for digit in number:
number='10200'; k=1; stack=['0', '2']; remaining=0; digit='0'

L8 remove_k_digits: stack.append(digit)
number='10200'; k=1; stack=['0', '2', '0']; remaining=0; digit='0'

L8 remove_k_digits: stack.append(digit)
number='10200'; k=1; stack=['0', '2', '0', '0']; remaining=0; digit='0'
```

Result: '200'. Removing one leaves 0200, normalized to 200.

Why the solution is correct

Replacing an earlier larger digit with a later smaller digit decreases the number at its first changed position, dominating all later choices. If no such descent remains, deleting from the suffix preserves the smallest available leading digits. Normalization changes representation, not the deletion choice.

Time and space

O(n) time and O(n) stack/output space.

Common mistakes

Using >= spends deletions unnecessarily on equal digits. Leading-zero normalization is separate from the exactly-k deletion budget.

Reasoning card

State: stack is the best retained prefix under deletions already spent. remaining is the number of required deletions still available.

Invariant: Stack is the smallest possible prefix after allowed removals so far.

Decision to rehearse: A smaller incoming digit invalidates larger unresolved previous digits.

Finish: Return zero for an empty normalized result

Transfer exercise

Maximize the number instead. Reverse the dominance comparison and justify the remaining-suffix deletion rule again.

Module 09
Queues, Heaps, and Streaming Frontiers

Choose FIFO for breadth/order and heaps for the next globally relevant candidate or a bounded winner pool.

Choose a frontier by the next event you need. Queues expose the oldest item, stacks the newest, and heaps the highest-priority candidate. Streaming top-k and medians retain summaries instead of the whole history. Distinguish strict per-operation cost from amortized cost across a sequence.

The ten-step progression

081. Queue Using Two Stacks

082. Stack Using One Queue

083. Recent Counter

084. Largest K Values from a Stream

085. Smallest K Values from a Stream

086. Kth Largest Stream Class

087. Top K Frequent Values

088. Merge K Sorted Arrays

089. K Closest Points to Origin

090. Running Median

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

081. Queue Using Two Stacks

Foundation | Module 09

Problem

Implement a FIFO queue using two stacks, with push(value), pop(), peek() and empty(). push returns None; pop removes and returns the oldest value. pop and peek on an empty queue raise IndexError. Use only stack-end operations internally.

Examples

Example 1. A new incoming value waits behind older outgoing values.

```
run_operations(TwoStackQueue,
    [("push",1),("push",2),("peek",),("pop",),("push",3),("pop",),("pop",),("empty",)]
    )
=> [None, None, 1, 1, None, 2, 3, True]
```

Example 2. Both stacks begin empty.

```
run_operations(TwoStackQueue, [("empty",)])
=> [True]
```

Example 3. Zero is a stored value, not an empty marker.

```
run_operations(TwoStackQueue, [("push",0),("peek",),("empty",)])
=> [None, 0, False]
```

Why this mechanism works

The incoming stack collects new values in reverse departure order. Moving all of them to an empty outgoing stack reverses that order again, placing the oldest at its top. Never transfer into a nonempty outgoing stack, because its older values must leave first.

State and boundaries

outgoing holds the older queue segment in pop order; incoming holds the newer segment in insertion order from bottom to top.

Protocol

1. Push new values onto incoming.
2. Before pop or peek, transfer only if outgoing is empty.
3. Pop or inspect outgoing's top.
4. Report empty only when both stacks are empty.

Visualization

```
push 1,2,3 -> incoming [1,2,3]
transfer -> outgoing [3,2,1]
pop top -> 1, then 2, then 3
```

Python implementation

```
from structures import run_operations

class TwoStackQueue:
    def __init__(self):
        self.incoming, self.outgoing = [], []

    def push(self, value):
        self.incoming.append(value)

    def _prepare(self):
        if not self.outgoing:
            while self.incoming:
                self.outgoing.append(self.incoming.pop())

    def pop(self):
        self._prepare()
        return self.outgoing.pop()

    def peek(self):
        self._prepare()
        return self.outgoing[-1]

    def empty(self):
        return not self.incoming and not self.outgoing
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(TwoStackQueue,
    [("push",1),("push",2),("peek",),("pop",),("push",3),("pop",),("pop",),("empty",)]
    )

L5 __init__: self.incoming, self.outgoing = [], []
self=TwoStackQueue{'incoming':[], 'outgoing':[]}

L8 push: self.incoming.append(value)
self=TwoStackQueue{'incoming':[...], 'outgoing':[]}; value=1

L8 push: self.incoming.append(value)
self=TwoStackQueue{'incoming':[..., ...], 'outgoing':[]}; value=2

L11 _prepare: if not self.outgoing:
self=TwoStackQueue{'incoming':[..., ...], 'outgoing':[]}

L13 _prepare: self.outgoing.append(self.incoming.pop())
self=TwoStackQueue{'incoming':[...], 'outgoing':[...]}

L13 _prepare: self.outgoing.append(self.incoming.pop())
self=TwoStackQueue{'incoming':[], 'outgoing':[..., ...]}
```

```
L20 peek: self._prepare()
self=TwoStackQueue{'incoming':[], 'outgoing':[..., ...]}

L11 _prepare: if not self.outgoing:
self=TwoStackQueue{'incoming':[], 'outgoing':[..., ...]}

... 7 intermediate state changes omitted ...

L13 _prepare: self.outgoing.append(self.incoming.pop())
self=TwoStackQueue{'incoming':[], 'outgoing':[...]}

L16 pop: self._prepare()
self=TwoStackQueue{'incoming':[], 'outgoing':[...]}

L17 pop: return self.outgoing.pop()
self=TwoStackQueue{'incoming':[], 'outgoing':[]}

L24 empty: return not self.incoming and not self.outgoing
self=TwoStackQueue{'incoming':[], 'outgoing':[]}
```

Result: [None, None, 1, 1, None, 2, 3, True]. A new incoming value waits behind older outgoing values.

```
run_operations(TwoStackQueue, [("empty",)])

L5 __init__: self.incoming, self.outgoing = [], []
self=TwoStackQueue{'incoming':[], 'outgoing':[]}

L24 empty: return not self.incoming and not self.outgoing
self=TwoStackQueue{'incoming':[], 'outgoing':[]}
```

Result: [True]. Both stacks begin empty.

Why the solution is correct

A transfer reverses the entire pending incoming segment, making its oldest element accessible first. Existing outgoing values predate all incoming values, so exhausting them before another transfer preserves FIFO order across all operations.

Time and space

O(1) amortized time per operation and O(n) space. One transfer-triggering operation may take O(n), but each value transfers at most once.

Common mistakes

Transferring on every pop can reorder pending elements. Calling every operation worst-case O(1) confuses amortized and individual costs.

Reasoning card

State: outgoing holds the older queue segment in pop order; incoming holds the newer segment in insertion order from bottom to top.

Invariant: Output-stack top is the oldest queued item.

Decision to rehearse: Transfer input to output only when output is empty.

Finish: Report empty only when both stacks are empty

Transfer exercise

Prove the amortized bound with a potential proportional to the number of elements in incoming.

082. Stack Using One Queue

Foundation | Module 09

Problem

Implement a LIFO stack using one FIFO queue, supporting push(value), pop(), top() and empty(). push returns None; pop returns the most recently pushed value. Empty pop and top raise IndexError. Queue operations are append at the back and remove at the front.

Examples

Example 1. The most recent insertion is at the queue front.

```
run_operations(OneQueueStack, [("push",1),("push",2),("top",),("pop",),("pop",)])
=> [None, None, 2, 2, 1]
```

Example 2. A fresh queue is an empty stack.

```
run_operations(OneQueueStack, [("empty",)])
=> [True]
```

Example 3. Removing the only value restores emptiness.

```
run_operations(OneQueueStack, [("push",7),("pop",),("empty",)])
=> [None, 7, True]
```

Why this mechanism works

Make push expensive so pop stays simple. After enqueuing a new value, rotate all older values behind it. The newest element becomes the front, and the older stack order is preserved behind that front.

State and boundaries

queue order from front to back equals stack order from top to bottom.

Protocol

1. Enqueue the new value.
2. Rotate exactly the previous queue size from front to back.
3. Pop or inspect the front for stack operations.
4. Test queue emptiness.

Visualization

```
existing top-to-bottom [2,1]
append 3 -> [2,1,3]
rotate twice -> [3,2,1]
```

Python implementation

```
from collections import deque
from structures import run_operations

class OneQueueStack:
    def __init__(self):
        self.queue = deque()

    def push(self, value):
        old_size = len(self.queue)
        self.queue.append(value)
        for _ in range(old_size):
            self.queue.append(self.queue.popleft())

    def pop(self):
        return self.queue.popleft()

    def top(self):
        return self.queue[0]

    def empty(self):
        return not self.queue
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(OneQueueStack, [("push",1),("push",2),("top",),("pop",),("pop",)])

L6 __init__: self.queue = deque()
self=OneQueueStack{'queue':deque([])}

L9 push: old_size = len(self.queue)
self=OneQueueStack{'queue':deque([])}; value=1; old_size=0

L10 push: self.queue.append(value)
self=OneQueueStack{'queue':deque([...])}; value=1; old_size=0

L9 push: old_size = len(self.queue)
self=OneQueueStack{'queue':deque([...])}; value=2; old_size=1

L10 push: self.queue.append(value)
self=OneQueueStack{'queue':deque([..., ...])}; value=2; old_size=1

L11 push: for _ in range(old_size):
self=OneQueueStack{'queue':deque([..., ...])}; value=2; old_size=1; _=0

L18 top: return self.queue[0]
self=OneQueueStack{'queue':deque([..., ...])}

L15 pop: return self.queue.popleft()
self=OneQueueStack{'queue':deque([...])}
```

```
L15 pop: return self.queue.popleft()
self=OneQueueStack{'queue':deque([])}
```

Result: [None, None, 2, 2, 1]. The most recent insertion is at the queue front.

```
run_operations(OneQueueStack, [("empty",)])

L6 __init__: self.queue = deque()
self=OneQueueStack{'queue':deque([])}

L21 empty: return not self.queue
self=OneQueueStack{'queue':deque([])}
```

Result: [True]. A fresh queue is an empty stack.

Why the solution is correct

Before a push, the queue already represents top-to-bottom order. Moving all older elements behind the new one creates exactly the new stack order without changing the older relative order. Front removal therefore implements LIFO.

Time and space

O(n) time for push with n existing elements; O(1) pop, top and empty; O(n) storage.

Common mistakes

Rotating the new queue length rather than the old length returns the queue to its original order. Using a second queue violates this kata's constraint.

Reasoning card

State: queue order from front to back equals stack order from top to bottom.

Invariant: Queue front is always the stack top.

Decision to rehearse: Rotate older elements behind the newest on push.

Finish: Test queue emptiness

Transfer exercise

Choose cheap push and expensive pop instead; specify the queue representation and the tradeoff per operation.

083. Recent Counter

Foundation | Module 09

Problem

Implement RecentCounter.ping(t) for strictly increasing integer timestamps. Return the number of pings in the inclusive interval [t-3000,t], including the current ping. The class starts empty. Timestamps outside the current interval never need to be counted again.

Examples

Example 1. The lower boundary is inclusive.

```
run_operations(RecentCounter, [("ping",1),("ping",100),("ping",3001),("ping",3002)])
=> [1, 2, 3, 3]
```

Example 2. No pings produce no counts.

```
run_operations(RecentCounter, [])
=> []
```

Example 3. The old ping has fully expired.

```
run_operations(RecentCounter, [("ping",0),("ping",4000)])
=> [1, 1]
```

Why this mechanism works

Increasing timestamps mean expiration happens in FIFO order. Keep only the active interval in a deque. The left endpoint is inclusive, so discard times strictly below t-3000, not times equal to it.

State and boundaries

times contains exactly the unexpired pings in arrival order after repair.

Protocol

1. Append the current timestamp.
2. Remove front timestamps smaller than t-3000.
3. Return the repaired queue length.

Visualization

```
t=3001 -> interval [1,3001]
pings 1,100,3001 all remain
t=3002 -> interval [2,3002], so 1 expires
```

Python implementation

```
from collections import deque
from structures import run_operations

class RecentCounter:
    def __init__(self):
        self.times = deque()

    def ping(self, t):
        self.times.append(t)
        while self.times[0] < t - 3000:
            self.times.popleft()
        return len(self.times)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(RecentCounter, [("ping",1),("ping",100),("ping",3001),("ping",3002)])

L6 __init__: self.times = deque()
self=RecentCounter{'times':deque([])}

L9 ping: self.times.append(t)
self=RecentCounter{'times':deque([...])}; t=1

L9 ping: self.times.append(t)
self=RecentCounter{'times':deque([..., ...])}; t=100

L9 ping: self.times.append(t)
self=RecentCounter{'times':deque([..., ..., ...])}; t=3001

L9 ping: self.times.append(t)
self=RecentCounter{'times':deque([..., ..., ..., ...])}; t=3002

L11 ping: self.times.popleft()
self=RecentCounter{'times':deque([..., ..., ...])}; t=3002
```

Result: [1, 2, 3, 3]. The lower boundary is inclusive.

```
run_operations(RecentCounter, [])

L6 __init__: self.times = deque()
self=RecentCounter{'times':deque([])}
```

Result: []. No pings produce no counts.

Why the solution is correct

The new ping belongs to its own interval. All expired pings form a prefix because arrivals are ordered, and future timestamps can never make those old pings relevant again. Removing that prefix leaves exactly the requested inclusive interval.

Time and space

O(1) amortized time per ping and O(w) space for the pings retained in the active window.

Common mistakes

Using <= expires the inclusive lower endpoint incorrectly. Out-of-order timestamps would invalidate FIFO expiration.

Reasoning card

State: times contains exactly the unexpired pings in arrival order after repair.

Invariant: Queue contains exactly pings in the active time window.

Decision to rehearse: Expire old timestamps from the front before reporting.

Finish: Return the repaired queue length

Transfer exercise

Support out-of-order timestamps. Identify which assumptions must change before choosing a replacement data structure.

084. Largest K Values from a Stream

Foundation | Module 09

Problem

Consume an integer stream and return its largest min(k,n) values in descending order, retaining multiplicity. k<=0 returns []. Keep at most k candidates while scanning; the entire stream need not be stored in memory.

Examples

Example 1. Only the three strongest values remain.

```
largest_k([4,5,8,2,10], 3)
=> [10, 8, 5]
```

Example 2. Repeated values remain separate occurrences.

```
largest_k([2,2,1], 2)
=> [2, 2]
```

Example 3. A zero-capacity winner pool is empty.

```
largest_k([1,2], 0)
=> []
```

Why this mechanism works

The weakest retained winner is the smallest of the current largest k, so a min-heap exposes precisely the value that should be evicted when capacity is exceeded. The heap itself is not sorted output; order the retained winners only at the end.

State and boundaries

heap contains the largest at-most-k values from the consumed stream; its root is the smallest retained winner.

Protocol

1. Return [] for nonpositive k.
2. Push each incoming value.
3. If size exceeds k, remove the minimum.
4. Sort the retained heap descending at the end.

Visualization

```
k=3, retained min-heap [4,5,8]
incoming 2 -> evict 2
incoming 10 -> evict 4
retained winners 5,8,10
```

Python implementation

```
import heapq

def largest_k(values, k):
    if k <= 0:
        return []
    heap = []
    for value in values:
        heapq.heappush(heap, value)
        if len(heap) > k:
            heapq.heappop(heap)
    return sorted(heap, reverse=True)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
largest_k([4,5,8,2,10], 3)

L4 largest_k: if k <= 0:
values=[4, 5, 8, 2, 10]; k=3

L6 largest_k: heap = []
values=[4, 5, 8, 2, 10]; k=3; heap=[]

L7 largest_k: for value in values:
values=[4, 5, 8, 2, 10]; k=3; heap=[]; value=4

L8 largest_k: heapq.heappush(heap, value)
values=[4, 5, 8, 2, 10]; k=3; heap=[4]; value=4

L7 largest_k: for value in values:
values=[4, 5, 8, 2, 10]; k=3; heap=[4]; value=5

L8 largest_k: heapq.heappush(heap, value)
values=[4, 5, 8, 2, 10]; k=3; heap=[4, 5]; value=5

L7 largest_k: for value in values:
values=[4, 5, 8, 2, 10]; k=3; heap=[4, 5]; value=8

L8 largest_k: heapq.heappush(heap, value)
values=[4, 5, 8, 2, 10]; k=3; heap=[4, 5, 8]; value=8

... 2 intermediate state changes omitted ...

L10 largest_k: heapq.heappop(heap)
values=[4, 5, 8, 2, 10]; k=3; heap=[4, 5, 8]; value=2

L7 largest_k: for value in values:
values=[4, 5, 8, 2, 10]; k=3; heap=[4, 5, 8]; value=10

L8 largest_k: heapq.heappush(heap, value)
```

```
values=[4, 5, 8, 2, 10]; k=3; heap=[4, 5, 8, 10]; value=10

L10 largest_k: heapq.heappop(heap)
values=[4, 5, 8, 2, 10]; k=3; heap=[5, 10, 8]; value=10
```

Result: [10, 8, 5]. Only the three strongest values remain.

```
largest_k([2,2,1], 2)

L4 largest_k: if k <= 0:
values=[2, 2, 1]; k=2

L6 largest_k: heap = []
values=[2, 2, 1]; k=2; heap=[]

L7 largest_k: for value in values:
values=[2, 2, 1]; k=2; heap=[]; value=2

L8 largest_k: heapq.heappush(heap, value)
values=[2, 2, 1]; k=2; heap=[2]; value=2

L8 largest_k: heapq.heappush(heap, value)
values=[2, 2, 1]; k=2; heap=[2, 2]; value=2

L7 largest_k: for value in values:
values=[2, 2, 1]; k=2; heap=[2, 2]; value=1

L8 largest_k: heapq.heappush(heap, value)
values=[2, 2, 1]; k=2; heap=[1, 2, 2]; value=1

L10 largest_k: heapq.heappop(heap)
values=[2, 2, 1]; k=2; heap=[2, 2]; value=1
```

Result: [2, 2]. Repeated values remain separate occurrences.

Why the solution is correct

Adding one value temporarily creates the old winners plus the new candidate. Removing their minimum leaves the largest k of that set. Discarded older losers cannot reenter the largest k without new values improving their rank, so this maintains the correct winner pool.

Time and space

O(n log(k+1)+k log k) time and O(k) working space; returned size is at most min(k,n).

Common mistakes

Using a max-heap exposes the strongest winner rather than the eviction candidate. Deduplicating would change the multiplicity contract.

Reasoning card

State: heap contains the largest at-most-k values from the consumed stream; its root is the smallest retained winner.

Invariant: Heap contains the largest min(k,n_seen) values.

Decision to rehearse: Incoming values compete with the weakest current winner.

Finish: Sort the retained heap descending at the end

Transfer exercise

Return the kth largest value after every insertion once enough values exist, without sorting the heap.

085. Smallest K Values from a Stream

Bridge | Module 09

Problem

Consume an integer stream and return its smallest min(k,n) values in ascending order, retaining duplicates. k<=0 returns []. Use a heap of at most k candidates rather than sorting or storing the full stream.

Examples

Example 1. Large candidates are evicted.

```
smallest_k([4,5,8,2,10], 3)
=> [2, 4, 5]
```

Example 2. Negation works for negative original values too.

```
smallest_k([-3,2,-1], 2)
=> [-3, -1]
```

Example 3. No candidate arrives.

```
smallest_k([], 3)
=> []
```

Why this mechanism works

For the smallest k, the eviction candidate is the largest retained value. Python's standard min-heap can act as a max-oriented heap by storing negated integers. Its smallest stored number then represents the largest original candidate.

State and boundaries

heap stores negatives of the smallest at-most-k consumed values. -heap[0] is the largest retained candidate.

Protocol

1. Negate each incoming value before pushing.
2. Remove the root when capacity is exceeded.
3. Negate retained values back.
4. Sort them ascending for output.

Visualization

```
original winners: 1,3,5
stored negatives: -5,-3,-1
root -5 represents the worst retained winner 5
```

Python implementation

```
import heapq

def smallest_k(values, k):
    if k <= 0:
        return []
    heap = []
    for value in values:
        heapq.heappush(heap, -value)
        if len(heap) > k:
            heapq.heappop(heap)
    return sorted(-value for value in heap)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
smallest_k([4,5,8,2,10], 3)

L4 smallest_k: if k <= 0:
values=[4, 5, 8, 2, 10]; k=3

L6 smallest_k: heap = []
values=[4, 5, 8, 2, 10]; k=3; heap=[]

L7 smallest_k: for value in values:
values=[4, 5, 8, 2, 10]; k=3; heap=[]; value=4

L8 smallest_k: heapq.heappush(heap, -value)
values=[4, 5, 8, 2, 10]; k=3; heap=[-4]; value=4

L7 smallest_k: for value in values:
values=[4, 5, 8, 2, 10]; k=3; heap=[-4]; value=5

L8 smallest_k: heapq.heappush(heap, -value)
values=[4, 5, 8, 2, 10]; k=3; heap=[-5, -4]; value=5

L7 smallest_k: for value in values:
values=[4, 5, 8, 2, 10]; k=3; heap=[-5, -4]; value=8

L8 smallest_k: heapq.heappush(heap, -value)
values=[4, 5, 8, 2, 10]; k=3; heap=[-8, -4, -5]; value=8

... 6 intermediate state changes omitted ...

L11 <genexpr>: return sorted(-value for value in heap)
.0=list_iterator; value=-5

L11 <genexpr>: return sorted(-value for value in heap)
.0=list_iterator; value=-4

L11 <genexpr>: return sorted(-value for value in heap)
.0=list_iterator; value=-2
```

```
L11 <genexpr>: return sorted(-value for value in heap)
.0=list_iterator; value=-2
```

Result: [2, 4, 5]. Large candidates are evicted.

```
smallest_k([-3,2,-1], 2)

L4 smallest_k: if k <= 0:
values=[-3, 2, -1]; k=2

L6 smallest_k: heap = []
values=[-3, 2, -1]; k=2; heap=[]

L7 smallest_k: for value in values:
values=[-3, 2, -1]; k=2; heap=[]; value=-3

L8 smallest_k: heapq.heappush(heap, -value)
values=[-3, 2, -1]; k=2; heap=[3]; value=-3

L7 smallest_k: for value in values:
values=[-3, 2, -1]; k=2; heap=[3]; value=2

L8 smallest_k: heapq.heappush(heap, -value)
values=[-3, 2, -1]; k=2; heap=[-2, 3]; value=2

L7 smallest_k: for value in values:
values=[-3, 2, -1]; k=2; heap=[-2, 3]; value=-1

L8 smallest_k: heapq.heappush(heap, -value)
values=[-3, 2, -1]; k=2; heap=[-2, 3, 1]; value=-1

L10 smallest_k: heapq.heappop(heap)
values=[-3, 2, -1]; k=2; heap=[1, 3]; value=-1

L11 <genexpr>: return sorted(-value for value in heap)
.0=list_iterator; value=1

L11 <genexpr>: return sorted(-value for value in heap)
.0=list_iterator; value=3

L11 <genexpr>: return sorted(-value for value in heap)
.0=list_iterator; value=3
```

Result: [-3, -1]. Negation works for negative original values too.

Why the solution is correct

Removing the minimum negated value removes the maximum original value from the candidate pool. The remaining k values are therefore the smallest among the previous winners and the new arrival. This preserves the desired pool inductively.

Time and space

O(n log(k+1)+k log k) time and O(k) working space.

Common mistakes

Negating only on insertion but not on output returns the wrong values. A plain min-heap would discard precisely the small values being sought.

Reasoning card

State: heap stores negatives of the smallest at-most-k consumed values. -heap[0] is the largest retained candidate.

Invariant: Heap contains the smallest min(k,n_seen) values.

Decision to rehearse: Incoming values compete with the largest current winner.

Finish: Sort them ascending for output

Transfer exercise

Abstract the winner-pool ordering so the same bounded-heap mechanism can retain either extreme.

086. Kth Largest Stream Class

Bridge | Module 09

Problem

Implement KthLargest(k, initial_values) with positive k. add(value) inserts one integer and returns the kth largest value including duplicates, or None until at least k values have been seen. Keep at most k values internally.

Examples

Example 1. Each result is the weakest member of the three-value winner pool.

```
run_operations(KthLargest, [("add",3),("add",5),("add",10),("add",9),("add",4)], 3,
    [4,5,8,2])
=> [4, 5, 5, 8, 8]
```

Example 2. The statistic is undefined before the third observation.

```
run_operations(KthLargest, [("add",1),("add",2)], 3, [])
=> [None, None]
```

Example 3. The first largest is the running maximum.

```
run_operations(KthLargest, [("add",2),("add",1)], 1, [])
=> [2, 2]
```

Why this mechanism works

The root of a min-heap containing the largest k values is the kth largest value itself. Unlike the previous exercise, there is no need to sort winners because only their weakest member is queried. Constructor inputs go through the same insertion transition as later arrivals.

State and boundaries

heap contains the largest at-most-k values observed. Its root is meaningful as kth largest only when its size reaches k.

Protocol

1. Initialize an empty heap and insert initial values.
2. For add, push the new value.
3. Evict the minimum if size exceeds k.
4. Return the root if size is k, else None.

Visualization

```
k=3, winners [4,5,8] -> kth=4
add 3 -> keep [4,5,8] -> 4
add 5 -> keep [5,5,8] -> 5
```

Python implementation

```
import heapq
from structures import run_operations

class KthLargest:
    def __init__(self, k, values):
        if k < 1:
            raise ValueError('k must be positive')
        self.k, self.heap = k, []
        for value in values:
            self.add(value)

    def add(self, value):
        heapq.heappush(self.heap, value)
        if len(self.heap) > self.k:
            heapq.heappop(self.heap)
        return self.heap[0] if len(self.heap) == self.k else None
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(KthLargest, [("add",3),("add",5),("add",10),("add",9),("add",4)], 3,
    [4,5,8,2])

L6 __init__: if k < 1:
self=KthLargest{}; k=3; values=[4, 5, 8, 2]

L8 __init__: self.k, self.heap = k, []
self=KthLargest{'k':3, 'heap':[]}; k=3; values=[4, 5, 8, 2]

L9 __init__: for value in values:
self=KthLargest{'k':3, 'heap':[]}; k=3; values=[4, 5, 8, 2]; value=4

L13 add: heapq.heappush(self.heap, value)
self=KthLargest{'k':3, 'heap':[...]}; value=4

L10 __init__: self.add(value)
self=KthLargest{'k':3, 'heap':[...]}; k=3; values=[4, 5, 8, 2]; value=4

L9 __init__: for value in values:
self=KthLargest{'k':3, 'heap':[...]}; k=3; values=[4, 5, 8, 2]; value=5

L13 add: heapq.heappush(self.heap, value)
self=KthLargest{'k':3, 'heap':[..., ...]}; value=5

L10 __init__: self.add(value)
self=KthLargest{'k':3, 'heap':[..., ...]}; k=3; values=[4, 5, 8, 2]; value=5

... 12 intermediate state changes omitted ...

L13 add: heapq.heappush(self.heap, value)
```

```
self=KthLargest{'k':3, 'heap':[..., ..., ..., ...]}; value=9

L15 add: heapq.heappop(self.heap)
self=KthLargest{'k':3, 'heap':[..., ..., ...]}; value=9

L13 add: heapq.heappush(self.heap, value)
self=KthLargest{'k':3, 'heap':[..., ..., ..., ...]}; value=4

L15 add: heapq.heappop(self.heap)
self=KthLargest{'k':3, 'heap':[..., ..., ...]}; value=4
```

Result: [4, 5, 5, 8, 8]. Each result is the weakest member of the three-value winner pool.

```
run_operations(KthLargest, [("add",1),("add",2)], 3, [])

L6 __init__: if k < 1:
self=KthLargest{}; k=3; values=[]

L8 __init__: self.k, self.heap = k, []
self=KthLargest{'k':3, 'heap':[]}; k=3; values=[]

L13 add: heapq.heappush(self.heap, value)
self=KthLargest{'k':3, 'heap':[...]}; value=1

L13 add: heapq.heappush(self.heap, value)
self=KthLargest{'k':3, 'heap':[..., ...]}; value=2
```

Result: [None, None]. The statistic is undefined before the third observation.

Why the solution is correct

The bounded winner-pool invariant follows from removing the minimum of k+1 candidates. With k retained values, exactly k-1 retained values are at least as large as the minimum, making that minimum the kth order statistic with multiplicity. Fewer values cannot define it.

Time and space

O(log(k+1)) time per add and O(k) space; constructor cost O(m log(k+1)) for m initial values.

Common mistakes

Returning a root before k observations gives the minimum-so-far rather than a defined kth largest. Duplicate values count separately.

Reasoning card

State: heap contains the largest at-most-k values observed. Its root is meaningful as kth largest only when its size reaches k.

Invariant: Heap root is the kth largest among all values seen.

Decision to rehearse: Maintain a min-heap of exactly the best k values.

Finish: Return the root if size is k, else None

Transfer exercise

Allow deletion of old values from a sliding stream. Explain why a plain bounded heap no longer contains enough history.

087. Top K Frequent Values

Bridge | Module 09

Problem

Return up to k distinct integers with greatest frequencies, ordered by descending frequency and then first appearance in nums. k<=0 returns []; if k exceeds the distinct count, return every distinct value. Repeated input values contribute to frequency but appear once in the output.

Examples

Example 1. Equal frequencies are ordered by first appearance.

```
top_k_frequent([2,1,2,1,3], 2)
=> [2, 1]
```

Example 2. The requested count is capped by distinct candidates.

```
top_k_frequent([1,1,2], 5)
=> [1, 2]
```

Example 3. There are no distinct candidates.

```
top_k_frequent([], 2)
=> []
```

Why this mechanism works

Count first, then rank distinct candidates. A bounded min-heap retains the best k by a key where larger is better: (frequency,-first_index). The negative index makes earlier appearances stronger. Sorting only the final heap gives the specified output order.

State and boundaries

counts and first contain complete statistics. heap stores the strongest at-most-k distinct candidates under the two-part ranking key.

Protocol

1. Count frequencies and first indices.
2. Push each distinct candidate with its ranking key.
3. Evict the weakest when size exceeds k.
4. Sort retained keys descending.
5. Return their values.

Visualization

```
nums: 2,1,2,1,3
keys: 2 -> (2,0), 1 -> (2,-1), 3 -> (1,-4)
frequency tie favors value 2
```

Python implementation

```
import heapq

def top_k_frequent(nums, k):
    if k <= 0:
        return []
    counts, first = {}, {}
    for i, value in enumerate(nums):
        counts[value] = counts.get(value, 0) + 1
        first.setdefault(value, i)
    heap = []
    for value, count in counts.items():
        heapq.heappush(heap, (count, -first[value], value))
        if len(heap) > k:
            heapq.heappop(heap)
    return [value for count, position, value in sorted(heap, reverse=True)]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
top_k_frequent([2,1,2,1,3], 2)

L4 top_k_frequent: if k <= 0:
nums=[2, 1, 2, 1, 3]; k=2

L6 top_k_frequent: counts, first = {}, {}
nums=[2, 1, 2, 1, 3]; k=2; counts={}; first={}

L7 top_k_frequent: for i, value in enumerate(nums):
nums=[2, 1, 2, 1, 3]; k=2; counts={}; first={}; i=0; value=2

L8 top_k_frequent: counts[value] = counts.get(value, 0) + 1
nums=[2, 1, 2, 1, 3]; k=2; counts={2:1}; first={}; i=0; value=2

L9 top_k_frequent: first.setdefault(value, i)
nums=[2, 1, 2, 1, 3]; k=2; counts={2:1}; first={2:0}; i=0; value=2

L7 top_k_frequent: for i, value in enumerate(nums):
nums=[2, 1, 2, 1, 3]; k=2; counts={2:1}; first={2:0}; i=1; value=1

L8 top_k_frequent: counts[value] = counts.get(value, 0) + 1
nums=[2, 1, 2, 1, 3]; k=2; counts={2:1, 1:1}; first={2:0}; i=1; value=1

L9 top_k_frequent: first.setdefault(value, i)
nums=[2, 1, 2, 1, 3]; k=2; counts={2:1, 1:1}; first={2:0, 1:1}; i=1; value=1

... 14 intermediate state changes omitted ...

L14 top_k_frequent: heapq.heappop(heap)
nums=[2, 1, 2, 1, 3]; k=2; counts={2:2, 1:2, 3:1}; first={2:0, 1:1, 3:4}; i=4; value=3;
    heap=[(2, -1, 1), (2, 0, 2)]; count=1
```

```
L15 top_k_frequent: return [value for count, position, value in sorted(heap, reverse=True)]
nums=[2, 1, 2, 1, 3]; k=2; counts={2:2, 1:2, 3:1}; first={2:0, 1:1, 3:4}; i=4; value=2;
    heap=[(2, -1, 1), (2, 0, 2)]; count=2; position=0

L15 top_k_frequent: return [value for count, position, value in sorted(heap, reverse=True)]
nums=[2, 1, 2, 1, 3]; k=2; counts={2:2, 1:2, 3:1}; first={2:0, 1:1, 3:4}; i=4; value=1;
    heap=[(2, -1, 1), (2, 0, 2)]; count=2; position=-1

L15 top_k_frequent: return [value for count, position, value in sorted(heap, reverse=True)]
nums=[2, 1, 2, 1, 3]; k=2; counts={2:2, 1:2, 3:1}; first={2:0, 1:1, 3:4}; i=4; value=3;
    heap=[(2, -1, 1), (2, 0, 2)]; count=1
```

Result: [2, 1]. Equal frequencies are ordered by first appearance.

`top_k_frequent([1,1,2], 5)`

```
L4 top_k_frequent: if k <= 0:
nums=[1, 1, 2]; k=5

L6 top_k_frequent: counts, first = {}, {}
nums=[1, 1, 2]; k=5; counts={}; first={}

L7 top_k_frequent: for i, value in enumerate(nums):
nums=[1, 1, 2]; k=5; counts={}; first={}; i=0; value=1

L8 top_k_frequent: counts[value] = counts.get(value, 0) + 1
nums=[1, 1, 2]; k=5; counts={1:1}; first={}; i=0; value=1

L9 top_k_frequent: first.setdefault(value, i)
nums=[1, 1, 2]; k=5; counts={1:1}; first={1:0}; i=0; value=1

L7 top_k_frequent: for i, value in enumerate(nums):
nums=[1, 1, 2]; k=5; counts={1:1}; first={1:0}; i=1; value=1

L8 top_k_frequent: counts[value] = counts.get(value, 0) + 1
nums=[1, 1, 2]; k=5; counts={1:2}; first={1:0}; i=1; value=1

L7 top_k_frequent: for i, value in enumerate(nums):
nums=[1, 1, 2]; k=5; counts={1:2}; first={1:0}; i=2; value=2

... 6 intermediate state changes omitted ...

L12 top_k_frequent: heapq.heappush(heap, (count, -first[value], value))
nums=[1, 1, 2]; k=5; counts={1:2, 2:1}; first={1:0, 2:2}; i=2; value=2; heap=[(1, -2, 2), (2,
    0, 1)]; count=1

L15 top_k_frequent: return [value for count, position, value in sorted(heap, reverse=True)]
nums=[1, 1, 2]; k=5; counts={1:2, 2:1}; first={1:0, 2:2}; i=2; value=1; heap=[(1, -2, 2), (2,
    0, 1)]; count=2; position=0

L15 top_k_frequent: return [value for count, position, value in sorted(heap, reverse=True)]
nums=[1, 1, 2]; k=5; counts={1:2, 2:1}; first={1:0, 2:2}; i=2; value=2; heap=[(1, -2, 2), (2,
    0, 1)]; count=1; position=-2

L15 top_k_frequent: return [value for count, position, value in sorted(heap, reverse=True)]
nums=[1, 1, 2]; k=5; counts={1:2, 2:1}; first={1:0, 2:2}; i=2; value=2; heap=[(1, -2, 2), (2,
    0, 1)]; count=1
```

Result: [1, 2]. The requested count is capped by distinct candidates.

Why the solution is correct

The ranking key exactly represents the stated priority. Bounded-heap eviction preserves the top k keys after each candidate. Sorting those winners by the same key gives descending frequency with the required earliest-appearance tie rule.

Time and space

O(n+u log(k+1)+k log k) expected time and O(u+k) space, with output size capped at u distinct values.

Common mistakes

Counting and selecting individual occurrences duplicates output values. A value-based tie breaker silently changes first-appearance semantics.

Reasoning card

State: counts and first contain complete statistics. heap stores the strongest at-most-k distinct candidates under the two-part ranking key.

Invariant: Heap entries reflect complete frequency counts.

Decision to rehearse: Build frequencies, then retain or extract the strongest frequency candidates.

Finish: Return their values

Transfer exercise

Use frequency buckets instead of a heap, retaining the same deterministic tie order.

088. Merge K Sorted Arrays

Interview | Module 09

Problem

Merge k nondecreasing integer lists into one new sorted list, preserving all occurrences. Some or all lists may be empty. Do not modify the inputs. Equal frontier values are consumed by source index, then position, for deterministic behavior.

Examples

Example 1. Only one frontier per source is needed.

```
merge_k_arrays([[1,4],[2,3],[0,5]])
=> [0, 1, 2, 3, 4, 5]
```

Example 2. Empty sources never enter the heap.

```
merge_k_arrays([[],[2,2],[]])
=> [2, 2]
```

Example 3. There are no sources.

```
merge_k_arrays([])
=> []
```

Why this mechanism works

Only the first unread value from each source can be its next contribution. A heap chooses the smallest among those k frontiers. After consuming one source's frontier, insert only its next value. This generalizes the two-source merge without placing every input value in the heap at once.

State and boundaries

heap contains one (value,source,position) entry per nonexhausted source. output is the sorted merge of all consumed prefixes.

Protocol

1. Seed the first value of each nonempty source.
2. Pop the smallest frontier.
3. Append its value.
4. Push the next value from that same source if present.
5. Continue until the heap is empty.

Visualization

```
frontiers: A:1, B:2, C:0
pop C:0 -> replenish C:5
new frontiers: A:1, B:2, C:5
```

Python implementation

```
import heapq

def merge_k_arrays(arrays):
    heap = []
    for source, values in enumerate(arrays):
        if values:
            heap.append((values[0], source, 0))
    heapq.heapify(heap)
    output = []
    while heap:
        value, source, position = heapq.heappop(heap)
        output.append(value)
        position += 1
        if position < len(arrays[source]):
            heapq.heappush(heap, (arrays[source][position], source, position))
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
merge_k_arrays([[1,4],[2,3],[0,5]])

L4 merge_k_arrays: heap = []
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[]

L5 merge_k_arrays: for source, values in enumerate(arrays):
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[]; source=0; values=[1, 4]

L7 merge_k_arrays: heap.append((values[0], source, 0))
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[(1, 0, 0)]; source=0; values=[1, 4]

L5 merge_k_arrays: for source, values in enumerate(arrays):
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[(1, 0, 0)]; source=1; values=[2, 3]

L7 merge_k_arrays: heap.append((values[0], source, 0))
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[(1, 0, 0), (2, 1, 0)]; source=1; values=[2, 3]

L5 merge_k_arrays: for source, values in enumerate(arrays):
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[(1, 0, 0), (2, 1, 0)]; source=2; values=[0, 5]

L7 merge_k_arrays: heap.append((values[0], source, 0))
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[(1, 0, 0), (2, 1, 0), (0, 2, 0)]; source=2; values=[0,
    5]

L8 merge_k_arrays: heapq.heapify(heap)
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[(0, 2, 0), (2, 1, 0), (1, 0, 0)]; source=2; values=[0,
    5]

... 18 intermediate state changes omitted ...
```

```
L13 merge_k_arrays: position += 1
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[(5, 2, 1)]; source=0; values=[0, 5]; output=[0, 1, 2,
    3, 4]; value=4; position=2

L11 merge_k_arrays: value, source, position = heapq.heappop(heap)
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[]; source=2; values=[0, 5]; output=[0, 1, 2, 3, 4];
    value=5; position=1

L12 merge_k_arrays: output.append(value)
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[]; source=2; values=[0, 5]; output=[0, 1, 2, 3, 4, 5];
    value=5; position=1

L13 merge_k_arrays: position += 1
arrays=[[1, 4], [2, 3], [0, 5]]; heap=[]; source=2; values=[0, 5]; output=[0, 1, 2, 3, 4, 5];
    value=5; position=2
```

Result: [0, 1, 2, 3, 4, 5]. Only one frontier per source is needed.

`merge_k_arrays([[],[2,2],[]])`

```
L4 merge_k_arrays: heap = []
arrays=[[], [2, 2], []]; heap=[]

L5 merge_k_arrays: for source, values in enumerate(arrays):
arrays=[[], [2, 2], []]; heap=[]; source=0; values=[]

L5 merge_k_arrays: for source, values in enumerate(arrays):
arrays=[[], [2, 2], []]; heap=[]; source=1; values=[2, 2]

L7 merge_k_arrays: heap.append((values[0], source, 0))
arrays=[[], [2, 2], []]; heap=[(2, 1, 0)]; source=1; values=[2, 2]

L5 merge_k_arrays: for source, values in enumerate(arrays):
arrays=[[], [2, 2], []]; heap=[(2, 1, 0)]; source=2; values=[]

L9 merge_k_arrays: output = []
arrays=[[], [2, 2], []]; heap=[(2, 1, 0)]; source=2; values=[]; output=[]

L11 merge_k_arrays: value, source, position = heapq.heappop(heap)
arrays=[[], [2, 2], []]; heap=[]; source=1; values=[]; output=[]; value=2; position=0

L12 merge_k_arrays: output.append(value)
arrays=[[], [2, 2], []]; heap=[]; source=1; values=[]; output=[2]; value=2; position=0

... 1 intermediate state changes omitted ...

L15 merge_k_arrays: heapq.heappush(heap, (arrays[source][position], source, position))
arrays=[[], [2, 2], []]; heap=[(2, 1, 1)]; source=1; values=[]; output=[2]; value=2;
    position=1

L11 merge_k_arrays: value, source, position = heapq.heappop(heap)
arrays=[[], [2, 2], []]; heap=[]; source=1; values=[]; output=[2]; value=2; position=1

L12 merge_k_arrays: output.append(value)
arrays=[[], [2, 2], []]; heap=[]; source=1; values=[]; output=[2, 2]; value=2; position=1

L13 merge_k_arrays: position += 1
arrays=[[], [2, 2], []]; heap=[]; source=1; values=[]; output=[2, 2]; value=2; position=2
```

Result: [2, 2]. Empty sources never enter the heap.

Why the solution is correct

The minimum source frontier is no larger than any unread value, so it is the next global output value. Replacing only its source frontier preserves the one-entry-per-source representation. Exhaustion means every occurrence has been consumed exactly once.

Time and space

O(N log(k+1)) time and O(k) frontier space plus O(N) output, where N is total input length.

Common mistakes

Pushing entire source arrays into the heap loses the bounded-frontier advantage. Omitting a source identifier makes it impossible to replenish the correct stream.

Reasoning card

State: heap contains one (value,source,position) entry per nonexhausted source. output is the sorted merge of all consumed prefixes.

Invariant: Heap holds the smallest unconsumed item from every active source.

Decision to rehearse: Pop global minimum, then advance only its source.

Finish: Continue until the heap is empty

Transfer exercise

Replace lists with iterators so the same frontier algorithm can merge files or streams larger than memory.

089. K Closest Points to Origin

Interview | Module 09

Problem

Return up to k two-dimensional integer points closest to the origin, ordered by squared distance and then original input index. Duplicated points count as distinct occurrences. Return coordinate tuples; k<=0 yields []. Do not use square roots or mutate the input.

Examples

Example 1. Squared distance eight is smaller than ten.

```
closest_points([(1,3),(-2,2)], 1)
=> [(-2, 2)]
```

Example 2. Equal distances retain the earliest input indices.

```
closest_points([(1,0),(0,1),(-1,0)], 2)
=> [(1, 0), (0, 1)]
```

Example 3. There are no points to retain.

```
closest_points([], 3)
=> []
```

Why this mechanism works

Distance ordering is unchanged by squaring, avoiding floating-point comparisons. Retain the k strongest candidates in a max-oriented heap represented by negative distance and negative index. The worst retained distance, with the latest tied index, is evicted first.

State and boundaries

heap stores (-squared_distance,-input_index) for retained points. The root is the least desirable retained point.

Protocol

1. Compute x*x+y*y.
2. Push the negated ranking key.
3. Evict one root when capacity is exceeded.
4. Sort retained negative keys descending.
5. Recover coordinate tuples by saved index.

Visualization

```
(1,3): squared distance 10
(-2,2): squared distance 8
retain the latter for k=1
```

Python implementation

```
import heapq

def closest_points(points, k):
    if k <= 0:
        return []
    heap = []
    for i, (x, y) in enumerate(points):
        heapq.heappush(heap, (-(x*x + y*y), -i))
        if len(heap) > k:
            heapq.heappop(heap)
    return [tuple(points[-index]) for distance, index in sorted(heap, reverse=True)]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
closest_points([(1,3),(-2,2)], 1)

L4 closest_points: if k <= 0:
points=[(1, 3), (-2, 2)]; k=1

L6 closest_points: heap = []
points=[(1, 3), (-2, 2)]; k=1; heap=[]

L7 closest_points: for i, (x, y) in enumerate(points):
points=[(1, 3), (-2, 2)]; k=1; heap=[]; i=0; x=1; y=3

L8 closest_points: heapq.heappush(heap, (-(x*x + y*y), -i))
points=[(1, 3), (-2, 2)]; k=1; heap=[(-10, 0)]; i=0; x=1; y=3

L7 closest_points: for i, (x, y) in enumerate(points):
points=[(1, 3), (-2, 2)]; k=1; heap=[(-10, 0)]; i=1; x=-2; y=2

L8 closest_points: heapq.heappush(heap, (-(x*x + y*y), -i))
points=[(1, 3), (-2, 2)]; k=1; heap=[(-10, 0), (-8, -1)]; i=1; x=-2; y=2

L10 closest_points: heapq.heappop(heap)
points=[(1, 3), (-2, 2)]; k=1; heap=[(-8, -1)]; i=1; x=-2; y=2

L11 closest_points: return [tuple(points[-index]) for distance, index in sorted(heap,
    reverse=True)]
points=[(1, 3), (-2, 2)]; k=1; heap=[(-8, -1)]; i=1; x=-2; y=2; distance=-8; index=-1

L11 closest_points: return [tuple(points[-index]) for distance, index in sorted(heap,
    reverse=True)]
points=[(1, 3), (-2, 2)]; k=1; heap=[(-8, -1)]; i=1; x=-2; y=2
```

Result: [(-2, 2)]. Squared distance eight is smaller than ten.

```
closest_points([(1,0),(0,1),(-1,0)], 2)

L4 closest_points: if k <= 0:
points=[(1, 0), (0, 1), (-1, 0)]; k=2

L6 closest_points: heap = []
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[]

L7 closest_points: for i, (x, y) in enumerate(points):
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[]; i=0; x=1; y=0

L8 closest_points: heapq.heappush(heap, (-(x*x + y*y), -i))
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, 0)]; i=0; x=1; y=0

L7 closest_points: for i, (x, y) in enumerate(points):
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, 0)]; i=1; x=0; y=1

L8 closest_points: heapq.heappush(heap, (-(x*x + y*y), -i))
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, -1), (-1, 0)]; i=1; x=0; y=1

L7 closest_points: for i, (x, y) in enumerate(points):
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, -1), (-1, 0)]; i=2; x=-1; y=0

L8 closest_points: heapq.heappush(heap, (-(x*x + y*y), -i))
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, -2), (-1, 0), (-1, -1)]; i=2; x=-1; y=0

L10 closest_points: heapq.heappop(heap)
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, -1), (-1, 0)]; i=2; x=-1; y=0

L11 closest_points: return [tuple(points[-index]) for distance, index in sorted(heap,
    reverse=True)]
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, -1), (-1, 0)]; i=2; x=-1; y=0; distance=-1;
    index=0

L11 closest_points: return [tuple(points[-index]) for distance, index in sorted(heap,
    reverse=True)]
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, -1), (-1, 0)]; i=2; x=-1; y=0; distance=-1;
    index=-1

L11 closest_points: return [tuple(points[-index]) for distance, index in sorted(heap,
    reverse=True)]
points=[(1, 0), (0, 1), (-1, 0)]; k=2; heap=[(-1, -1), (-1, 0)]; i=2; x=-1; y=0
```

Result: [(1, 0), (0, 1)]. Equal distances retain the earliest input indices.

Why the solution is correct

Squared distances preserve nonnegative Euclidean-distance order. Negated keys expose the worst winner at the min-heap root, including the later-index tie rule. Eviction retains exactly the best k keys, and final sorting restores the required closest-first order.

Time and space

O(n log(k+1)+k log k) time and O(k) working space under unit-cost integer arithmetic.

Common mistakes

Square roots add unnecessary rounding risk. Comparing point coordinates to resolve ties does not implement the stated input-index policy.

Reasoning card

State: heap stores (-squared_distance,-input_index) for retained points. The root is the least desirable retained point.

Invariant: Retained points are the best k among those processed.

Decision to rehearse: Use a bounded max-oriented heap or a full min-heap deliberately.

Finish: Recover coordinate tuples by saved index

Transfer exercise

Use quickselect to partition the nearest k points, and distinguish selection cost from the cost of ordered output.

090. Running Median

Interview | Module 09

Problem

Return the median after every integer insertion. For an odd prefix length, use the middle value; for an even length, use the mean of the two middle values. Results are floats. Values are bounded by absolute magnitude 10^9 so these integer and half-integer medians are exactly representable.

Examples

Example 1. The exposed middle roots give each prefix median.

```
running_medians([1,2,3,4])
=> [1.0, 1.5, 2.0, 2.5]
```

Example 2. Duplicates may occupy either side without breaking order.

```
running_medians([5,5,5])
=> [5.0, 5.0, 5.0]
```

Example 3. No prefix requires a median.

```
running_medians([])
=> []
```

Why this mechanism works

Split the observations into a lower half and an upper half. A max-oriented heap exposes the largest lower value; a min-heap exposes the smallest upper value. Maintain the lower half with either equal size or one extra element so the median is always at the heap roots.

State and boundaries

lower stores negated lower-half values. upper stores upper-half values. Every lower value is <= every upper value; len(lower) is len(upper) or one larger.

Protocol

1. Insert into the appropriate half by comparing with the lower maximum.
2. Rebalance by moving one root if necessary.
3. Read the lower root for odd size.
4. Average the two roots for even size.

Visualization

```
lower (max side) <= upper (min side)
[1,2] | [3,4] -> median (2+3)/2
[1,2,3] | [4,5] -> median 3
```

Python implementation

```
import heapq

def running_medians(values):
    lower, upper, output = [], [], []
    for value in values:
        if not lower or value <= -lower[0]:
            heapq.heappush(lower, -value)
        else:
            heapq.heappush(upper, value)
        if len(lower) > len(upper) + 1:
            heapq.heappush(upper, -heapq.heappop(lower))
        elif len(upper) > len(lower):
            heapq.heappush(lower, -heapq.heappop(upper))
        if len(lower) == len(upper):
            output.append((-lower[0] + upper[0]) / 2)
        else:
            output.append(float(-lower[0]))
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
running_medians([1,2,3,4])

L4 running_medians: lower, upper, output = [], [], []
values=[1, 2, 3, 4]; lower=[]; upper=[]; output=[]

L5 running_medians: for value in values:
values=[1, 2, 3, 4]; lower=[]; upper=[]; output=[]; value=1

L7 running_medians: heapq.heappush(lower, -value)
values=[1, 2, 3, 4]; lower=[-1]; upper=[]; output=[]; value=1

L17 running_medians: output.append(float(-lower[0]))
values=[1, 2, 3, 4]; lower=[-1]; upper=[]; output=[1.0]; value=1

L5 running_medians: for value in values:
values=[1, 2, 3, 4]; lower=[-1]; upper=[]; output=[1.0]; value=2

L9 running_medians: heapq.heappush(upper, value)
values=[1, 2, 3, 4]; lower=[-1]; upper=[2]; output=[1.0]; value=2

L15 running_medians: output.append((-lower[0] + upper[0]) / 2)
values=[1, 2, 3, 4]; lower=[-1]; upper=[2]; output=[1.0, 1.5]; value=2

L5 running_medians: for value in values:
values=[1, 2, 3, 4]; lower=[-1]; upper=[2]; output=[1.0, 1.5]; value=3

... 2 intermediate state changes omitted ...
```

```
L17 running_medians: output.append(float(-lower[0]))
values=[1, 2, 3, 4]; lower=[-2, -1]; upper=[3]; output=[1.0, 1.5, 2.0]; value=3

L5 running_medians: for value in values:
values=[1, 2, 3, 4]; lower=[-2, -1]; upper=[3]; output=[1.0, 1.5, 2.0]; value=4

L9 running_medians: heapq.heappush(upper, value)
values=[1, 2, 3, 4]; lower=[-2, -1]; upper=[3, 4]; output=[1.0, 1.5, 2.0]; value=4

L15 running_medians: output.append((-lower[0] + upper[0]) / 2)
values=[1, 2, 3, 4]; lower=[-2, -1]; upper=[3, 4]; output=[1.0, 1.5, 2.0, 2.5]; value=4
```

Result: [1.0, 1.5, 2.0, 2.5]. The exposed middle roots give each prefix median.

```
running_medians([5,5,5])

L4 running_medians: lower, upper, output = [], [], []
values=[5, 5, 5]; lower=[]; upper=[]; output=[]

L5 running_medians: for value in values:
values=[5, 5, 5]; lower=[]; upper=[]; output=[]; value=5

L7 running_medians: heapq.heappush(lower, -value)
values=[5, 5, 5]; lower=[-5]; upper=[]; output=[]; value=5

L17 running_medians: output.append(float(-lower[0]))
values=[5, 5, 5]; lower=[-5]; upper=[]; output=[5.0]; value=5

L7 running_medians: heapq.heappush(lower, -value)
values=[5, 5, 5]; lower=[-5, -5]; upper=[]; output=[5.0]; value=5

L11 running_medians: heapq.heappush(upper, -heapq.heappop(lower))
values=[5, 5, 5]; lower=[-5]; upper=[5]; output=[5.0]; value=5

L15 running_medians: output.append((-lower[0] + upper[0]) / 2)
values=[5, 5, 5]; lower=[-5]; upper=[5]; output=[5.0, 5.0]; value=5

L7 running_medians: heapq.heappush(lower, -value)
values=[5, 5, 5]; lower=[-5, -5]; upper=[5]; output=[5.0, 5.0]; value=5

L17 running_medians: output.append(float(-lower[0]))
values=[5, 5, 5]; lower=[-5, -5]; upper=[5]; output=[5.0, 5.0, 5.0]; value=5
```

Result: [5.0, 5.0, 5.0]. Duplicates may occupy either side without breaking order.

Why the solution is correct

Insertion into the correct side preserves ordering; transferring a boundary root during rebalancing preserves that order too. The size invariant positions the middle element or middle pair exactly at the exposed roots, so each reported median is correct.

Time and space

O(n log n) total time and O(n) storage; each insertion uses O(log n) heap work.

Common mistakes

Balancing sizes without preserving cross-heap order is insufficient. Negated lower values must be converted back when reporting or transferring.

Reasoning card

State: lower stores negated lower-half values. upper stores upper-half values. Every lower value is <= every upper value; len(lower) is len(upper) or one larger.

Invariant: Sizes differ by at most one and every lower value <= every upper value.

Decision to rehearse: Balance lower max-heap and upper min-heap.

Finish: Average the two roots for even size

Transfer exercise

Support a fixed-size sliding median. Identify why expired values require deletion support or lazy removal in both heaps.

Module 10
Intervals and Sweep-Line Events

Represent durations through ordered boundaries and maintain only currently active state.

Translate intervals into ordered events. Endpoint conventions decide whether a departure and arrival at the same time overlap. Sweep-line state describes what is active between events, while interval merging summarizes completed coverage. Dynamic calendars expose an additional issue: sorted lookup may be fast even when insertion still shifts a list.

The ten-step progression

091. Maximum Concurrent Intervals

092. Minimum Meeting Rooms

093. Car Pooling Capacity

094. Total Covered Length

095. Employee Free Time

096. Calendar Booking I

097. Calendar Maximum Overlap

098. Minimum Points to Cover Intervals

099. Skyline Key Points

100. Active Interval Count Queries

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

091. Maximum Concurrent Intervals

Foundation | Module 10

Problem

Given half-open intervals [start,end) with start<=end, return the maximum number active at one time. Ignore zero-length intervals. An interval ending when another begins does not overlap it. Empty input returns zero.

Examples

Example 1. Two intervals overlap but all three never do.

```
maximum_concurrency([(0,30),(5,10),(15,20)])
=> 2
```

Example 2. Shared endpoints are not overlap.

```
maximum_concurrency([(1,3),(3,5)])
=> 1
```

Example 3. An empty interval is active nowhere.

```
maximum_concurrency([(2,2)])
=> 0
```

Why this mechanism works

Replace each duration by a +1 start event and a -1 end event. At equal timestamps, process endings before starts so the sweep respects half-open membership. The active count after an event is a running balance; its maximum is the requested concurrency.

State and boundaries

active counts intervals covering the current sweep position after the processed boundary events. peak is the maximum observed active count.

Protocol

1. Create two events for each nonempty interval.
2. Sort by time then delta, putting -1 before +1.
3. Apply each delta.
4. Update peak.
5. Return peak.

Visualization

```
[1,3) [3,5)
time 3: end -1, then start +1
active never exceeds 1
```

Python implementation

```
def maximum_concurrency(intervals):
    events = []
    for start, end in intervals:
        if start < end:
            events.extend(((start, 1), (end, -1)))
    active = peak = 0
    for time, delta in sorted(events):
        active += delta
        peak = max(peak, active)
    return peak
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
maximum_concurrency([(0,30),(5,10),(15,20)])

L2 maximum_concurrency: events = []
intervals=[(0, 30), (5, 10), (15, 20)]; events=[]

L3 maximum_concurrency: for start, end in intervals:
intervals=[(0, 30), (5, 10), (15, 20)]; events=[]; start=0; end=30

L5 maximum_concurrency: events.extend(((start, 1), (end, -1)))
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1)]; start=0; end=30

L3 maximum_concurrency: for start, end in intervals:
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1)]; start=5; end=10

L5 maximum_concurrency: events.extend(((start, 1), (end, -1)))
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1), (5, 1), (10, -1)]; start=5;
    end=10

L3 maximum_concurrency: for start, end in intervals:
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1), (5, 1), (10, -1)]; start=15;
     end=20

L5 maximum_concurrency: events.extend(((start, 1), (end, -1)))
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1), (5, 1), (10, -1), (15, 1),
    (20, -1)]; start=15; end=20

L6 maximum_concurrency: active = peak = 0
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1), (5, 1), (10, -1), (15, 1),
    (20, -1)]; start=15; end=20; active=0; peak=0

... 10 intermediate state changes omitted ...

L7 maximum_concurrency: for time, delta in sorted(events):
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1), (5, 1), (10, -1), (15, 1),
    (20, -1)]; start=15; end=20; active=2; peak=2; time=20; delta=-1

L8 maximum_concurrency: active += delta
```

```
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1), (5, 1), (10, -1), (15, 1),
    (20, -1)]; start=15; end=20; active=1; peak=2; time=20; delta=-1

L7 maximum_concurrency: for time, delta in sorted(events):
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1), (5, 1), (10, -1), (15, 1),
    (20, -1)]; start=15; end=20; active=1; peak=2; time=30; delta=-1

L8 maximum_concurrency: active += delta
intervals=[(0, 30), (5, 10), (15, 20)]; events=[(0, 1), (30, -1), (5, 1), (10, -1), (15, 1),
    (20, -1)]; start=15; end=20; active=0; peak=2; time=30; delta=-1
```

Result: 2. Two intervals overlap but all three never do.

```
maximum_concurrency([(1,3),(3,5)])

L2 maximum_concurrency: events = []
intervals=[(1, 3), (3, 5)]; events=[]

L3 maximum_concurrency: for start, end in intervals:
intervals=[(1, 3), (3, 5)]; events=[]; start=1; end=3

L5 maximum_concurrency: events.extend(((start, 1), (end, -1)))
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1)]; start=1; end=3

L3 maximum_concurrency: for start, end in intervals:
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1)]; start=3; end=5

L5 maximum_concurrency: events.extend(((start, 1), (end, -1)))
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1), (3, 1), (5, -1)]; start=3; end=5

L6 maximum_concurrency: active = peak = 0
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1), (3, 1), (5, -1)]; start=3; end=5;
    active=0; peak=0

L7 maximum_concurrency: for time, delta in sorted(events):
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1), (3, 1), (5, -1)]; start=3; end=5;
    active=0; peak=0; time=1; delta=1

L8 maximum_concurrency: active += delta
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1), (3, 1), (5, -1)]; start=3; end=5;
    active=1; peak=0; time=1; delta=1

... 3 intermediate state changes omitted ...

L7 maximum_concurrency: for time, delta in sorted(events):
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1), (3, 1), (5, -1)]; start=3; end=5;
    active=0; peak=1; time=3; delta=1

L8 maximum_concurrency: active += delta
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1), (3, 1), (5, -1)]; start=3; end=5;
    active=1; peak=1; time=3; delta=1

L7 maximum_concurrency: for time, delta in sorted(events):
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1), (3, 1), (5, -1)]; start=3; end=5;
    active=1; peak=1; time=5; delta=-1

L8 maximum_concurrency: active += delta
intervals=[(1, 3), (3, 5)]; events=[(1, 1), (3, -1), (3, 1), (5, -1)]; start=3; end=5;
    active=0; peak=1; time=5; delta=-1
```

Result: 1. Shared endpoints are not overlap.

Why the solution is correct

A start adds an interval exactly where it begins to be included; an end removes it exactly where inclusion stops. End-before-start ordering excludes endpoint-only contact. Between boundaries the count is unchanged, so checking event states finds its maximum.

Time and space

O(n log n) time and O(n) event space.

Common mistakes

Start-before-end ordering invents overlap at shared endpoints. Zero-length intervals must not create spurious start and end events.

Reasoning card

State: active counts intervals covering the current sweep position after the processed boundary events. peak is the maximum observed active count.

Invariant: active is the number of intervals active after processed events.

Decision to rehearse: Start is +1; end is -1; process ends before starts on ties.

Finish: Return peak

Transfer exercise

Return the time segments on which peak concurrency occurs, grouping all events at each timestamp before emitting a segment.

092. Minimum Meeting Rooms

Foundation | Module 10

Problem

Return the minimum number of rooms needed for half-open meetings [start,end). Meetings can reuse a room exactly when its previous meeting ends. Ignore zero-length meetings. The input may be unsorted and is not modified.

Examples

Example 1. The long meeting overlaps each shorter meeting separately.

```
minimum_rooms([(0,30),(5,10),(15,20)])
=> 2
```

Example 2. The same room can be reused at time two.

```
minimum_rooms([(1,2),(2,3)])
=> 1
```

Example 3. No meetings require no rooms.

```
minimum_rooms([])
=> 0
```

Why this mechanism works

Sort meetings by start time and keep end times of currently active meetings in a min-heap. Expire every meeting that has ended before allocating the new one. The largest active heap size is a lower bound on rooms and is achievable by reusing freed rooms.

State and boundaries

ends contains end times of meetings active at the next start. rooms is the largest simultaneous heap size observed.

Protocol

1. Sort meetings by start.
2. Skip empty meetings.
3. Pop all end times <= the new start.
4. Push the new end.
5. Update the maximum heap size.

Visualization

```
meeting starts 0,5,15
active ends: [30] -> [10,30] -> [20,30]
maximum active size=2
```

Python implementation

```
import heapq

def minimum_rooms(meetings):
    ends = []
    rooms = 0
    for start, end in sorted(meetings):
        if start == end:
            continue
        while ends and ends[0] <= start:
            heapq.heappop(ends)
        heapq.heappush(ends, end)
        rooms = max(rooms, len(ends))
    return rooms
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
minimum_rooms([(0,30),(5,10),(15,20)])

L4 minimum_rooms: ends = []
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[]

L5 minimum_rooms: rooms = 0
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[]; rooms=0

L6 minimum_rooms: for start, end in sorted(meetings):
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[]; rooms=0; start=0; end=30

L11 minimum_rooms: heapq.heappush(ends, end)
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[30]; rooms=0; start=0; end=30

L12 minimum_rooms: rooms = max(rooms, len(ends))
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[30]; rooms=1; start=0; end=30

L6 minimum_rooms: for start, end in sorted(meetings):
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[30]; rooms=1; start=5; end=10

L11 minimum_rooms: heapq.heappush(ends, end)
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[10, 30]; rooms=1; start=5; end=10

L12 minimum_rooms: rooms = max(rooms, len(ends))
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[10, 30]; rooms=2; start=5; end=10

L6 minimum_rooms: for start, end in sorted(meetings):
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[10, 30]; rooms=2; start=15; end=20

L10 minimum_rooms: heapq.heappop(ends)
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[30]; rooms=2; start=15; end=20
```

```
L11 minimum_rooms: heapq.heappush(ends, end)
meetings=[(0, 30), (5, 10), (15, 20)]; ends=[20, 30]; rooms=2; start=15; end=20
```

Result: 2. The long meeting overlaps each shorter meeting separately.

```
minimum_rooms([(1,2),(2,3)])

L4 minimum_rooms: ends = []
meetings=[(1, 2), (2, 3)]; ends=[]

L5 minimum_rooms: rooms = 0
meetings=[(1, 2), (2, 3)]; ends=[]; rooms=0

L6 minimum_rooms: for start, end in sorted(meetings):
meetings=[(1, 2), (2, 3)]; ends=[]; rooms=0; start=1; end=2

L11 minimum_rooms: heapq.heappush(ends, end)
meetings=[(1, 2), (2, 3)]; ends=[2]; rooms=0; start=1; end=2

L12 minimum_rooms: rooms = max(rooms, len(ends))
meetings=[(1, 2), (2, 3)]; ends=[2]; rooms=1; start=1; end=2

L6 minimum_rooms: for start, end in sorted(meetings):
meetings=[(1, 2), (2, 3)]; ends=[2]; rooms=1; start=2; end=3

L10 minimum_rooms: heapq.heappop(ends)
meetings=[(1, 2), (2, 3)]; ends=[]; rooms=1; start=2; end=3

L11 minimum_rooms: heapq.heappush(ends, end)
meetings=[(1, 2), (2, 3)]; ends=[3]; rooms=1; start=2; end=3
```

Result: 1. The same room can be reused at time two.

Why the solution is correct

Every active meeting needs a different room, giving a concurrency lower bound. A new meeting may use any room released by a popped end; if none is free, one additional room suffices. This construction meets the lower bound at the peak.

Time and space

O(n log n) time and O(n) worst-case heap/sorting space.

Common mistakes

Popping only one expired end leaves stale active meetings when several ended. Returning final heap size misses an earlier peak.

Reasoning card

State: ends contains end times of meetings active at the next start. rooms is the largest simultaneous heap size observed.

Invariant: Heap contains end times of currently active meetings.

Decision to rehearse: Sort starts; expire every meeting that ends before the next start.

Finish: Update the maximum heap size

Transfer exercise

Return actual room assignments by storing room identifiers alongside end times and recycling freed identifiers.

093. Car Pooling Capacity

Foundation | Module 10

Problem

Each trip is (passengers,start,end), occupying seats on [start,end). Given nonnegative capacity, return whether all trips fit. Passenger counts are nonnegative and start<=end; ignore empty trips. Drop-offs at a location happen before pickups there.

Examples

Example 1. The overlap needs five seats.

```
car_pooling([(2,1,5),(3,3,7)], 4)
=> False
```

Example 2. Drop-off seats are available for the new trip.

```
car_pooling([(2,1,5),(3,5,7)], 3)
=> True
```

Example 3. An empty schedule has zero demand.

```
car_pooling([], 0)
=> True
```

Why this mechanism works

Trips contribute weighted boundary deltas rather than unit events. Summing those deltas gives current occupancy. Half-open ordering matters: a seat released at an endpoint is immediately available to a trip starting at that same point.

State and boundaries

active is current passenger occupancy. Events carry passenger-count changes, with negative drop-offs preceding positive pickups at ties.

Protocol

1. Create pickup and drop-off events.
2. Sort by location and delta.
3. Apply each occupancy change.
4. Reject immediately if active exceeds capacity.
5. Accept after all events.

Visualization

```
2 passengers [1,5)
3 passengers [5,7)
at 5: release 2, then board 3
capacity 3 is sufficient
```

Python implementation

```
def car_pooling(trips, capacity):
    events = []
    for passengers, start, end in trips:
        if start < end:
            events.extend(((start, passengers), (end, -passengers)))
    active = 0
    for location, delta in sorted(events):
        active += delta
        if active > capacity:
            return False
    return True
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
car_pooling([(2,1,5),(3,3,7)], 4)

L2 car_pooling: events = []
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[]

L3 car_pooling: for passengers, start, end in trips:
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[]; passengers=2; start=1; end=5

L5 car_pooling: events.extend(((start, passengers), (end, -passengers)))
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[(1, 2), (5, -2)]; passengers=2; start=1;
    end=5

L3 car_pooling: for passengers, start, end in trips:
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[(1, 2), (5, -2)]; passengers=3; start=3;
    end=7

L5 car_pooling: events.extend(((start, passengers), (end, -passengers)))
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[(1, 2), (5, -2), (3, 3), (7, -3)];
    passengers=3; start=3; end=7

L6 car_pooling: active = 0
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[(1, 2), (5, -2), (3, 3), (7, -3)];
    passengers=3; start=3; end=7; active=0

L7 car_pooling: for location, delta in sorted(events):
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[(1, 2), (5, -2), (3, 3), (7, -3)];
    passengers=3; start=3; end=7; active=0; location=1; delta=2

L8 car_pooling: active += delta
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[(1, 2), (5, -2), (3, 3), (7, -3)];
    passengers=3; start=3; end=7; active=2; location=1; delta=2

L7 car_pooling: for location, delta in sorted(events):
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[(1, 2), (5, -2), (3, 3), (7, -3)];
    passengers=3; start=3; end=7; active=2; location=3; delta=3
```

```
L8 car_pooling: active += delta
trips=[(2, 1, 5), (3, 3, 7)]; capacity=4; events=[(1, 2), (5, -2), (3, 3), (7, -3)];
    passengers=3; start=3; end=7; active=5; location=3; delta=3
```

Result: False. The overlap needs five seats.

```
car_pooling([(2,1,5),(3,5,7)], 3)

L2 car_pooling: events = []
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[]

L3 car_pooling: for passengers, start, end in trips:
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[]; passengers=2; start=1; end=5

L5 car_pooling: events.extend(((start, passengers), (end, -passengers)))
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2)]; passengers=2; start=1;
    end=5

L3 car_pooling: for passengers, start, end in trips:
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2)]; passengers=3; start=5;
    end=7

L5 car_pooling: events.extend(((start, passengers), (end, -passengers)))
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2), (5, 3), (7, -3)];
    passengers=3; start=5; end=7

L6 car_pooling: active = 0
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2), (5, 3), (7, -3)];
    passengers=3; start=5; end=7; active=0

L7 car_pooling: for location, delta in sorted(events):
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2), (5, 3), (7, -3)];
    passengers=3; start=5; end=7; active=0; location=1; delta=2

L8 car_pooling: active += delta
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2), (5, 3), (7, -3)];
    passengers=3; start=5; end=7; active=2; location=1; delta=2

... 2 intermediate state changes omitted ...

L7 car_pooling: for location, delta in sorted(events):
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2), (5, 3), (7, -3)];
    passengers=3; start=5; end=7; active=0; location=5; delta=3

L8 car_pooling: active += delta
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2), (5, 3), (7, -3)];
    passengers=3; start=5; end=7; active=3; location=5; delta=3

L7 car_pooling: for location, delta in sorted(events):
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2), (5, 3), (7, -3)];
    passengers=3; start=5; end=7; active=3; location=7; delta=-3

L8 car_pooling: active += delta
trips=[(2, 1, 5), (3, 5, 7)]; capacity=3; events=[(1, 2), (5, -2), (5, 3), (7, -3)];
    passengers=3; start=5; end=7; active=0; location=7; delta=-3
```

Result: True. Drop-off seats are available for the new trip.

Why the solution is correct

Every trip contributes its passenger count exactly throughout its occupied range. The sweep therefore measures actual occupancy at every change point. Capacity is respected everywhere exactly when no event state exceeds it.

Time and space

O(t log t) time and O(t) space for t trips.

Common mistakes

Comparing each trip's passengers separately misses overlapping demand. Pickup-before-drop-off ordering can falsely reject capacity-feasible schedules.

Reasoning card

State: active is current passenger occupancy. Events carry passenger-count changes, with negative drop-offs preceding positive pickups at ties.

Invariant: Running passengers equals active trip load.

Decision to rehearse: Use sweep-line deltas at pickup and drop-off points.

Finish: Accept after all events

Transfer exercise

Return the minimum required capacity and the first location where it is reached.

094. Total Covered Length

Foundation | Module 10

Problem

Return the total length covered by the union of half-open numeric intervals [start,end), with start<=end. Overlaps count only once. Empty intervals add zero length, and empty input returns zero. Do not modify the intervals.

Examples

Example 1. The merged lengths are five and two.

```
covered_length([(1,4),(2,6),(8,10)])
=> 7
```

Example 2. Touching ranges can be coalesced for length measurement.

```
covered_length([(1,2),(2,3)])
=> 2
```

Example 3. The empty union has length zero.

```
covered_length([])
=> 0
```

Why this mechanism works

After sorting by start, overlapping or touching intervals can be represented by one active union segment. Only when a real gap begins is the current segment's length final. Summing finalized disjoint lengths avoids double-counting overlap.

State and boundaries

start,end describe the active merged segment. total contains lengths of earlier finalized segments.

Protocol

1. Sort intervals and handle an empty collection.
2. Extend the active end while the next start<=end.
3. On a gap add end-start and start a new segment.
4. Add the final segment length.

Visualization

```
[1,4) + [2,6) -> [1,6), length 5
[8,10) adds 2
total covered length 7
```

Python implementation

```
def covered_length(intervals):
    ordered = sorted(intervals)
    if not ordered:
        return 0
    start, end = ordered[0]
    total = 0
    for next_start, next_end in ordered[1:]:
        if next_start <= end:
            end = max(end, next_end)
        else:
            total += end - start
            start, end = next_start, next_end
    return total + end - start
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
covered_length([(1,4),(2,6),(8,10)])

L2 covered_length: ordered = sorted(intervals)
intervals=[(1, 4), (2, 6), (8, 10)]; ordered=[(1, 4), (2, 6), (8, 10)]

L5 covered_length: start, end = ordered[0]
intervals=[(1, 4), (2, 6), (8, 10)]; ordered=[(1, 4), (2, 6), (8, 10)]; start=1; end=4

L6 covered_length: total = 0
intervals=[(1, 4), (2, 6), (8, 10)]; ordered=[(1, 4), (2, 6), (8, 10)]; start=1; end=4;
    total=0

L7 covered_length: for next_start, next_end in ordered[1:]:
intervals=[(1, 4), (2, 6), (8, 10)]; ordered=[(1, 4), (2, 6), (8, 10)]; start=1; end=4;
    total=0; next_start=2; next_end=6

L9 covered_length: end = max(end, next_end)
intervals=[(1, 4), (2, 6), (8, 10)]; ordered=[(1, 4), (2, 6), (8, 10)]; start=1; end=6;
    total=0; next_start=2; next_end=6

L7 covered_length: for next_start, next_end in ordered[1:]:
intervals=[(1, 4), (2, 6), (8, 10)]; ordered=[(1, 4), (2, 6), (8, 10)]; start=1; end=6;
    total=0; next_start=8; next_end=10

L11 covered_length: total += end - start
intervals=[(1, 4), (2, 6), (8, 10)]; ordered=[(1, 4), (2, 6), (8, 10)]; start=1; end=6;
    total=5; next_start=8; next_end=10

L12 covered_length: start, end = next_start, next_end
intervals=[(1, 4), (2, 6), (8, 10)]; ordered=[(1, 4), (2, 6), (8, 10)]; start=8; end=10;
    total=5; next_start=8; next_end=10
```

Result: 7. The merged lengths are five and two.

```
covered_length([(1,2),(2,3)])

L2 covered_length: ordered = sorted(intervals)
intervals=[(1, 2), (2, 3)]; ordered=[(1, 2), (2, 3)]

L5 covered_length: start, end = ordered[0]
intervals=[(1, 2), (2, 3)]; ordered=[(1, 2), (2, 3)]; start=1; end=2

L6 covered_length: total = 0
intervals=[(1, 2), (2, 3)]; ordered=[(1, 2), (2, 3)]; start=1; end=2; total=0

L7 covered_length: for next_start, next_end in ordered[1:]:
intervals=[(1, 2), (2, 3)]; ordered=[(1, 2), (2, 3)]; start=1; end=2; total=0; next_start=2;
    next_end=3

L9 covered_length: end = max(end, next_end)
intervals=[(1, 2), (2, 3)]; ordered=[(1, 2), (2, 3)]; start=1; end=3; total=0; next_start=2;
    next_end=3
```

Result: 2. Touching ranges can be coalesced for length measurement.

Why the solution is correct

Extending an overlapping segment preserves its union while counting no length twice. A gap proves later starts cannot extend the old segment, so its length can be committed. Finalized segments are disjoint and together cover exactly the input union.

Time and space

O(n log n) time and O(n) sorting space; O(1) scan state.

Common mistakes

Summing input lengths double-counts overlap. Forgetting the final commit loses the last union segment.

Reasoning card

State: start,end describe the active merged segment. total contains lengths of earlier finalized segments.

Invariant: Active segment is the union since the latest gap.

Decision to rehearse: Merge an active segment and commit only when a true gap appears.

Finish: Add the final segment length

Transfer exercise

Return the covered length within a query interval by intersecting with the already merged union.

095. Employee Free Time

Bridge | Module 10

Problem

Given each employee's sorted nonoverlapping half-open busy intervals, return positive-length intervals when every employee is free, restricted to gaps between the earliest and latest busy times. Do not return unbounded time before or after all work. Empty schedules are allowed; no busy intervals means [].

Examples

Example 1. The interior gap contains no busy interval.

```
employee_free_time([[(1,3),(6,7)],[(2,4)]])
=> [[4, 6]]
```

Example 2. Without busy endpoints there is no bounded interior gap.

```
employee_free_time([[],[]])
=> []
```

Example 3. Touching busy intervals leave no positive-length gap.

```
employee_free_time([[(1,2)],[(2,3)]])
=> []
```

Why this mechanism works

Everyone is free exactly outside the union of all busy intervals. Merge that union and emit its interior gaps. The finite-gap convention is essential: without a specified observation horizon, the free time outside the first and last busy intervals would be unbounded.

State and boundaries

end is the farthest endpoint of the current busy union. gaps contains finalized positive-length gaps between busy segments.

Protocol

1. Flatten and sort all busy intervals.
2. Seed the active end.
3. When a later start exceeds end, emit the gap.
4. Extend end to the larger endpoint.
5. Return only interior gaps.

Visualization

```
employee A: [1,3), [6,7)
employee B: [2,4)
busy union: [1,4), [6,7)
common finite gap: [4,6)
```

Python implementation

```
def employee_free_time(schedules):
    busy = sorted(interval for schedule in schedules for interval in schedule
                  if interval[0] < interval[1])
    if not busy:
        return []
    end = busy[0][1]
    gaps = []
    for start, next_end in busy[1:]:
        if start > end:
            gaps.append([end, start])
        end = max(end, next_end)
    return gaps
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
employee_free_time([[(1,3),(6,7)],[(2,4)]])

L2 <genexpr>: busy = sorted(interval for schedule in schedules for interval in schedule
.0=list_iterator; schedule=[(1, 3), (6, 7)]; interval=(1, 3)

L2 <genexpr>: busy = sorted(interval for schedule in schedules for interval in schedule
.0=list_iterator; schedule=[(1, 3), (6, 7)]; interval=(6, 7)

L2 <genexpr>: busy = sorted(interval for schedule in schedules for interval in schedule
.0=list_iterator; schedule=[(1, 3), (6, 7)]; interval=(6, 7)

L2 <genexpr>: busy = sorted(interval for schedule in schedules for interval in schedule
.0=list_iterator; schedule=[(2, 4)]; interval=(2, 4)

L2 <genexpr>: busy = sorted(interval for schedule in schedules for interval in schedule
.0=list_iterator; schedule=[(2, 4)]; interval=(2, 4)

L2 employee_free_time: busy = sorted(interval for schedule in schedules for interval in
    schedule
schedules=[[(..., ...), (..., ...)], [(..., ...)]]; busy=[(1, 3), (2, 4), (6, 7)]

L6 employee_free_time: end = busy[0][1]
schedules=[[(..., ...), (..., ...)], [(..., ...)]]; busy=[(1, 3), (2, 4), (6, 7)]; end=3

L7 employee_free_time: gaps = []
schedules=[[(..., ...), (..., ...)], [(..., ...)]]; busy=[(1, 3), (2, 4), (6, 7)]; end=3;
    gaps=[]

... 1 intermediate state changes omitted ...

L11 employee_free_time: end = max(end, next_end)
schedules=[[(..., ...), (..., ...)], [(..., ...)]]; busy=[(1, 3), (2, 4), (6, 7)]; end=4;
    gaps=[]; start=2; next_end=4
```

```
L8 employee_free_time: for start, next_end in busy[1:]:
schedules=[[(..., ...), (..., ...)], [(..., ...)]]; busy=[(1, 3), (2, 4), (6, 7)]; end=4;
    gaps=[]; start=6; next_end=7

L10 employee_free_time: gaps.append([end, start])
schedules=[[(..., ...), (..., ...)], [(..., ...)]]; busy=[(1, 3), (2, 4), (6, 7)]; end=4;
    gaps=[[4, 6]]; start=6; next_end=7

L11 employee_free_time: end = max(end, next_end)
schedules=[[(..., ...), (..., ...)], [(..., ...)]]; busy=[(1, 3), (2, 4), (6, 7)]; end=7;
    gaps=[[4, 6]]; start=6; next_end=7
```

Result: [[4, 6]]. The interior gap contains no busy interval.

```
employee_free_time([[],[]])

L2 <genexpr>: busy = sorted(interval for schedule in schedules for interval in schedule
.0=list_iterator; schedule=[]

L2 employee_free_time: busy = sorted(interval for schedule in schedules for interval in
    schedule
schedules=[[], []]; busy=[]
```

Result: []. Without busy endpoints there is no bounded interior gap.

Why the solution is correct

An emitted gap lies after every previous busy interval and before the next start, so no employee is working there. A merged busy span excludes every time when at least one employee works. These complementary interior regions are exactly common finite free time.

Time and space

O(N log N) time and O(N) space for N busy intervals. A k-way merge can exploit the pre-sorted employee lists more directly.

Common mistakes

Intersecting busy schedules finds common work time, not common free time. Inventing finite outer bounds changes the requested observation horizon.

Reasoning card

State: end is the farthest endpoint of the current busy union. gaps contains finalized positive-length gaps between busy segments.

Invariant: Merged busy coverage is final before each reported gap.

Decision to rehearse: Merge all busy intervals, then inspect gaps between merged segments.

Finish: Return only interior gaps

Transfer exercise

Add an explicit workday [open,close) so leading and trailing free time can be returned meaningfully.

096. Calendar Booking I

Bridge | Module 10

Problem

Implement Calendar.book(start,end) for nonempty half-open intervals. Return True and store the booking only when it overlaps no previously accepted booking; otherwise return False without changing the calendar. Shared endpoints are allowed.

Examples

Example 1. Rejection leaves the endpoint-adjacent third booking possible.

```
run_operations(Calendar, [("book",10,20),("book",15,25),("book",20,30)])
=> [True, False, True]
```

Example 2. No bookings produce no decisions.

```
run_operations(Calendar, [])
=> []
```

Example 3. An exact duplicate overlaps the accepted interval.

```
run_operations(Calendar, [("book",1,2),("book",1,2)])
=> [True, False]
```

Why this mechanism works

Keep accepted intervals sorted and disjoint. At the insertion position, only the immediate predecessor and successor can overlap the new interval; all farther intervals lie beyond them. Binary search locates these neighbors, although insertion into a Python list still shifts elements.

State and boundaries

bookings is sorted by start and pairwise nonoverlapping. A rejected operation leaves it unchanged.

Protocol

1. Find the insertion index by start.
2. Reject if the predecessor ends after start.
3. Reject if the successor starts before end.
4. Insert only after both checks pass.
5. Return acceptance.

Visualization

```
accepted [10,20), [30,40)
new [20,30) fits between both
new [15,25) crosses predecessor -> reject
```

Python implementation

```
from bisect import bisect_left
from structures import run_operations

class Calendar:
    def __init__(self):
        self.bookings = []

    def book(self, start, end):
        i = bisect_left(self.bookings, (start, end))
        if i and self.bookings[i - 1][1] > start:
            return False
        if i < len(self.bookings) and self.bookings[i][0] < end:
            return False
        self.bookings.insert(i, (start, end))
        return True
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(Calendar, [("book",10,20),("book",15,25),("book",20,30)])

L6 __init__: self.bookings = []
self=Calendar{'bookings':[]}

L9 book: i = bisect_left(self.bookings, (start, end))
self=Calendar{'bookings':[]}; start=10; end=20; i=0

L14 book: self.bookings.insert(i, (start, end))
self=Calendar{'bookings':[...]}; start=10; end=20; i=0

L9 book: i = bisect_left(self.bookings, (start, end))
self=Calendar{'bookings':[...]}; start=15; end=25; i=1

L9 book: i = bisect_left(self.bookings, (start, end))
self=Calendar{'bookings':[...]}; start=20; end=30; i=1

L14 book: self.bookings.insert(i, (start, end))
self=Calendar{'bookings':[..., ...]}; start=20; end=30; i=1
```

Result: [True, False, True]. Rejection leaves the endpoint-adjacent third booking possible.

```
run_operations(Calendar, [])

L6 __init__: self.bookings = []
self=Calendar{'bookings':[]}
```

Result: []. No bookings produce no decisions.

Why the solution is correct

The predecessor is the latest earlier-starting interval, so if it ends no later than start, every earlier interval is safe. The successor is the earliest later-starting interval, so if end does not exceed its start, all later intervals are safe. Insertion therefore preserves disjointness exactly when accepted.

Time and space

O(log n) neighbor search plus O(n) list insertion per accepted booking; O(n) stored intervals. This is not an O(log n) update structure.

Common mistakes

Using >= for predecessor overlap rejects legitimate endpoint contact. Mutating before validation leaves rejected bookings in state.

Reasoning card

State: bookings is sorted by start and pairwise nonoverlapping. A rejected operation leaves it unchanged.

Invariant: Stored bookings remain pairwise non-overlapping.

Decision to rehearse: Check candidate against stored intervals or ordered neighbours.

Finish: Return acceptance

Transfer exercise

Replace the sorted list with a balanced ordered tree and separate lookup cost from insertion cost in the new bound.

097. Calendar Maximum Overlap

Bridge | Module 10

Problem

Implement OverlapCalendar.book(start,end) for nonempty half-open intervals. Every booking is accepted, and the method returns the maximum overlap among all bookings so far. Duplicate bookings increase overlap. Endpoints use half-open semantics.

Examples

Example 1. Endpoint contact does not create a third simultaneous booking.

```
run_operations(OverlapCalendar, [("book",10,20),("book",15,25),("book",20,30)])
=> [1, 2, 2]
```

Example 2. Duplicate bookings add independent occupancy.

```
run_operations(OverlapCalendar, [("book",1,3),("book",1,3)])
=> [1, 2]
```

Example 3. No booking has been made.

```
run_operations(OverlapCalendar, [])
=> []
```

Why this mechanism works

Accumulate net changes at boundary times. After each insertion, sweep the sorted boundary map to recompute peak occupancy. Grouping equal timestamps into one delta naturally handles simultaneous endings and starts. This straightforward dynamic reference favors inspectable state over an advanced augmented-tree implementation.

State and boundaries

changes[time] is the net start-minus-end delta of all bookings. A sweep integrates those deltas into overlap counts.

Protocol

1. Increment the start delta and decrement the end delta.

2. Visit boundary times in sorted order.

3. Integrate active overlap and track its maximum.

4. Return the new maximum.

Visualization

```
book [10,20): changes +1 at 10, -1 at 20
book [15,25): add +1 at 15, -1 at 25
integrated peak between 15 and 20 is 2
```

Python implementation

```
from structures import run_operations

class OverlapCalendar:
    def __init__(self):
        self.changes = {}

    def book(self, start, end):
        self.changes[start] = self.changes.get(start, 0) + 1
        self.changes[end] = self.changes.get(end, 0) - 1
        active = peak = 0
        for time in sorted(self.changes):
            active += self.changes[time]
            peak = max(peak, active)
        return peak
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(OverlapCalendar, [("book",10,20),("book",15,25),("book",20,30)])

L5 __init__: self.changes = {}
self=OverlapCalendar{'changes':{}}

L8 book: self.changes[start] = self.changes.get(start, 0) + 1
self=OverlapCalendar{'changes':{...:...}}; start=10; end=20

L9 book: self.changes[end] = self.changes.get(end, 0) - 1
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=10; end=20

L10 book: active = peak = 0
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=10; end=20; active=0; peak=0

L11 book: for time in sorted(self.changes):
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=10; end=20; active=0; peak=0;
    time=10

L12 book: active += self.changes[time]
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=10; end=20; active=1; peak=0;
    time=10

L13 book: peak = max(peak, active)
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=10; end=20; active=1; peak=1;
    time=10

L11 book: for time in sorted(self.changes):
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=10; end=20; active=1; peak=1;
    time=20

... 24 intermediate state changes omitted ...
```

```
L11 book: for time in sorted(self.changes):
self=OverlapCalendar{'changes':{...:..., ...:..., ...:..., ...:..., ...:...}}; start=20;
    end=30; active=2; peak=2; time=25

L12 book: active += self.changes[time]
self=OverlapCalendar{'changes':{...:..., ...:..., ...:..., ...:..., ...:...}}; start=20;
    end=30; active=1; peak=2; time=25

L11 book: for time in sorted(self.changes):
self=OverlapCalendar{'changes':{...:..., ...:..., ...:..., ...:..., ...:...}}; start=20;
    end=30; active=1; peak=2; time=30

L12 book: active += self.changes[time]
self=OverlapCalendar{'changes':{...:..., ...:..., ...:..., ...:..., ...:...}}; start=20;
    end=30; active=0; peak=2; time=30
```

Result: [1, 2, 2]. Endpoint contact does not create a third simultaneous booking.

```
run_operations(OverlapCalendar, [("book",1,3),("book",1,3)])

L5 __init__: self.changes = {}
self=OverlapCalendar{'changes':{}}

L8 book: self.changes[start] = self.changes.get(start, 0) + 1
self=OverlapCalendar{'changes':{...:...}}; start=1; end=3

L9 book: self.changes[end] = self.changes.get(end, 0) - 1
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3

L10 book: active = peak = 0
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=0; peak=0

L11 book: for time in sorted(self.changes):
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=0; peak=0; time=1

L12 book: active += self.changes[time]
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=1; peak=0; time=1

L13 book: peak = max(peak, active)
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=1; peak=1; time=1

L11 book: for time in sorted(self.changes):
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=1; peak=1; time=3

... 4 intermediate state changes omitted ...

L12 book: active += self.changes[time]
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=2; peak=0; time=1

L13 book: peak = max(peak, active)
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=2; peak=2; time=1

L11 book: for time in sorted(self.changes):
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=2; peak=2; time=3

L12 book: active += self.changes[time]
self=OverlapCalendar{'changes':{...:..., ...:...}}; start=1; end=3; active=0; peak=2; time=3
```

Result: [1, 2]. Duplicate bookings add independent occupancy.

Why the solution is correct

Each stored interval contributes one exactly on its half-open duration. Summed deltas therefore reconstruct the full occupancy function after every insertion. Its maximum over boundary segments is the correct calendar-wide peak.

Time and space

O(b log b) time for the bth booking and O(b) state; O(B^2 log B) total upper bound over B bookings. An augmented ordered tree can improve updates.

Common mistakes

Reporting only overlap with the newest interval is not the requested global maximum. Claiming logarithmic updates for a full resweep is incorrect.

Reasoning card

State: changes[time] is the net start-minus-end delta of all bookings. A sweep integrates those deltas into overlap counts.

Invariant: Sweep balance equals active bookings at each boundary.

Decision to rehearse: Accumulate boundary deltas and recompute/maintain the sweep count.

Finish: Return the new maximum

Transfer exercise

Design a tree node summary containing subtree delta sum and maximum prefix sum; derive how two child summaries combine.

098. Minimum Points to Cover Intervals

Interview | Module 10

Problem

Given closed intervals [start,end] with start<=end, return a minimum-size increasing list of points such that every interval contains at least one point. Endpoint hits count. Empty input returns []. The number of returned points is the minimum number of balloon arrows in the equivalent interpretation.

Examples

Example 1. Two endpoint choices cover all four intervals.

```
cover_interval_points([(10,16),(2,8),(1,6),(7,12)])
=> [6, 12]
```

Example 2. The shared closed endpoint hits both intervals.

```
cover_interval_points([(1,2),(2,3)])
=> [2]
```

Example 3. No interval requires a point.

```
cover_interval_points([])
=> []
```

Why this mechanism works

Choose the earliest ending interval and place a point at its end. That point is as far right as possible while still covering the interval, giving it the best chance to cover later intervals. Skip intervals already containing the chosen point and repeat only when a new interval starts beyond it.

State and boundaries

point is the latest selected covering point. All processed intervals are covered; points contains the committed choices.

Protocol

1. Sort intervals by end.

2. If no point exists or the next start exceeds point, choose that interval's end.

3. Otherwise reuse the current point.

4. Return chosen points.

Visualization

```
[1,6], [2,8] -> choose 6
[7,12], [10,16] -> choose 12
points [6,12] cover all four
```

Python implementation

```
def cover_interval_points(intervals):
    points = []
    point = None
    for start, end in sorted(intervals, key=lambda pair: pair[1]):
        if point is None or start > point:
            point = end
            points.append(point)
    return points
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
cover_interval_points([(10,16),(2,8),(1,6),(7,12)])

L2 cover_interval_points: points = []
intervals=[(10, 16), (2, 8), (1, 6), (7, 12)]; points=[]

L3 cover_interval_points: point = None
intervals=[(10, 16), (2, 8), (1, 6), (7, 12)]; points=[]; point=None

L4 <lambda>: for start, end in sorted(intervals, key=lambda pair: pair[1]):
pair=(10, 16)

L4 <lambda>: for start, end in sorted(intervals, key=lambda pair: pair[1]):
pair=(2, 8)

L4 <lambda>: for start, end in sorted(intervals, key=lambda pair: pair[1]):
pair=(1, 6)

L4 <lambda>: for start, end in sorted(intervals, key=lambda pair: pair[1]):
pair=(7, 12)

L4 cover_interval_points: for start, end in sorted(intervals, key=lambda pair: pair[1]):
intervals=[(10, 16), (2, 8), (1, 6), (7, 12)]; points=[]; point=None; start=1; end=6

L6 cover_interval_points: point = end
intervals=[(10, 16), (2, 8), (1, 6), (7, 12)]; points=[]; point=6; start=1; end=6

... 2 intermediate state changes omitted ...

L4 cover_interval_points: for start, end in sorted(intervals, key=lambda pair: pair[1]):
intervals=[(10, 16), (2, 8), (1, 6), (7, 12)]; points=[6]; point=6; start=7; end=12

L6 cover_interval_points: point = end
intervals=[(10, 16), (2, 8), (1, 6), (7, 12)]; points=[6]; point=12; start=7; end=12

L7 cover_interval_points: points.append(point)
intervals=[(10, 16), (2, 8), (1, 6), (7, 12)]; points=[6, 12]; point=12; start=7; end=12
```

```
L4 cover_interval_points: for start, end in sorted(intervals, key=lambda pair: pair[1]):
intervals=[(10, 16), (2, 8), (1, 6), (7, 12)]; points=[6, 12]; point=12; start=10; end=16
```

Result: [6, 12]. Two endpoint choices cover all four intervals.

```
cover_interval_points([(1,2),(2,3)])

L2 cover_interval_points: points = []
intervals=[(1, 2), (2, 3)]; points=[]

L3 cover_interval_points: point = None
intervals=[(1, 2), (2, 3)]; points=[]; point=None

L4 <lambda>: for start, end in sorted(intervals, key=lambda pair: pair[1]):
pair=(1, 2)

L4 <lambda>: for start, end in sorted(intervals, key=lambda pair: pair[1]):
pair=(2, 3)

L4 cover_interval_points: for start, end in sorted(intervals, key=lambda pair: pair[1]):
intervals=[(1, 2), (2, 3)]; points=[]; point=None; start=1; end=2

L6 cover_interval_points: point = end
intervals=[(1, 2), (2, 3)]; points=[]; point=2; start=1; end=2

L7 cover_interval_points: points.append(point)
intervals=[(1, 2), (2, 3)]; points=[2]; point=2; start=1; end=2

L4 cover_interval_points: for start, end in sorted(intervals, key=lambda pair: pair[1]):
intervals=[(1, 2), (2, 3)]; points=[2]; point=2; start=2; end=3
```

Result: [2]. The shared closed endpoint hits both intervals.

Why the solution is correct

Any solution must hit the earliest-ending uncovered interval. Moving its chosen hit rightward to that interval's end cannot lose coverage of any remaining interval that contained the old hit, because those intervals end no earlier. Thus the greedy point is compatible with an optimum, and the argument repeats.

Time and space

O(n log n) time and O(n) sorting/output space.

Common mistakes

This kata uses closed intervals, so start==point is covered. Choosing interval starts lacks the earliest-finish exchange argument.

Reasoning card

State: point is the latest selected covering point. All processed intervals are covered; points contains the committed choices.

Invariant: Chosen points cover all processed intervals with maximal future flexibility.

Decision to rehearse: Sort by earliest end and place a point there when needed.

Finish: Return chosen points

Transfer exercise

Why does the same endpoint choice fail for half-open intervals over real coordinates? Specify a new point-domain convention before adapting it.

099. Skyline Key Points

Interview | Module 10

Problem

Buildings are (left,right,height) with left<right and positive height, covering [left,right). Return skyline key points [x,height] in increasing x order, emitting a point only when visible height changes. Include the final return to zero; empty input returns [].

Examples

Example 1. The shorter building ending at four does not change visibility.

```
skyline([(1,4,3),(2,5,4)])
=> [[1, 3], [2, 4], [5, 0]]
```

Example 2. Equal-height touching buildings make one continuous outline.

```
skyline([(1,2,3),(2,3,3)])
=> [[1, 3], [3, 0]]
```

Example 3. Ground alone has no skyline change points.

```
skyline([])
=> []
```

Why this mechanism works

Only the tallest active building determines the outline. Sweep every start and end coordinate, pushing starting buildings into a max-oriented heap. Expired buildings may remain below the top harmlessly; remove them when exposed. Process all starts and expirations at a coordinate before emitting one height change.

State and boundaries

heap stores (-height,end) candidates, possibly including hidden expired entries. After cleanup its top is the tallest active building; a zero-height infinite sentinel represents ground.

Protocol

1. Group starts and register every endpoint.
2. Sweep sorted coordinates.
3. Push all buildings starting here.
4. Pop expired heap tops with end<=x.
5. Read the visible height.

6. Emit only if it differs from the previous height.

Visualization

```
buildings [1,4) height 3; [2,5) height 4
visible: 0 -> 3 at 1 -> 4 at 2 -> 0 at 5
end at 4 is hidden, so no key point
```

Python implementation

```
import heapq

def skyline(buildings):
    starts = {}
    for left, right, height in buildings:
        starts.setdefault(left, []).append((-height, right))
        starts.setdefault(right, [])
    heap = [(0, float('inf'))]
    output = []
    previous = 0
    for x in sorted(starts):
        for entry in starts[x]:
            heapq.heappush(heap, entry)
        while heap[0][1] <= x:
            heapq.heappop(heap)
        height = -heap[0][0]
        if height != previous:
            output.append([x, height])
            previous = height
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
skyline([(1,4,3),(2,5,4)])

L4 skyline: starts = {}
buildings=[(1, 4, 3), (2, 5, 4)]; starts={}

L5 skyline: for left, right, height in buildings:
buildings=[(1, 4, 3), (2, 5, 4)]; starts={}; left=1; right=4; height=3

L6 skyline: starts.setdefault(left, []).append((-height, right))
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)]}; left=1; right=4; height=3

L7 skyline: starts.setdefault(right, [])
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[]}; left=1; right=4; height=3

L5 skyline: for left, right, height in buildings:
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[]}; left=2; right=5; height=4

L6 skyline: starts.setdefault(left, []).append((-height, right))
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[], 2:[(..., ...)]}; left=2;
    right=5; height=4

L7 skyline: starts.setdefault(right, [])
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[], 2:[(..., ...)], 5:[]}; left=2;
```

```
     right=5; height=4

L8 skyline: heap = [(0, float('inf'))]
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[], 2:[(..., ...)], 5:[]}; left=2;
     right=5; height=4; heap=[(0, inf)]

... 17 intermediate state changes omitted ...

L15 skyline: heapq.heappop(heap)
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[], 2:[(..., ...)], 5:[]}; left=2;
     right=5; height=4; heap=[(0, inf)]; output=[[1, 3], [2, 4]]; previous=4; x=5; entry=(-4,
    5)

L16 skyline: height = -heap[0][0]
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[], 2:[(..., ...)], 5:[]}; left=2;
     right=5; height=0; heap=[(0, inf)]; output=[[1, 3], [2, 4]]; previous=4; x=5; entry=(-4,
    5)

L18 skyline: output.append([x, height])
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[], 2:[(..., ...)], 5:[]}; left=2;
     right=5; height=0; heap=[(0, inf)]; output=[[1, 3], [2, 4], [5, 0]]; previous=4; x=5;
    entry=(-4, 5)

L19 skyline: previous = height
buildings=[(1, 4, 3), (2, 5, 4)]; starts={1:[(..., ...)], 4:[], 2:[(..., ...)], 5:[]}; left=2;
     right=5; height=0; heap=[(0, inf)]; output=[[1, 3], [2, 4], [5, 0]]; previous=0; x=5;
    entry=(-4, 5)
```

Result: [[1, 3], [2, 4], [5, 0]]. The shorter building ending at four does not change visibility.

`skyline([(1,2,3),(2,3,3)])`

```
L4 skyline: starts = {}
buildings=[(1, 2, 3), (2, 3, 3)]; starts={}

L5 skyline: for left, right, height in buildings:
buildings=[(1, 2, 3), (2, 3, 3)]; starts={}; left=1; right=2; height=3

L6 skyline: starts.setdefault(left, []).append((-height, right))
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)]}; left=1; right=2; height=3

L7 skyline: starts.setdefault(right, [])
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[]}; left=1; right=2; height=3

L5 skyline: for left, right, height in buildings:
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[]}; left=2; right=3; height=3

L6 skyline: starts.setdefault(left, []).append((-height, right))
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[(..., ...)]}; left=2; right=3;
    height=3

L7 skyline: starts.setdefault(right, [])
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[(..., ...)], 3:[]}; left=2;
    right=3; height=3

L8 skyline: heap = [(0, float('inf'))]
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[(..., ...)], 3:[]}; left=2;
    right=3; height=3; heap=[(0, inf)]

... 12 intermediate state changes omitted ...

L15 skyline: heapq.heappop(heap)
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[(..., ...)], 3:[]}; left=2;
    right=3; height=3; heap=[(0, inf)]; output=[[1, 3]]; previous=3; x=3; entry=(-3, 3)
```

```
L16 skyline: height = -heap[0][0]
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[(..., ...)], 3:[]}; left=2;
    right=3; height=0; heap=[(0, inf)]; output=[[1, 3]]; previous=3; x=3; entry=(-3, 3)

L18 skyline: output.append([x, height])
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[(..., ...)], 3:[]}; left=2;
    right=3; height=0; heap=[(0, inf)]; output=[[1, 3], [3, 0]]; previous=3; x=3; entry=(-3,
    3)

L19 skyline: previous = height
buildings=[(1, 2, 3), (2, 3, 3)]; starts={1:[(..., ...)], 2:[(..., ...)], 3:[]}; left=2;
    right=3; height=0; heap=[(0, inf)]; output=[[1, 3], [3, 0]]; previous=0; x=3; entry=(-3,
    3)
```

Result: [[1, 3], [3, 0]]. Equal-height touching buildings make one continuous outline.

Why the solution is correct

Any hidden expired entry cannot affect the maximum until it reaches the top, at which point cleanup removes it. After cleanup the top is therefore the tallest active building. Height is constant between event coordinates, so emitting only changes reconstructs the complete outline without duplicate-coordinate artifacts.

Time and space

O(n log n) time and O(n) space; each building is pushed and popped at most once.

Common mistakes

Emitting after each individual event creates false vertical blips at shared coordinates. Removing only buildings whose end equals x leaves earlier expired heap entries stale.

Reasoning card

State: heap stores (-height,end) candidates, possibly including hidden expired entries. After cleanup its top is the tallest active building; a zero-height infinite sentinel represents ground.

Invariant: Current maximum active height defines the skyline segment.

Decision to rehearse: Process start/end events with an active-height multiset/heap.

Finish: Emit only if it differs from the previous height

Transfer exercise

Integrate consecutive skyline heights times x-distance to compute the area under the visible outline.

100. Active Interval Count Queries

Interview | Module 10

Problem

Given half-open intervals [start,end) with start<=end and query times, return the active interval count at each time in original query order. Empty intervals contribute zero. Queries may be unsorted and repeated.

Examples

Example 1. The count falls at each excluded right endpoint.

```
active_counts([(1,3),(2,5)], [0,1,2,3,5])
=> [0, 1, 2, 1, 0]
```

Example 2. An empty interval contributes equal start and end counts.

```
active_counts([(2,2)], [2])
=> [0]
```

Example 3. No intervals are active at any query.

```
active_counts([], [1,9])
=> [0, 0]
```

Why this mechanism works

At time t, active intervals equal those that have started by t minus those that have ended by t. Sorting starts and ends separately makes each quantity an upper-bound query. Both use <=t because starts are included and ends are excluded.

State and boundaries

starts and ends are sorted endpoint multisets. bisect_right counts endpoints <= the query.

Protocol

1. Sort all starts and all ends.
2. For each t count starts<=t.
3. Subtract ends<=t.
4. Return counts in query order.

Visualization

```
interval [1,3)
t=1: started 1 - ended 0 = 1
t=3: started 1 - ended 1 = 0
```

Python implementation

```
from bisect import bisect_right

def active_counts(intervals, queries):
    starts = sorted(start for start, end in intervals)
    ends = sorted(end for start, end in intervals)
    return [bisect_right(starts, t) - bisect_right(ends, t) for t in queries]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
active_counts([(1,3),(2,5)], [0,1,2,3,5])

L4 <genexpr>: starts = sorted(start for start, end in intervals)
.0=list_iterator; start=1; end=3

L4 <genexpr>: starts = sorted(start for start, end in intervals)
.0=list_iterator; start=2; end=5

L4 <genexpr>: starts = sorted(start for start, end in intervals)
.0=list_iterator; start=2; end=5

L4 active_counts: starts = sorted(start for start, end in intervals)
intervals=[(1, 3), (2, 5)]; queries=[0, 1, 2, 3, 5]; starts=[1, 2]

L5 <genexpr>: ends = sorted(end for start, end in intervals)
.0=list_iterator; start=1; end=3

L5 <genexpr>: ends = sorted(end for start, end in intervals)
.0=list_iterator; start=2; end=5

L5 <genexpr>: ends = sorted(end for start, end in intervals)
.0=list_iterator; start=2; end=5

L5 active_counts: ends = sorted(end for start, end in intervals)
intervals=[(1, 3), (2, 5)]; queries=[0, 1, 2, 3, 5]; starts=[1, 2]; ends=[3, 5]

... 2 intermediate state changes omitted ...

L6 active_counts: return [bisect_right(starts, t) - bisect_right(ends, t) for t in queries]
intervals=[(1, 3), (2, 5)]; queries=[0, 1, 2, 3, 5]; starts=[1, 2]; ends=[3, 5]; t=2

L6 active_counts: return [bisect_right(starts, t) - bisect_right(ends, t) for t in queries]
intervals=[(1, 3), (2, 5)]; queries=[0, 1, 2, 3, 5]; starts=[1, 2]; ends=[3, 5]; t=3

L6 active_counts: return [bisect_right(starts, t) - bisect_right(ends, t) for t in queries]
intervals=[(1, 3), (2, 5)]; queries=[0, 1, 2, 3, 5]; starts=[1, 2]; ends=[3, 5]; t=5

L6 active_counts: return [bisect_right(starts, t) - bisect_right(ends, t) for t in queries]
intervals=[(1, 3), (2, 5)]; queries=[0, 1, 2, 3, 5]; starts=[1, 2]; ends=[3, 5]
```

Result: [0, 1, 2, 1, 0]. The count falls at each excluded right endpoint.

```
active_counts([(2,2)], [2])

L4 <genexpr>: starts = sorted(start for start, end in intervals)
.0=list_iterator; start=2; end=2

L4 <genexpr>: starts = sorted(start for start, end in intervals)
.0=list_iterator; start=2; end=2

L4 active_counts: starts = sorted(start for start, end in intervals)
intervals=[(2, 2)]; queries=[2]; starts=[2]

L5 <genexpr>: ends = sorted(end for start, end in intervals)
.0=list_iterator; start=2; end=2

L5 <genexpr>: ends = sorted(end for start, end in intervals)
.0=list_iterator; start=2; end=2

L5 active_counts: ends = sorted(end for start, end in intervals)
intervals=[(2, 2)]; queries=[2]; starts=[2]; ends=[2]

L6 active_counts: return [bisect_right(starts, t) - bisect_right(ends, t) for t in queries]
intervals=[(2, 2)]; queries=[2]; starts=[2]; ends=[2]; t=2

L6 active_counts: return [bisect_right(starts, t) - bisect_right(ends, t) for t in queries]
intervals=[(2, 2)]; queries=[2]; starts=[2]; ends=[2]
```

Result: [0]. An empty interval contributes equal start and end counts.

Why the solution is correct

Every interval started by t contributes one to the first count. If it has also ended by t, it contributes one to the second and cancels. The uncanceled intervals satisfy start<=t<end, exactly the half-open membership condition.

Time and space

O(n log n+q log(n+1)) time and O(n+q) storage including answers.

Common mistakes

Using a lower bound for ends counts an interval at its excluded endpoint. Pairing endpoints after sorting is unnecessary because only aggregate counts are queried.

Reasoning card

State: starts and ends are sorted endpoint multisets. bisect_right counts endpoints <= the query.

Invariant: Counts include starts not after query and exclude ends at/before query for half-open rules.

Decision to rehearse: Sort starts, ends, and queries or use prefix events.

Finish: Return counts in query order

Transfer exercise

Return weighted active demand by storing prefix sums over sorted weighted endpoint events.

Module 11
Linked Lists and Pointer Mutation

Treat links as the structure; preserve unread nodes before changing pointers.

Linked structures make reference ownership visible. Before replacing a next pointer, preserve the node you still need to reach. Dummy nodes remove special cases at the head; fast/slow pointers compare traversal progress. Diagrams should name objects and arrows, because equal values do not imply equal node identity.

The ten-step progression

101. Build and Serialize a Singly Linked List

102. Reverse a Linked List

103. Merge Two Sorted Linked Lists

104. Find the Middle Node

105. Detect a Cycle

106. Find Cycle Entry

107. Remove Nth Node from End

108. Reverse a Sublist

109. Reorder List

110. Copy List with Random Pointer

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

101. Build and Serialize a Singly Linked List

Foundation | Module 11

Problem

Implement conversion from an integer list to a singly linked chain and back. Each node has val and next. Build fresh nodes in input order; an empty list becomes None. Serialization accepts a finite acyclic chain and returns its values.

Examples

Example 1. Following links reproduces construction order.

```
to_values(to_linked([1,2,3]))
=> [1, 2, 3]
```

Example 2. No real node is attached to the dummy.

```
to_linked([])
=> None
```

Example 3. Equal values still occupy distinct nodes.

```
to_values(to_linked([7,7]))
=> [7, 7]
```

Why this mechanism works

A dummy node removes the special case for the first real node. The tail always names the last built node, so each new allocation can be attached in constant time. Serialization follows links rather than assuming that nodes occupy adjacent memory locations.

State and boundaries

dummy.next is the constructed head. tail is the last constructed node; during serialization current is the next unread node.

Protocol

1. Create a dummy and tail.
2. Allocate one node per value and link it after tail.
3. Advance tail.
4. Return dummy.next.

5. Serialize by reading val and following next until None.

Visualization

```
dummy -> 1 -> 2 -> 3 -> None
         head       tail
return dummy.next, not dummy
```

Python implementation

```
from structures import ListNode

def to_linked(values):
    dummy = ListNode()
    tail = dummy
    for value in values:
        tail.next = ListNode(value)
        tail = tail.next
    return dummy.next

def to_values(head):
    output = []
    current = head
    while current is not None:
        output.append(current.val)
        current = current.next
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
to_values(to_linked([1,2,3]))

L4 to_linked: dummy = ListNode()
values=[1, 2, 3]; dummy=N1(val=0,next=None,random=None)

L5 to_linked: tail = dummy
values=[1, 2, 3]; dummy=N1(val=0,next=None,random=None); tail=N1(val=0,next=None,random=None)

L6 to_linked: for value in values:
values=[1, 2, 3]; dummy=N1(val=0,next=None,random=None); tail=N1(val=0,next=None,random=None);
     value=1

L7 to_linked: tail.next = ListNode(value)
values=[1, 2, 3]; dummy=N1(val=0,next=N2,random=None); tail=N1(val=0,next=N2,random=None);
    value=1

L8 to_linked: tail = tail.next
values=[1, 2, 3]; dummy=N1(val=0,next=N2,random=None); tail=N2(val=1,next=None,random=None);
    value=1

L6 to_linked: for value in values:
values=[1, 2, 3]; dummy=N1(val=0,next=N2,random=None); tail=N2(val=1,next=None,random=None);
    value=2

L7 to_linked: tail.next = ListNode(value)
values=[1, 2, 3]; dummy=N1(val=0,next=N2,random=None); tail=N2(val=1,next=N3,random=None);
```

```
    value=2

L8 to_linked: tail = tail.next
values=[1, 2, 3]; dummy=N1(val=0,next=N2,random=None); tail=N3(val=2,next=None,random=None);
    value=2

... 7 intermediate state changes omitted ...

L15 to_values: output.append(current.val)
head=N2(val=1,next=N3,random=None); output=[1, 2]; current=N3(val=2,next=N4,random=None)

L16 to_values: current = current.next
head=N2(val=1,next=N3,random=None); output=[1, 2]; current=N4(val=3,next=None,random=None)

L15 to_values: output.append(current.val)
head=N2(val=1,next=N3,random=None); output=[1, 2, 3]; current=N4(val=3,next=None,random=None)

L16 to_values: current = current.next
head=N2(val=1,next=N3,random=None); output=[1, 2, 3]; current=None
```

Result: [1, 2, 3]. Following links reproduces construction order.

```
to_linked([])

L4 to_linked: dummy = ListNode()
values=[]; dummy=N1(val=0,next=None,random=None)

L5 to_linked: tail = dummy
values=[]; dummy=N1(val=0,next=None,random=None); tail=N1(val=0,next=None,random=None)
```

Result: None. No real node is attached to the dummy.

Why the solution is correct

Each allocation appends exactly the next input value and leaves prior links unchanged. The chain therefore preserves order and contains fresh nodes. Following next visits precisely that construction order until the final None link.

Time and space

O(n) time for building or serialization. Building returns O(n) nodes; serialization returns O(n) values, with O(1) other traversal state.

Common mistakes

Returning the dummy includes an artificial value. Repeatedly searching for the tail makes construction quadratic.

Reasoning card

State: dummy.next is the constructed head. tail is the last constructed node; during serialization current is the next unread node.

Invariant: Current points to the next unread node.

Decision to rehearse: Advance through links until None.

Finish: Serialize by reading val and following next until None

Transfer exercise

Add a serializer guard that detects a repeated node identity and reports a cycle instead of looping forever.

102. Reverse a Linked List

Foundation | Module 11

Problem

Reverse a finite acyclic singly linked list by changing next links, and return the new head. Reuse the existing nodes; do not reverse by copying values into a separate array. None returns None.

Examples

Example 1. The original nodes are linked in reverse order.

```
list_values(reverse_list(build_list([1,2,3])))
=> [3, 2, 1]
```

Example 2. Both regions are initially empty.

```
reverse_list(None)
=> None
```

Example 3. A singleton remains its own head and tail.

```
list_values(reverse_list(build_list([9])))
=> [9]
```

Why this mechanism works

Changing current.next destroys the route to the unread suffix unless it is saved first. Maintain a reversed prefix and an original-order suffix. Transfer one node from the suffix to the front of the prefix at each step.

State and boundaries

previous is the head of the reversed consumed prefix. current is the head of the unread suffix. following temporarily preserves its next node.

Protocol

1. Initialize previous=None and current=head.
2. Save current.next.
3. Point current.next to previous.
4. Advance previous to current.
5. Advance current to the saved following node.
6. Return previous.

Visualization

```
reversed prefix <- previous | current -> unread suffix
save following; reverse one arrow; advance both
None <- 1 <- 2 <- 3
```

Python implementation

```
from structures import build_list, list_values

def reverse_list(head):
    previous, current = None, head
    while current is not None:
        following = current.next
        current.next = previous
        previous = current
        current = following
    return previous
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
list_values(reverse_list(build_list([1,2,3])))

L4 reverse_list: previous, current = None, head
head=N1(val=1,next=N2,random=None); previous=None; current=N1(val=1,next=N2,random=None)

L6 reverse_list: following = current.next
head=N1(val=1,next=N2,random=None); previous=None; current=N1(val=1,next=N2,random=None);
    following=N2(val=2,next=N3,random=None)

L7 reverse_list: current.next = previous
head=N1(val=1,next=None,random=None); previous=None; current=N1(val=1,next=None,random=None);
    following=N2(val=2,next=N3,random=None)

L8 reverse_list: previous = current
head=N1(val=1,next=None,random=None); previous=N1(val=1,next=None,random=None);
    current=N1(val=1,next=None,random=None); following=N2(val=2,next=N3,random=None)

L9 reverse_list: current = following
head=N1(val=1,next=None,random=None); previous=N1(val=1,next=None,random=None);
    current=N2(val=2,next=N3,random=None); following=N2(val=2,next=N3,random=None)

L6 reverse_list: following = current.next
head=N1(val=1,next=None,random=None); previous=N1(val=1,next=None,random=None);
    current=N2(val=2,next=N3,random=None); following=N3(val=3,next=None,random=None)

L7 reverse_list: current.next = previous
head=N1(val=1,next=None,random=None); previous=N1(val=1,next=None,random=None);
    current=N2(val=2,next=N1,random=None); following=N3(val=3,next=None,random=None)

L8 reverse_list: previous = current
head=N1(val=1,next=None,random=None); previous=N2(val=2,next=N1,random=None);
    current=N2(val=2,next=N1,random=None); following=N3(val=3,next=None,random=None)

... 1 intermediate state changes omitted ...

L6 reverse_list: following = current.next
head=N1(val=1,next=None,random=None); previous=N2(val=2,next=N1,random=None);
```

```
    current=N3(val=3,next=None,random=None); following=None

L7 reverse_list: current.next = previous
head=N1(val=1,next=None,random=None); previous=N2(val=2,next=N1,random=None);
    current=N3(val=3,next=N2,random=None); following=None

L8 reverse_list: previous = current
head=N1(val=1,next=None,random=None); previous=N3(val=3,next=N2,random=None);
    current=N3(val=3,next=N2,random=None); following=None

L9 reverse_list: current = following
head=N1(val=1,next=None,random=None); previous=N3(val=3,next=N2,random=None); current=None;
    following=None
```

Result: [3, 2, 1]. The original nodes are linked in reverse order.

```
reverse_list(None)

L4 reverse_list: previous, current = None, head
head=None; previous=None; current=None
```

Result: None. Both regions are initially empty.

Why the solution is correct

The saved link preserves all unread nodes before mutation. Redirecting one link prepends exactly the current node to the reversed prefix. These disjoint regions still contain all original nodes; when the suffix is empty, previous heads the complete reversal.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Advancing through current.next after overwriting it follows the reversed prefix instead of the unread suffix. Returning the original head returns the new tail.

Reasoning card

State: previous is the head of the reversed consumed prefix. current is the head of the unread suffix. following temporarily preserves its next node.

Invariant: The processed prefix is fully reversed and ends at previous.

Decision to rehearse: Save next, reverse link, advance previous/current.

Finish: Return previous

Transfer exercise

Reverse twice and verify that both the values and original node identities return to their initial order.

103. Merge Two Sorted Linked Lists

Foundation | Module 11

Problem

Merge two nondecreasing, finite, node-disjoint singly linked lists by relinking their existing nodes. Return the sorted merged head. Equal values are taken from the first list before the second. Either head may be None.

Examples

Example 1. Only the selected source advances.

```
list_values(merge_lists(build_list([1,4]), build_list([2,3])))
=> [1, 2, 3, 4]
```

Example 2. The nonempty suffix becomes the result.

```
list_values(merge_lists(None, build_list([2])))
=> [2]
```

Example 3. No real node is attached.

```
merge_lists(None, None)
=> None
```

Why this mechanism works

This is the two-frontier sorted merge with links replacing array indices. A dummy tail gives every appended node the same attachment operation. Once one list ends, its partner's suffix is already sorted and can be attached with one link assignment.

State and boundaries

a and b are unread heads. tail is the last node of the committed sorted chain beginning at dummy.next.

Protocol

1. Compare unread heads while both exist.
2. Link the smaller head after tail.
3. Advance only that source.
4. Advance tail.
5. Attach the remaining suffix.

6. Return dummy.next.

Visualization

```
a: 1 -> 4 -> None
b: 2 -> 3 -> None
merged: dummy -> 1 -> 2 -> 3 -> 4
```

Python implementation

```
from structures import ListNode, build_list, list_values

def merge_lists(a, b):
    dummy = ListNode()
    tail = dummy
    while a is not None and b is not None:
        if a.val <= b.val:
            tail.next, a = a, a.next
        else:
            tail.next, b = b, b.next
        tail = tail.next
    tail.next = a if a is not None else b
    return dummy.next
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
list_values(merge_lists(build_list([1,4]), build_list([2,3])))

L4 merge_lists: dummy = ListNode()
a=N1(val=1,next=N2,random=None); b=N3(val=2,next=N4,random=None);
    dummy=N5(val=0,next=None,random=None)

L5 merge_lists: tail = dummy
a=N1(val=1,next=N2,random=None); b=N3(val=2,next=N4,random=None);
    dummy=N5(val=0,next=None,random=None); tail=N5(val=0,next=None,random=None)

L8 merge_lists: tail.next, a = a, a.next
a=N2(val=4,next=None,random=None); b=N3(val=2,next=N4,random=None);
    dummy=N5(val=0,next=N1,random=None); tail=N5(val=0,next=N1,random=None)

L11 merge_lists: tail = tail.next
a=N2(val=4,next=None,random=None); b=N3(val=2,next=N4,random=None);
    dummy=N5(val=0,next=N1,random=None); tail=N1(val=1,next=N2,random=None)

L10 merge_lists: tail.next, b = b, b.next
a=N2(val=4,next=None,random=None); b=N4(val=3,next=None,random=None);
    dummy=N5(val=0,next=N1,random=None); tail=N1(val=1,next=N3,random=None)

L11 merge_lists: tail = tail.next
a=N2(val=4,next=None,random=None); b=N4(val=3,next=None,random=None);
    dummy=N5(val=0,next=N1,random=None); tail=N3(val=2,next=N4,random=None)

L10 merge_lists: tail.next, b = b, b.next
a=N2(val=4,next=None,random=None); b=None; dummy=N5(val=0,next=N1,random=None);
    tail=N3(val=2,next=N4,random=None)
```

```
L11 merge_lists: tail = tail.next
a=N2(val=4,next=None,random=None); b=None; dummy=N5(val=0,next=N1,random=None);
    tail=N4(val=3,next=None,random=None)

L12 merge_lists: tail.next = a if a is not None else b
a=N2(val=4,next=None,random=None); b=None; dummy=N5(val=0,next=N1,random=None);
    tail=N4(val=3,next=N2,random=None)
```

Result: [1, 2, 3, 4]. Only the selected source advances.

```
list_values(merge_lists(None, build_list([2])))

L4 merge_lists: dummy = ListNode()
a=None; b=N1(val=2,next=None,random=None); dummy=N2(val=0,next=None,random=None)

L5 merge_lists: tail = dummy
a=None; b=N1(val=2,next=None,random=None); dummy=N2(val=0,next=None,random=None);
    tail=N2(val=0,next=None,random=None)

L12 merge_lists: tail.next = a if a is not None else b
a=None; b=N1(val=2,next=None,random=None); dummy=N2(val=0,next=N1,random=None);
    tail=N2(val=0,next=N1,random=None)
```

Result: [2]. The nonempty suffix becomes the result.

Why the solution is correct

The smaller head is no greater than any unread value in either sorted list. Attaching it preserves sorted order and consumes one node. The remaining suffix can be attached after exhaustion because all its values follow the last committed frontier choice.

Time and space

O(n+m) time and O(1) auxiliary space beyond one dummy node; inputs are structurally consumed.

Common mistakes

Advancing a source before saving or attaching its node can lose it. Shared input nodes violate the node-disjoint precondition and can create unintended cycles.

Reasoning card

State: a and b are unread heads. tail is the last node of the committed sorted chain beginning at dummy.next.

Invariant: Everything after dummy through tail is final and sorted.

Decision to rehearse: Use a dummy tail and consume the smaller current node.

Finish: Return dummy.next

Transfer exercise

Merge k sorted linked lists with one heap entry per source, retaining a source identifier for equal values.

104. Find the Middle Node

Foundation | Module 11

Problem

Return the middle node of a finite acyclic singly linked list. For even length, return the second of the two middle nodes. Return None for an empty chain. Return a node reference, not a copied value or list.

Examples

Example 1. Serializing from the returned node reveals the second-middle suffix.

```
list_values(middle_node(build_list([1,2,3,4])))
=> [3, 4]
```

Example 2. The odd-length list has one middle.

```
list_values(middle_node(build_list([1,2,3])))
=> [2, 3]
```

Example 3. There is no middle node.

```
middle_node(None)
=> None
```

Why this mechanism works

Advance a slow pointer once for every two fast-pointer steps. Starting both at the head naturally chooses the second middle for even lengths. The loop guard must establish that both fast and fast.next exist before advancing twice.

State and boundaries

After t iterations slow has moved t links and fast has moved 2t links unless it reached None.

Protocol

1. Start slow and fast at head.

2. While fast and its next exist, move slow once and fast twice.

3. Return slow when fast cannot advance twice.

Visualization

```
1 -> 2 -> 3 -> 4
slow: 1 -> 2 -> 3
fast: 1 ------> 3 ------> None
return node 3
```

Python implementation

```
from structures import build_list, list_values

def middle_node(head):
    slow = fast = head
    while fast is not None and fast.next is not None:
        slow = slow.next
        fast = fast.next.next
    return slow
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
list_values(middle_node(build_list([1,2,3,4])))

L4 middle_node: slow = fast = head
head=N1(val=1,next=N2,random=None); slow=N1(val=1,next=N2,random=None);
    fast=N1(val=1,next=N2,random=None)

L6 middle_node: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N1(val=1,next=N2,random=None)

L7 middle_node: fast = fast.next.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N3(val=3,next=N4,random=None)

L6 middle_node: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N3(val=3,next=N4,random=None);
    fast=N3(val=3,next=N4,random=None)

L7 middle_node: fast = fast.next.next
head=N1(val=1,next=N2,random=None); slow=N3(val=3,next=N4,random=None); fast=None
```

Result: [3, 4]. Serializing from the returned node reveals the second-middle suffix.

```
list_values(middle_node(build_list([1,2,3])))

L4 middle_node: slow = fast = head
head=N1(val=1,next=N2,random=None); slow=N1(val=1,next=N2,random=None);
    fast=N1(val=1,next=N2,random=None)

L6 middle_node: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N1(val=1,next=N2,random=None)

L7 middle_node: fast = fast.next.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N3(val=3,next=None,random=None)
```

Result: [2, 3]. The odd-length list has one middle.

Why the solution is correct

The pointer-distance relation holds after every iteration. For length 2m, slow reaches index m; for length 2m+1, it reaches index m. These are exactly the required second middle and unique middle positions.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Starting fast one node ahead changes even-length tie behavior. Accessing fast.next.next without guarding fast.next can fail.

Reasoning card

State: After t iterations slow has moved t links and fast has moved 2t links unless it reached None.

Invariant: When fast finishes, slow has traversed half as many nodes.

Decision to rehearse: Slow moves one step, fast two.

Finish: Return slow when fast cannot advance twice

Transfer exercise

Return the first middle of an even-length list and explain the changed initialization or stopping condition.

105. Detect a Cycle

Bridge | Module 11

Problem

Return whether following next links from head eventually revisits a node. Values may repeat without creating a cycle. Use constant auxiliary space and leave links unchanged. None is acyclic.

Examples

Example 1. The tail returns to the second node.

```
head = build_list([1,2,3]); head.next.next.next = head.next
has_cycle(head)
=> True
```

Example 2. Equal values do not imply equal node identities.

```
has_cycle(build_list([1,1]))
=> False
```

Example 3. No links can form a cycle.

```
has_cycle(None)
=> False
```

Why this mechanism works

Inside a cycle, a fast pointer gains one step per iteration on a slow pointer. Their relative position modulo the cycle length must eventually become zero. In an acyclic list the fast pointer reaches the end instead, providing the other termination condition.

State and boundaries

slow moves one link and fast moves two. Equality is tested by node identity after movement, not by equal stored values.

Protocol

1. Start both pointers at head.
2. Guard the fast two-step movement.
3. Advance slow once and fast twice.
4. Return True if they are the same node.
5. Return False when fast reaches the end.

Visualization

```
1 -> 2 -> 3
     ^    |
     +----+
fast gains one cycle position per round
```

Python implementation

```
from structures import build_list

def has_cycle(head):
    slow = fast = head
    while fast is not None and fast.next is not None:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            return True
    return False
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
has_cycle(head)

L4 has_cycle: slow = fast = head
head=N1(val=1,next=N2,random=None); slow=N1(val=1,next=N2,random=None);
    fast=N1(val=1,next=N2,random=None)

L6 has_cycle: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N1(val=1,next=N2,random=None)

L7 has_cycle: fast = fast.next.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N3(val=3,next=N2,random=None)

L6 has_cycle: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N3(val=3,next=N2,random=None);
    fast=N3(val=3,next=N2,random=None)
```

Result: True. The tail returns to the second node.

```
has_cycle(build_list([1,1]))

L4 has_cycle: slow = fast = head
head=N1(val=1,next=N2,random=None); slow=N1(val=1,next=N2,random=None);
    fast=N1(val=1,next=N2,random=None)

L6 has_cycle: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N2(val=1,next=None,random=None);
    fast=N1(val=1,next=N2,random=None)

L7 has_cycle: fast = fast.next.next
head=N1(val=1,next=N2,random=None); slow=N2(val=1,next=None,random=None); fast=None
```

Result: False. Equal values do not imply equal node identities.

Why the solution is correct

Without a cycle, fast eventually reaches None and no repeated-node meeting after movement is possible. With a cycle, both enter it and the relative offset advances by one modulo its length, guaranteeing a meeting within one cycle traversal.

Time and space

O(n) time in the number of reachable distinct nodes and O(1) auxiliary space.

Common mistakes

Comparing values confuses duplicate data with shared nodes. Testing equality before the first movement falsely detects a cycle in every nonempty list.

Reasoning card

State: slow moves one link and fast moves two. Equality is tested by node identity after movement, not by equal stored values.

Invariant: Without a cycle, fast reaches None.

Decision to rehearse: Fast eventually meets slow inside a cycle.

Finish: Return False when fast reaches the end

Transfer exercise

After finding a meeting, walk once around the cycle to determine its length.

106. Find Cycle Entry

Bridge | Module 11

Problem

Return the node where a reachable next-link cycle begins, or None if the chain is acyclic. Values may repeat. Do not change links or use a visited set. The cycle entry is the first cycle node reached from head.

Examples

Example 1. The returned reference is the entry node, even with duplicate values.

```
head = build_list([7,7,7]); head.next.next.next = head.next
cycle_entry(head) is head.next
=> True
```

Example 2. The fast pointer reaches the end.

```
cycle_entry(build_list([1,2]))
=> None
```

Example 3. A self-loop begins at the head.

```
head = build_list([9]); head.next = head
cycle_entry(head) is head
=> True
```

Why this mechanism works

First find a slow/fast meeting. If the noncyclic prefix length is mu and cycle length is lambda, the meeting relation implies mu plus the meeting's cycle offset is a multiple of lambda. Moving one pointer from head and one from the meeting at equal speed therefore makes them meet at the entry.

State and boundaries

The first phase locates any meeting in the cycle. In the second, entry starts at head while slow remains at the meeting; both move one link per step.

Protocol

1. Run guarded slow/fast movement until meeting or exhaustion.
2. Return None if there is no meeting.
3. Reset one pointer to head.
4. Advance both one step until identities match.
5. Return that node.

Visualization

```
head -- mu links --> entry -- cycle offset --> meeting
meeting pointer + mu steps wraps to entry
head pointer + mu steps reaches entry
```

Python implementation

```
from structures import build_list

def cycle_entry(head):
    slow = fast = head
    while fast is not None and fast.next is not None:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:
            break
    else:
        return None
    entry = head
    while entry is not slow:
        entry = entry.next
        slow = slow.next
    return entry
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
cycle_entry(head) is head.next

L4 cycle_entry: slow = fast = head
head=N1(val=7,next=N2,random=None); slow=N1(val=7,next=N2,random=None);
    fast=N1(val=7,next=N2,random=None)

L6 cycle_entry: slow = slow.next
head=N1(val=7,next=N2,random=None); slow=N2(val=7,next=N3,random=None);
    fast=N1(val=7,next=N2,random=None)

L7 cycle_entry: fast = fast.next.next
head=N1(val=7,next=N2,random=None); slow=N2(val=7,next=N3,random=None);
    fast=N3(val=7,next=N2,random=None)

L6 cycle_entry: slow = slow.next
head=N1(val=7,next=N2,random=None); slow=N3(val=7,next=N2,random=None);
    fast=N3(val=7,next=N2,random=None)

L12 cycle_entry: entry = head
head=N1(val=7,next=N2,random=None); slow=N3(val=7,next=N2,random=None);
    fast=N3(val=7,next=N2,random=None); entry=N1(val=7,next=N2,random=None)

L14 cycle_entry: entry = entry.next
head=N1(val=7,next=N2,random=None); slow=N3(val=7,next=N2,random=None);
    fast=N3(val=7,next=N2,random=None); entry=N2(val=7,next=N3,random=None)
```

```
L15 cycle_entry: slow = slow.next
head=N1(val=7,next=N2,random=None); slow=N2(val=7,next=N3,random=None);
    fast=N3(val=7,next=N2,random=None); entry=N2(val=7,next=N3,random=None)
```

Result: True. The returned reference is the entry node, even with duplicate values.

```
cycle_entry(build_list([1,2]))

L4 cycle_entry: slow = fast = head
head=N1(val=1,next=N2,random=None); slow=N1(val=1,next=N2,random=None);
    fast=N1(val=1,next=N2,random=None)

L6 cycle_entry: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=None,random=None);
    fast=N1(val=1,next=N2,random=None)

L7 cycle_entry: fast = fast.next.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=None,random=None); fast=None
```

Result: None. The fast pointer reaches the end.

Why the solution is correct

At the first meeting, the fast-minus-slow distance is a multiple of the cycle length, yielding the modular relation for the entry offset. After mu equal-speed steps, the head pointer reaches the entry and the meeting pointer reaches the same cycle position. Before then, the head pointer is outside the cycle, so this is their first match.

Time and space

O(n) time and O(1) auxiliary space.

Common mistakes

Resetting both pointers to head loses the needed meeting offset. Returning the first meeting node is not generally returning the cycle entry.

Reasoning card

State: The first phase locates any meeting in the cycle. In the second, entry starts at head while slow remains at the meeting; both move one link per step.

Invariant: Equal-distance reasoning makes the next meeting the entry.

Decision to rehearse: After a meeting, move one pointer to head and advance both one step.

Finish: Return that node

Transfer exercise

Derive an alternative entry search using the measured cycle length as a fixed gap between two pointers.

107. Remove Nth Node from End

Bridge | Module 11

Problem

Remove the nth node from the end of a nonempty finite singly linked list, with 1<=n<=length. Return the possibly changed head and reuse all remaining nodes. Use one traversal with two pointers and a dummy predecessor.

Examples

Example 1. Node four is second from the end.

```
list_values(remove_nth_from_end(build_list([1,2,3,4,5]), 2))
=> [1, 2, 3, 5]
```

Example 2. Removing the only node returns None.

```
list_values(remove_nth_from_end(build_list([1]), 1))
=> []
```

Example 3. The dummy supports removing the head.

```
list_values(remove_nth_from_end(build_list([1,2]), 2))
=> [2]
```

Why this mechanism works

Place fast n links ahead of slow, starting both at a dummy. Then move them together until fast reaches the tail. Slow is now immediately before the node to remove. The dummy makes removing the head the same link operation as removing any interior node.

State and boundaries

fast stays n links ahead of slow. When fast.next is None, slow.next is the nth node from the end.

Protocol

1. Attach head after a dummy.
2. Move fast n steps.
3. Advance both until fast is the tail.
4. Bypass slow.next.
5. Return dummy.next.

Visualization

```
dummy -> 1 -> 2 -> 3 -> 4 -> 5
n=2: when fast=5, slow=3
bypass 4: 3.next=5
```

Python implementation

```
from structures import ListNode, build_list, list_values

def remove_nth_from_end(head, n):
    dummy = ListNode(next=head)
    slow = fast = dummy
    for _ in range(n):
        fast = fast.next
    while fast.next is not None:
        slow = slow.next
        fast = fast.next
    slow.next = slow.next.next
    return dummy.next
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
list_values(remove_nth_from_end(build_list([1,2,3,4,5]), 2))

L4 remove_nth_from_end: dummy = ListNode(next=head)
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None)

L5 remove_nth_from_end: slow = fast = dummy
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N3(val=0,next=N1,random=None); fast=N3(val=0,next=N1,random=None)

L6 remove_nth_from_end: for _ in range(n):
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N3(val=0,next=N1,random=None); fast=N3(val=0,next=N1,random=None); _=0

L7 remove_nth_from_end: fast = fast.next
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N3(val=0,next=N1,random=None); fast=N1(val=1,next=N2,random=None); _=0

L6 remove_nth_from_end: for _ in range(n):
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N3(val=0,next=N1,random=None); fast=N1(val=1,next=N2,random=None); _=1

L7 remove_nth_from_end: fast = fast.next
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N3(val=0,next=N1,random=None); fast=N2(val=2,next=N4,random=None); _=1

L9 remove_nth_from_end: slow = slow.next
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N1(val=1,next=N2,random=None); fast=N2(val=2,next=N4,random=None); _=1

L10 remove_nth_from_end: fast = fast.next
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N1(val=1,next=N2,random=None); fast=N4(val=3,next=N5,random=None); _=1

... 1 intermediate state changes omitted ...
```

```
L10 remove_nth_from_end: fast = fast.next
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N2(val=2,next=N4,random=None); fast=N5(val=4,next=N6,random=None); _=1

L9 remove_nth_from_end: slow = slow.next
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N4(val=3,next=N5,random=None); fast=N5(val=4,next=N6,random=None); _=1

L10 remove_nth_from_end: fast = fast.next
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N4(val=3,next=N5,random=None); fast=N6(val=5,next=None,random=None); _=1

L11 remove_nth_from_end: slow.next = slow.next.next
head=N1(val=1,next=N2,random=None); n=2; dummy=N3(val=0,next=N1,random=None);
    slow=N4(val=3,next=N6,random=None); fast=N6(val=5,next=None,random=None); _=1
```

Result: [1, 2, 3, 5]. Node four is second from the end.

```
list_values(remove_nth_from_end(build_list([1]), 1))

L4 remove_nth_from_end: dummy = ListNode(next=head)
head=N1(val=1,next=None,random=None); n=1; dummy=N2(val=0,next=N1,random=None)

L5 remove_nth_from_end: slow = fast = dummy
head=N1(val=1,next=None,random=None); n=1; dummy=N2(val=0,next=N1,random=None);
    slow=N2(val=0,next=N1,random=None); fast=N2(val=0,next=N1,random=None)

L6 remove_nth_from_end: for _ in range(n):
head=N1(val=1,next=None,random=None); n=1; dummy=N2(val=0,next=N1,random=None);
    slow=N2(val=0,next=N1,random=None); fast=N2(val=0,next=N1,random=None); _=0

L7 remove_nth_from_end: fast = fast.next
head=N1(val=1,next=None,random=None); n=1; dummy=N2(val=0,next=N1,random=None);
    slow=N2(val=0,next=N1,random=None); fast=N1(val=1,next=None,random=None); _=0

L11 remove_nth_from_end: slow.next = slow.next.next
head=N1(val=1,next=None,random=None); n=1; dummy=N2(val=0,next=None,random=None);
    slow=N2(val=0,next=None,random=None); fast=N1(val=1,next=None,random=None); _=0
```

Result: []. Removing the only node returns None.

Why the solution is correct

The fixed gap is established before tandem movement and preserved by every paired step. At termination exactly n nodes begin at slow.next through the tail, so bypassing slow.next removes the requested node. The dummy preserves a valid predecessor even when that node is the head.

Time and space

O(length) time and O(1) auxiliary space.

Common mistakes

Advancing fast by a different number requires a different stopping condition. Returning head after removing the first node returns a detached node.

Reasoning card

State: fast stays n links ahead of slow. When fast.next is None, slow.next is the nth node from the end.

Invariant: When fast reaches the end, slow precedes the node to remove.

Decision to rehearse: Create an n-node gap, then advance both pointers.

Finish: Return dummy.next

Transfer exercise

Accept invalid n values and choose a clear error contract without partially mutating the list.

108. Reverse a Sublist

Interview | Module 11

Problem

Reverse the nodes at one-based positions left through right, inclusive, in a finite singly linked list. Assume 1<=left<=right<=length. Return the resulting head, preserving all nodes and links outside the specified segment.

Examples

Example 1. Only the inclusive middle segment is reversed.

```
list_values(reverse_sublist(build_list([1,2,3,4,5]), 2, 4))
=> [1, 4, 3, 2, 5]
```

Example 2. The dummy also supports reversing from the head.

```
list_values(reverse_sublist(build_list([1,2,3]), 1, 3))
=> [3, 2, 1]
```

Example 3. A one-node segment requires no transfers.

```
list_values(reverse_sublist(build_list([7]), 1, 1))
=> [7]
```

Why this mechanism works

Keep an anchor immediately before the segment. Repeatedly remove the node after the segment's original first node and insert it immediately after the anchor. Each move extends the reversed prefix while the original first node becomes the fixed tail of the reversed region.

State and boundaries

before is the anchor. current is the original first segment node and remains its tail. moving is the next node transferred to the front.

Protocol

1. Use a dummy and locate before.

2. Set current=before.next.

3. Repeat right-left times: detach current.next.

4. Insert that detached node immediately after before.

5. Return dummy.next.

Visualization

```
1 -> [2 -> 3 -> 4] -> 5
move 3 to front: 1 -> [3 -> 2 -> 4] -> 5
move 4 to front: 1 -> [4 -> 3 -> 2] -> 5
```

Python implementation

```
from structures import ListNode, build_list, list_values

def reverse_sublist(head, left, right):
    dummy = ListNode(next=head)
    before = dummy
    for _ in range(left - 1):
        before = before.next
    current = before.next
    for _ in range(right - left):
        moving = current.next
        current.next = moving.next
        moving.next = before.next
        before.next = moving
    return dummy.next
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
list_values(reverse_sublist(build_list([1,2,3,4,5]), 2, 4))

L4 reverse_sublist: dummy = ListNode(next=head)
head=N1(val=1,next=N2,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None)

L5 reverse_sublist: before = dummy
head=N1(val=1,next=N2,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N3(val=0,next=N1,random=None)

L6 reverse_sublist: for _ in range(left - 1):
head=N1(val=1,next=N2,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N3(val=0,next=N1,random=None); _=0

L7 reverse_sublist: before = before.next
head=N1(val=1,next=N2,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N2,random=None); _=0

L8 reverse_sublist: current = before.next
head=N1(val=1,next=N2,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N2,random=None); _=0; current=N2(val=2,next=N4,random=None)

L10 reverse_sublist: moving = current.next
head=N1(val=1,next=N2,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N2,random=None); _=0; current=N2(val=2,next=N4,random=None);
    moving=N4(val=3,next=N5,random=None)

L11 reverse_sublist: current.next = moving.next
head=N1(val=1,next=N2,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N2,random=None); _=0; current=N2(val=2,next=N5,random=None);
    moving=N4(val=3,next=N5,random=None)

L12 reverse_sublist: moving.next = before.next
```

```
head=N1(val=1,next=N2,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N2,random=None); _=0; current=N2(val=2,next=N5,random=None);
    moving=N4(val=3,next=N2,random=None)

... 2 intermediate state changes omitted ...

L10 reverse_sublist: moving = current.next
head=N1(val=1,next=N4,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N4,random=None); _=1; current=N2(val=2,next=N5,random=None);
    moving=N5(val=4,next=N6,random=None)

L11 reverse_sublist: current.next = moving.next
head=N1(val=1,next=N4,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N4,random=None); _=1; current=N2(val=2,next=N6,random=None);
    moving=N5(val=4,next=N6,random=None)

L12 reverse_sublist: moving.next = before.next
head=N1(val=1,next=N4,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N4,random=None); _=1; current=N2(val=2,next=N6,random=None);
    moving=N5(val=4,next=N4,random=None)

L13 reverse_sublist: before.next = moving
head=N1(val=1,next=N5,random=None); left=2; right=4; dummy=N3(val=0,next=N1,random=None);
    before=N1(val=1,next=N5,random=None); _=1; current=N2(val=2,next=N6,random=None);
    moving=N5(val=4,next=N4,random=None)
```

Result: [1, 4, 3, 2, 5]. Only the inclusive middle segment is reversed.

```
list_values(reverse_sublist(build_list([1,2,3]), 1, 3))

L4 reverse_sublist: dummy = ListNode(next=head)
head=N1(val=1,next=N2,random=None); left=1; right=3; dummy=N3(val=0,next=N1,random=None)

L5 reverse_sublist: before = dummy
head=N1(val=1,next=N2,random=None); left=1; right=3; dummy=N3(val=0,next=N1,random=None);
    before=N3(val=0,next=N1,random=None)

L8 reverse_sublist: current = before.next
head=N1(val=1,next=N2,random=None); left=1; right=3; dummy=N3(val=0,next=N1,random=None);
    before=N3(val=0,next=N1,random=None); current=N1(val=1,next=N2,random=None)

L9 reverse_sublist: for _ in range(right - left):
head=N1(val=1,next=N2,random=None); left=1; right=3; dummy=N3(val=0,next=N1,random=None);
    before=N3(val=0,next=N1,random=None); current=N1(val=1,next=N2,random=None); _=0

L10 reverse_sublist: moving = current.next
head=N1(val=1,next=N2,random=None); left=1; right=3; dummy=N3(val=0,next=N1,random=None);
    before=N3(val=0,next=N1,random=None); current=N1(val=1,next=N2,random=None); _=0;
    moving=N2(val=2,next=N4,random=None)

L11 reverse_sublist: current.next = moving.next
head=N1(val=1,next=N4,random=None); left=1; right=3; dummy=N3(val=0,next=N1,random=None);
    before=N3(val=0,next=N1,random=None); current=N1(val=1,next=N4,random=None); _=0;
    moving=N2(val=2,next=N4,random=None)

L12 reverse_sublist: moving.next = before.next
head=N1(val=1,next=N4,random=None); left=1; right=3; dummy=N3(val=0,next=N1,random=None);
    before=N3(val=0,next=N1,random=None); current=N1(val=1,next=N4,random=None); _=0;
    moving=N2(val=2,next=N1,random=None)

L13 reverse_sublist: before.next = moving
head=N1(val=1,next=N4,random=None); left=1; right=3; dummy=N3(val=0,next=N2,random=None);
    before=N3(val=0,next=N2,random=None); current=N1(val=1,next=N4,random=None); _=0;
    moving=N2(val=2,next=N1,random=None)
```

```
... 1 intermediate state changes omitted ...

L10 reverse_sublist: moving = current.next
head=N1(val=1,next=N4,random=None); left=1; right=3; dummy=N3(val=0,next=N2,random=None);
    before=N3(val=0,next=N2,random=None); current=N1(val=1,next=N4,random=None); _=1;
    moving=N4(val=3,next=None,random=None)

L11 reverse_sublist: current.next = moving.next
head=N1(val=1,next=None,random=None); left=1; right=3; dummy=N3(val=0,next=N2,random=None);
    before=N3(val=0,next=N2,random=None); current=N1(val=1,next=None,random=None); _=1;
    moving=N4(val=3,next=None,random=None)

L12 reverse_sublist: moving.next = before.next
head=N1(val=1,next=None,random=None); left=1; right=3; dummy=N3(val=0,next=N2,random=None);
    before=N3(val=0,next=N2,random=None); current=N1(val=1,next=None,random=None); _=1;
    moving=N4(val=3,next=N2,random=None)

L13 reverse_sublist: before.next = moving
head=N1(val=1,next=None,random=None); left=1; right=3; dummy=N3(val=0,next=N4,random=None);
    before=N3(val=0,next=N4,random=None); current=N1(val=1,next=None,random=None); _=1;
    moving=N4(val=3,next=N2,random=None)
```

Result: [3, 2, 1]. The dummy also supports reversing from the head.

Why the solution is correct

After t transfers, the first t+1 original segment nodes are reversed after the anchor, and current links to the unread segment suffix. One transfer preserves that form. After right-left transfers the entire requested segment is reversed and its outside connections are intact.

Time and space

O(length) time in the worst case and O(1) auxiliary space.

Common mistakes

Moving the anchor during reversal loses the insertion point. Detaching without first retaining moving loses the node being repositioned.

Reasoning card

State: before is the anchor. current is the original first segment node and remains its tail. moving is the next node transferred to the front.

Invariant: Nodes before predecessor and after the active sublist remain connected correctly.

Decision to rehearse: Detach via a dummy predecessor and repeatedly move nodes to the front of the sublist.

Finish: Return dummy.next

Transfer exercise

Reverse nodes in groups of k, deciding explicitly what happens to a final incomplete group.

109. Reorder List

Interview | Module 11

Problem

Reorder a finite singly linked list from L0,L1,...,Ln to L0,Ln,L1,Ln-1,... by changing links rather than values. Return the original head reference, now heading the reordered chain. Empty and singleton chains are unchanged.

Examples

Example 1. An odd-length first half has one final unpaired node.

```
list_values(reorder_list(build_list([1,2,3,4,5])))
=> [1, 5, 2, 4, 3]
```

Example 2. Even halves weave completely.

```
list_values(reorder_list(build_list([1,2,3,4])))
=> [1, 4, 2, 3]
```

Example 3. The empty chain is unchanged.

```
reorder_list(None)
=> None
```

Why this mechanism works

The requested order alternates a forward traversal of the first half with a backward traversal of the second. Split the chain, reverse the second half, then weave the two lists. Severing the split link before weaving prevents the old connection from creating a cycle.

State and boundaries

slow locates the first-half tail. second is the reversed second-half head. During weaving, first and second name the next unconsumed nodes of their respective halves.

Protocol

1. Locate the midpoint with slow/fast pointers.
2. Cut after slow.
3. Reverse the detached second half.
4. Alternately link one node from each half, saving both next pointers first.
5. Return head.

Visualization

```
1->2->3 | 4->5
reverse second: 5->4
weave: 1->5->2->4->3
```

Python implementation

```
from structures import build_list, list_values

def reorder_list(head):
    if head is None or head.next is None:
        return head
    slow, fast = head, head.next
    while fast is not None and fast.next is not None:
        slow = slow.next
        fast = fast.next.next
    current = slow.next
    slow.next = None
    second = None
    while current is not None:
        following = current.next
        current.next = second
        second, current = current, following
    first = head
    while second is not None:
        next_first, next_second = first.next, second.next
        first.next = second
        second.next = next_first
        first, second = next_first, next_second
    return head
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
list_values(reorder_list(build_list([1,2,3,4,5])))

L4 reorder_list: if head is None or head.next is None:
head=N1(val=1,next=N2,random=None)

L6 reorder_list: slow, fast = head, head.next
head=N1(val=1,next=N2,random=None); slow=N1(val=1,next=N2,random=None);
    fast=N2(val=2,next=N3,random=None)

L8 reorder_list: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N2(val=2,next=N3,random=None)

L9 reorder_list: fast = fast.next.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N4(val=4,next=N5,random=None)

L8 reorder_list: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N3(val=3,next=N4,random=None);
    fast=N4(val=4,next=N5,random=None)

L9 reorder_list: fast = fast.next.next
```

```
head=N1(val=1,next=N2,random=None); slow=N3(val=3,next=N4,random=None); fast=None

L10 reorder_list: current = slow.next
head=N1(val=1,next=N2,random=None); slow=N3(val=3,next=N4,random=None); fast=None;
    current=N4(val=4,next=N5,random=None)

L11 reorder_list: slow.next = None
head=N1(val=1,next=N2,random=None); slow=N3(val=3,next=None,random=None); fast=None;
    current=N4(val=4,next=N5,random=None)

... 12 intermediate state changes omitted ...

L19 reorder_list: next_first, next_second = first.next, second.next
head=N1(val=1,next=N5,random=None); slow=N3(val=3,next=None,random=None); fast=None;
    current=None; second=N4(val=4,next=None,random=None); following=None;
    first=N2(val=2,next=N3,random=None); next_first=N3(val=3,next=None,random=None);
    next_second=None

L20 reorder_list: first.next = second
head=N1(val=1,next=N5,random=None); slow=N3(val=3,next=None,random=None); fast=None;
    current=None; second=N4(val=4,next=None,random=None); following=None;
    first=N2(val=2,next=N4,random=None); next_first=N3(val=3,next=None,random=None);
    next_second=None

L21 reorder_list: second.next = next_first
head=N1(val=1,next=N5,random=None); slow=N3(val=3,next=None,random=None); fast=None;
    current=None; second=N4(val=4,next=N3,random=None); following=None;
    first=N2(val=2,next=N4,random=None); next_first=N3(val=3,next=None,random=None);
    next_second=None

L22 reorder_list: first, second = next_first, next_second
head=N1(val=1,next=N5,random=None); slow=N3(val=3,next=None,random=None); fast=None;
    current=None; second=None; following=None; first=N3(val=3,next=None,random=None);
    next_first=N3(val=3,next=None,random=None); next_second=None
```

Result: [1, 5, 2, 4, 3]. An odd-length first half has one final unpaired node.

```
list_values(reorder_list(build_list([1,2,3,4])))

L4 reorder_list: if head is None or head.next is None:
head=N1(val=1,next=N2,random=None)

L6 reorder_list: slow, fast = head, head.next
head=N1(val=1,next=N2,random=None); slow=N1(val=1,next=N2,random=None);
    fast=N2(val=2,next=N3,random=None)

L8 reorder_list: slow = slow.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N2(val=2,next=N3,random=None)

L9 reorder_list: fast = fast.next.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N4(val=4,next=None,random=None)

L10 reorder_list: current = slow.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N4(val=4,next=None,random=None); current=N3(val=3,next=N4,random=None)

L11 reorder_list: slow.next = None
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=None,random=None);
    fast=N4(val=4,next=None,random=None); current=N3(val=3,next=N4,random=None)

L12 reorder_list: second = None
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=None,random=None);
```

```
    fast=N4(val=4,next=None,random=None); current=N3(val=3,next=N4,random=None); second=None

L14 reorder_list: following = current.next
head=N1(val=1,next=N2,random=None); slow=N2(val=2,next=None,random=None);
    fast=N4(val=4,next=None,random=None); current=N3(val=3,next=N4,random=None); second=None;
    following=N4(val=4,next=None,random=None)

... 9 intermediate state changes omitted ...

L22 reorder_list: first, second = next_first, next_second
head=N1(val=1,next=N4,random=None); slow=N2(val=2,next=None,random=None);
    fast=N4(val=4,next=N2,random=None); current=None; second=N3(val=3,next=None,random=None);
    following=None; first=N2(val=2,next=None,random=None);
    next_first=N2(val=2,next=None,random=None); next_second=N3(val=3,next=None,random=None)

L19 reorder_list: next_first, next_second = first.next, second.next
head=N1(val=1,next=N4,random=None); slow=N2(val=2,next=None,random=None);
    fast=N4(val=4,next=N2,random=None); current=None; second=N3(val=3,next=None,random=None);
    following=None; first=N2(val=2,next=None,random=None); next_first=None; next_second=None

L20 reorder_list: first.next = second
head=N1(val=1,next=N4,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N4(val=4,next=N2,random=None); current=None; second=N3(val=3,next=None,random=None);
    following=None; first=N2(val=2,next=N3,random=None); next_first=None; next_second=None

L22 reorder_list: first, second = next_first, next_second
head=N1(val=1,next=N4,random=None); slow=N2(val=2,next=N3,random=None);
    fast=N4(val=4,next=N2,random=None); current=None; second=None; following=None; first=None;
     next_first=None; next_second=None
```

Result: [1, 4, 2, 3]. Even halves weave completely.

Why the solution is correct

The split gives the first half at least as many nodes as the second. Reversal makes the second half appear in the needed end-to-middle order. Each weave step commits the next required pair and preserves both unread suffixes, so all nodes appear once without a cycle.

Time and space

O(n) time across three linear phases and O(1) auxiliary space.

Common mistakes

Leaving the midpoint connection intact can create a cycle. Overwriting a next link before saving both suffixes can lose remaining nodes.

Reasoning card

State: slow locates the first-half tail. second is the reversed second-half head. During weaving, first and second name the next unconsumed nodes of their respective halves.

Invariant: Each phase has a separate structural invariant.

Decision to rehearse: Find middle, reverse second half, merge alternating nodes.

Finish: Return head

Transfer exercise

Write a structural test that checks node identity preservation, absence of cycles and the final value order independently.

110. Copy List with Random Pointer

Interview | Module 11

Problem

Deep-copy a finite acyclic next-chain whose nodes also have random links to any node in that same chain or None. Return a new head with identical values and next/random relationships, sharing no nodes with the original. Original links must remain unchanged.

Examples

Example 1. The clone preserves both cross-links using fresh identities.

```
head=build_list([1,2]); head.random=head.next; head.next.random=head;
    clone=copy_random_list(head)
(clone.val, clone.next.val, clone.random is clone.next, clone.next.random is clone,
    clone is not head)
=> (1, 2, True, True, True)
```

Example 2. An empty chain has an empty copy.

```
copy_random_list(None)
=> None
```

Example 3. A self-random link must point to the clone itself.

```
head=build_list([7]); head.random=head; clone=copy_random_list(head)
(clone is not head, clone.random is clone)
=> (True, True)
```

Why this mechanism works

Values are not unique identifiers, so map original node identities to clone identities. First allocate every clone; only then wire links. This separation ensures that forward random references already have destinations when they are connected.

State and boundaries

clones maps each original node to exactly one fresh node. After allocation, every legal next or random target has a mapped clone.

Protocol

1. Walk next links and allocate one clone per original node.

2. Walk the chain again.

3. Set clone.next from the next-node mapping.

4. Set clone.random from the random-node mapping.

5. Return the mapped head.

Visualization

```
original A.next=B, A.random=B, B.random=A
map A->A', B->B'
clone A'.next=B', A'.random=B', B'.random=A'
```

Python implementation

```
from structures import ListNode, build_list

def copy_random_list(head):
    clones = {}
    current = head
    while current is not None:
        clones[current] = ListNode(current.val)
        current = current.next
    current = head
    while current is not None:
        clones[current].next = clones.get(current.next)
        clones[current].random = clones.get(current.random)
        current = current.next
    return clones.get(head)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
(clone.val, clone.next.val, clone.random is clone.next, clone.next.random is clone,
    clone is not head)

L4 copy_random_list: clones = {}
head=N1(val=1,next=N2,random=N2); clones={}

L5 copy_random_list: current = head
head=N1(val=1,next=N2,random=N2); clones={}; current=N1(val=1,next=N2,random=N2)

L7 copy_random_list: clones[current] = ListNode(current.val)
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=None,random=None)};
    current=N1(val=1,next=N2,random=N2)

L8 copy_random_list: current = current.next
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=None,random=None)};
    current=N2(val=2,next=None,random=N1)

L7 copy_random_list: clones[current] = ListNode(current.val)
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=None,random=None),
    N2(val=2,next=None,random=N1):...; current=N2(val=2,next=None,random=N1)

L8 copy_random_list: current = current.next
```

```
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=None,random=None),
    N2(val=2,next=None,random=N1):...; current=None

L9 copy_random_list: current = head
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=None,random=None),
    N2(val=2,next=None,random=N1):...; current=N1(val=1,next=N2,random=N2)

L11 copy_random_list: clones[current].next = clones.get(current.next)
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=N4,random=None),
    N2(val=2,next=None,random=N1):N4...; current=N1(val=1,next=N2,random=N2)

L12 copy_random_list: clones[current].random = clones.get(current.random)
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=N4,random=N4),
    N2(val=2,next=None,random=N1):N4(v...; current=N1(val=1,next=N2,random=N2)

L13 copy_random_list: current = current.next
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=N4,random=N4),
    N2(val=2,next=None,random=N1):N4(v...; current=N2(val=2,next=None,random=N1)

L13 copy_random_list: current = current.next
head=N1(val=1,next=N2,random=N2);
    clones={N1(val=1,next=N2,random=N2):N3(val=1,next=N4,random=N4),
    N2(val=2,next=None,random=N1):N4(v...; current=None
```

Result: (1, 2, True, True, True). The clone preserves both cross-links using fresh identities.

```
copy_random_list(None)

L4 copy_random_list: clones = {}
head=None; clones={}

L5 copy_random_list: current = head
head=None; clones={}; current=None
```

Result: None. An empty chain has an empty copy.

Why the solution is correct

The first pass creates a bijection between reachable original and fresh nodes. The second pass maps each edge through that bijection, preserving structure and aliasing patterns without sharing identities. Since no original field is assigned, the source remains unchanged.

Time and space

O(n) expected time and O(n) mapping space plus O(n) returned nodes.

Common mistakes

Mapping by value merges distinct equal-valued nodes. Copying a random pointer directly leaves the clone linked into the original structure.

Reasoning card

State: clones maps each original node to exactly one fresh node. After allocation, every legal next or random target has a mapped clone.

Invariant: Every copied pointer targets a copied node or None.

Decision to rehearse: Map originals to copies, then wire references; or interleave copies.

Finish: Return the mapped head

Transfer exercise

Explore the interleaving-clones method that removes the dictionary, and explain how the original next links must be restored.

Module 12
Binary Search and Monotone Answer Spaces

Maintain a candidate interval and use one comparison or feasibility test to discard half safely.

Binary search is controlled elimination. Start from a candidate interval and a statement explaining what remains possible inside it. Exact lookup, lower bound, rotated lookup and answer-space search use related boundary mechanics but different predicates. The rotated-array lesson expands both successful branches in detail so the familiar updates retain a precise meaning.

The ten-step progression

111. Exact Binary Search

112. Lower Bound

113. Upper Bound

114. First and Last Target Position

115. Integer Square Root

116. Search Rotated Sorted Array

117. Find Minimum in Rotated Sorted Array

118. Search a Row-Major Sorted Matrix

119. Minimum Eating Speed

120. Minimum Shipping Capacity

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

111. Exact Binary Search

Foundation | Module 12

Problem

Return an index of target in a nondecreasing integer list, or -1 if it is absent. With duplicates, return the first equality encountered by the specified midpoint search, not necessarily the first occurrence. Empty input is valid. Do not use a library search function.

Examples

Example 1. The target survives successive interval exclusions.

```
binary_search([1,3,5,7,9], 7)
=> 3
```

Example 2. No midpoint is read from an empty list.

```
binary_search([], 2)
=> -1
```

Example 3. The single rejected candidate is removed.

```
binary_search([4], 3)
=> -1
```

Why this mechanism works

Sorted order allows one comparison to reject half the candidates. Use [left,right) so right is a boundary, not an element. After checking and rejecting mid, the remaining left candidates are [left,mid) and the remaining right candidates are [mid+1,right). Equality may terminate immediately.

State and boundaries

If target exists, at least one matching index remains in [left,right). mid is a valid index only while left<right.

Protocol

1. Start left=0,right=n.
2. While the interval is nonempty compute mid.
3. Return mid on equality.
4. Set left=mid+1 when nums[mid]<target.
5. Otherwise set right=mid.

6. Return -1 when candidates are exhausted.

Visualization

```
candidates: [left ........ right)
reject mid -> [left,mid) or [mid+1,right)
right itself is never a candidate cell
```

Python implementation

```
def binary_search(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        if nums[mid] < target:
            left = mid + 1
        else:
            right = mid
    return -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
binary_search([1,3,5,7,9], 7)

L2 binary_search: left, right = 0, len(nums)
nums=[1, 3, 5, 7, 9]; target=7; left=0; right=5

L4 binary_search: mid = (left + right) // 2
nums=[1, 3, 5, 7, 9]; target=7; left=0; right=5; mid=2

L8 binary_search: left = mid + 1
nums=[1, 3, 5, 7, 9]; target=7; left=3; right=5; mid=2

L4 binary_search: mid = (left + right) // 2
nums=[1, 3, 5, 7, 9]; target=7; left=3; right=5; mid=4

L10 binary_search: right = mid
nums=[1, 3, 5, 7, 9]; target=7; left=3; right=4; mid=4

L4 binary_search: mid = (left + right) // 2
nums=[1, 3, 5, 7, 9]; target=7; left=3; right=4; mid=3
```

Result: 3. The target survives successive interval exclusions.

```
binary_search([], 2)

L2 binary_search: left, right = 0, len(nums)
nums=[]; target=2; left=0; right=0
```

Result: -1. No midpoint is read from an empty list.

Why the solution is correct

A smaller midpoint value rules out every index through mid; a larger one rules out every index from mid onward. Each update preserves any possible match and strictly shrinks the interval. Equality returns a valid match, while an empty interval proves absence.

Time and space

O(log(n+1)) time and O(1) auxiliary space.

Common mistakes

Using left=mid may not shrink a one-element interval. Reading nums[right] violates the exclusive boundary convention when right==n.

Reasoning card

State: If target exists, at least one matching index remains in [left,right). mid is a valid index only while left<right.

Invariant: Target, if present, remains inside the candidate interval.

Decision to rehearse: Choose one interval convention and preserve it consistently.

Finish: Return -1 when candidates are exhausted

Transfer exercise

Change the output to the first matching index. Explain why immediate return on equality no longer satisfies the contract.

112. Lower Bound

Foundation | Module 12

Problem

Return the first index i with nums[i]>=target in a nondecreasing integer list. Return n when no element qualifies. This insertion boundary is defined even when target is absent or the list is empty.

Examples

Example 1. Equality keeps searching for an earlier qualifying position.

```
lower_bound([1,3,3,3,8], 3)
=> 1
```

Example 2. The answer may be the exclusive end boundary.

```
lower_bound([1,2], 9)
=> 2
```

Example 3. The empty list has insertion boundary zero.

```
lower_bound([], 1)
=> 0
```

Why this mechanism works

The predicate nums[i]>=target is false, then true. An equality is not a final answer because an earlier equal value may exist. Keep searching left on every true midpoint, while a false midpoint and all earlier indices can be excluded.

State and boundaries

Indices before left are known false; indices at or after right are known true when they exist. The first-true boundary lies between them.

Protocol

1. Start [left,right)=[0,n).
2. Test the midpoint predicate.
3. On True set right=mid.
4. On False set left=mid+1.
5. Return the meeting boundary.

Visualization

```
values: 1 3 3 3 8, target 3
predicate: F T T T T
first True index: 1
```

Python implementation

```
def lower_bound(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] >= target:
            right = mid
        else:
            left = mid + 1
    return left
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
lower_bound([1,3,3,3,8], 3)

L2 lower_bound: left, right = 0, len(nums)
nums=[1, 3, 3, 3, 8]; target=3; left=0; right=5

L4 lower_bound: mid = (left + right) // 2
nums=[1, 3, 3, 3, 8]; target=3; left=0; right=5; mid=2

L6 lower_bound: right = mid
nums=[1, 3, 3, 3, 8]; target=3; left=0; right=2; mid=2

L4 lower_bound: mid = (left + right) // 2
nums=[1, 3, 3, 3, 8]; target=3; left=0; right=2; mid=1

L6 lower_bound: right = mid
nums=[1, 3, 3, 3, 8]; target=3; left=0; right=1; mid=1

L4 lower_bound: mid = (left + right) // 2
nums=[1, 3, 3, 3, 8]; target=3; left=0; right=1; mid=0

L8 lower_bound: left = mid + 1
nums=[1, 3, 3, 3, 8]; target=3; left=1; right=1; mid=0
```

Result: 1. Equality keeps searching for an earlier qualifying position.

```
lower_bound([1,2], 9)

L2 lower_bound: left, right = 0, len(nums)
nums=[1, 2]; target=9; left=0; right=2

L4 lower_bound: mid = (left + right) // 2
nums=[1, 2]; target=9; left=0; right=2; mid=1

L8 lower_bound: left = mid + 1
nums=[1, 2]; target=9; left=2; right=2; mid=1
```

Result: 2. The answer may be the exclusive end boundary.

Why the solution is correct

Monotonicity makes the whole prefix through a false midpoint false and the whole suffix from a true midpoint true. Each update narrows the unknown boundary region. When left==right, all earlier positions are false and that position is the first true one, or n if none exists.

Time and space

O(log(n+1)) time and O(1) auxiliary space.

Common mistakes

Returning immediately on equality can choose a later duplicate. Treating n as an error discards a legitimate insertion boundary.

Reasoning card

State: Indices before left are known false; indices at or after right are known true when they exist. The first-true boundary lies between them.

Invariant: All indices before left are known too small.

Decision to rehearse: A true midpoint remains a candidate, so move right to middle.

Finish: Return the meeting boundary

Transfer exercise

Use lower bound to insert a value before all existing equal values.

113. Upper Bound

Foundation | Module 12

Problem

Return the first index i with nums[i]>target in a nondecreasing integer list, or n if none exists. Equal values belong before the returned boundary. Empty input returns zero.

Examples

Example 1. All copies of three lie before the boundary.

```
upper_bound([1,3,3,3,8], 3)
=> 4
```

Example 2. No strictly greater element exists.

```
upper_bound([3,3], 3)
=> 2
```

Example 3. The empty boundary is zero.

```
upper_bound([], 0)
=> 0
```

Why this mechanism works

Upper bound changes only the predicate from >= to >. That small change moves the boundary past the complete run of target values. The interval mechanics remain identical; copying a new pointer template is unnecessary.

State and boundaries

Indices before left have values<=target; indices from right onward have values>target. The first strictly greater boundary remains between them.

Protocol

1. Initialize left=0,right=n.
2. On nums[mid]>target move right to mid.
3. Otherwise move left to mid+1.
4. Return left at convergence.

Visualization

```
values: 1 3 3 3 8, target 3
predicate >3: F F F F T
upper bound: 4
```

Python implementation

```
def upper_bound(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] > target:
            right = mid
        else:
            left = mid + 1
    return left
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
upper_bound([1,3,3,3,8], 3)

L2 upper_bound: left, right = 0, len(nums)
nums=[1, 3, 3, 3, 8]; target=3; left=0; right=5

L4 upper_bound: mid = (left + right) // 2
nums=[1, 3, 3, 3, 8]; target=3; left=0; right=5; mid=2

L8 upper_bound: left = mid + 1
nums=[1, 3, 3, 3, 8]; target=3; left=3; right=5; mid=2

L4 upper_bound: mid = (left + right) // 2
nums=[1, 3, 3, 3, 8]; target=3; left=3; right=5; mid=4

L6 upper_bound: right = mid
nums=[1, 3, 3, 3, 8]; target=3; left=3; right=4; mid=4

L4 upper_bound: mid = (left + right) // 2
nums=[1, 3, 3, 3, 8]; target=3; left=3; right=4; mid=3

L8 upper_bound: left = mid + 1
nums=[1, 3, 3, 3, 8]; target=3; left=4; right=4; mid=3
```

Result: 4. All copies of three lie before the boundary.

```
upper_bound([3,3], 3)

L2 upper_bound: left, right = 0, len(nums)
nums=[3, 3]; target=3; left=0; right=2

L4 upper_bound: mid = (left + right) // 2
nums=[3, 3]; target=3; left=0; right=2; mid=1

L8 upper_bound: left = mid + 1
nums=[3, 3]; target=3; left=2; right=2; mid=1
```

Result: 2. No strictly greater element exists.

Why the solution is correct

A false midpoint belongs to the <=target prefix and can be excluded with everything before it. A true midpoint bounds the >target suffix. Convergence identifies their unique boundary, including n when the suffix is empty.

Time and space

O(log(n+1)) time and O(1) auxiliary space.

Common mistakes

Reusing the lower-bound equality branch returns the beginning rather than the end of a duplicate run. right remains exclusive throughout.

Reasoning card

State: Indices before left have values<=target; indices from right onward have values>target. The first strictly greater boundary remains between them.

Invariant: All indices before left are <= target when the search ends.

Decision to rehearse: Change the predicate while keeping the same boundary-search skeleton.

Finish: Return left at convergence

Transfer exercise

Count occurrences of target as upper_bound-lower_bound without scanning the duplicate run.

114. First and Last Target Position

Foundation | Module 12

Problem

Return [first,last], the inclusive zero-based positions of target in a nondecreasing integer list, or [-1,-1] if absent. Use logarithmic searches even when target occupies most of the list.

Examples

Example 1. The exclusive end boundary five becomes last index four.

```
target_range([5,7,7,8,8,10], 8)
=> [3, 4]
```

Example 2. The lower bound points to seven, so six is absent.

```
target_range([5,7,7,8,8,10], 6)
=> [-1, -1]
```

Example 3. The lower boundary equals the list length.

```
target_range([], 0)
=> [-1, -1]
```

Why this mechanism works

The target run is the half-open interval [lower_bound(target),upper_bound(target)). Check whether its lower boundary actually points to target before converting the upper boundary to an inclusive last index. This composes two already-proven searches instead of expanding linearly from an arbitrary match.

State and boundaries

first is the first value>=target. end is the first value>target, so a present target occupies [first,end).

Protocol

1. Find lower bound.
2. Reject if it is n or its value is not target.
3. Find upper bound.
4. Return first and end-1.

Visualization

```
values: 5 7 7 8 8 10
target run: indices [3,5)
reported inclusive endpoints: [3,4]
```

Python implementation

```
from kata_112 import lower_bound
from kata_113 import upper_bound

def target_range(nums, target):
    first = lower_bound(nums, target)
    if first == len(nums) or nums[first] != target:
        return [-1, -1]
    return [first, upper_bound(nums, target) - 1]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
target_range([5,7,7,8,8,10], 8)

L5 target_range: first = lower_bound(nums, target)
nums=[5, 7, 7, 8, 8, 10]; target=8; first=3
```

Result: [3, 4]. The exclusive end boundary five becomes last index four.

```
target_range([5,7,7,8,8,10], 6)

L5 target_range: first = lower_bound(nums, target)
nums=[5, 7, 7, 8, 8, 10]; target=6; first=1
```

Result: [-1, -1]. The lower bound points to seven, so six is absent.

Why the solution is correct

Sorted order places all equal targets in one consecutive run. Lower and upper bounds identify its exact two boundaries. The presence check distinguishes an empty target run from a real one, and subtracting one converts its exclusive end to the required inclusive index.

Time and space

O(log(n+1)) time and O(1) auxiliary space; the imported helpers are katas 112 and 113.

Common mistakes

Subtracting one before checking presence can invent a range for an absent value. Linear expansion around a found target can be O(n).

Reasoning card

State: first is the first value>=target. end is the first value>target, so a present target occupies [first,end).

Invariant: Each helper returns a mathematically defined boundary.

Decision to rehearse: Compose two boundary searches instead of special-case scanning.

Finish: Return first and end-1

Transfer exercise

Return the count or the half-open target interval directly, avoiding unnecessary conversion when the caller uses slicing.

115. Integer Square Root

Bridge | Module 12

Problem

For a nonnegative integer n, return floor(sqrt(n)) without using square-root functions. The result r must satisfy r*r<=n<(r+1)*(r+1). Zero is allowed.

Examples

Example 1. Four squared fits; five squared exceeds seventeen.

```
integer_sqrt(17)
=> 4
```

Example 2. The first excluded square belongs to one.

```
integer_sqrt(0)
=> 0
```

Example 3. Perfect squares retain their exact integer root.

```
integer_sqrt(16)
=> 4
```

Why this mechanism works

Search for the first integer whose square exceeds n, then subtract one. Squares are monotone over nonnegative integers, creating a false-then-true predicate. The search boundary n+1 is sufficient even for the small cases zero and one, where it may serve as the first excluded sentinel.

State and boundaries

left and right bound the first integer x with x*x>n. All smaller confirmed candidates satisfy x*x<=n.

Protocol

1. Search integers with left=0,right=n+1.
2. On mid*mid>n set right=mid.
3. Otherwise set left=mid+1.
4. Return left-1.

Visualization

```
n=17
x:       0 1 2 3 4 5 ...
x*x>17: F F F F F T ...
first True 5 -> floor root 4
```

Python implementation

```
def integer_sqrt(n):
    left, right = 0, n + 1
    while left < right:
        mid = (left + right) // 2
        if mid * mid > n:
            right = mid
        else:
            left = mid + 1
    return left - 1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
integer_sqrt(17)

L2 integer_sqrt: left, right = 0, n + 1
n=17; left=0; right=18

L4 integer_sqrt: mid = (left + right) // 2
n=17; left=0; right=18; mid=9

L6 integer_sqrt: right = mid
n=17; left=0; right=9; mid=9

L4 integer_sqrt: mid = (left + right) // 2
n=17; left=0; right=9; mid=4

L8 integer_sqrt: left = mid + 1
n=17; left=5; right=9; mid=4

L4 integer_sqrt: mid = (left + right) // 2
n=17; left=5; right=9; mid=7

L6 integer_sqrt: right = mid
n=17; left=5; right=7; mid=7

L4 integer_sqrt: mid = (left + right) // 2
n=17; left=5; right=7; mid=6

L6 integer_sqrt: right = mid
n=17; left=5; right=6; mid=6

L4 integer_sqrt: mid = (left + right) // 2
n=17; left=5; right=6; mid=5

L6 integer_sqrt: right = mid
n=17; left=5; right=5; mid=5
```

Result: 4. Four squared fits; five squared exceeds seventeen.

```
integer_sqrt(0)

L2 integer_sqrt: left, right = 0, n + 1
n=0; left=0; right=1

L4 integer_sqrt: mid = (left + right) // 2
n=0; left=0; right=1; mid=0

L8 integer_sqrt: left = mid + 1
n=0; left=1; right=1; mid=0
```

Result: 0. The first excluded square belongs to one.

Why the solution is correct

Monotonic squares allow the standard first-true boundary proof. At convergence left is the first integer with square too large, including a sentinel beyond the tested range in the smallest cases. Its predecessor is therefore the largest integer with square at most n.

Time and space

O(log(n+1)) integer comparisons and multiplications; O(1) integer variables. Big-integer multiplication has bit-length-dependent cost.

Common mistakes

Searching for exact square equality fails for nonsquares. Floating-point square roots can round near large perfect squares.

Reasoning card

State: left and right bound the first integer x with x*x>n. All smaller confirmed candidates satisfy x*x<=n.

Invariant: Feasible and infeasible regions remain separated.

Decision to rehearse: Binary-search feasible values x where x*x <= n.

Finish: Return left-1

Transfer exercise

In a fixed-width language, replace mid*mid with a division-based comparison while guarding the zero case.

116. Search Rotated Sorted Array

Bridge | Module 12

Problem

A strictly increasing integer array has been rotated at an unknown boundary, possibly not rotated at all. Its values remain distinct. Return the zero-based index of target, or -1 if absent. Empty input is valid. Use O(log n) time with a half-open candidate interval.

Examples

Example 1. The target begins in the sorted right side.

```
search_rotated([15,18,20,1,3,5,8,9,13], 5)
=> 5
```

Example 2. The first decision keeps the other, rotated-looking side.

```
search_rotated([15,18,20,1,3,5,8,9,13], 18)
=> 1
```

Example 3. No midpoint is accessed.

```
search_rotated([], 3)
=> -1
```

Example 4. Equality is checked before any range exclusion.

```
search_rotated([7], 7)
=> 0
```

Example 5. A value between present neighbors may still be absent.

```
search_rotated([15,18,20,1,3,5,8,9,13], 2)
=> -1
```

Why this mechanism works

Rotation breaks global sortedness but introduces at most one descending break. Around a midpoint, at least one side is still normally sorted. Identify that trustworthy side, test whether target lies within its value range, and keep either it or the opposite side. Do not launch a separate binary search inside that side. The local sortedness fact is used only to justify one safe exclusion in the current search.

State and boundaries

[left,right) contains every remaining candidate. right-1 is the last candidate index. Equality at mid is checked before strict range tests exclude it.

Protocol

1. Compute mid inside the nonempty candidate interval.

2. Return immediately on equality.

3. If nums[left]<=nums[mid], the left side is sorted.

4. Keep it only when nums[left]<=target<nums[mid].

5. Otherwise the right side is sorted: keep it only when nums[mid]<target<=nums[right-1].

6. Use right=mid or left=mid+1 to discard the rejected midpoint.

7. Return -1 if the interval empties.

Visualization

```
values: 15 18 20 | 1 3 5 8 9 13
index:   0  1  2   3 4 5 6 7  8
left=0, mid=4, right=9 (boundary after 13)
15>3 -> right side [3,5,8,9,13] is sorted
```

Complete boundary walkthroughs

Why one side remains trustworthy

Think of the array as two increasing blocks: [15,18,20] followed by [1,3,5,8,9,13]. There is only one descending edge, 20 to 1. Two overlapping midpoint sides cannot both contain that one edge in their interiors. Distinct values make their endpoints reveal which side is ordered. A later candidate interval may contain the break or lie entirely inside one increasing block; the same test still applies.

Complete trace for target 5

Iteration 1: [left,right)=[0,9), mid=4, value=3. Since 15>3, the right side is sorted. The test 3<5<=13 is true, so discard the checked midpoint and keep [5,9).

Iteration 2: [5,9), mid=7, value=9. Now 5<=9, so the left side is sorted. The test 5<=5<9 is true. Set right=7, leaving [5,7).

Iteration 3: [5,7), mid=6, value=8. The left side is sorted and 5<=5<8 is true. Set right=6, leaving [5,6).

Iteration 4: [5,6), mid=5, value=5. Equality succeeds and returns index 5. The exclusive right boundary is 6; it is not a value to read.

Complete trace for target 18

Iteration 1: [0,9), mid=4, value=3. The right side is sorted, but 3<18<=13 is false. If 18 exists, it must be in the other side. Set right=4, preserving [0,4), whose values are [15,18,20,1]. It is safe to keep a rotated-looking side; it need not already be sorted as a whole.

Iteration 2: [0,4), mid=2, value=20. The left side [15,18,20] is sorted and 15<=18<20 is true. Set right=2, leaving [0,2).

Iteration 3: [0,2), mid=1, value=18. Return index 1.

Why the two updates are asymmetric

After rejecting mid, the candidates on its left occupy [left,mid). Assigning right=mid retains exactly that interval because right is excluded. Candidates on its right occupy [mid+1,right), so left must become mid+1. With left=3,right=4,mid=3, assigning left=mid changes nothing and cannot prove progress.

Why this is not one global first-true search

The comparison nums[i]>=target can change truth more than once across a rotation. This exact search relies on a local sorted-half argument instead of pretending that global predicate is monotone. The next kata shows a different task where a carefully chosen predicate relative to the fixed final value does become monotone.

Python implementation

```
def search_rotated(nums, target):
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid
            else:
                left = mid + 1
        else:
            if nums[mid] < target <= nums[right - 1]:
                left = mid + 1
            else:
                right = mid
    return -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
search_rotated([15,18,20,1,3,5,8,9,13], 5)

L2 search_rotated: left, right = 0, len(nums)
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=5; left=0; right=9

L4 search_rotated: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=5; left=0; right=9; mid=4

L14 search_rotated: left = mid + 1
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=5; left=5; right=9; mid=4
```

```
L4 search_rotated: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=5; left=5; right=9; mid=7

L9 search_rotated: right = mid
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=5; left=5; right=7; mid=7

L4 search_rotated: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=5; left=5; right=7; mid=6

L9 search_rotated: right = mid
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=5; left=5; right=6; mid=6

L4 search_rotated: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=5; left=5; right=6; mid=5
```

Result: 5. The target begins in the sorted right side.

```
search_rotated([15,18,20,1,3,5,8,9,13], 18)

L2 search_rotated: left, right = 0, len(nums)
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=18; left=0; right=9

L4 search_rotated: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=18; left=0; right=9; mid=4

L16 search_rotated: right = mid
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=18; left=0; right=4; mid=4

L4 search_rotated: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=18; left=0; right=4; mid=2

L9 search_rotated: right = mid
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=18; left=0; right=2; mid=2

L4 search_rotated: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; target=18; left=0; right=2; mid=1
```

Result: 1. The first decision keeps the other, rotated-looking side.

Why the solution is correct

With distinct values and one rotation break, nums[left]<=nums[mid] identifies a sorted left side; otherwise the right side must be sorted. A target outside the sorted side's endpoint range cannot occur there. A target inside that range cannot occur in the disjoint value block on the other side. Each branch therefore preserves a possible match while discarding mid, which was already checked. The interval strictly shrinks, so the search terminates.

Time and space

O(log(n+1)) time and O(1) auxiliary space. Distinctness is essential to this logarithmic sorted-half decision.

Common mistakes

left=mid can loop forever on one candidate. nums[right] is outside the interval. Duplicates can make nums[left]==nums[mid]==nums[right-1], hiding the rotation and invalidating this exact decision rule.

Reasoning card

State: [left,right) contains every remaining candidate. right-1 is the last candidate index. Equality at mid is checked before strict range tests exclude it.

Invariant: The target-containing region is never discarded.

Decision to rehearse: At least one half is sorted; determine whether target lies inside it.

Finish: Return -1 if the interval empties

Transfer exercise

Permit duplicates, construct an ambiguous equal-endpoint example, and explain why discarding one equal boundary can degrade the worst case to linear time.

117. Find Minimum in Rotated Sorted Array

Bridge | Module 12

Problem

Return the smallest value in a nonempty rotation of a strictly increasing integer array. Distinctness is guaranteed; an unrotated array is allowed. Use a half-open binary search and do not scan for the descending edge.

Examples

Example 1. The first value in the low block is the minimum.

```
rotated_minimum([15,18,20,1,3,5,8,9,13])
=> 1
```

Example 2. All predicate values are true when unrotated.

```
rotated_minimum([1,2,3])
=> 1
```

Example 3. The only candidate is also the last value.

```
rotated_minimum([7])
=> 7
```

Why this mechanism works

Compare every candidate with the fixed final array value. Values in the rotated high block are greater than it; values in the low block are <= it. The predicate nums[i]<=nums[-1] is therefore false then true, and its first true index is the minimum. For an unrotated array, it is true everywhere.

State and boundaries

last is fixed for the whole search. left and right bound the first index whose value is<=last.

Protocol

1. Remember the final value.
2. Run a first-true search on nums[mid]<=last.
3. Move right to mid on True.
4. Move left to mid+1 on False.

5. Return the value at the boundary.

Visualization

```
[15,18,20,1,3,5,8,9,13], last=13
predicate <=13: F F F T T T T T T
first True points to 1
```

Python implementation

```
def rotated_minimum(nums):
    last = nums[-1]
    left, right = 0, len(nums)
    while left < right:
        mid = (left + right) // 2
        if nums[mid] <= last:
            right = mid
        else:
            left = mid + 1
    return nums[left]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
rotated_minimum([15,18,20,1,3,5,8,9,13])

L2 rotated_minimum: last = nums[-1]
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; last=13

L3 rotated_minimum: left, right = 0, len(nums)
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; last=13; left=0; right=9

L5 rotated_minimum: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; last=13; left=0; right=9; mid=4

L7 rotated_minimum: right = mid
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; last=13; left=0; right=4; mid=4

L5 rotated_minimum: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; last=13; left=0; right=4; mid=2

L9 rotated_minimum: left = mid + 1
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; last=13; left=3; right=4; mid=2

L5 rotated_minimum: mid = (left + right) // 2
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; last=13; left=3; right=4; mid=3

L7 rotated_minimum: right = mid
nums=[15, 18, 20, 1, 3, 5, 8, 9, ...]; last=13; left=3; right=3; mid=3
```

Result: 1. The first value in the low block is the minimum.

```
rotated_minimum([1,2,3])

L2 rotated_minimum: last = nums[-1]
nums=[1, 2, 3]; last=3

L3 rotated_minimum: left, right = 0, len(nums)
```

```
nums=[1, 2, 3]; last=3; left=0; right=3

L5 rotated_minimum: mid = (left + right) // 2
nums=[1, 2, 3]; last=3; left=0; right=3; mid=1

L7 rotated_minimum: right = mid
nums=[1, 2, 3]; last=3; left=0; right=1; mid=1

L5 rotated_minimum: mid = (left + right) // 2
nums=[1, 2, 3]; last=3; left=0; right=1; mid=0

L7 rotated_minimum: right = mid
nums=[1, 2, 3]; last=3; left=0; right=0; mid=0
```

Result: 1. All predicate values are true when unrotated.

Why the solution is correct

Strictly increasing source order separates the rotated high and low blocks relative to the final value. The minimum starts the low block, exactly the first true position. Standard monotone-boundary updates preserve that boundary and converge to it.

Time and space

O(log n) time and O(1) auxiliary space.

Common mistakes

Changing last to a moving boundary value changes the predicate and requires a different proof. An empty array has no minimum and is excluded by the contract.

Reasoning card

State: last is fixed for the whole search. left and right bound the first index whose value is<=last.

Invariant: The minimum remains inside [left,right].

Decision to rehearse: Compare middle with right to locate the unsorted boundary.

Finish: Return the value at the boundary

Transfer exercise

Return the rotation index instead of its value, then use it to map ordinary binary-search positions into the rotated array.

118. Search a Row-Major Sorted Matrix

Interview | Module 12

Problem

Return whether target occurs in a rectangular matrix whose row-major sequence is globally nondecreasing. Each row is ordered and every row's first value is at least the previous row's last value. Empty matrices or zero columns return False.

Examples

Example 1. Virtual index four maps to row one, column one.

```
search_matrix([[1,3,5],[7,9,11]], 9)
=> True
```

Example 2. Six lies between values but does not occur.

```
search_matrix([[1,3,5],[7,9,11]], 6)
=> False
```

Example 3. There is no virtual candidate interval.

```
search_matrix([], 1)
=> False
```

Why this mechanism works

The matrix is one sorted sequence stored in two dimensions. A virtual index mid maps to row mid//cols and column mid%cols. Search virtual boundaries without copying the matrix into a flat array.

State and boundaries

[left,right) is a candidate range in the virtual rows*cols sequence. divmod converts each midpoint into one matrix cell.

Protocol

1. Handle an empty matrix.
2. Set the virtual search interval to [0,rows*cols).
3. Map each midpoint to a row and column.
4. Apply exact binary-search comparison.
5. Return False on exhaustion.

Visualization

```
matrix [[1,3,5],[7,9,11]]
virtual [1,3,5,7,9,11]
mid=4 -> divmod(4,3)=(1,1) -> value 9
```

Python implementation

```
def search_matrix(matrix, target):
    if not matrix or not matrix[0]:
        return False
    rows, cols = len(matrix), len(matrix[0])
    left, right = 0, rows * cols
    while left < right:
        mid = (left + right) // 2
        r, c = divmod(mid, cols)
        value = matrix[r][c]
        if value == target:
            return True
        if value < target:
            left = mid + 1
        else:
            right = mid
    return False
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
search_matrix([[1,3,5],[7,9,11]], 9)

L2 search_matrix: if not matrix or not matrix[0]:
matrix=[[1, 3, 5], [7, 9, 11]]; target=9

L4 search_matrix: rows, cols = len(matrix), len(matrix[0])
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3

L5 search_matrix: left, right = 0, rows * cols
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=0; right=6

L7 search_matrix: mid = (left + right) // 2
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=0; right=6; mid=3

L8 search_matrix: r, c = divmod(mid, cols)
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=0; right=6; mid=3; r=1; c=0

L9 search_matrix: value = matrix[r][c]
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=0; right=6; mid=3; r=1; c=0;
    value=7

L13 search_matrix: left = mid + 1
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=4; right=6; mid=3; r=1; c=0;
    value=7

L7 search_matrix: mid = (left + right) // 2
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=4; right=6; mid=5; r=1; c=0;
    value=7

... 2 intermediate state changes omitted ...
```

```
L15 search_matrix: right = mid
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=4; right=5; mid=5; r=1; c=2;
    value=11

L7 search_matrix: mid = (left + right) // 2
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=4; right=5; mid=4; r=1; c=2;
    value=11

L8 search_matrix: r, c = divmod(mid, cols)
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=4; right=5; mid=4; r=1; c=1;
    value=11

L9 search_matrix: value = matrix[r][c]
matrix=[[1, 3, 5], [7, 9, 11]]; target=9; rows=2; cols=3; left=4; right=5; mid=4; r=1; c=1;
    value=9
```

Result: True. Virtual index four maps to row one, column one.

```
search_matrix([[1,3,5],[7,9,11]], 6)

L2 search_matrix: if not matrix or not matrix[0]:
matrix=[[1, 3, 5], [7, 9, 11]]; target=6

L4 search_matrix: rows, cols = len(matrix), len(matrix[0])
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3

L5 search_matrix: left, right = 0, rows * cols
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=0; right=6

L7 search_matrix: mid = (left + right) // 2
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=0; right=6; mid=3

L8 search_matrix: r, c = divmod(mid, cols)
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=0; right=6; mid=3; r=1; c=0

L9 search_matrix: value = matrix[r][c]
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=0; right=6; mid=3; r=1; c=0;
    value=7

L15 search_matrix: right = mid
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=0; right=3; mid=3; r=1; c=0;
    value=7

L7 search_matrix: mid = (left + right) // 2
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=0; right=3; mid=1; r=1; c=0;
    value=7

... 3 intermediate state changes omitted ...

L7 search_matrix: mid = (left + right) // 2
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=2; right=3; mid=2; r=0; c=1;
    value=3

L8 search_matrix: r, c = divmod(mid, cols)
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=2; right=3; mid=2; r=0; c=2;
    value=3

L9 search_matrix: value = matrix[r][c]
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=2; right=3; mid=2; r=0; c=2;
    value=5
```

```
L13 search_matrix: left = mid + 1
matrix=[[1, 3, 5], [7, 9, 11]]; target=6; rows=2; cols=3; left=3; right=3; mid=2; r=0; c=2;
    value=5
```

Result: False. Six lies between values but does not occur.

Why the solution is correct

Row-major global order makes the virtual sequence sorted. The index mapping is a bijection between virtual positions and cells, preserving that order. Therefore the usual exact-search exclusion proof applies without any physical flattening.

Time and space

O(log(rows*cols+1)) time and O(1) auxiliary space.

Common mistakes

Merely sorting each row independently is not enough for this method. Dividing by the row count instead of the column count maps positions incorrectly.

Reasoning card

State: [left,right) is a candidate range in the virtual rows*cols sequence. divmod converts each midpoint into one matrix cell.

Invariant: The virtual candidate interval contains the target if present.

Decision to rehearse: Map virtual index to row/column.

Finish: Return False on exhaustion

Transfer exercise

When rows and columns are individually sorted but row-major order is not, derive a top-right staircase search instead.

119. Minimum Eating Speed

Interview | Module 12

Problem

Given positive integer pile sizes and h available hours, choose the smallest positive integer speed. In one hour you process up to speed units from one pile; unused capacity in that hour cannot transfer to another pile. Return None if h is smaller than the number of nonempty piles, and zero for no piles.

Examples

Example 1. Speed four is the first to fit eight hours.

```
minimum_eating_speed([3,6,7,11], 8)
=> 4
```

Example 2. Two piles require at least two separate hours.

```
minimum_eating_speed([2,3], 1)
=> None
```

Example 3. No work needs no processing speed.

```
minimum_eating_speed([], 0)
=> 0
```

Why this mechanism works

At speed s, each pile needs ceil(pile/s) hours. Increasing speed cannot increase total hours, so feasibility is false then true. With at least one hour per pile, speed max(piles) is feasible, giving a finite upper bound for a first-true search.

State and boundaries

left and right bound the first feasible speed. hours is a complete feasibility calculation for the current midpoint.

Protocol

1. Handle no piles or too few hours.
2. Search speeds in [1,max(piles)+1).
3. Compute total ceiling-divided hours.
4. On feasibility move right to mid.
5. Otherwise move left to mid+1.
6. Return the first feasible speed.

Visualization

```
piles 3,6,7,11; h=8
speed 3 -> 1+2+3+4=10 hours
speed 4 -> 1+2+2+3=8 hours
first feasible speed is 4
```

Python implementation

```
def minimum_eating_speed(piles, h):
    if not piles:
        return 0
    if h < len(piles):
        return None
    left, right = 1, max(piles) + 1
    while left < right:
        mid = (left + right) // 2
        hours = sum((pile + mid - 1) // mid for pile in piles)
        if hours <= h:
            right = mid
        else:
            left = mid + 1
    return left
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
minimum_eating_speed([3,6,7,11], 8)

L2 minimum_eating_speed: if not piles:
piles=[3, 6, 7, 11]; h=8

L6 minimum_eating_speed: left, right = 1, max(piles) + 1
piles=[3, 6, 7, 11]; h=8; left=1; right=12

L8 minimum_eating_speed: mid = (left + right) // 2
piles=[3, 6, 7, 11]; h=8; left=1; right=12; mid=6

L9 <genexpr>: hours = sum((pile + mid - 1) // mid for pile in piles)
.0=list_iterator; pile=3; mid=6

L9 <genexpr>: hours = sum((pile + mid - 1) // mid for pile in piles)
.0=list_iterator; pile=6; mid=6

L9 <genexpr>: hours = sum((pile + mid - 1) // mid for pile in piles)
.0=list_iterator; pile=7; mid=6

L9 <genexpr>: hours = sum((pile + mid - 1) // mid for pile in piles)
.0=list_iterator; pile=11; mid=6

L9 <genexpr>: hours = sum((pile + mid - 1) // mid for pile in piles)
.0=list_iterator; pile=11; mid=6

... 21 intermediate state changes omitted ...

L9 <genexpr>: hours = sum((pile + mid - 1) // mid for pile in piles)
.0=list_iterator; pile=7; mid=4

L9 <genexpr>: hours = sum((pile + mid - 1) // mid for pile in piles)
```

```
.0=list_iterator; pile=11; mid=4

L9 <genexpr>: hours = sum((pile + mid - 1) // mid for pile in piles)
.0=list_iterator; pile=11; mid=4

L11 minimum_eating_speed: right = mid
piles=[3, 6, 7, 11]; h=8; left=4; right=4; mid=4; hours=8
```

Result: 4. Speed four is the first to fit eight hours.

```
minimum_eating_speed([2,3], 1)

L2 minimum_eating_speed: if not piles:
piles=[2, 3]; h=1
```

Result: None. Two piles require at least two separate hours.

Why the solution is correct

The per-pile ceiling formula follows from the no-cross-pile hour rule. Its sum is nonincreasing with speed. Binary search therefore finds the first feasible integer, and the established feasible upper bound guarantees an answer when the preliminary impossibility check passes.

Time and space

O(p log M) time and O(1) auxiliary space for p piles and maximum size M.

Common mistakes

Using floor division underestimates hours for partial final work. Searching array indices instead of speed values models the wrong answer space.

Reasoning card

State: left and right bound the first feasible speed. hours is a complete feasibility calculation for the current midpoint.

Invariant: Speeds below the answer fail; at or above it succeed.

Decision to rehearse: Binary-search a monotone feasibility predicate.

Finish: Return the first feasible speed

Transfer exercise

Stop the feasibility scan once accumulated hours exceeds h; explain why this optimization cannot change its Boolean answer.

120. Minimum Shipping Capacity

Interview | Module 12

Problem

Given positive package weights in fixed order and a positive number of days, return the smallest integer ship capacity that sends all packages within that many days. Each day takes a contiguous next portion whose total weight fits capacity. Packages cannot be split or reordered. Empty input returns zero.

Examples

Example 1. Capacity six allows the three displayed contiguous loads.

```
minimum_shipping_capacity([3,2,2,4,1,4], 3)
=> 6
```

Example 2. One day must carry the full sum.

```
minimum_shipping_capacity([1,2,3], 1)
=> 6
```

Example 3. No packages need no capacity.

```
minimum_shipping_capacity([], 2)
=> 0
```

Why this mechanism works

For a proposed capacity, greedily load the longest possible next prefix each day. Taking fewer packages cannot help later days when all weights are positive and order is fixed. Larger capacities never require more days, creating a monotone feasibility predicate. Bounds are the heaviest package and the total weight.

State and boundaries

used counts days opened and load is the weight assigned to the current day. The outer half-open interval bounds the first feasible capacity.

Protocol

1. Handle empty input.
2. Search capacity from max(weights) through sum(weights).
3. Simulate ordered loading at the midpoint.
4. Open a new day when the next package would overflow.
5. Narrow toward the first capacity using no more than days.

Visualization

```
weights 3,2,2,4,1,4; days=3
capacity 6: [3,2] | [2,4] | [1,4]
capacity 5 requires more days
```

Python implementation

```
def minimum_shipping_capacity(weights, days):
    if not weights:
        return 0
    left, right = max(weights), sum(weights) + 1
    while left < right:
        mid = (left + right) // 2
        used, load = 1, 0
        for weight in weights:
            if load + weight > mid:
                used += 1
                load = 0
            load += weight
        if used <= days:
            right = mid
        else:
            left = mid + 1
    return left
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
minimum_shipping_capacity([3,2,2,4,1,4], 3)

L2 minimum_shipping_capacity: if not weights:
weights=[3, 2, 2, 4, 1, 4]; days=3

L4 minimum_shipping_capacity: left, right = max(weights), sum(weights) + 1
weights=[3, 2, 2, 4, 1, 4]; days=3; left=4; right=17

L6 minimum_shipping_capacity: mid = (left + right) // 2
weights=[3, 2, 2, 4, 1, 4]; days=3; left=4; right=17; mid=10

L7 minimum_shipping_capacity: used, load = 1, 0
weights=[3, 2, 2, 4, 1, 4]; days=3; left=4; right=17; mid=10; used=1; load=0

L8 minimum_shipping_capacity: for weight in weights:
weights=[3, 2, 2, 4, 1, 4]; days=3; left=4; right=17; mid=10; used=1; load=0; weight=3

L12 minimum_shipping_capacity: load += weight
weights=[3, 2, 2, 4, 1, 4]; days=3; left=4; right=17; mid=10; used=1; load=3; weight=3

L8 minimum_shipping_capacity: for weight in weights:
weights=[3, 2, 2, 4, 1, 4]; days=3; left=4; right=17; mid=10; used=1; load=3; weight=2

L12 minimum_shipping_capacity: load += weight
weights=[3, 2, 2, 4, 1, 4]; days=3; left=4; right=17; mid=10; used=1; load=5; weight=2

... 62 intermediate state changes omitted ...

L12 minimum_shipping_capacity: load += weight
```

```
weights=[3, 2, 2, 4, 1, 4]; days=3; left=6; right=7; mid=6; used=3; load=1; weight=1

L8 minimum_shipping_capacity: for weight in weights:
weights=[3, 2, 2, 4, 1, 4]; days=3; left=6; right=7; mid=6; used=3; load=1; weight=4

L12 minimum_shipping_capacity: load += weight
weights=[3, 2, 2, 4, 1, 4]; days=3; left=6; right=7; mid=6; used=3; load=5; weight=4

L14 minimum_shipping_capacity: right = mid
weights=[3, 2, 2, 4, 1, 4]; days=3; left=6; right=6; mid=6; used=3; load=5; weight=4
```

Result: 6. Capacity six allows the three displayed contiguous loads.

```
minimum_shipping_capacity([1,2,3], 1)

L2 minimum_shipping_capacity: if not weights:
weights=[1, 2, 3]; days=1

L4 minimum_shipping_capacity: left, right = max(weights), sum(weights) + 1
weights=[1, 2, 3]; days=1; left=3; right=7

L6 minimum_shipping_capacity: mid = (left + right) // 2
weights=[1, 2, 3]; days=1; left=3; right=7; mid=5

L7 minimum_shipping_capacity: used, load = 1, 0
weights=[1, 2, 3]; days=1; left=3; right=7; mid=5; used=1; load=0

L8 minimum_shipping_capacity: for weight in weights:
weights=[1, 2, 3]; days=1; left=3; right=7; mid=5; used=1; load=0; weight=1

L12 minimum_shipping_capacity: load += weight
weights=[1, 2, 3]; days=1; left=3; right=7; mid=5; used=1; load=1; weight=1

L8 minimum_shipping_capacity: for weight in weights:
weights=[1, 2, 3]; days=1; left=3; right=7; mid=5; used=1; load=1; weight=2

L12 minimum_shipping_capacity: load += weight
weights=[1, 2, 3]; days=1; left=3; right=7; mid=5; used=1; load=3; weight=2

... 10 intermediate state changes omitted ...

L12 minimum_shipping_capacity: load += weight
weights=[1, 2, 3]; days=1; left=6; right=7; mid=6; used=1; load=3; weight=2

L8 minimum_shipping_capacity: for weight in weights:
weights=[1, 2, 3]; days=1; left=6; right=7; mid=6; used=1; load=3; weight=3

L12 minimum_shipping_capacity: load += weight
weights=[1, 2, 3]; days=1; left=6; right=7; mid=6; used=1; load=6; weight=3

L14 minimum_shipping_capacity: right = mid
weights=[1, 2, 3]; days=1; left=6; right=6; mid=6; used=1; load=6; weight=3
```

Result: 6. One day must carry the full sum.

Why the solution is correct

For fixed capacity, the greedy first day carries at least as long a prefix as any feasible first day. Repeating on the suffix proves it minimizes required days. This minimum day count is nonincreasing with capacity, so first-true binary search returns the smallest feasible capacity.

Time and space

O(n log(S-M+1)) time for sum S and maximum M, and O(1) auxiliary space.

Common mistakes

Sorting the weights violates shipment order. A lower bound below the heaviest package permits a feasibility routine to place an impossible single package.

Reasoning card

State: used counts days opened and load is the weight assigned to the current day. The outer half-open interval bounds the first feasible capacity.

Invariant: Capacities form a false-to-true monotone search space.

Decision to rehearse: Simulate required days for a candidate capacity.

Finish: Narrow toward the first capacity using no more than days

Transfer exercise

Return the actual daily groups for the selected capacity, and distinguish using at most days from requiring exactly days nonempty groups.

Module 13
Matrix Traversal and Simulation

Coordinate row/column indices, shrinking boundaries, and old-versus-new state in two dimensions.

A matrix adds a coordinate system, not an excuse to abandon state definitions. Name row and column meanings, traversal direction, and whether an update reads original or already-modified cells. In-place transformations depend on preserving enough old information before it is overwritten.

The ten-step progression

121. Row and Column Totals

122. Transpose a Matrix

123. Rotate a Square Matrix Clockwise

124. Spiral Traversal

125. Set Matrix Zeroes

126. Diagonal Zigzag Traversal

127. Game of Life Step

128. Matrix Reshape

129. Sparse Matrix Multiplication

130. Validate Sudoku Board

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

121. Row and Column Totals

Foundation | Module 13

Problem

For a rectangular integer matrix, return (row_totals,column_totals). Each row total includes all its columns and each column total includes all its rows. Empty input returns two empty lists. A matrix with rows but zero columns has zero for each row total.

Examples

Example 1. A nonsquare matrix distinguishes the two dimensions.

```
matrix_totals([[1,2,3],[4,5,6]])
=> ([6, 15], [5, 7, 9])
```

Example 2. Neither dimension has entries.

```
matrix_totals([])
=> ([], [])
```

Example 3. Empty rows still have zero row totals.

```
matrix_totals([[],[]])
=> ([0, 0], [])
```

Why this mechanism works

Each cell contributes to exactly two independent summaries, one named by its row and one by its column. A single nested traversal can update both. The row and column indices are coordinates, while their corresponding accumulators have different output dimensions.

State and boundaries

row_totals has one accumulator per row; column_totals has one per column. Each includes exactly the cells consumed so far in that row or column.

Protocol

1. Allocate zero accumulators for both dimensions.

2. Visit every (r,c) cell.

3. Add its value to row_totals[r] and column_totals[c].

4. Return both lists.

Visualization

```
1 2 3 -> row total 6
4 5 6 -> row total 15
| | |
5 7 9 -> column totals
```

Python implementation

```
def matrix_totals(matrix):
    rows = len(matrix)
    cols = len(matrix[0]) if rows else 0
    row_totals, column_totals = [0] * rows, [0] * cols
    for r in range(rows):
        for c in range(cols):
            row_totals[r] += matrix[r][c]
            column_totals[c] += matrix[r][c]
    return row_totals, column_totals
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
matrix_totals([[1,2,3],[4,5,6]])

L2 matrix_totals: rows = len(matrix)
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2

L3 matrix_totals: cols = len(matrix[0]) if rows else 0
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3

L4 matrix_totals: row_totals, column_totals = [0] * rows, [0] * cols
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[0, 0]; column_totals=[0, 0, 0]

L5 matrix_totals: for r in range(rows):
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[0, 0]; column_totals=[0, 0, 0]; r=0

L6 matrix_totals: for c in range(cols):
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[0, 0]; column_totals=[0, 0, 0];
    r=0; c=0

L7 matrix_totals: row_totals[r] += matrix[r][c]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[1, 0]; column_totals=[0, 0, 0];
    r=0; c=0

L8 matrix_totals: column_totals[c] += matrix[r][c]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[1, 0]; column_totals=[1, 0, 0];
    r=0; c=0

L6 matrix_totals: for c in range(cols):
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[1, 0]; column_totals=[1, 0, 0];
    r=0; c=1

... 11 intermediate state changes omitted ...

L8 matrix_totals: column_totals[c] += matrix[r][c]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[6, 9]; column_totals=[5, 7, 3];
    r=1; c=1

L6 matrix_totals: for c in range(cols):
```

```
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[6, 9]; column_totals=[5, 7, 3];
    r=1; c=2

L7 matrix_totals: row_totals[r] += matrix[r][c]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[6, 15]; column_totals=[5, 7, 3];
    r=1; c=2

L8 matrix_totals: column_totals[c] += matrix[r][c]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; row_totals=[6, 15]; column_totals=[5, 7, 9];
    r=1; c=2
```

Result: ([6, 15], [5, 7, 9]). A nonsquare matrix distinguishes the two dimensions.

```
matrix_totals([])

L2 matrix_totals: rows = len(matrix)
matrix=[]; rows=0

L3 matrix_totals: cols = len(matrix[0]) if rows else 0
matrix=[]; rows=0; cols=0

L4 matrix_totals: row_totals, column_totals = [0] * rows, [0] * cols
matrix=[]; rows=0; cols=0; row_totals=[]; column_totals=[]
```

Result: ([], []). Neither dimension has entries.

Why the solution is correct

Each cell is visited once and added once to each relevant accumulator. No other row or column receives that contribution. Therefore each final accumulator contains precisely the sum of its own cells.

Time and space

O(rows*cols) time and O(rows+cols) returned storage.

Common mistakes

Allocating column totals using the row count fails for nonsquare matrices. Swapping r and c can pass square examples while remaining wrong.

Reasoning card

State: row_totals has one accumulator per row; column_totals has one per column. Each includes exactly the cells consumed so far in that row or column.

Invariant: Each cell contributes once to its row and column accumulator.

Decision to rehearse: Reset row-local state; retain column-global state.

Finish: Return both lists

Transfer exercise

Also compute main and anti-diagonal totals for a square matrix without another traversal.

122. Transpose a Matrix

Foundation | Module 13

Problem

Return the transpose of a rectangular matrix: original cell (r,c) becomes output cell (c,r). Do not mutate the input. An empty matrix or a matrix with zero columns transposes to an empty outer list.

Examples

Example 1. Rows and columns exchange roles.

```
transpose([[1,2,3],[4,5,6]])
=> [[1, 4], [2, 5], [3, 6]]
```

Example 2. There are no columns to become rows.

```
transpose([])
=> []
```

Example 3. The single coordinate maps to itself.

```
transpose([[7]])
=> [[7]]
```

Why this mechanism works

The transformation exchanges coordinate roles. Consequently the output has as many rows as the input has columns, and each output row is one complete original column. Deriving the dimensions first prevents a loop that works only for square inputs.

State and boundaries

output[c][r] receives matrix[r][c]. The output shape is cols by rows.

Protocol

1. Determine row and column counts.
2. Allocate cols independent rows of length rows.
3. Copy each cell to exchanged coordinates.
4. Return output.

Visualization



Python implementation

```
def transpose(matrix):
    rows = len(matrix)
    cols = len(matrix[0]) if rows else 0
    output = [[0] * rows for _ in range(cols)]
    for r in range(rows):
        for c in range(cols):
            output[c][r] = matrix[r][c]
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
transpose([[1,2,3],[4,5,6]])

L2 transpose: rows = len(matrix)
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2

L3 transpose: cols = len(matrix[0]) if rows else 0
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3

L4 transpose: output = [[0] * rows for _ in range(cols)]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; _=0

L4 transpose: output = [[0] * rows for _ in range(cols)]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; _=1

L4 transpose: output = [[0] * rows for _ in range(cols)]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; _=2

L4 transpose: output = [[0] * rows for _ in range(cols)]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; output=[[0, 0], [0, 0], [0, 0]]

L5 transpose: for r in range(rows):
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; output=[[0, 0], [0, 0], [0, 0]]; r=0

L6 transpose: for c in range(cols):
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; output=[[0, 0], [0, 0], [0, 0]]; r=0; c=0

... 8 intermediate state changes omitted ...

L6 transpose: for c in range(cols):
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; output=[[1, 4], [2, 0], [3, 0]]; r=1; c=1

L7 transpose: output[c][r] = matrix[r][c]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; output=[[1, 4], [2, 5], [3, 0]]; r=1; c=1

L6 transpose: for c in range(cols):
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; output=[[1, 4], [2, 5], [3, 0]]; r=1; c=2

L7 transpose: output[c][r] = matrix[r][c]
matrix=[[1, 2, 3], [4, 5, 6]]; rows=2; cols=3; output=[[1, 4], [2, 5], [3, 6]]; r=1; c=2
```

Result: [[1, 4], [2, 5], [3, 6]]. Rows and columns exchange roles.

```
transpose([])

L2 transpose: rows = len(matrix)
matrix=[]; rows=0
```

```
L3 transpose: cols = len(matrix[0]) if rows else 0
matrix=[]; rows=0; cols=0

L4 transpose: output = [[0] * rows for _ in range(cols)]
matrix=[]; rows=0; cols=0; output=[]
```

Result: []. There are no columns to become rows.

Why the solution is correct

The coordinate exchange is a one-to-one mapping from input cells to output cells. Every cell is copied once to exactly its transposed position, preserving all values and producing the required dimensions.

Time and space

O(rows*cols) time and O(rows*cols) output space.

Common mistakes

Using [[0]*rows]*cols aliases output rows. Swapping entries in place is not a general rectangular transpose.

Reasoning card

State: output[c][r] receives matrix[r][c]. The output shape is cols by rows.

Invariant: output[c][r] equals input[r][c].

Decision to rehearse: Swap the roles of row and column in the output.

Finish: Return output

Transfer exercise

For nonempty rectangular matrices, verify that transposing twice restores the original values and shape.

123. Rotate a Square Matrix Clockwise

Foundation | Module 13

Problem

Rotate an n-by-n matrix 90 degrees clockwise in place and return the same matrix. Reuse its existing rows and cells rather than allocating another matrix. n may be zero.

Examples

Example 1. Every coordinate follows the clockwise mapping.

```
rotate_clockwise([[1,2,3],[4,5,6],[7,8,9]])
=> [[7, 4, 1], [8, 5, 2], [9, 6, 3]]
```

Example 2. The zero-size matrix requires no swaps.

```
rotate_clockwise([])
=> []
```

Example 3. The center remains fixed.

```
rotate_clockwise([[5]])
=> [[5]]
```

Why this mechanism works

Clockwise rotation maps (r,c) to (c,n-1-r). Decompose that mapping into a transpose, which produces (c,r), followed by reversing each row, which changes its column to n-1-r. Swap only one triangular half during transposition so pairs are not undone.

State and boundaries

After transposition, row r holds original column r. Row reversal then establishes the final clockwise coordinate mapping.

Protocol

1. Swap matrix[r][c] with matrix[c][r] only for c>r.

2. Reverse each row in place.

3. Return the matrix.

Visualization

```
1 2 3       7 4 1
4 5 6  ->   8 5 2
7 8 9       9 6 3
```

Python implementation

```
def rotate_clockwise(matrix):
    n = len(matrix)
    for r in range(n):
        for c in range(r + 1, n):
            matrix[r][c], matrix[c][r] = matrix[c][r], matrix[r][c]
    for row in matrix:
        row.reverse()
    return matrix
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
rotate_clockwise([[1,2,3],[4,5,6],[7,8,9]])

L2 rotate_clockwise: n = len(matrix)
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; n=3

L3 rotate_clockwise: for r in range(n):
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; n=3; r=0

L4 rotate_clockwise: for c in range(r + 1, n):
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; n=3; r=0; c=1

L5 rotate_clockwise: matrix[r][c], matrix[c][r] = matrix[c][r], matrix[r][c]
matrix=[[1, 4, 3], [2, 5, 6], [7, 8, 9]]; n=3; r=0; c=1

L4 rotate_clockwise: for c in range(r + 1, n):
matrix=[[1, 4, 3], [2, 5, 6], [7, 8, 9]]; n=3; r=0; c=2

L5 rotate_clockwise: matrix[r][c], matrix[c][r] = matrix[c][r], matrix[r][c]
matrix=[[1, 4, 7], [2, 5, 6], [3, 8, 9]]; n=3; r=0; c=2

L3 rotate_clockwise: for r in range(n):
matrix=[[1, 4, 7], [2, 5, 6], [3, 8, 9]]; n=3; r=1; c=2

L5 rotate_clockwise: matrix[r][c], matrix[c][r] = matrix[c][r], matrix[r][c]
matrix=[[1, 4, 7], [2, 5, 8], [3, 6, 9]]; n=3; r=1; c=2

... 3 intermediate state changes omitted ...

L6 rotate_clockwise: for row in matrix:
matrix=[[7, 4, 1], [2, 5, 8], [3, 6, 9]]; n=3; r=2; c=2; row=[2, 5, 8]

L7 rotate_clockwise: row.reverse()
matrix=[[7, 4, 1], [8, 5, 2], [3, 6, 9]]; n=3; r=2; c=2; row=[8, 5, 2]

L6 rotate_clockwise: for row in matrix:
matrix=[[7, 4, 1], [8, 5, 2], [3, 6, 9]]; n=3; r=2; c=2; row=[3, 6, 9]

L7 rotate_clockwise: row.reverse()
matrix=[[7, 4, 1], [8, 5, 2], [9, 6, 3]]; n=3; r=2; c=2; row=[9, 6, 3]
```

Result: [[7, 4, 1], [8, 5, 2], [9, 6, 3]]. Every coordinate follows the clockwise mapping.

```
rotate_clockwise([])

L2 rotate_clockwise: n = len(matrix)
matrix=[]; n=0
```

Result: []. The zero-size matrix requires no swaps.

Why the solution is correct

Each off-diagonal pair is exchanged once and diagonal cells stay fixed, giving a transpose. Reversing row positions transforms each copied coordinate (c,r) into (c,n-1-r), exactly the clockwise rotation formula.

Time and space

O(n^2) time and O(1) auxiliary space.

Common mistakes

Swapping both triangular halves undoes the transpose. Reversing columns rather than rows after transposition rotates in the other direction.

Reasoning card

State: After transposition, row r holds original column r. Row reversal then establishes the final clockwise coordinate mapping.

Invariant: Each transformation has a clear coordinate mapping.

Decision to rehearse: Transpose, then reverse each row; or perform four-way cycles.

Finish: Return the matrix

Transfer exercise

Derive counterclockwise and 180-degree rotations by composing reflections, and test four clockwise rotations as an identity.

124. Spiral Traversal

Foundation | Module 13

Problem

Return a rectangular matrix's values in clockwise spiral order, beginning at its top-left cell and initially moving right. Visit each cell once without changing the matrix. Empty input or zero columns returns [].

Examples

Example 1. The outer ring precedes the center.

```
spiral_order([[1,2,3],[4,5,6],[7,8,9]])
=> [1, 2, 3, 6, 9, 8, 7, 4, 5]
```

Example 2. A single column must not be traversed twice.

```
spiral_order([[1],[2],[3]])
=> [1, 2, 3]
```

Example 3. There is no perimeter.

```
spiral_order([])
=> []
```

Why this mechanism works

Peel one perimeter at a time. Four half-open boundaries describe the remaining rectangle. After consuming an edge, move its boundary inward before traversing the next edge. Recheck whether a bottom row or left column still exists so thin final layers are not visited twice.

State and boundaries

Unvisited cells lie in rows [top,bottom) and columns [left,right). output contains completed perimeter edges in traversal order.

Protocol

1. Traverse and remove the top row.

2. Traverse and remove the right column.

3. If rows remain, traverse and remove the bottom row backward.

4. If columns remain, traverse and remove the left column upward.

5. Repeat while the rectangle is nonempty.

Visualization

```
top row ->
^         |
|         v
<- bottom row
then shrink all four boundaries
```

Python implementation

```
def spiral_order(matrix):
    if not matrix or not matrix[0]:
        return []
    top, bottom = 0, len(matrix)
    left, right = 0, len(matrix[0])
    output = []
    while top < bottom and left < right:
        for c in range(left, right):
            output.append(matrix[top][c])
        top += 1
        for r in range(top, bottom):
            output.append(matrix[r][right - 1])
        right -= 1
        if top < bottom:
            bottom -= 1
            for c in range(right - 1, left - 1, -1):
                output.append(matrix[bottom][c])
        if left < right:
            for r in range(bottom - 1, top - 1, -1):
                output.append(matrix[r][left])
            left += 1
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
spiral_order([[1,2,3],[4,5,6],[7,8,9]])

L2 spiral_order: if not matrix or not matrix[0]:
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

L4 spiral_order: top, bottom = 0, len(matrix)
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=0; bottom=3

L5 spiral_order: left, right = 0, len(matrix[0])
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=0; bottom=3; left=0; right=3

L6 spiral_order: output = []
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=0; bottom=3; left=0; right=3; output=[]

L8 spiral_order: for c in range(left, right):
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=0; bottom=3; left=0; right=3; output=[]; c=0

L9 spiral_order: output.append(matrix[top][c])
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=0; bottom=3; left=0; right=3; output=[1]; c=0

L8 spiral_order: for c in range(left, right):
```

```
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=0; bottom=3; left=0; right=3; output=[1]; c=1

L9 spiral_order: output.append(matrix[top][c])
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=0; bottom=3; left=0; right=3; output=[1, 2]; c=1

... 16 intermediate state changes omitted ...

L8 spiral_order: for c in range(left, right):
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=1; bottom=2; left=1; right=2; output=[1, 2, 3,
    6, 9, 8, 7, 4]; c=1; r=1

L9 spiral_order: output.append(matrix[top][c])
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=1; bottom=2; left=1; right=2; output=[1, 2, 3,
    6, 9, 8, 7, 4, ...]; c=1; r=1

L10 spiral_order: top += 1
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=2; bottom=2; left=1; right=2; output=[1, 2, 3,
    6, 9, 8, 7, 4, ...]; c=1; r=1

L13 spiral_order: right -= 1
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; top=2; bottom=2; left=1; right=1; output=[1, 2, 3,
    6, 9, 8, 7, 4, ...]; c=1; r=1
```

Result: [1, 2, 3, 6, 9, 8, 7, 4, 5]. The outer ring precedes the center.

`spiral_order([[1],[2],[3]])`

```
L2 spiral_order: if not matrix or not matrix[0]:
matrix=[[1], [2], [3]]

L4 spiral_order: top, bottom = 0, len(matrix)
matrix=[[1], [2], [3]]; top=0; bottom=3

L5 spiral_order: left, right = 0, len(matrix[0])
matrix=[[1], [2], [3]]; top=0; bottom=3; left=0; right=1

L6 spiral_order: output = []
matrix=[[1], [2], [3]]; top=0; bottom=3; left=0; right=1; output=[]

L8 spiral_order: for c in range(left, right):
matrix=[[1], [2], [3]]; top=0; bottom=3; left=0; right=1; output=[]; c=0

L9 spiral_order: output.append(matrix[top][c])
matrix=[[1], [2], [3]]; top=0; bottom=3; left=0; right=1; output=[1]; c=0

L10 spiral_order: top += 1
matrix=[[1], [2], [3]]; top=1; bottom=3; left=0; right=1; output=[1]; c=0

L11 spiral_order: for r in range(top, bottom):
matrix=[[1], [2], [3]]; top=1; bottom=3; left=0; right=1; output=[1]; c=0; r=1

... 1 intermediate state changes omitted ...

L11 spiral_order: for r in range(top, bottom):
matrix=[[1], [2], [3]]; top=1; bottom=3; left=0; right=1; output=[1, 2]; c=0; r=2

L12 spiral_order: output.append(matrix[r][right - 1])
matrix=[[1], [2], [3]]; top=1; bottom=3; left=0; right=1; output=[1, 2, 3]; c=0; r=2

L13 spiral_order: right -= 1
matrix=[[1], [2], [3]]; top=1; bottom=3; left=0; right=0; output=[1, 2, 3]; c=0; r=2
```

```
L15 spiral_order: bottom -= 1
matrix=[[1], [2], [3]]; top=1; bottom=2; left=0; right=0; output=[1, 2, 3]; c=0; r=2
```

Result: [1, 2, 3]. A single column must not be traversed twice.

Why the solution is correct

Each edge traversal visits only the current unvisited rectangle, and the boundary update excludes that edge from subsequent traversals. The extra existence checks handle a final single row or column. Repeated peeling therefore visits every cell once in the required order.

Time and space

O(rows*cols) time and O(rows*cols) output space, with O(1) boundary state.

Common mistakes

Updating a boundary after rather than before the next edge can duplicate corner cells. Thin rectangles expose missing existence checks.

Reasoning card

State: Unvisited cells lie in rows [top,bottom) and columns [left,right). output contains completed perimeter edges in traversal order.

Invariant: Outside the boundary rectangle is already visited exactly once.

Decision to rehearse: Traverse one perimeter and shrink top/right/bottom/left boundaries.

Finish: Repeat while the rectangle is nonempty

Transfer exercise

Generate an n-by-n spiral matrix filled with 1 through n^2 using the same boundary protocol.

125. Set Matrix Zeroes

Bridge | Module 13

Problem

Modify a rectangular integer matrix in place so every row and column containing an original zero becomes entirely zero. New zeros created during the operation must not trigger additional rows or columns. Return the same matrix; empty input is unchanged.

Examples

Example 1. Only the original center row and column propagate.

```
set_zeroes([[1,1,1],[1,0,1],[1,1,1]])
=> [[1, 0, 1], [0, 0, 0], [1, 0, 1]]
```

Example 2. The two saved boundary flags are both needed.

```
set_zeroes([[0,1],[2,3]])
=> [[0, 0], [0, 3]]
```

Example 3. No markers are needed.

```
set_zeroes([])
=> []
```

Why this mechanism works

The first row and first column can store markers for the rest of the matrix, but their original zero status must be saved separately. Discover all markers before applying them. This separates observation of original state from propagation of new state.

State and boundaries

first_row and first_col remember original boundary-zero flags. Interior zeros mark matrix[r][0] and matrix[0][c].

Protocol

1. Save the first row and column flags.
2. Mark rows and columns from original interior zeros.
3. Zero interior cells indicated by markers.
4. Apply the saved first-row and first-column flags last.

Visualization

```
original center zero
mark its row at first column
mark its column at first row
apply markers only after discovery is complete
```

Python implementation

```
def set_zeroes(matrix):
    if not matrix or not matrix[0]:
        return matrix
    rows, cols = len(matrix), len(matrix[0])
    first_row = any(value == 0 for value in matrix[0])
    first_col = any(row[0] == 0 for row in matrix)
    for r in range(1, rows):
        for c in range(1, cols):
            if matrix[r][c] == 0:
                matrix[r][0] = matrix[0][c] = 0
    for r in range(1, rows):
        for c in range(1, cols):
            if matrix[r][0] == 0 or matrix[0][c] == 0:
                matrix[r][c] = 0
    if first_row:
        for c in range(cols):
            matrix[0][c] = 0
    if first_col:
        for r in range(rows):
            matrix[r][0] = 0
    return matrix
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
set_zeroes([[1,1,1],[1,0,1],[1,1,1]])

L2 set_zeroes: if not matrix or not matrix[0]:
matrix=[[1, 1, 1], [1, 0, 1], [1, 1, 1]]

L4 set_zeroes: rows, cols = len(matrix), len(matrix[0])
matrix=[[1, 1, 1], [1, 0, 1], [1, 1, 1]]; rows=3; cols=3

L5 <genexpr>: first_row = any(value == 0 for value in matrix[0])
.0=list_iterator; value=1

L5 <genexpr>: first_row = any(value == 0 for value in matrix[0])
.0=list_iterator; value=1

L5 <genexpr>: first_row = any(value == 0 for value in matrix[0])
.0=list_iterator; value=1

L5 <genexpr>: first_row = any(value == 0 for value in matrix[0])
.0=list_iterator; value=1

L5 set_zeroes: first_row = any(value == 0 for value in matrix[0])
matrix=[[1, 1, 1], [1, 0, 1], [1, 1, 1]]; rows=3; cols=3; first_row=False

L6 <genexpr>: first_col = any(row[0] == 0 for row in matrix)
```

```
.0=list_iterator; row=[1, 1, 1]

... 15 intermediate state changes omitted ...

L11 set_zeroes: for r in range(1, rows):
matrix=[[1, 0, 1], [0, 0, 0], [1, 1, 1]]; rows=3; cols=3; first_row=False; first_col=False;
    r=2; c=2

L12 set_zeroes: for c in range(1, cols):
matrix=[[1, 0, 1], [0, 0, 0], [1, 1, 1]]; rows=3; cols=3; first_row=False; first_col=False;
    r=2; c=1

L14 set_zeroes: matrix[r][c] = 0
matrix=[[1, 0, 1], [0, 0, 0], [1, 0, 1]]; rows=3; cols=3; first_row=False; first_col=False;
    r=2; c=1

L12 set_zeroes: for c in range(1, cols):
matrix=[[1, 0, 1], [0, 0, 0], [1, 0, 1]]; rows=3; cols=3; first_row=False; first_col=False;
    r=2; c=2
```

Result: [[1, 0, 1], [0, 0, 0], [1, 0, 1]]. Only the original center row and column propagate.

`set_zeroes([[0,1],[2,3]])`

```
L2 set_zeroes: if not matrix or not matrix[0]:
matrix=[[0, 1], [2, 3]]

L4 set_zeroes: rows, cols = len(matrix), len(matrix[0])
matrix=[[0, 1], [2, 3]]; rows=2; cols=2

L5 <genexpr>: first_row = any(value == 0 for value in matrix[0])
.0=list_iterator; value=0

L5 set_zeroes: first_row = any(value == 0 for value in matrix[0])
matrix=[[0, 1], [2, 3]]; rows=2; cols=2; first_row=True

L6 <genexpr>: first_col = any(row[0] == 0 for row in matrix)
.0=list_iterator; row=[0, 1]

L6 set_zeroes: first_col = any(row[0] == 0 for row in matrix)
matrix=[[0, 1], [2, 3]]; rows=2; cols=2; first_row=True; first_col=True

L7 set_zeroes: for r in range(1, rows):
matrix=[[0, 1], [2, 3]]; rows=2; cols=2; first_row=True; first_col=True; r=1

L8 set_zeroes: for c in range(1, cols):
matrix=[[0, 1], [2, 3]]; rows=2; cols=2; first_row=True; first_col=True; r=1; c=1

... 2 intermediate state changes omitted ...

L17 set_zeroes: matrix[0][c] = 0
matrix=[[0, 0], [2, 3]]; rows=2; cols=2; first_row=True; first_col=True; r=1; c=1

L19 set_zeroes: for r in range(rows):
matrix=[[0, 0], [2, 3]]; rows=2; cols=2; first_row=True; first_col=True; r=0; c=1

L19 set_zeroes: for r in range(rows):
matrix=[[0, 0], [2, 3]]; rows=2; cols=2; first_row=True; first_col=True; r=1; c=1

L20 set_zeroes: matrix[r][0] = 0
matrix=[[0, 0], [0, 3]]; rows=2; cols=2; first_row=True; first_col=True; r=1; c=1
```

Result: [[0, 0], [0, 3]]. The two saved boundary flags are both needed.

Why the solution is correct

Marker discovery reads only original interior cells and writes only boundary cells, so it cannot create new interior evidence. The markers therefore identify exactly the required rows and columns. Applying them after discovery, then restoring boundary behavior from saved flags, implements precisely the original-zero rule.

Time and space

O(rows*cols) time and O(1) auxiliary space.

Common mistakes

Zeroing immediately while scanning causes cascades from newly written zeros. Using matrix[0][0] alone cannot independently remember both boundary flags.

Reasoning card

State: first_row and first_col remember original boundary-zero flags. Interior zeros mark matrix[r][0] and matrix[0][c].

Invariant: Markers reflect only zeros from the original state.

Decision to rehearse: Record markers without letting newly written zeros create extra effects.

Finish: Apply the saved first-row and first-column flags last

Transfer exercise

Build a simpler O(rows+cols)-space reference and use it as an independent oracle for the marker implementation.

126. Diagonal Zigzag Traversal

Bridge | Module 13

Problem

Traverse a rectangular matrix by diagonals of constant r+c, alternating direction. Start at (0,0); the next diagonal visits (0,1) then (1,0). Return all values in that order without changing the matrix. Empty input returns [].

Examples

Example 1. The direction changes on each coordinate-sum diagonal.

```
diagonal_zigzag([[1,2,3],[4,5,6],[7,8,9]])
=> [1, 2, 4, 7, 5, 3, 6, 8, 9]
```

Example 2. Each diagonal contains one cell.

```
diagonal_zigzag([[1,2,3]])
=> [1, 2, 3]
```

Example 3. No diagonal exists.

```
diagonal_zigzag([])
=> []
```

Why this mechanism works

A diagonal is identified by its coordinate sum. Compute the legal row range for each sum and derive the column as sum-row. Collecting in increasing row order gives one direction; reverse every even-numbered diagonal to obtain the required zigzag.

State and boundaries

diagonal identifies r+c. The legal row interval clips that diagonal to the matrix bounds.

Protocol

1. Iterate coordinate sums from zero to rows+cols-2.
2. Collect cells with legal rows and column diagonal-row.
3. Reverse the collected values on even sums.
4. Append them to output.

Visualization

```
matrix 1 2 3 / 4 5 6 / 7 8 9
diagonals: [1], [2,4], [7,5,3], [6,8], [9]
```

Python implementation

```
def diagonal_zigzag(matrix):
    if not matrix or not matrix[0]:
        return []
    rows, cols = len(matrix), len(matrix[0])
    output = []
    for diagonal in range(rows + cols - 1):
        first = max(0, diagonal - cols + 1)
        last = min(rows - 1, diagonal)
        values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
        if diagonal % 2 == 0:
            values.reverse()
        output.extend(values)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
diagonal_zigzag([[1,2,3],[4,5,6],[7,8,9]])

L2 diagonal_zigzag: if not matrix or not matrix[0]:
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

L4 diagonal_zigzag: rows, cols = len(matrix), len(matrix[0])
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3

L5 diagonal_zigzag: output = []
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[]

L6 diagonal_zigzag: for diagonal in range(rows + cols - 1):
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[]; diagonal=0

L7 diagonal_zigzag: first = max(0, diagonal - cols + 1)
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[]; diagonal=0; first=0

L8 diagonal_zigzag: last = min(rows - 1, diagonal)
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[]; diagonal=0; first=0;
    last=0

L9 diagonal_zigzag: values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[]; diagonal=0; first=0;
    last=0; r=0

L9 diagonal_zigzag: values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[]; diagonal=0; first=0;
    last=0; values=[1]

... 22 intermediate state changes omitted ...

L7 diagonal_zigzag: first = max(0, diagonal - cols + 1)
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[1, 2, 4, 7, 5, 3, 6, 8];
    diagonal=4; first=2; last=2; values=[6, 8]

L9 diagonal_zigzag: values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[1, 2, 4, 7, 5, 3, 6, 8];
    diagonal=4; first=2; last=2; values=[6, 8]; r=2

L9 diagonal_zigzag: values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[1, 2, 4, 7, 5, 3, 6, 8];
```

```
    diagonal=4; first=2; last=2; values=[9]

L12 diagonal_zigzag: output.extend(values)
matrix=[[1, 2, 3], [4, 5, 6], [7, 8, 9]]; rows=3; cols=3; output=[1, 2, 4, 7, 5, 3, 6, 8,
    ...]; diagonal=4; first=2; last=2; values=[9]
```

Result: [1, 2, 4, 7, 5, 3, 6, 8, 9]. The direction changes on each coordinate-sum diagonal.

```
diagonal_zigzag([[1,2,3]])

L2 diagonal_zigzag: if not matrix or not matrix[0]:
matrix=[[1, 2, 3]]

L4 diagonal_zigzag: rows, cols = len(matrix), len(matrix[0])
matrix=[[1, 2, 3]]; rows=1; cols=3

L5 diagonal_zigzag: output = []
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[]

L6 diagonal_zigzag: for diagonal in range(rows + cols - 1):
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[]; diagonal=0

L7 diagonal_zigzag: first = max(0, diagonal - cols + 1)
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[]; diagonal=0; first=0

L8 diagonal_zigzag: last = min(rows - 1, diagonal)
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[]; diagonal=0; first=0; last=0

L9 diagonal_zigzag: values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[]; diagonal=0; first=0; last=0; r=0

L9 diagonal_zigzag: values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[]; diagonal=0; first=0; last=0; values=[1]

... 5 intermediate state changes omitted ...

L6 diagonal_zigzag: for diagonal in range(rows + cols - 1):
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[1, 2]; diagonal=2; first=0; last=0; values=[2]

L9 diagonal_zigzag: values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[1, 2]; diagonal=2; first=0; last=0; values=[2];
    r=0

L9 diagonal_zigzag: values = [matrix[r][diagonal - r] for r in range(first, last + 1)]
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[1, 2]; diagonal=2; first=0; last=0; values=[3]

L12 diagonal_zigzag: output.extend(values)
matrix=[[1, 2, 3]]; rows=1; cols=3; output=[1, 2, 3]; diagonal=2; first=0; last=0; values=[3]
```

Result: [1, 2, 3]. Each diagonal contains one cell.

Why the solution is correct

Every cell has one unique coordinate sum and appears once in that diagonal's legal row range. Alternating reversal supplies the required direction without changing membership. Concatenating diagonals in increasing sum gives the complete traversal.

Time and space

O(rows*cols) time; O(min(rows,cols)) temporary diagonal space plus O(rows*cols) output.

Common mistakes

Assuming every diagonal has the same length fails near boundaries. Reversing odd rather than even sums produces the opposite initial direction.

Reasoning card

State: diagonal identifies r+c. The legal row interval clips that diagonal to the matrix bounds.

Invariant: Each diagonal group is emitted in its prescribed order.

Decision to rehearse: Cells on one diagonal share row+column.

Finish: Append them to output

Transfer exercise

Emit each diagonal directly in its desired row direction instead of allocating a temporary list.

127. Game of Life Step

Bridge | Module 13

Problem

Compute one simultaneous Game of Life generation on a rectangular matrix of zeros and ones, treating outside cells as dead. A live cell survives with two or three live neighbors; a dead cell is born with exactly three. Use all eight neighbors, update in place, and return the matrix.

Examples

Example 1. A vertical blinker becomes horizontal simultaneously.

```
game_of_life([[0,1,0],[0,1,0],[0,1,0]])
=> [[0, 0, 0], [1, 1, 1], [0, 0, 0]]
```

Example 2. Each live cell has three live neighbors and survives.

```
game_of_life([[1,1],[1,1]])
=> [[1, 1], [1, 1]]
```

Example 3. There are no cells to update.

```
game_of_life([])
=> []
```

Why this mechanism works

Sequentially overwriting old values would change later neighbor counts. Store old and new state in separate bits of each cell: bit zero is the original value, bit one is the next generation. Every neighbor read masks the old bit, and a final pass shifts the new bit into place.

State and boundaries

cell & 1 always gives original state during discovery. cell's second bit records the independently computed next state.

Protocol

1. For each cell count eight-neighbor old bits.

2. Set bit one if the survival or birth rule holds.

3. After all cells are evaluated, shift every cell right by one.

Visualization

```
encoding old -> new:
0: dead->dead   1: live->dead
2: dead->live   3: live->live
final shift keeps the new bit
```

Python implementation

```
def game_of_life(board):
    rows = len(board)
    cols = len(board[0]) if rows else 0
    for r in range(rows):
        for c in range(cols):
            neighbors = 0
            for dr in (-1, 0, 1):
                for dc in (-1, 0, 1):
                    nr, nc = r + dr, c + dc
                    if (dr or dc) and 0 <= nr < rows and 0 <= nc < cols:
                        neighbors += board[nr][nc] & 1
            old = board[r][c] & 1
            if neighbors == 3 or (old == 1 and neighbors == 2):
                board[r][c] |= 2
    for r in range(rows):
        for c in range(cols):
            board[r][c] >>= 1
    return board
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
game_of_life([[0,1,0],[0,1,0],[0,1,0]])

L2 game_of_life: rows = len(board)
board=[[0, 1, 0], [0, 1, 0], [0, 1, 0]]; rows=3

L3 game_of_life: cols = len(board[0]) if rows else 0
board=[[0, 1, 0], [0, 1, 0], [0, 1, 0]]; rows=3; cols=3

L4 game_of_life: for r in range(rows):
board=[[0, 1, 0], [0, 1, 0], [0, 1, 0]]; rows=3; cols=3; r=0

L5 game_of_life: for c in range(cols):
board=[[0, 1, 0], [0, 1, 0], [0, 1, 0]]; rows=3; cols=3; r=0; c=0

L6 game_of_life: neighbors = 0
board=[[0, 1, 0], [0, 1, 0], [0, 1, 0]]; rows=3; cols=3; r=0; c=0; neighbors=0

L7 game_of_life: for dr in (-1, 0, 1):
board=[[0, 1, 0], [0, 1, 0], [0, 1, 0]]; rows=3; cols=3; r=0; c=0; neighbors=0; dr=-1

L8 game_of_life: for dc in (-1, 0, 1):
board=[[0, 1, 0], [0, 1, 0], [0, 1, 0]]; rows=3; cols=3; r=0; c=0; neighbors=0; dr=-1; dc=-1

L9 game_of_life: nr, nc = r + dr, c + dc
board=[[0, 1, 0], [0, 1, 0], [0, 1, 0]]; rows=3; cols=3; r=0; c=0; neighbors=0; dr=-1; dc=-1;
    nr=-1; nc=-1
```

```
... 245 intermediate state changes omitted ...

L16 game_of_life: for c in range(cols):
board=[[0, 0, 0], [1, 1, 1], [0, 1, 0]]; rows=3; cols=3; r=2; c=0; neighbors=2; dr=1; dc=1;
    nr=3; nc=3; old=0

L16 game_of_life: for c in range(cols):
board=[[0, 0, 0], [1, 1, 1], [0, 1, 0]]; rows=3; cols=3; r=2; c=1; neighbors=2; dr=1; dc=1;
    nr=3; nc=3; old=0

L17 game_of_life: board[r][c] >>= 1
board=[[0, 0, 0], [1, 1, 1], [0, 0, 0]]; rows=3; cols=3; r=2; c=1; neighbors=2; dr=1; dc=1;
    nr=3; nc=3; old=0

L16 game_of_life: for c in range(cols):
board=[[0, 0, 0], [1, 1, 1], [0, 0, 0]]; rows=3; cols=3; r=2; c=2; neighbors=2; dr=1; dc=1;
    nr=3; nc=3; old=0
```

Result: [[0, 0, 0], [1, 1, 1], [0, 0, 0]]. A vertical blinker becomes horizontal simultaneously.

`game_of_life([[1,1],[1,1]])`

```
L2 game_of_life: rows = len(board)
board=[[1, 1], [1, 1]]; rows=2

L3 game_of_life: cols = len(board[0]) if rows else 0
board=[[1, 1], [1, 1]]; rows=2; cols=2

L4 game_of_life: for r in range(rows):
board=[[1, 1], [1, 1]]; rows=2; cols=2; r=0

L5 game_of_life: for c in range(cols):
board=[[1, 1], [1, 1]]; rows=2; cols=2; r=0; c=0

L6 game_of_life: neighbors = 0
board=[[1, 1], [1, 1]]; rows=2; cols=2; r=0; c=0; neighbors=0

L7 game_of_life: for dr in (-1, 0, 1):
board=[[1, 1], [1, 1]]; rows=2; cols=2; r=0; c=0; neighbors=0; dr=-1

L8 game_of_life: for dc in (-1, 0, 1):
board=[[1, 1], [1, 1]]; rows=2; cols=2; r=0; c=0; neighbors=0; dr=-1; dc=-1

L9 game_of_life: nr, nc = r + dr, c + dc
board=[[1, 1], [1, 1]]; rows=2; cols=2; r=0; c=0; neighbors=0; dr=-1; dc=-1; nr=-1; nc=-1

... 111 intermediate state changes omitted ...

L16 game_of_life: for c in range(cols):
board=[[1, 1], [3, 3]]; rows=2; cols=2; r=1; c=0; neighbors=3; dr=1; dc=1; nr=2; nc=2; old=1

L17 game_of_life: board[r][c] >>= 1
board=[[1, 1], [1, 3]]; rows=2; cols=2; r=1; c=0; neighbors=3; dr=1; dc=1; nr=2; nc=2; old=1

L16 game_of_life: for c in range(cols):
board=[[1, 1], [1, 3]]; rows=2; cols=2; r=1; c=1; neighbors=3; dr=1; dc=1; nr=2; nc=2; old=1

L17 game_of_life: board[r][c] >>= 1
board=[[1, 1], [1, 1]]; rows=2; cols=2; r=1; c=1; neighbors=3; dr=1; dc=1; nr=2; nc=2; old=1
```

Result: [[1, 1], [1, 1]]. Each live cell has three live neighbors and survives.

Why the solution is correct

All transition decisions read only original low bits, so they match a simultaneous update. Writing a high bit cannot affect another cell's old-state count. The final shift exposes exactly the next state that was independently recorded for each cell.

Time and space

O(rows*cols) time and O(1) auxiliary space.

Common mistakes

Reading the whole encoded cell counts future state as current state. Using only four neighbors implements a different cellular rule.

Reasoning card

State: cell & 1 always gives original state during discovery. cell's second bit records the independently computed next state.

Invariant: Every output cell depends only on the unchanged original board.

Decision to rehearse: Read only old state while writing new state or encoded transitions.

Finish: After all cells are evaluated, shift every cell right by one

Transfer exercise

Implement a copy-based generation and verify two blinker steps restore the initial board.

128. Matrix Reshape

Interview | Module 13

Problem

Reshape a rectangular matrix to r rows and c columns while preserving row-major value order. r and c are nonnegative. If element counts differ, return the original matrix object unchanged. If they match, return a newly allocated matrix with the requested row count.

Examples

Example 1. Flat order is preserved.

```
reshape_matrix([[1,2],[3,4]], 1, 4)
=> [[1, 2, 3, 4]]
```

Example 2. An incompatible request returns the original object.

```
matrix=[[1,2],[3,4]]
reshape_matrix(matrix, 3, 2) is matrix
=> True
```

Example 3. Zero values can form two empty rows.

```
reshape_matrix([], 2, 0)
=> [[], []]
```

Why this mechanism works

Reshaping changes coordinates but not the flattened position of an element. A flat position k maps to new coordinates (k//c,k%c). Verify equal element counts before copying so every source position has a destination and no destination is missing.

State and boundaries

index counts consumed values in row-major order. output uses the requested dimensions; input remains unchanged.

Protocol

1. Compare old and new element counts.
2. Return the original object if incompatible.
3. Allocate independent output rows.
4. Walk original values and map each flat index to its new coordinates.
5. Return output.

Visualization

```
[[1,2],[3,4]] -> flat positions 0,1,2,3
reshape 1x4 -> [[1,2,3,4]]
reshape 3x2 rejected: element counts differ
```

Python implementation

```
def reshape_matrix(matrix, r, c):
    rows = len(matrix)
    cols = len(matrix[0]) if rows else 0
    if rows * cols != r * c:
        return matrix
    output = [[0] * c for _ in range(r)]
    index = 0
    for row in matrix:
        for value in row:
            output[index // c][index % c] = value
            index += 1
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
reshape_matrix([[1,2],[3,4]], 1, 4)

L2 reshape_matrix: rows = len(matrix)
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2

L3 reshape_matrix: cols = len(matrix[0]) if rows else 0
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2

L6 reshape_matrix: output = [[0] * c for _ in range(r)]
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; _=0

L6 reshape_matrix: output = [[0] * c for _ in range(r)]
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[0, 0, 0, 0]]

L7 reshape_matrix: index = 0
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[0, 0, 0, 0]]; index=0

L8 reshape_matrix: for row in matrix:
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[0, 0, 0, 0]]; index=0; row=[1, 2]

L9 reshape_matrix: for value in row:
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[0, 0, 0, 0]]; index=0; row=[1, 2];
     value=1

L10 reshape_matrix: output[index // c][index % c] = value
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[1, 0, 0, 0]]; index=0; row=[1, 2];
     value=1

... 7 intermediate state changes omitted ...

L11 reshape_matrix: index += 1
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[1, 2, 3, 0]]; index=3; row=[3, 4];
     value=3

L9 reshape_matrix: for value in row:
```

```
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[1, 2, 3, 0]]; index=3; row=[3, 4];
     value=4

L10 reshape_matrix: output[index // c][index % c] = value
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[1, 2, 3, 4]]; index=3; row=[3, 4];
     value=4

L11 reshape_matrix: index += 1
matrix=[[1, 2], [3, 4]]; r=1; c=4; rows=2; cols=2; output=[[1, 2, 3, 4]]; index=4; row=[3, 4];
     value=4
```

Result: [[1, 2, 3, 4]]. Flat order is preserved.

```
reshape_matrix(matrix, 3, 2) is matrix

L2 reshape_matrix: rows = len(matrix)
matrix=[[1, 2], [3, 4]]; r=3; c=2; rows=2

L3 reshape_matrix: cols = len(matrix[0]) if rows else 0
matrix=[[1, 2], [3, 4]]; r=3; c=2; rows=2; cols=2
```

Result: True. An incompatible request returns the original object.

Why the solution is correct

Equal element counts make the row-major index mapping a bijection between source and destination cells. Copying in that order therefore preserves the flattened sequence exactly. An incompatible request is rejected before mutation or copying.

Time and space

O(N) time and O(N+r) returned storage for N values and r output row objects; incompatible requests take O(1) after dimensions are read.

Common mistakes

Reshaping by rows alone may reorder values. Repeated-list allocation aliases output rows. A zero target column count is safe only when no value needs mapping.

Reasoning card

State: index counts consumed values in row-major order. output uses the requested dimensions; input remains unchanged.

Invariant: The first t output positions equal the first t input positions.

Decision to rehearse: Map one linear index to old and new coordinates.

Finish: Return output

Transfer exercise

Return a lazy view based on index arithmetic and discuss how later mutations of the original matrix should be reflected.

129. Sparse Matrix Multiplication

Interview | Module 13

Problem

Multiply compatible rectangular integer matrices A of shape m-by-k and B of shape k-by-n, returning a dense m-by-n result. Skip zero contributions. Empty A returns []; when the shared dimension and B are empty, return m empty rows. Inputs are unchanged.

Examples

Example 1. The zero coefficient skips the middle row of B.

```
sparse_multiply([[1,0,2]], [[1,2],[9,9],[3,4]])
=> [[7, 10]]
```

Example 2. All product contributions are zero.

```
sparse_multiply([[0,0]], [[1],[2]])
=> [[0]]
```

Example 3. There are no result rows.

```
sparse_multiply([], [])
=> []
```

Why this mechanism works

A[i,p] contributes only through nonzero entries of B's row p. Pre-index those entries, then skip zero A values entirely. This retains the ordinary matrix-product formula while avoiding multiplications known to contribute zero.

State and boundaries

nonzero[p] lists (column,value) entries in B row p. output[i][j] accumulates contributions from processed shared-dimension indices.

Protocol

1. Index nonzero entries of each B row.

2. Allocate the dense output.

3. For each nonzero A[i,p], visit only indexed nonzeros of B[p].

4. Add A[i,p]*B[p,j] to output[i,j].

Visualization

```
A row [1,0,2]
use B row 0 and twice B row 2
B row 1 contributes nothing for this output row
```

Python implementation

```
def sparse_multiply(a, b):
    rows = len(a)
    cols = len(b[0]) if b else 0
    nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
               for row in b]
    output = [[0] * cols for _ in range(rows)]
    for i, row in enumerate(a):
        for p, value in enumerate(row):
            if value != 0:
                for j, other in nonzero[p]:
                    output[i][j] += value * other
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
sparse_multiply([[1,0,2]], [[1,2],[9,9],[3,4]])

L2 sparse_multiply: rows = len(a)
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1

L3 sparse_multiply: cols = len(b[0]) if b else 0
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2

L5 sparse_multiply: for row in b]
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; row=[1, 2]

L4 sparse_multiply: nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; row=[1, 2]; j=0; value=1

L4 sparse_multiply: nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; row=[1, 2]; j=1; value=2

L4 sparse_multiply: nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; row=[1, 2]

L5 sparse_multiply: for row in b]
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; row=[9, 9]

L4 sparse_multiply: nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; row=[9, 9]; j=0; value=9

... 17 intermediate state changes omitted ...

L10 sparse_multiply: for j, other in nonzero[p]:
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; nonzero=[[(..., ...), (..., ...)],
    [(..., ...), (..., ...)], [(..., ...), (..., ...)]]; output=[[1, 2]]; row=[1, 0, 2]; i=0;
    value=2; p=2; j=0; other=3

L11 sparse_multiply: output[i][j] += value * other
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; nonzero=[[(..., ...), (..., ...)],
```

```
    [(..., ...), (..., ...)], [(..., ...), (..., ...)]]; output=[[7, 2]]; row=[1, 0, 2]; i=0;
    value=2; p=2; j=0; other=3

L10 sparse_multiply: for j, other in nonzero[p]:
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; nonzero=[[(..., ...), (..., ...)],
    [(..., ...), (..., ...)], [(..., ...), (..., ...)]]; output=[[7, 2]]; row=[1, 0, 2]; i=0;
    value=2; p=2; j=1; other=4

L11 sparse_multiply: output[i][j] += value * other
a=[[1, 0, 2]]; b=[[1, 2], [9, 9], [3, 4]]; rows=1; cols=2; nonzero=[[(..., ...), (..., ...)],
    [(..., ...), (..., ...)], [(..., ...), (..., ...)]]; output=[[7, 10]]; row=[1, 0, 2]; i=0;
     value=2; p=2; j=1; other=4
```

Result: [[7, 10]]. The zero coefficient skips the middle row of B.

`sparse_multiply([[0,0]], [[1],[2]])`

```
L2 sparse_multiply: rows = len(a)
a=[[0, 0]]; b=[[1], [2]]; rows=1

L3 sparse_multiply: cols = len(b[0]) if b else 0
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1

L5 sparse_multiply: for row in b]
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; row=[1]

L4 sparse_multiply: nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; row=[1]; j=0; value=1

L4 sparse_multiply: nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; row=[1]

L5 sparse_multiply: for row in b]
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; row=[2]

L4 sparse_multiply: nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; row=[2]; j=0; value=2

L4 sparse_multiply: nonzero = [[(j, value) for j, value in enumerate(row) if value != 0]
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; row=[2]

... 2 intermediate state changes omitted ...

L6 sparse_multiply: output = [[0] * cols for _ in range(rows)]
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; nonzero=[[(..., ...)], [(..., ...)]]; output=[[0]]

L7 sparse_multiply: for i, row in enumerate(a):
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; nonzero=[[(..., ...)], [(..., ...)]]; output=[[0]];
    row=[0, 0]; i=0

L8 sparse_multiply: for p, value in enumerate(row):
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; nonzero=[[(..., ...)], [(..., ...)]]; output=[[0]];
    row=[0, 0]; i=0; value=0; p=0

L8 sparse_multiply: for p, value in enumerate(row):
a=[[0, 0]]; b=[[1], [2]]; rows=1; cols=1; nonzero=[[(..., ...)], [(..., ...)]]; output=[[0]];
    row=[0, 0]; i=0; value=0; p=1
```

Result: [[0]]. All product contributions are zero.

Why the solution is correct

The product definition sums A[i,p]*B[p,j] over every shared index p. Skipped terms have a zero factor and contribute nothing; all nonzero-factor terms are added exactly once to the correct output coordinate. Thus sparsity changes work, not the result.

Time and space

O(kn+mk+mn+W) time, where W is the number of visited nonzero product pairs; O(nnz(B)+mn) space. Worst-case time remains O(mkn) for dense inputs.

Common mistakes

Skipping an entire output column because one A entry is zero loses other contributions. Dense output allocation still costs O(mn), even with sparse inputs.

Reasoning card

State: nonzero[p] lists (column,value) entries in B row p. output[i][j] accumulates contributions from processed shared-dimension indices.

Invariant: Accumulated output entries include all processed inner-dimension contributions.

Decision to rehearse: Iterate only meaningful nonzero products.

Finish: Add A[i,p]*B[p,j] to output[i,j]

Transfer exercise

Represent the output sparsely too, and distinguish multiplication work from the cost of materializing a dense result.

130. Validate Sudoku Board

Interview | Module 13

Problem

Check a 9-by-9 Sudoku board containing characters '1' through '9' or '.' for empty cells. Return whether no filled digit repeats within any row, column or 3-by-3 box. This checks local consistency only, not whether the board can be completed.

Examples

Example 1. One fully distinct row is locally consistent with empty remaining rows.

```
valid_sudoku(["123456789"] + ["........."] * 8)
=> True
```

Example 2. A row contains two copies of one.

```
valid_sudoku(["11......."] + ["........."] * 8)
=> False
```

Example 3. Different rows and columns can still repeat a digit inside one box.

```
valid_sudoku(["1........", ".1......."] + ["........."] * 7)
=> False
```

Why this mechanism works

Every filled cell belongs to three independently constrained groups. Maintain a set for each row, column and box. The box identifier is (r//3)*3+c//3, which maps the nine boxes to distinct indices.

State and boundaries

row_sets, column_sets and boxes contain digits already encountered in their respective groups.

Protocol

1. Skip empty cells.
2. Compute the cell's box index.
3. Reject if the digit is already in any of its three groups.
4. Otherwise add it to all three.
5. Accept after every filled cell passes.

Visualization

```
box index grid:
0 0 0 | 1 1 1 | 2 2 2
3 3 3 | 4 4 4 | 5 5 5
6 6 6 | 7 7 7 | 8 8 8
(each displayed band represents three rows)
```

Python implementation

```
def valid_sudoku(board):
    row_sets = [set() for _ in range(9)]
    column_sets = [set() for _ in range(9)]
    boxes = [set() for _ in range(9)]
    for r in range(9):
        for c in range(9):
            digit = board[r][c]
            if digit == '.':
                continue
            box = (r // 3) * 3 + c // 3
            if digit in row_sets[r] or digit in column_sets[c] or digit in boxes[box]:
                return False
            row_sets[r].add(digit)
            column_sets[c].add(digit)
            boxes[box].add(digit)
    return True
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
valid_sudoku(["123456789"] + ["........."] * 8)

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=0

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=1

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=2

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=3

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=4

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=5

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['123456789', '.........', '.........', '.........', '.........', '.........',
```

```
    '.........', ...; _=6

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=7

... 147 intermediate state changes omitted ...

L6 valid_sudoku: for c in range(9):
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; row_sets=[{'1', '2', '3', '4', '5', '6', '7', '8', ...}, set(), set(),
    set(), set(), set(), set(), se...; column_sets=[{'1'}, {'2'}, {'3'}, {'4'}, {'5'}, {'6'},
    {'7'}, {'8'}, ...]; boxes=[{'1', '2', '3'}, {'4', '5', '6'}, {'7', '8', '9'}, set(),
    set(), set(), set(), set(), ...]; r=8; c=5; digit='.'; box=2

L6 valid_sudoku: for c in range(9):
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; row_sets=[{'1', '2', '3', '4', '5', '6', '7', '8', ...}, set(), set(),
    set(), set(), set(), set(), se...; column_sets=[{'1'}, {'2'}, {'3'}, {'4'}, {'5'}, {'6'},
    {'7'}, {'8'}, ...]; boxes=[{'1', '2', '3'}, {'4', '5', '6'}, {'7', '8', '9'}, set(),
    set(), set(), set(), set(), ...]; r=8; c=6; digit='.'; box=2

L6 valid_sudoku: for c in range(9):
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; row_sets=[{'1', '2', '3', '4', '5', '6', '7', '8', ...}, set(), set(),
    set(), set(), set(), set(), se...; column_sets=[{'1'}, {'2'}, {'3'}, {'4'}, {'5'}, {'6'},
    {'7'}, {'8'}, ...]; boxes=[{'1', '2', '3'}, {'4', '5', '6'}, {'7', '8', '9'}, set(),
    set(), set(), set(), set(), ...]; r=8; c=7; digit='.'; box=2

L6 valid_sudoku: for c in range(9):
board=['123456789', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; row_sets=[{'1', '2', '3', '4', '5', '6', '7', '8', ...}, set(), set(),
    set(), set(), set(), set(), se...; column_sets=[{'1'}, {'2'}, {'3'}, {'4'}, {'5'}, {'6'},
    {'7'}, {'8'}, ...]; boxes=[{'1', '2', '3'}, {'4', '5', '6'}, {'7', '8', '9'}, set(),
    set(), set(), set(), set(), ...]; r=8; c=8; digit='.'; box=2
```

Result: True. One fully distinct row is locally consistent with empty remaining rows.

```
valid_sudoku(["11......."] + ["........."] * 8)

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=0

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=1

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=2

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=3

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=4

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=5
```

```
L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=6

L2 valid_sudoku: row_sets = [set() for _ in range(9)]
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; _=7

... 26 intermediate state changes omitted ...

L13 valid_sudoku: row_sets[r].add(digit)
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; row_sets=[{'1'}, set(), set(), set(), set(), set(), set(), set(), ...];
    column_sets=[set(), set(), set(), set(), set(), set(), set(), set(), ...]; boxes=[set(),
    set(), set(), set(), set(), set(), set(), set(), ...]; r=0; c=0; digit='1'; box=0

L14 valid_sudoku: column_sets[c].add(digit)
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; row_sets=[{'1'}, set(), set(), set(), set(), set(), set(), set(), ...];
    column_sets=[{'1'}, set(), set(), set(), set(), set(), set(), set(), ...]; boxes=[set(),
    set(), set(), set(), set(), set(), set(), set(), ...]; r=0; c=0; digit='1'; box=0

L15 valid_sudoku: boxes[box].add(digit)
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; row_sets=[{'1'}, set(), set(), set(), set(), set(), set(), set(), ...];
    column_sets=[{'1'}, set(), set(), set(), set(), set(), set(), set(), ...]; boxes=[{'1'},
    set(), set(), set(), set(), set(), set(), set(), ...]; r=0; c=0; digit='1'; box=0

L6 valid_sudoku: for c in range(9):
board=['11.......', '.........', '.........', '.........', '.........', '.........',
    '.........', ...; row_sets=[{'1'}, set(), set(), set(), set(), set(), set(), set(), ...];
    column_sets=[{'1'}, set(), set(), set(), set(), set(), set(), set(), ...]; boxes=[{'1'},
    set(), set(), set(), set(), set(), set(), set(), ...]; r=0; c=1; digit='1'; box=0
```

Result: False. A row contains two copies of one.

Why the solution is correct

A duplicate is rejected when its later occurrence is visited in the shared row, column or box. Conversely, if every membership check passes, no group contains two occurrences of any digit. The algorithm therefore checks exactly the three stated local constraints.

Time and space

O(81) time and bounded storage for a standard 9-by-9 board; the dimensions are fixed by the contract.

Common mistakes

Using r//3+c//3 merges distinct boxes. A locally valid partial board is not automatically guaranteed to have a full solution.

Reasoning card

State: row_sets, column_sets and boxes contain digits already encountered in their respective groups.

Invariant: No processed unit contains a repeated digit.

Decision to rehearse: Create three independent membership identities per filled cell.

Finish: Accept after every filled cell passes

Transfer exercise

Build a solver by maintaining these same constraints during backtracking, restoring every set when a candidate is undone.

Module 14
Grid DFS, BFS, and Multi-Source Search

Treat cells as graph nodes, directions as edges, and marking time as part of correctness.

Grid traversal is graph traversal with implicit neighbors. Model allowed directions explicitly, then choose DFS for region consumption or BFS for shortest unit-cost distance. Multi-source BFS begins with an entire initial frontier. Mark at discovery to prevent duplicate work; reserve path-local visitation for later backtracking problems.

The ten-step progression

131. Flood Fill with DFS

132. Flood Fill with BFS

133. Count Islands

134. Maximum Island Area

135. Surrounded Regions

136. Rotting Oranges

137. Shortest Binary-Matrix Path

138. Nearest-Zero Distances

139. Walls and Gates

140. Pacific Atlantic Water Flow

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

131. Flood Fill with DFS

Foundation | Module 14

Problem

Recolor the four-directionally connected region containing (sr,sc) in a rectangular integer image. Only cells equal to the starting cell's original color belong to the region. Mutate and return the image. Coordinates are valid for nonempty input; empty input is returned unchanged. Use a LIFO worklist for DFS.

Examples

Example 1. The diagonally separated one is outside the four-neighbor component.

```
flood_fill_dfs([[1,1,0],[1,0,1]], 0, 0, 2)
=> [[2, 2, 0], [2, 0, 1]]
```

Example 2. The unchanged-color guard avoids meaningless revisits.

```
flood_fill_dfs([[1,1]], 0, 0, 1)
=> [[1, 1]]
```

Example 3. Empty input is returned before coordinates are accessed.

```
flood_fill_dfs([], 0, 0, 2)
=> []
```

Why this mechanism works

Color equality defines which edges may be traversed. Mark a cell by recoloring it when discovered, before pushing it, so several neighbors cannot schedule it repeatedly. If the replacement equals the original color, this marking method cannot distinguish visited cells; return immediately because the desired output already exists.

State and boundaries

stack holds discovered recolored cells awaiting neighbor exploration. original is fixed from the starting cell.

Protocol

1. Handle empty input or unchanged color.
2. Recolor and push the start.
3. Pop one cell.
4. Recolor and push each in-bounds original-color neighbor.
5. Return the image after the stack empties.

Visualization

```
same-color component: explore through up/down/left/right
LIFO: newest discovered cell is expanded next
recolor before push = discovery mark
```

Python implementation

```
def flood_fill_dfs(image, sr, sc, color):
    if not image or not image[0]:
        return image
    original = image[sr][sc]
    if original == color:
        return image
    rows, cols = len(image), len(image[0])
    stack = [(sr, sc)]
    image[sr][sc] = color
    while stack:
        r, c = stack.pop()
        for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and image[nr][nc] == original:
                image[nr][nc] = color
                stack.append((nr, nc))
    return image
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
flood_fill_dfs([[1,1,0],[1,0,1]], 0, 0, 2)

L2 flood_fill_dfs: if not image or not image[0]:
image=[[1, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2

L4 flood_fill_dfs: original = image[sr][sc]
image=[[1, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1

L7 flood_fill_dfs: rows, cols = len(image), len(image[0])
image=[[1, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3

L8 flood_fill_dfs: stack = [(sr, sc)]
image=[[1, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[(0, 0)]

L9 flood_fill_dfs: image[sr][sc] = color
image=[[2, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[(0, 0)]

L11 flood_fill_dfs: r, c = stack.pop()
image=[[2, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[]; r=0;
    c=0

L12 flood_fill_dfs: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
image=[[2, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[]; r=0;
    c=0; dr=1; dc=0

L13 flood_fill_dfs: nr, nc = r + dr, c + dc
image=[[2, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[]; r=0;
    c=0; dr=1; dc=0; nr=1; nc=0
```

```
... 24 intermediate state changes omitted ...

L12 flood_fill_dfs: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
image=[[2, 2, 0], [2, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[]; r=1;
    c=0; dr=0; dc=1; nr=0; nc=0

L13 flood_fill_dfs: nr, nc = r + dr, c + dc
image=[[2, 2, 0], [2, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[]; r=1;
    c=0; dr=0; dc=1; nr=1; nc=1

L12 flood_fill_dfs: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
image=[[2, 2, 0], [2, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[]; r=1;
    c=0; dr=0; dc=-1; nr=1; nc=1

L13 flood_fill_dfs: nr, nc = r + dr, c + dc
image=[[2, 2, 0], [2, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3; stack=[]; r=1;
    c=0; dr=0; dc=-1; nr=1; nc=-1
```

Result: [[2, 2, 0], [2, 0, 1]]. The diagonally separated one is outside the four-neighbor component.

```
flood_fill_dfs([[1,1]], 0, 0, 1)

L2 flood_fill_dfs: if not image or not image[0]:
image=[[1, 1]]; sr=0; sc=0; color=1

L4 flood_fill_dfs: original = image[sr][sc]
image=[[1, 1]]; sr=0; sc=0; color=1; original=1
```

Result: [[1, 1]]. The unchanged-color guard avoids meaningless revisits.

Why the solution is correct

Every discovered neighbor is connected to the start through original-color cells, so no outside cell is recolored. Conversely, every cell in the component is eventually reached along such a path. Recoloring at discovery schedules each cell only once.

Time and space

O(R*C) worst-case time and O(R*C) worklist space; only the reached component is explored after initialization.

Common mistakes

Marking only when popping permits duplicate pending cells. Omitting the unchanged-color guard can create repeated scheduling indefinitely.

Reasoning card

State: stack holds discovered recolored cells awaiting neighbor exploration. original is fixed from the starting cell.

Invariant: Every successful call processes one previously unvisited eligible cell.

Decision to rehearse: Reject bounds/wrong colour; mark before recursive expansion.

Finish: Return the image after the stack empties

Transfer exercise

Use a separate visited set instead of recoloring as the discovery mark, and compare state and mutation costs.

132. Flood Fill with BFS

Foundation | Module 14

Problem

Perform the same four-neighbor region recoloring as kata 131, but use a FIFO queue for breadth-first exploration. Mutate and return the image. The unchanged-color and empty-input policies remain the same.

Examples

Example 1. Traversal order changes but the component does not.

```
flood_fill_bfs([[1,1,0],[1,0,1]], 0, 0, 2)
=> [[2, 2, 0], [2, 0, 1]]
```

Example 2. The image already has the requested color.

```
flood_fill_bfs([[3]], 0, 0, 3)
=> [[3]]
```

Example 3. There is no starting cell to explore.

```
flood_fill_bfs([], 0, 0, 9)
=> []
```

Why this mechanism works

Changing the worklist changes exploration order, not component membership. FIFO expands cells in nondecreasing hop distance from the start, while LIFO follows a deeper frontier first. Marking at enqueue time remains essential in either traversal.

State and boundaries

queue contains discovered recolored cells awaiting expansion in discovery order. original is the target component color.

Protocol

1. Handle empty or unchanged-color input.
2. Recolor and enqueue the start.
3. Remove the oldest queued cell.
4. Discover original-color neighbors by recoloring before enqueue.
5. Continue until the queue empties.

Visualization

```
start -> distance 1 neighbors -> distance 2 neighbors
FIFO preserves this layer order
final recolored component matches DFS
```

Python implementation

```
from collections import deque

def flood_fill_bfs(image, sr, sc, color):
    if not image or not image[0]:
        return image
    original = image[sr][sc]
    if original == color:
        return image
    rows, cols = len(image), len(image[0])
    queue = deque([(sr, sc)])
    image[sr][sc] = color
    while queue:
        r, c = queue.popleft()
        for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and image[nr][nc] == original:
                image[nr][nc] = color
                queue.append((nr, nc))
    return image
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
flood_fill_bfs([[1,1,0],[1,0,1]], 0, 0, 2)

L4 flood_fill_bfs: if not image or not image[0]:
image=[[1, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2

L6 flood_fill_bfs: original = image[sr][sc]
image=[[1, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1

L9 flood_fill_bfs: rows, cols = len(image), len(image[0])
image=[[1, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3

L10 flood_fill_bfs: queue = deque([(sr, sc)])
image=[[1, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([(0, 0)])

L11 flood_fill_bfs: image[sr][sc] = color
image=[[2, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([(0, 0)])

L13 flood_fill_bfs: r, c = queue.popleft()
image=[[2, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([]); r=0; c=0

L14 flood_fill_bfs: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
image=[[2, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([]); r=0; c=0; dr=1; dc=0
```

```
L15 flood_fill_bfs: nr, nc = r + dr, c + dc
image=[[2, 1, 0], [1, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([]); r=0; c=0; dr=1; dc=0; nr=1; nc=0

... 24 intermediate state changes omitted ...

L14 flood_fill_bfs: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
image=[[2, 2, 0], [2, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([]); r=0; c=1; dr=0; dc=1; nr=-1; nc=1

L15 flood_fill_bfs: nr, nc = r + dr, c + dc
image=[[2, 2, 0], [2, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([]); r=0; c=1; dr=0; dc=1; nr=0; nc=2

L14 flood_fill_bfs: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
image=[[2, 2, 0], [2, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([]); r=0; c=1; dr=0; dc=-1; nr=0; nc=2

L15 flood_fill_bfs: nr, nc = r + dr, c + dc
image=[[2, 2, 0], [2, 0, 1]]; sr=0; sc=0; color=2; original=1; rows=2; cols=3;
    queue=deque([]); r=0; c=1; dr=0; dc=-1; nr=0; nc=0
```

Result: [[2, 2, 0], [2, 0, 1]]. Traversal order changes but the component does not.

```
flood_fill_bfs([[3]], 0, 0, 3)

L4 flood_fill_bfs: if not image or not image[0]:
image=[[3]]; sr=0; sc=0; color=3

L6 flood_fill_bfs: original = image[sr][sc]
image=[[3]]; sr=0; sc=0; color=3; original=3
```

Result: [[3]]. The image already has the requested color.

Why the solution is correct

The same component reachability argument as DFS proves that exactly the connected original-color region is recolored. FIFO order additionally ensures each discovery layer is expanded before the next layer, which will matter when distance becomes the output.

Time and space

O(R*C) worst-case time and O(R*C) queue space.

Common mistakes

Using list.pop(0) introduces shifting cost. Switching to a queue does not justify delaying the discovery mark until removal.

Reasoning card

State: queue contains discovered recolored cells awaiting expansion in discovery order. original is the target component color.

Invariant: Every queued cell is valid, reachable, and scheduled once.

Decision to rehearse: Mark when enqueueing, not when dequeuing.

Finish: Continue until the queue empties

Transfer exercise

Return the number of BFS layers needed to reach the farthest cell in the recolored component.

133. Count Islands

Foundation | Module 14

Problem

Count four-neighbor connected components of land in a rectangular binary matrix, where 1 is land and 0 is water. Leave the grid unchanged. Diagonal contact does not join islands; empty input returns zero.

Examples

Example 1. The left vertical pair is one island; two singleton islands remain.

```
count_islands([[1,0,1],[1,0,0],[0,0,1]])
=> 3
```

Example 2. Diagonal contact does not connect land.

```
count_islands([[1,0],[0,1]])
=> 2
```

Example 3. There are no land seeds.

```
count_islands([])
=> 0
```

Why this mechanism works

A component traversal answers reachability from one seed. To count all components, scan every cell and start a new traversal only from unvisited land. One new seed contributes one island, and its traversal marks the entire component so no later cell counts it again.

State and boundaries

seen contains discovered land cells. count increases once per unvisited land seed, not once per visited cell.

Protocol

1. Scan cells in row-major order.
2. On unvisited land increment count and mark the seed.
3. Explore its four-neighbor land component with a stack.
4. Mark neighbors before pushing.
5. Continue the outer scan.

Visualization

```
1 0 1
1 0 0
0 0 1
three independent DFS seeds -> three islands
```

Python implementation

```
def count_islands(grid):
    rows = len(grid)
    cols = len(grid[0]) if rows else 0
    seen = set()
    count = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] != 1 or (r, c) in seen:
                continue
            count += 1
            seen.add((r, c))
            stack = [(r, c)]
            while stack:
                x, y = stack.pop()
                for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
                    nx, ny = x + dx, y + dy
                    if (0 <= nx < rows and 0 <= ny < cols
                            and grid[nx][ny] == 1 and (nx, ny) not in seen):
                        seen.add((nx, ny))
                        stack.append((nx, ny))
    return count
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
count_islands([[1,0,1],[1,0,0],[0,0,1]])

L2 count_islands: rows = len(grid)
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3

L3 count_islands: cols = len(grid[0]) if rows else 0
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3

L4 count_islands: seen = set()
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen=set()

L5 count_islands: count = 0
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen=set(); count=0

L6 count_islands: for r in range(rows):
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen=set(); count=0; r=0

L7 count_islands: for c in range(cols):
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen=set(); count=0; r=0; c=0

L10 count_islands: count += 1
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen=set(); count=1; r=0; c=0

L11 count_islands: seen.add((r, c))
```

```
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen={(0, 0)}; count=1; r=0; c=0

... 51 intermediate state changes omitted ...

L15 count_islands: for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen={(0, 0), (0, 2), (1, 0), (2, 2)};
     count=3; r=2; c=2; stack=[]; x=2; y=2; dx=0; dy=1; nx=1; ny=2

L16 count_islands: nx, ny = x + dx, y + dy
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen={(0, 0), (0, 2), (1, 0), (2, 2)};
     count=3; r=2; c=2; stack=[]; x=2; y=2; dx=0; dy=1; nx=2; ny=3

L15 count_islands: for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen={(0, 0), (0, 2), (1, 0), (2, 2)};
     count=3; r=2; c=2; stack=[]; x=2; y=2; dx=0; dy=-1; nx=2; ny=3

L16 count_islands: nx, ny = x + dx, y + dy
grid=[[1, 0, 1], [1, 0, 0], [0, 0, 1]]; rows=3; cols=3; seen={(0, 0), (0, 2), (1, 0), (2, 2)};
     count=3; r=2; c=2; stack=[]; x=2; y=2; dx=0; dy=-1; nx=2; ny=1
```

Result: 3. The left vertical pair is one island; two singleton islands remain.

```
count_islands([[1,0],[0,1]])

L2 count_islands: rows = len(grid)
grid=[[1, 0], [0, 1]]; rows=2

L3 count_islands: cols = len(grid[0]) if rows else 0
grid=[[1, 0], [0, 1]]; rows=2; cols=2

L4 count_islands: seen = set()
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen=set()

L5 count_islands: count = 0
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen=set(); count=0

L6 count_islands: for r in range(rows):
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen=set(); count=0; r=0

L7 count_islands: for c in range(cols):
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen=set(); count=0; r=0; c=0

L10 count_islands: count += 1
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen=set(); count=1; r=0; c=0

L11 count_islands: seen.add((r, c))
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen={(0, 0)}; count=1; r=0; c=0

... 22 intermediate state changes omitted ...

L15 count_islands: for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen={(0, 0), (1, 1)}; count=2; r=1; c=1; stack=[];
    x=1; y=1; dx=0; dy=1; nx=0; ny=1

L16 count_islands: nx, ny = x + dx, y + dy
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen={(0, 0), (1, 1)}; count=2; r=1; c=1; stack=[];
    x=1; y=1; dx=0; dy=1; nx=1; ny=2

L15 count_islands: for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen={(0, 0), (1, 1)}; count=2; r=1; c=1; stack=[];
    x=1; y=1; dx=0; dy=-1; nx=1; ny=2
```

```
L16 count_islands: nx, ny = x + dx, y + dy
grid=[[1, 0], [0, 1]]; rows=2; cols=2; seen={(0, 0), (1, 1)}; count=2; r=1; c=1; stack=[];
    x=1; y=1; dx=0; dy=-1; nx=1; ny=0
```

Result: 2. Diagonal contact does not connect land.

Why the solution is correct

Each traversal stays inside one island and reaches all its cells. The first scanned cell of each island starts exactly one traversal; all remaining cells are already marked when scanned. Thus the number of seeds equals the number of components.

Time and space

O(R*C) time and O(R*C) visited/worklist space.

Common mistakes

Counting each DFS visit counts land area rather than islands. Eight-neighbor movement merges diagonal islands contrary to the contract.

Reasoning card

State: seen contains discovered land cells. count increases once per unvisited land seed, not once per visited cell.

Invariant: Unvisited land starts exactly one new island.

Decision to rehearse: Outer scan discovers components; inner traversal consumes one.

Finish: Continue the outer scan

Transfer exercise

Record each component's cells or bounding box instead of only counting its seed.

134. Maximum Island Area

Foundation | Module 14

Problem

Return the maximum number of cells in any four-neighbor land component of a rectangular binary grid. Land is 1, water is 0, and the grid must remain unchanged. Return zero when no land exists.

Examples

Example 1. The top component has three cells.

```
maximum_island_area([[1,1,0],[0,1,0],[1,0,1]])
=> 3
```

Example 2. No land starts a traversal.

```
maximum_island_area([[0,0]])
=> 0
```

Example 3. An empty grid has maximum area zero.

```
maximum_island_area([])
=> 0
```

Why this mechanism works

Reuse component traversal but change the aggregation: count each cell visited within the current island, then compare that completed component area with a global maximum. A fresh area counter belongs to each seed, while the visited set belongs to the whole scan.

State and boundaries

area counts popped cells in the active island. best summarizes completed islands. seen prevents overlap between traversals.

Protocol

1. Find an unvisited land seed.
2. Start area at zero and mark the seed.
3. Traverse its component, incrementing area once per cell.
4. Compare the completed area with best.
5. Return best after all seeds.

Visualization

```
component areas: 2, 1, 4
local area resets at each seed
global best: 2 -> 2 -> 4
```

Python implementation

```
def maximum_island_area(grid):
    rows = len(grid)
    cols = len(grid[0]) if rows else 0
    seen = set()
    best = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] != 1 or (r, c) in seen:
                continue
            area = 0
            stack = [(r, c)]
            seen.add((r, c))
            while stack:
                x, y = stack.pop()
                area += 1
                for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
                    nx, ny = x + dx, y + dy
                    if (0 <= nx < rows and 0 <= ny < cols
                            and grid[nx][ny] == 1 and (nx, ny) not in seen):
                        seen.add((nx, ny))
                        stack.append((nx, ny))
            best = max(best, area)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
maximum_island_area([[1,1,0],[0,1,0],[1,0,1]])

L2 maximum_island_area: rows = len(grid)
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3

L3 maximum_island_area: cols = len(grid[0]) if rows else 0
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3

L4 maximum_island_area: seen = set()
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen=set()

L5 maximum_island_area: best = 0
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen=set(); best=0

L6 maximum_island_area: for r in range(rows):
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen=set(); best=0; r=0

L7 maximum_island_area: for c in range(cols):
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen=set(); best=0; r=0; c=0

L10 maximum_island_area: area = 0
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen=set(); best=0; r=0; c=0; area=0
```

```
L11 maximum_island_area: stack = [(r, c)]
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen=set(); best=0; r=0; c=0; area=0;
    stack=[(0, 0)]

... 68 intermediate state changes omitted ...

L16 maximum_island_area: for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen={(0, 0), (0, 1), (1, 1), (2, 0),
    (2, 2)}; best=3; r=2; c=2; area=1; stack=[]; x=2; y=2; dx=0; dy=1; nx=1; ny=2

L17 maximum_island_area: nx, ny = x + dx, y + dy
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen={(0, 0), (0, 1), (1, 1), (2, 0),
    (2, 2)}; best=3; r=2; c=2; area=1; stack=[]; x=2; y=2; dx=0; dy=1; nx=2; ny=3

L16 maximum_island_area: for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen={(0, 0), (0, 1), (1, 1), (2, 0),
    (2, 2)}; best=3; r=2; c=2; area=1; stack=[]; x=2; y=2; dx=0; dy=-1; nx=2; ny=3

L17 maximum_island_area: nx, ny = x + dx, y + dy
grid=[[1, 1, 0], [0, 1, 0], [1, 0, 1]]; rows=3; cols=3; seen={(0, 0), (0, 1), (1, 1), (2, 0),
    (2, 2)}; best=3; r=2; c=2; area=1; stack=[]; x=2; y=2; dx=0; dy=-1; nx=2; ny=1
```

Result: 3. The top component has three cells.

```
maximum_island_area([[0,0]])

L2 maximum_island_area: rows = len(grid)
grid=[[0, 0]]; rows=1

L3 maximum_island_area: cols = len(grid[0]) if rows else 0
grid=[[0, 0]]; rows=1; cols=2

L4 maximum_island_area: seen = set()
grid=[[0, 0]]; rows=1; cols=2; seen=set()

L5 maximum_island_area: best = 0
grid=[[0, 0]]; rows=1; cols=2; seen=set(); best=0

L6 maximum_island_area: for r in range(rows):
grid=[[0, 0]]; rows=1; cols=2; seen=set(); best=0; r=0

L7 maximum_island_area: for c in range(cols):
grid=[[0, 0]]; rows=1; cols=2; seen=set(); best=0; r=0; c=0

L7 maximum_island_area: for c in range(cols):
grid=[[0, 0]]; rows=1; cols=2; seen=set(); best=0; r=0; c=1
```

Result: 0. No land starts a traversal.

Why the solution is correct

Discovery marking ensures each land cell contributes exactly once to its own component area. Each island is traversed from one seed, so the maximum over completed areas is exactly the largest island size.

Time and space

O(R*C) time and O(R*C) auxiliary space.

Common mistakes

Resetting seen for every seed repeats components. Failing to reset area accumulates multiple islands into one fictitious area.

Reasoning card

State: area counts popped cells in the active island. best summarizes completed islands. seen prevents overlap between traversals.

Invariant: Current component count includes exactly its visited land cells.

Decision to rehearse: Make component traversal return or accumulate its size.

Finish: Return best after all seeds

Transfer exercise

Compute each island's perimeter by counting land-to-water or land-to-outside edges during the same traversal.

135. Surrounded Regions

Bridge | Module 14

Problem

Given a rectangular mutable board of 'O' and 'X', flip every O region that cannot reach the border through four-neighbor O cells. Preserve border-connected O cells and return the same board. Empty input is unchanged.

Examples

Example 1. The bottom-border O survives; the interior component flips.

```
surrounded_regions([list("XXXX"),list("XOOX"),list("XXOX"),list("XOXX")])
=> [['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'O', 'X',
    'X']]
```

Example 2. Every cell of a one-row board touches the border.

```
surrounded_regions([list("OO")])
=> [['O', 'O']]
```

Example 3. There are no regions to classify.

```
surrounded_regions([])
=> []
```

Why this mechanism works

It is easier to prove which cells must survive than to examine every enclosed region independently. Start from all border O cells and mark their reachable region temporarily. Every remaining O is enclosed; after flipping those cells, restore the temporary marks.

State and boundaries

Temporary 'S' marks O cells connected to a border. queue contains discovered safe cells awaiting expansion.

Protocol

1. Mark and enqueue all border O cells.
2. Flood through four-neighbor O cells, marking on discovery.
3. Scan the board.
4. Convert unmarked O to X and S back to O.

Visualization

```
border O -> mark all reachable O as S
remaining O has no border path -> X
restore S -> O
```

Python implementation

```
from collections import deque

def surrounded_regions(board):
    if not board or not board[0]:
        return board
    rows, cols = len(board), len(board[0])
    queue = deque()
    def mark(r, c):
        if 0 <= r < rows and 0 <= c < cols and board[r][c] == 'O':
            board[r][c] = 'S'
            queue.append((r, c))
    for r in range(rows):
        mark(r, 0)
        mark(r, cols - 1)
    for c in range(cols):
        mark(0, c)
        mark(rows - 1, c)
    while queue:
        r, c = queue.popleft()
        for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
            mark(r + dr, c + dc)
    for r in range(rows):
        for c in range(cols):
            if board[r][c] == 'O':
                board[r][c] = 'X'
            elif board[r][c] == 'S':
                board[r][c] = 'O'
    return board
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
surrounded_regions([list("XXXX"),list("XOOX"),list("XXOX"),list("XOXX")])

L4 surrounded_regions: if not board or not board[0]:
board=[['X', 'X', 'X', 'X'], ['X', 'O', 'O', 'X'], ['X', 'X', 'O', 'X'], ['X', 'O', 'X', 'X']]

L6 surrounded_regions: rows, cols = len(board), len(board[0])
board=[['X', 'X', 'X', 'X'], ['X', 'O', 'O', 'X'], ['X', 'X', 'O', 'X'], ['X', 'O', 'X',
    'X']]; cols=4; rows=4

L7 surrounded_regions: queue = deque()
board=[['X', 'X', 'X', 'X'], ['X', 'O', 'O', 'X'], ['X', 'X', 'O', 'X'], ['X', 'O', 'X',
    'X']]; cols=4; rows=4; queue=deque([])

L12 surrounded_regions: for r in range(rows):
board=[['X', 'X', 'X', 'X'], ['X', 'O', 'O', 'X'], ['X', 'X', 'O', 'X'], ['X', 'O', 'X',
    'X']]; cols=4; rows=4; queue=deque([]); r=0
```

```
L9 mark: if 0 <= r < rows and 0 <= c < cols and board[r][c] == 'O':
r=0; c=0; board=[['X', 'X', 'X', 'X'], ['X', 'O', 'O', 'X'], ['X', 'X', 'O', 'X'], ['X', 'O',
    'X', 'X']]; cols=4; queue=deque([]); rows=4

L9 mark: if 0 <= r < rows and 0 <= c < cols and board[r][c] == 'O':
r=0; c=3; board=[['X', 'X', 'X', 'X'], ['X', 'O', 'O', 'X'], ['X', 'X', 'O', 'X'], ['X', 'O',
    'X', 'X']]; cols=4; queue=deque([]); rows=4

L12 surrounded_regions: for r in range(rows):
board=[['X', 'X', 'X', 'X'], ['X', 'O', 'O', 'X'], ['X', 'X', 'O', 'X'], ['X', 'O', 'X',
    'X']]; cols=4; rows=4; queue=deque([]); r=1

L9 mark: if 0 <= r < rows and 0 <= c < cols and board[r][c] == 'O':
r=1; c=0; board=[['X', 'X', 'X', 'X'], ['X', 'O', 'O', 'X'], ['X', 'X', 'O', 'X'], ['X', 'O',
    'X', 'X']]; cols=4; queue=deque([]); rows=4

... 51 intermediate state changes omitted ...

L23 surrounded_regions: for c in range(cols):
board=[['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'S', 'X',
    'X']]; cols=4; rows=4; queue=deque([]); r=3; c=1; dr=0; dc=-1

L27 surrounded_regions: board[r][c] = 'O'
board=[['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'O', 'X',
    'X']]; cols=4; rows=4; queue=deque([]); r=3; c=1; dr=0; dc=-1

L23 surrounded_regions: for c in range(cols):
board=[['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'O', 'X',
    'X']]; cols=4; rows=4; queue=deque([]); r=3; c=2; dr=0; dc=-1

L23 surrounded_regions: for c in range(cols):
board=[['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'O', 'X',
    'X']]; cols=4; rows=4; queue=deque([]); r=3; c=3; dr=0; dc=-1
```

Result: [['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'X', 'X', 'X'], ['X', 'O', 'X', 'X']]. The bottom-border O survives; the interior component flips.

```
surrounded_regions([list("OO")])

L4 surrounded_regions: if not board or not board[0]:
board=[['O', 'O']]

L6 surrounded_regions: rows, cols = len(board), len(board[0])
board=[['O', 'O']]; cols=2; rows=1

L7 surrounded_regions: queue = deque()
board=[['O', 'O']]; cols=2; rows=1; queue=deque([])

L12 surrounded_regions: for r in range(rows):
board=[['O', 'O']]; cols=2; rows=1; queue=deque([]); r=0

L9 mark: if 0 <= r < rows and 0 <= c < cols and board[r][c] == 'O':
r=0; c=0; board=[['O', 'O']]; cols=2; queue=deque([]); rows=1

L10 mark: board[r][c] = 'S'
r=0; c=0; board=[['S', 'O']]; cols=2; queue=deque([]); rows=1

L11 mark: queue.append((r, c))
r=0; c=0; board=[['S', 'O']]; cols=2; queue=deque([(0, 0)]); rows=1

L13 surrounded_regions: mark(r, 0)
board=[['S', 'O']]; cols=2; rows=1; queue=deque([(0, 0)]); r=0

... 28 intermediate state changes omitted ...
```

```
L23 surrounded_regions: for c in range(cols):
board=[['S', 'S']]; cols=2; rows=1; queue=deque([]); r=0; c=0; dr=0; dc=-1

L27 surrounded_regions: board[r][c] = 'O'
board=[['O', 'S']]; cols=2; rows=1; queue=deque([]); r=0; c=0; dr=0; dc=-1

L23 surrounded_regions: for c in range(cols):
board=[['O', 'S']]; cols=2; rows=1; queue=deque([]); r=0; c=1; dr=0; dc=-1

L27 surrounded_regions: board[r][c] = 'O'
board=[['O', 'O']]; cols=2; rows=1; queue=deque([]); r=0; c=1; dr=0; dc=-1
```

Result: [['O', 'O']]. Every cell of a one-row board touches the border.

Why the solution is correct

A marked cell has an O-only path to a border, so it must survive. Every border-connected O is reachable from one of the seeded cells and is marked. Consequently the unmarked O cells are exactly the regions with no border path, which the contract requires flipping.

Time and space

O(R*C) time and O(R*C) queue space; temporary marks reuse the board.

Common mistakes

Starting only from corners misses other border connections. Restoring marks before the final classification loses the safe/enclosed distinction.

Reasoning card

State: Temporary 'S' marks O cells connected to a border. queue contains discovered safe cells awaiting expansion.

Invariant: Marked safe cells are exactly those connected to a border O.

Decision to rehearse: Traverse safe regions from the border, then flip the rest.

Finish: Convert unmarked O to X and S back to O

Transfer exercise

Model an outside super-node and solve the same preservation question with union-find.

136. Rotting Oranges

Bridge | Module 14

Problem

In a rectangular grid, 0 is empty, 1 is fresh orange and 2 is rotten orange. Each minute, every rotten orange rots adjacent fresh oranges in four directions simultaneously. Return the minimum minutes until none remain fresh, or -1 if impossible. Mutate fresh cells to rotten as discovered; no fresh oranges initially means zero.

Examples

Example 1. The farthest reachable fresh orange rots at minute four.

```
rotting_oranges([[2,1,1],[1,1,0],[0,1,1]])
=> 4
```

Example 2. The fresh orange is isolated by empty space.

```
rotting_oranges([[2,0,1]])
=> -1
```

Example 3. No fresh orange needs time to rot.

```
rotting_oranges([[0,2]])
=> 0
```

Why this mechanism works

All initial rotten cells are distance-zero sources. A multi-source BFS discovers each fresh orange at its earliest possible infection time. Track the remaining fresh count separately: a finished queue does not prove success when isolated fresh cells remain.

State and boundaries

queue stores rotten cells with their infection minute. fresh counts undiscovered fresh oranges; minutes is the latest processed infection time.

Protocol

1. Seed every rotten cell at time zero and count fresh cells.
2. Pop in FIFO order.
3. Recolor each fresh neighbor to rotten before enqueueing it at time+1.
4. Decrement fresh once per discovery.
5. Return minutes only if fresh becomes zero.

Visualization

```
all initial rotten cells: time 0
their fresh neighbors: time 1
next frontier: time 2
answer is last infection time, not visited count
```

Python implementation

```
from collections import deque

def rotting_oranges(grid):
    rows = len(grid)
    cols = len(grid[0]) if rows else 0
    queue = deque()
    fresh = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
                queue.append((r, c, 0))
            elif grid[r][c] == 1:
                fresh += 1
    minutes = 0
    while queue:
        r, c, time = queue.popleft()
        minutes = max(minutes, time)
        for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 1:
                grid[nr][nc] = 2
                fresh -= 1
                queue.append((nr, nc, time + 1))
    return minutes if fresh == 0 else -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
rotting_oranges([[2,1,1],[1,1,0],[0,1,1]])

L4 rotting_oranges: rows = len(grid)
grid=[[2, 1, 1], [1, 1, 0], [0, 1, 1]]; rows=3

L5 rotting_oranges: cols = len(grid[0]) if rows else 0
grid=[[2, 1, 1], [1, 1, 0], [0, 1, 1]]; rows=3; cols=3

L6 rotting_oranges: queue = deque()
grid=[[2, 1, 1], [1, 1, 0], [0, 1, 1]]; rows=3; cols=3; queue=deque([])

L7 rotting_oranges: fresh = 0
grid=[[2, 1, 1], [1, 1, 0], [0, 1, 1]]; rows=3; cols=3; queue=deque([]); fresh=0

L8 rotting_oranges: for r in range(rows):
grid=[[2, 1, 1], [1, 1, 0], [0, 1, 1]]; rows=3; cols=3; queue=deque([]); fresh=0; r=0

L9 rotting_oranges: for c in range(cols):
grid=[[2, 1, 1], [1, 1, 0], [0, 1, 1]]; rows=3; cols=3; queue=deque([]); fresh=0; r=0; c=0
```

```
L11 rotting_oranges: queue.append((r, c, 0))
grid=[[2, 1, 1], [1, 1, 0], [0, 1, 1]]; rows=3; cols=3; queue=deque([(0, 0, 0)]); fresh=0;
    r=0; c=0

L9 rotting_oranges: for c in range(cols):
grid=[[2, 1, 1], [1, 1, 0], [0, 1, 1]]; rows=3; cols=3; queue=deque([(0, 0, 0)]); fresh=0;
    r=0; c=1

... 97 intermediate state changes omitted ...

L18 rotting_oranges: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[2, 2, 2], [2, 2, 0], [0, 2, 2]]; rows=3; cols=3; queue=deque([]); fresh=0; r=2; c=2;
    minutes=4; time=4; dr=0; dc=1; nr=1; nc=2

L19 rotting_oranges: nr, nc = r + dr, c + dc
grid=[[2, 2, 2], [2, 2, 0], [0, 2, 2]]; rows=3; cols=3; queue=deque([]); fresh=0; r=2; c=2;
    minutes=4; time=4; dr=0; dc=1; nr=2; nc=3

L18 rotting_oranges: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[2, 2, 2], [2, 2, 0], [0, 2, 2]]; rows=3; cols=3; queue=deque([]); fresh=0; r=2; c=2;
    minutes=4; time=4; dr=0; dc=-1; nr=2; nc=3

L19 rotting_oranges: nr, nc = r + dr, c + dc
grid=[[2, 2, 2], [2, 2, 0], [0, 2, 2]]; rows=3; cols=3; queue=deque([]); fresh=0; r=2; c=2;
    minutes=4; time=4; dr=0; dc=-1; nr=2; nc=1
```

Result: 4. The farthest reachable fresh orange rots at minute four.

```
rotting_oranges([[2,0,1]])

L4 rotting_oranges: rows = len(grid)
grid=[[2, 0, 1]]; rows=1

L5 rotting_oranges: cols = len(grid[0]) if rows else 0
grid=[[2, 0, 1]]; rows=1; cols=3

L6 rotting_oranges: queue = deque()
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([])

L7 rotting_oranges: fresh = 0
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([]); fresh=0

L8 rotting_oranges: for r in range(rows):
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([]); fresh=0; r=0

L9 rotting_oranges: for c in range(cols):
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([]); fresh=0; r=0; c=0

L11 rotting_oranges: queue.append((r, c, 0))
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([(0, 0, 0)]); fresh=0; r=0; c=0

L9 rotting_oranges: for c in range(cols):
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([(0, 0, 0)]); fresh=0; r=0; c=1

... 8 intermediate state changes omitted ...

L18 rotting_oranges: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([]); fresh=1; r=0; c=0; minutes=0; time=0; dr=0;
     dc=1; nr=-1; nc=0

L19 rotting_oranges: nr, nc = r + dr, c + dc
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([]); fresh=1; r=0; c=0; minutes=0; time=0; dr=0;
     dc=1; nr=0; nc=1
```

```
L18 rotting_oranges: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([]); fresh=1; r=0; c=0; minutes=0; time=0; dr=0;
     dc=-1; nr=0; nc=1

L19 rotting_oranges: nr, nc = r + dr, c + dc
grid=[[2, 0, 1]]; rows=1; cols=3; queue=deque([]); fresh=1; r=0; c=0; minutes=0; time=0; dr=0;
     dc=-1; nr=0; nc=-1
```

Result: -1. The fresh orange is isolated by empty space.

Why the solution is correct

BFS processes infection layers in increasing time, so the first discovery gives each reachable orange its minimum infection time. The maximum such time is when all reachable fresh oranges have rotted. A positive remaining fresh count identifies exactly the unreachable ones.

Time and space

O(R*C) time and O(R*C) queue space.

Common mistakes

Starting from only one rotten orange ignores parallel spread. Counting one minute per popped cell confuses cells with BFS layers.

Reasoning card

State: queue stores rotten cells with their infection minute. fresh counts undiscovered fresh oranges; minutes is the latest processed infection time.

Invariant: First reach time is the shortest time from any initial rotten source.

Decision to rehearse: Enqueue all sources first; process one BFS layer per minute.

Finish: Return minutes only if fresh becomes zero

Transfer exercise

Assign different nonnegative transmission times to edges and explain why FIFO BFS must become a weighted shortest-path algorithm.

137. Shortest Binary-Matrix Path

Bridge | Module 14

Problem

Find the shortest open-cell path from top-left to bottom-right in a rectangular binary matrix, where 0 is open and 1 is blocked. Eight-direction movement is allowed, including diagonals past blocked side neighbors. Return the number of cells on the path, including both endpoints, or -1 if none exists. Leave input unchanged.

Examples

Example 1. A diagonal is a legal unit-cost move.

```
shortest_binary_path([[0,1],[1,0]])
=> 2
```

Example 2. The start is already the destination and counts as one cell.

```
shortest_binary_path([[0]])
=> 1
```

Example 3. A blocked endpoint prevents every path.

```
shortest_binary_path([[1]])
=> -1
```

Why this mechanism works

Every move has equal cost, so BFS first reaches each cell by a shortest route. Store distance in cells, starting at one for the initial cell. Mark on enqueue to avoid duplicate frontier entries. Blocked endpoints can be rejected before traversal.

State and boundaries

queue contains discovered open cells with their shortest cell-count distance. seen prevents rediscovery.

Protocol

1. Reject empty or blocked endpoints.
2. Enqueue the start at distance one.
3. Pop the next BFS cell.
4. Return its distance when it is the destination.
5. Enqueue unseen open neighbors at distance+1.
6. Return -1 if exhausted.

Visualization

```
0 1
1 0
diagonal path visits two open cells
answer 2, despite blocked side neighbors
```

Python implementation

```
from collections import deque

def shortest_binary_path(grid):
    if not grid or not grid[0] or grid[0][0] != 0 or grid[-1][-1] != 0:
        return -1
    rows, cols = len(grid), len(grid[0])
    queue = deque([(0, 0, 1)])
    seen = {(0, 0)}
    while queue:
        r, c, distance = queue.popleft()
        if (r, c) == (rows - 1, cols - 1):
            return distance
        for dr in (-1, 0, 1):
            for dc in (-1, 0, 1):
                nr, nc = r + dr, c + dc
                if ((dr or dc) and 0 <= nr < rows and 0 <= nc < cols
                        and grid[nr][nc] == 0 and (nr, nc) not in seen):
                    seen.add((nr, nc))
                    queue.append((nr, nc, distance + 1))
    return -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
shortest_binary_path([[0,1],[1,0]])

L4 shortest_binary_path: if not grid or not grid[0] or grid[0][0] != 0 or grid[-1][-1] != 0:
grid=[[0, 1], [1, 0]]

L6 shortest_binary_path: rows, cols = len(grid), len(grid[0])
grid=[[0, 1], [1, 0]]; rows=2; cols=2

L7 shortest_binary_path: queue = deque([(0, 0, 1)])
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([(0, 0, 1)])

L8 shortest_binary_path: seen = {(0, 0)}
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([(0, 0, 1)]); seen={(0, 0)}

L10 shortest_binary_path: r, c, distance = queue.popleft()
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([]); seen={(0, 0)}; r=0; c=0; distance=1

L13 shortest_binary_path: for dr in (-1, 0, 1):
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([]); seen={(0, 0)}; r=0; c=0; distance=1;
    dr=-1

L14 shortest_binary_path: for dc in (-1, 0, 1):
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([]); seen={(0, 0)}; r=0; c=0; distance=1;
    dr=-1; dc=-1
```

```
L15 shortest_binary_path: nr, nc = r + dr, c + dc
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([]); seen={(0, 0)}; r=0; c=0; distance=1;
    dr=-1; dc=-1; nr=-1; nc=-1

... 17 intermediate state changes omitted ...

L15 shortest_binary_path: nr, nc = r + dr, c + dc
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([]); seen={(0, 0)}; r=0; c=0; distance=1;
    dr=1; dc=1; nr=1; nc=1

L18 shortest_binary_path: seen.add((nr, nc))
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([]); seen={(0, 0), (1, 1)}; r=0; c=0;
    distance=1; dr=1; dc=1; nr=1; nc=1

L19 shortest_binary_path: queue.append((nr, nc, distance + 1))
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([(1, 1, 2)]); seen={(0, 0), (1, 1)}; r=0;
    c=0; distance=1; dr=1; dc=1; nr=1; nc=1

L10 shortest_binary_path: r, c, distance = queue.popleft()
grid=[[0, 1], [1, 0]]; rows=2; cols=2; queue=deque([]); seen={(0, 0), (1, 1)}; r=1; c=1;
    distance=2; dr=1; dc=1; nr=1; nc=1
```

Result: 2. A diagonal is a legal unit-cost move.

```
shortest_binary_path([[0]])

L4 shortest_binary_path: if not grid or not grid[0] or grid[0][0] != 0 or grid[-1][-1] != 0:
grid=[[0]]

L6 shortest_binary_path: rows, cols = len(grid), len(grid[0])
grid=[[0]]; rows=1; cols=1

L7 shortest_binary_path: queue = deque([(0, 0, 1)])
grid=[[0]]; rows=1; cols=1; queue=deque([(0, 0, 1)])

L8 shortest_binary_path: seen = {(0, 0)}
grid=[[0]]; rows=1; cols=1; queue=deque([(0, 0, 1)]); seen={(0, 0)}

L10 shortest_binary_path: r, c, distance = queue.popleft()
grid=[[0]]; rows=1; cols=1; queue=deque([]); seen={(0, 0)}; r=0; c=0; distance=1
```

Result: 1. The start is already the destination and counts as one cell.

Why the solution is correct

FIFO order expands all paths of one move count before longer paths. Therefore a cell's first discovery has minimum distance, and the first destination removal yields the shortest path. Exhaustion means every reachable open cell was considered without reaching the destination.

Time and space

O(R*C) time and O(R*C) auxiliary space; eight neighbors per cell is a constant factor.

Common mistakes

Starting distance at zero reports edges rather than cells. Using only four neighbors rejects legal diagonal paths under this contract.

Reasoning card

State: queue contains discovered open cells with their shortest cell-count distance. seen prevents rediscovery.

Invariant: Dequeued distance is minimal for that cell.

Decision to rehearse: BFS because every move has equal cost.

Finish: Return -1 if exhausted

Transfer exercise

Store one predecessor per discovered cell to return a shortest path, not just its length.

138. Nearest-Zero Distances

Interview | Module 14

Problem

For every cell of a rectangular binary matrix, return its minimum four-neighbor distance to any zero. Zeros have distance zero. If there is no zero, use -1 for every cell. Return a new matrix without changing the input.

Examples

Example 1. Each cell is reached by its nearest zero frontier.

```
nearest_zero_distances([[0,0,0],[0,1,0],[1,1,1]])
=> [[0, 0, 0], [0, 1, 0], [1, 2, 1]]
```

Example 2. No source exists.

```
nearest_zero_distances([[1,1]])
=> [[-1, -1]]
```

Example 3. No distance cells are allocated.

```
nearest_zero_distances([])
=> []
```

Why this mechanism works

Running BFS from every one repeats work. Reverse the viewpoint: seed all zeros simultaneously and let their distance frontiers compete. The first frontier reaching a cell is necessarily from a nearest zero because all sources begin at the same distance and all edges cost one.

State and boundaries

distance is -1 for undiscovered cells and final shortest distance for discovered cells. queue begins with every zero.

Protocol

1. Allocate distance=-1.
2. Set zero cells to distance zero and enqueue them.
3. Pop cells in FIFO order.
4. Assign each undiscovered neighbor distance+1 and enqueue.
5. Return the distance matrix.

Visualization

```
0 1 1
1 1 0
all zeros seed layer 0
nearest-source waves fill distances 1,2,...
```

Python implementation

```
from collections import deque

def nearest_zero_distances(matrix):
    rows = len(matrix)
    cols = len(matrix[0]) if rows else 0
    distance = [[-1] * cols for _ in range(rows)]
    queue = deque()
    for r in range(rows):
        for c in range(cols):
            if matrix[r][c] == 0:
                distance[r][c] = 0
                queue.append((r, c))
    while queue:
        r, c = queue.popleft()
        for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and distance[nr][nc] == -1:
                distance[nr][nc] = distance[r][c] + 1
                queue.append((nr, nc))
    return distance
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
nearest_zero_distances([[0,0,0],[0,1,0],[1,1,1]])

L4 nearest_zero_distances: rows = len(matrix)
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3

L5 nearest_zero_distances: cols = len(matrix[0]) if rows else 0
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3

L6 nearest_zero_distances: distance = [[-1] * cols for _ in range(rows)]
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; _=0

L6 nearest_zero_distances: distance = [[-1] * cols for _ in range(rows)]
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; _=1

L6 nearest_zero_distances: distance = [[-1] * cols for _ in range(rows)]
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; _=2

L6 nearest_zero_distances: distance = [[-1] * cols for _ in range(rows)]
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; distance=[[-1, -1, -1], [-1, -1,
    -1], [-1, -1, -1]]

L7 nearest_zero_distances: queue = deque()
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; distance=[[-1, -1, -1], [-1, -1,
    -1], [-1, -1, -1]]; queue=deque([])
```

```
L8 nearest_zero_distances: for r in range(rows):
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; distance=[[-1, -1, -1], [-1, -1,
    -1], [-1, -1, -1]]; queue=deque([]); r=0

... 106 intermediate state changes omitted ...

L15 nearest_zero_distances: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; distance=[[0, 0, 0], [0, 1, 0], [1,
    2, 1]]; queue=deque([]); r=2; c=1; dr=0; dc=1; nr=1; nc=1

L16 nearest_zero_distances: nr, nc = r + dr, c + dc
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; distance=[[0, 0, 0], [0, 1, 0], [1,
    2, 1]]; queue=deque([]); r=2; c=1; dr=0; dc=1; nr=2; nc=2

L15 nearest_zero_distances: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; distance=[[0, 0, 0], [0, 1, 0], [1,
    2, 1]]; queue=deque([]); r=2; c=1; dr=0; dc=-1; nr=2; nc=2

L16 nearest_zero_distances: nr, nc = r + dr, c + dc
matrix=[[0, 0, 0], [0, 1, 0], [1, 1, 1]]; rows=3; cols=3; distance=[[0, 0, 0], [0, 1, 0], [1,
    2, 1]]; queue=deque([]); r=2; c=1; dr=0; dc=-1; nr=2; nc=0
```

Result: [[0, 0, 0], [0, 1, 0], [1, 2, 1]]. Each cell is reached by its nearest zero frontier.

```
nearest_zero_distances([[1,1]])

L4 nearest_zero_distances: rows = len(matrix)
matrix=[[1, 1]]; rows=1

L5 nearest_zero_distances: cols = len(matrix[0]) if rows else 0
matrix=[[1, 1]]; rows=1; cols=2

L6 nearest_zero_distances: distance = [[-1] * cols for _ in range(rows)]
matrix=[[1, 1]]; rows=1; cols=2; _=0

L6 nearest_zero_distances: distance = [[-1] * cols for _ in range(rows)]
matrix=[[1, 1]]; rows=1; cols=2; distance=[[-1, -1]]

L7 nearest_zero_distances: queue = deque()
matrix=[[1, 1]]; rows=1; cols=2; distance=[[-1, -1]]; queue=deque([])

L8 nearest_zero_distances: for r in range(rows):
matrix=[[1, 1]]; rows=1; cols=2; distance=[[-1, -1]]; queue=deque([]); r=0

L9 nearest_zero_distances: for c in range(cols):
matrix=[[1, 1]]; rows=1; cols=2; distance=[[-1, -1]]; queue=deque([]); r=0; c=0

L9 nearest_zero_distances: for c in range(cols):
matrix=[[1, 1]]; rows=1; cols=2; distance=[[-1, -1]]; queue=deque([]); r=0; c=1
```

Result: [[-1, -1]]. No source exists.

Why the solution is correct

Multi-source BFS is equivalent to adding a virtual source connected equally to all zeros. Its layer order makes first discovery the minimum distance from any source. Cells left at -1 have no source reachable, which occurs here only when no zero exists.

Time and space

O(R*C) time and O(R*C) distance/queue space.

Common mistakes

Initializing every distance to zero confuses unseen cells with source cells. Starting separate searches from every one can multiply the work by the grid size.

Reasoning card

State: distance is -1 for undiscovered cells and final shortest distance for discovered cells. queue begins with every zero.

Invariant: First assignment to a cell is its nearest-source distance.

Decision to rehearse: Start BFS from all zero cells simultaneously.

Finish: Return the distance matrix

Transfer exercise

Replace zeros with a supplied set of service locations and interpret the result as a nearest-facility distance map.

139. Walls and Gates

Interview | Module 14

Problem

A rectangular room grid contains walls -1, gates 0 and empty rooms INF=2**31-1. Replace each reachable empty room with its shortest four-neighbor distance to a gate, leaving walls, gates and unreachable INF rooms unchanged. Mutate and return the grid.

Examples

Example 1. The wall is preserved while room distances spread from the gate.

```
walls_and_gates([[0,INF,-1],[INF,INF,INF]])
=> [[0, 1, -1], [1, 2, 3]]
```

Example 2. The isolated room remains INF.

```
walls_and_gates([[INF,-1,0]])
=> [[2147483647, -1, 0]]
```

Example 3. There is no frontier to seed.

```
walls_and_gates([])
=> []
```

Why this mechanism works

All gates are simultaneous BFS sources. The INF marker means both empty room and undiscovered distance. Writing a finite distance at enqueue time prevents duplicate discovery and leaves unreachable rooms visibly distinct from distance-zero gates.

State and boundaries

queue holds discovered rooms or gates. A finite nonnegative grid entry is its final nearest-gate distance; INF remains undiscovered.

Protocol

1. Enqueue every gate.
2. Pop the next known-distance cell.
3. For each neighboring INF room, assign current distance+1.
4. Enqueue it immediately.
5. Return the mutated grid after exhaustion.

Visualization

```
0 INF -1
INF INF INF
becomes
0 1 -1
1 2 3
```

Python implementation

```
from collections import deque

INF = 2**31 - 1

def walls_and_gates(rooms):
    rows = len(rooms)
    cols = len(rooms[0]) if rows else 0
    queue = deque((r, c) for r in range(rows) for c in range(cols)
                  if rooms[r][c] == 0)
    while queue:
        r, c = queue.popleft()
        for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and rooms[nr][nc] == INF:
                rooms[nr][nc] = rooms[r][c] + 1
                queue.append((nr, nc))
    return rooms
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
walls_and_gates([[0,INF,-1],[INF,INF,INF]])

L6 walls_and_gates: rows = len(rooms)
rooms=[[0, 2147483647, -1], [2147483647, 2147483647, 2147483647]]; rows=2

L7 walls_and_gates: cols = len(rooms[0]) if rows else 0
rooms=[[0, 2147483647, -1], [2147483647, 2147483647, 2147483647]]; rows=2; cols=3

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=0; c=0; cols=3; rooms=[[0, 2147483647, -1], [2147483647, 2147483647,
    2147483647]]

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=0; c=1; cols=3; rooms=[[0, 2147483647, -1], [2147483647, 2147483647,
    2147483647]]

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=0; c=2; cols=3; rooms=[[0, 2147483647, -1], [2147483647, 2147483647,
    2147483647]]

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=1; c=0; cols=3; rooms=[[0, 2147483647, -1], [2147483647, 2147483647,
    2147483647]]

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=1; c=1; cols=3; rooms=[[0, 2147483647, -1], [2147483647, 2147483647,
    2147483647]]
```

```
L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=1; c=2; cols=3; rooms=[[0, 2147483647, -1], [2147483647, 2147483647,
    2147483647]]

... 50 intermediate state changes omitted ...

L12 walls_and_gates: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
rooms=[[0, 1, -1], [1, 2, 3]]; rows=2; cols=3; queue=deque([]); r=1; c=2; dr=0; dc=1; nr=0;
    nc=2

L13 walls_and_gates: nr, nc = r + dr, c + dc
rooms=[[0, 1, -1], [1, 2, 3]]; rows=2; cols=3; queue=deque([]); r=1; c=2; dr=0; dc=1; nr=1;
    nc=3

L12 walls_and_gates: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
rooms=[[0, 1, -1], [1, 2, 3]]; rows=2; cols=3; queue=deque([]); r=1; c=2; dr=0; dc=-1; nr=1;
    nc=3

L13 walls_and_gates: nr, nc = r + dr, c + dc
rooms=[[0, 1, -1], [1, 2, 3]]; rows=2; cols=3; queue=deque([]); r=1; c=2; dr=0; dc=-1; nr=1;
    nc=1
```

Result: [[0, 1, -1], [1, 2, 3]]. The wall is preserved while room distances spread from the gate.

`walls_and_gates([[INF,-1,0]])`

```
L6 walls_and_gates: rows = len(rooms)
rooms=[[2147483647, -1, 0]]; rows=1

L7 walls_and_gates: cols = len(rooms[0]) if rows else 0
rooms=[[2147483647, -1, 0]]; rows=1; cols=3

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=0; c=0; cols=3; rooms=[[2147483647, -1, 0]]

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=0; c=1; cols=3; rooms=[[2147483647, -1, 0]]

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=0; c=2; cols=3; rooms=[[2147483647, -1, 0]]

L8 <genexpr>: queue = deque((r, c) for r in range(rows) for c in range(cols)
.0=range_iterator; r=0; c=2; cols=3; rooms=[[2147483647, -1, 0]]

L8 walls_and_gates: queue = deque((r, c) for r in range(rows) for c in range(cols)
rooms=[[2147483647, -1, 0]]; rows=1; cols=3; queue=deque([(0, 2)])

L11 walls_and_gates: r, c = queue.popleft()
rooms=[[2147483647, -1, 0]]; rows=1; cols=3; queue=deque([]); r=0; c=2

... 4 intermediate state changes omitted ...

L12 walls_and_gates: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
rooms=[[2147483647, -1, 0]]; rows=1; cols=3; queue=deque([]); r=0; c=2; dr=0; dc=1; nr=-1;
    nc=2

L13 walls_and_gates: nr, nc = r + dr, c + dc
rooms=[[2147483647, -1, 0]]; rows=1; cols=3; queue=deque([]); r=0; c=2; dr=0; dc=1; nr=0; nc=3

L12 walls_and_gates: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
rooms=[[2147483647, -1, 0]]; rows=1; cols=3; queue=deque([]); r=0; c=2; dr=0; dc=-1; nr=0;
    nc=3
```

```
L13 walls_and_gates: nr, nc = r + dr, c + dc
rooms=[[2147483647, -1, 0]]; rows=1; cols=3; queue=deque([]); r=0; c=2; dr=0; dc=-1; nr=0;
    nc=1
```

Result: [[2147483647, -1, 0]]. The isolated room remains INF.

Why the solution is correct

BFS layer order guarantees the first path reaching an INF room is shortest from any gate. Walls are never traversed and already assigned cells need no later improvement under unit edge costs. Remaining INF cells have no gate path.

Time and space

O(R*C) time and O(R*C) queue space; distances reuse the input matrix.

Common mistakes

Using zero as an unvisited marker overwrites gates. Treating walls as candidates lets paths cross forbidden cells.

Reasoning card

State: queue holds discovered rooms or gates. A finite nonnegative grid entry is its final nearest-gate distance; INF remains undiscovered.

Invariant: Distances expand monotonically outward from all gates.

Decision to rehearse: Multi-source BFS from all gates.

Finish: Return the mutated grid after exhaustion

Transfer exercise

Also record which gate serves each room, defining a deterministic tie rule for equal-distance gates.

140. Pacific Atlantic Water Flow

Interview | Module 14

Problem

Given a rectangular height matrix, return all cells from which water can reach both oceans. The Pacific touches the top and left borders; the Atlantic touches the bottom and right borders. Water moves to four-neighbor cells of equal or lower height. Return [row,column] pairs in row-major order; input is unchanged.

Examples

Example 1. Only the lowest top-left cell cannot drain to the Atlantic.

```
pacific_atlantic([[1,2],[4,3]])
=> [[0, 1], [1, 0], [1, 1]]
```

Example 2. The singleton touches both oceans.

```
pacific_atlantic([[7]])
=> [[0, 0]]
```

Example 3. No cell reaches either ocean.

```
pacific_atlantic([])
=> []
```

Why this mechanism works

Searching downhill from every cell repeats paths. Reverse the flow: start from each ocean's borders and move uphill or level. A reverse-reachable cell has a valid downhill path to that ocean. Intersect the two reverse-reachability sets.

State and boundaries

Each reachable set contains cells that can drain to its seeded ocean. Reverse traversal accepts a neighbor only when its height is >= the current height.

Protocol

1. Seed Pacific border cells and traverse reverse flow.
2. Independently seed Atlantic borders and traverse reverse flow.
3. Scan cells in row-major order.
4. Emit those in both reached sets.

Visualization

```
forward water: high -> low -> ocean
reverse search: ocean -> low -> high
answer = Pacific-reachable AND Atlantic-reachable
```

Python implementation

```
from collections import deque

def pacific_atlantic(heights):
    if not heights or not heights[0]:
        return []
    rows, cols = len(heights), len(heights[0])
    def reachable(starts):
        seen = set(starts)
        queue = deque(seen)
        while queue:
            r, c = queue.popleft()
            for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
                nr, nc = r + dr, c + dc
                if (0 <= nr < rows and 0 <= nc < cols and (nr, nc) not in seen
                        and heights[nr][nc] >= heights[r][c]):
                    seen.add((nr, nc))
                    queue.append((nr, nc))
        return seen
    pacific = reachable([(0,c) for c in range(cols)] + [(r,0) for r in range(rows)])
    atlantic = reachable([(rows-1,c) for c in range(cols)]
                         + [(r,cols-1) for r in range(rows)])
    return [[r,c] for r in range(rows) for c in range(cols)
            if (r,c) in pacific and (r,c) in atlantic]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
pacific_atlantic([[1,2],[4,3]])

L4 pacific_atlantic: if not heights or not heights[0]:
heights=[[1, 2], [4, 3]]

L6 pacific_atlantic: rows, cols = len(heights), len(heights[0])
heights=[[1, 2], [4, 3]]; cols=2; rows=2

L19 pacific_atlantic: pacific = reachable([(0,c) for c in range(cols)] + [(r,0) for r in
    range(rows)])
heights=[[1, 2], [4, 3]]; cols=2; rows=2; c=0

L19 pacific_atlantic: pacific = reachable([(0,c) for c in range(cols)] + [(r,0) for r in
    range(rows)])
heights=[[1, 2], [4, 3]]; cols=2; rows=2; c=1

L19 pacific_atlantic: pacific = reachable([(0,c) for c in range(cols)] + [(r,0) for r in
    range(rows)])
heights=[[1, 2], [4, 3]]; cols=2; rows=2; r=0

L19 pacific_atlantic: pacific = reachable([(0,c) for c in range(cols)] + [(r,0) for r in
    range(rows)])
```

```
heights=[[1, 2], [4, 3]]; cols=2; rows=2; r=1

L8 reachable: seen = set(starts)
starts=[(0, 0), (0, 1), (0, 0), (1, 0)]; seen={(0, 0), (0, 1), (1, 0)}; cols=2; heights=[[1,
    2], [4, 3]]; rows=2

L9 reachable: queue = deque(seen)
starts=[(0, 0), (0, 1), (0, 0), (1, 0)]; seen={(0, 0), (0, 1), (1, 0)}; cols=2; heights=[[1,
    2], [4, 3]]; rows=2; queue=deque([(0, 1), (1, 0), (0, 0)])

... 76 intermediate state changes omitted ...

L22 pacific_atlantic: return [[r,c] for r in range(rows) for c in range(cols)
heights=[[1, 2], [4, 3]]; cols=2; rows=2; pacific={(0, 0), (0, 1), (1, 0), (1, 1)};
    atlantic={(0, 1), (1, 0), (1, 1)}; c=1; r=0

L22 pacific_atlantic: return [[r,c] for r in range(rows) for c in range(cols)
heights=[[1, 2], [4, 3]]; cols=2; rows=2; pacific={(0, 0), (0, 1), (1, 0), (1, 1)};
    atlantic={(0, 1), (1, 0), (1, 1)}; c=0; r=1

L22 pacific_atlantic: return [[r,c] for r in range(rows) for c in range(cols)
heights=[[1, 2], [4, 3]]; cols=2; rows=2; pacific={(0, 0), (0, 1), (1, 0), (1, 1)};
    atlantic={(0, 1), (1, 0), (1, 1)}; c=1; r=1

L22 pacific_atlantic: return [[r,c] for r in range(rows) for c in range(cols)
heights=[[1, 2], [4, 3]]; cols=2; rows=2; pacific={(0, 0), (0, 1), (1, 0), (1, 1)};
    atlantic={(0, 1), (1, 0), (1, 1)}
```

Result: [[0, 1], [1, 0], [1, 1]]. Only the lowest top-left cell cannot drain to the Atlantic.

```
pacific_atlantic([[7]])

L4 pacific_atlantic: if not heights or not heights[0]:
heights=[[7]]

L6 pacific_atlantic: rows, cols = len(heights), len(heights[0])
heights=[[7]]; cols=1; rows=1

L19 pacific_atlantic: pacific = reachable([(0,c) for c in range(cols)] + [(r,0) for r in
    range(rows)])
heights=[[7]]; cols=1; rows=1; c=0

L19 pacific_atlantic: pacific = reachable([(0,c) for c in range(cols)] + [(r,0) for r in
    range(rows)])
heights=[[7]]; cols=1; rows=1; r=0

L8 reachable: seen = set(starts)
starts=[(0, 0), (0, 0)]; seen={(0, 0)}; cols=1; heights=[[7]]; rows=1

L9 reachable: queue = deque(seen)
starts=[(0, 0), (0, 0)]; seen={(0, 0)}; cols=1; heights=[[7]]; rows=1; queue=deque([(0, 0)])

L11 reachable: r, c = queue.popleft()
starts=[(0, 0), (0, 0)]; seen={(0, 0)}; cols=1; heights=[[7]]; rows=1; queue=deque([]); r=0;
    c=0

L12 reachable: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
starts=[(0, 0), (0, 0)]; seen={(0, 0)}; cols=1; heights=[[7]]; rows=1; queue=deque([]); r=0;
    c=0; dr=1; dc=0

... 22 intermediate state changes omitted ...

L13 reachable: nr, nc = r + dr, c + dc
starts=[(0, 0), (0, 0)]; seen={(0, 0)}; cols=1; heights=[[7]]; rows=1; queue=deque([]); r=0;
```

```
    c=0; dr=0; dc=-1; nr=0; nc=-1

L20 pacific_atlantic: atlantic = reachable([(rows-1,c) for c in range(cols)]
heights=[[7]]; cols=1; rows=1; pacific={(0, 0)}; atlantic={(0, 0)}

L22 pacific_atlantic: return [[r,c] for r in range(rows) for c in range(cols)
heights=[[7]]; cols=1; rows=1; pacific={(0, 0)}; atlantic={(0, 0)}; c=0; r=0

L22 pacific_atlantic: return [[r,c] for r in range(rows) for c in range(cols)
heights=[[7]]; cols=1; rows=1; pacific={(0, 0)}; atlantic={(0, 0)}
```

Result: [[0, 0]]. The singleton touches both oceans.

Why the solution is correct

Reversing each downhill edge creates exactly the uphill-or-level traversal rule. Thus a cell is reached from an ocean border precisely when a forward water path exists from it to that ocean. Membership in both sets is equivalent to reaching both oceans.

Time and space

O(R*C) time and O(R*C) auxiliary space for two traversals and their reached sets.

Common mistakes

Using downhill movement from the ocean computes the wrong relation. Requiring strictly greater height incorrectly blocks level plateaus.

Reasoning card

State: Each reachable set contains cells that can drain to its seeded ocean. Reverse traversal accepts a neighbor only when its height is >= the current height.

Invariant: Each visited set contains cells capable of reaching that ocean.

Decision to rehearse: Reverse the flow: start from oceans and traverse uphill/reverse-valid edges.

Finish: Emit those in both reached sets

Transfer exercise

Generalize the intersection to three destination regions and decide whether to keep separate sets or a bitmask per cell.

Module 15 Recursion and Divide-and-Conquer

Define what one call returns, identify the base case, and combine smaller solved instances.

Recursion replaces a loop driver with a call contract. A child call should return a complete, named result; the parent combines it once. Practice descent and unwinding before branching into many choices. Divide-and-conquer adds a proof that smaller solved pieces can be combined without losing order or duplicating work.

The ten-step progression

141. Factorial with Call Trace

142. Sum a Nested List

143. Flatten a Nested List

144. Reverse a String Recursively

145. Fast Exponentiation

146. Merge Sort

147. Quicksort Partition

148. Quickselect Kth Statistic

149. Tower of Hanoi

150. Merge K Sorted Arrays by Divide-and-Conquer

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

141. Factorial with Call Trace

Foundation | Module 15

Problem

Implement factorial recursively for a nonnegative integer n, returning n! and printing one call event and one return event for every invocation. Use 0!=1 and 1!=1. The n=4 example must expose both descent and unwinding; large n is limited by Python recursion depth and integer size.

Examples

Example 1. The printed events show four frames and their reverse return order.

```
factorial(4)
=> 24
Printed call/return trace:
call factorial(4)
call factorial(3)
call factorial(2)
call factorial(1)
return factorial(1) = 1
return factorial(2) = 2
return factorial(3) = 6
return factorial(4) = 24
```

Example 2. The empty product is the base value one.

```
factorial(0)
=> 1
Printed call/return trace:
call factorial(0)
return factorial(0) = 1
```

Example 3. The singleton product is another base case.

```
factorial(1)
=> 1
Printed call/return trace:
call factorial(1)
return factorial(1) = 1
```

Why this mechanism works

A recursive call should promise one complete answer: factorial(n-1) returns the product through n-1. The current call then multiplies that answer by n. Call events occur while descending; return events occur in reverse order as pending multiplications become computable.

State and boundaries

Each frame owns its n and result. The caller waits for one smaller factorial before computing its own result.

Protocol

1. Print the call event.
2. Return one for n<=1 after printing its return.
3. Otherwise obtain factorial(n-1) once.
4. Multiply by n.
5. Print and return the completed result.

Visualization

```
call 4 -> call 3 -> call 2 -> call 1
return 1 <- return 2 <- return 6 <- return 24
work pending on descent completes on unwind
```

Python implementation

```
def factorial(n):
    print(f'call factorial({n})')
    if n <= 1:
        result = 1
    else:
        result = n * factorial(n - 1)
    print(f'return factorial({n}) = {result}')
    return result
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
factorial(4)

L2 factorial: print(f'call factorial({n})')
n=4

L2 factorial: print(f'call factorial({n})')
n=3

L2 factorial: print(f'call factorial({n})')
n=2

L2 factorial: print(f'call factorial({n})')
n=1

L4 factorial: result = 1
n=1; result=1

L6 factorial: result = n * factorial(n - 1)
```

```
n=2; result=2

L6 factorial: result = n * factorial(n - 1)
n=3; result=6

L6 factorial: result = n * factorial(n - 1)
n=4; result=24
```

Result: 24. The printed events show four frames and their reverse return order.

```
factorial(0)

L2 factorial: print(f'call factorial({n})')
n=0

L4 factorial: result = 1
n=0; result=1
```

Result: 1. The empty product is the base value one.

Why the solution is correct

The base cases match the factorial definition. Assuming the smaller call returns (n-1)!, multiplying by n returns n!, proving correctness by induction. Each recursive argument decreases, so it eventually reaches a base case.

Time and space

O(n) multiplications and O(n) call frames; output integers and arithmetic costs grow with n. Printed trace size is also linear in the number of calls, apart from number lengths.

Common mistakes

Multiplying before the child has returned is not a separate completed result. A missing base case recurses indefinitely until the runtime limit.

Reasoning card

State: Each frame owns its n and result. The caller waits for one smaller factorial before computing its own result.

Invariant: factorial(n) returns the product 1..n.

Decision to rehearse: Separate downward calls from upward combination.

Finish: Print and return the completed result

Transfer exercise

Replace implicit call frames with an explicit stack of pending multipliers, then compare the traces.

142. Sum a Nested List

Foundation | Module 15

Problem

Return the sum of all integer leaves in a finite nested structure whose nodes are either integers or lists of such nodes. Empty lists contribute zero, negative integers are allowed, and the structure has no cycles. Do not change it.

Examples

Example 1. Each integer leaf contributes once.

```
nested_sum([1,[2,[3]],4])
=> 10
```

Example 2. Empty lists contribute zero and signs are retained.

```
nested_sum([[],[-2,3]])
=> 1
```

Example 3. An integer is a complete base-case structure.

```
nested_sum(7)
=> 7
```

Why this mechanism works

There are two structural cases. An integer already is its own contribution. A list delegates each child to the same contract and adds the returned contributions. The recursive shape follows the data rather than a fixed number of nesting levels.

State and boundaries

Each call returns the sum of exactly its supplied subtree. total accumulates completed child results for a list node.

Protocol

1. If the current item is an integer, return it.
2. Otherwise start total at zero.
3. Recursively sum each child and add its result.
4. Return the accumulated total.

Visualization

```
[1,[2,[3]],4]
 sum = 1 + sum([2,[3]]) + 4
     = 1 + (2 + 3) + 4 = 10
```

Python implementation

```
def nested_sum(item):
    if isinstance(item, int):
        return item
    total = 0
    for child in item:
        total += nested_sum(child)
    return total
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
nested_sum([1,[2,[3]],4])

L2 nested_sum: if isinstance(item, int):
item=[1, [2, [...]], 4]

L4 nested_sum: total = 0
item=[1, [2, [...]], 4]; total=0

L5 nested_sum: for child in item:
item=[1, [2, [...]], 4]; total=0; child=1

L2 nested_sum: if isinstance(item, int):
item=1

L6 nested_sum: total += nested_sum(child)
item=[1, [2, [...]], 4]; total=1; child=1

L5 nested_sum: for child in item:
item=[1, [2, [...]], 4]; total=1; child=[2, [3]]

L2 nested_sum: if isinstance(item, int):
item=[2, [3]]

L4 nested_sum: total = 0
item=[2, [3]]; total=0

... 10 intermediate state changes omitted ...

L6 nested_sum: total += nested_sum(child)
item=[1, [2, [...]], 4]; total=6; child=[2, [3]]

L5 nested_sum: for child in item:
item=[1, [2, [...]], 4]; total=6; child=4

L2 nested_sum: if isinstance(item, int):
item=4
```

```
L6 nested_sum: total += nested_sum(child)
item=[1, [2, [...]], 4]; total=10; child=4
```

Result: 10. Each integer leaf contributes once.

```
nested_sum([[],[-2,3]])

L2 nested_sum: if isinstance(item, int):
item=[[], [-2, 3]]

L4 nested_sum: total = 0
item=[[], [-2, 3]]; total=0

L5 nested_sum: for child in item:
item=[[], [-2, 3]]; total=0; child=[]

L2 nested_sum: if isinstance(item, int):
item=[]

L4 nested_sum: total = 0
item=[]; total=0

L5 nested_sum: for child in item:
item=[[], [-2, 3]]; total=0; child=[-2, 3]

L2 nested_sum: if isinstance(item, int):
item=[-2, 3]

L4 nested_sum: total = 0
item=[-2, 3]; total=0

... 3 intermediate state changes omitted ...

L5 nested_sum: for child in item:
item=[-2, 3]; total=-2; child=3

L2 nested_sum: if isinstance(item, int):
item=3

L6 nested_sum: total += nested_sum(child)
item=[-2, 3]; total=1; child=3

L6 nested_sum: total += nested_sum(child)
item=[[], [-2, 3]]; total=1; child=[-2, 3]
```

Result: 1. Empty lists contribute zero and signs are retained.

Why the solution is correct

Integer leaves are correct directly. If every child call returns its own leaf sum, their sum is exactly the parent list's leaf sum because child occurrences partition that structure. Finite acyclic nesting guarantees termination at leaves or empty lists.

Time and space

O(N) time for N visited integer/list nodes and O(d) call-stack space for maximum nesting depth d.

Common mistakes

Summing only the top level ignores nested leaves. Sharing one global subtotal without a clear contract can double-add child results.

Reasoning card

State: Each call returns the sum of exactly its supplied subtree. total accumulates completed child results for a list node.

Invariant: Each call returns the sum of exactly its own nested value.

Decision to rehearse: A value is either a base integer or a recursive container.

Finish: Return the accumulated total

Transfer exercise

Return both the sum and the number of leaves in one recursive result tuple.

143. Flatten a Nested List

Foundation | Module 15

Problem

Flatten a finite acyclic nested structure of integers and lists into a new list of integer leaves in left-to-right order. Empty lists add no values. A single integer becomes a singleton list. Do not mutate the nested input.

Examples

Example 1. Depth-first left-to-right visitation preserves leaf order.

```
flatten_nested([1,[2,[3]],4])
=> [1, 2, 3, 4]
```

Example 2. Empty containers append nothing.

```
flatten_nested([[],[]])
=> []
```

Example 3. A root integer becomes one output leaf.

```
flatten_nested(5)
=> [5]
```

Why this mechanism works

The recursive traversal order determines output order. Pass through lists in their existing order and append an integer immediately when reached. One shared output accumulator avoids repeatedly concatenating growing intermediate lists.

State and boundaries

visit(item) appends exactly that item's leaves to output, preserving order. Each call owns traversal control; output is shared append-only state.

Protocol

1. Create an empty output.
2. On an integer append it.
3. On a list recursively visit children from left to right.
4. Return output after visiting the root.

Visualization

```
[1,[2,[3]],4]
visit leaves: 1 -> 2 -> 3 -> 4
append-only output: [1,2,3,4]
```

Python implementation

```
def flatten_nested(item):
    output = []
    def visit(current):
        if isinstance(current, int):
            output.append(current)
        else:
            for child in current:
                visit(child)
    visit(item)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
flatten_nested([1,[2,[3]],4])

L2 flatten_nested: output = []
item=[1, [2, [...]], 4]; output=[]

L4 visit: if isinstance(current, int):
current=[1, [2, [...]], 4]; output=[]

L7 visit: for child in current:
current=[1, [2, [...]], 4]; output=[]; child=1

L4 visit: if isinstance(current, int):
current=1; output=[]

L5 visit: output.append(current)
current=1; output=[1]

L8 visit: visit(child)
current=[1, [2, [...]], 4]; output=[1]; child=1

L7 visit: for child in current:
current=[1, [2, [...]], 4]; output=[1]; child=[2, [3]]

L4 visit: if isinstance(current, int):
current=[2, [3]]; output=[1]

... 13 intermediate state changes omitted ...

L4 visit: if isinstance(current, int):
current=4; output=[1, 2, 3]

L5 visit: output.append(current)
current=4; output=[1, 2, 3, 4]

L8 visit: visit(child)
current=[1, [2, [...]], 4]; output=[1, 2, 3, 4]; child=4
```

```
L9 flatten_nested: visit(item)
item=[1, [2, [...]], 4]; output=[1, 2, 3, 4]
```

Result: [1, 2, 3, 4]. Depth-first left-to-right visitation preserves leaf order.

```
flatten_nested([[],[]])

L2 flatten_nested: output = []
item=[[], []]; output=[]

L4 visit: if isinstance(current, int):
current=[[], []]; output=[]

L7 visit: for child in current:
current=[[], []]; output=[]; child=[]

L4 visit: if isinstance(current, int):
current=[]; output=[]

L4 visit: if isinstance(current, int):
current=[]; output=[]
```

Result: []. Empty containers append nothing.

Why the solution is correct

An integer appends its single correct leaf. A list visits each child in order, and each child appends its own ordered leaves by the call contract. Their concatenation is precisely the list's flattened leaf order.

Time and space

O(N) traversal time, O(L) output space for L integer leaves and O(d) call-stack space.

Common mistakes

Appending a nested list directly leaves the structure unflattened. Returning and repeatedly adding lists can cause avoidable copying.

Reasoning card

State: visit(item) appends exactly that item's leaves to output, preserving order. Each call owns traversal control; output is shared append-only state.

Invariant: Returned list is the complete flattening of the current object.

Decision to rehearse: Concatenate child results in left-to-right order.

Finish: Return output after visiting the root

Transfer exercise

Implement the traversal with an explicit stack, pushing children in reverse order to preserve left-to-right output.

144. Reverse a String Recursively

Foundation | Module 15

Problem

Return a string reversed through a recursive algorithm. Strings are treated as sequences of Python characters, without Unicode grapheme normalization. Empty input returns an empty string. Use a mutable character buffer so recursion need not create a new suffix string at every level.

Examples

Example 1. Each recursive level finalizes one mirrored pair.

```
recursive_reverse("abcd")
=> 'dcba'
```

Example 2. The initial interval is empty.

```
recursive_reverse("")
=> ''
```

Example 3. The middle character is its own mirror.

```
recursive_reverse("abc")
=> 'cba'
```

Why this mechanism works

Define the helper contract on an inclusive interval of the character buffer: reverse that interval. Swap its two endpoints, which have fixed final destinations, then delegate the smaller interior. The recursion stops once fewer than two positions remain.

State and boundaries

chars is the mutable output buffer. reverse(left,right) owns the interval it must reverse; outside positions are already final.

Protocol

1. Copy characters into a list.
2. Stop the helper when left>=right.
3. Swap endpoints.
4. Recursively reverse the interior.
5. Join the buffer into the returned string.

Visualization

```
abcd -> swap a,d -> dbca
reverse interior bc -> cb
result dcba
```

Python implementation

```
def recursive_reverse(text):
    chars = list(text)
    def reverse(left, right):
        if left >= right:
            return
        chars[left], chars[right] = chars[right], chars[left]
        reverse(left + 1, right - 1)
    reverse(0, len(chars) - 1)
    return ''.join(chars)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
recursive_reverse("abcd")

L2 recursive_reverse: chars = list(text)
text='abcd'; chars=['a', 'b', 'c', 'd']

L4 reverse: if left >= right:
left=0; right=3; chars=['a', 'b', 'c', 'd']

L6 reverse: chars[left], chars[right] = chars[right], chars[left]
left=0; right=3; chars=['d', 'b', 'c', 'a']

L4 reverse: if left >= right:
left=1; right=2; chars=['d', 'b', 'c', 'a']

L6 reverse: chars[left], chars[right] = chars[right], chars[left]
left=1; right=2; chars=['d', 'c', 'b', 'a']

L4 reverse: if left >= right:
left=2; right=1; chars=['d', 'c', 'b', 'a']

L7 reverse: reverse(left + 1, right - 1)
left=0; right=3; chars=['d', 'c', 'b', 'a']

L8 recursive_reverse: reverse(0, len(chars) - 1)
text='abcd'; chars=['d', 'c', 'b', 'a']
```

Result: 'dcba'. Each recursive level finalizes one mirrored pair.

```
recursive_reverse("")

L2 recursive_reverse: chars = list(text)
text=''; chars=[]

L4 reverse: if left >= right:
left=0; right=-1; chars=[]
```

Result: ''. The initial interval is empty.

Why the solution is correct

The base interval is empty or a singleton and is already reversed. For a larger interval, swapping endpoints puts them in their mirrored destinations, while the recursive call correctly reverses the remaining interior. Induction on interval size proves the whole reversal.

Time and space

O(n) time and O(n) space for the character buffer and linear-depth call stack.

Common mistakes

Recursing on text[1:] copies shrinking suffixes and can use quadratic aggregate storage. A string itself cannot be mutated at an index.

Reasoning card

State: chars is the mutable output buffer. reverse(left,right) owns the interval it must reverse; outside positions are already final.

Invariant: Each call returns the reversed form of its own substring.

Decision to rehearse: Solve the suffix and place the first character appropriately.

Finish: Join the buffer into the returned string

Transfer exercise

Explain why the iterative two-pointer version uses constant auxiliary state while this recursive version still has linear call depth.

145. Fast Exponentiation

Bridge | Module 15

Problem

Compute x raised to integer exponent n using recursive exponentiation by squaring. Return a floating-point result. Negative exponents require x!=0; define x**0 as 1.0, including x=0. Ordinary floating-point rounding and range limits apply.

Examples

Example 1. Repeated squaring replaces ten linear multiplications with logarithmic depth.

```
fast_power(2, 10)
=> 1024.0
```

Example 2. The reciprocal base handles a negative exponent.

```
fast_power(2, -3)
=> 0.125
```

Example 3. The contract explicitly uses the exponent-zero identity.

```
fast_power(0, 0)
=> 1.0
```

Why this mechanism works

If the exponent is even, x^n=(x^(n//2))^2. If odd, multiply that square by one additional x. Compute the half power once and reuse it; making two identical recursive calls would undo the logarithmic work reduction. A negative exponent first inverts the base.

State and boundaries

Each call returns the requested power. half is the single result of the exponent-halved child call.

Protocol

1. For a negative exponent invert x and negate n.
2. Return 1.0 at exponent zero.
3. Recursively compute the half power once.
4. Square it.
5. Multiply by x only for an odd exponent.

Visualization

```
2^10 -> (2^5)^2
2^5 -> (2^2)^2 * 2
2^2 -> (2^1)^2
one child call per level
```

Python implementation

```
def fast_power(x, n):
    if n < 0:
        return fast_power(1 / x, -n)
    if n == 0:
        return 1.0
    half = fast_power(x, n // 2)
    result = half * half
    return result * x if n % 2 else result
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
fast_power(2, 10)

L2 fast_power: if n < 0:
x=2; n=10

L2 fast_power: if n < 0:
x=2; n=5

L2 fast_power: if n < 0:
x=2; n=2

L2 fast_power: if n < 0:
x=2; n=1

L2 fast_power: if n < 0:
x=2; n=0

L6 fast_power: half = fast_power(x, n // 2)
x=2; n=1; half=1.0

L7 fast_power: result = half * half
x=2; n=1; half=1.0; result=1.0

L6 fast_power: half = fast_power(x, n // 2)
x=2; n=2; half=2.0

... 1 intermediate state changes omitted ...

L6 fast_power: half = fast_power(x, n // 2)
x=2; n=5; half=4.0

L7 fast_power: result = half * half
x=2; n=5; half=4.0; result=16.0

L6 fast_power: half = fast_power(x, n // 2)
x=2; n=10; half=32.0
```

```
L7 fast_power: result = half * half
x=2; n=10; half=32.0; result=1024.0
```

Result: 1024.0. Repeated squaring replaces ten linear multiplications with logarithmic depth.

```
fast_power(2, -3)

L2 fast_power: if n < 0:
x=2; n=-3

L2 fast_power: if n < 0:
x=0.5; n=3

L2 fast_power: if n < 0:
x=0.5; n=1

L2 fast_power: if n < 0:
x=0.5; n=0

L6 fast_power: half = fast_power(x, n // 2)
x=0.5; n=1; half=1.0

L7 fast_power: result = half * half
x=0.5; n=1; half=1.0; result=1.0

L6 fast_power: half = fast_power(x, n // 2)
x=0.5; n=3; half=0.5

L7 fast_power: result = half * half
x=0.5; n=3; half=0.5; result=0.25
```

Result: 0.125. The reciprocal base handles a negative exponent.

Why the solution is correct

The zero exponent is the multiplicative identity. The even and odd exponent identities combine a correct smaller power into the requested one. Halving a positive exponent reaches zero, and the reciprocal identity handles negative exponents.

Time and space

O(log(|n|+1)) multiplications and call-stack depth, excluding floating-point implementation details.

Common mistakes

Calling the same half-power recursion twice creates a much larger call tree. Negating n without inverting x computes the wrong negative power.

Reasoning card

State: Each call returns the requested power. half is the single result of the exponent-halved child call.

Invariant: The returned value equals the requested exponent for that call.

Decision to rehearse: Square a half-power rather than multiplying n times.

Finish: Multiply by x only for an odd exponent

Transfer exercise

Compute modular powers with integer arithmetic and reduce after each multiplication; discuss why negative exponents need a modular inverse.

146. Merge Sort

Bridge | Module 15

Problem

Return a stable nondecreasing copy of an integer list using merge sort. Leave the input unchanged. Empty and singleton lists return new lists with the same values. Stability means equal values retain source order when carrying records with keys.

Examples

Example 1. Sorted child results are merged.

```
merge_sort([4,1,3,2])
=> [1, 2, 3, 4]
```

Example 2. The empty list is a base case.

```
merge_sort([])
=> []
```

Example 3. Multiplicity is preserved across the split.

```
merge_sort([2,1,2])
=> [1, 2, 2]
```

Why this mechanism works

A call promises a sorted version of its entire input segment. Divide into two smaller segments, let each recursive call fulfill that promise, then combine them with the stable two-frontier merge from kata 021. The merge is valid because the recursive results are already sorted.

State and boundaries

left and right are complete sorted child results. mid defines the split; neither child includes the other's positions.

Protocol

1. Return a copy for length<=1.
2. Split at the midpoint.
3. Recursively sort both halves.
4. Merge the two sorted results stably.
5. Return the merged list.

Visualization

```
[4,1,3,2]
[4,1] | [3,2]
[1,4] | [2,3]
merge -> [1,2,3,4]
```

Python implementation

```
from kata_021 import merge_sorted

def merge_sort(nums):
    if len(nums) <= 1:
        return list(nums)
    mid = len(nums) // 2
    left = merge_sort(nums[:mid])
    right = merge_sort(nums[mid:])
    return merge_sorted(left, right)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
merge_sort([4,1,3,2])

L4 merge_sort: if len(nums) <= 1:
nums=[4, 1, 3, 2]

L6 merge_sort: mid = len(nums) // 2
nums=[4, 1, 3, 2]; mid=2

L4 merge_sort: if len(nums) <= 1:
nums=[4, 1]

L6 merge_sort: mid = len(nums) // 2
nums=[4, 1]; mid=1

L4 merge_sort: if len(nums) <= 1:
nums=[4]

L7 merge_sort: left = merge_sort(nums[:mid])
nums=[4, 1]; mid=1; left=[4]

L4 merge_sort: if len(nums) <= 1:
nums=[1]

L8 merge_sort: right = merge_sort(nums[mid:])
nums=[4, 1]; mid=1; left=[4]; right=[1]

... 4 intermediate state changes omitted ...

L7 merge_sort: left = merge_sort(nums[:mid])
nums=[3, 2]; mid=1; left=[3]

L4 merge_sort: if len(nums) <= 1:
nums=[2]

L8 merge_sort: right = merge_sort(nums[mid:])
nums=[3, 2]; mid=1; left=[3]; right=[2]
```

```
L8 merge_sort: right = merge_sort(nums[mid:])
nums=[4, 1, 3, 2]; mid=2; left=[1, 4]; right=[2, 3]
```

Result: [1, 2, 3, 4]. Sorted child results are merged.

```
merge_sort([])

L4 merge_sort: if len(nums) <= 1:
nums=[]
```

Result: []. The empty list is a base case.

Why the solution is correct

Singletons and empty lists are sorted. Assuming each smaller call returns a sorted permutation of its input, stable merge returns a sorted permutation of both halves together while preserving equal-key order. Induction on input length establishes the result.

Time and space

O(n log n) time and O(n) peak auxiliary storage for slices and merged results, plus O(log n) call depth.

Common mistakes

Merging before the child calls finish violates the merge precondition. A non-strict equality choice in the wrong direction can break stability for records.

Reasoning card

State: left and right are complete sorted child results. mid defines the split; neither child includes the other's positions.

Invariant: Each call returns a sorted permutation of its input slice.

Decision to rehearse: Combine two sorted child results.

Finish: Return the merged list

Transfer exercise

Count inversions during merging: when a right value is chosen before remaining left values, how many inversions does it create?

147. Quicksort Partition

Bridge | Module 15

Problem

Sort a mutable integer list in place using quicksort with an explicit partition around a pivot. Return the same list. Use half-open recursive ranges. Duplicates are allowed, but this two-way partition can become inefficient when many values equal the pivot.

Examples

Example 1. Partition fixes one pivot before recursing on independent ranges.

```
quick_sort([4,1,3,2])
=> [1, 2, 3, 4]
```

Example 2. The output is correct despite unbalanced equal-value partitions.

```
quick_sort([2,2,2])
=> [2, 2, 2]
```

Example 3. The initial range is already a base case.

```
quick_sort([])
=> []
```

Why this mechanism works

Move a randomly selected pivot to the final cell of the current range. A write frontier separates values <=pivot from larger values while scanning the rest. The final pivot swap places it between those regions, after which recursive calls exclude its fixed position. Random pivots reduce consistent imbalance on distinct data, but do not solve the all-equal case for this partition.

State and boundaries

During partition, [left,write) is <=pivot and [write,i) is >pivot. After the final swap, write is the pivot's final index.

Protocol

1. Stop ranges of length<=1.
2. Choose and move a pivot to right-1.
3. Scan and compact <=pivot values before write.
4. Swap the pivot into write.
5. Recurse on [left,write) and [write+1,right).

Visualization

```
[ <= pivot | > pivot | pivot ]
            write       right-1
final swap fixes pivot at write
recurse on the two sides only
```

Python implementation

```
import random

def quick_sort(nums):
    def sort(left, right):
        if right - left <= 1:
            return
        chosen = random.randrange(left, right)
        nums[chosen], nums[right - 1] = nums[right - 1], nums[chosen]
        pivot = nums[right - 1]
        write = left
        for i in range(left, right - 1):
            if nums[i] <= pivot:
                nums[write], nums[i] = nums[i], nums[write]
                write += 1
        nums[write], nums[right - 1] = nums[right - 1], nums[write]
        sort(left, write)
        sort(write + 1, right)
    sort(0, len(nums))
    return nums
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
quick_sort([4,1,3,2])

L4 quick_sort: def sort(left, right):
nums=[4, 1, 3, 2]

L5 sort: if right - left <= 1:
left=0; right=4; nums=[4, 1, 3, 2]

L7 sort: chosen = random.randrange(left, right)
left=0; right=4; nums=[4, 1, 3, 2]; chosen=0

L8 sort: nums[chosen], nums[right - 1] = nums[right - 1], nums[chosen]
left=0; right=4; nums=[2, 1, 3, 4]; chosen=0

L9 sort: pivot = nums[right - 1]
left=0; right=4; nums=[2, 1, 3, 4]; chosen=0; pivot=4

L10 sort: write = left
left=0; right=4; nums=[2, 1, 3, 4]; chosen=0; pivot=4; write=0

L11 sort: for i in range(left, right - 1):
left=0; right=4; nums=[2, 1, 3, 4]; chosen=0; pivot=4; write=0; i=0

L14 sort: write += 1
left=0; right=4; nums=[2, 1, 3, 4]; chosen=0; pivot=4; write=1; i=0
```

```
... 15 intermediate state changes omitted ...

L5 sort: if right - left <= 1:
left=2; right=3; nums=[1, 2, 3, 4]

L16 sort: sort(left, write)
left=0; right=4; nums=[1, 2, 3, 4]; chosen=0; pivot=4; write=3; i=2

L5 sort: if right - left <= 1:
left=4; right=4; nums=[1, 2, 3, 4]

L18 quick_sort: sort(0, len(nums))
nums=[1, 2, 3, 4]
```

Result: [1, 2, 3, 4]. Partition fixes one pivot before recursing on independent ranges.

```
quick_sort([2,2,2])

L4 quick_sort: def sort(left, right):
nums=[2, 2, 2]

L5 sort: if right - left <= 1:
left=0; right=3; nums=[2, 2, 2]

L7 sort: chosen = random.randrange(left, right)
left=0; right=3; nums=[2, 2, 2]; chosen=2

L9 sort: pivot = nums[right - 1]
left=0; right=3; nums=[2, 2, 2]; chosen=2; pivot=2

L10 sort: write = left
left=0; right=3; nums=[2, 2, 2]; chosen=2; pivot=2; write=0

L11 sort: for i in range(left, right - 1):
left=0; right=3; nums=[2, 2, 2]; chosen=2; pivot=2; write=0; i=0

L14 sort: write += 1
left=0; right=3; nums=[2, 2, 2]; chosen=2; pivot=2; write=1; i=0

L11 sort: for i in range(left, right - 1):
left=0; right=3; nums=[2, 2, 2]; chosen=2; pivot=2; write=1; i=1

... 6 intermediate state changes omitted ...

L14 sort: write += 1
left=0; right=2; nums=[2, 2, 2]; chosen=0; pivot=2; write=1; i=0

L5 sort: if right - left <= 1:
left=0; right=1; nums=[2, 2, 2]

L5 sort: if right - left <= 1:
left=2; right=2; nums=[2, 2, 2]

L5 sort: if right - left <= 1:
left=3; right=3; nums=[2, 2, 2]
```

Result: [2, 2, 2]. The output is correct despite unbalanced equal-value partitions.

Why the solution is correct

The partition loop preserves its two classified regions, and the final swap leaves every left value <=pivot and every right value >pivot. Recursive sorting of these disjoint regions therefore sorts the whole range while the pivot stays fixed. Each child range is strictly smaller.

Time and space

Expected O(n log n) time on distinct values with random pivots; O(n^2) worst case, including all-equal values here. Expected O(log n), worst O(n) call depth; no second array is allocated.

Common mistakes

Including the pivot again in a recursive child can prevent progress. Storing only a pivot index while swapping cells can accidentally change which value is the pivot.

Reasoning card

State: During partition, [left,write) is <=pivot and [write,i) is >pivot. After the final swap, write is the pivot's final index.

Invariant: After partition, left values obey one relation and right values another.

Decision to rehearse: Move the pivot into its final position before recursing.

Finish: Recurse on [left,write) and [write+1,right)

Transfer exercise

Replace the two-way partition with a three-way less/equal/greater partition and skip the entire equal region in recursion.

148. Quickselect Kth Statistic

Interview | Module 15

Problem

Return the kth smallest integer in nums, or kth largest when largest=True, counting duplicates as separate occurrences. k is one-based and satisfies 1<=k<=n. Do not fully sort or mutate the input; work on a copy with randomized three-way partitioning.

Examples

Example 1. The second largest has ascending rank n-2.

```
quickselect([3,2,1,5,6,4], 2, largest=True)
=> 5
```

Example 2. Duplicates occupy separate ranks inside the equal band.

```
quickselect([2,1,2,3], 3)
=> 2
```

Example 3. The entire candidate interval is one equal band.

```
quickselect([7], 1)
=> 7
```

Why this mechanism works

Selection needs only the partition containing the target rank, not both sorted sides. Three-way partitioning identifies an entire equal-pivot rank band. If the target falls inside that band, its value is known immediately; otherwise discard the band and the irrelevant side.

State and boundaries

target is a zero-based rank in the copied array. [left,right) contains that rank's unresolved region. [low,high) is the equal-pivot band after partition.

Protocol

1. Translate k to an ascending target rank.
2. Choose a random pivot value in the candidate interval.
3. Partition into less, equal and greater regions.
4. Keep only the side containing target.
5. Return pivot when target lies in the equal region.

Visualization

```
[ values < p | values == p | values > p ]
             low          high
target in [low,high) -> answer p
```

Python implementation

```
import random

def quickselect(nums, k, largest=False):
    values = list(nums)
    target = len(values) - k if largest else k - 1
    left, right = 0, len(values)
    while left < right:
        pivot = values[random.randrange(left, right)]
        low = scan = left
        high = right
        while scan < high:
            if values[scan] < pivot:
                values[low], values[scan] = values[scan], values[low]
                low += 1
                scan += 1
            elif values[scan] > pivot:
                high -= 1
                values[scan], values[high] = values[high], values[scan]
            else:
                scan += 1
        if target < low:
            right = low
        elif target >= high:
            left = high
        else:
            return pivot
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
quickselect([3,2,1,5,6,4], 2, largest=True)

L4 quickselect: values = list(nums)
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 5, 6, 4]

L5 quickselect: target = len(values) - k if largest else k - 1
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 5, 6, 4]; target=4

L6 quickselect: left, right = 0, len(values)
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 5, 6, 4]; target=4; left=0;
    right=6

L8 quickselect: pivot = values[random.randrange(left, right)]
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 5, 6, 4]; target=4; left=0;
    right=6; pivot=4

L9 quickselect: low = scan = left
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 5, 6, 4]; target=4; left=0;
    right=6; pivot=4; low=0; scan=0
```

```
L10 quickselect: high = right
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 5, 6, 4]; target=4; left=0;
    right=6; pivot=4; low=0; scan=0; high=6

L14 quickselect: low += 1
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 5, 6, 4]; target=4; left=0;
    right=6; pivot=4; low=1; scan=0; high=6

L15 quickselect: scan += 1
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 5, 6, 4]; target=4; left=0;
    right=6; pivot=4; low=1; scan=1; high=6

... 17 intermediate state changes omitted ...

L8 quickselect: pivot = values[random.randrange(left, right)]
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 4, 5, 6]; target=4; left=4;
    right=5; pivot=5; low=5; scan=6; high=6

L9 quickselect: low = scan = left
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 4, 5, 6]; target=4; left=4;
    right=5; pivot=5; low=4; scan=4; high=6

L10 quickselect: high = right
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 4, 5, 6]; target=4; left=4;
    right=5; pivot=5; low=4; scan=4; high=5

L20 quickselect: scan += 1
nums=[3, 2, 1, 5, 6, 4]; k=2; largest=True; values=[3, 2, 1, 4, 5, 6]; target=4; left=4;
    right=5; pivot=5; low=4; scan=5; high=5
```

Result: 5. The second largest has ascending rank n-2.

```
quickselect([2,1,2,3], 3)

L4 quickselect: values = list(nums)
nums=[2, 1, 2, 3]; k=3; largest=False; values=[2, 1, 2, 3]

L5 quickselect: target = len(values) - k if largest else k - 1
nums=[2, 1, 2, 3]; k=3; largest=False; values=[2, 1, 2, 3]; target=2

L6 quickselect: left, right = 0, len(values)
nums=[2, 1, 2, 3]; k=3; largest=False; values=[2, 1, 2, 3]; target=2; left=0; right=4

L8 quickselect: pivot = values[random.randrange(left, right)]
nums=[2, 1, 2, 3]; k=3; largest=False; values=[2, 1, 2, 3]; target=2; left=0; right=4; pivot=2

L9 quickselect: low = scan = left
nums=[2, 1, 2, 3]; k=3; largest=False; values=[2, 1, 2, 3]; target=2; left=0; right=4;
    pivot=2; low=0; scan=0

L10 quickselect: high = right
nums=[2, 1, 2, 3]; k=3; largest=False; values=[2, 1, 2, 3]; target=2; left=0; right=4;
    pivot=2; low=0; scan=0; high=4

L20 quickselect: scan += 1
nums=[2, 1, 2, 3]; k=3; largest=False; values=[2, 1, 2, 3]; target=2; left=0; right=4;
    pivot=2; low=0; scan=1; high=4

L13 quickselect: values[low], values[scan] = values[scan], values[low]
nums=[2, 1, 2, 3]; k=3; largest=False; values=[1, 2, 2, 3]; target=2; left=0; right=4;
    pivot=2; low=0; scan=1; high=4

L14 quickselect: low += 1
nums=[2, 1, 2, 3]; k=3; largest=False; values=[1, 2, 2, 3]; target=2; left=0; right=4;
```

```
    pivot=2; low=1; scan=1; high=4

L15 quickselect: scan += 1
nums=[2, 1, 2, 3]; k=3; largest=False; values=[1, 2, 2, 3]; target=2; left=0; right=4;
    pivot=2; low=1; scan=2; high=4

L20 quickselect: scan += 1
nums=[2, 1, 2, 3]; k=3; largest=False; values=[1, 2, 2, 3]; target=2; left=0; right=4;
    pivot=2; low=1; scan=3; high=4

L17 quickselect: high -= 1
nums=[2, 1, 2, 3]; k=3; largest=False; values=[1, 2, 2, 3]; target=2; left=0; right=4;
    pivot=2; low=1; scan=3; high=3
```

Result: 2. Duplicates occupy separate ranks inside the equal band.

Why the solution is correct

Partition places all smaller values before the equal band and all larger values after it, establishing the band's exact rank interval within the unresolved range. The target value must lie in the selected band or side, so discarding other regions is safe. The retained region strictly shrinks unless the answer is returned.

Time and space

Expected O(n) time with randomized pivots and O(n^2) worst case; O(n) copy storage and O(1) partition state. Three-way partition handles duplicate bands directly.

Common mistakes

Using k as a zero-based index shifts the rank. Recursing into both sides turns selection back into sorting work.

Reasoning card

State: target is a zero-based rank in the copied array. [left,right) contains that rank's unresolved region. [low,high) is the equal-pivot band after partition.

Invariant: The target rank remains in the chosen partition.

Decision to rehearse: Partition and recurse only into the side containing the target rank.

Finish: Return pivot when target lies in the equal region

Transfer exercise

Select several quantiles together and discuss when reusing partitions is better than running independent selections.

149. Tower of Hanoi

Interview | Module 15

Problem

Generate the legal moves for n disks in the Tower of Hanoi, moving the stack from peg A to peg C using peg B. Return a list of (source,target) moves. n is nonnegative; zero disks requires no moves. Only one top disk may move at a time, never onto a smaller disk.

Examples

Example 1. Seven legal moves transfer three disks.

```
hanoi(3)
=> [('A', 'C'), ('A', 'B'), ('C', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('A',
    'C')]
```

Example 2. The largest disk is also the only disk.

```
hanoi(1)
=> [('A', 'C')]
```

Example 3. There is no disk to move.

```
hanoi(0)
=> []
```

Why this mechanism works

The largest disk cannot move until the n-1 smaller disks have been moved aside. Transfer them to the spare peg, move the largest disk to its destination, then transfer the smaller stack onto it. Each child call has the same contract with renamed peg roles.

State and boundaries

transfer(count,source,target,spare) appends moves that relocate exactly that legal stack while preserving disk order.

Protocol

1. Return immediately for zero disks.

2. Transfer count-1 disks from source to spare.

3. Append one source-to-target move for the largest disk.

4. Transfer count-1 disks from spare to target.

5. Return the accumulated move list.

Visualization

```
move n-1: A -> B using C
move largest: A -> C
move n-1: B -> C using A
```

Python implementation

```
def hanoi(n):
    moves = []
    def transfer(count, source, target, spare):
        if count == 0:
            return
        transfer(count - 1, source, spare, target)
        moves.append((source, target))
        transfer(count - 1, spare, target, source)
    transfer(n, 'A', 'C', 'B')
    return moves
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
hanoi(3)

L2 hanoi: moves = []
n=3; moves=[]

L4 transfer: if count == 0:
count=3; source='A'; target='C'; spare='B'; moves=[]

L4 transfer: if count == 0:
count=2; source='A'; target='B'; spare='C'; moves=[]

L4 transfer: if count == 0:
count=1; source='A'; target='C'; spare='B'; moves=[]

L4 transfer: if count == 0:
count=0; source='A'; target='B'; spare='C'; moves=[]

L7 transfer: moves.append((source, target))
count=1; source='A'; target='C'; spare='B'; moves=[('A', 'C')]

L4 transfer: if count == 0:
count=0; source='B'; target='C'; spare='A'; moves=[('A', 'C')]

L6 transfer: transfer(count - 1, source, spare, target)
count=2; source='A'; target='B'; spare='C'; moves=[('A', 'C')]

... 18 intermediate state changes omitted ...

L4 transfer: if count == 0:
count=0; source='B'; target='C'; spare='A'; moves=[('A', 'C'), ('A', 'B'), ('C', 'B'), ('A',
    'C'), ('B', 'A'), ('B', 'C'), ('A', 'C')]

L8 transfer: transfer(count - 1, spare, target, source)
count=2; source='B'; target='C'; spare='A'; moves=[('A', 'C'), ('A', 'B'), ('C', 'B'), ('A',
```

```
    'C'), ('B', 'A'), ('B', 'C'), ('A', 'C')]

L8 transfer: transfer(count - 1, spare, target, source)
count=3; source='A'; target='C'; spare='B'; moves=[('A', 'C'), ('A', 'B'), ('C', 'B'), ('A',
    'C'), ('B', 'A'), ('B', 'C'), ('A', 'C')]

L9 hanoi: transfer(n, 'A', 'C', 'B')
n=3; moves=[('A', 'C'), ('A', 'B'), ('C', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('A',
    'C')]
```

Result: [('A', 'C'), ('A', 'B'), ('C', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('A', 'C')]. Seven legal moves transfer three disks.

```
hanoi(1)

L2 hanoi: moves = []
n=1; moves=[]

L4 transfer: if count == 0:
count=1; source='A'; target='C'; spare='B'; moves=[]

L4 transfer: if count == 0:
count=0; source='A'; target='B'; spare='C'; moves=[]

L7 transfer: moves.append((source, target))
count=1; source='A'; target='C'; spare='B'; moves=[('A', 'C')]

L4 transfer: if count == 0:
count=0; source='B'; target='C'; spare='A'; moves=[('A', 'C')]

L9 hanoi: transfer(n, 'A', 'C', 'B')
n=1; moves=[('A', 'C')]
```

Result: [('A', 'C')]. The largest disk is also the only disk.

Why the solution is correct

By induction, the first child legally clears the largest disk, which can then move to the destination. The second child legally places the smaller stack on it. Any solution must clear and restore those smaller disks around the largest move, yielding the same 2*T(n-1)+1 lower bound met by this construction.

Time and space

Exactly 2^n-1 moves: O(2^n) time/output space and O(n) call-stack space. Materializing large n is inherently impractical.

Common mistakes

Peg names change roles between child calls. Forgetting to exchange the spare and target roles produces illegal moves.

Reasoning card

State: transfer(count,source,target,spare) appends moves that relocate exactly that legal stack while preserving disk order.

Invariant: Each call legally transfers its top n-disk stack.

Decision to rehearse: Move n-1 aside, move largest, move n-1 onto it.

Finish: Return the accumulated move list

Transfer exercise

Write a simulator that rejects illegal moves and verifies the final peg contents for every generated small-n sequence.

150. Merge K Sorted Arrays by Divide-and-Conquer

Interview | Module 15

Problem

Merge k nondecreasing integer lists into a new sorted list using balanced divide-and-conquer rather than a heap. Preserve all occurrences and leave inputs unchanged. Empty source lists and an empty collection are allowed.

Examples

Example 1. Balanced source groups are merged recursively.

```
merge_k_divide([[1,4],[2,3],[0,5]])
=> [0, 1, 2, 3, 4, 5]
```

Example 2. Empty sources contribute no values.

```
merge_k_divide([[],[2,2],[]])
=> [2, 2]
```

Example 3. The initial source interval is empty.

```
merge_k_divide([])
=> []
```

Why this mechanism works

Define a call over a half-open range of source lists. It returns the sorted merge of exactly those sources. Split that range in half, obtain two sorted child results, and merge them with kata 021. Balanced splitting prevents repeatedly merging a growing result with one small next list.

State and boundaries

merge_range(left,right) owns source indices [left,right). Its returned list contains every occurrence from those sources in sorted order.

Protocol

1. Return [] for an empty source range.

2. Copy the sole source for a singleton range.

3. Split the source range at its midpoint.

4. Recursively merge both halves.

5. Merge the two child outputs.

Visualization

```
sources A B C D
merge(A,B) | merge(C,D)
          merge both results
logarithmic merge levels
```

Python implementation

```
from kata_021 import merge_sorted

def merge_k_divide(arrays):
    def merge_range(left, right):
        if left == right:
            return []
        if right - left == 1:
            return list(arrays[left])
        mid = (left + right) // 2
        a = merge_range(left, mid)
        b = merge_range(mid, right)
        return merge_sorted(a, b)
    return merge_range(0, len(arrays))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
merge_k_divide([[1,4],[2,3],[0,5]])

L4 merge_k_divide: def merge_range(left, right):
arrays=[[1, 4], [2, 3], [0, 5]]

L5 merge_range: if left == right:
left=0; right=3; arrays=[[1, 4], [2, 3], [0, 5]]

L9 merge_range: mid = (left + right) // 2
left=0; right=3; arrays=[[1, 4], [2, 3], [0, 5]]; mid=1

L5 merge_range: if left == right:
left=0; right=1; arrays=[[1, 4], [2, 3], [0, 5]]

L10 merge_range: a = merge_range(left, mid)
left=0; right=3; arrays=[[1, 4], [2, 3], [0, 5]]; mid=1; a=[1, 4]

L5 merge_range: if left == right:
left=1; right=3; arrays=[[1, 4], [2, 3], [0, 5]]

L9 merge_range: mid = (left + right) // 2
left=1; right=3; arrays=[[1, 4], [2, 3], [0, 5]]; mid=2

L5 merge_range: if left == right:
left=1; right=2; arrays=[[1, 4], [2, 3], [0, 5]]

L10 merge_range: a = merge_range(left, mid)
left=1; right=3; arrays=[[1, 4], [2, 3], [0, 5]]; mid=2; a=[2, 3]

L5 merge_range: if left == right:
```

```
left=2; right=3; arrays=[[1, 4], [2, 3], [0, 5]]

L11 merge_range: b = merge_range(mid, right)
left=1; right=3; arrays=[[1, 4], [2, 3], [0, 5]]; mid=2; a=[2, 3]; b=[0, 5]

L11 merge_range: b = merge_range(mid, right)
left=0; right=3; arrays=[[1, 4], [2, 3], [0, 5]]; mid=1; a=[1, 4]; b=[0, 2, 3, 5]
```

Result: [0, 1, 2, 3, 4, 5]. Balanced source groups are merged recursively.

```
merge_k_divide([[],[2,2],[]])

L4 merge_k_divide: def merge_range(left, right):
arrays=[[], [2, 2], []]

L5 merge_range: if left == right:
left=0; right=3; arrays=[[], [2, 2], []]

L9 merge_range: mid = (left + right) // 2
left=0; right=3; arrays=[[], [2, 2], []]; mid=1

L5 merge_range: if left == right:
left=0; right=1; arrays=[[], [2, 2], []]

L10 merge_range: a = merge_range(left, mid)
left=0; right=3; arrays=[[], [2, 2], []]; mid=1; a=[]

L5 merge_range: if left == right:
left=1; right=3; arrays=[[], [2, 2], []]

L9 merge_range: mid = (left + right) // 2
left=1; right=3; arrays=[[], [2, 2], []]; mid=2

L5 merge_range: if left == right:
left=1; right=2; arrays=[[], [2, 2], []]

L10 merge_range: a = merge_range(left, mid)
left=1; right=3; arrays=[[], [2, 2], []]; mid=2; a=[2, 2]

L5 merge_range: if left == right:
left=2; right=3; arrays=[[], [2, 2], []]

L11 merge_range: b = merge_range(mid, right)
left=1; right=3; arrays=[[], [2, 2], []]; mid=2; a=[2, 2]; b=[]

L11 merge_range: b = merge_range(mid, right)
left=0; right=3; arrays=[[], [2, 2], []]; mid=1; a=[]; b=[2, 2]
```

Result: [2, 2]. Empty sources contribute no values.

Why the solution is correct

The empty and singleton source ranges satisfy the contract directly. If both child outputs correctly merge their disjoint source subsets, merging them produces the correct sorted union for the parent range. Balanced source partitioning establishes the result for all k lists.

Time and space

O(N log(k+1)+k) time for N total values; O(N+k) conservative peak storage and O(log(k+1)) call depth.

Common mistakes

Sequentially merging a growing accumulator may repeatedly copy early values across many stages. Splitting by source indices avoids unnecessary slicing of the outer list.

Reasoning card

State: merge_range(left,right) owns source indices [left,right). Its returned list contains every occurrence from those sources in sorted order.

Invariant: Every intermediate array is sorted and contains its source elements.

Decision to rehearse: Reduce the number of sorted sources by half each round.

Finish: Merge the two child outputs

Transfer exercise

Compare heap-frontier merging with divide-and-conquer when sources are live streams that cannot be reread or fully materialized.

Module 16 Backtracking and Combinatorial Search

Represent a partial candidate, enumerate legal choices, restore shared state, and prune impossible branches.

Backtracking explores alternative histories. Unlike permanent graph visitation, a chosen value or reserved cell belongs to one path and must be restored before its sibling starts. Use choose, explore, undo as an ownership protocol. Pruning is justified only when the rejected prefix cannot lead to a valid completed answer.

The ten-step progression

151. Generate Binary Strings

152. Generate All Subsets

153. Choose K Combinations

154. Generate Permutations

155. Generate Unique Permutations

156. Combination Sum

157. Generate Balanced Parentheses

158. Word Search in Grid

159. Palindrome Partitioning

160. N-Queens

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

151. Generate Binary Strings

Foundation | Module 16

Problem

Return every binary string of length n in lexicographic order, for integer n>=0. Each position contains '0' or '1'. There is one length-zero string, so n=0 returns ['']. The result must contain independent completed strings.

Examples

Example 1. Each of two positions branches twice.

```
binary_strings(2)
=> ['00', '01', '10', '11']
```

Example 2. The empty prefix is already complete.

```
binary_strings(0)
=> ['']
```

Example 3. The two single-position choices are emitted in order.

```
binary_strings(1)
=> ['0', '1']
```

Why this mechanism works

Treat a recursive call as a promise to complete the current prefix. At every unfinished position there are exactly two choices. Append one choice, explore every completion beneath it, then remove that choice before exploring its sibling. This choose-explore-undo structure will become the common protocol for the entire chapter.

State and boundaries

path contains the chosen prefix. Its length is the next position to fill; output contains only complete strings.

Protocol

1. Start with an empty path.

2. At length n, join and save the path.

3. Otherwise append 0 and explore, then undo.

4. Append 1 and explore, then undo.

Visualization



Python implementation

```
def binary_strings(n):
    output, path = [], []
    def visit():
        if len(path) == n:
            output.append(''.join(path))
            return
        for bit in '01':
            path.append(bit)
            visit()
            path.pop()
    visit()
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
binary_strings(2)

L2 binary_strings: output, path = [], []
n=2; output=[]; path=[]

L4 visit: if len(path) == n:
n=2; output=[]; path=[]

L7 visit: for bit in '01':
n=2; output=[]; path=[]; bit='0'

L8 visit: path.append(bit)
n=2; output=[]; path=['0']; bit='0'

L4 visit: if len(path) == n:
n=2; output=[]; path=['0']

L7 visit: for bit in '01':
n=2; output=[]; path=['0']; bit='0'

L8 visit: path.append(bit)
n=2; output=[]; path=['0', '0']; bit='0'

L4 visit: if len(path) == n:
n=2; output=[]; path=['0', '0']

... 25 intermediate state changes omitted ...

L10 visit: path.pop()
n=2; output=['00', '01', '10', '11']; path=['1']; bit='1'

L9 visit: visit()
n=2; output=['00', '01', '10', '11']; path=['1']; bit='1'
```

```
L10 visit: path.pop()
n=2; output=['00', '01', '10', '11']; path=[]; bit='1'

L11 binary_strings: visit()
n=2; output=['00', '01', '10', '11']; path=[]
```

Result: ['00', '01', '10', '11']. Each of two positions branches twice.

```
binary_strings(0)

L2 binary_strings: output, path = [], []
n=0; output=[]; path=[]

L4 visit: if len(path) == n:
n=0; output=[]; path=[]

L5 visit: output.append(''.join(path))
n=0; output=['']; path=[]

L11 binary_strings: visit()
n=0; output=['']; path=[]
```

Result: ['']. The empty prefix is already complete.

Why the solution is correct

Every root-to-leaf route makes exactly n binary choices and produces one valid string. Distinct routes differ at some position, so there are no duplicates. Both choices are explored at each position, covering every string; choosing 0 first gives lexical order. Depth increases toward n.

Time and space

O(n*2^n) time and output characters for n>=1; O(n) auxiliary path and recursion depth. At n=0 the work and output are constant.

Common mistakes

Saving the mutable path itself aliases later changes. Forgetting pop leaves a sibling with the previous sibling's choices.

Reasoning card

State: path contains the chosen prefix. Its length is the next position to fill; output contains only complete strings.

Invariant: path contains one choice for every assigned position.

Decision to rehearse: Choose 0/1, recurse, unchoose.

Finish: Append 1 and explore, then undo

Transfer exercise

Forbid adjacent ones. State which next choice can be pruned using only the last chosen bit.

152. Generate All Subsets

Foundation | Module 16

Problem

Return all subsets of a list of distinct integers, retaining input order within each subset. Use depth-first order that emits the current subset before extending it with later indices. Empty input returns [[]]. Do not mutate nums.

Examples

Example 1. A path is emitted before its extensions.

```
subsets([1,2])
=> [[], [1], [1, 2], [2]]
```

Example 2. The empty set has one subset.

```
subsets([])
=> [[]]
```

Example 3. Either stop at the root or select the only value.

```
subsets([7])
=> [[], [7]]
```

Why this mechanism works

A subset is identified by its increasing sequence of chosen indices. The next index may be any position at or after start. Emit the current path even before it is extended: stopping now is itself a valid subset. Advancing start prevents reordered duplicates such as [2,1] after [1,2].

State and boundaries

path is a subset from strictly increasing indices. start is the first index still eligible for extension.

Protocol

1. Save a copy of path.

2. For each i from start onward, choose nums[i].

3. Recurse with start=i+1.

4. Undo that choice before trying the next i.

Visualization

```
[]
  choose index 0 -> [1] -> [1,2]
  undo to []
  choose index 1 -> [2]
```

Python implementation

```
def subsets(nums):
    output, path = [], []
    def visit(start):
        output.append(path.copy())
        for i in range(start, len(nums)):
            path.append(nums[i])
            visit(i + 1)
            path.pop()
    visit(0)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
subsets([1,2])

L2 subsets: output, path = [], []
nums=[1, 2]; output=[]; path=[]

L4 visit: output.append(path.copy())
start=0; nums=[1, 2]; output=[[]]; path=[]

L5 visit: for i in range(start, len(nums)):
start=0; nums=[1, 2]; output=[[]]; path=[]; i=0

L6 visit: path.append(nums[i])
start=0; nums=[1, 2]; output=[[]]; path=[1]; i=0

L4 visit: output.append(path.copy())
start=1; nums=[1, 2]; output=[[], [1]]; path=[1]

L5 visit: for i in range(start, len(nums)):
start=1; nums=[1, 2]; output=[[], [1]]; path=[1]; i=1

L6 visit: path.append(nums[i])
start=1; nums=[1, 2]; output=[[], [1]]; path=[1, 2]; i=1

L4 visit: output.append(path.copy())
start=2; nums=[1, 2]; output=[[], [1], [1, 2]]; path=[1, 2]

... 6 intermediate state changes omitted ...

L4 visit: output.append(path.copy())
start=2; nums=[1, 2]; output=[[], [1], [1, 2], [2]]; path=[2]

L7 visit: visit(i + 1)
start=0; nums=[1, 2]; output=[[], [1], [1, 2], [2]]; path=[2]; i=1

L8 visit: path.pop()
start=0; nums=[1, 2]; output=[[], [1], [1, 2], [2]]; path=[]; i=1
```

```
L9 subsets: visit(0)
nums=[1, 2]; output=[[], [1], [1, 2], [2]]; path=[]
```

Result: [[], [1], [1, 2], [2]]. A path is emitted before its extensions.

```
subsets([])

L2 subsets: output, path = [], []
nums=[]; output=[]; path=[]

L4 visit: output.append(path.copy())
start=0; nums=[]; output=[[]]; path=[]

L9 subsets: visit(0)
nums=[]; output=[[]]; path=[]
```

Result: [[]]. The empty set has one subset.

Why the solution is correct

The empty subset is emitted at the root. Every nonempty subset has a unique increasing index sequence, followed by exactly one route through the recursion. Emitting each route's prefix covers all subsets once. Eligible indices strictly advance, so recursion terminates.

Time and space

O(n*2^n) time and output space; O(n) auxiliary path and recursion depth.

Common mistakes

Reusing i instead of i+1 permits repeated selection. Appending path without copying changes previously saved answers.

Reasoning card

State: path is a subset from strictly increasing indices. start is the first index still eligible for extension.

Invariant: path reflects decisions made before the current index.

Decision to rehearse: Use include/exclude decisions or choose-next indexing.

Finish: Undo that choice before trying the next i

Transfer exercise

Allow duplicate input values and skip equal choices only at the same recursion depth.

153. Choose K Combinations

Foundation | Module 16

Problem

Return all increasing combinations of k distinct integers chosen from 1 through n, in lexicographic order. n and k are nonnegative. Return [[]] for k=0, and [] when k>n. Each answer has exactly k values.

Examples

Example 1. Increasing choices prevent reorderings of the same combination.

```
combinations(4,2)
=> [[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

Example 2. Zero selections form one combination.

```
combinations(3,0)
=> [[]]
```

Example 3. There are too few available values.

```
combinations(2,3)
=> []
```

Why this mechanism works

This is subset generation with a required final length. If need positions remain, choosing a value too close to n leaves too few larger values to finish. Restrict the candidate loop to n-need+1. The pruning follows from a count of available choices, rather than from trial and error.

State and boundaries

path is increasing. start is its next possible value; need=k-len(path) is the number of missing choices.

Protocol

1. If path has k values, save a copy and return.

2. Compute need.

3. Try values start through n-need+1.

4. Choose, recurse from value+1, then undo.

Visualization

```
n=4, k=2
first choice <= 4-2+1 = 3
[1] -> [1,2] [1,3] [1,4]
[2] -> [2,3] [2,4]
[3] -> [3,4]
```

Python implementation

```
def combinations(n, k):
    output, path = [], []
    def visit(start):
        need = k - len(path)
        if need == 0:
            output.append(path.copy())
            return
        for value in range(start, n - need + 2):
            path.append(value)
            visit(value + 1)
            path.pop()
    visit(1)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
combinations(4,2)

L2 combinations: output, path = [], []
n=4; k=2; output=[]; path=[]

L4 visit: need = k - len(path)
start=1; need=2; k=2; n=4; output=[]; path=[]

L8 visit: for value in range(start, n - need + 2):
start=1; need=2; k=2; n=4; output=[]; path=[]; value=1

L9 visit: path.append(value)
start=1; need=2; k=2; n=4; output=[]; path=[1]; value=1

L4 visit: need = k - len(path)
start=2; need=1; k=2; n=4; output=[]; path=[1]

L8 visit: for value in range(start, n - need + 2):
start=2; need=1; k=2; n=4; output=[]; path=[1]; value=2

L9 visit: path.append(value)
start=2; need=1; k=2; n=4; output=[]; path=[1, 2]; value=2

L4 visit: need = k - len(path)
start=3; need=0; k=2; n=4; output=[]; path=[1, 2]

... 42 intermediate state changes omitted ...

L11 visit: path.pop()
start=4; need=1; k=2; n=4; output=[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]; path=[3];
    value=4
```

```
L10 visit: visit(value + 1)
start=1; need=2; k=2; n=4; output=[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]; path=[3];
    value=3

L11 visit: path.pop()
start=1; need=2; k=2; n=4; output=[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]; path=[];
    value=3

L12 combinations: visit(1)
n=4; k=2; output=[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]; path=[]
```

Result: [[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]. Increasing choices prevent reorderings of the same combination.

```
combinations(3,0)

L2 combinations: output, path = [], []
n=3; k=0; output=[]; path=[]

L4 visit: need = k - len(path)
start=1; need=0; k=0; n=3; output=[]; path=[]

L6 visit: output.append(path.copy())
start=1; need=0; k=0; n=3; output=[[]]; path=[]

L12 combinations: visit(1)
n=3; k=0; output=[[]]; path=[]
```

Result: [[]]. Zero selections form one combination.

Why the solution is correct

Increasing choices generate each k-element combination once. The last allowed candidate has exactly need-1 larger values available; every later candidate is impossible. Thus pruning removes no valid answer. Each call either finishes or increases path length and start.

Time and space

O(k*C(n,k)) output-copying work plus the search tree, bounded by O((k+1)*C(n,k)) for 0<=k<=n; O(k) auxiliary depth. k>n returns immediately.

Common mistakes

An exclusive Python range needs stop=n-need+2. Returning [] for k=0 incorrectly removes the unique empty combination.

Reasoning card

State: path is increasing. start is its next possible value; need=k-len(path) is the number of missing choices.

Invariant: Path is strictly increasing and has no repeated choice.

Decision to rehearse: Use a start index and prune when too few candidates remain.

Finish: Choose, recurse from value+1, then undo

Transfer exercise

Return only the count using a recurrence. Explain why enumeration cannot avoid the size of its output.

154. Generate Permutations

Foundation | Module 16

Problem

Return all permutations of distinct integers in nums. At each position, try still-unused input indices in their original order. Empty input returns [[]]. Leave nums unchanged and return independent lists.

Examples

Example 1. Order makes the two leaves different.

```
permutations([1,2])
=> [[1, 2], [2, 1]]
```

Example 2. There is one arrangement of no values.

```
permutations([])
=> [[]]
```

Example 3. Only one index can be placed.

```
permutations([8])
=> [[8]]
```

Why this mechanism works

Unlike a subset, a permutation distinguishes different orders. There is no increasing-index restriction: any unused index may fill the next position. The state therefore needs a used flag for every index. Each recursive frame owns one placement and must restore both that flag and the path when it returns.

State and boundaries

path contains distinct chosen input positions in order. used[i] is true exactly when index i appears in path.

Protocol

1. Save path at length n.
2. Scan all indices.
3. Skip used indices.
4. Mark an unused index and append its value.
5. Recurse, then pop and clear its flag.

Visualization

```
[1,2] -> place 1 -> place 2 -> [1,2]
undo both choices
      -> place 2 -> place 1 -> [2,1]
```

Python implementation

```
def permutations(nums):
    output, path = [], []
    used = [False] * len(nums)
    def visit():
        if len(path) == len(nums):
            output.append(path.copy())
            return
        for i, value in enumerate(nums):
            if used[i]:
                continue
            used[i] = True
            path.append(value)
            visit()
            path.pop()
            used[i] = False
    visit()
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
permutations([1,2])

L2 permutations: output, path = [], []
nums=[1, 2]; output=[]; path=[]

L3 permutations: used = [False] * len(nums)
nums=[1, 2]; output=[]; path=[]; used=[False, False]

L5 visit: if len(path) == len(nums):
nums=[1, 2]; output=[]; path=[]; used=[False, False]

L8 visit: for i, value in enumerate(nums):
nums=[1, 2]; output=[]; path=[]; used=[False, False]; i=0; value=1

L11 visit: used[i] = True
nums=[1, 2]; output=[]; path=[]; used=[True, False]; i=0; value=1

L12 visit: path.append(value)
nums=[1, 2]; output=[]; path=[1]; used=[True, False]; i=0; value=1

L5 visit: if len(path) == len(nums):
nums=[1, 2]; output=[]; path=[1]; used=[True, False]

L8 visit: for i, value in enumerate(nums):
nums=[1, 2]; output=[]; path=[1]; used=[True, False]; i=0; value=1

... 24 intermediate state changes omitted ...

L13 visit: visit()
```

```
nums=[1, 2]; output=[[1, 2], [2, 1]]; path=[2]; used=[False, True]; i=1; value=2

L14 visit: path.pop()
nums=[1, 2]; output=[[1, 2], [2, 1]]; path=[]; used=[False, True]; i=1; value=2

L15 visit: used[i] = False
nums=[1, 2]; output=[[1, 2], [2, 1]]; path=[]; used=[False, False]; i=1; value=2

L16 permutations: visit()
nums=[1, 2]; output=[[1, 2], [2, 1]]; path=[]; used=[False, False]
```

Result: [[1, 2], [2, 1]]. Order makes the two leaves different.

```
permutations([])

L2 permutations: output, path = [], []
nums=[]; output=[]; path=[]

L3 permutations: used = [False] * len(nums)
nums=[]; output=[]; path=[]; used=[]

L5 visit: if len(path) == len(nums):
nums=[]; output=[]; path=[]; used=[]

L6 visit: output.append(path.copy())
nums=[]; output=[[]]; path=[]; used=[]

L16 permutations: visit()
nums=[]; output=[[]]; path=[]; used=[]
```

Result: [[]]. There is one arrangement of no values.

Why the solution is correct

The flags prevent using a position twice, so every leaf is a valid permutation. Any permutation specifies a unique sequence of unused-index choices that the search explores. Restoring both structures preserves the state for sibling branches. Each placement reduces the number of unused indices.

Time and space

O(n*n!) time including copying answers and scanning candidate indices; O(n*n!) output space and O(n) auxiliary state.

Common mistakes

A global visited set that is never cleared turns path-local usage into permanent exclusion. Distinct values are required for uniqueness here.

Reasoning card

State: path contains distinct chosen input positions in order. used[i] is true exactly when index i appears in path.

Invariant: Path plus unused values equals the original set.

Decision to rehearse: At each position choose one unused value.

Finish: Recurse, then pop and clear its flag

Transfer exercise

Use in-place swaps to place each prefix position, and identify exactly which swap must be undone.

155. Generate Unique Permutations

Bridge | Module 16

Problem

Return every distinct permutation of nums, which may contain duplicate integers, in lexicographic order. Sort a copy and leave the original unchanged. Empty input returns [[]]. Equal occurrences may all be used, but identical value sequences must appear only once.

Examples

Example 1. Equal occurrences do not multiply identical output sequences.

```
unique_permutations([1,1,2])
=> [[1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

Example 2. Both occurrences are used in the sole distinct arrangement.

```
unique_permutations([2,2])
=> [[2, 2]]
```

Example 3. The empty arrangement is retained.

```
unique_permutations([])
=> [[]]
```

Why this mechanism works

The used-index protocol still works, but identical unused values at the same decision level create identical subtrees. Require equal occurrences to be selected from left to right: skip an occurrence if its equal predecessor is currently unused. If that predecessor is already on the path, the next equal occurrence remains a valid choice.

State and boundaries

values is sorted. used marks selected occurrences. A branch chooses the first currently unused occurrence of any equal value.

Protocol

1. Sort a copy.

2. At each depth scan unused indices.

3. Skip an equal value whose previous occurrence is unused.

4. Choose, explore and undo.

5. Copy complete paths.

Visualization

```
sorted [1a,1b,2]
root may choose 1a; skip 1b while 1a is unused
after choosing 1a, choosing 1b is allowed
```

Python implementation

```
def unique_permutations(nums):
    values = sorted(nums)
    used = [False] * len(values)
    output, path = [], []
    def visit():
        if len(path) == len(values):
            output.append(path.copy())
            return
        for i, value in enumerate(values):
            if used[i]:
                continue
            if i and value == values[i - 1] and not used[i - 1]:
                continue
            used[i] = True
            path.append(value)
            visit()
            path.pop()
            used[i] = False
    visit()
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
unique_permutations([1,1,2])

L2 unique_permutations: values = sorted(nums)
nums=[1, 1, 2]; values=[1, 1, 2]

L3 unique_permutations: used = [False] * len(values)
nums=[1, 1, 2]; values=[1, 1, 2]; used=[False, False, False]

L4 unique_permutations: output, path = [], []
nums=[1, 1, 2]; values=[1, 1, 2]; used=[False, False, False]; output=[]; path=[]

L6 visit: if len(path) == len(values):
output=[]; path=[]; used=[False, False, False]; values=[1, 1, 2]

L9 visit: for i, value in enumerate(values):
output=[]; path=[]; used=[False, False, False]; values=[1, 1, 2]; i=0; value=1

L14 visit: used[i] = True
output=[]; path=[]; used=[True, False, False]; values=[1, 1, 2]; i=0; value=1

L15 visit: path.append(value)
output=[]; path=[1]; used=[True, False, False]; values=[1, 1, 2]; i=0; value=1
```

```
L6 visit: if len(path) == len(values):
output=[]; path=[1]; used=[True, False, False]; values=[1, 1, 2]

... 62 intermediate state changes omitted ...

L16 visit: visit()
output=[[1, 1, 2], [1, 2, 1], [2, 1, 1]]; path=[2]; used=[False, False, True]; values=[1, 1,
    2]; i=2; value=2

L17 visit: path.pop()
output=[[1, 1, 2], [1, 2, 1], [2, 1, 1]]; path=[]; used=[False, False, True]; values=[1, 1,
    2]; i=2; value=2

L18 visit: used[i] = False
output=[[1, 1, 2], [1, 2, 1], [2, 1, 1]]; path=[]; used=[False, False, False]; values=[1, 1,
    2]; i=2; value=2

L19 unique_permutations: visit()
nums=[1, 1, 2]; values=[1, 1, 2]; used=[False, False, False]; output=[[1, 1, 2], [1, 2, 1],
    [2, 1, 1]]; path=[]
```

Result: [[1, 1, 2], [1, 2, 1], [2, 1, 1]]. Equal occurrences do not multiply identical output sequences.

`unique_permutations([2,2])`

```
L2 unique_permutations: values = sorted(nums)
nums=[2, 2]; values=[2, 2]

L3 unique_permutations: used = [False] * len(values)
nums=[2, 2]; values=[2, 2]; used=[False, False]

L4 unique_permutations: output, path = [], []
nums=[2, 2]; values=[2, 2]; used=[False, False]; output=[]; path=[]

L6 visit: if len(path) == len(values):
output=[]; path=[]; used=[False, False]; values=[2, 2]

L9 visit: for i, value in enumerate(values):
output=[]; path=[]; used=[False, False]; values=[2, 2]; i=0; value=2

L14 visit: used[i] = True
output=[]; path=[]; used=[True, False]; values=[2, 2]; i=0; value=2

L15 visit: path.append(value)
output=[]; path=[2]; used=[True, False]; values=[2, 2]; i=0; value=2

L6 visit: if len(path) == len(values):
output=[]; path=[2]; used=[True, False]; values=[2, 2]

... 10 intermediate state changes omitted ...

L17 visit: path.pop()
output=[[2, 2]]; path=[]; used=[True, False]; values=[2, 2]; i=0; value=2

L18 visit: used[i] = False
output=[[2, 2]]; path=[]; used=[False, False]; values=[2, 2]; i=0; value=2

L9 visit: for i, value in enumerate(values):
output=[[2, 2]]; path=[]; used=[False, False]; values=[2, 2]; i=1; value=2

L19 unique_permutations: visit()
nums=[2, 2]; values=[2, 2]; used=[False, False]; output=[[2, 2]]; path=[]
```

Result: [[2, 2]]. Both occurrences are used in the sole distinct arrangement.

Why the solution is correct

Every distinct value sequence has one canonical occurrence assignment: equal occurrences are used in increasing index order. The skip rule enforces precisely that assignment without preventing repeated equal values at later depths. Consequently every distinct permutation is generated exactly once in sorted choice order.

Time and space

O(n log n + n*n!) worst-case time, O(n) auxiliary state, and O(n*P) output for P distinct permutations. Duplicate pruning often greatly reduces work.

Common mistakes

Skipping every repeated value would forbid valid answers such as [1,1,2]. Checking whether the previous equal occurrence is used, with the wrong polarity, changes the canonical-choice rule.

Reasoning card

State: values is sorted. used marks selected occurrences. A branch chooses the first currently unused occurrence of any equal value.

Invariant: Every distinct multiset arrangement is generated once.

Decision to rehearse: Use sorted duplicate skipping or remaining frequency counts.

Finish: Copy complete paths

Transfer exercise

Replace used flags with a value-to-remaining-count dictionary. Compare the branching state for duplicate-heavy inputs.

156. Combination Sum

Bridge | Module 16

Problem

Given positive integer candidates and a nonnegative target, return unique nondecreasing combinations summing to target. A candidate may be reused without limit. Duplicate candidate values are consolidated. Output follows ascending depth-first choice order. Target zero returns [[]]; an unreachable target returns [].

Examples

Example 1. Reuse and a singleton both reach seven.

```
combination_sum([2,3,6,7],7)
=> [[2, 2, 3], [7]]
```

Example 2. Positive even values cannot form three.

```
combination_sum([2],3)
=> []
```

Example 3. No choices are required for target zero.

```
combination_sum([2,2],0)
=> [[]]
```

Why this mechanism works

A sorted choice list makes both deduplication and pruning explicit. Recurse from the same candidate index to allow reuse, while forbidding earlier indices to prevent reordered combinations. Positive values make remaining shrink on every choice; they also justify stopping the loop once a candidate exceeds remaining.

State and boundaries

path is nondecreasing and sums to target-remaining. start is the earliest reusable candidate index.

Protocol

1. Sort distinct candidates.
2. Save path when remaining is zero.
3. Try indices from start.
4. Break when the candidate exceeds remaining.
5. Choose, recurse from the same index with a smaller remainder, then undo.

Visualization

```
target 7, choices [2,3,6,7]
[2,2] remaining 3 -> choose 3 -> [2,2,3]
[7] remaining 0 -> [7]
larger-than-remaining choices stop the loop
```

Python implementation

```
def combination_sum(candidates, target):
    values = sorted(set(candidates))
    output, path = [], []
    def visit(start, remaining):
        if remaining == 0:
            output.append(path.copy())
            return
        for i in range(start, len(values)):
            value = values[i]
            if value > remaining:
                break
            path.append(value)
            visit(i, remaining - value)
            path.pop()
    visit(0, target)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
combination_sum([2,3,6,7],7)

L2 combination_sum: values = sorted(set(candidates))
candidates=[2, 3, 6, 7]; target=7; values=[2, 3, 6, 7]

L3 combination_sum: output, path = [], []
candidates=[2, 3, 6, 7]; target=7; values=[2, 3, 6, 7]; output=[]; path=[]

L5 visit: if remaining == 0:
start=0; remaining=7; output=[]; path=[]; values=[2, 3, 6, 7]

L8 visit: for i in range(start, len(values)):
start=0; remaining=7; output=[]; path=[]; values=[2, 3, 6, 7]; i=0

L9 visit: value = values[i]
start=0; remaining=7; output=[]; path=[]; values=[2, 3, 6, 7]; i=0; value=2

L12 visit: path.append(value)
start=0; remaining=7; output=[]; path=[2]; values=[2, 3, 6, 7]; i=0; value=2

L5 visit: if remaining == 0:
start=0; remaining=5; output=[]; path=[2]; values=[2, 3, 6, 7]

L8 visit: for i in range(start, len(values)):
start=0; remaining=5; output=[]; path=[2]; values=[2, 3, 6, 7]; i=0

... 57 intermediate state changes omitted ...

L6 visit: output.append(path.copy())
```

```
start=3; remaining=0; output=[[2, 2, 3], [7]]; path=[7]; values=[2, 3, 6, 7]

L13 visit: visit(i, remaining - value)
start=0; remaining=7; output=[[2, 2, 3], [7]]; path=[7]; values=[2, 3, 6, 7]; i=3; value=7

L14 visit: path.pop()
start=0; remaining=7; output=[[2, 2, 3], [7]]; path=[]; values=[2, 3, 6, 7]; i=3; value=7

L15 combination_sum: visit(0, target)
candidates=[2, 3, 6, 7]; target=7; values=[2, 3, 6, 7]; output=[[2, 2, 3], [7]]; path=[]
```

Result: [[2, 2, 3], [7]]. Reuse and a singleton both reach seven.

```
combination_sum([2],3)

L2 combination_sum: values = sorted(set(candidates))
candidates=[2]; target=3; values=[2]

L3 combination_sum: output, path = [], []
candidates=[2]; target=3; values=[2]; output=[]; path=[]

L5 visit: if remaining == 0:
start=0; remaining=3; output=[]; path=[]; values=[2]

L8 visit: for i in range(start, len(values)):
start=0; remaining=3; output=[]; path=[]; values=[2]; i=0

L9 visit: value = values[i]
start=0; remaining=3; output=[]; path=[]; values=[2]; i=0; value=2

L12 visit: path.append(value)
start=0; remaining=3; output=[]; path=[2]; values=[2]; i=0; value=2

L5 visit: if remaining == 0:
start=0; remaining=1; output=[]; path=[2]; values=[2]

L8 visit: for i in range(start, len(values)):
start=0; remaining=1; output=[]; path=[2]; values=[2]; i=0

L9 visit: value = values[i]
start=0; remaining=1; output=[]; path=[2]; values=[2]; i=0; value=2

L14 visit: path.pop()
start=0; remaining=3; output=[]; path=[]; values=[2]; i=0; value=2
```

Result: []. Positive even values cannot form three.

Why the solution is correct

Every emitted path sums to target and has canonical nondecreasing order. Every valid combination can be selected in that order and is explored once. A value larger than remaining and all later values cannot complete a nonnegative remainder. Positive choices guarantee finite depth.

Time and space

Let d=floor(target/min_candidate) and m be distinct candidate count. A coarse time bound is O(m log m + (m+1)^(d+1)*(d+1)); actual work follows visited prefixes and output copies. O(d+m) auxiliary state, excluding output. Empty candidates are constant-search work.

Common mistakes

Zero or negative candidates invalidate the decreasing-remainder termination proof. Advancing to i+1 accidentally prohibits reuse.

Reasoning card

State: path is nondecreasing and sums to target-remaining. start is the earliest reusable candidate index.

Invariant: Path is nondecreasing and sums to original-target minus remaining.

Decision to rehearse: Carry remaining target and a start index.

Finish: Choose, recurse from the same index with a smaller remainder, then undo

Transfer exercise

Allow each input occurrence only once. Explain both the new start index and the same-depth duplicate skip.

157. Generate Balanced Parentheses

Bridge | Module 16

Problem

Return all well-formed strings containing n pairs of parentheses, for n>=0. Try '(' before ')' at every decision, giving lexicographic output. A valid prefix never has more closing than opening parentheses. n=0 returns [''].

Examples

Example 1. Only prefixes with recoverable balance are explored.

```
balanced_parentheses(3)
=> ['((()))', '(()())', '(())()', '()(())', '()()()']
```

Example 2. The empty string is balanced.

```
balanced_parentheses(0)
=> ['']
```

Example 3. A closing cannot be chosen first.

```
balanced_parentheses(1)
=> ['()']
```

Why this mechanism works

Generating all 2^(2n) strings and validating them wastes work on prefixes that can never recover. Keep opened and closed counts while constructing the answer. An opening is available until n have been placed; a closing is available only when it can match an unmatched opening. These local rules preserve the global balance requirement.

State and boundaries

path has opened openings and closed closings, with 0<=closed<=opened<=n.

Protocol

1. At length 2*n save the joined path.

2. If opened<n, append an opening, explore and undo.

3. If closed<opened, append a closing, explore and undo.

Visualization

```
opened=2, closed=1 -> '((' + ')' = '(()'
unmatched openings = 1
closing allowed; opening allowed only if opened<n
```

Python implementation

```
def balanced_parentheses(n):
    output, path = [], []
    def visit(opened, closed):
        if len(path) == 2 * n:
            output.append(''.join(path))
            return
        if opened < n:
            path.append('(')
            visit(opened + 1, closed)
            path.pop()
        if closed < opened:
            path.append(')')
            visit(opened, closed + 1)
            path.pop()
    visit(0, 0)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
balanced_parentheses(3)

L2 balanced_parentheses: output, path = [], []
n=3; output=[]; path=[]

L4 visit: if len(path) == 2 * n:
opened=0; closed=0; n=3; output=[]; path=[]

L8 visit: path.append('(')
opened=0; closed=0; n=3; output=[]; path=['(']

L4 visit: if len(path) == 2 * n:
opened=1; closed=0; n=3; output=[]; path=['(']

L8 visit: path.append('(')
opened=1; closed=0; n=3; output=[]; path=['(', '(']

L4 visit: if len(path) == 2 * n:
opened=2; closed=0; n=3; output=[]; path=['(', '(']

L8 visit: path.append('(')
opened=2; closed=0; n=3; output=[]; path=['(', '(', '(']

L4 visit: if len(path) == 2 * n:
opened=3; closed=0; n=3; output=[]; path=['(', '(', '(']

... 80 intermediate state changes omitted ...

L14 visit: path.pop()
opened=1; closed=0; n=3; output=['((()))', '(()())', '(())()', '()(())', '()()()']; path=['(']
```

```
L9 visit: visit(opened + 1, closed)
opened=0; closed=0; n=3; output=['((()))', '(()())', '(())()', '()(())', '()()()']; path=['(']

L10 visit: path.pop()
opened=0; closed=0; n=3; output=['((()))', '(()())', '(())()', '()(())', '()()()']; path=[]

L15 balanced_parentheses: visit(0, 0)
n=3; output=['((()))', '(()())', '(())()', '()(())', '()()()']; path=[]
```

Result: ['((()))', '(()())', '(())()', '()(())', '()()()']. Only prefixes with recoverable balance are explored.

```
balanced_parentheses(0)

L2 balanced_parentheses: output, path = [], []
n=0; output=[]; path=[]

L4 visit: if len(path) == 2 * n:
opened=0; closed=0; n=0; output=[]; path=[]

L5 visit: output.append(''.join(path))
opened=0; closed=0; n=0; output=['']; path=[]

L15 balanced_parentheses: visit(0, 0)
n=0; output=['']; path=[]
```

Result: ['']. The empty string is balanced.

Why the solution is correct

The two guards preserve valid prefix balance and the opening limit. A completed length-2n prefix must have exactly n of each symbol. Every valid string obeys the guards at every position, so its unique choice route is explored. Each choice increases length toward 2n.

Time and space

O(n*C_n) time and output characters for the nth Catalan number C_n, with O(n) auxiliary path and stack. n=0 is constant.

Common mistakes

Allowing a closing whenever closed<n admits invalid prefixes. Checking balance only at the end loses the pruning benefit.

Reasoning card

State: path has opened openings and closed closings, with 0<=closed<=opened<=n.

Invariant: Every prefix obeys closes <= opens <= n.

Decision to rehearse: Add open when available; add close only when closes < opens.

Finish: If closed<opened, append a closing, explore and undo

Transfer exercise

Impose a maximum nesting depth h by adding one guard on opened-closed.

158. Word Search in Grid

Interview | Module 16

Problem

Return whether word can be read along a path in a rectangular character board, moving up, down, left or right without reusing a cell in that path. Leave the board unchanged. The empty word succeeds even on an empty board; a nonempty word fails on an empty board.

Examples

Example 1. The route bends once and uses three distinct cells.

```
word_exists([list("ABC"),list("DEF")],"ABE")
=> True
```

Example 2. The final A would reuse the first cell.

```
word_exists([list("AB")],"ABA")
=> False
```

Example 3. The empty word requires no board cells.

```
word_exists([],"")
=> True
```

Why this mechanism works

Unlike flood fill, a cell's availability belongs to the current candidate path. A globally permanent visited set would incorrectly exclude cells from later starting points. Match one character, reserve that cell, explore the next character, then release the cell even after a successful branch. The recursive contract includes both location and word position.

State and boundaries

used contains exactly the cells reserved by earlier matched characters. visit(r,c,i) asks whether word[i:] can begin at this unused cell.

Protocol

1. Accept an empty word.

2. Try every cell as a start.

3. Reject out-of-bounds, used or mismatching cells.

4. Accept a matching final character.

5. Reserve the current cell, explore four neighbors, release it, and return the result.

Visualization

```
A B C
D E F
word ABE: (0,0) -> (0,1) -> (1,1)
used is local to this route, then restored
```

Python implementation

```
def word_exists(board, word):
    if not word:
        return True
    if not board or not board[0]:
        return False
    rows, cols = len(board), len(board[0])
    used = set()
    def visit(r, c, i):
        if not (0 <= r < rows and 0 <= c < cols):
            return False
        if (r, c) in used or board[r][c] != word[i]:
            return False
        if i == len(word) - 1:
            return True
        used.add((r, c))
        found = any(visit(r + dr, c + dc, i + 1)
                    for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)))
        used.remove((r, c))
        return found
    return any(visit(r, c, 0) for r in range(rows) for c in range(cols))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
word_exists([list("ABC"),list("DEF")],"ABE")

L2 word_exists: if not word:
board=[['A', 'B', 'C'], ['D', 'E', 'F']]; word='ABE'

L6 word_exists: rows, cols = len(board), len(board[0])
board=[['A', 'B', 'C'], ['D', 'E', 'F']]; word='ABE'; cols=3; rows=2

L7 word_exists: used = set()
board=[['A', 'B', 'C'], ['D', 'E', 'F']]; word='ABE'; cols=3; rows=2; used=set()

L9 visit: if not (0 <= r < rows and 0 <= c < cols):
r=0; c=0; i=0; board=[['A', 'B', 'C'], ['D', 'E', 'F']]; cols=3; rows=2; used=set();
    word='ABE'

L15 visit: used.add((r, c))
r=0; c=0; i=0; board=[['A', 'B', 'C'], ['D', 'E', 'F']]; cols=3; rows=2; used={(0, 0)};
    word='ABE'

L16 <genexpr>: found = any(visit(r + dr, c + dc, i + 1)
.0=tuple_iterator; c=0; i=0; r=0
```

```
L17 <genexpr>: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)))
.0=tuple_iterator; c=0; i=0; r=0; dr=1; dc=0

L9 visit: if not (0 <= r < rows and 0 <= c < cols):
r=1; c=0; i=1; board=[['A', 'B', 'C'], ['D', 'E', 'F']]; cols=3; rows=2; used={(0, 0)};
    word='ABE'

... 9 intermediate state changes omitted ...

L18 visit: used.remove((r, c))
r=0; c=1; i=1; board=[['A', 'B', 'C'], ['D', 'E', 'F']]; cols=3; rows=2; used={(0, 0)};
    word='ABE'; found=True

L16 visit: found = any(visit(r + dr, c + dc, i + 1)
r=0; c=0; i=0; board=[['A', 'B', 'C'], ['D', 'E', 'F']]; cols=3; rows=2; used={(0, 0)};
    word='ABE'; found=True

L18 visit: used.remove((r, c))
r=0; c=0; i=0; board=[['A', 'B', 'C'], ['D', 'E', 'F']]; cols=3; rows=2; used=set();
    word='ABE'; found=True

L20 <genexpr>: return any(visit(r, c, 0) for r in range(rows) for c in range(cols))
.0=range_iterator; r=0; c=0; cols=3
```

Result: True. The route bends once and uses three distinct cells.

```
word_exists([list("AB")],"ABA")

L2 word_exists: if not word:
board=[['A', 'B']]; word='ABA'

L6 word_exists: rows, cols = len(board), len(board[0])
board=[['A', 'B']]; word='ABA'; cols=2; rows=1

L7 word_exists: used = set()
board=[['A', 'B']]; word='ABA'; cols=2; rows=1; used=set()

L9 visit: if not (0 <= r < rows and 0 <= c < cols):
r=0; c=0; i=0; board=[['A', 'B']]; cols=2; rows=1; used=set(); word='ABA'

L15 visit: used.add((r, c))
r=0; c=0; i=0; board=[['A', 'B']]; cols=2; rows=1; used={(0, 0)}; word='ABA'

L16 <genexpr>: found = any(visit(r + dr, c + dc, i + 1)
.0=tuple_iterator; c=0; i=0; r=0

L17 <genexpr>: for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)))
.0=tuple_iterator; c=0; i=0; r=0; dr=1; dc=0

L9 visit: if not (0 <= r < rows and 0 <= c < cols):
r=1; c=0; i=1; board=[['A', 'B']]; cols=2; rows=1; used={(0, 0)}; word='ABA'

... 22 intermediate state changes omitted ...

L20 <genexpr>: return any(visit(r, c, 0) for r in range(rows) for c in range(cols))
.0=range_iterator; r=0; c=0; cols=2

L9 visit: if not (0 <= r < rows and 0 <= c < cols):
r=0; c=1; i=0; board=[['A', 'B']]; cols=2; rows=1; used=set(); word='ABA'

L20 <genexpr>: return any(visit(r, c, 0) for r in range(rows) for c in range(cols))
.0=range_iterator; r=0; c=1; cols=2
```

```
L20 <genexpr>: return any(visit(r, c, 0) for r in range(rows) for c in range(cols))
.0=range_iterator; r=0; c=1; cols=2
```

Result: False. The final A would reuse the first cell.

Why the solution is correct

A successful call matches its current character and extends through adjacent distinct cells by the recursive contract. Every valid path has a starting cell and a sequence of neighbor choices the search explores. Releasing reservations restores sibling state. The word index advances, bounding depth.

Time and space

O(R*C*4^L) coarse worst-case time for word length L; O(min(L,R*C)) auxiliary path/set and stack, excluding input.

Common mistakes

Permanent visitation confuses reachability with path enumeration. Returning immediately after a recursive success without cleanup is dangerous when mutable state is reused.

Reasoning card

State: used contains exactly the cells reserved by earlier matched characters. visit(r,c,i) asks whether word[i:] can begin at this unused cell.

Invariant: Current path matches the word prefix and uses unique cells.

Decision to rehearse: Mark cell, recurse to next character, restore cell.

Finish: Reserve the current cell, explore four neighbors, release it, and return the result

Transfer exercise

Search for many words together with a trie. Which shared prefixes would avoid repeating the same board exploration?

159. Palindrome Partitioning

Interview | Module 16

Problem

Partition text into nonempty palindromic substrings and return every partition. Preserve substring order and try each next cut from shortest to longest. Empty text returns [[]]. Characters are compared exactly, without case folding or normalization.

Examples

Example 1. The first two letters may be separate or one palindrome.

```
palindrome_partitions("aab")
=> [['a', 'a', 'b'], ['aa', 'b']]
```

Example 2. The empty prefix already covers the whole input.

```
palindrome_partitions("")
=> [[]]
```

Example 3. Both singleton cuts and the whole palindrome are valid.

```
palindrome_partitions("aba")
=> [['a', 'b', 'a'], ['aba']]
```

Why this mechanism works

At index start, the next piece can end at any later position, but only a palindrome is a legal choice. Precompute which intervals are palindromes so the recursive search can focus on partitions. An interval is palindromic when its endpoints match and its interior is palindromic, with lengths one and two as bases.

State and boundaries

pal[i][j] describes inclusive substring text[i:j+1]. path partitions text[:start] completely into palindromes.

Protocol

1. Fill palindrome intervals from larger start indices toward smaller ones.
2. At start==n copy path.
3. Try each palindromic interval starting at start.
4. Append its substring, recurse after its end, then undo.

Visualization

```
aab
cut a | a | b -> all palindromes
cut aa | b    -> all palindromes
cut a | ab    -> reject ab
```

Python implementation

```
def palindrome_partitions(text):
    n = len(text)
    pal = [[False] * n for _ in range(n)]
    for i in range(n - 1, -1, -1):
        for j in range(i, n):
            pal[i][j] = text[i] == text[j] and (j - i < 2 or pal[i + 1][j - 1])
    output, path = [], []
    def visit(start):
        if start == n:
            output.append(path.copy())
            return
        for end in range(start, n):
            if pal[start][end]:
                path.append(text[start:end + 1])
                visit(end + 1)
                path.pop()
    visit(0)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
palindrome_partitions("aab")

L2 palindrome_partitions: n = len(text)
text='aab'; n=3

L3 palindrome_partitions: pal = [[False] * n for _ in range(n)]
text='aab'; n=3; _=0

L3 palindrome_partitions: pal = [[False] * n for _ in range(n)]
text='aab'; n=3; _=1

L3 palindrome_partitions: pal = [[False] * n for _ in range(n)]
text='aab'; n=3; _=2

L3 palindrome_partitions: pal = [[False] * n for _ in range(n)]
text='aab'; n=3; pal=[[False, False, False], [False, False, False], [False, False, False]]

L4 palindrome_partitions: for i in range(n - 1, -1, -1):
text='aab'; n=3; pal=[[False, False, False], [False, False, False], [False, False, False]];
    i=2

L5 palindrome_partitions: for j in range(i, n):
text='aab'; n=3; pal=[[False, False, False], [False, False, False], [False, False, False]];
    i=2; j=2

L6 palindrome_partitions: pal[i][j] = text[i] == text[j] and (j - i < 2 or pal[i + 1][j - 1])
text='aab'; n=3; pal=[[False, False, False], [False, False, False], [False, False, True]];
```

```
    i=2; j=2

... 38 intermediate state changes omitted ...

L15 visit: visit(end + 1)
start=0; n=3; output=[['a', 'a', 'b'], ['aa', 'b']]; pal=[[True, True, False], [False, True,
    False], [False, False, True]]; path=['aa']; text='aab'; end=1

L16 visit: path.pop()
start=0; n=3; output=[['a', 'a', 'b'], ['aa', 'b']]; pal=[[True, True, False], [False, True,
    False], [False, False, True]]; path=[]; text='aab'; end=1

L12 visit: for end in range(start, n):
start=0; n=3; output=[['a', 'a', 'b'], ['aa', 'b']]; pal=[[True, True, False], [False, True,
    False], [False, False, True]]; path=[]; text='aab'; end=2

L17 palindrome_partitions: visit(0)
text='aab'; n=3; pal=[[True, True, False], [False, True, False], [False, False, True]]; i=0;
    j=2; output=[['a', 'a', 'b'], ['aa', 'b']]; path=[]
```

Result: [['a', 'a', 'b'], ['aa', 'b']]. The first two letters may be separate or one palindrome.

```
palindrome_partitions("")

L2 palindrome_partitions: n = len(text)
text=''; n=0

L3 palindrome_partitions: pal = [[False] * n for _ in range(n)]
text=''; n=0; pal=[]

L7 palindrome_partitions: output, path = [], []
text=''; n=0; pal=[]; output=[]; path=[]

L9 visit: if start == n:
start=0; n=0; output=[]; pal=[]; path=[]; text=''

L10 visit: output.append(path.copy())
start=0; n=0; output=[[]]; pal=[]; path=[]; text=''

L17 palindrome_partitions: visit(0)
text=''; n=0; pal=[]; output=[[]]; path=[]
```

Result: [[]]. The empty prefix already covers the whole input.

Why the solution is correct

The interval recurrence is correct from the inside outward. The search appends only palindromes and never leaves gaps or overlaps. Every valid partition defines a unique increasing sequence of cuts, all of which are tried. start strictly increases until the entire text is covered.

Time and space

O(n^2 + n*2^n) worst-case time and output storage; O(n^2) palindrome table plus O(n) recursive/path content auxiliary space. Substring copies are included in the bound.

Common mistakes

Treating the empty suffix as failure loses completed partitions. Mixing inclusive table ends with exclusive slices drops the final character.

Reasoning card

State: pal[i][j] describes inclusive substring text[i:j+1]. path partitions text[:start] completely into palindromes.

Invariant: Path partitions exactly the processed prefix.

Decision to rehearse: Choose the next palindromic prefix and recurse on the suffix.

Finish: Append its substring, recurse after its end, then undo

Transfer exercise

Find the minimum number of cuts using the same palindrome table but a dynamic program over prefixes.

160. N-Queens

Interview | Module 16

Problem

Return every arrangement of n nonattacking queens on an n-by-n board, for n>=0. Represent each board as n strings using Q and '.'. Place one queen per row and try columns in ascending order. n=0 returns [[]]. Leave no shared mutable boards in output.

Examples

Example 1. Column-order search finds the two four-queen boards.

```
n_queens(4)
=> [['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.', 'Q...', '...Q', '.Q..']]
```

Example 2. Every second-row position is attacked.

```
n_queens(2)
=> []
```

Example 3. The empty board has one empty arrangement.

```
n_queens(0)
=> [[]]
```

Why this mechanism works

Placing exactly one queen in each successive row handles row conflicts automatically. A column set handles vertical conflicts. Diagonals are identified by row-column and row+column, so two more sets replace repeated scans of the board. Each recursive frame reserves these three identifiers and releases them after exploring its row choice.

State and boundaries

columns holds chosen column indices in row order. used_cols, down and up describe all attacks from those placed queens.

Protocol

1. At row n construct and save board strings.
2. Try each column.
3. Skip any occupied column or diagonal.
4. Reserve the three identifiers and append the column.
5. Recurse to the next row.

6. Undo every reservation.

Visualization

```
row 0: .Q..   column 1, diagonals -1 and 1
row 1: ...Q   column 3, diagonals -2 and 4
row 2: Q...
row 3: ..Q.   one complete nonattacking board
```

Python implementation

```
def n_queens(n):
    output, columns = [], []
    used_cols, down, up = set(), set(), set()
    def visit(row):
        if row == n:
            output.append(['.' * c + 'Q' + '.' * (n - c - 1) for c in columns])
            return
        for col in range(n):
            if col in used_cols or row - col in down or row + col in up:
                continue
            columns.append(col)
            used_cols.add(col)
            down.add(row - col)
            up.add(row + col)
            visit(row + 1)
            up.remove(row + col)
            down.remove(row - col)
            used_cols.remove(col)
            columns.pop()
    visit(0)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
n_queens(4)

L2 n_queens: output, columns = [], []
n=4; columns=[]; output=[]

L3 n_queens: used_cols, down, up = set(), set(), set()
n=4; columns=[]; output=[]; down=set(); up=set(); used_cols=set()

L5 visit: if row == n:
row=0; columns=[]; down=set(); n=4; output=[]; up=set(); used_cols=set()

L8 visit: for col in range(n):
row=0; columns=[]; down=set(); n=4; output=[]; up=set(); used_cols=set(); col=0

L11 visit: columns.append(col)
row=0; columns=[0]; down=set(); n=4; output=[]; up=set(); used_cols=set(); col=0

L12 visit: used_cols.add(col)
row=0; columns=[0]; down=set(); n=4; output=[]; up=set(); used_cols={0}; col=0

L13 visit: down.add(row - col)
```

```
row=0; columns=[0]; down={0}; n=4; output=[]; up=set(); used_cols={0}; col=0

L14 visit: up.add(row + col)
row=0; columns=[0]; down={0}; n=4; output=[]; up={0}; used_cols={0}; col=0

... 214 intermediate state changes omitted ...

L17 visit: down.remove(row - col)
row=0; columns=[3]; down=set(); n=4; output=[['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.',
    'Q...', '...Q', '.Q..']]; up=set(); used_cols={3}; col=3

L18 visit: used_cols.remove(col)
row=0; columns=[3]; down=set(); n=4; output=[['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.',
    'Q...', '...Q', '.Q..']]; up=set(); used_cols=set(); col=3

L19 visit: columns.pop()
row=0; columns=[]; down=set(); n=4; output=[['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.', 'Q...',
     '...Q', '.Q..']]; up=set(); used_cols=set(); col=3

L20 n_queens: visit(0)
n=4; columns=[]; output=[['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.', 'Q...', '...Q', '.Q..']];
    down=set(); up=set(); used_cols=set()
```

Result: [['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.', 'Q...', '...Q', '.Q..']]. Column-order search finds the two four-queen boards.

`n_queens(2)`

```
L2 n_queens: output, columns = [], []
n=2; columns=[]; output=[]

L3 n_queens: used_cols, down, up = set(), set(), set()
n=2; columns=[]; output=[]; down=set(); up=set(); used_cols=set()

L5 visit: if row == n:
row=0; columns=[]; down=set(); n=2; output=[]; up=set(); used_cols=set()

L8 visit: for col in range(n):
row=0; columns=[]; down=set(); n=2; output=[]; up=set(); used_cols=set(); col=0

L11 visit: columns.append(col)
row=0; columns=[0]; down=set(); n=2; output=[]; up=set(); used_cols=set(); col=0

L12 visit: used_cols.add(col)
row=0; columns=[0]; down=set(); n=2; output=[]; up=set(); used_cols={0}; col=0

L13 visit: down.add(row - col)
row=0; columns=[0]; down={0}; n=2; output=[]; up=set(); used_cols={0}; col=0

L14 visit: up.add(row + col)
row=0; columns=[0]; down={0}; n=2; output=[]; up={0}; used_cols={0}; col=0

... 15 intermediate state changes omitted ...

L16 visit: up.remove(row + col)
row=0; columns=[1]; down={-1}; n=2; output=[]; up=set(); used_cols={1}; col=1

L17 visit: down.remove(row - col)
row=0; columns=[1]; down=set(); n=2; output=[]; up=set(); used_cols={1}; col=1

L18 visit: used_cols.remove(col)
row=0; columns=[1]; down=set(); n=2; output=[]; up=set(); used_cols=set(); col=1
```

```
L19 visit: columns.pop()
row=0; columns=[]; down=set(); n=2; output=[]; up=set(); used_cols=set(); col=1
```

Result: []. Every second-row position is attacked.

Why the solution is correct

The guards prevent every possible attack from an earlier queen, while row progression prevents same-row attacks. Every valid board has a unique sequence of row columns, whose choices remain unblocked and are therefore explored. Cleanup preserves sibling state, and row increases to n.

Time and space

O(n*n! + S*n^2) coarse time bound, including board materialization for S solutions; O(n) auxiliary search state and O(S*n^2) output characters.

Common mistakes

The two diagonal families use different expressions. Removing only the column on return leaves stale diagonal conflicts and silently loses solutions.

Reasoning card

State: columns holds chosen column indices in row order. used_cols, down and up describe all attacks from those placed queens.

Invariant: All placed queens are mutually non-attacking.

Decision to rehearse: Track blocked columns and diagonals; place by row.

Finish: Undo every reservation

Transfer exercise

Count solutions without constructing boards, then represent attack sets with bit masks and explain the shifted diagonal masks.

Module 17
Binary Trees: Recursive DFS and Structural Contracts

Make each recursive call promise one precise fact about its subtree and combine answers bottom-up.

A tree makes recursive subproblems structural: the left and right child subtrees are already disjoint. Traversal order depends on when the parent acts. Summary problems require a stronger call contract. Height is an upward contribution; diameter is a completed path, so a single variable cannot serve both meanings.

The ten-step progression

161. Build and Serialize a Binary Tree

162. Maximum Depth

163. Recursive Preorder Traversal

164. Recursive Inorder Traversal

165. Recursive Postorder Traversal

166. Same Tree

167. Invert Binary Tree

168. Diameter of Binary Tree

169. Balanced Binary Tree

170. Root-to-Leaf Target Sum

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

161. Build and Serialize a Binary Tree

Foundation | Module 17

Problem

Build a binary tree from compact level-order values, where None marks a missing child, then serialize it back to that format. Values are integers; [] represents an empty tree. Each real node consumes up to two following child entries. Input contains no orphan entries. Omit trailing None markers when serializing.

Examples

Example 1. A missing parent does not consume child slots.

```
serialize_binary_tree(build_binary_tree([1,None,2,3]))
=> [1, None, 2, 3]
```

Example 2. Both empty representations correspond.

```
serialize_binary_tree(build_binary_tree([]))
=> []
```

Example 3. Redundant trailing missing markers are removed.

```
serialize_binary_tree(build_binary_tree([5,None,None]))
=> [5]
```

Why this mechanism works

The representation describes pending child slots of real parents, not array indices in a complete tree. A queue holds those parents in level order. Serialization reverses this process: emit a missing marker for a missing child, but never expand that marker into further slots. Trimming only trailing markers produces a canonical representation.

State and boundaries

The build queue contains real nodes whose child slots are unfilled. The serialization queue contains the next real or missing slots to emit.

Protocol

1. Create the root unless input is empty or begins with None.

2. For each queued parent consume its next two child entries.

3. Create and enqueue nonmissing children.

4. Serialize by FIFO slot traversal.

5. Trim trailing missing markers.

Visualization

```
values [1,None,2,3]
    1
     \
      2
     /
    3
3 fills node 2's left slot, not an absent parent's slot
```

Python implementation

```
from collections import deque
from structures import TreeNode

def build_binary_tree(values):
    if not values or values[0] is None:
        return None
    root = TreeNode(values[0])
    queue = deque([root])
    i = 1
    while queue and i < len(values):
        parent = queue.popleft()
        for side in ('left', 'right'):
            if i == len(values):
                break
            value = values[i]
            i += 1
            if value is not None:
                child = TreeNode(value)
                setattr(parent, side, child)
                queue.append(child)
    return root

def serialize_binary_tree(root):
    output, queue = [], deque([root])
    while queue:
        node = queue.popleft()
        output.append(None if node is None else node.val)
        if node is not None:
            queue.extend((node.left, node.right))
    while output and output[-1] is None:
        output.pop()
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
serialize_binary_tree(build_binary_tree([1,None,2,3]))

L5 build_binary_tree: if not values or values[0] is None:
values=[1, None, 2, 3]
```

```
L7 build_binary_tree: root = TreeNode(values[0])
values=[1, None, 2, 3]; root=N1(val=1,left=None,right=None)

L8 build_binary_tree: queue = deque([root])
values=[1, None, 2, 3]; root=N1(val=1,left=None,right=None);
    queue=deque([N1(val=1,left=None,right=None)])

L9 build_binary_tree: i = 1
values=[1, None, 2, 3]; root=N1(val=1,left=None,right=None);
    queue=deque([N1(val=1,left=None,right=None)]); i=1

L11 build_binary_tree: parent = queue.popleft()
values=[1, None, 2, 3]; root=N1(val=1,left=None,right=None); queue=deque([]); i=1;
    parent=N1(val=1,left=None,right=None)

L12 build_binary_tree: for side in ('left', 'right'):
values=[1, None, 2, 3]; root=N1(val=1,left=None,right=None); queue=deque([]); i=1;
    parent=N1(val=1,left=None,right=None); side='left'

L15 build_binary_tree: value = values[i]
values=[1, None, 2, 3]; root=N1(val=1,left=None,right=None); queue=deque([]); i=1;
    parent=N1(val=1,left=None,right=None); side='left'; value=None

L16 build_binary_tree: i += 1
values=[1, None, 2, 3]; root=N1(val=1,left=None,right=None); queue=deque([]); i=2;
    parent=N1(val=1,left=None,right=None); side='left'; value=None

... 31 intermediate state changes omitted ...

L27 serialize_binary_tree: output.append(None if node is None else node.val)
root=N1(val=1,left=None,right=N2); output=[1, None, 2, 3, None, None, None]; queue=deque([]);
    node=None

L31 serialize_binary_tree: output.pop()
root=N1(val=1,left=None,right=N2); output=[1, None, 2, 3, None, None]; queue=deque([]);
    node=None

L31 serialize_binary_tree: output.pop()
root=N1(val=1,left=None,right=N2); output=[1, None, 2, 3, None]; queue=deque([]); node=None

L31 serialize_binary_tree: output.pop()
root=N1(val=1,left=None,right=N2); output=[1, None, 2, 3]; queue=deque([]); node=None
```

Result: [1, None, 2, 3]. A missing parent does not consume child slots.

`serialize_binary_tree(build_binary_tree([]))`

```
L5 build_binary_tree: if not values or values[0] is None:
values=[]

L24 serialize_binary_tree: output, queue = [], deque([root])
root=None; output=[]; queue=deque([None])

L26 serialize_binary_tree: node = queue.popleft()
root=None; output=[]; queue=deque([]); node=None

L27 serialize_binary_tree: output.append(None if node is None else node.val)
root=None; output=[None]; queue=deque([]); node=None

L31 serialize_binary_tree: output.pop()
root=None; output=[]; queue=deque([]); node=None
```

Result: []. Both empty representations correspond.

Why the solution is correct

Each consumed entry is attached to the next slot in the specified level order, so construction preserves every supplied relationship. Serialization visits those same slots in the same order. Removing trailing missing slots cannot change any real node's position or relationships. Both queues process finitely many nodes and slots.

Time and space

O(n) time and O(n) output/queue space for n real nodes, with at most 2n+1 serialized slots before trimming.

Common mistakes

Using children at 2*i+1 and 2*i+2 interprets a different representation when missing parents occur. Expanding None nodes during serialization creates endless empty descendants.

Reasoning card

State: The build queue contains real nodes whose child slots are unfilled. The serialization queue contains the next real or missing slots to emit.

Invariant: Queue slots correspond to parent child positions.

Decision to rehearse: Distinguish absent children from traversal order.

Finish: Trim trailing missing markers

Transfer exercise

Design a preorder format with explicit missing markers. Explain why omitting those interior markers would lose tree shape.

162. Maximum Depth

Foundation | Module 17

Problem

Return the maximum depth of a finite binary tree, measured in nodes on a root-to-leaf path. An empty tree has depth zero; a single node has depth one. Nodes may have any integer values, which do not affect the answer. Do not mutate the tree.

Examples

Example 1. The longest path is 1,2,4 and contains three nodes.

```
max_depth(build_tree([1,2,3,4]))
=> 3
```

Example 2. No nodes contribute to depth.

```
max_depth(None)
=> 0
```

Example 3. A singleton has node-count depth one.

```
max_depth(build_tree([9]))
=> 1
```

Why this mechanism works

A helper should return one complete fact about its subtree. If a node has no subtree, its depth is zero. Otherwise a longest path takes the current node and then whichever child offers the larger depth. This is a postorder computation even though the code is a compact expression.

State and boundaries

max_depth(node) returns the exact node-count height of node's subtree; child calls are independent results.

Protocol

1. Return zero for None.

2. Obtain left and right subtree depths.

3. Return one plus their maximum.

Visualization

```
      1                depth 3
     / \
    2   3              child depths 2 and 1
   /
  4                    leaf depth 1
```

Python implementation

```
from structures import build_tree

def max_depth(root):
    if root is None:
        return 0
    left = max_depth(root.left)
    right = max_depth(root.right)
    return 1 + max(left, right)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
max_depth(build_tree([1,2,3,4]))

L4 max_depth: if root is None:
root=N1(val=1,left=N2,right=N3)

L4 max_depth: if root is None:
root=N2(val=2,left=N4,right=None)

L4 max_depth: if root is None:
root=N4(val=4,left=None,right=None)

L4 max_depth: if root is None:
root=None

L6 max_depth: left = max_depth(root.left)
root=N4(val=4,left=None,right=None); left=0

L4 max_depth: if root is None:
root=None

L7 max_depth: right = max_depth(root.right)
root=N4(val=4,left=None,right=None); left=0; right=0

L6 max_depth: left = max_depth(root.left)
root=N2(val=2,left=N4,right=None); left=1

... 5 intermediate state changes omitted ...

L6 max_depth: left = max_depth(root.left)
root=N3(val=3,left=None,right=None); left=0

L4 max_depth: if root is None:
root=None

L7 max_depth: right = max_depth(root.right)
root=N3(val=3,left=None,right=None); left=0; right=0
```

```
L7 max_depth: right = max_depth(root.right)
root=N1(val=1,left=N2,right=N3); left=2; right=1
```

Result: 3. The longest path is 1,2,4 and contains three nodes.

```
max_depth(None)

L4 max_depth: if root is None:
root=None
```

Result: 0. No nodes contribute to depth.

Why the solution is correct

The empty-tree base matches the definition. Every nonempty root-to-leaf path uses the root and one child path. Choosing the larger correct child depth therefore yields exactly the longest such path. Structural induction proves every return value; finite child subtrees ensure termination.

Time and space

O(n) time and O(h) recursive stack space for n nodes and tree height h. A skewed tree can have h=n and exceed Python's recursion limit.

Common mistakes

Returning zero for a leaf changes the unit to edges. Using the sum of child depths measures neither depth nor a valid single downward path.

Reasoning card

State: max_depth(node) returns the exact node-count height of node's subtree; child calls are independent results.

Invariant: Each call returns the correct depth of its subtree.

Decision to rehearse: Children return subtree depths upward.

Finish: Return one plus their maximum

Transfer exercise

Compute the same result with an explicit stack carrying (node,depth), avoiding recursion depth limits.

163. Recursive Preorder Traversal

Foundation | Module 17

Problem

Return the values of a binary tree in recursive preorder: root, left subtree, right subtree. Empty input returns []. Preserve duplicate values and visit every node occurrence once. Leave the tree unchanged.

Examples

Example 1. The entire left subtree follows its parent before the right subtree.

```
preorder(build_tree([1,2,3,4]))
=> [1, 2, 4, 3]
```

Example 2. An empty subtree emits nothing.

```
preorder(None)
=> []
```

Example 3. Equal-valued node occurrences are distinct visits.

```
preorder(build_tree([2,2,2]))
=> [2, 2, 2]
```

Why this mechanism works

Traversal order is controlled by when the current node's value is emitted relative to child calls. Preorder emits before either delegation, so a parent always precedes all of its descendants. A shared append-only output avoids the repeated list copying of concatenated recursive results.

State and boundaries

visit(node) appends the complete preorder sequence of that subtree. output already holds all earlier traversal events.

Protocol

1. Return immediately for None.
2. Append the current value.
3. Visit the left subtree.
4. Visit the right subtree.

Visualization

```
    1
   / \
  2   3
preorder events: emit 1 -> emit 2 -> emit 3
```

Python implementation

```
from structures import build_tree

def preorder(root):
    output = []
    def visit(node):
        if node is None:
            return
        output.append(node.val)
        visit(node.left)
        visit(node.right)
    visit(root)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
preorder(build_tree([1,2,3,4]))

L4 preorder: output = []
root=N1(val=1,left=N2,right=N3); output=[]

L6 visit: if node is None:
node=N1(val=1,left=N2,right=N3); output=[]

L8 visit: output.append(node.val)
node=N1(val=1,left=N2,right=N3); output=[1]

L6 visit: if node is None:
node=N2(val=2,left=N4,right=None); output=[1]

L8 visit: output.append(node.val)
node=N2(val=2,left=N4,right=None); output=[1, 2]

L6 visit: if node is None:
node=N4(val=4,left=None,right=None); output=[1, 2]

L8 visit: output.append(node.val)
node=N4(val=4,left=None,right=None); output=[1, 2, 4]

L6 visit: if node is None:
node=None; output=[1, 2, 4]

... 6 intermediate state changes omitted ...

L6 visit: if node is None:
node=None; output=[1, 2, 4, 3]

L6 visit: if node is None:
node=None; output=[1, 2, 4, 3]
```

```
L10 visit: visit(node.right)
node=N1(val=1,left=N2,right=N3); output=[1, 2, 4, 3]

L11 preorder: visit(root)
root=N1(val=1,left=N2,right=N3); output=[1, 2, 4, 3]
```

Result: [1, 2, 4, 3]. The entire left subtree follows its parent before the right subtree.

```
preorder(None)

L4 preorder: output = []
root=None; output=[]

L6 visit: if node is None:
node=None; output=[]
```

Result: []. An empty subtree emits nothing.

Why the solution is correct

For an empty subtree no values are appended. For a real node, the append followed by correct left and right child traversals is exactly root-left-right order. By structural induction the root call produces the complete preorder. Every recursive call descends into a finite child subtree.

Time and space

O(n) time, O(n) output space and O(h) recursive stack space.

Common mistakes

Appending after visiting a child changes the traversal. Treating equal values as already visited would incorrectly omit distinct nodes.

Reasoning card

State: visit(node) appends the complete preorder sequence of that subtree. output already holds all earlier traversal events.

Invariant: Output contains the preorder of completed subtrees.

Decision to rehearse: Process current node before child calls.

Finish: Visit the right subtree

Transfer exercise

Write iterative preorder. Explain why a LIFO stack must receive the right child before the left child.

164. Recursive Inorder Traversal

Foundation | Module 17

Problem

Return a binary tree's values in recursive inorder: left subtree, root, right subtree. No binary-search-tree ordering is assumed. Return [] for an empty tree and preserve all node occurrences without mutation.

Examples

Example 1. The traversal follows structure, not numerical sorting.

```
inorder(build_tree([2,3,1]))
=> [3, 2, 1]
```

Example 2. A valid BST does yield sorted inorder.

```
inorder(build_tree([2,1,3]))
=> [1, 2, 3]
```

Example 3. The empty tree has no traversal events.

```
inorder(None)
=> []
```

Why this mechanism works

Move the emission between the two child calls. The left call must finish before the parent value is appended; the right call starts afterward. Inorder becomes sorted only when the input satisfies the BST property. The traversal algorithm itself works on arbitrary binary trees.

State and boundaries

visit(node) appends the complete inorder of its subtree; each frame waits for its left child before emitting its own value.

Protocol

1. Ignore None.

2. Visit left.

3. Append the current value.

4. Visit right.

Visualization

```
    2
   / \
  3   1
inorder: 3 -> 2 -> 1
this tree is not a BST, so output is not sorted
```

Python implementation

```
from structures import build_tree

def inorder(root):
    output = []
    def visit(node):
        if node is None:
            return
        visit(node.left)
        output.append(node.val)
        visit(node.right)
    visit(root)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
inorder(build_tree([2,3,1]))

L4 inorder: output = []
root=N1(val=2,left=N2,right=N3); output=[]

L6 visit: if node is None:
node=N1(val=2,left=N2,right=N3); output=[]

L6 visit: if node is None:
node=N2(val=3,left=None,right=None); output=[]

L6 visit: if node is None:
node=None; output=[]

L9 visit: output.append(node.val)
node=N2(val=3,left=None,right=None); output=[3]

L6 visit: if node is None:
node=None; output=[3]

L8 visit: visit(node.left)
node=N1(val=2,left=N2,right=N3); output=[3]

L9 visit: output.append(node.val)
node=N1(val=2,left=N2,right=N3); output=[3, 2]

... 2 intermediate state changes omitted ...

L9 visit: output.append(node.val)
node=N3(val=1,left=None,right=None); output=[3, 2, 1]

L6 visit: if node is None:
node=None; output=[3, 2, 1]
```

```
L10 visit: visit(node.right)
node=N1(val=2,left=N2,right=N3); output=[3, 2, 1]

L11 inorder: visit(root)
root=N1(val=2,left=N2,right=N3); output=[3, 2, 1]
```

Result: [3, 2, 1]. The traversal follows structure, not numerical sorting.

```
inorder(build_tree([2,1,3]))

L4 inorder: output = []
root=N1(val=2,left=N2,right=N3); output=[]

L6 visit: if node is None:
node=N1(val=2,left=N2,right=N3); output=[]

L6 visit: if node is None:
node=N2(val=1,left=None,right=None); output=[]

L6 visit: if node is None:
node=None; output=[]

L9 visit: output.append(node.val)
node=N2(val=1,left=None,right=None); output=[1]

L6 visit: if node is None:
node=None; output=[1]

L8 visit: visit(node.left)
node=N1(val=2,left=N2,right=N3); output=[1]

L9 visit: output.append(node.val)
node=N1(val=2,left=N2,right=N3); output=[1, 2]

... 2 intermediate state changes omitted ...

L9 visit: output.append(node.val)
node=N3(val=3,left=None,right=None); output=[1, 2, 3]

L6 visit: if node is None:
node=None; output=[1, 2, 3]

L10 visit: visit(node.right)
node=N1(val=2,left=N2,right=N3); output=[1, 2, 3]

L11 inorder: visit(root)
root=N1(val=2,left=N2,right=N3); output=[1, 2, 3]
```

Result: [1, 2, 3]. A valid BST does yield sorted inorder.

Why the solution is correct

The base emits the empty sequence. Assuming each child emits its own correct inorder, placing the current value between those sequences gives exactly left-root-right order. This inductive composition covers every node once and terminates on missing children.

Time and space

O(n) time, O(n) output space and O(h) recursive stack space.

Common mistakes

Expecting sorted output from an arbitrary tree confuses a traversal rule with a data-ordering invariant. Appending before the left call gives preorder instead.

Reasoning card

State: visit(node) appends the complete inorder of its subtree; each frame waits for its left child before emitting its own value.

Invariant: For a BST, emitted values are nondecreasing.

Decision to rehearse: Place the current-node action between child calls.

Finish: Visit right

Transfer exercise

Simulate recursion with a stack that stores ancestors whose left descent is unfinished or whose value has not yet been emitted.

165. Recursive Postorder Traversal

Bridge | Module 17

Problem

Return a binary tree's values in recursive postorder: left subtree, right subtree, root. Empty input returns []. Visit each node occurrence exactly once and leave all links intact.

Examples

Example 1. Each parent follows its complete child subtrees.

```
postorder(build_tree([1,2,3,4]))
=> [4, 2, 3, 1]
```

Example 2. No frame has a value to emit.

```
postorder(None)
=> []
```

Example 3. The leaf emits after its two empty child calls.

```
postorder(build_tree([7]))
=> [7]
```

Why this mechanism works

Postorder delays the current node's work until both child calls have completed. It is the natural order for computing a parent summary from child summaries, as the depth and diameter problems demonstrate. Here the completed work is simply appending the parent value.

State and boundaries

visit(node) appends a subtree's entire postorder. A frame emits its value only after both delegated traversals return.

Protocol

1. Ignore None.

2. Visit left.

3. Visit right.

4. Append the current value.

Visualization

```
    1
   / \
  2   3
postorder: finish 2 -> finish 3 -> emit 1
```

Python implementation

```
from structures import build_tree

def postorder(root):
    output = []
    def visit(node):
        if node is None:
            return
        visit(node.left)
        visit(node.right)
        output.append(node.val)
    visit(root)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
postorder(build_tree([1,2,3,4]))

L4 postorder: output = []
root=N1(val=1,left=N2,right=N3); output=[]

L6 visit: if node is None:
node=N1(val=1,left=N2,right=N3); output=[]

L6 visit: if node is None:
node=N2(val=2,left=N4,right=None); output=[]

L6 visit: if node is None:
node=N4(val=4,left=None,right=None); output=[]

L6 visit: if node is None:
node=None; output=[]

L6 visit: if node is None:
node=None; output=[]

L10 visit: output.append(node.val)
node=N4(val=4,left=None,right=None); output=[4]

L8 visit: visit(node.left)
node=N2(val=2,left=N4,right=None); output=[4]

... 6 intermediate state changes omitted ...

L10 visit: output.append(node.val)
node=N3(val=3,left=None,right=None); output=[4, 2, 3]

L9 visit: visit(node.right)
node=N1(val=1,left=N2,right=N3); output=[4, 2, 3]
```

```
L10 visit: output.append(node.val)
node=N1(val=1,left=N2,right=N3); output=[4, 2, 3, 1]

L11 postorder: visit(root)
root=N1(val=1,left=N2,right=N3); output=[4, 2, 3, 1]
```

Result: [4, 2, 3, 1]. Each parent follows its complete child subtrees.

```
postorder(None)

L4 postorder: output = []
root=None; output=[]

L6 visit: if node is None:
node=None; output=[]
```

Result: []. No frame has a value to emit.

Why the solution is correct

Empty subtrees append nothing. Correct left and right child sequences followed by the parent value give exactly left-right-root order. Every node is appended once in its own frame, and finite descent ends at missing children.

Time and space

O(n) time, O(n) output space and O(h) recursive stack space.

Common mistakes

Appending between child calls produces inorder. In an iterative version, merely pushing children without remembering whether the parent is ready can emit it too early.

Reasoning card

State: visit(node) appends a subtree's entire postorder. A frame emits its value only after both delegated traversals return.

Invariant: A parent is processed after its full subtree.

Decision to rehearse: Combine or emit only after both children are solved.

Finish: Append the current value

Transfer exercise

Use stack entries (node,expanded) to implement postorder without recursive calls.

166. Same Tree

Bridge | Module 17

Problem

Return whether two binary trees have the same shape and equal values at corresponding nodes. Two empty trees are equal; an empty and nonempty tree are not. Different object identities may still represent equal trees. Do not mutate either input.

Examples

Example 1. Different objects represent the same structure and values.

```
same_tree(build_tree([1,2,3]),build_tree([1,2,3]))
=> True
```

Example 2. The child appears on different sides.

```
same_tree(build_tree([1,2]),build_tree([1,None,2]))
=> False
```

Example 3. Both subtrees are empty.

```
same_tree(None,None)
=> True
```

Why this mechanism works

Compare the pair of current subtrees, not two independent traversal lists. Missing-node positions matter: equal values in different shapes are not the same tree. Resolve absence first, then compare values, then delegate the matching left pair and matching right pair. Short-circuiting stops as soon as a mismatch is proven.

State and boundaries

same_tree(a,b) decides equality of exactly these two corresponding subtrees.

Protocol

1. If either node is missing, succeed only if both are missing.
2. Reject unequal current values.
3. Recursively compare matching left children.
4. Recursively compare matching right children.

Visualization

```
tree A: 1 with left child 2
tree B: 1 with right child 2
same values, different missing-child positions -> False
```

Python implementation

```
from structures import build_tree

def same_tree(a, b):
    if a is None or b is None:
        return a is b
    return (a.val == b.val and same_tree(a.left, b.left)
            and same_tree(a.right, b.right))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
same_tree(build_tree([1,2,3]),build_tree([1,2,3]))

L4 same_tree: if a is None or b is None:
a=N1(val=1,left=N2,right=N3); b=N4(val=1,left=N5,right=N6)

L4 same_tree: if a is None or b is None:
a=N2(val=2,left=None,right=None); b=N5(val=2,left=None,right=None)

L4 same_tree: if a is None or b is None:
a=None; b=None

L4 same_tree: if a is None or b is None:
a=None; b=None

L4 same_tree: if a is None or b is None:
a=N3(val=3,left=None,right=None); b=N6(val=3,left=None,right=None)

L4 same_tree: if a is None or b is None:
a=None; b=None

L4 same_tree: if a is None or b is None:
a=None; b=None
```

Result: True. Different objects represent the same structure and values.

```
same_tree(build_tree([1,2]),build_tree([1,None,2]))

L4 same_tree: if a is None or b is None:
a=N1(val=1,left=N2,right=None); b=N3(val=1,left=None,right=N4)

L4 same_tree: if a is None or b is None:
a=N2(val=2,left=None,right=None); b=None
```

Result: False. The child appears on different sides.

Why the solution is correct

The missing-node test decides the base shapes correctly. Two nonempty trees are equal exactly when their root values and both corresponding child trees are equal. The recursive conjunction implements that definition, proving correctness by structural induction.

Time and space

O(min(n,m)+1) worst-case comparisons before exhaustion or mismatch for trees of n and m nodes; O(min(h1,h2)+1) recursive depth. Equal trees require O(n) work.

Common mistakes

Comparing preorder value lists without missing markers loses shape information. Comparing node identity rejects valid deep copies.

Reasoning card

State: same_tree(a,b) decides equality of exactly these two corresponding subtrees.

Invariant: Each call decides equality of exactly its two subtrees.

Decision to rehearse: Compare current pair, then recurse on matching child pairs.

Finish: Recursively compare matching right children

Transfer exercise

Define mirror equality by comparing opposite child sides, then use it to test tree symmetry.

167. Invert Binary Tree

Bridge | Module 17

Problem

Invert a binary tree in place and return its original root object. Every node's left and right subtrees exchange sides recursively. None remains None. Node values and node identities must be preserved.

Examples

Example 1. The deeper left child also changes side.

```
tree_values(invert_tree(build_tree([1,2,3,4])))
=> [1, 3, 2, None, None, None, 4]
```

Example 2. An empty subtree stays empty.

```
invert_tree(None)
=> None
```

Example 3. A singleton keeps its identity and shape.

```
root=build_tree([7])
(invert_tree(root) is root, tree_values(root))
=> (True, [7])
```

Why this mechanism works

The helper promises the mirror of one entire subtree. First obtain the mirrored old left and old right subtrees, then attach them on opposite sides. Saving both results in separate variables makes ownership explicit and prevents an overwritten pointer from being read as though it were still the original child.

State and boundaries

left and right are completed mirrors of the original child subtrees. The current root object is reused.

Protocol

1. Return None for an absent node.
2. Invert the old left subtree.
3. Invert the old right subtree.
4. Attach mirrored right as left and mirrored left as right.
5. Return the current node.

Visualization

```
before: 1(left=2,right=3)
mirror children: 2 and 3
after:  1(left=3,right=2)
the root object is unchanged
```

Python implementation

```
from structures import build_tree, tree_values

def invert_tree(root):
    if root is None:
        return None
    left = invert_tree(root.left)
    right = invert_tree(root.right)
    root.left, root.right = right, left
    return root
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
tree_values(invert_tree(build_tree([1,2,3,4])))

L4 invert_tree: if root is None:
root=N1(val=1,left=N2,right=N3)

L4 invert_tree: if root is None:
root=N2(val=2,left=N4,right=None)

L4 invert_tree: if root is None:
root=N4(val=4,left=None,right=None)

L4 invert_tree: if root is None:
root=None

L6 invert_tree: left = invert_tree(root.left)
root=N4(val=4,left=None,right=None); left=None

L4 invert_tree: if root is None:
root=None

L7 invert_tree: right = invert_tree(root.right)
root=N4(val=4,left=None,right=None); left=None; right=None

L6 invert_tree: left = invert_tree(root.left)
root=N2(val=2,left=N4,right=None); left=N4(val=4,left=None,right=None)

... 7 intermediate state changes omitted ...

L4 invert_tree: if root is None:
root=None

L7 invert_tree: right = invert_tree(root.right)
root=N3(val=3,left=None,right=None); left=None; right=None

L7 invert_tree: right = invert_tree(root.right)
root=N1(val=1,left=N2,right=N3); left=N2(val=2,left=None,right=N4);
    right=N3(val=3,left=None,right=None)
```

```
L8 invert_tree: root.left, root.right = right, left
root=N1(val=1,left=N3,right=N2); left=N2(val=2,left=None,right=N4);
    right=N3(val=3,left=None,right=None)
```

Result: [1, 3, 2, None, None, None, 4]. The deeper left child also changes side.

```
invert_tree(None)
```

```
L4 invert_tree: if root is None:
root=None
```

Result: None. An empty subtree stays empty.

Why the solution is correct

An empty tree is its own mirror. By the child-call contract, both returned subtrees are already mirrored; exchanging their sides around the unchanged root produces precisely the mirror of the original tree. Structural induction covers all nodes without allocating replacements.

Time and space

O(n) time and O(h) recursive stack space; mutation reuses all existing nodes.

Common mistakes

Sequential assignments that read root.left after overwriting it can attach the same subtree twice. Reversing values alone does not mirror the structure.

Reasoning card

State: left and right are completed mirrors of the original child subtrees. The current root object is reused.

Invariant: Returned subtree is the mirror of the original subtree.

Decision to rehearse: Invert child subtrees, then attach them on opposite sides.

Finish: Return the current node

Transfer exercise

Prove that inverting twice restores every original pointer relationship, then use that property in a test.

168. Diameter of Binary Tree

Interview | Module 17

Problem

Return the diameter of a binary tree: the largest number of edges on any simple path between two nodes. The path need not pass through the root. Return zero for an empty tree or a singleton. Leave the tree unchanged.

Examples

Example 1. The path 4-2-1-3 has three edges.

```
tree_diameter(build_tree([1,2,3,4,5]))
=> 3
```

Example 2. There is no nonzero path.

```
tree_diameter(None)
=> 0
```

Example 3. A four-node chain has three edges.

```
tree_diameter(build_tree([1,2,None,3,None,4]))
=> 3
```

Why this mechanism works

A downward path can choose only one child, while a complete path through a node can join two child arms. Therefore each call returns height to its parent and separately updates the best diameter. Using node-count child heights makes left_height+right_height equal the number of edges on the joined path.

State and boundaries

height(node) returns node-count height. best stores the maximum through-node edge count among nodes already summarized.

Protocol

1. Return height zero for None.
2. Obtain both child heights.
3. Update best with their sum.
4. Return one plus their maximum.
5. Return best after the root summary.

Visualization

```
left height 2 -- current node -- right height 3
through-node path = 2+3 = 5 edges
upward height = 1+max(2,3) = 4 nodes
```

Python implementation

```
from structures import build_tree

def tree_diameter(root):
    best = 0
    def height(node):
        nonlocal best
        if node is None:
            return 0
        left, right = height(node.left), height(node.right)
        best = max(best, left + right)
        return 1 + max(left, right)
    height(root)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
tree_diameter(build_tree([1,2,3,4,5]))

L4 tree_diameter: best = 0
root=N1(val=1,left=N2,right=N3); best=0

L7 height: if node is None:
node=N1(val=1,left=N2,right=N3); best=0

L7 height: if node is None:
node=N2(val=2,left=N4,right=N5); best=0

L7 height: if node is None:
node=N4(val=4,left=None,right=None); best=0

L7 height: if node is None:
node=None; best=0

L7 height: if node is None:
node=None; best=0

L9 height: left, right = height(node.left), height(node.right)
node=N4(val=4,left=None,right=None); best=0; left=0; right=0

L7 height: if node is None:
node=N5(val=5,left=None,right=None); best=0

... 8 intermediate state changes omitted ...

L9 height: left, right = height(node.left), height(node.right)
node=N3(val=3,left=None,right=None); best=2; left=0; right=0

L9 height: left, right = height(node.left), height(node.right)
node=N1(val=1,left=N2,right=N3); best=2; left=2; right=1
```

```
L10 height: best = max(best, left + right)
node=N1(val=1,left=N2,right=N3); best=3; left=2; right=1

L12 tree_diameter: height(root)
root=N1(val=1,left=N2,right=N3); best=3
```

Result: 3. The path 4-2-1-3 has three edges.

```
tree_diameter(None)

L4 tree_diameter: best = 0
root=None; best=0

L7 height: if node is None:
node=None; best=0
```

Result: 0. There is no nonzero path.

Why the solution is correct

Every simple path has a highest node, where it splits into at most one arm in each child subtree. The child heights give the longest possible arms through that node. Taking the maximum of these candidates over all nodes considers an optimum path, while no candidate exceeds a valid path length.

Time and space

O(n) time and O(h) recursive stack space. Each subtree height is computed once.

Common mistakes

Returning left+right to the parent offers a forked path that cannot be extended as one arm. Recomputing heights separately at every node can take quadratic time.

Reasoning card

State: height(node) returns node-count height. best stores the maximum through-node edge count among nodes already summarized.

Invariant: Height and best diameter have distinct meanings.

Decision to rehearse: Each call returns height while updating a global/nonlocal through-node path.

Finish: Return best after the root summary

Transfer exercise

Return the actual diameter endpoints along with height and deepest descendant information.

169. Balanced Binary Tree

Interview | Module 17

Problem

Return whether every node in a binary tree has left and right subtree heights differing by at most one. This condition must hold throughout the tree, not just at the root. An empty tree is balanced. Do not mutate it.

Examples

Example 1. Every local child-height difference is at most one.

```
is_balanced(build_tree([1,2,3,4,5]))
=> True
```

Example 2. The root has child heights two and zero.

```
is_balanced(build_tree([1,2,None,3]))
=> False
```

Example 3. There is no violating node.

```
is_balanced(None)
=> True
```

Why this mechanism works

A parent needs either an exact child height or evidence that the child is already unbalanced. Use -1 as a failure sentinel because real node-count heights are nonnegative. This combines validation and height computation into one traversal and can stop propagation immediately after a failure.

State and boundaries

height_or_fail(node) returns the exact nonnegative height if its entire subtree is balanced, otherwise -1.

Protocol

1. Return zero for None.

2. Summarize left and stop on failure.

3. Summarize right and stop on failure.

4. Reject a height difference greater than one.

5. Otherwise return one plus the larger height.

Visualization

```
child summaries: height 3 and height 1
abs(3-1)>1 -> return -1
every ancestor propagates failure, never recomputes height
```

Python implementation

```
from structures import build_tree

def is_balanced(root):
    def height_or_fail(node):
        if node is None:
            return 0
        left = height_or_fail(node.left)
        if left == -1:
            return -1
        right = height_or_fail(node.right)
        if right == -1 or abs(left - right) > 1:
            return -1
        return 1 + max(left, right)
    return height_or_fail(root) != -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
is_balanced(build_tree([1,2,3,4,5]))

L4 is_balanced: def height_or_fail(node):
root=N1(val=1,left=N2,right=N3)

L5 height_or_fail: if node is None:
node=N1(val=1,left=N2,right=N3)

L5 height_or_fail: if node is None:
node=N2(val=2,left=N4,right=N5)

L5 height_or_fail: if node is None:
node=N4(val=4,left=None,right=None)

L5 height_or_fail: if node is None:
node=None

L7 height_or_fail: left = height_or_fail(node.left)
node=N4(val=4,left=None,right=None); left=0

L5 height_or_fail: if node is None:
node=None

L10 height_or_fail: right = height_or_fail(node.right)
node=N4(val=4,left=None,right=None); left=0; right=0

... 10 intermediate state changes omitted ...

L7 height_or_fail: left = height_or_fail(node.left)
node=N3(val=3,left=None,right=None); left=0

L5 height_or_fail: if node is None:
node=None
```

```
L10 height_or_fail: right = height_or_fail(node.right)
node=N3(val=3,left=None,right=None); left=0; right=0

L10 height_or_fail: right = height_or_fail(node.right)
node=N1(val=1,left=N2,right=N3); left=2; right=1
```

Result: True. Every local child-height difference is at most one.

```
is_balanced(build_tree([1,2,None,3]))

L4 is_balanced: def height_or_fail(node):
root=N1(val=1,left=N2,right=None)

L5 height_or_fail: if node is None:
node=N1(val=1,left=N2,right=None)

L5 height_or_fail: if node is None:
node=N2(val=2,left=N3,right=None)

L5 height_or_fail: if node is None:
node=N3(val=3,left=None,right=None)

L5 height_or_fail: if node is None:
node=None

L7 height_or_fail: left = height_or_fail(node.left)
node=N3(val=3,left=None,right=None); left=0

L5 height_or_fail: if node is None:
node=None

L10 height_or_fail: right = height_or_fail(node.right)
node=N3(val=3,left=None,right=None); left=0; right=0

... 2 intermediate state changes omitted ...

L10 height_or_fail: right = height_or_fail(node.right)
node=N2(val=2,left=N3,right=None); left=1; right=0

L7 height_or_fail: left = height_or_fail(node.left)
node=N1(val=1,left=N2,right=None); left=2

L5 height_or_fail: if node is None:
node=None

L10 height_or_fail: right = height_or_fail(node.right)
node=N1(val=1,left=N2,right=None); left=2; right=0
```

Result: False. The root has child heights two and zero.

Why the solution is correct

The empty subtree satisfies the contract. If either child fails, the parent subtree contains an unbalanced node. Otherwise both child heights are exact; the local difference test is the only remaining condition. Returning the combined height preserves the contract for the parent.

Time and space

O(n) worst-case time and O(h) recursive stack space; short-circuit failures may visit fewer nodes.

Common mistakes

Checking only the root misses an unbalanced descendant. Treating -1 as an ordinary height can accidentally turn failure into a plausible parent result.

Reasoning card

State: height_or_fail(node) returns the exact nonnegative height if its entire subtree is balanced, otherwise -1.

Invariant: A non-failure return is the exact subtree height.

Decision to rehearse: Return height or a failure sentinel to stop repeated work.

Finish: Otherwise return one plus the larger height

Transfer exercise

Replace the sentinel with a tuple (balanced,height). Compare readability and the information available after failure.

170. Root-to-Leaf Target Sum

Interview | Module 17

Problem

Return whether a binary tree contains a root-to-leaf path whose integer values sum to target. A leaf has neither child. Negative and zero values are allowed. An empty tree has no such path, even when target is zero. Leave the tree unchanged.

Examples

Example 1. The left leaf completes a sum of seven.

```
has_path_sum(build_tree([5,2,9]),7)
=> True
```

Example 2. Stopping at the internal root is forbidden.

```
has_path_sum(build_tree([5,2]),5)
=> False
```

Example 3. An empty tree supplies no root-to-leaf path.

```
has_path_sum(None,0)
=> False
```

Example 4. Negative values are handled by subtraction without pruning.

```
has_path_sum(build_tree([3,-5]),-2)
=> True
```

Why this mechanism works

Pass the unfinished obligation down the tree: after visiting a node, subtract its value from remaining. Reaching zero at an internal node is insufficient because the requested path must end at a leaf. Negative values also make remaining<0 an invalid pruning rule.

State and boundaries

has_path_sum(node,remaining) asks whether a path starting at node and ending at a leaf contributes exactly remaining.

Protocol

1. Reject an absent node.

2. Subtract the current value.

3. At a leaf test whether the remainder is zero.

4. Otherwise ask either child to fulfill the new remainder.

Visualization

```
target 7
root 5 -> remainder 2
child 2 is a leaf -> remainder 0 -> success
an internal sum of 7 alone would not suffice
```

Python implementation

```
from structures import build_tree

def has_path_sum(root, target):
    if root is None:
        return False
    remaining = target - root.val
    if root.left is None and root.right is None:
        return remaining == 0
    return (has_path_sum(root.left, remaining)
            or has_path_sum(root.right, remaining))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
has_path_sum(build_tree([5,2,9]),7)

L4 has_path_sum: if root is None:
root=N1(val=5,left=N2,right=N3); target=7

L6 has_path_sum: remaining = target - root.val
root=N1(val=5,left=N2,right=N3); target=7; remaining=2

L4 has_path_sum: if root is None:
root=N2(val=2,left=None,right=None); target=2

L6 has_path_sum: remaining = target - root.val
root=N2(val=2,left=None,right=None); target=2; remaining=0
```

Result: True. The left leaf completes a sum of seven.

```
has_path_sum(build_tree([5,2]),5)

L4 has_path_sum: if root is None:
root=N1(val=5,left=N2,right=None); target=5

L6 has_path_sum: remaining = target - root.val
root=N1(val=5,left=N2,right=None); target=5; remaining=0

L4 has_path_sum: if root is None:
root=N2(val=2,left=None,right=None); target=0

L6 has_path_sum: remaining = target - root.val
root=N2(val=2,left=None,right=None); target=0; remaining=-2
```

```
L4 has_path_sum: if root is None:
root=None; target=0
```

Result: False. Stopping at the internal root is forbidden.

Why the solution is correct

At a leaf the only candidate path consists of that node, so the subtraction test is exact. At an internal node every valid path must continue through one child; the recursive disjunction covers precisely those options with the correct remaining sum. Finite subtree descent guarantees termination.

Time and space

O(n) worst-case time and O(h) recursive stack space. Early success can avoid exploring the other subtree.

Common mistakes

Accepting remaining==0 before checking leaf status permits incomplete paths. Rejecting negative remainders loses paths containing later negative values.

Reasoning card

State: has_path_sum(node,remaining) asks whether a path starting at node and ending at a leaf contributes exactly remaining.

Invariant: Remaining target is what the unfinished path must contribute.

Decision to rehearse: Subtract current value before delegating; succeed only at a leaf.

Finish: Otherwise ask either child to fulfill the new remainder

Transfer exercise

Enumerate all matching paths, adding choose-explore-undo path ownership to the same remaining-sum contract.

Module 18
Tree BFS, Binary Search Trees, and Tries

Use level frontiers, ordering invariants, and prefix trees for hierarchical search.

Different tree tasks exploit different structure. BFS groups by depth, BST search carries value-order constraints, and a trie consumes a character prefix along edges. Do not assume a generic binary tree is sorted. Wildcard trie search reconnects hierarchical lookup with the branching discipline from backtracking.

The ten-step progression

171. Binary Tree Level Order

172. Minimum Tree Depth with BFS

173. Right Side View

174. Zigzag Level Order

175. Validate a BST

176. Kth Smallest in BST

177. Lowest Common Ancestor in BST

178. Sorted Array to Balanced BST

179. Implement Trie

180. Wildcard Word Dictionary

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

171. Binary Tree Level Order

Foundation | Module 18

Problem

Return a binary tree's values grouped by depth, with each level ordered left to right. Empty input returns []. Preserve repeated values as distinct node occurrences and do not mutate the tree.

Examples

Example 1. Captured queue boundaries separate three depths.

```
level_order(build_tree([1,2,3,4,5]))
=> [[1], [2, 3], [4, 5]]
```

Example 2. There are no levels.

```
level_order(None)
=> []
```

Example 3. The root forms one complete level.

```
level_order(build_tree([7]))
=> [[7]]
```

Why this mechanism works

A FIFO queue already visits nodes in depth order, but grouping requires an explicit boundary. Capture the queue length before processing a level. Children appended during that loop belong to the next level and must wait until the next outer iteration.

State and boundaries

At each outer-loop start the queue contains exactly the next level, left to right. level holds the current layer's emitted values.

Protocol

1. Return [] for None.

2. Enqueue the root.

3. Capture the current queue length.

4. Remove exactly that many nodes, recording values and enqueuing left then right children.

5. Append the completed level.

Visualization

```
queue [1] -> level [1], next queue [2,3]
capture 2 -> level [2,3], next queue [4,5]
new children wait for the next outer loop
```

Python implementation

```
from collections import deque
from structures import build_tree

def level_order(root):
    if root is None:
        return []
    queue, output = deque([root]), []
    while queue:
        level = []
        width = len(queue)
        for _ in range(width):
            node = queue.popleft()
            level.append(node.val)
            if node.left is not None:
                queue.append(node.left)
            if node.right is not None:
                queue.append(node.right)
        output.append(level)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
level_order(build_tree([1,2,3,4,5]))

L5 level_order: if root is None:
root=N1(val=1,left=N2,right=N3)

L7 level_order: queue, output = deque([root]), []
root=N1(val=1,left=N2,right=N3); queue=deque([N1(val=1,left=N2,right=N3)]); output=[]

L9 level_order: level = []
root=N1(val=1,left=N2,right=N3); queue=deque([N1(val=1,left=N2,right=N3)]); output=[];
    level=[]

L10 level_order: width = len(queue)
root=N1(val=1,left=N2,right=N3); queue=deque([N1(val=1,left=N2,right=N3)]); output=[];
    level=[]; width=1

L11 level_order: for _ in range(width):
root=N1(val=1,left=N2,right=N3); queue=deque([N1(val=1,left=N2,right=N3)]); output=[];
    level=[]; width=1; _=0

L12 level_order: node = queue.popleft()
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[]; level=[]; width=1; _=0;
    node=N1(val=1,left=N2,right=N3)

L13 level_order: level.append(node.val)
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[]; level=[1]; width=1; _=0;
    node=N1(val=1,left=N2,right=N3)
```

```
L15 level_order: queue.append(node.left)
root=N1(val=1,left=N2,right=N3); queue=deque([N2(val=2,left=N4,right=N5)]); output=[];
    level=[1]; width=1; _=0; node=N1(val=1,left=N2,right=N3)

... 16 intermediate state changes omitted ...

L11 level_order: for _ in range(width):
root=N1(val=1,left=N2,right=N3); queue=deque([N5(val=5,left=None,right=None)]); output=[[1],
    [2, 3]]; level=[4]; width=2; _=1; node=N4(val=4,left=None,right=None)

L12 level_order: node = queue.popleft()
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[[1], [2, 3]]; level=[4]; width=2;
    _=1; node=N5(val=5,left=None,right=None)

L13 level_order: level.append(node.val)
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[[1], [2, 3]]; level=[4, 5]; width=2;
     _=1; node=N5(val=5,left=None,right=None)

L18 level_order: output.append(level)
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[[1], [2, 3], [4, 5]]; level=[4, 5];
    width=2; _=1; node=N5(val=5,left=None,right=None)
```

Result: [[1], [2, 3], [4, 5]]. Captured queue boundaries separate three depths.

```
level_order(None)

L5 level_order: if root is None:
root=None
```

Result: []. There are no levels.

Why the solution is correct

Initially the queue is the root level. Processing exactly that prefix emits the whole current level; appending its children in parent order creates the next level in left-to-right order. Induction over depth proves every group correct. Every node enters and leaves once.

Time and space

O(n) time, O(n) output and O(w) queue space, where w is maximum level width.

Common mistakes

Using a changing queue length as the level limit can consume newly appended children in the same group. A list pop(0) adds linear shifting cost.

Reasoning card

State: At each outer-loop start the queue contains exactly the next level, left to right. level holds the current layer's emitted values.

Invariant: The current queue prefix contains exactly one tree level.

Decision to rehearse: Capture queue length before each level.

Finish: Append the completed level

Transfer exercise

Return each level's average while keeping only its sum and node count.

172. Minimum Tree Depth with BFS

Foundation | Module 18

Problem

Return the minimum depth of a binary tree, measured in nodes on the shortest root-to-leaf path. A leaf has neither child. Empty input returns zero. A missing child of a nonleaf does not count as a zero-length route to a leaf.

Examples

Example 1. Node 3 is a leaf before the deeper leaf 4 is reached.

```
min_depth(build_tree([1,2,3,4]))
=> 2
```

Example 2. The sole leaf lies at the end of a three-node chain.

```
min_depth(build_tree([1,2,None,3]))
=> 3
```

Example 3. Empty depth is defined as zero.

```
min_depth(None)
=> 0
```

Why this mechanism works

BFS processes candidate endpoints by increasing depth. Consequently the first actual leaf removed from the queue is the nearest leaf, and search can stop. Carry depth alongside each node so the return value is already known at discovery of the endpoint.

State and boundaries

Queue entries (node,depth) have nondecreasing depth, and depth counts nodes from the root through that node.

Protocol

1. Return zero for None.
2. Enqueue (root,1).
3. Remove the next entry.
4. Return its depth if both children are absent.
5. Enqueue each real child with depth+1.

Visualization

```
1 -> left 2 -> left 3
depths 1,2,3; only node 3 is a leaf
missing right children do not terminate a path
```

Python implementation

```
from collections import deque
from structures import build_tree

def min_depth(root):
    if root is None:
        return 0
    queue = deque([(root, 1)])
    while queue:
        node, depth = queue.popleft()
        if node.left is None and node.right is None:
            return depth
        for child in (node.left, node.right):
            if child is not None:
                queue.append((child, depth + 1))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
min_depth(build_tree([1,2,3,4]))

L5 min_depth: if root is None:
root=N1(val=1,left=N2,right=N3)

L7 min_depth: queue = deque([(root, 1)])
root=N1(val=1,left=N2,right=N3); queue=deque([(N1(val=1,left=N2,right=N3), 1)])

L9 min_depth: node, depth = queue.popleft()
root=N1(val=1,left=N2,right=N3); queue=deque([]); node=N1(val=1,left=N2,right=N3); depth=1

L12 min_depth: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=N3); queue=deque([]); node=N1(val=1,left=N2,right=N3); depth=1;
    child=N2(val=2,left=N4,right=None)

L14 min_depth: queue.append((child, depth + 1))
root=N1(val=1,left=N2,right=N3); queue=deque([(N2(val=2,left=N4,right=None), 2)]);
    node=N1(val=1,left=N2,right=N3); depth=1; child=N2(val=2,left=N4,right=None)

L12 min_depth: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=N3); queue=deque([(N2(val=2,left=N4,right=None), 2)]);
    node=N1(val=1,left=N2,right=N3); depth=1; child=N3(val=3,left=None,right=None)

L14 min_depth: queue.append((child, depth + 1))
root=N1(val=1,left=N2,right=N3); queue=deque([(N2(val=2,left=N4,right=None), 2),
    (N3(val=3,left=None,right=None), 2)]); node=N1(val=1,left=N2,right=N3); depth=1;
    child=N3(val=3,left=None,right=None)

L9 min_depth: node, depth = queue.popleft()
root=N1(val=1,left=N2,right=N3); queue=deque([(N3(val=3,left=None,right=None), 2)]);
    node=N2(val=2,left=N4,right=None); depth=2; child=N3(val=3,left=None,right=None)
```

```
L12 min_depth: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=N3); queue=deque([(N3(val=3,left=None,right=None), 2)]);
    node=N2(val=2,left=N4,right=None); depth=2; child=N4(val=4,left=None,right=None)

L14 min_depth: queue.append((child, depth + 1))
root=N1(val=1,left=N2,right=N3); queue=deque([(N3(val=3,left=None,right=None), 2),
    (N4(val=4,left=None,right=None), 3)]); node=N2(val=2,left=N4,right=None); depth=2;
    child=N4(val=4,left=None,right=None)

L12 min_depth: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=N3); queue=deque([(N3(val=3,left=None,right=None), 2),
    (N4(val=4,left=None,right=None), 3)]); node=N2(val=2,left=N4,right=None); depth=2;
    child=None

L9 min_depth: node, depth = queue.popleft()
root=N1(val=1,left=N2,right=N3); queue=deque([(N4(val=4,left=None,right=None), 3)]);
    node=N3(val=3,left=None,right=None); depth=2; child=None
```

Result: 2. Node 3 is a leaf before the deeper leaf 4 is reached.

```
min_depth(build_tree([1,2,None,3]))

L5 min_depth: if root is None:
root=N1(val=1,left=N2,right=None)

L7 min_depth: queue = deque([(root, 1)])
root=N1(val=1,left=N2,right=None); queue=deque([(N1(val=1,left=N2,right=None), 1)])

L9 min_depth: node, depth = queue.popleft()
root=N1(val=1,left=N2,right=None); queue=deque([]); node=N1(val=1,left=N2,right=None); depth=1

L12 min_depth: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=None); queue=deque([]); node=N1(val=1,left=N2,right=None);
    depth=1; child=N2(val=2,left=N3,right=None)

L14 min_depth: queue.append((child, depth + 1))
root=N1(val=1,left=N2,right=None); queue=deque([(N2(val=2,left=N3,right=None), 2)]);
    node=N1(val=1,left=N2,right=None); depth=1; child=N2(val=2,left=N3,right=None)

L12 min_depth: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=None); queue=deque([(N2(val=2,left=N3,right=None), 2)]);
    node=N1(val=1,left=N2,right=None); depth=1; child=None

L9 min_depth: node, depth = queue.popleft()
root=N1(val=1,left=N2,right=None); queue=deque([]); node=N2(val=2,left=N3,right=None);
    depth=2; child=None

L12 min_depth: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=None); queue=deque([]); node=N2(val=2,left=N3,right=None);
    depth=2; child=N3(val=3,left=None,right=None)

L14 min_depth: queue.append((child, depth + 1))
root=N1(val=1,left=N2,right=None); queue=deque([(N3(val=3,left=None,right=None), 3)]);
    node=N2(val=2,left=N3,right=None); depth=2; child=N3(val=3,left=None,right=None)

L12 min_depth: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=None); queue=deque([(N3(val=3,left=None,right=None), 3)]);
    node=N2(val=2,left=N3,right=None); depth=2; child=None

L9 min_depth: node, depth = queue.popleft()
root=N1(val=1,left=N2,right=None); queue=deque([]); node=N3(val=3,left=None,right=None);
    depth=3; child=None
```

Result: 3. The sole leaf lies at the end of a three-node chain.

Why the solution is correct

FIFO insertion ensures every depth-d node is processed before any depth-(d+1) node. Thus no unprocessed leaf can have smaller depth than the first leaf returned. Enqueued depths are exact because each child adds one node to its parent's path.

Time and space

O(n) worst-case time and O(w) queue space; often stops before visiting the whole tree.

Common mistakes

Using 1+min(left_depth,right_depth) without handling missing children incorrectly shortens single-child chains. A node with only one missing child is not a leaf.

Reasoning card

State: Queue entries (node,depth) have nondecreasing depth, and depth counts nodes from the root through that node.

Invariant: Queue order is nondecreasing depth.

Decision to rehearse: The first leaf reached in level order has minimum depth.

Finish: Enqueue each real child with depth+1

Transfer exercise

Implement recursive minimum depth with explicit cases for zero, one, and two children.

173. Right Side View

Foundation | Module 18

Problem

Return the value of the rightmost node at each depth of a binary tree, from top to bottom. This models the right-side view. Empty input returns []. A visible node at a deeper level can belong to the root's left subtree.

Examples

Example 1. The left subtree supplies the only node on the deepest level.

```
right_side_view(build_tree([1,2,3,None,5]))
=> [1, 3, 5]
```

Example 2. No level has a visible node.

```
right_side_view(None)
=> []
```

Example 3. Every node of the chain is visible.

```
right_side_view(build_tree([1,2,None,3]))
=> [1, 2, 3]
```

Why this mechanism works

Keep ordinary left-to-right BFS. The final node removed from each captured layer is that layer's rightmost existing node. This avoids guessing visibility from whether a node is a right child; visibility is a property of an entire depth.

State and boundaries

The queue begins each layer in left-to-right order. i identifies a node's position inside the captured width.

Protocol

1. Enqueue the root if present.
2. Capture layer width.
3. Process that many nodes left to right.
4. Record the value at i==width-1.
5. Enqueue children for the next layer.

Visualization

```
    1
   / \
  2   3
   \
    5
right view [1,3,5]
```

Python implementation

```
from collections import deque
from structures import build_tree

def right_side_view(root):
    if root is None:
        return []
    queue, output = deque([root]), []
    while queue:
        width = len(queue)
        for i in range(width):
            node = queue.popleft()
            if i == width - 1:
                output.append(node.val)
            for child in (node.left, node.right):
                if child is not None:
                    queue.append(child)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
right_side_view(build_tree([1,2,3,None,5]))

L5 right_side_view: if root is None:
root=N1(val=1,left=N2,right=N3)

L7 right_side_view: queue, output = deque([root]), []
root=N1(val=1,left=N2,right=N3); queue=deque([N1(val=1,left=N2,right=N3)]); output=[]

L9 right_side_view: width = len(queue)
root=N1(val=1,left=N2,right=N3); queue=deque([N1(val=1,left=N2,right=N3)]); output=[]; width=1

L10 right_side_view: for i in range(width):
root=N1(val=1,left=N2,right=N3); queue=deque([N1(val=1,left=N2,right=N3)]); output=[];
    width=1; i=0

L11 right_side_view: node = queue.popleft()
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[]; width=1; i=0;
    node=N1(val=1,left=N2,right=N3)

L13 right_side_view: output.append(node.val)
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[1]; width=1; i=0;
    node=N1(val=1,left=N2,right=N3)

L14 right_side_view: for child in (node.left, node.right):
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[1]; width=1; i=0;
    node=N1(val=1,left=N2,right=N3); child=N2(val=2,left=None,right=N4)
```

```
L16 right_side_view: queue.append(child)
root=N1(val=1,left=N2,right=N3); queue=deque([N2(val=2,left=None,right=N4)]); output=[1];
    width=1; i=0; node=N1(val=1,left=N2,right=N3); child=N2(val=2,left=None,right=N4)

... 11 intermediate state changes omitted ...

L9 right_side_view: width = len(queue)
root=N1(val=1,left=N2,right=N3); queue=deque([N4(val=5,left=None,right=None)]); output=[1, 3];
     width=1; i=1; node=N3(val=3,left=None,right=None); child=None

L10 right_side_view: for i in range(width):
root=N1(val=1,left=N2,right=N3); queue=deque([N4(val=5,left=None,right=None)]); output=[1, 3];
     width=1; i=0; node=N3(val=3,left=None,right=None); child=None

L11 right_side_view: node = queue.popleft()
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[1, 3]; width=1; i=0;
    node=N4(val=5,left=None,right=None); child=None

L13 right_side_view: output.append(node.val)
root=N1(val=1,left=N2,right=N3); queue=deque([]); output=[1, 3, 5]; width=1; i=0;
    node=N4(val=5,left=None,right=None); child=None
```

Result: [1, 3, 5]. The left subtree supplies the only node on the deepest level.

```
right_side_view(None)

L5 right_side_view: if root is None:
root=None
```

Result: []. No level has a visible node.

Why the solution is correct

The level-order invariant places each depth's rightmost node last. Recording exactly that last value therefore selects one correct visible node per depth. Left-before-right child insertion preserves the invariant for the next layer.

Time and space

O(n) time, O(w) queue space and O(h) output values.

Common mistakes

Following only right child pointers misses deeper visible nodes in left subtrees. Recording the last value in the entire traversal loses layer boundaries.

Reasoning card

State: The queue begins each layer in left-to-right order. i identifies a node's position inside the captured width.

Invariant: Layer boundaries are explicit before processing.

Decision to rehearse: Record the final node processed in each BFS layer.

Finish: Enqueue children for the next layer

Transfer exercise

Use right-first DFS and record the first node reached at each previously unseen depth.

174. Zigzag Level Order

Foundation | Module 18

Problem

Return binary-tree levels alternating output direction: root level left to right, next level right to left, and so on. Empty input returns []. The tree and the actual BFS child order remain unchanged.

Examples

Example 1. Only the second level is reversed.

```
zigzag_levels(build_tree([1,2,3,4,5,6]))
=> [[1], [3, 2], [4, 5, 6]]
```

Example 2. No groups require processing.

```
zigzag_levels(None)
=> []
```

Example 3. The root keeps the initial direction.

```
zigzag_levels(build_tree([9]))
=> [[9]]
```

Why this mechanism works

Separate visitation order from presentation order. Collect every level with the reliable FIFO protocol, then reverse only the completed list for odd-numbered depths. Changing queue direction and child insertion together is unnecessary and creates more cases to reason about.

State and boundaries

levels is ordinary left-to-right level order. Index depth determines whether that complete output group is reversed.

Protocol

1. Obtain the level groups using kata 171.
2. Visit odd-numbered level indices.
3. Reverse each such list in place.
4. Return the groups.

Visualization

```
BFS groups: [1] [2,3] [4,5,6]
directions:   ->   <-       ->
output:     [1] [3,2] [4,5,6]
```

Python implementation

```
from kata_171 import level_order
from structures import build_tree

def zigzag_levels(root):
    levels = level_order(root)
    for depth in range(1, len(levels), 2):
        levels[depth].reverse()
    return levels
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
zigzag_levels(build_tree([1,2,3,4,5,6]))

L5 zigzag_levels: levels = level_order(root)
root=N1(val=1,left=N2,right=N3); levels=[[1], [2, 3], [4, 5, 6]]

L6 zigzag_levels: for depth in range(1, len(levels), 2):
root=N1(val=1,left=N2,right=N3); levels=[[1], [2, 3], [4, 5, 6]]; depth=1

L7 zigzag_levels: levels[depth].reverse()
root=N1(val=1,left=N2,right=N3); levels=[[1], [3, 2], [4, 5, 6]]; depth=1
```

Result: [[1], [3, 2], [4, 5, 6]]. Only the second level is reversed.

```
zigzag_levels(None)

L5 zigzag_levels: levels = level_order(root)
root=None; levels=[]
```

Result: []. No groups require processing.

Why the solution is correct

Kata 171 produces the correct nodes for every depth in left-to-right order. Reversing exactly the odd-depth lists changes only their presentation direction and preserves membership. Even-depth groups remain unchanged, satisfying the alternating contract.

Time and space

O(n) time including all reversals, O(n) output space and O(w) BFS auxiliary queue space inherited from kata 171.

Common mistakes

Reversing the whole flattened output changes level order. Mutating traversal mechanics is unnecessary when only emitted order must change.

Reasoning card

State: levels is ordinary left-to-right level order. Index depth determines whether that complete output group is reversed.

Invariant: Each level is collected completely before direction reversal.

Decision to rehearse: Traversal may stay FIFO while only level output order changes.

Finish: Return the groups

Transfer exercise

Write a streaming version that yields one completed level at a time instead of retaining every group.

175. Validate a BST

Bridge | Module 18

Problem

Return whether a binary tree is a strict binary search tree: all values in a node's left subtree are smaller, and all values in its right subtree are larger. Duplicate values invalidate the tree. Values are integers; an empty tree is valid.

Examples

Example 1. The value 4 violates an inherited lower bound of 5.

```
valid_bst(build_tree([5,3,7,None,None,4,8]))
=> False
```

Example 2. All strict subtree bounds hold.

```
valid_bst(build_tree([2,1,3]))
=> True
```

Example 3. Equal values are excluded.

```
valid_bst(build_tree([2,2]))
=> False
```

Example 4. There is no ordering violation.

```
valid_bst(None)
=> True
```

Why this mechanism works

A node must satisfy every ancestor's ordering restriction, not merely its parent's. Pass an open allowed interval down each call. Moving left replaces the upper bound with the current value; moving right replaces the lower bound. None represents an absent bound without assuming a fixed integer range.

State and boundaries

valid(node,low,high) requires every node in that subtree to lie strictly between the inherited bounds.

Protocol

1. Accept None.

2. Reject a value <=low or >=high when that bound exists.

3. Validate left with the current value as upper bound.

4. Validate right with it as lower bound.

Visualization

```
root 5 -> right child 7 -> left child 4
node 4 inherits interval (5,7)
4<7 locally, but 4<=5 violates the ancestor bound
```

Python implementation

```
from structures import build_tree

def valid_bst(root):
    def valid(node, low, high):
        if node is None:
            return True
        if low is not None and node.val <= low:
            return False
        if high is not None and node.val >= high:
            return False
        return (valid(node.left, low, node.val)
                and valid(node.right, node.val, high))
    return valid(root, None, None)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
valid_bst(build_tree([5,3,7,None,None,4,8]))

L4 valid_bst: def valid(node, low, high):
root=N1(val=5,left=N2,right=N3)

L5 valid: if node is None:
node=N1(val=5,left=N2,right=N3); low=None; high=None

L5 valid: if node is None:
node=N2(val=3,left=None,right=None); low=None; high=5

L5 valid: if node is None:
node=None; low=None; high=3

L5 valid: if node is None:
node=None; low=3; high=5

L5 valid: if node is None:
node=N3(val=7,left=N4,right=N5); low=5; high=None

L5 valid: if node is None:
node=N4(val=4,left=None,right=None); low=5; high=7
```

Result: False. The value 4 violates an inherited lower bound of 5.

```
valid_bst(build_tree([2,1,3]))

L4 valid_bst: def valid(node, low, high):
root=N1(val=2,left=N2,right=N3)

L5 valid: if node is None:
```

```
node=N1(val=2,left=N2,right=N3); low=None; high=None

L5 valid: if node is None:
node=N2(val=1,left=None,right=None); low=None; high=2

L5 valid: if node is None:
node=None; low=None; high=1

L5 valid: if node is None:
node=None; low=1; high=2

L5 valid: if node is None:
node=N3(val=3,left=None,right=None); low=2; high=None

L5 valid: if node is None:
node=None; low=2; high=3

L5 valid: if node is None:
node=None; low=3; high=None
```

Result: True. All strict subtree bounds hold.

Why the solution is correct

Each inherited bound captures a necessary ancestor restriction. A successful current test plus recursively valid bounded child subtrees enforces all required comparisons, including distant ancestors. Conversely, a strict BST satisfies every propagated bound, so no valid tree is rejected.

Time and space

O(n) worst-case time and O(h) recursive stack space.

Common mistakes

Checking only immediate children accepts a right-subtree descendant smaller than the root. Non-strict comparisons would silently permit duplicates.

Reasoning card

State: valid(node,low,high) requires every node in that subtree to lie strictly between the inherited bounds.

Invariant: Every node in a call must lie strictly within its bounds.

Decision to rehearse: Local parent comparison is insufficient; carry global allowed range.

Finish: Validate right with it as lower bound

Transfer exercise

Validate by iterative inorder with a previous-value variable. Explain why a strictly increasing sequence is equivalent here.

176. Kth Smallest in BST

Bridge | Module 18

Problem

Given a valid strict BST and one-based rank k, return its kth smallest value. Raise ValueError if k<=0 or the tree contains fewer than k nodes. Do not mutate the tree. The algorithm may stop before visiting the entire tree.

Examples

Example 1. The third inorder emission is the root value.

```
kth_smallest_bst(build_tree([3,1,4,None,2]),3)
=> 3
```

Example 2. The first emission is the leftmost node.

```
kth_smallest_bst(build_tree([2,1,3]),1)
=> 1
```

Example 3. A singleton has one valid rank.

```
kth_smallest_bst(build_tree([7]),1)
=> 7
```

Why this mechanism works

Inorder of a BST is increasing, so ranking becomes counting emission events. An explicit stack performs the left descent, then emits the next smallest node and moves to its right subtree. The stack contains unfinished ancestors rather than all values in sorted order.

State and boundaries

stack holds ancestors waiting for their inorder emission. k is the number of remaining emissions needed to reach the requested rank.

Protocol

1. Reject nonpositive k.
2. Push the entire current left spine.
3. Pop and emit the next node, decrementing k.
4. Return its value when k becomes zero.
5. Continue from its right child.
6. Reject if traversal ends first.

Visualization

```
BST values in inorder: 1 -> 2 -> 3 -> 4
k starts 3; remaining: 2,1,0
return at the third pop, without collecting all values
```

Python implementation

```
from structures import build_tree

def kth_smallest_bst(root, k):
    if k <= 0:
        raise ValueError('k must be positive')
    stack, node = [], root
    while node is not None or stack:
        while node is not None:
            stack.append(node)
            node = node.left
        node = stack.pop()
        k -= 1
        if k == 0:
            return node.val
        node = node.right
    raise ValueError('rank exceeds tree size')
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
kth_smallest_bst(build_tree([3,1,4,None,2]),3)

L4 kth_smallest_bst: if k <= 0:
root=N1(val=3,left=N2,right=N3); k=3

L6 kth_smallest_bst: stack, node = [], root
root=N1(val=3,left=N2,right=N3); k=3; stack=[]; node=N1(val=3,left=N2,right=N3)

L9 kth_smallest_bst: stack.append(node)
root=N1(val=3,left=N2,right=N3); k=3; stack=[N1(val=3,left=N2,right=N3)];
    node=N1(val=3,left=N2,right=N3)

L10 kth_smallest_bst: node = node.left
root=N1(val=3,left=N2,right=N3); k=3; stack=[N1(val=3,left=N2,right=N3)];
    node=N2(val=1,left=None,right=N4)

L9 kth_smallest_bst: stack.append(node)
root=N1(val=3,left=N2,right=N3); k=3; stack=[N1(val=3,left=N2,right=N3),
    N2(val=1,left=None,right=N4)]; node=N2(val=1,left=None,right=N4)

L10 kth_smallest_bst: node = node.left
root=N1(val=3,left=N2,right=N3); k=3; stack=[N1(val=3,left=N2,right=N3),
    N2(val=1,left=None,right=N4)]; node=None

L11 kth_smallest_bst: node = stack.pop()
root=N1(val=3,left=N2,right=N3); k=3; stack=[N1(val=3,left=N2,right=N3)];
    node=N2(val=1,left=None,right=N4)

L12 kth_smallest_bst: k -= 1
root=N1(val=3,left=N2,right=N3); k=2; stack=[N1(val=3,left=N2,right=N3)];
```

```
    node=N2(val=1,left=None,right=N4)

... 4 intermediate state changes omitted ...

L12 kth_smallest_bst: k -= 1
root=N1(val=3,left=N2,right=N3); k=1; stack=[N1(val=3,left=N2,right=N3)];
    node=N4(val=2,left=None,right=None)

L15 kth_smallest_bst: node = node.right
root=N1(val=3,left=N2,right=N3); k=1; stack=[N1(val=3,left=N2,right=N3)]; node=None

L11 kth_smallest_bst: node = stack.pop()
root=N1(val=3,left=N2,right=N3); k=1; stack=[]; node=N1(val=3,left=N2,right=N3)

L12 kth_smallest_bst: k -= 1
root=N1(val=3,left=N2,right=N3); k=0; stack=[]; node=N1(val=3,left=N2,right=N3)
```

Result: 3. The third inorder emission is the root value.

```
kth_smallest_bst(build_tree([2,1,3]),1)

L4 kth_smallest_bst: if k <= 0:
root=N1(val=2,left=N2,right=N3); k=1

L6 kth_smallest_bst: stack, node = [], root
root=N1(val=2,left=N2,right=N3); k=1; stack=[]; node=N1(val=2,left=N2,right=N3)

L9 kth_smallest_bst: stack.append(node)
root=N1(val=2,left=N2,right=N3); k=1; stack=[N1(val=2,left=N2,right=N3)];
    node=N1(val=2,left=N2,right=N3)

L10 kth_smallest_bst: node = node.left
root=N1(val=2,left=N2,right=N3); k=1; stack=[N1(val=2,left=N2,right=N3)];
    node=N2(val=1,left=None,right=None)

L9 kth_smallest_bst: stack.append(node)
root=N1(val=2,left=N2,right=N3); k=1; stack=[N1(val=2,left=N2,right=N3),
    N2(val=1,left=None,right=None)]; node=N2(val=1,left=None,right=None)

L10 kth_smallest_bst: node = node.left
root=N1(val=2,left=N2,right=N3); k=1; stack=[N1(val=2,left=N2,right=N3),
    N2(val=1,left=None,right=None)]; node=None

L11 kth_smallest_bst: node = stack.pop()
root=N1(val=2,left=N2,right=N3); k=1; stack=[N1(val=2,left=N2,right=N3)];
    node=N2(val=1,left=None,right=None)

L12 kth_smallest_bst: k -= 1
root=N1(val=2,left=N2,right=N3); k=0; stack=[N1(val=2,left=N2,right=N3)];
    node=N2(val=1,left=None,right=None)
```

Result: 1. The first emission is the leftmost node.

Why the solution is correct

Every pop is the next inorder node because all of its left subtree is complete and its right subtree has not begun. BST ordering makes these emissions globally increasing. The kth pop therefore has the requested rank; exhaustion proves that rank does not exist.

Time and space

O(h+k) time for a valid rank, bounded by O(n), and O(h) auxiliary stack space. Invalid large k requires O(n) traversal.

Common mistakes

Using zero-based counting with a one-based contract returns the wrong rank. A BST can be skewed, so O(log n) height is not guaranteed.

Reasoning card

State: stack holds ancestors waiting for their inorder emission. k is the number of remaining emissions needed to reach the requested rank.

Invariant: Visited inorder nodes are the smallest values in order.

Decision to rehearse: Exploit sorted inorder traversal and stop at count k.

Finish: Reject if traversal ends first

Transfer exercise

Augment nodes with subtree sizes to answer repeated rank queries without traversing k nodes.

177. Lowest Common Ancestor in BST

Bridge | Module 18

Problem

Given a strict BST and two values p and q that are both present, return the node object of their lowest common ancestor. A node is its own ancestor, and p may equal q. Leave the tree unchanged. Presence is a precondition, not an extra search performed here.

Examples

Example 1. The targets first split at the root.

```
lca_bst(build_tree([6,2,8,0,4,7,9]),2,8).val
=> 6
```

Example 2. A target node can itself be the ancestor.

```
lca_bst(build_tree([6,2,8,0,4]),2,4).val
=> 2
```

Example 3. Equal targets have that node as their lowest ancestor.

```
lca_bst(build_tree([2,1,3]),3,3).val
=> 3
```

Why this mechanism works

If both target values are smaller than the current value, their common ancestor must remain in the left subtree; the symmetric rule holds on the right. Otherwise the targets split across the current node, or one equals it. That is the first point where neither single child can contain both.

State and boundaries

The current root remains an ancestor candidate whose subtree contains both target values.

Protocol

1. While the current node exists, compare both targets with its value.
2. Move left if both are smaller.
3. Move right if both are larger.
4. Otherwise return the current node.

Visualization

```
at 6: targets 2 and 8 split -> return 6
at 6: targets 2 and 4 both smaller -> move left
at 2: one target equals current -> return 2
```

Python implementation

```
from structures import build_tree

def lca_bst(root, p, q):
    while root is not None:
        if p < root.val and q < root.val:
            root = root.left
        elif p > root.val and q > root.val:
            root = root.right
        else:
            return root
    return None
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
lca_bst(build_tree([6,2,8,0,4,7,9]),2,8).val

L4 lca_bst: while root is not None:
root=N1(val=6,left=N2,right=N3); p=2; q=8
```

Result: 6. The targets first split at the root.

```
lca_bst(build_tree([6,2,8,0,4]),2,4).val

L4 lca_bst: while root is not None:
root=N1(val=6,left=N2,right=N3); p=2; q=4

L6 lca_bst: root = root.left
root=N2(val=2,left=N4,right=N5); p=2; q=4
```

Result: 2. A target node can itself be the ancestor.

Why the solution is correct

Each same-side move retains both targets and discards ancestors that cannot be lowest. At a split, no child subtree contains both targets; when one target equals the current node, no descendant can be its ancestor. The returned node is therefore the lowest common ancestor.

Time and space

O(h) time and O(1) auxiliary space; O(n) time in a skewed BST.

Common mistakes

Moving on only one target's comparison can discard the other target. Without the presence precondition, a split point need not be an ancestor of existing target nodes.

Reasoning card

State: The current root remains an ancestor candidate whose subtree contains both target values.

Invariant: Current node remains an ancestor candidate for both targets.

Decision to rehearse: Move left/right only when both targets lie on the same side.

Finish: Otherwise return the current node

Transfer exercise

Support possibly absent targets by validating membership, and state the extra traversal cost.

178. Sorted Array to Balanced BST

Interview | Module 18

Problem

Build a height-balanced strict BST from a strictly increasing integer list. Return newly allocated nodes and leave the list unchanged. Choose mid=(left+right)//2 on each half-open range, so even-sized ranges use their upper middle. Empty input returns None.

Examples

Example 1. The upper-middle rule fixes a deterministic balanced shape.

```
tree_values(sorted_array_to_bst([1,2,3,4]))
=> [3, 2, 4, 1]
```

Example 2. An empty interval creates no node.

```
sorted_array_to_bst([])
=> None
```

Example 3. Both child ranges are empty.

```
tree_values(sorted_array_to_bst([7]))
=> [7]
```

Why this mechanism works

The middle value separates smaller values from larger ones and keeps the two child sizes within one. Recursing on index ranges avoids slicing copies. The call contract combines two properties: its inorder reproduces the assigned range, and its subtrees are height-balanced.

State and boundaries

build(left,right) constructs exactly nums[left:right]. The midpoint belongs to the current root and is excluded from both children.

Protocol

1. Return None when left==right.
2. Choose the midpoint.
3. Allocate its node.
4. Build [left,mid) as left child.
5. Build [mid+1,right) as right child.

6. Return the node.

Visualization

```
[1,2,3,4] -> mid index 2 -> root 3
left range [1,2] -> root 2 with left 1
right range [4] -> root 4
```

Python implementation

```
from structures import TreeNode, tree_values

def sorted_array_to_bst(nums):
    def build(left, right):
        if left == right:
            return None
        mid = (left + right) // 2
        return TreeNode(nums[mid], build(left, mid), build(mid + 1, right))
    return build(0, len(nums))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
tree_values(sorted_array_to_bst([1,2,3,4]))

L4 sorted_array_to_bst: def build(left, right):
nums=[1, 2, 3, 4]

L5 build: if left == right:
left=0; right=4; nums=[1, 2, 3, 4]

L7 build: mid = (left + right) // 2
left=0; right=4; nums=[1, 2, 3, 4]; mid=2

L5 build: if left == right:
left=0; right=2; nums=[1, 2, 3, 4]

L7 build: mid = (left + right) // 2
left=0; right=2; nums=[1, 2, 3, 4]; mid=1

L5 build: if left == right:
left=0; right=1; nums=[1, 2, 3, 4]

L7 build: mid = (left + right) // 2
left=0; right=1; nums=[1, 2, 3, 4]; mid=0

L5 build: if left == right:
left=0; right=0; nums=[1, 2, 3, 4]

... 2 intermediate state changes omitted ...

L5 build: if left == right:
left=3; right=4; nums=[1, 2, 3, 4]

L7 build: mid = (left + right) // 2
left=3; right=4; nums=[1, 2, 3, 4]; mid=3

L5 build: if left == right:
```

```
left=3; right=3; nums=[1, 2, 3, 4]

L5 build: if left == right:
left=4; right=4; nums=[1, 2, 3, 4]
```

Result: [3, 2, 4, 1]. The upper-middle rule fixes a deterministic balanced shape.

```
sorted_array_to_bst([])

L4 sorted_array_to_bst: def build(left, right):
nums=[]

L5 build: if left == right:
left=0; right=0; nums=[]
```

Result: None. An empty interval creates no node.

Why the solution is correct

Both child ranges are strictly ordered around the midpoint, so recursively valid child BSTs form a valid parent BST whose inorder reproduces the range. Repeated near-equal size splits give subtree heights differing by at most one. Each child range is smaller, establishing termination.

Time and space

O(n) time, O(n) new-node output and O(log(n+1)) recursive stack space.

Common mistakes

Including mid in a child repeats a value and can prevent progress. Slicing is unnecessary when the contract already identifies an index interval.

Reasoning card

State: build(left,right) constructs exactly nums[left:right]. The midpoint belongs to the current root and is excluded from both children.

Invariant: Inorder traversal of the returned tree reproduces the input.

Decision to rehearse: Split sorted input into smaller left/right ranges.

Finish: Return the node

Transfer exercise

Accept sorted duplicate values under an explicitly chosen BST duplicate policy. Does midpoint splitting still guarantee both that policy and balance?

179. Implement Trie

Interview | Module 18

Problem

Implement Trie with insert(word), search(word), and startsWith(prefix) for strings. Matching is exact and case-sensitive. insert returns None. Empty strings are supported: an empty prefix always exists, while search('') becomes true only after inserting the empty word. Repeated insertion is harmless.

Examples

Example 1. Prefix existence and exact-word membership differ.

```
run_operations(Trie,[("insert","apple"),("search","app"),("startsWith","app"),("insert
    ","app"),("search","app")])
=> [None, False, True, None, True]
```

Example 2. The root represents the empty prefix and can also become terminal.

```
run_operations(Trie,[("startsWith",""),("search",""),("insert",""),("search","")])
=> [True, False, None, True]
```

Example 3. Character matching is case-sensitive.

```
run_operations(Trie,[("insert","A"),("search","a"),("search","A")])
=> [None, False, True]
```

Why this mechanism works

Each edge consumes one character, so a root-to-node path identifies a prefix. A separate terminal flag distinguishes a complete inserted word from a prefix of a longer word. Search and prefix lookup share traversal; only exact search needs the terminal check.

State and boundaries

Each node has a children dictionary and terminal flag. The current node represents all characters consumed so far.

Protocol

1. Start at the root.
2. During insertion create missing character edges.
3. Mark the final node terminal.
4. During lookup fail on a missing edge.

5. For exact search require terminal; for prefix search require only a reached node.

Visualization

```
root - a - p - p - l - e*
prefix 'app': path exists
word 'app': false until its node is marked *
```

Python implementation

```
from structures import run_operations

class TrieNode:
    def __init__(self):
        self.children = {}
        self.terminal = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.terminal = True

    def _walk(self, text):
        node = self.root
        for char in text:
            node = node.children.get(char)
            if node is None:
                return None
        return node

    def search(self, word):
        node = self._walk(word)
        return node is not None and node.terminal

    def startsWith(self, prefix):
        return self._walk(prefix) is not None
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(Trie,[("insert","apple"),("search","app"),("startsWith","app"),("insert
    ","app"),("search","app")])

L5 __init__: self.children = {}
self=TrieNode{'children':{}}

L6 __init__: self.terminal = False
self=TrieNode{'children':{}, 'terminal':False}
```

```
L10 __init__: self.root = TrieNode()
self=Trie{'root':TrieNode...}

L13 insert: node = self.root
self=Trie{'root':TrieNode...}; word='apple'; node=TrieNode{'children':{}, 'terminal':False}

L14 insert: for char in word:
self=Trie{'root':TrieNode...}; word='apple'; node=TrieNode{'children':{}, 'terminal':False};
    char='a'

L5 __init__: self.children = {}
self=TrieNode{'children':{}}

L6 __init__: self.terminal = False
self=TrieNode{'children':{}, 'terminal':False}

L16 insert: node.children[char] = TrieNode()
self=Trie{'root':TrieNode...}; word='apple'; node=TrieNode{'children':{...:...},
    'terminal':False}; char='a'

... 34 intermediate state changes omitted ...

L22 _walk: for char in text:
self=Trie{'root':TrieNode...}; text='app'; node=TrieNode{'children':{...:...},
    'terminal':False}; char='a'

L22 _walk: for char in text:
self=Trie{'root':TrieNode...}; text='app'; node=TrieNode{'children':{...:...},
    'terminal':False}; char='p'

L23 _walk: node = node.children.get(char)
self=Trie{'root':TrieNode...}; text='app'; node=TrieNode{'children':{...:...},
    'terminal':True}; char='p'

L29 search: node = self._walk(word)
self=Trie{'root':TrieNode...}; word='app'; node=TrieNode{'children':{...:...},
    'terminal':True}
```

Result: [None, False, True, None, True]. Prefix existence and exact-word membership differ.

```
run_operations(Trie,[("startsWith",""),("search",""),("insert",""),("search","")])

L5 __init__: self.children = {}
self=TrieNode{'children':{}}

L6 __init__: self.terminal = False
self=TrieNode{'children':{}, 'terminal':False}

L10 __init__: self.root = TrieNode()
self=Trie{'root':TrieNode...}

L21 _walk: node = self.root
self=Trie{'root':TrieNode...}; text=''; node=TrieNode{'children':{}, 'terminal':False}

L33 startsWith: return self._walk(prefix) is not None
self=Trie{'root':TrieNode...}; prefix=''

L21 _walk: node = self.root
self=Trie{'root':TrieNode...}; text=''; node=TrieNode{'children':{}, 'terminal':False}

L29 search: node = self._walk(word)
self=Trie{'root':TrieNode...}; word=''; node=TrieNode{'children':{}, 'terminal':False}

L13 insert: node = self.root
```

```
self=Trie{'root':TrieNode...}; word=''; node=TrieNode{'children':{}, 'terminal':False}

L18 insert: node.terminal = True
self=Trie{'root':TrieNode...}; word=''; node=TrieNode{'children':{}, 'terminal':True}

L21 _walk: node = self.root
self=Trie{'root':TrieNode...}; text=''; node=TrieNode{'children':{}, 'terminal':True}

L29 search: node = self._walk(word)
self=Trie{'root':TrieNode...}; word=''; node=TrieNode{'children':{}, 'terminal':True}
```

Result: [True, False, None, True]. The root represents the empty prefix and can also become terminal.

Why the solution is correct

Insertion creates exactly the path spelling the word and marks its endpoint. Lookup follows that unique character path, so a reached node proves prefix existence; its terminal flag proves exact insertion. Each edge consumes a character, bounding all operations by input length.

Time and space

Expected O(L) time per length-L operation with dictionary edges. O(S) stored nodes/edges for total inserted characters S, plus the root; lookup uses O(1) auxiliary state.

Common mistakes

Treating every reached node as a word makes search('app') succeed after inserting only 'apple'. Using a character as a terminal sentinel can collide with valid input; a separate flag avoids that.

Reasoning card

State: Each node has a children dictionary and terminal flag. The current node represents all characters consumed so far.

Invariant: A root-to-node path represents exactly one prefix.

Decision to rehearse: Each edge consumes one character; mark end-of-word separately.

Finish: For exact search require terminal; for prefix search require only a reached node

Transfer exercise

Add deletion without removing nodes still needed by another word or prefix.

180. Wildcard Word Dictionary

Interview | Module 18

Problem

Implement a word dictionary with addWord(word) and search(pattern), where '.' matches exactly one arbitrary character. Stored words contain no literal dot. All other characters match exactly. Empty words and patterns are supported; addWord returns None.

Examples

Example 1. A dot consumes exactly one character and completion must be terminal.

```
run_operations(WordDictionary,[("addWord","bad"),("addWord","dad"),("search",".ad"),("
    search","b.")])
=> [None, None, True, False]
```

Example 2. The empty word cannot supply a wildcard character.

```
run_operations(WordDictionary,[("addWord",""),("search",""),("search",".")])
=> [None, True, False]
```

Example 3. Literal edges still restrict wildcard searches.

```
run_operations(WordDictionary,[("addWord","cat"),("search","c.t"),("search","dog")])
=> [None, True, False]
```

Why this mechanism works

Reuse the trie from kata 179. A literal pattern character chooses one edge, while a wildcard must try every child edge. The recursive state needs both the current trie node and the pattern index. Reaching the end of the pattern succeeds only at a terminal node, even after wildcards.

State and boundaries

match(node,i) asks whether some stored suffix below node matches pattern[i:]. The prefix before i has already matched.

Protocol

1. Insert words using the trie.

2. At pattern end return the terminal flag.

3. On a literal follow its child if present.

4. On a dot try every child at index i+1.

5. Succeed if any branch completes.

Visualization

```
stored bad, dad, mad
pattern .ad -> branch b/d/m -> literal a -> literal d -> terminal
pattern b.  -> ends at nonterminal prefix -> False
```

Python implementation

```
from kata_179 import Trie
from structures import run_operations

class WordDictionary:
    def __init__(self):
        self.trie = Trie()

    def addWord(self, word):
        self.trie.insert(word)

    def search(self, pattern):
        def match(node, i):
            if i == len(pattern):
                return node.terminal
            char = pattern[i]
            if char == '.':
                return any(match(child, i + 1) for child in node.children.values())
            child = node.children.get(char)
            return child is not None and match(child, i + 1)
        return match(self.trie.root, 0)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(WordDictionary,[("addWord","bad"),("addWord","dad"),("search",".ad"),("
    search","b.")])

L6 __init__: self.trie = Trie()
self=WordDictionary{'trie':Trie...}

L9 addWord: self.trie.insert(word)
self=WordDictionary{'trie':Trie...}; word='bad'

L9 addWord: self.trie.insert(word)
self=WordDictionary{'trie':Trie...}; word='dad'

L12 search: def match(node, i):
self=WordDictionary{'trie':Trie...}; pattern='.ad'

L13 match: if i == len(pattern):
node=TrieNode{'children':{...:..., ...:...}, 'terminal':False}; i=0; pattern='.ad'

L15 match: char = pattern[i]
node=TrieNode{'children':{...:..., ...:...}, 'terminal':False}; i=0; pattern='.ad'; char='.'

L13 match: if i == len(pattern):
node=TrieNode{'children':{...:...}, 'terminal':False}; i=1; pattern='.ad'
```

```
L15 match: char = pattern[i]
node=TrieNode{'children':{...:...}, 'terminal':False}; i=1; pattern='.ad'; char='a'

... 11 intermediate state changes omitted ...

L15 match: char = pattern[i]
node=TrieNode{'children':{...:...}, 'terminal':False}; i=1; pattern='b.'; char='.'

L13 match: if i == len(pattern):
node=TrieNode{'children':{...:...}, 'terminal':False}; i=2; pattern='b.'

L17 <genexpr>: return any(match(child, i + 1) for child in node.children.values())
.0=dict_valueiterator; child=TrieNode{'children':{...:...}, 'terminal':False}; i=1

L17 <genexpr>: return any(match(child, i + 1) for child in node.children.values())
.0=dict_valueiterator; child=TrieNode{'children':{...:...}, 'terminal':False}; i=1
```

Result: [None, None, True, False]. A dot consumes exactly one character and completion must be terminal.

```
run_operations(WordDictionary,[("addWord",""),("search",""),("search",".")])

L6 __init__: self.trie = Trie()
self=WordDictionary{'trie':Trie...}

L9 addWord: self.trie.insert(word)
self=WordDictionary{'trie':Trie...}; word=''

L12 search: def match(node, i):
self=WordDictionary{'trie':Trie...}; pattern=''

L13 match: if i == len(pattern):
node=TrieNode{'children':{}, 'terminal':True}; i=0; pattern=''

L12 search: def match(node, i):
self=WordDictionary{'trie':Trie...}; pattern='.'

L13 match: if i == len(pattern):
node=TrieNode{'children':{}, 'terminal':True}; i=0; pattern='.'

L15 match: char = pattern[i]
node=TrieNode{'children':{}, 'terminal':True}; i=0; pattern='.'; char='.'

L17 <genexpr>: return any(match(child, i + 1) for child in node.children.values())
.0=dict_valueiterator; i=0
```

Result: [None, True, False]. The empty word cannot supply a wildcard character.

Why the solution is correct

A literal transition matches precisely its required character. A wildcard transition enumerates all possible one-character matches and no zero- or multi-character matches. The terminal base enforces exact length and word completion. Induction on remaining pattern length proves the search result.

Time and space

Insertion takes expected O(L). Search visits at most O(V) relevant trie nodes; a coarse branching bound is O(b^L) for branching factor b and length L. O(L) recursive depth, plus shared trie storage.

Common mistakes

Returning true merely when the pattern ends accepts prefixes that are not words. Advancing the trie without advancing i makes a dot behave like an unbounded wildcard.

Reasoning card

State: match(node,i) asks whether some stored suffix below node matches pattern[i:]. The prefix before i has already matched.

Invariant: Recursive state is (trie node, pattern index).

Decision to rehearse: Normal characters choose one edge; wildcard branches across all children.

Finish: Succeed if any branch completes

Transfer exercise

Add '*' meaning any number of characters. Identify the new branching transitions and why memoizing (node,index) becomes useful.

Module 19
Graph Traversal and Connected Components

Identify nodes/edges explicitly, select DFS or BFS, and mark discovery at the correct time.

Make vertices, edges, direction and identity explicit before traversing a graph. Discovery must happen when work is scheduled. Reachability uses permanent visitation; path enumeration may revisit a shared descendant through another route. BFS shortest-hop reasoning depends on every edge having the same cost.

The ten-step progression

181. Build an Adjacency List

182. DFS Reachability

183. BFS Shortest Hops

184. Count Graph Components

185. Clone a Graph

186. Validate an Undirected Tree

187. Bipartite Graph Check

188. All Paths from Source to Target in DAG

189. Number of Provinces

190. Word Ladder Length

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

181. Build an Adjacency List

Foundation | Module 19

Problem

Convert n vertices numbered 0 through n-1 and valid edge pairs (u,v) into adjacency lists. directed=False adds both directions; directed=True adds only u to v. Preserve input edge order and multiplicity. An undirected self-loop contributes two entries. Isolated vertices still receive empty lists.

Examples

Example 1. Reverse entries represent undirected edges, including an isolated vertex.

```
adjacency_list(4,[(0,1),(1,2)])
=> [[1], [0, 2], [1], []]
```

Example 2. Directed edges do not imply a return edge.

```
adjacency_list(3,[(0,1)],True)
=> [[1], [], []]
```

Example 3. An undirected loop contributes both endpoint incidences.

```
adjacency_list(1,[(0,0)])
=> [[0, 0]]
```

Why this mechanism works

Representation is part of the problem contract. Allocate one independent list per vertex before reading edges, then translate each edge according to directionality. Keeping multiplicity makes this a faithful representation of a multigraph; deduplication would be a separate policy.

State and boundaries

adj[u] contains all neighbor entries contributed by the processed edge prefix for vertex u.

Protocol

1. Allocate n independent lists.

2. For each edge append v to adj[u].

3. If undirected also append u to adj[v].

4. Return all n lists.

Visualization

```
edges (0,1),(1,2), n=4
undirected: 0:[1] 1:[0,2] 2:[1] 3:[]
directed:   0:[1] 1:[2]   2:[]  3:[]
```

Python implementation

```
def adjacency_list(n, edges, directed=False):
    adj = [[] for _ in range(n)]
    for u, v in edges:
        adj[u].append(v)
        if not directed:
            adj[v].append(u)
    return adj
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
adjacency_list(4,[(0,1),(1,2)])

L2 adjacency_list: adj = [[] for _ in range(n)]
n=4; edges=[(0, 1), (1, 2)]; directed=False; _=0

L2 adjacency_list: adj = [[] for _ in range(n)]
n=4; edges=[(0, 1), (1, 2)]; directed=False; _=1

L2 adjacency_list: adj = [[] for _ in range(n)]
n=4; edges=[(0, 1), (1, 2)]; directed=False; _=2

L2 adjacency_list: adj = [[] for _ in range(n)]
n=4; edges=[(0, 1), (1, 2)]; directed=False; _=3

L2 adjacency_list: adj = [[] for _ in range(n)]
n=4; edges=[(0, 1), (1, 2)]; directed=False; adj=[[], [], [], []]

L3 adjacency_list: for u, v in edges:
n=4; edges=[(0, 1), (1, 2)]; directed=False; adj=[[], [], [], []]; u=0; v=1

L4 adjacency_list: adj[u].append(v)
n=4; edges=[(0, 1), (1, 2)]; directed=False; adj=[[1], [], [], []]; u=0; v=1

L6 adjacency_list: adj[v].append(u)
n=4; edges=[(0, 1), (1, 2)]; directed=False; adj=[[1], [0], [], []]; u=0; v=1

L3 adjacency_list: for u, v in edges:
n=4; edges=[(0, 1), (1, 2)]; directed=False; adj=[[1], [0], [], []]; u=1; v=2

L4 adjacency_list: adj[u].append(v)
n=4; edges=[(0, 1), (1, 2)]; directed=False; adj=[[1], [0, 2], [], []]; u=1; v=2

L6 adjacency_list: adj[v].append(u)
n=4; edges=[(0, 1), (1, 2)]; directed=False; adj=[[1], [0, 2], [1], []]; u=1; v=2
```

Result: [[1], [0, 2], [1], []]. Reverse entries represent undirected edges, including an isolated vertex.

```
adjacency_list(3,[(0,1)],True)

L2 adjacency_list: adj = [[] for _ in range(n)]
n=3; edges=[(0, 1)]; directed=True; _=0

L2 adjacency_list: adj = [[] for _ in range(n)]
n=3; edges=[(0, 1)]; directed=True; _=1

L2 adjacency_list: adj = [[] for _ in range(n)]
n=3; edges=[(0, 1)]; directed=True; _=2

L2 adjacency_list: adj = [[] for _ in range(n)]
n=3; edges=[(0, 1)]; directed=True; adj=[[], [], []]

L3 adjacency_list: for u, v in edges:
n=3; edges=[(0, 1)]; directed=True; adj=[[], [], []]; u=0; v=1

L4 adjacency_list: adj[u].append(v)
n=3; edges=[(0, 1)]; directed=True; adj=[[1], [], []]; u=0; v=1
```

Result: [[1], [], []]. Directed edges do not imply a return edge.

Why the solution is correct

Initially no processed edge requires an entry. Each iteration adds exactly the one or two entries required for its edge and changes no others. After all edges, every required occurrence is represented, including isolated vertices and repeated edges.

Time and space

O(n+m) time and space for m input edges. Undirected storage has 2m neighbor entries.

Common mistakes

Using [[]]*n aliases every vertex's list. Adding reverse edges to a directed graph changes its reachability.

Reasoning card

State: adj[u] contains all neighbor entries contributed by the processed edge prefix for vertex u.

Invariant: Every input edge appears exactly as required.

Decision to rehearse: Add one or two directions according to the graph contract.

Finish: Return all n lists

Transfer exercise

Store weighted edges as (neighbor,weight) pairs while retaining the same directionality contract.

182. DFS Reachability

Foundation | Module 19

Problem

Given an adjacency list and valid start and target vertices, return whether a directed path exists from start to target. Undirected graphs work when both directions are present. A zero-edge path makes start==target succeed. Cycles and repeated edges are allowed.

Examples

Example 1. A directed route reaches 3 despite the cycle near the start.

```
dfs_reachable([[1],[0,2],[3],[]],0,3)
=> True
```

Example 2. The isolated target is never discovered.

```
dfs_reachable([[1],[],[]],0,2)
=> False
```

Example 3. The zero-edge path is sufficient.

```
dfs_reachable([[]],0,0)
=> True
```

Why this mechanism works

Reachability needs to explore each discovered vertex only once. Mark a vertex when scheduling it on the DFS stack, so several incoming edges cannot create duplicate pending work. A stack determines exploration order but does not imply a shortest route.

State and boundaries

seen contains exactly the discovered vertices, each reachable from start. stack contains discovered vertices awaiting processing.

Protocol

1. Mark and push start.
2. Pop a vertex and test for target.
3. Mark and push each unseen neighbor.
4. Return false if the stack empties.

Visualization

```
0 -> 1 -> 2
^    |    |
|____|    v
          3
seen prevents the 1->0 cycle from rescheduling 0
```

Python implementation

```
def dfs_reachable(adj, start, target):
    stack, seen = [start], {start}
    while stack:
        u = stack.pop()
        if u == target:
            return True
        for v in adj[u]:
            if v not in seen:
                seen.add(v)
                stack.append(v)
    return False
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
dfs_reachable([[1],[0,2],[3],[]],0,3)

L2 dfs_reachable: stack, seen = [start], {start}
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[0]; seen={0}

L4 dfs_reachable: u = stack.pop()
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0}; u=0

L7 dfs_reachable: for v in adj[u]:
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0}; u=0; v=1

L9 dfs_reachable: seen.add(v)
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0, 1}; u=0; v=1

L10 dfs_reachable: stack.append(v)
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[1]; seen={0, 1}; u=0; v=1

L4 dfs_reachable: u = stack.pop()
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0, 1}; u=1; v=1

L7 dfs_reachable: for v in adj[u]:
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0, 1}; u=1; v=0

L7 dfs_reachable: for v in adj[u]:
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0, 1}; u=1; v=2

... 3 intermediate state changes omitted ...

L7 dfs_reachable: for v in adj[u]:
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0, 1, 2}; u=2; v=3

L9 dfs_reachable: seen.add(v)
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0, 1, 2, 3}; u=2; v=3
```

```
L10 dfs_reachable: stack.append(v)
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[3]; seen={0, 1, 2, 3}; u=2; v=3

L4 dfs_reachable: u = stack.pop()
adj=[[1], [0, 2], [3], []]; start=0; target=3; stack=[]; seen={0, 1, 2, 3}; u=3; v=3
```

Result: True. A directed route reaches 3 despite the cycle near the start.

```
dfs_reachable([[1],[],[]],0,2)

L2 dfs_reachable: stack, seen = [start], {start}
adj=[[1], [], []]; start=0; target=2; stack=[0]; seen={0}

L4 dfs_reachable: u = stack.pop()
adj=[[1], [], []]; start=0; target=2; stack=[]; seen={0}; u=0

L7 dfs_reachable: for v in adj[u]:
adj=[[1], [], []]; start=0; target=2; stack=[]; seen={0}; u=0; v=1

L9 dfs_reachable: seen.add(v)
adj=[[1], [], []]; start=0; target=2; stack=[]; seen={0, 1}; u=0; v=1

L10 dfs_reachable: stack.append(v)
adj=[[1], [], []]; start=0; target=2; stack=[1]; seen={0, 1}; u=0; v=1

L4 dfs_reachable: u = stack.pop()
adj=[[1], [], []]; start=0; target=2; stack=[]; seen={0, 1}; u=1; v=1
```

Result: False. The isolated target is never discovered.

Why the solution is correct

The start is reachable, and every newly scheduled neighbor extends a known path. Thus a returned target is reachable. If the search exhausts, every edge from every discovered vertex has been explored, so no reachable vertex remains undiscovered. Each vertex is scheduled at most once.

Time and space

O(V+E) worst-case time and O(V) auxiliary space over the reachable portion of the graph.

Common mistakes

Marking only after pop permits duplicate scheduling. A visited value must identify the vertex, not a possibly repeated display label.

Reasoning card

State: seen contains exactly the discovered vertices, each reachable from start. stack contains discovered vertices awaiting processing.

Invariant: Every scheduled node is reachable and processed at most once.

Decision to rehearse: Use a stack/recursion and visited set.

Finish: Return false if the stack empties

Transfer exercise

Store a parent when discovering each vertex and reconstruct one path when the target is reached.

183. BFS Shortest Hops

Foundation | Module 19

Problem

Return the minimum number of edges from start to target in an unweighted adjacency-list graph. Vertices are valid indices; edges may be directed. Return -1 when unreachable and zero when start equals target. Cycles and duplicate edges are allowed.

Examples

Example 1. The two-edge branch reaches the target first.

```
shortest_hops([[1,2],[3],[4],[],[3]],0,3)
=> 2
```

Example 2. No path reaches the target.

```
shortest_hops([[1],[],[]],0,2)
=> -1
```

Example 3. No edge is needed.

```
shortest_hops([[]],0,0)
=> 0
```

Why this mechanism works

Each edge costs one, so processing all distance-d vertices before distance-(d+1) vertices makes first discovery optimal. Use FIFO order and assign a distance when enqueuing. A vertex needs no later improvement because every competing route arrives from an equal or greater depth.

State and boundaries

dist[v] is -1 until discovery, then its shortest hop count. The queue is ordered by nondecreasing assigned distance.

Protocol

1. Set start distance to zero and enqueue it.
2. Pop the next vertex.
3. Return if it is target.
4. For each undiscovered neighbor assign current distance+1 and enqueue.
5. Return -1 on exhaustion.

Visualization

```
0 -> 1 -> 3
 \-> 2 -> 4 -> 3
BFS layers: [0], [1,2], [3,4]
distance to 3 is 2
```

Python implementation

```
from collections import deque

def shortest_hops(adj, start, target):
    dist = [-1] * len(adj)
    dist[start] = 0
    queue = deque([start])
    while queue:
        u = queue.popleft()
        if u == target:
            return dist[u]
        for v in adj[u]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                queue.append(v)
    return -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
shortest_hops([[1,2],[3],[4],[],[3]],0,3)

L4 shortest_hops: dist = [-1] * len(adj)
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[-1, -1, -1, -1, -1]

L5 shortest_hops: dist[start] = 0
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, -1, -1, -1, -1]

L6 shortest_hops: queue = deque([start])
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, -1, -1, -1, -1]; queue=deque([0])

L8 shortest_hops: u = queue.popleft()
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, -1, -1, -1, -1]; queue=deque([]);
     u=0

L11 shortest_hops: for v in adj[u]:
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, -1, -1, -1, -1]; queue=deque([]);
     u=0; v=1

L13 shortest_hops: dist[v] = dist[u] + 1
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, 1, -1, -1, -1]; queue=deque([]);
    u=0; v=1

L14 shortest_hops: queue.append(v)
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, 1, -1, -1, -1]; queue=deque([1]);
     u=0; v=1

L11 shortest_hops: for v in adj[u]:
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, 1, -1, -1, -1]; queue=deque([1]);
     u=0; v=2
```

```
... 7 intermediate state changes omitted ...

L11 shortest_hops: for v in adj[u]:
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, 1, 1, 2, -1]; queue=deque([3]);
    u=2; v=4

L13 shortest_hops: dist[v] = dist[u] + 1
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, 1, 1, 2, 2]; queue=deque([3]);
    u=2; v=4

L14 shortest_hops: queue.append(v)
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, 1, 1, 2, 2]; queue=deque([3, 4]);
     u=2; v=4

L8 shortest_hops: u = queue.popleft()
adj=[[1, 2], [3], [4], [], [3]]; start=0; target=3; dist=[0, 1, 1, 2, 2]; queue=deque([4]);
    u=3; v=4
```

Result: 2. The two-edge branch reaches the target first.

```
shortest_hops([[1],[],[]],0,2)

L4 shortest_hops: dist = [-1] * len(adj)
adj=[[1], [], []]; start=0; target=2; dist=[-1, -1, -1]

L5 shortest_hops: dist[start] = 0
adj=[[1], [], []]; start=0; target=2; dist=[0, -1, -1]

L6 shortest_hops: queue = deque([start])
adj=[[1], [], []]; start=0; target=2; dist=[0, -1, -1]; queue=deque([0])

L8 shortest_hops: u = queue.popleft()
adj=[[1], [], []]; start=0; target=2; dist=[0, -1, -1]; queue=deque([]); u=0

L11 shortest_hops: for v in adj[u]:
adj=[[1], [], []]; start=0; target=2; dist=[0, -1, -1]; queue=deque([]); u=0; v=1

L13 shortest_hops: dist[v] = dist[u] + 1
adj=[[1], [], []]; start=0; target=2; dist=[0, 1, -1]; queue=deque([]); u=0; v=1

L14 shortest_hops: queue.append(v)
adj=[[1], [], []]; start=0; target=2; dist=[0, 1, -1]; queue=deque([1]); u=0; v=1

L8 shortest_hops: u = queue.popleft()
adj=[[1], [], []]; start=0; target=2; dist=[0, 1, -1]; queue=deque([]); u=1; v=1
```

Result: -1. No path reaches the target.

Why the solution is correct

Inductively, all vertices at distance at most d are discovered before processing the next layer. A newly discovered neighbor has a length-(d+1) route, and a shorter route would already have discovered it. Therefore every assigned distance is minimal, including the returned target distance.

Time and space

O(V+E) time and O(V) auxiliary distance/queue storage.

Common mistakes

A DFS visitation distance is not generally shortest. This first-discovery proof fails when edges have unequal weights.

Reasoning card

State: dist[v] is -1 until discovery, then its shortest hop count. The queue is ordered by nondecreasing assigned distance.

Invariant: Queue order is nondecreasing distance.

Decision to rehearse: Queue stores nodes with their distance; mark on enqueue.

Finish: Return -1 on exhaustion

Transfer exercise

Count shortest paths by accumulating counts when another edge reaches a vertex at its already-known optimal distance.

184. Count Graph Components

Foundation | Module 19

Problem

Count connected components in an undirected adjacency-list graph whose vertices are 0 through n-1. Both directions of every edge are represented. Isolated vertices count as components, and an empty graph returns zero. Do not change the graph.

Examples

Example 1. Two connected pairs and one isolated vertex form three components.

```
graph_components([[1],[0],[3],[2],[]])
=> 3
```

Example 2. The empty graph has no components.

```
graph_components([])
=> 0
```

Example 3. Every isolated vertex is its own component.

```
graph_components([[],[],[]])
=> 3
```

Why this mechanism works

One traversal consumes one entire component. Scan all vertices outside that traversal; every still-unseen vertex begins a new component. Keep the visited set across traversals so a component is counted only at its first encountered member.

State and boundaries

seen contains the union of completed components and the currently explored one. count increases only when an unseen start is chosen.

Protocol

1. Scan every vertex.
2. Skip discovered vertices.
3. Increment count for a new start.
4. Mark it and consume its entire component with DFS.
5. Continue the outer scan.

Visualization

```
0--1    2--3    4
outer starts: 0, then 2, then 4
component count: 1 -> 2 -> 3
```

Python implementation

```
def graph_components(adj):
    seen, count = set(), 0
    for start in range(len(adj)):
        if start in seen:
            continue
        count += 1
        seen.add(start)
        stack = [start]
        while stack:
            u = stack.pop()
            for v in adj[u]:
                if v not in seen:
                    seen.add(v)
                    stack.append(v)
    return count
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
graph_components([[1],[0],[3],[2],[]])

L2 graph_components: seen, count = set(), 0
adj=[[1], [0], [3], [2], []]; seen=set(); count=0

L3 graph_components: for start in range(len(adj)):
adj=[[1], [0], [3], [2], []]; seen=set(); count=0; start=0

L6 graph_components: count += 1
adj=[[1], [0], [3], [2], []]; seen=set(); count=1; start=0

L7 graph_components: seen.add(start)
adj=[[1], [0], [3], [2], []]; seen={0}; count=1; start=0

L8 graph_components: stack = [start]
adj=[[1], [0], [3], [2], []]; seen={0}; count=1; start=0; stack=[0]

L10 graph_components: u = stack.pop()
adj=[[1], [0], [3], [2], []]; seen={0}; count=1; start=0; stack=[]; u=0

L11 graph_components: for v in adj[u]:
adj=[[1], [0], [3], [2], []]; seen={0}; count=1; start=0; stack=[]; u=0; v=1

L13 graph_components: seen.add(v)
adj=[[1], [0], [3], [2], []]; seen={0, 1}; count=1; start=0; stack=[]; u=0; v=1

... 16 intermediate state changes omitted ...

L6 graph_components: count += 1
adj=[[1], [0], [3], [2], []]; seen={0, 1, 2, 3}; count=3; start=4; stack=[]; u=3; v=2

L7 graph_components: seen.add(start)
```

```
adj=[[1], [0], [3], [2], []]; seen={0, 1, 2, 3, 4}; count=3; start=4; stack=[]; u=3; v=2

L8 graph_components: stack = [start]
adj=[[1], [0], [3], [2], []]; seen={0, 1, 2, 3, 4}; count=3; start=4; stack=[4]; u=3; v=2

L10 graph_components: u = stack.pop()
adj=[[1], [0], [3], [2], []]; seen={0, 1, 2, 3, 4}; count=3; start=4; stack=[]; u=4; v=2
```

Result: 3. Two connected pairs and one isolated vertex form three components.

```
graph_components([])

L2 graph_components: seen, count = set(), 0
adj=[]; seen=set(); count=0
```

Result: 0. The empty graph has no components.

Why the solution is correct

Undirected reachability from a start is exactly its component. The inner traversal discovers all and only that component. Later scans skip its vertices, while every other component eventually supplies an unseen start, so each component is counted once.

Time and space

O(V+E) time and O(V) auxiliary space.

Common mistakes

Resetting seen for each start counts a component repeatedly. Applying this directly to directed graphs does not compute strongly connected components.

Reasoning card

State: seen contains the union of completed components and the currently explored one. count increases only when an unseen start is chosen.

Invariant: Every visited node belongs to exactly one counted component.

Decision to rehearse: Outer node scan discovers components; inner traversal consumes one.

Finish: Continue the outer scan

Transfer exercise

Return each component's member list as well as the count.

185. Clone a Graph

Bridge | Module 19

Problem

Deep-copy every node reachable from a GraphNode reference, preserving values, neighbor order, repeated edges, cycles and self-loops. Return the corresponding cloned root, or None for None. Nodes with equal values remain distinct objects; no clone may reference an original node.

Examples

Example 1. A self-loop points to the new object.

```
a=GraphNode(7); a.neighbors=[a]; copy=clone_graph(a)
(copy is not a, copy.val, copy.neighbors[0] is copy)
=> (True, 7, True)
```

Example 2. No root means no reachable graph.

```
clone_graph(None)
=> None
```

Example 3. Equal values do not collapse two nodes in a cycle.

```
a=GraphNode(1); b=GraphNode(1); a.neighbors=[b]; b.neighbors=[a]; copy=clone_graph(a)
(copy.neighbors[0] is not copy, copy.neighbors[0].neighbors[0] is copy)
=> (True, True)
```

Why this mechanism works

Allocate a node's copy as soon as that original is discovered, before following its edges. An identity-keyed mapping then provides a stable destination for back edges and shared neighbors. Every outgoing edge can append the already-known clone or trigger creation of a new one.

State and boundaries

copies maps each discovered original object to exactly one new object. The queue holds originals whose neighbor lists still need copying.

Protocol

1. Allocate and map the root copy.
2. Pop an original.
3. For each neighbor create its copy if absent and enqueue it.
4. Append that mapped copy to the current clone's neighbors.

5. Return the mapped root.

Visualization

```
original A -> B -> A
create A' first; map A:A'
create B'; edge A'->B'
back edge B'->copies[A] closes the clone cycle
```

Python implementation

```
from collections import deque
from structures import GraphNode

def clone_graph(root):
    if root is None:
        return None
    copies = {root: GraphNode(root.val)}
    queue = deque([root])
    while queue:
        original = queue.popleft()
        for neighbor in original.neighbors:
            if neighbor not in copies:
                copies[neighbor] = GraphNode(neighbor.val)
                queue.append(neighbor)
            copies[original].neighbors.append(copies[neighbor])
    return copies[root]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
(copy is not a, copy.val, copy.neighbors[0] is copy)

L5 clone_graph: if root is None:
root=N1(val=7,neighbors=[N1])

L7 clone_graph: copies = {root: GraphNode(root.val)}
root=N1(val=7,neighbors=[N1]); copies={N1(val=7,neighbors=[N1]):N2(val=7,neighbors=[])}

L8 clone_graph: queue = deque([root])
root=N1(val=7,neighbors=[N1]); copies={N1(val=7,neighbors=[N1]):N2(val=7,neighbors=[])};
    queue=deque([N1(val=7,neighbors=[N1])])

L10 clone_graph: original = queue.popleft()
root=N1(val=7,neighbors=[N1]); copies={N1(val=7,neighbors=[N1]):N2(val=7,neighbors=[])};
    queue=deque([]); original=N1(val=7,neighbors=[N1])

L11 clone_graph: for neighbor in original.neighbors:
root=N1(val=7,neighbors=[N1]); copies={N1(val=7,neighbors=[N1]):N2(val=7,neighbors=[])};
    queue=deque([]); original=N1(val=7,neighbors=[N1]); neighbor=N1(val=7,neighbors=[N1])

L15 clone_graph: copies[original].neighbors.append(copies[neighbor])
root=N1(val=7,neighbors=[N1]); copies={N1(val=7,neighbors=[N1]):N2(val=7,neighbors=[N2])};
    queue=deque([]); original=N1(val=7,neighbors=[N1]); neighbor=N1(val=7,neighbors=[N1])
```

Result: (True, 7, True). A self-loop points to the new object.

```
clone_graph(None)

L5 clone_graph: if root is None:
root=None
```

Result: None. No root means no reachable graph.

Why the solution is correct

The map guarantees a one-to-one copy for every discovered original. Each original adjacency entry becomes an entry to the corresponding clone in the same order. All reachable originals are eventually processed, so all edges and values are copied without retaining original references.

Time and space

O(V+E) time, O(V) auxiliary map/queue space, and O(V+E) newly allocated graph output.

Common mistakes

Mapping by value merges distinct equal-valued nodes. Allocating only after recursively exploring neighbors fails on cycles.

Reasoning card

State: copies maps each discovered original object to exactly one new object. The queue holds originals whose neighbor lists still need copying.

Invariant: Every original discovered so far has one stable copy.

Decision to rehearse: Map original nodes to copies before exploring neighbours.

Finish: Return the mapped root

Transfer exercise

Serialize the reachable graph with explicit node IDs, then explain why values alone cannot serve as those IDs.

186. Validate an Undirected Tree

Bridge | Module 19

Problem

Return whether n vertices and an undirected edge list form a tree. Vertices are 0 through n-1. A tree is nonempty, connected and acyclic; n=0 returns false, while one vertex with no edges returns true. Self-loops and repeated edges are allowed as input but cannot form a tree.

Examples

Example 1. Three edges connect four vertices without a cycle.

```
valid_tree(4,[(0,1),(1,2),(1,3)])
=> True
```

Example 2. The edge count matches, but vertex 3 is disconnected.

```
valid_tree(4,[(0,1),(1,2),(2,0)])
=> False
```

Example 3. A singleton is a tree.

```
valid_tree(1,[])
=> True
```

Example 4. The contract requires a nonempty tree.

```
valid_tree(0,[])
=> False
```

Why this mechanism works

A finite undirected tree has exactly n-1 edges. With that count established, connectivity is enough: a connected graph with a cycle would contain an extra edge beyond a spanning tree. Check the cheap edge count before building the adjacency list and traversing from vertex zero.

State and boundaries

seen contains vertices connected to zero through processed paths. The edge-count precheck fixes the only count compatible with a tree.

Protocol

1. Reject n=0 or an edge count other than n-1.

2. Build undirected adjacency.

3. Traverse from zero, marking on push.

4. Return whether all n vertices were reached.

Visualization

```
triangle 0-1-2-0 plus isolated 3
n=4, edges=3 passes n-1
only 3 reachable vertices -> reject
```

Python implementation

```
from kata_181 import adjacency_list

def valid_tree(n, edges):
    if n == 0 or len(edges) != n - 1:
        return False
    adj = adjacency_list(n, edges)
    seen, stack = {0}, [0]
    while stack:
        u = stack.pop()
        for v in adj[u]:
            if v not in seen:
                seen.add(v)
                stack.append(v)
    return len(seen) == n
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
valid_tree(4,[(0,1),(1,2),(1,3)])

L4 valid_tree: if n == 0 or len(edges) != n - 1:
n=4; edges=[(0, 1), (1, 2), (1, 3)]

L6 valid_tree: adj = adjacency_list(n, edges)
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]

L7 valid_tree: seen, stack = {0}, [0]
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0}; stack=[0]

L9 valid_tree: u = stack.pop()
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0}; stack=[]; u=0

L10 valid_tree: for v in adj[u]:
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0}; stack=[]; u=0;
    v=1

L12 valid_tree: seen.add(v)
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0, 1}; stack=[];
    u=0; v=1

L13 valid_tree: stack.append(v)
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0, 1}; stack=[1];
    u=0; v=1

L9 valid_tree: u = stack.pop()
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0, 1}; stack=[];
    u=1; v=1
```

```
... 6 intermediate state changes omitted ...

L13 valid_tree: stack.append(v)
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0, 1, 2, 3};
    stack=[2, 3]; u=1; v=3

L9 valid_tree: u = stack.pop()
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0, 1, 2, 3};
    stack=[2]; u=3; v=3

L10 valid_tree: for v in adj[u]:
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0, 1, 2, 3};
    stack=[2]; u=3; v=1

L9 valid_tree: u = stack.pop()
n=4; edges=[(0, 1), (1, 2), (1, 3)]; adj=[[1], [0, 2, 3], [1], [1]]; seen={0, 1, 2, 3};
    stack=[]; u=2; v=1
```

Result: True. Three edges connect four vertices without a cycle.

```
valid_tree(4,[(0,1),(1,2),(2,0)])

L4 valid_tree: if n == 0 or len(edges) != n - 1:
n=4; edges=[(0, 1), (1, 2), (2, 0)]

L6 valid_tree: adj = adjacency_list(n, edges)
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]

L7 valid_tree: seen, stack = {0}, [0]
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0}; stack=[0]

L9 valid_tree: u = stack.pop()
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0}; stack=[]; u=0

L10 valid_tree: for v in adj[u]:
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0}; stack=[];
    u=0; v=1

L12 valid_tree: seen.add(v)
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0, 1}; stack=[];
    u=0; v=1

L13 valid_tree: stack.append(v)
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0, 1}; stack=[1];
     u=0; v=1

L10 valid_tree: for v in adj[u]:
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0, 1}; stack=[1];
     u=0; v=2

... 3 intermediate state changes omitted ...

L10 valid_tree: for v in adj[u]:
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0, 1, 2};
    stack=[1]; u=2; v=1

L10 valid_tree: for v in adj[u]:
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0, 1, 2};
    stack=[1]; u=2; v=0

L9 valid_tree: u = stack.pop()
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0, 1, 2};
    stack=[]; u=1; v=0
```

```
L10 valid_tree: for v in adj[u]:
n=4; edges=[(0, 1), (1, 2), (2, 0)]; adj=[[1, 2], [0, 2], [1, 0], []]; seen={0, 1, 2};
    stack=[]; u=1; v=2
```

Result: False. The edge count matches, but vertex 3 is disconnected.

Why the solution is correct

Every tree passes the count and connectivity tests. Conversely, a connected graph on n vertices contains a spanning tree with n-1 edges. If the whole graph has only those many edges, none remain for a cycle, loop or parallel-edge cycle. Thus acceptance implies a tree.

Time and space

O(n+m) time and space after the count check; incompatible counts are rejected in O(1) when edges is a list.

Common mistakes

The n-1 count alone accepts a cyclic component plus an isolated vertex. Connectivity alone accepts graphs with extra cycle edges.

Reasoning card

State: seen contains vertices connected to zero through processed paths. The edge-count precheck fixes the only count compatible with a tree.

Invariant: A tree has n-1 edges and one component.

Decision to rehearse: Check edge count and connectivity or detect parent-aware cycles.

Finish: Return whether all n vertices were reached

Transfer exercise

Detect cycles during DFS with edge IDs rather than only a parent vertex, so parallel edges are handled correctly.

187. Bipartite Graph Check

Bridge | Module 19

Problem

Return whether an undirected graph can be colored with two colors such that every edge joins opposite colors. Process every component, including isolated vertices. A self-loop makes the answer false; the empty graph is bipartite.

Examples

Example 1. The even cycle supports alternating colors.

```
is_bipartite([[1,3],[0,2],[1,3],[0,2]])
=> True
```

Example 2. The odd cycle forces equal colors across one edge.

```
is_bipartite([[1,2],[0,2],[0,1]])
=> False
```

Example 3. There is no contradictory edge.

```
is_bipartite([])
=> True
```

Why this mechanism works

Choose either color for an unseen component root. Every edge then forces its neighbor to the opposite color. A conflict occurs when an edge joins two vertices already assigned the same color. Disconnected components can choose their starting colors independently.

State and boundaries

color[v] is -1 if undiscovered, otherwise 0 or 1. Every processed edge has been checked for opposite endpoint colors.

Protocol

1. Scan all component starts.
2. Assign color zero to an unseen start.
3. Traverse its edges.
4. Color each unseen neighbor oppositely and schedule it.
5. Reject any same-color edge.
6. Accept after all components.

Visualization

```
square: 0(A)-1(B)-2(A)-3(B)-0(A) -> valid
triangle: 0(A)-1(B)-2(A)-0(A) -> conflict
```

Python implementation

```
def is_bipartite(adj):
    color = [-1] * len(adj)
    for start in range(len(adj)):
        if color[start] != -1:
            continue
        color[start] = 0
        stack = [start]
        while stack:
            u = stack.pop()
            for v in adj[u]:
                if color[v] == -1:
                    color[v] = 1 - color[u]
                    stack.append(v)
                elif color[v] == color[u]:
                    return False
    return True
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
is_bipartite([[1,3],[0,2],[1,3],[0,2]])

L2 is_bipartite: color = [-1] * len(adj)
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[-1, -1, -1, -1]

L3 is_bipartite: for start in range(len(adj)):
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[-1, -1, -1, -1]; start=0

L6 is_bipartite: color[start] = 0
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, -1, -1, -1]; start=0

L7 is_bipartite: stack = [start]
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, -1, -1, -1]; start=0; stack=[0]

L9 is_bipartite: u = stack.pop()
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, -1, -1, -1]; start=0; stack=[]; u=0

L10 is_bipartite: for v in adj[u]:
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, -1, -1, -1]; start=0; stack=[]; u=0; v=1

L12 is_bipartite: color[v] = 1 - color[u]
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, 1, -1, -1]; start=0; stack=[]; u=0; v=1

L13 is_bipartite: stack.append(v)
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, 1, -1, -1]; start=0; stack=[1]; u=0; v=1

... 13 intermediate state changes omitted ...

L10 is_bipartite: for v in adj[u]:
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, 1, 0, 1]; start=0; stack=[]; u=1; v=2

L3 is_bipartite: for start in range(len(adj)):
```

```
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, 1, 0, 1]; start=1; stack=[]; u=1; v=2

L3 is_bipartite: for start in range(len(adj)):
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, 1, 0, 1]; start=2; stack=[]; u=1; v=2

L3 is_bipartite: for start in range(len(adj)):
adj=[[1, 3], [0, 2], [1, 3], [0, 2]]; color=[0, 1, 0, 1]; start=3; stack=[]; u=1; v=2
```

Result: True. The even cycle supports alternating colors.

```
is_bipartite([[1,2],[0,2],[0,1]])

L2 is_bipartite: color = [-1] * len(adj)
adj=[[1, 2], [0, 2], [0, 1]]; color=[-1, -1, -1]

L3 is_bipartite: for start in range(len(adj)):
adj=[[1, 2], [0, 2], [0, 1]]; color=[-1, -1, -1]; start=0

L6 is_bipartite: color[start] = 0
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, -1, -1]; start=0

L7 is_bipartite: stack = [start]
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, -1, -1]; start=0; stack=[0]

L9 is_bipartite: u = stack.pop()
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, -1, -1]; start=0; stack=[]; u=0

L10 is_bipartite: for v in adj[u]:
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, -1, -1]; start=0; stack=[]; u=0; v=1

L12 is_bipartite: color[v] = 1 - color[u]
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, 1, -1]; start=0; stack=[]; u=0; v=1

L13 is_bipartite: stack.append(v)
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, 1, -1]; start=0; stack=[1]; u=0; v=1

... 2 intermediate state changes omitted ...

L13 is_bipartite: stack.append(v)
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, 1, 1]; start=0; stack=[1, 2]; u=0; v=2

L9 is_bipartite: u = stack.pop()
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, 1, 1]; start=0; stack=[1]; u=2; v=2

L10 is_bipartite: for v in adj[u]:
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, 1, 1]; start=0; stack=[1]; u=2; v=0

L10 is_bipartite: for v in adj[u]:
adj=[[1, 2], [0, 2], [0, 1]]; color=[0, 1, 1]; start=0; stack=[1]; u=2; v=1
```

Result: False. The odd cycle forces equal colors across one edge.

Why the solution is correct

The algorithm assigns colors consistent with every traversed edge unless it detects a conflict. If none exists, the assignments themselves witness bipartiteness. A conflict corresponds to incompatible parity requirements along discovered paths, so no two-coloring can satisfy that component.

Time and space

O(V+E) time and O(V) auxiliary color/frontier space.

Common mistakes

Starting from only vertex zero misses an odd cycle elsewhere. Marking a vertex without storing its required color loses the property being checked.

Reasoning card

State: color[v] is -1 if undiscovered, otherwise 0 or 1. Every processed edge has been checked for opposite endpoint colors.

Invariant: Every processed edge connects opposite colours.

Decision to rehearse: Assign opposite colour when discovering neighbours.

Finish: Accept after all components

Transfer exercise

Return an odd-cycle witness when a conflict is found by retaining traversal parents and depths.

188. All Paths from Source to Target in DAG

Interview | Module 19

Problem

Enumerate all directed paths from vertex 0 to vertex n-1 in a DAG adjacency list. Neighbor lists contain no duplicate edges; preserve their order during depth-first enumeration. Return [] for n=0 and [[0]] for a singleton. The input is guaranteed acyclic.

Examples

Example 1. The shared target is visited once per candidate route.

```
all_dag_paths([[1,2],[3],[3],[]])
=> [[0, 1, 3], [0, 2, 3]]
```

Example 2. The source is already the target.

```
all_dag_paths([[]])
=> [[0]]
```

Example 3. Neither explored branch reaches the final vertex.

```
all_dag_paths([[1],[],[]])
=> []
```

Why this mechanism works

This is path enumeration, so two routes may legitimately revisit the same graph vertex in different branches. A permanent visited set would remove valid answers after a merge point. The DAG guarantee prevents cycles within a route; a mutable path needs only append, recurse and pop.

State and boundaries

path is exactly the current valid route from zero to u. output contains independent copies of routes ending at the target.

Protocol

1. Handle the empty graph.

2. Start path with zero.

3. At the target save a copy and return.

4. For each outgoing neighbor append it, recurse and undo.

Visualization

```
0 -> 1 -> 3
 \-> 2 -> 3
3 belongs to two valid routes
output [0,1,3] and [0,2,3]
```

Python implementation

```
def all_dag_paths(adj):
    if not adj:
        return []
    output, path = [], [0]
    def visit(u):
        if u == len(adj) - 1:
            output.append(path.copy())
            return
        for v in adj[u]:
            path.append(v)
            visit(v)
            path.pop()
    visit(0)
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
all_dag_paths([[1,2],[3],[3],[]])

L2 all_dag_paths: if not adj:
adj=[[1, 2], [3], [3], []]

L4 all_dag_paths: output, path = [], [0]
adj=[[1, 2], [3], [3], []]; output=[]; path=[0]

L6 visit: if u == len(adj) - 1:
u=0; adj=[[1, 2], [3], [3], []]; output=[]; path=[0]

L9 visit: for v in adj[u]:
u=0; adj=[[1, 2], [3], [3], []]; output=[]; path=[0]; v=1

L10 visit: path.append(v)
u=0; adj=[[1, 2], [3], [3], []]; output=[]; path=[0, 1]; v=1

L6 visit: if u == len(adj) - 1:
u=1; adj=[[1, 2], [3], [3], []]; output=[]; path=[0, 1]

L9 visit: for v in adj[u]:
u=1; adj=[[1, 2], [3], [3], []]; output=[]; path=[0, 1]; v=3

L10 visit: path.append(v)
u=1; adj=[[1, 2], [3], [3], []]; output=[]; path=[0, 1, 3]; v=3

... 14 intermediate state changes omitted ...

L12 visit: path.pop()
u=2; adj=[[1, 2], [3], [3], []]; output=[[0, 1, 3], [0, 2, 3]]; path=[0, 2]; v=3

L11 visit: visit(v)
```

```
u=0; adj=[[1, 2], [3], [3], []]; output=[[0, 1, 3], [0, 2, 3]]; path=[0, 2]; v=2

L12 visit: path.pop()
u=0; adj=[[1, 2], [3], [3], []]; output=[[0, 1, 3], [0, 2, 3]]; path=[0]; v=2

L13 all_dag_paths: visit(0)
adj=[[1, 2], [3], [3], []]; output=[[0, 1, 3], [0, 2, 3]]; path=[0]
```

Result: [[0, 1, 3], [0, 2, 3]]. The shared target is visited once per candidate route.

```
all_dag_paths([[]])

L2 all_dag_paths: if not adj:
adj=[[]]

L4 all_dag_paths: output, path = [], [0]
adj=[[]]; output=[]; path=[0]

L6 visit: if u == len(adj) - 1:
u=0; adj=[[]]; output=[]; path=[0]

L7 visit: output.append(path.copy())
u=0; adj=[[]]; output=[[0]]; path=[0]

L13 all_dag_paths: visit(0)
adj=[[]]; output=[[0]]; path=[0]
```

Result: [[0]]. The source is already the target.

Why the solution is correct

Every emitted sequence follows directed edges and ends at the target. Every source-to-target path determines a unique sequence of neighbor choices that is explored. No repeated edge entries duplicate that route, and acyclicity bounds every path length by n.

Time and space

Output-sensitive exponential worst case. A coarse bound is O(n*2^n) time/output for a simple DAG; O(n) auxiliary path and recursive depth.

Common mistakes

A global visited set incorrectly discards paths sharing a suffix. Removing the DAG precondition requires path-local cycle prevention and a clear definition of allowed paths.

Reasoning card

State: path is exactly the current valid route from zero to u. output contains independent copies of routes ending at the target.

Invariant: Path is a valid directed route from source to current node.

Decision to rehearse: DFS/backtrack over graph neighbours; no global visited needed in a DAG path enumeration.

Finish: For each outgoing neighbor append it, recurse and undo

Transfer exercise

Count paths with reverse-topological dynamic programming when materializing every route is too expensive.

189. Number of Provinces

Interview | Module 19

Problem

Given a square symmetric zero/one adjacency matrix of cities, return the number of connected provinces. A one at matrix[u][v] denotes a connection. Diagonal entries do not affect component membership. Isolated cities count, and an empty matrix returns zero.

Examples

Example 1. The first two cities form one component.

```
provinces([[1,1,0],[1,1,0],[0,0,1]])
=> 2
```

Example 2. There are no cities.

```
provinces([])
=> 0
```

Example 3. Missing diagonal self-links do not remove isolated cities.

```
provinces([[0,0],[0,0]])
=> 2
```

Why this mechanism works

The component protocol is unchanged from adjacency lists, but finding a city's neighbors now requires scanning its complete matrix row. This representation makes the traversal quadratic even for a graph with few edges. Mark each city before placing it on the stack.

State and boundaries

seen identifies cities assigned to a province. Each new outer-loop start creates exactly one province traversal.

Protocol

1. Scan city indices.
2. For an unseen city increment the province count and schedule it.
3. Pop cities and inspect every column in their row.
4. Schedule unseen connected cities.
5. Return the count.

Visualization

```
1 1 0
1 1 0    rows 0 and 1 belong together
0 0 1    city 2 begins a second province
```

Python implementation

```
def provinces(matrix):
    n = len(matrix)
    seen, count = set(), 0
    for start in range(n):
        if start in seen:
            continue
        count += 1
        seen.add(start)
        stack = [start]
        while stack:
            u = stack.pop()
            for v, connected in enumerate(matrix[u]):
                if connected and v not in seen:
                    seen.add(v)
                    stack.append(v)
    return count
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
provinces([[1,1,0],[1,1,0],[0,0,1]])

L2 provinces: n = len(matrix)
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3

L3 provinces: seen, count = set(), 0
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen=set(); count=0

L4 provinces: for start in range(n):
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen=set(); count=0; start=0

L7 provinces: count += 1
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen=set(); count=1; start=0

L8 provinces: seen.add(start)
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen={0}; count=1; start=0

L9 provinces: stack = [start]
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen={0}; count=1; start=0; stack=[0]

L11 provinces: u = stack.pop()
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen={0}; count=1; start=0; stack=[]; u=0

L12 provinces: for v, connected in enumerate(matrix[u]):
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen={0}; count=1; start=0; stack=[]; u=0; v=0;
     connected=1

... 13 intermediate state changes omitted ...

L11 provinces: u = stack.pop()
```

```
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen={0, 1, 2}; count=2; start=2; stack=[];
    u=2; v=2; connected=0

L12 provinces: for v, connected in enumerate(matrix[u]):
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen={0, 1, 2}; count=2; start=2; stack=[];
    u=2; v=0; connected=0

L12 provinces: for v, connected in enumerate(matrix[u]):
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen={0, 1, 2}; count=2; start=2; stack=[];
    u=2; v=1; connected=0

L12 provinces: for v, connected in enumerate(matrix[u]):
matrix=[[1, 1, 0], [1, 1, 0], [0, 0, 1]]; n=3; seen={0, 1, 2}; count=2; start=2; stack=[];
    u=2; v=2; connected=1
```

Result: 2. The first two cities form one component.

```
provinces([])

L2 provinces: n = len(matrix)
matrix=[]; n=0

L3 provinces: seen, count = set(), 0
matrix=[]; n=0; seen=set(); count=0
```

Result: 0. There are no cities.

Why the solution is correct

Each inner traversal reaches exactly the cities connected to its start. Persistent visitation prevents a province from being counted again, and the outer scan eventually begins a traversal for every province. Symmetry ensures the intended undirected component interpretation.

Time and space

O(n^2) time because each visited city's full row is scanned, and O(n) auxiliary space.

Common mistakes

Counting diagonal ones counts cities, not provinces. Reporting O(V+E) without accounting for full matrix-row scans understates this implementation's cost.

Reasoning card

State: seen identifies cities assigned to a province. Each new outer-loop start creates exactly one province traversal.

Invariant: Visited cities are assigned to a counted province.

Decision to rehearse: Translate matrix relationships into DFS/BFS neighbours.

Finish: Return the count

Transfer exercise

For a large sparse graph, convert the representation to adjacency lists and compare storage and traversal costs.

190. Word Ladder Length

Interview | Module 19

Problem

Return the length in words of the shortest transformation from begin to end, changing exactly one lowercase English letter at each step. Every word after begin must belong to the supplied dictionary; all words have equal length. Return zero if impossible. begin==end returns one even if absent from the dictionary.

Examples

Example 1. The shortest route contains five words.

```
word_ladder("hit","cog",["hot","dot","dog","lot","log","cog"])
=> 5
```

Example 2. The required final word is unavailable.

```
word_ladder("hit","cog",["hot","dot","dog"])
=> 0
```

Example 3. Equal endpoints need no transformation.

```
word_ladder("same","same",[])
=> 1
```

Why this mechanism works

Words are graph vertices and legal one-letter substitutions are unweighted edges. Generate neighbors when needed instead of building all pairwise comparisons. Remove a dictionary word when enqueuing it, combining membership and undiscovered status. BFS then yields the shortest word sequence.

State and boundaries

remaining contains dictionary words not yet discovered. Queue entries pair a word with its path length in words, including begin.

Protocol

1. Handle equal endpoints and a missing end.
2. Enqueue begin at length one.
3. Replace each position by each different letter.
4. Remove and enqueue every remaining candidate.
5. Return on reaching end, or zero on exhaustion.

Visualization

```
hit -> hot -> dot -> dog -> cog
word counts: 1,2,3,4,5
each arrow changes exactly one letter
```

Python implementation

```
from collections import deque
from string import ascii_lowercase

def word_ladder(begin, end, words):
    if begin == end:
        return 1
    remaining = set(words)
    if end not in remaining:
        return 0
    remaining.discard(begin)
    queue = deque([(begin, 1)])
    while queue:
        word, length = queue.popleft()
        for i in range(len(word)):
            for char in ascii_lowercase:
                if char == word[i]:
                    continue
                candidate = word[:i] + char + word[i + 1:]
                if candidate not in remaining:
                    continue
                if candidate == end:
                    return length + 1
                remaining.remove(candidate)
                queue.append((candidate, length + 1))
    return 0
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
word_ladder("hit","cog",["hot","dot","dog","lot","log","cog"])

L5 word_ladder: if begin == end:
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']

L7 word_ladder: remaining = set(words)
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog',
    'dog', 'dot', 'hot', 'log', 'lot'}

L11 word_ladder: queue = deque([(begin, 1)])
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog',
    'dog', 'dot', 'hot', 'log', 'lot'}; queue=deque([('hit', 1)])

L13 word_ladder: word, length = queue.popleft()
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog',
    'dog', 'dot', 'hot', 'log', 'lot'}; queue=deque([]); word='hit'; length=1

L14 word_ladder: for i in range(len(word)):
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog',
    'dog', 'dot', 'hot', 'log', 'lot'}; queue=deque([]); word='hit'; length=1; i=0
```

```
L15 word_ladder: for char in ascii_lowercase:
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog',
    'dog', 'dot', 'hot', 'log', 'lot'}; queue=deque([]); word='hit'; length=1; i=0; char='a'

L18 word_ladder: candidate = word[:i] + char + word[i + 1:]
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog',
    'dog', 'dot', 'hot', 'log', 'lot'}; queue=deque([]); word='hit'; length=1; i=0; char='a';
    candidate='ait'

L15 word_ladder: for char in ascii_lowercase:
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog',
    'dog', 'dot', 'hot', 'log', 'lot'}; queue=deque([]); word='hit'; length=1; i=0; char='b';
    candidate='ait'

... 637 intermediate state changes omitted ...

L15 word_ladder: for char in ascii_lowercase:
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog'};
    queue=deque([('log', 4)]); word='dog'; length=4; i=0; char='b'; candidate='aog'

L18 word_ladder: candidate = word[:i] + char + word[i + 1:]
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog'};
    queue=deque([('log', 4)]); word='dog'; length=4; i=0; char='b'; candidate='bog'

L15 word_ladder: for char in ascii_lowercase:
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog'};
    queue=deque([('log', 4)]); word='dog'; length=4; i=0; char='c'; candidate='bog'

L18 word_ladder: candidate = word[:i] + char + word[i + 1:]
begin='hit'; end='cog'; words=['hot', 'dot', 'dog', 'lot', 'log', 'cog']; remaining={'cog'};
    queue=deque([('log', 4)]); word='dog'; length=4; i=0; char='c'; candidate='cog'
```

Result: 5. The shortest route contains five words.

```
word_ladder("hit","cog",["hot","dot","dog"])

L5 word_ladder: if begin == end:
begin='hit'; end='cog'; words=['hot', 'dot', 'dog']

L7 word_ladder: remaining = set(words)
begin='hit'; end='cog'; words=['hot', 'dot', 'dog']; remaining={'dog', 'dot', 'hot'}
```

Result: 0. The required final word is unavailable.

Why the solution is correct

Every generated edge changes one character and lands in the dictionary. BFS first-discovery distances are shortest edge counts; starting at one converts them to word counts. Removing candidates at discovery prevents repeated scheduling without losing a shorter route.

Time and space

O(N*26*L^2) time in Python for up to N discovered words of length L, including slicing and hashing each candidate; O(N*L) dictionary/queue string storage.

Common mistakes

Starting the distance at zero returns transformations instead of words. Failing to remove on enqueue can greatly duplicate work across neighboring words.

Reasoning card

State: remaining contains dictionary words not yet discovered. Queue entries pair a word with its path length in words, including begin.

Invariant: First discovery of a word gives its shortest transformation count.

Decision to rehearse: Generate valid word neighbours and use BFS.

Finish: Return on reaching end, or zero on exhaustion

Transfer exercise

Use bidirectional BFS, always expanding the smaller frontier, and define how two frontier distances combine.

Module 20
Directed Graphs, Cycles, and Topological Order

Model prerequisite relationships and distinguish nodes currently on the DFS path from fully finished nodes.

Directed dependencies require a distinction between active recursion and completed work. Topological order turns a DAG into a valid computation schedule. Once that schedule is known, ancestor sets, path values and downstream effects become dynamic programs over vertices. Shared dependencies require set union rather than blindly adding counts.

The ten-step progression

191. Detect a Directed Cycle with Three States

192. Course Schedule Feasibility

193. Return One Course Order

194. Minimum Semesters

195. Eventual Safe Nodes

196. Longest Path in a DAG

197. Prerequisite Reachability Queries

198. Alien Dictionary Order

199. Reconstruct Itinerary

200. Dependency Impact Counts

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

191. Detect a Directed Cycle with Three States

Foundation | Module 20

Problem

Return whether an adjacency-list directed graph contains any directed cycle, including a self-loop. Inspect every component. Vertices are 0 through n-1; repeated edges are permitted. Empty input returns false.

Examples

Example 1. The final edge returns to an active ancestor.

```
directed_cycle([[1],[2],[0]])
=> True
```

Example 2. The shared finished descendant does not create a cycle.

```
directed_cycle([[1,2],[2],[]])
=> False
```

Example 3. There is no cycle in an empty graph.

```
directed_cycle([])
=> False
```

Why this mechanism works

A single visited flag cannot distinguish a completed branch from the current recursive ancestry. Use three states: unseen, active and finished. An edge to an active vertex closes a route back to an ancestor. An edge to a finished vertex merely reuses previously explored graph structure.

State and boundaries

state 0 means unvisited, 1 means on the active DFS call path, and 2 means all outgoing edges have been completely checked.

Protocol

1. Start DFS at each unseen vertex.
2. Mark the current vertex active.
3. Reject acyclicity on an edge to an active vertex.
4. Recursively inspect unseen neighbors.

5. Mark finished only after all outgoing edges return without a cycle.

Visualization

```
0(active) -> 1(active) -> 2(active) -> 0
edge to active ancestor: cycle
edge to a finished node: already-checked branch
```

Python implementation

```
def directed_cycle(adj):
    state = [0] * len(adj)
    def visit(u):
        state[u] = 1
        for v in adj[u]:
            if state[v] == 1:
                return True
            if state[v] == 0 and visit(v):
                return True
        state[u] = 2
        return False
    return any(state[u] == 0 and visit(u) for u in range(len(adj)))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
directed_cycle([[1],[2],[0]])

L2 directed_cycle: state = [0] * len(adj)
adj=[[1], [2], [0]]; state=[0, 0, 0]

L4 visit: state[u] = 1
u=0; adj=[[1], [2], [0]]; state=[1, 0, 0]

L5 visit: for v in adj[u]:
u=0; adj=[[1], [2], [0]]; state=[1, 0, 0]; v=1

L4 visit: state[u] = 1
u=1; adj=[[1], [2], [0]]; state=[1, 1, 0]

L5 visit: for v in adj[u]:
u=1; adj=[[1], [2], [0]]; state=[1, 1, 0]; v=2

L4 visit: state[u] = 1
u=2; adj=[[1], [2], [0]]; state=[1, 1, 1]

L5 visit: for v in adj[u]:
u=2; adj=[[1], [2], [0]]; state=[1, 1, 1]; v=0

L8 visit: if state[v] == 0 and visit(v):
u=1; adj=[[1], [2], [0]]; state=[1, 1, 1]; v=2

L8 visit: if state[v] == 0 and visit(v):
u=0; adj=[[1], [2], [0]]; state=[1, 1, 1]; v=1

L12 <genexpr>: return any(state[u] == 0 and visit(u) for u in range(len(adj)))
.0=range_iterator; u=0; state=[1, 1, 1]
```

```
L12 directed_cycle: return any(state[u] == 0 and visit(u) for u in range(len(adj)))
adj=[[1], [2], [0]]; state=[1, 1, 1]
```

Result: True. The final edge returns to an active ancestor.

```
directed_cycle([[1,2],[2],[]])

L2 directed_cycle: state = [0] * len(adj)
adj=[[1, 2], [2], []]; state=[0, 0, 0]

L4 visit: state[u] = 1
u=0; adj=[[1, 2], [2], []]; state=[1, 0, 0]

L5 visit: for v in adj[u]:
u=0; adj=[[1, 2], [2], []]; state=[1, 0, 0]; v=1

L4 visit: state[u] = 1
u=1; adj=[[1, 2], [2], []]; state=[1, 1, 0]

L5 visit: for v in adj[u]:
u=1; adj=[[1, 2], [2], []]; state=[1, 1, 0]; v=2

L4 visit: state[u] = 1
u=2; adj=[[1, 2], [2], []]; state=[1, 1, 1]

L10 visit: state[u] = 2
u=2; adj=[[1, 2], [2], []]; state=[1, 1, 2]

L8 visit: if state[v] == 0 and visit(v):
u=1; adj=[[1, 2], [2], []]; state=[1, 1, 2]; v=2

... 5 intermediate state changes omitted ...

L12 <genexpr>: return any(state[u] == 0 and visit(u) for u in range(len(adj)))
.0=range_iterator; u=1; state=[2, 2, 2]

L12 <genexpr>: return any(state[u] == 0 and visit(u) for u in range(len(adj)))
.0=range_iterator; u=2; state=[2, 2, 2]

L12 <genexpr>: return any(state[u] == 0 and visit(u) for u in range(len(adj)))
.0=range_iterator; u=2; state=[2, 2, 2]

L12 directed_cycle: return any(state[u] == 0 and visit(u) for u in range(len(adj)))
adj=[[1, 2], [2], []]; state=[2, 2, 2]
```

Result: False. The shared finished descendant does not create a cycle.

Why the solution is correct

An edge to an active ancestor combines with the DFS tree path to form a directed cycle. If a cycle exists, the first DFS exploration entering its unfinished vertices eventually encounters such a back edge. Finished vertices have already had every descendant edge checked, so cross edges to them are harmless.

Time and space

O(V+E) time and O(V) state plus recursion space. Deep graphs may require an iterative implementation.

Common mistakes

Marking a vertex finished before exploring its neighbors hides back edges. Treating every edge to any visited vertex as a cycle rejects DAGs with shared descendants.

Reasoning card

State: state 0 means unvisited, 1 means on the active DFS call path, and 2 means all outgoing edges have been completely checked.

Invariant: Visiting nodes are exactly the active recursion path.

Decision to rehearse: An edge to a visiting node is a back edge.

Finish: Mark finished only after all outgoing edges return without a cycle

Transfer exercise

Implement the same three-state protocol with explicit frames storing each node's next neighbor index.

192. Course Schedule Feasibility

Foundation | Module 20

Problem

Given n courses labeled 0 through n-1 and pairs (course,prerequisite), return whether all courses can be completed. Each pair means prerequisite must precede course. Duplicate pairs may appear. Zero courses is feasible; a self-prerequisite is not.

Examples

Example 1. The dependency chain can be completed in order.

```
can_finish(3,[(1,0),(2,1)])
=> True
```

Example 2. Each course waits for the other.

```
can_finish(2,[(1,0),(0,1)])
=> False
```

Example 3. No requirement obstructs the empty schedule.

```
can_finish(0,[])
=> True
```

Why this mechanism works

Turn each requirement into prerequisite->course. A course becomes available when all its incoming requirements have been removed. Repeatedly process zero-indegree courses and release their dependents. If a cycle remains, its members retain at least one unmet requirement and cannot enter the queue.

State and boundaries

indegree[v] counts prerequisite-edge occurrences from unprocessed courses. The queue contains currently available courses.

Protocol

1. Build outgoing lists and indegrees.
2. Enqueue every zero-indegree course.
3. Process one course and decrement each dependent's indegree.
4. Enqueue a dependent exactly when its count reaches zero.
5. Compare processed count with n.

Visualization

```
requirements (1,0),(2,1) -> 0 -> 1 -> 2
indegrees [0,1,1]
release 0, then 1, then 2 -> all complete
```

Python implementation

```
from collections import deque

def can_finish(n, prerequisites):
    adj = [[] for _ in range(n)]
    indegree = [0] * n
    for course, prerequisite in prerequisites:
        adj[prerequisite].append(course)
        indegree[course] += 1
    queue = deque(u for u in range(n) if indegree[u] == 0)
    processed = 0
    while queue:
        u = queue.popleft()
        processed += 1
        for v in adj[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
    return processed == n
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
can_finish(3,[(1,0),(2,1)])

L4 can_finish: adj = [[] for _ in range(n)]
n=3; prerequisites=[(1, 0), (2, 1)]; _=0

L4 can_finish: adj = [[] for _ in range(n)]
n=3; prerequisites=[(1, 0), (2, 1)]; _=1

L4 can_finish: adj = [[] for _ in range(n)]
n=3; prerequisites=[(1, 0), (2, 1)]; _=2

L4 can_finish: adj = [[] for _ in range(n)]
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[], [], []]

L5 can_finish: indegree = [0] * n
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[], [], []]; indegree=[0, 0, 0]

L6 can_finish: for course, prerequisite in prerequisites:
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[], [], []]; indegree=[0, 0, 0]; course=1;
    prerequisite=0

L7 can_finish: adj[prerequisite].append(course)
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[1], [], []]; indegree=[0, 0, 0]; course=1;
    prerequisite=0

L8 can_finish: indegree[course] += 1
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[1], [], []]; indegree=[0, 1, 0]; course=1;
    prerequisite=0
```

```
... 16 intermediate state changes omitted ...

L15 can_finish: indegree[v] -= 1
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[1], [2], []]; indegree=[0, 0, 0]; course=2;
    prerequisite=1; queue=deque([]); processed=2; u=1; v=2

L17 can_finish: queue.append(v)
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[1], [2], []]; indegree=[0, 0, 0]; course=2;
    prerequisite=1; queue=deque([2]); processed=2; u=1; v=2

L12 can_finish: u = queue.popleft()
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[1], [2], []]; indegree=[0, 0, 0]; course=2;
    prerequisite=1; queue=deque([]); processed=2; u=2; v=2

L13 can_finish: processed += 1
n=3; prerequisites=[(1, 0), (2, 1)]; adj=[[1], [2], []]; indegree=[0, 0, 0]; course=2;
    prerequisite=1; queue=deque([]); processed=3; u=2; v=2
```

Result: True. The dependency chain can be completed in order.

`can_finish(2,[(1,0),(0,1)])`

```
L4 can_finish: adj = [[] for _ in range(n)]
n=2; prerequisites=[(1, 0), (0, 1)]; _=0

L4 can_finish: adj = [[] for _ in range(n)]
n=2; prerequisites=[(1, 0), (0, 1)]; _=1

L4 can_finish: adj = [[] for _ in range(n)]
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[], []]

L5 can_finish: indegree = [0] * n
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[], []]; indegree=[0, 0]

L6 can_finish: for course, prerequisite in prerequisites:
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[], []]; indegree=[0, 0]; course=1; prerequisite=0

L7 can_finish: adj[prerequisite].append(course)
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], []]; indegree=[0, 0]; course=1; prerequisite=0

L8 can_finish: indegree[course] += 1
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], []]; indegree=[0, 1]; course=1; prerequisite=0

L6 can_finish: for course, prerequisite in prerequisites:
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], []]; indegree=[0, 1]; course=0; prerequisite=1

... 2 intermediate state changes omitted ...

L9 <genexpr>: queue = deque(u for u in range(n) if indegree[u] == 0)
.0=range_iterator; u=0; indegree=[1, 1]

L9 <genexpr>: queue = deque(u for u in range(n) if indegree[u] == 0)
.0=range_iterator; u=1; indegree=[1, 1]

L9 can_finish: queue = deque(u for u in range(n) if indegree[u] == 0)
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], [0]]; indegree=[1, 1]; course=0;
    prerequisite=1; queue=deque([])

L10 can_finish: processed = 0
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], [0]]; indegree=[1, 1]; course=0;
    prerequisite=1; queue=deque([]); processed=0
```

Result: False. Each course waits for the other.

Why the solution is correct

Each processed course has no remaining prerequisite, so the produced prefix is feasible. If processing stops early, every remaining vertex has an incoming edge from the remaining graph; following predecessors in a finite set yields a cycle. Conversely, a cycle can never release its own first member.

Time and space

O(n+m) time and space for m prerequisite pairs.

Common mistakes

Reversing pair meaning changes the intended order even though cycle existence may remain the same. Deduplicating adjacency without also deduplicating indegrees can leave counts permanently positive.

Reasoning card

State: indegree[v] counts prerequisite-edge occurrences from unprocessed courses. The queue contains currently available courses.

Invariant: A cycle is the only obstruction to a complete prerequisite order.

Decision to rehearse: Detect a directed cycle or process all zero-indegree nodes.

Finish: Compare processed count with n

Transfer exercise

Return the unprocessed courses after failure. Explain why they can include descendants of cycles as well as cycle members.

193. Return One Course Order

Foundation | Module 20

Problem

Return one feasible ordering of n courses with (course,prerequisite) pairs, or [] if impossible. Use FIFO Kahn traversal: initial zero-indegree vertices enter in numeric order and outgoing edges retain input order. The result is deterministic under this rule, but need not be lexicographically smallest.

Examples

Example 1. Course 3 is released only after both prerequisites.

```
course_order(4,[(1,0),(2,0),(3,1),(3,2)])
=> [0, 1, 2, 3]
```

Example 2. No full order exists.

```
course_order(2,[(1,0),(0,1)])
=> []
```

Example 3. Initially available courses follow numeric order.

```
course_order(3,[])
=> [0, 1, 2]
```

Why this mechanism works

Keep the actual sequence from the feasibility algorithm. Every popped vertex can safely follow the existing prefix because all of its prerequisites already appear there. A reusable adjacency-list topological_order helper makes that guarantee available to the following DAG problems.

State and boundaries

order is a valid topological prefix. indegree counts incoming edges whose sources are outside that prefix.

Protocol

1. Build prerequisite-to-course edges.
2. Initialize all indegrees and zero-indegree queue.
3. Append each popped vertex to order.
4. Release outgoing neighbors.
5. Return order only if its length equals n.

Visualization

```
ready [0,1]; order []
pop 0, release 2 -> ready [1,2]
pop 1, then 2 -> valid prefix grows monotonically
```

Python implementation

```
from collections import deque

def topological_order(adj):
    indegree = [0] * len(adj)
    for neighbors in adj:
        for v in neighbors:
            indegree[v] += 1
    queue = deque(u for u, degree in enumerate(indegree) if degree == 0)
    order = []
    while queue:
        u = queue.popleft()
        order.append(u)
        for v in adj[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
    return order if len(order) == len(adj) else []

def course_order(n, prerequisites):
    adj = [[] for _ in range(n)]
    for course, prerequisite in prerequisites:
        adj[prerequisite].append(course)
    return topological_order(adj)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
course_order(4,[(1,0),(2,0),(3,1),(3,2)])

L20 course_order: adj = [[] for _ in range(n)]
n=4; prerequisites=[(1, 0), (2, 0), (3, 1), (3, 2)]; _=0

L20 course_order: adj = [[] for _ in range(n)]
n=4; prerequisites=[(1, 0), (2, 0), (3, 1), (3, 2)]; _=1

L20 course_order: adj = [[] for _ in range(n)]
n=4; prerequisites=[(1, 0), (2, 0), (3, 1), (3, 2)]; _=2

L20 course_order: adj = [[] for _ in range(n)]
n=4; prerequisites=[(1, 0), (2, 0), (3, 1), (3, 2)]; _=3

L20 course_order: adj = [[] for _ in range(n)]
n=4; prerequisites=[(1, 0), (2, 0), (3, 1), (3, 2)]; adj=[[], [], [], []]

L21 course_order: for course, prerequisite in prerequisites:
n=4; prerequisites=[(1, 0), (2, 0), (3, 1), (3, 2)]; adj=[[], [], [], []]; course=1;
    prerequisite=0

L22 course_order: adj[prerequisite].append(course)
```

```
n=4; prerequisites=[(1, 0), (2, 0), (3, 1), (3, 2)]; adj=[[1], [], [], []]; course=1;
    prerequisite=0

L21 course_order: for course, prerequisite in prerequisites:
n=4; prerequisites=[(1, 0), (2, 0), (3, 1), (3, 2)]; adj=[[1], [], [], []]; course=2;
    prerequisite=0

... 36 intermediate state changes omitted ...

L14 topological_order: indegree[v] -= 1
adj=[[1, 2], [3], [3], []]; indegree=[0, 0, 0, 0]; neighbors=[]; v=3; queue=deque([]);
    order=[0, 1, 2]; u=2

L16 topological_order: queue.append(v)
adj=[[1, 2], [3], [3], []]; indegree=[0, 0, 0, 0]; neighbors=[]; v=3; queue=deque([3]);
    order=[0, 1, 2]; u=2

L11 topological_order: u = queue.popleft()
adj=[[1, 2], [3], [3], []]; indegree=[0, 0, 0, 0]; neighbors=[]; v=3; queue=deque([]);
    order=[0, 1, 2]; u=3

L12 topological_order: order.append(u)
adj=[[1, 2], [3], [3], []]; indegree=[0, 0, 0, 0]; neighbors=[]; v=3; queue=deque([]);
    order=[0, 1, 2, 3]; u=3
```

Result: [0, 1, 2, 3]. Course 3 is released only after both prerequisites.

`course_order(2,[(1,0),(0,1)])`

```
L20 course_order: adj = [[] for _ in range(n)]
n=2; prerequisites=[(1, 0), (0, 1)]; _=0

L20 course_order: adj = [[] for _ in range(n)]
n=2; prerequisites=[(1, 0), (0, 1)]; _=1

L20 course_order: adj = [[] for _ in range(n)]
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[], []]

L21 course_order: for course, prerequisite in prerequisites:
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[], []]; course=1; prerequisite=0

L22 course_order: adj[prerequisite].append(course)
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], []]; course=1; prerequisite=0

L21 course_order: for course, prerequisite in prerequisites:
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], []]; course=0; prerequisite=1

L22 course_order: adj[prerequisite].append(course)
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], [0]]; course=0; prerequisite=1

L4 topological_order: indegree = [0] * len(adj)
adj=[[1], [0]]; indegree=[0, 0]

... 6 intermediate state changes omitted ...

L8 <genexpr>: queue = deque(u for u, degree in enumerate(indegree) if degree == 0)
.0=enumerate; u=0; degree=1

L8 <genexpr>: queue = deque(u for u, degree in enumerate(indegree) if degree == 0)
.0=enumerate; u=1; degree=1

L8 topological_order: queue = deque(u for u, degree in enumerate(indegree) if degree == 0)
adj=[[1], [0]]; indegree=[1, 1]; neighbors=[0]; v=0; queue=deque([])
```

```
L9 topological_order: order = []
adj=[[1], [0]]; indegree=[1, 1]; neighbors=[0]; v=0; queue=deque([]); order=[]
```

Result: []. No full order exists.

Why the solution is correct

Every appended vertex has zero remaining indegree, so all of its predecessors precede it. Hence a full result respects every edge. If fewer than n vertices are processed, the remaining directed cycle prevents any full ordering, justifying the empty failure result.

Time and space

O(V+E) time and space for the generic helper; O(n+m) for courses.

Common mistakes

Returning a partial order on failure violates the all-courses contract. A FIFO queue does not maintain smallest available vertex after later releases.

Reasoning card

State: order is a valid topological prefix. indegree counts incoming edges whose sources are outside that prefix.

Invariant: Queue contains exactly currently prerequisite-free nodes.

Decision to rehearse: Pop zero-indegree nodes and decrement outgoing neighbours.

Finish: Return order only if its length equals n

Transfer exercise

Replace the FIFO queue with a min-heap to obtain the lexicographically smallest valid order.

194. Minimum Semesters

Foundation | Module 20

Problem

Return the minimum semesters needed to complete n courses with (course,prerequisite) pairs. Any number of available courses may be taken in parallel, each taking one semester; prerequisites must finish in earlier semesters. Return -1 for a cycle and zero for no courses.

Examples

Example 1. The two independent prerequisites share the first semester.

```
min_semesters(4,[(2,0),(2,1),(3,2)])
=> 3
```

Example 2. A cycle prevents completion.

```
min_semesters(2,[(1,0),(0,1)])
=> -1
```

Example 3. No semester is required.

```
min_semesters(0,[])
=> 0
```

Why this mechanism works

Kahn's available frontier forms one semester. Capture its width so dependents released during processing wait until the next semester. Delaying an already available course cannot help under unlimited parallel capacity, so taking the entire frontier is optimal.

State and boundaries

The queue at each outer-loop start contains exactly courses whose prerequisites finished in earlier semesters. semesters counts fully scheduled layers.

Protocol

1. Build prerequisite edges and indegrees.
2. Queue all initially available courses.
3. Increment semesters once per captured layer.
4. Process only that layer and release next-semester courses.
5. Return semesters if every course was processed, otherwise -1.

Visualization

```
0 -> 2
1 -> 2 -> 3
semester 1: [0,1]
semester 2: [2]
semester 3: [3]
```

Python implementation

```
from collections import deque

def min_semesters(n, prerequisites):
    adj, indegree = [[] for _ in range(n)], [0] * n
    for course, prerequisite in prerequisites:
        adj[prerequisite].append(course)
        indegree[course] += 1
    queue = deque(u for u in range(n) if indegree[u] == 0)
    semesters = processed = 0
    while queue:
        semesters += 1
        width = len(queue)
        for _ in range(width):
            u = queue.popleft()
            processed += 1
            for v in adj[u]:
                indegree[v] -= 1
                if indegree[v] == 0:
                    queue.append(v)
    return semesters if processed == n else -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
min_semesters(4,[(2,0),(2,1),(3,2)])

L4 min_semesters: adj, indegree = [[] for _ in range(n)], [0] * n
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; _=0

L4 min_semesters: adj, indegree = [[] for _ in range(n)], [0] * n
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; _=1

L4 min_semesters: adj, indegree = [[] for _ in range(n)], [0] * n
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; _=2

L4 min_semesters: adj, indegree = [[] for _ in range(n)], [0] * n
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; _=3

L4 min_semesters: adj, indegree = [[] for _ in range(n)], [0] * n
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; adj=[[], [], [], []]; indegree=[0, 0, 0, 0]

L5 min_semesters: for course, prerequisite in prerequisites:
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; adj=[[], [], [], []]; indegree=[0, 0, 0, 0];
    course=2; prerequisite=0

L6 min_semesters: adj[prerequisite].append(course)
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; adj=[[2], [], [], []]; indegree=[0, 0, 0, 0];
    course=2; prerequisite=0
```

```
L7 min_semesters: indegree[course] += 1
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; adj=[[2], [], [], []]; indegree=[0, 0, 1, 0];
    course=2; prerequisite=0

... 31 intermediate state changes omitted ...

L19 min_semesters: queue.append(v)
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; adj=[[2], [2], [3], []]; indegree=[0, 0, 0, 0];
    course=3; prerequisite=2; queue=deque([3]); semesters=2; processed=3; width=1; _=0; u=2;
    v=3

L11 min_semesters: semesters += 1
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; adj=[[2], [2], [3], []]; indegree=[0, 0, 0, 0];
    course=3; prerequisite=2; queue=deque([3]); semesters=3; processed=3; width=1; _=0; u=2;
    v=3

L14 min_semesters: u = queue.popleft()
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; adj=[[2], [2], [3], []]; indegree=[0, 0, 0, 0];
    course=3; prerequisite=2; queue=deque([]); semesters=3; processed=3; width=1; _=0; u=3;
    v=3

L15 min_semesters: processed += 1
n=4; prerequisites=[(2, 0), (2, 1), (3, 2)]; adj=[[2], [2], [3], []]; indegree=[0, 0, 0, 0];
    course=3; prerequisite=2; queue=deque([]); semesters=3; processed=4; width=1; _=0; u=3;
    v=3
```

Result: 3. The two independent prerequisites share the first semester.

```
min_semesters(2,[(1,0),(0,1)])

L4 min_semesters: adj, indegree = [[] for _ in range(n)], [0] * n
n=2; prerequisites=[(1, 0), (0, 1)]; _=0

L4 min_semesters: adj, indegree = [[] for _ in range(n)], [0] * n
n=2; prerequisites=[(1, 0), (0, 1)]; _=1

L4 min_semesters: adj, indegree = [[] for _ in range(n)], [0] * n
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[], []]; indegree=[0, 0]

L5 min_semesters: for course, prerequisite in prerequisites:
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[], []]; indegree=[0, 0]; course=1; prerequisite=0

L6 min_semesters: adj[prerequisite].append(course)
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], []]; indegree=[0, 0]; course=1; prerequisite=0

L7 min_semesters: indegree[course] += 1
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], []]; indegree=[0, 1]; course=1; prerequisite=0

L5 min_semesters: for course, prerequisite in prerequisites:
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], []]; indegree=[0, 1]; course=0; prerequisite=1

L6 min_semesters: adj[prerequisite].append(course)
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], [0]]; indegree=[0, 1]; course=0; prerequisite=1

... 1 intermediate state changes omitted ...

L8 <genexpr>: queue = deque(u for u in range(n) if indegree[u] == 0)
.0=range_iterator; u=0; indegree=[1, 1]

L8 <genexpr>: queue = deque(u for u in range(n) if indegree[u] == 0)
.0=range_iterator; u=1; indegree=[1, 1]

L8 min_semesters: queue = deque(u for u in range(n) if indegree[u] == 0)
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], [0]]; indegree=[1, 1]; course=0;
```

```
    prerequisite=1; queue=deque([])

L9 min_semesters: semesters = processed = 0
n=2; prerequisites=[(1, 0), (0, 1)]; adj=[[1], [0]]; indegree=[1, 1]; course=0;
    prerequisite=1; queue=deque([]); semesters=0; processed=0
```

Result: -1. A cycle prevents completion.

Why the solution is correct

A course cannot finish before one semester after its latest prerequisite. The layer algorithm schedules it at exactly that earliest possible semester. Induction over layers proves all finish times minimal; the largest is the minimum total semesters. An unprocessed cycle makes completion impossible.

Time and space

O(n+m) time and space.

Common mistakes

Processing newly released courses in the current semester lets prerequisites and dependents finish together illegally. A per-semester capacity limit would require a different optimization problem.

Reasoning card

State: The queue at each outer-loop start contains exactly courses whose prerequisites finished in earlier semesters. semesters counts fully scheduled layers.

Invariant: A layer contains all courses currently available in parallel.

Decision to rehearse: Each Kahn BFS layer is one semester.

Finish: Return semesters if every course was processed, otherwise -1

Transfer exercise

Return each course's earliest finishing semester and reconstruct a longest prerequisite chain.

195. Eventual Safe Nodes

Bridge | Module 20

Problem

Return, in ascending vertex order, all eventual safe nodes of a directed graph: vertices from which every possible path eventually ends at a terminal vertex. A vertex that can reach any directed cycle is unsafe. Empty input returns [].

Examples

Example 1. Only the separate terminating chain is safe.

```
safe_nodes([[1],[1],[3],[]])
=> [2, 3]
```

Example 2. Node 0 remains unsafe despite having one safe successor.

```
safe_nodes([[1,2],[],[2]])
=> [1]
```

Example 3. There are no vertices to classify.

```
safe_nodes([])
=> []
```

Why this mechanism works

Start with terminal vertices, whose safety is immediate. Reverse edges to notify predecessors whenever a successor becomes safe. A predecessor becomes safe only after every outgoing edge leads to a confirmed safe vertex. This is elimination by remaining outgoing risk, rather than ordinary incoming prerequisites.

State and boundaries

remaining[u] counts outgoing edges to successors not yet removed as safe. reverse[v] lists predecessors to notify when v is removed.

Protocol

1. Build reverse edges and outgoing counts.
2. Queue all terminal vertices.
3. Mark each removed vertex safe.
4. Decrement each predecessor's remaining count.
5. Queue a predecessor when its count reaches zero.
6. Return marked indices in ascending order.

Visualization

```
0 -> 1 -> 1   unsafe cycle and its predecessor
2 -> 3        terminal 3 is safe, then 2 becomes safe
output [2,3]
```

Python implementation

```
from collections import deque

def safe_nodes(adj):
    reverse = [[] for _ in adj]
    remaining = [len(neighbors) for neighbors in adj]
    for u, neighbors in enumerate(adj):
        for v in neighbors:
            reverse[v].append(u)
    queue = deque(u for u, degree in enumerate(remaining) if degree == 0)
    safe = [False] * len(adj)
    while queue:
        v = queue.popleft()
        safe[v] = True
        for u in reverse[v]:
            remaining[u] -= 1
            if remaining[u] == 0:
                queue.append(u)
    return [u for u in range(len(adj)) if safe[u]]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
safe_nodes([[1],[1],[3],[]])

L4 safe_nodes: reverse = [[] for _ in adj]
adj=[[1], [1], [3], []]; _=[1]

L4 safe_nodes: reverse = [[] for _ in adj]
adj=[[1], [1], [3], []]; _=[3]

L4 safe_nodes: reverse = [[] for _ in adj]
adj=[[1], [1], [3], []]; _=[]

L4 safe_nodes: reverse = [[] for _ in adj]
adj=[[1], [1], [3], []]; reverse=[[], [], [], []]

L5 safe_nodes: remaining = [len(neighbors) for neighbors in adj]
adj=[[1], [1], [3], []]; reverse=[[], [], [], []]; neighbors=[1]

L5 safe_nodes: remaining = [len(neighbors) for neighbors in adj]
adj=[[1], [1], [3], []]; reverse=[[], [], [], []]; neighbors=[3]

L5 safe_nodes: remaining = [len(neighbors) for neighbors in adj]
adj=[[1], [1], [3], []]; reverse=[[], [], [], []]; neighbors=[]

L5 safe_nodes: remaining = [len(neighbors) for neighbors in adj]
adj=[[1], [1], [3], []]; reverse=[[], [], [], []]; remaining=[1, 1, 1, 0]

... 24 intermediate state changes omitted ...
```

```
L18 safe_nodes: return [u for u in range(len(adj)) if safe[u]]
adj=[[1], [1], [3], []]; reverse=[[], [0, 1], [], [2]]; remaining=[1, 1, 0, 0]; neighbors=[];
    u=1; v=2; queue=deque([]); safe=[False, False, True, True]

L18 safe_nodes: return [u for u in range(len(adj)) if safe[u]]
adj=[[1], [1], [3], []]; reverse=[[], [0, 1], [], [2]]; remaining=[1, 1, 0, 0]; neighbors=[];
    u=2; v=2; queue=deque([]); safe=[False, False, True, True]

L18 safe_nodes: return [u for u in range(len(adj)) if safe[u]]
adj=[[1], [1], [3], []]; reverse=[[], [0, 1], [], [2]]; remaining=[1, 1, 0, 0]; neighbors=[];
    u=3; v=2; queue=deque([]); safe=[False, False, True, True]

L18 safe_nodes: return [u for u in range(len(adj)) if safe[u]]
adj=[[1], [1], [3], []]; reverse=[[], [0, 1], [], [2]]; remaining=[1, 1, 0, 0]; neighbors=[];
    u=2; v=2; queue=deque([]); safe=[False, False, True, True]
```

Result: [2, 3]. Only the separate terminating chain is safe.

`safe_nodes([[1,2],[],[2]])`

```
L4 safe_nodes: reverse = [[] for _ in adj]
adj=[[1, 2], [], [2]]; _=[1, 2]

L4 safe_nodes: reverse = [[] for _ in adj]
adj=[[1, 2], [], [2]]; _=[]

L4 safe_nodes: reverse = [[] for _ in adj]
adj=[[1, 2], [], [2]]; _=[2]

L4 safe_nodes: reverse = [[] for _ in adj]
adj=[[1, 2], [], [2]]; reverse=[[], [], []]

L5 safe_nodes: remaining = [len(neighbors) for neighbors in adj]
adj=[[1, 2], [], [2]]; reverse=[[], [], []]; neighbors=[1, 2]

L5 safe_nodes: remaining = [len(neighbors) for neighbors in adj]
adj=[[1, 2], [], [2]]; reverse=[[], [], []]; neighbors=[]

L5 safe_nodes: remaining = [len(neighbors) for neighbors in adj]
adj=[[1, 2], [], [2]]; reverse=[[], [], []]; neighbors=[2]

L5 safe_nodes: remaining = [len(neighbors) for neighbors in adj]
adj=[[1, 2], [], [2]]; reverse=[[], [], []]; remaining=[2, 0, 1]

... 16 intermediate state changes omitted ...

L15 safe_nodes: remaining[u] -= 1
adj=[[1, 2], [], [2]]; reverse=[[], [0], [0, 2]]; remaining=[1, 0, 1]; neighbors=[2]; u=0;
    v=1; queue=deque([]); safe=[False, True, False]

L18 safe_nodes: return [u for u in range(len(adj)) if safe[u]]
adj=[[1, 2], [], [2]]; reverse=[[], [0], [0, 2]]; remaining=[1, 0, 1]; neighbors=[2]; u=1;
    v=1; queue=deque([]); safe=[False, True, False]

L18 safe_nodes: return [u for u in range(len(adj)) if safe[u]]
adj=[[1, 2], [], [2]]; reverse=[[], [0], [0, 2]]; remaining=[1, 0, 1]; neighbors=[2]; u=2;
    v=1; queue=deque([]); safe=[False, True, False]

L18 safe_nodes: return [u for u in range(len(adj)) if safe[u]]
adj=[[1, 2], [], [2]]; reverse=[[], [0], [0, 2]]; remaining=[1, 0, 1]; neighbors=[2]; u=0;
    v=1; queue=deque([]); safe=[False, True, False]
```

Result: [1]. Node 0 remains unsafe despite having one safe successor.

Why the solution is correct

Terminals are safe. If all successors are safe, every path from their predecessor takes one edge into a terminating path, so it too is safe. Vertices left over each retain a successor in the remainder; repeatedly following those successors reaches a cycle. Thus the eliminated set is exactly the safe set.

Time and space

O(V+E) time and space, including the final ascending index scan.

Common mistakes

One safe successor is insufficient because the definition quantifies over every path. Incoming degree measures a different property.

Reasoning card

State: remaining[u] counts outgoing edges to successors not yet removed as safe. reverse[v] lists predecessors to notify when v is removed.

Invariant: Safe status is assigned only after all successors are safe.

Decision to rehearse: Reverse edges and eliminate nodes whose outgoing risk reaches zero, or DFS colours.

Finish: Return marked indices in ascending order

Transfer exercise

Explain how three-state DFS can classify safety, and distinguish a current-path marker from a permanently unsafe result.

196. Longest Path in a DAG

Bridge | Module 20

Problem

For a weighted DAG with n vertices and edges (u,v,weight), return the maximum total weight of a path that may start and finish anywhere. Weights may be negative. A zero-edge path has weight zero, so the answer is at least zero, including n=0. Raise ValueError if the graph is cyclic.

Examples

Example 1. A path may begin after an unfavorable prefix.

```
longest_dag_path(3,[(0,1,-5),(1,2,7)])
=> 7
```

Example 2. Unit weights count edges on the longest route.

```
longest_dag_path(4,[(0,1,1),(1,2,1),(0,3,1)])
=> 2
```

Example 3. The empty graph uses the defined zero result.

```
longest_dag_path(0,[])
=> 0
```

Why this mechanism works

Topological order ensures every predecessor contributes before a vertex is used to extend a path. Store the best weight of a path ending at each vertex. Initializing every value to zero allows a path to start anywhere and prevents a negative prefix from being forced into a later path. Unit weights specialize this to unweighted path length in edges.

State and boundaries

best[v] is the largest known weight ending at v. When v is processed in topological order, all incoming alternatives have already been considered.

Protocol

1. Build weighted and unweighted adjacency.

2. Obtain a full topological order or reject a cycle.

3. Initialize all best values to zero.

4. For each ordered u, relax every edge with best[u]+weight.

5. Return the maximum, defaulting to zero.

Visualization

```
0 --(-5)--> 1 --(7)--> 2
best initially [0,0,0]
best[1] remains 0; best[2] becomes 7
optimal path starts at 1, avoiding the negative prefix
```

Python implementation

```
from kata_193 import topological_order

def longest_dag_path(n, edges):
    adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
    for u, v, weight in edges:
        adj[u].append(v)
        weighted[u].append((v, weight))
    order = topological_order(adj)
    if len(order) != n:
        raise ValueError('graph must be acyclic')
    best = [0] * n
    for u in order:
        for v, weight in weighted[u]:
            best[v] = max(best[v], best[u] + weight)
    return max(best, default=0)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
longest_dag_path(3,[(0,1,-5),(1,2,7)])

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=3; edges=[(0, 1, -5), (1, 2, 7)]; _=0

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=3; edges=[(0, 1, -5), (1, 2, 7)]; _=1

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=3; edges=[(0, 1, -5), (1, 2, 7)]; _=2

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=3; edges=[(0, 1, -5), (1, 2, 7)]; _=0

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=3; edges=[(0, 1, -5), (1, 2, 7)]; _=1

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=3; edges=[(0, 1, -5), (1, 2, 7)]; _=2

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=3; edges=[(0, 1, -5), (1, 2, 7)]; adj=[[], [], []]; weighted=[[], [], []]

L5 longest_dag_path: for u, v, weight in edges:
n=3; edges=[(0, 1, -5), (1, 2, 7)]; adj=[[], [], []]; weighted=[[], [], []]; u=0; v=1;
    weight=-5
```

```
... 9 intermediate state changes omitted ...

L12 longest_dag_path: for u in order:
n=3; edges=[(0, 1, -5), (1, 2, 7)]; adj=[[1], [2], []]; weighted=[[(..., ...)], [(..., ...)],
    []]; u=1; v=1; weight=-5; order=[0, 1, 2]; best=[0, 0, 0]

L13 longest_dag_path: for v, weight in weighted[u]:
n=3; edges=[(0, 1, -5), (1, 2, 7)]; adj=[[1], [2], []]; weighted=[[(..., ...)], [(..., ...)],
    []]; u=1; v=2; weight=7; order=[0, 1, 2]; best=[0, 0, 0]

L14 longest_dag_path: best[v] = max(best[v], best[u] + weight)
n=3; edges=[(0, 1, -5), (1, 2, 7)]; adj=[[1], [2], []]; weighted=[[(..., ...)], [(..., ...)],
    []]; u=1; v=2; weight=7; order=[0, 1, 2]; best=[0, 0, 7]

L12 longest_dag_path: for u in order:
n=3; edges=[(0, 1, -5), (1, 2, 7)]; adj=[[1], [2], []]; weighted=[[(..., ...)], [(..., ...)],
    []]; u=2; v=2; weight=7; order=[0, 1, 2]; best=[0, 0, 7]
```

Result: 7. A path may begin after an unfavorable prefix.

```
longest_dag_path(4,[(0,1,1),(1,2,1),(0,3,1)])

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; _=0

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; _=1

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; _=2

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; _=3

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; _=0

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; _=1

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; _=2

L4 longest_dag_path: adj, weighted = [[] for _ in range(n)], [[] for _ in range(n)]
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; _=3

... 17 intermediate state changes omitted ...

L13 longest_dag_path: for v, weight in weighted[u]:
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; adj=[[1, 3], [2], [], []]; weighted=[[(...,
    ...), (..., ...)], [(..., ...)], [], []]; u=1; v=2; weight=1; order=[0, 1, 3, 2]; best=[0,
     1, 0, 1]

L14 longest_dag_path: best[v] = max(best[v], best[u] + weight)
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; adj=[[1, 3], [2], [], []]; weighted=[[(...,
    ...), (..., ...)], [(..., ...)], [], []]; u=1; v=2; weight=1; order=[0, 1, 3, 2]; best=[0,
     1, 2, 1]

L12 longest_dag_path: for u in order:
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; adj=[[1, 3], [2], [], []]; weighted=[[(...,
    ...), (..., ...)], [(..., ...)], [], []]; u=3; v=2; weight=1; order=[0, 1, 3, 2]; best=[0,
     1, 2, 1]
```

```
L12 longest_dag_path: for u in order:
n=4; edges=[(0, 1, 1), (1, 2, 1), (0, 3, 1)]; adj=[[1, 3], [2], [], []]; weighted=[[(...,
    ...), (..., ...)], [(..., ...)], [], []]; u=2; v=2; weight=1; order=[0, 1, 3, 2]; best=[0,
     1, 2, 1]
```

Result: 2. Unit weights count edges on the longest route.

Why the solution is correct

Any nonempty path ending at v extends a path ending at a predecessor u. The recurrence considers every such last edge after best[u] is final; zero also considers starting at v. Induction along topological order makes each result exact, and their maximum covers every endpoint.

Time and space

O(V+E) time and space. Negative edges are safe because acyclicity, rather than nonnegative weights, orders dependencies.

Common mistakes

Using a single-source initialization answers a different question. Processing vertices by numeric ID is not necessarily a topological order.

Reasoning card

State: best[v] is the largest known weight ending at v. When v is processed in topological order, all incoming alternatives have already been considered.

Invariant: All predecessors are finalized before a node is processed.

Decision to rehearse: Relax edges in topological order.

Finish: Return the maximum, defaulting to zero

Transfer exercise

Require a nonempty path, then revise the initialization and all-negative case explicitly.

197. Prerequisite Reachability Queries

Bridge | Module 20

Problem

For a DAG adjacency list where u->v means u is a prerequisite of v, answer queries (u,v) asking whether a nonempty directed path exists from u to v. Return booleans in query order. Self-queries are false in a DAG. Raise ValueError for a cyclic input.

Examples

Example 1. Transitive reachability, unrelated vertices and self-queries differ.

```
prerequisite_queries([[1,2],[3],[3],[]],[(0,3),(1,2),(3,3)])
=> [True, False, False]
```

Example 2. No edge implies no strict prerequisite.

```
prerequisite_queries([[],[]],[(0,1)])
=> [False]
```

Example 3. An empty query batch produces no answers.

```
prerequisite_queries([],[])
=> []
```

Why this mechanism works

Propagate complete ancestor sets forward through topological order. For an edge u->v, every ancestor of u is also an ancestor of v, and u itself is a direct ancestor. Set union prevents shared prerequisite routes from duplicating facts.

State and boundaries

ancestors[v] contains discovered strict prerequisites of v. When u is processed, its ancestor set is complete.

Protocol

1. Obtain topological order and validate acyclicity.

2. Allocate an empty set per vertex.

3. For each ordered u and outgoing v, union ancestors[u] into ancestors[v] and add u.

4. Answer each query by set membership.

Visualization

```
0 -> 1 -> 3
 \-> 2 -> 3
ancestors[3] = {0,1,2}, with 0 included once
```

Python implementation

```
from kata_193 import topological_order

def prerequisite_queries(adj, queries):
    order = topological_order(adj)
    if len(order) != len(adj):
        raise ValueError('graph must be acyclic')
    ancestors = [set() for _ in adj]
    for u in order:
        for v in adj[u]:
            ancestors[v].update(ancestors[u])
            ancestors[v].add(u)
    return [u in ancestors[v] for u, v in queries]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
prerequisite_queries([[1,2],[3],[3],[]],[(0,3),(1,2),(3,3)])

L4 prerequisite_queries: order = topological_order(adj)
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3]

L7 prerequisite_queries: ancestors = [set() for _ in adj]
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3]; _=[1, 2]

L7 prerequisite_queries: ancestors = [set() for _ in adj]
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3]; _=[3]

L7 prerequisite_queries: ancestors = [set() for _ in adj]
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3]; _=[]

L7 prerequisite_queries: ancestors = [set() for _ in adj]
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3];
    ancestors=[set(), set(), set(), set()]

L8 prerequisite_queries: for u in order:
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3];
    ancestors=[set(), set(), set(), set()]; u=0

L9 prerequisite_queries: for v in adj[u]:
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3];
    ancestors=[set(), set(), set(), set()]; u=0; v=1

L11 prerequisite_queries: ancestors[v].add(u)
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3];
    ancestors=[set(), {0}, set(), set()]; u=0; v=1

... 8 intermediate state changes omitted ...

L8 prerequisite_queries: for u in order:
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3];
    ancestors=[set(), {0}, {0}, {0, 1, 2}]; u=3; v=3
```

```
L12 prerequisite_queries: return [u in ancestors[v] for u, v in queries]
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3];
    ancestors=[set(), {0}, {0}, {0, 1, 2}]; u=0; v=3

L12 prerequisite_queries: return [u in ancestors[v] for u, v in queries]
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3];
    ancestors=[set(), {0}, {0}, {0, 1, 2}]; u=1; v=2

L12 prerequisite_queries: return [u in ancestors[v] for u, v in queries]
adj=[[1, 2], [3], [3], []]; queries=[(0, 3), (1, 2), (3, 3)]; order=[0, 1, 2, 3];
    ancestors=[set(), {0}, {0}, {0, 1, 2}]; u=3; v=3
```

Result: [True, False, False]. Transitive reachability, unrelated vertices and self-queries differ.

```
prerequisite_queries([[],[]],[(0,1)])

L4 prerequisite_queries: order = topological_order(adj)
adj=[[], []]; queries=[(0, 1)]; order=[0, 1]

L7 prerequisite_queries: ancestors = [set() for _ in adj]
adj=[[], []]; queries=[(0, 1)]; order=[0, 1]; _=[]

L7 prerequisite_queries: ancestors = [set() for _ in adj]
adj=[[], []]; queries=[(0, 1)]; order=[0, 1]; ancestors=[set(), set()]

L8 prerequisite_queries: for u in order:
adj=[[], []]; queries=[(0, 1)]; order=[0, 1]; ancestors=[set(), set()]; u=0

L8 prerequisite_queries: for u in order:
adj=[[], []]; queries=[(0, 1)]; order=[0, 1]; ancestors=[set(), set()]; u=1

L12 prerequisite_queries: return [u in ancestors[v] for u, v in queries]
adj=[[], []]; queries=[(0, 1)]; order=[0, 1]; ancestors=[set(), set()]; u=0; v=1

L12 prerequisite_queries: return [u in ancestors[v] for u, v in queries]
adj=[[], []]; queries=[(0, 1)]; order=[0, 1]; ancestors=[set(), set()]; u=1
```

Result: [False]. No edge implies no strict prerequisite.

Why the solution is correct

Every propagated fact corresponds to an existing path extended by one edge. Conversely, every path to v ends at some predecessor u, whose complete ancestors are propagated before v is processed. Thus all and only strict reachability facts are present when queries are answered.

Time and space

O(V+E+E*V+Q) worst-case time with hash sets and Q queries; O(V^2+E+Q) total storage including output. Dense reachability can be quadratic even in a sparse DAG.

Common mistakes

Adding a node to its own ancestor set changes self-query semantics. Summing counts cannot answer which prerequisite is shared by two routes.

Reasoning card

State: ancestors[v] contains discovered strict prerequisites of v. When u is processed, its ancestor set is complete.

Invariant: Known prerequisite relationships are transitively closed for processed nodes.

Decision to rehearse: Propagate ancestor sets in topological order or run graph search per source.

Finish: Answer each query by set membership

Transfer exercise

Represent ancestor sets as bit masks and discuss word-level speedups and Python big-integer costs.

198. Alien Dictionary Order

Interview | Module 20

Problem

Given words sorted according to an unknown alphabet, return one valid ordering of all distinct characters. Choose the lexicographically smallest valid character ordering under ordinary Python character comparison. Return '' for inconsistent constraints, including a longer word immediately preceding its strict prefix. Empty input also returns ''.

Examples

Example 1. The adjacent comparisons force this complete character order.

```
alien_order(["wrt","wrf","er","ett","rftt"])
=> 'wertf'
```

Example 2. A longer word cannot precede its own strict prefix.

```
alien_order(["abc","ab"])
=> ''
```

Example 3. Unconstrained characters are emitted in the smallest available order.

```
alien_order(["ab"])
=> 'ab'
```

Example 4. The inferred z<x and x<z constraints form a cycle.

```
alien_order(["z","x","z"])
=> ''
```

Why this mechanism works

Adjacent sorted words reveal only their first differing character: that pair imposes one necessary ordering edge. Later positions cannot add constraints once the comparison is decided. Collect all characters, deduplicate edges, detect the prefix contradiction, then topologically sort with a min-heap to resolve unconstrained choices deterministically.

State and boundaries

adj stores distinct necessary character-order edges. indegree counts unmet distinct constraints; the heap contains currently eligible characters.

Protocol

1. Register every character.

2. Compare adjacent words until their first mismatch.

3. Add that edge once, or reject a reversed prefix.

4. Heapify zero-indegree characters.

5. Emit the smallest available character and release neighbors.

6. Reject if not all characters were emitted.

Visualization

```
'wrt' before 'wrf' -> t < f
'wrf' before 'er'  -> w < e
'abc' before 'ab'  -> impossible prefix order
```

Python implementation

```
import heapq

def alien_order(words):
    adj = {char: set() for word in words for char in word}
    indegree = dict.fromkeys(adj, 0)
    for first, second in zip(words, words[1:]):
        for a, b in zip(first, second):
            if a != b:
                if b not in adj[a]:
                    adj[a].add(b)
                    indegree[b] += 1
                break
        else:
            if len(first) > len(second):
                return ''
    heap = [char for char in adj if indegree[char] == 0]
    heapq.heapify(heap)
    output = []
    while heap:
        char = heapq.heappop(heap)
        output.append(char)
        for neighbor in adj[char]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                heapq.heappush(heap, neighbor)
    return ''.join(output) if len(output) == len(adj) else ''
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
alien_order(["wrt","wrf","er","ett","rftt"])

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; word='wrt'; char='w'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; word='wrt'; char='r'
```

```
L4 alien_order: adj = {char: set() for word in words for char in word}
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; word='wrt'; char='t'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; word='wrf'; char='w'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; word='wrf'; char='r'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; word='wrf'; char='f'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; word='er'; char='e'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; word='er'; char='r'

... 51 intermediate state changes omitted ...

L23 alien_order: indegree[neighbor] -= 1
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; adj={'w':{'e'}, 'r':{'t'}, 't':{'f'}, 'f':set(),
    'e':{'r'}}; indegree={'w':0, 'r':0, 't':0, 'f':0, 'e':0}; first='ett'; second='rftt';
    a='e'; b='r'; heap=[]; output=['w', 'e', 'r', 't']; char='t'; neighbor='f'

L25 alien_order: heapq.heappush(heap, neighbor)
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; adj={'w':{'e'}, 'r':{'t'}, 't':{'f'}, 'f':set(),
    'e':{'r'}}; indegree={'w':0, 'r':0, 't':0, 'f':0, 'e':0}; first='ett'; second='rftt';
    a='e'; b='r'; heap=['f']; output=['w', 'e', 'r', 't']; char='t'; neighbor='f'

L20 alien_order: char = heapq.heappop(heap)
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; adj={'w':{'e'}, 'r':{'t'}, 't':{'f'}, 'f':set(),
    'e':{'r'}}; indegree={'w':0, 'r':0, 't':0, 'f':0, 'e':0}; first='ett'; second='rftt';
    a='e'; b='r'; heap=[]; output=['w', 'e', 'r', 't']; char='f'; neighbor='f'

L21 alien_order: output.append(char)
words=['wrt', 'wrf', 'er', 'ett', 'rftt']; adj={'w':{'e'}, 'r':{'t'}, 't':{'f'}, 'f':set(),
    'e':{'r'}}; indegree={'w':0, 'r':0, 't':0, 'f':0, 'e':0}; first='ett'; second='rftt';
    a='e'; b='r'; heap=[]; output=['w', 'e', 'r', 't', 'f']; char='f'; neighbor='f'
```

Result: 'wertf'. The adjacent comparisons force this complete character order.

```
alien_order(["abc","ab"])

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['abc', 'ab']; word='abc'; char='a'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['abc', 'ab']; word='abc'; char='b'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['abc', 'ab']; word='abc'; char='c'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['abc', 'ab']; word='ab'; char='a'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['abc', 'ab']; word='ab'; char='b'

L4 alien_order: adj = {char: set() for word in words for char in word}
words=['abc', 'ab']; adj={'a':set(), 'b':set(), 'c':set()}

L5 alien_order: indegree = dict.fromkeys(adj, 0)
words=['abc', 'ab']; adj={'a':set(), 'b':set(), 'c':set()}; indegree={'a':0, 'b':0, 'c':0}
```

```
L6 alien_order: for first, second in zip(words, words[1:]):
words=['abc', 'ab']; adj={'a':set(), 'b':set(), 'c':set()}; indegree={'a':0, 'b':0, 'c':0};
    first='abc'; second='ab'

L7 alien_order: for a, b in zip(first, second):
words=['abc', 'ab']; adj={'a':set(), 'b':set(), 'c':set()}; indegree={'a':0, 'b':0, 'c':0};
    first='abc'; second='ab'; a='a'; b='a'

L7 alien_order: for a, b in zip(first, second):
words=['abc', 'ab']; adj={'a':set(), 'b':set(), 'c':set()}; indegree={'a':0, 'b':0, 'c':0};
    first='abc'; second='ab'; a='b'; b='b'
```

Result: ''. A longer word cannot precede its own strict prefix.

Why the solution is correct

The first mismatch or prefix relation fully determines each adjacent lexical comparison. Therefore satisfying all extracted edges satisfies the given list. Kahn processing respects those edges; choosing the smallest currently feasible character is the smallest possible next character of any valid order. A cycle or reversed prefix makes an order impossible.

Time and space

O(S+E+A log A) time for S input characters, A distinct characters and E distinct inferred edges; O(A+E) auxiliary space.

Common mistakes

Inferring edges from every mismatching position invents constraints. Counting a duplicate edge twice but removing it once breaks indegrees.

Reasoning card

State: adj stores distinct necessary character-order edges. indegree counts unmet distinct constraints; the heap contains currently eligible characters.

Invariant: Every inferred edge is necessary to preserve the given word order.

Decision to rehearse: Create edges from the first differing character and topologically sort.

Finish: Reject if not all characters were emitted

Transfer exercise

Detect whether the valid alphabet order is unique by observing how many characters are eligible at each step.

199. Reconstruct Itinerary

Interview | Module 20

Problem

Given directed airline tickets (origin,destination) and a starting airport, return the lexicographically smallest route using every ticket exactly once. Repeated tickets are distinct edges. A valid complete itinerary from start is guaranteed. With no tickets, return [start]. Leave the ticket list unchanged.

Examples

Example 1. The alphabetically smaller dead end is placed last so all tickets are used.

```
reconstruct_itinerary([("JFK","KUL"),("JFK","NRT"),("NRT","JFK")])
=> ['JFK', 'NRT', 'JFK', 'KUL']
```

Example 2. The empty route contains its starting airport.

```
reconstruct_itinerary([],"SIN")
=> ['SIN']
```

Example 3. Both copies of A-to-B are consumed.

```
reconstruct_itinerary([("A","B"),("B","A"),("A","B")],"A")
=> ['A', 'B', 'A', 'B']
```

Why this mechanism works

A naive greedy route can reach a dead end too early. Hierholzer's method consumes edges while deferring a vertex's final placement until it has no outgoing tickets left. Sorting outgoing destinations in reverse permits efficient removal of the smallest one; reversing the completed postorder stitches detours into a full Euler route.

State and boundaries

stack is the current unfinished walk. adj contains unconsumed tickets. reverse_route collects vertices only after their outgoing tickets are exhausted.

Protocol

1. Build reverse-sorted destination lists.
2. Start a stack at the requested airport.
3. Follow and remove the smallest available outgoing ticket.
4. When none remains, pop that airport into reverse_route.
5. Reverse the completed route.

Visualization

```
tickets JFK->KUL, JFK->NRT, NRT->JFK
KUL is explored first but finishes first
reverse postorder -> JFK,NRT,JFK,KUL
the dead end belongs at the end
```

Python implementation

```
from collections import defaultdict

def reconstruct_itinerary(tickets, start='JFK'):
    adj = defaultdict(list)
    for origin, destination in tickets:
        adj[origin].append(destination)
    for destinations in adj.values():
        destinations.sort(reverse=True)
    stack, reverse_route = [start], []
    while stack:
        if adj[stack[-1]]:
            stack.append(adj[stack[-1]].pop())
        else:
            reverse_route.append(stack.pop())
    return reverse_route[::-1]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
reconstruct_itinerary([("JFK","KUL"),("JFK","NRT"),("NRT","JFK")])

L4 reconstruct_itinerary: adj = defaultdict(list)
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={}

L5 reconstruct_itinerary: for origin, destination in tickets:
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={}; origin='JFK';
    destination='KUL'

L6 reconstruct_itinerary: adj[origin].append(destination)
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':['KUL']};
    origin='JFK'; destination='KUL'

L5 reconstruct_itinerary: for origin, destination in tickets:
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':['KUL']};
    origin='JFK'; destination='NRT'

L6 reconstruct_itinerary: adj[origin].append(destination)
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':['KUL',
    'NRT']}; origin='JFK'; destination='NRT'

L5 reconstruct_itinerary: for origin, destination in tickets:
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':['KUL',
    'NRT']}; origin='NRT'; destination='JFK'

L6 reconstruct_itinerary: adj[origin].append(destination)
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':['KUL',
    'NRT'], 'NRT':['JFK']}; origin='NRT'; destination='JFK'

L7 reconstruct_itinerary: for destinations in adj.values():
```

```
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':['KUL',
    'NRT'], 'NRT':['JFK']}; origin='NRT'; destination='JFK'; destinations=['KUL', 'NRT']

... 7 intermediate state changes omitted ...

L12 reconstruct_itinerary: stack.append(adj[stack[-1]].pop())
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':[],
    'NRT':[], 'KUL':[]}; origin='NRT'; destination='JFK'; destinations=[]; stack=['JFK',
    'NRT', 'JFK']; reverse_route=['KUL']

L14 reconstruct_itinerary: reverse_route.append(stack.pop())
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':[],
    'NRT':[], 'KUL':[]}; origin='NRT'; destination='JFK'; destinations=[]; stack=['JFK',
    'NRT']; reverse_route=['KUL', 'JFK']

L14 reconstruct_itinerary: reverse_route.append(stack.pop())
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':[],
    'NRT':[], 'KUL':[]}; origin='NRT'; destination='JFK'; destinations=[]; stack=['JFK'];
    reverse_route=['KUL', 'JFK', 'NRT']

L14 reconstruct_itinerary: reverse_route.append(stack.pop())
tickets=[('JFK', 'KUL'), ('JFK', 'NRT'), ('NRT', 'JFK')]; start='JFK'; adj={'JFK':[],
    'NRT':[], 'KUL':[]}; origin='NRT'; destination='JFK'; destinations=[]; stack=[];
    reverse_route=['KUL', 'JFK', 'NRT', 'JFK']
```

Result: ['JFK', 'NRT', 'JFK', 'KUL']. The alphabetically smaller dead end is placed last so all tickets are used.

```
reconstruct_itinerary([],"SIN")

L4 reconstruct_itinerary: adj = defaultdict(list)
tickets=[]; start='SIN'; adj={}

L9 reconstruct_itinerary: stack, reverse_route = [start], []
tickets=[]; start='SIN'; adj={}; stack=['SIN']; reverse_route=[]

L11 reconstruct_itinerary: if adj[stack[-1]]:
tickets=[]; start='SIN'; adj={'SIN':[]}; stack=['SIN']; reverse_route=[]

L14 reconstruct_itinerary: reverse_route.append(stack.pop())
tickets=[]; start='SIN'; adj={'SIN':[]}; stack=[]; reverse_route=['SIN']
```

Result: ['SIN']. The empty route contains its starting airport.

Why the solution is correct

Each ticket is removed once, and postorder splices completed trails into their shared vertices, yielding an Euler route under the existence precondition. Smallest-first exploration gives lexical priority to smaller feasible outgoing choices; a premature dead-end segment is placed at the route's end by postorder rather than blocking remaining tickets.

Time and space

O(E log E) sorting time and O(E) traversal time/space, including the E+1 airport output.

Common mistakes

Appending airports directly to the final forward route can strand unused tickets. Deduplicating repeated tickets loses required edge occurrences. Invalid-input detection is outside the guaranteed-existence contract.

Reasoning card

State: stack is the current unfinished walk. adj contains unconsumed tickets. reverse_route collects vertices only after their outgoing tickets are exhausted.

Invariant: Each ticket edge is consumed exactly once; route is built backward.

Decision to rehearse: Use Hierholzer-style postorder edge consumption.

Finish: Reverse the completed route

Transfer exercise

Remove the existence guarantee and validate the final consecutive airport pairs against the original ticket multiset.

200. Dependency Impact Counts

Interview | Module 20

Problem

For each vertex of a DAG, return the number of distinct downstream vertices reachable by one or more directed edges. An edge u->v means v depends on u. Exclude the vertex itself and count shared descendants once. Raise ValueError for a cycle.

Examples

Example 1. The shared descendant is counted once at vertex 0.

```
dependency_counts([[1,2],[3],[3],[]])
=> [3, 1, 1, 0]
```

Example 2. A chain accumulates all later vertices.

```
dependency_counts([[1],[2],[]])
=> [2, 1, 0]
```

Example 3. Isolated vertices have no downstream dependents.

```
dependency_counts([[],[]])
=> [0, 0]
```

Why this mechanism works

A node's downstream set consists of each direct successor plus that successor's downstream set. Process reverse topological order so every successor summary is complete before combining it. Counts alone are insufficient because two successors may share the same dependent.

State and boundaries

downstream[u] is the set of distinct descendants of u. Reverse order ensures every set read from a successor is final.

Protocol

1. Validate a topological order.
2. Allocate one empty set per vertex.
3. Process vertices in reverse order.
4. Add each successor and union its downstream set.
5. Return the set sizes in vertex order.

Visualization

```
0 -> 1 -> 3
 \-> 2 -> 3
downstream[0] = {1,2,3}
count is 3, not (1+1)+(1+1)=4
```

Python implementation

```
from kata_193 import topological_order

def dependency_counts(adj):
    order = topological_order(adj)
    if len(order) != len(adj):
        raise ValueError('graph must be acyclic')
    downstream = [set() for _ in adj]
    for u in reversed(order):
        for v in adj[u]:
            downstream[u].add(v)
            downstream[u].update(downstream[v])
    return [len(nodes) for nodes in downstream]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
dependency_counts([[1,2],[3],[3],[]])

L4 dependency_counts: order = topological_order(adj)
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]

L7 dependency_counts: downstream = [set() for _ in adj]
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; _=[1, 2]

L7 dependency_counts: downstream = [set() for _ in adj]
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; _=[3]

L7 dependency_counts: downstream = [set() for _ in adj]
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; _=[]

L7 dependency_counts: downstream = [set() for _ in adj]
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; downstream=[set(), set(), set(), set()]

L8 dependency_counts: for u in reversed(order):
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; downstream=[set(), set(), set(), set()]; u=3

L8 dependency_counts: for u in reversed(order):
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; downstream=[set(), set(), set(), set()]; u=2

L9 dependency_counts: for v in adj[u]:
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; downstream=[set(), set(), set(), set()]; u=2;
    v=3

... 9 intermediate state changes omitted ...

L12 dependency_counts: return [len(nodes) for nodes in downstream]
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; downstream=[{1, 2, 3}, {3}, {3}, set()]; u=0;
    v=2; nodes={1, 2, 3}

L12 dependency_counts: return [len(nodes) for nodes in downstream]
```

```
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; downstream=[{1, 2, 3}, {3}, {3}, set()]; u=0;
    v=2; nodes={3}

L12 dependency_counts: return [len(nodes) for nodes in downstream]
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; downstream=[{1, 2, 3}, {3}, {3}, set()]; u=0;
    v=2; nodes=set()

L12 dependency_counts: return [len(nodes) for nodes in downstream]
adj=[[1, 2], [3], [3], []]; order=[0, 1, 2, 3]; downstream=[{1, 2, 3}, {3}, {3}, set()]; u=0;
    v=2
```

Result: [3, 1, 1, 0]. The shared descendant is counted once at vertex 0.

```
dependency_counts([[1],[2],[]])

L4 dependency_counts: order = topological_order(adj)
adj=[[1], [2], []]; order=[0, 1, 2]

L7 dependency_counts: downstream = [set() for _ in adj]
adj=[[1], [2], []]; order=[0, 1, 2]; _=[1]

L7 dependency_counts: downstream = [set() for _ in adj]
adj=[[1], [2], []]; order=[0, 1, 2]; _=[2]

L7 dependency_counts: downstream = [set() for _ in adj]
adj=[[1], [2], []]; order=[0, 1, 2]; _=[]

L7 dependency_counts: downstream = [set() for _ in adj]
adj=[[1], [2], []]; order=[0, 1, 2]; downstream=[set(), set(), set()]

L8 dependency_counts: for u in reversed(order):
adj=[[1], [2], []]; order=[0, 1, 2]; downstream=[set(), set(), set()]; u=2

L8 dependency_counts: for u in reversed(order):
adj=[[1], [2], []]; order=[0, 1, 2]; downstream=[set(), set(), set()]; u=1

L9 dependency_counts: for v in adj[u]:
adj=[[1], [2], []]; order=[0, 1, 2]; downstream=[set(), set(), set()]; u=1; v=2

... 5 intermediate state changes omitted ...

L12 dependency_counts: return [len(nodes) for nodes in downstream]
adj=[[1], [2], []]; order=[0, 1, 2]; downstream=[{1, 2}, {2}, set()]; u=0; v=1; nodes={1, 2}

L12 dependency_counts: return [len(nodes) for nodes in downstream]
adj=[[1], [2], []]; order=[0, 1, 2]; downstream=[{1, 2}, {2}, set()]; u=0; v=1; nodes={2}

L12 dependency_counts: return [len(nodes) for nodes in downstream]
adj=[[1], [2], []]; order=[0, 1, 2]; downstream=[{1, 2}, {2}, set()]; u=0; v=1; nodes=set()

L12 dependency_counts: return [len(nodes) for nodes in downstream]
adj=[[1], [2], []]; order=[0, 1, 2]; downstream=[{1, 2}, {2}, set()]; u=0; v=1
```

Result: [2, 1, 0]. A chain accumulates all later vertices.

Why the solution is correct

Every inserted vertex is reachable through a direct successor. Every nonempty path begins with such a successor, whose completed set includes the rest of the path's endpoint. Set union removes duplicate routes to the same endpoint, giving exactly the distinct downstream vertices.

Time and space

O(V+E+E*V) worst-case time and O(V^2+E) auxiliary storage; O(V) returned counts.

Common mistakes

Adding successor counts overcounts shared descendants in a diamond. Processing forward topological order reads incomplete successor summaries.

Reasoning card

State: downstream[u] is the set of distinct descendants of u. Reverse order ensures every set read from a successor is final.

Invariant: A node is summarized only after all successors are complete.

Decision to rehearse: Process reverse topological order and combine successor reachability carefully.

Finish: Return the set sizes in vertex order

Transfer exercise

Return the downstream set itself for change-impact analysis, then discuss the cost of maintaining it after edge insertions.

Module 21
Union-Find, Weighted Graphs, and Minimum Spanning Trees

Maintain dynamic components and use priority frontiers for weighted shortest paths or minimum connection cost.

Connectivity, minimum connection cost and shortest travel cost are different questions. DSU summarizes which components are joined. Kruskal and Prim choose safe connection edges; Dijkstra prioritizes accumulated path length. Edge-budgeted flights need another state axis, while zero/one weights permit a cheaper deque frontier.

The ten-step progression

201. Implement Disjoint-Set Union

202. Count Components with Union-Find

203. Redundant Connection

204. Accounts Merge

205. Kruskal Minimum Spanning Tree

206. Prim Minimum Spanning Tree

207. Dijkstra Shortest Paths

208. Network Delay Time

209. Cheapest Flight Within K Stops

210. Zero-One BFS

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

201. Implement Disjoint-Set Union

Foundation | Module 21

Problem

Implement DSU for n elements numbered 0 through n-1, supporting find(x) and union(a,b). find returns a representative index; union returns true only when two previously different components merge. Maintain count and component sizes at roots. Use path compression and union by size; ties keep a's root.

Examples

Example 1. Only the first three unions merge distinct components.

```
run_operations(DSU,[("union",0,1),("union",2,3),("union",1,3),("find",3),("union",0,3)
    ],4)
=> [True, True, True, 0, False]
```

Example 2. A singleton is already connected to itself.

```
run_operations(DSU,[("find",0),("union",0,0)],1)
=> [0, False]
```

Example 3. One successful union reduces the component count once.

```
dsu=DSU(3); dsu.union(0,2)
dsu.count
=> 2
```

Why this mechanism works

Represent each component as a parent-pointer tree. A root points to itself. find shortens future routes by redirecting each visited node toward its grandparent. union first obtains roots, then attaches the smaller tree beneath the larger so depth grows slowly even before compression.

State and boundaries

parent links remain within a component and lead to its root. size is meaningful at roots; count equals the number of roots.

Protocol

1. Initialize each element as a singleton.
2. During find repeatedly halve the parent path.
3. During union find both roots.
4. Return false if equal.

5. Attach the smaller root, add its size, decrement count and return true.

Visualization

```
0<-1   2<-3
union(1,3): roots 0 and 2 merge
0<-2<-3; a later find(3) shortens its route to 0
```

Python implementation

```
from structures import run_operations

class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.size = [1] * n
        self.count = n

    def find(self, x):
        while x != self.parent[x]:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def union(self, a, b):
        a, b = self.find(a), self.find(b)
        if a == b:
            return False
        if self.size[a] < self.size[b]:
            a, b = b, a
        self.parent[b] = a
        self.size[a] += self.size[b]
        self.count -= 1
        return True
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(DSU,[("union",0,1),("union",2,3),("union",1,3),("find",3),("union",0,3)
    ],4)

L5 __init__: self.parent = list(range(n))
self=DSU{'parent':[..., ..., ..., ...]}; n=4

L6 __init__: self.size = [1] * n
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...]}; n=4

L7 __init__: self.count = n
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':4}; n=4

L10 find: while x != self.parent[x]:
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':4}; x=0

L10 find: while x != self.parent[x]:
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':4}; x=1
```

```
L16 union: a, b = self.find(a), self.find(b)
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':4}; a=0; b=1

L23 union: self.count -= 1
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':3}; a=0; b=1

L10 find: while x != self.parent[x]:
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':3}; x=2

... 11 intermediate state changes omitted ...

L10 find: while x != self.parent[x]:
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':1}; x=0

L10 find: while x != self.parent[x]:
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':1}; x=3

L12 find: x = self.parent[x]
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':1}; x=0

L16 union: a, b = self.find(a), self.find(b)
self=DSU{'parent':[..., ..., ..., ...], 'size':[..., ..., ..., ...], 'count':1}; a=0; b=0
```

Result: [True, True, True, 0, False]. Only the first three unions merge distinct components.

```
run_operations(DSU,[("find",0),("union",0,0)],1)

L5 __init__: self.parent = list(range(n))
self=DSU{'parent':[...]}; n=1

L6 __init__: self.size = [1] * n
self=DSU{'parent':[...], 'size':[...]}; n=1

L7 __init__: self.count = n
self=DSU{'parent':[...], 'size':[...], 'count':1}; n=1

L10 find: while x != self.parent[x]:
self=DSU{'parent':[...], 'size':[...], 'count':1}; x=0

L10 find: while x != self.parent[x]:
self=DSU{'parent':[...], 'size':[...], 'count':1}; x=0

L10 find: while x != self.parent[x]:
self=DSU{'parent':[...], 'size':[...], 'count':1}; x=0

L16 union: a, b = self.find(a), self.find(b)
self=DSU{'parent':[...], 'size':[...], 'count':1}; a=0; b=0
```

Result: [0, False]. A singleton is already connected to itself.

Why the solution is correct

Path shortening preserves the same reachable root and therefore component identity. Linking two distinct roots merges exactly their two sets without affecting others. Root sizes add because the sets were disjoint, and one fewer root justifies decrementing count. Size-based attachment and compression preserve a forest.

Time and space

O(n) initialization and O(alpha(n)) amortized time per operation over a sequence, with O(n) storage. alpha is the inverse Ackermann function; an individual operation is not guaranteed strict O(1).

Common mistakes

Attaching arbitrary nonroot nodes can disconnect represented relationships. Decrementing count after an already-connected union undercounts components.

Reasoning card

State: parent links remain within a component and lead to its root. size is meaningful at roots; count equals the number of roots.

Invariant: Nodes have the same representative exactly when connected.

Decision to rehearse: Represent each component by a root.

Finish: Attach the smaller root, add its size, decrement count and return true

Transfer exercise

Support rollback by recording parent/size changes. Explain why ordinary path compression complicates undo.

202. Count Components with Union-Find

Foundation | Module 21

Problem

Return the number of connected components after adding an undirected edge list to n initially isolated vertices. Vertices are valid indices. Repeated edges and self-loops do not create additional merges. n=0 with no edges returns zero.

Examples

Example 1. The reversed repeated connection does not reduce the count again.

```
dsu_components(4,[(0,1),(1,0),(2,3)])
=> 2
```

Example 2. Every vertex remains isolated.

```
dsu_components(3,[])
=> 3
```

Example 3. There are no initial components.

```
dsu_components(0,[])
=> 0
```

Why this mechanism works

Unlike a traversal, DSU can process edges incrementally without storing adjacency. Start with n components. An edge reduces this number precisely when its endpoints currently have different representatives. Reuse the tested union contract from kata 201.

State and boundaries

dsu.count is the component count induced by the processed edge prefix.

Protocol

1. Create DSU(n).

2. Union the endpoints of every edge.

3. Return the maintained count.

Visualization

```
n=4: count 4
edge 0-1 -> 3
edge 1-0 -> still 3
edge 2-3 -> 2
```

Python implementation

```
from kata_201 import DSU

def dsu_components(n, edges):
    dsu = DSU(n)
    for a, b in edges:
        dsu.union(a, b)
    return dsu.count
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
dsu_components(4,[(0,1),(1,0),(2,3)])

L4 dsu_components: dsu = DSU(n)
n=4; edges=[(0, 1), (1, 0), (2, 3)]; dsu=DSU{'parent':[..., ..., ..., ...], 'size':[..., ...,
    ..., ...], 'count':4}

L5 dsu_components: for a, b in edges:
n=4; edges=[(0, 1), (1, 0), (2, 3)]; dsu=DSU{'parent':[..., ..., ..., ...], 'size':[..., ...,
    ..., ...], 'count':4}; a=0; b=1

L6 dsu_components: dsu.union(a, b)
n=4; edges=[(0, 1), (1, 0), (2, 3)]; dsu=DSU{'parent':[..., ..., ..., ...], 'size':[..., ...,
    ..., ...], 'count':3}; a=0; b=1

L5 dsu_components: for a, b in edges:
n=4; edges=[(0, 1), (1, 0), (2, 3)]; dsu=DSU{'parent':[..., ..., ..., ...], 'size':[..., ...,
    ..., ...], 'count':3}; a=1; b=0

L5 dsu_components: for a, b in edges:
n=4; edges=[(0, 1), (1, 0), (2, 3)]; dsu=DSU{'parent':[..., ..., ..., ...], 'size':[..., ...,
    ..., ...], 'count':3}; a=2; b=3

L6 dsu_components: dsu.union(a, b)
n=4; edges=[(0, 1), (1, 0), (2, 3)]; dsu=DSU{'parent':[..., ..., ..., ...], 'size':[..., ...,
    ..., ...], 'count':2}; a=2; b=3
```

Result: 2. The reversed repeated connection does not reduce the count again.

```
dsu_components(3,[])

L4 dsu_components: dsu = DSU(n)
n=3; edges=[]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...], 'count':3}
```

Result: 3. Every vertex remains isolated.

Why the solution is correct

The empty edge prefix has n singleton components. An edge within one component leaves connectivity unchanged; an edge across two components merges exactly those two. DSU applies those cases correctly, so induction over the edge list proves the final count.

Time and space

O(n+m*alpha(n)) amortized time and O(n) auxiliary storage for m edges; no adjacency lists are required.

Common mistakes

Subtracting the number of edges from n fails when edges are redundant. Directed reachability cannot be summarized by this undirected union operation.

Reasoning card

State: dsu.count is the component count induced by the processed edge prefix.

Invariant: components changes only when two distinct roots unite.

Decision to rehearse: Union endpoints of every undirected edge.

Finish: Return the maintained count

Transfer exercise

Emit the count after each incoming edge to answer an online connectivity-progress query.

203. Redundant Connection

Foundation | Module 21

Problem

Given n vertices and an ordered undirected edge list, return the first edge whose addition connects vertices already connected by earlier edges. Return its original endpoint tuple, or None if every edge merges components. A self-loop is immediately redundant; repeated edges can be redundant.

Examples

Example 1. The third edge closes the first cycle.

```
first_redundant_edge(3,[(0,1),(1,2),(0,2)])
=> (0, 2)
```

Example 2. The only edge is necessary for connectivity.

```
first_redundant_edge(2,[(0,1)])
=> None
```

Example 3. A self-loop connects an element to itself.

```
first_redundant_edge(1,[(0,0)])
=> (0, 0)
```

Why this mechanism works

An edge creates an undirected cycle exactly when a path already joins its endpoints. DSU tests this condition without reconstructing the path. Scan in the supplied order and return the first failed union, rather than sorting edges or choosing the last failure.

State and boundaries

DSU represents connectivity from the edge prefix before the current edge. A false union result is the redundancy witness.

Protocol

1. Create singleton components.
2. Read edges in input order.
3. Attempt each union.
4. Return the current pair on the first false result.
5. Return None if no union fails.

Visualization

```
0-1 accepted; 1-2 accepted
0-2 finds both endpoints in the same component
return (0,2) before considering later edges
```

Python implementation

```
from kata_201 import DSU

def first_redundant_edge(n, edges):
    dsu = DSU(n)
    for a, b in edges:
        if not dsu.union(a, b):
            return (a, b)
    return None
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
first_redundant_edge(3,[(0,1),(1,2),(0,2)])

L4 first_redundant_edge: dsu = DSU(n)
n=3; edges=[(0, 1), (1, 2), (0, 2)]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
     'count':3}

L5 first_redundant_edge: for a, b in edges:
n=3; edges=[(0, 1), (1, 2), (0, 2)]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
     'count':3}; a=0; b=1

L6 first_redundant_edge: if not dsu.union(a, b):
n=3; edges=[(0, 1), (1, 2), (0, 2)]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
     'count':2}; a=0; b=1

L5 first_redundant_edge: for a, b in edges:
n=3; edges=[(0, 1), (1, 2), (0, 2)]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
     'count':2}; a=1; b=2

L6 first_redundant_edge: if not dsu.union(a, b):
n=3; edges=[(0, 1), (1, 2), (0, 2)]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
     'count':1}; a=1; b=2

L5 first_redundant_edge: for a, b in edges:
n=3; edges=[(0, 1), (1, 2), (0, 2)]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
     'count':1}; a=0; b=2
```

Result: (0, 2). The third edge closes the first cycle.

```
first_redundant_edge(2,[(0,1)])

L4 first_redundant_edge: dsu = DSU(n)
n=2; edges=[(0, 1)]; dsu=DSU{'parent':[..., ...], 'size':[..., ...], 'count':2}

L5 first_redundant_edge: for a, b in edges:
n=2; edges=[(0, 1)]; dsu=DSU{'parent':[..., ...], 'size':[..., ...], 'count':2}; a=0; b=1
```

```
L6 first_redundant_edge: if not dsu.union(a, b):
n=2; edges=[(0, 1)]; dsu=DSU{'parent':[..., ...], 'size':[..., ...], 'count':1}; a=0; b=1
```

Result: None. The only edge is necessary for connectivity.

Why the solution is correct

If endpoints are already connected, their existing path plus the new edge forms a cycle, so that edge is redundant. Otherwise it merges components and creates no cycle. Because the scan stops at the first such case, the returned edge is the earliest required witness.

Time and space

O(n+m*alpha(n)) amortized worst-case time and O(n) storage, with early exit possible.

Common mistakes

Returning the final failed union violates this first-edge contract. Sorting the input changes which edge is first.

Reasoning card

State: DSU represents connectivity from the edge prefix before the current edge. A false union result is the redundancy witness.

Invariant: A same-root edge closes an undirected cycle.

Decision to rehearse: Query roots before union.

Finish: Return None if no union fails

Transfer exercise

Return the actual cycle by retaining accepted forest edges and finding the existing endpoint path when union fails.

204. Accounts Merge

Foundation | Module 21

Problem

Merge accounts represented as [name,email1,...], with at least one email per account. Accounts sharing any email belong to the same person, transitively; connected accounts are guaranteed to have the same name. Equal names alone do not imply identity. Return each group's sorted unique emails after its name, and sort the resulting records lexicographically.

Examples

Example 1. Shared email b merges only the first two accounts.

```
merge_accounts([["Ana","a","b"],["Ana","b","c"],["Ana","z"]])
=> [['Ana', 'a', 'b', 'c'], ['Ana', 'z']]
```

Example 2. No accounts produce no groups.

```
merge_accounts([])
=> []
```

Example 3. Repeated email occurrences appear once in output.

```
merge_accounts([["Bo","x","x"]])
=> [['Bo', 'x']]
```

Why this mechanism works

Emails are identity evidence and account indices are convenient DSU elements. Remember the first account that owns each email; every later occurrence unions its account with that owner. After all unions, group each distinct email by its owner's final representative.

State and boundaries

owner maps each email to an account containing it. DSU components represent all shared-email links processed so far.

Protocol

1. Create one DSU element per account.

2. Union accounts sharing an email.

3. Find the final root for every distinct email's owner.

4. Group emails by root.

5. Prefix the consistent name, sort emails, and sort records.

Visualization

```
[Ana,a,b] + [Ana,b,c] -> [Ana,a,b,c]
[Ana,z] stays separate: same name is not shared-email evidence
```

Python implementation

```
from collections import defaultdict
from kata_201 import DSU

def merge_accounts(accounts):
    dsu, owner = DSU(len(accounts)), {}
    for i, account in enumerate(accounts):
        for email in account[1:]:
            if email in owner:
                dsu.union(i, owner[email])
            else:
                owner[email] = i
    groups = defaultdict(list)
    for email, i in owner.items():
        groups[dsu.find(i)].append(email)
    return sorted([accounts[root][0]] + sorted(emails)
                  for root, emails in groups.items())
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
merge_accounts([["Ana","a","b"],["Ana","b","c"],["Ana","z"]])

L5 merge_accounts: dsu, owner = DSU(len(accounts)), {}
accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; dsu=DSU{'parent':[..., ...,
    ...], 'size':[..., ..., ...], 'count':3}; owner={}

L6 merge_accounts: for i, account in enumerate(accounts):
accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; dsu=DSU{'parent':[..., ...,
    ...], 'size':[..., ..., ...], 'count':3}; owner={}; i=0; account=['Ana', 'a', 'b']

L7 merge_accounts: for email in account[1:]:
accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; dsu=DSU{'parent':[..., ...,
    ...], 'size':[..., ..., ...], 'count':3}; owner={}; i=0; account=['Ana', 'a', 'b'];
    email='a'

L11 merge_accounts: owner[email] = i
accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; dsu=DSU{'parent':[..., ...,
    ...], 'size':[..., ..., ...], 'count':3}; owner={'a':0}; i=0; account=['Ana', 'a', 'b'];
    email='a'

L7 merge_accounts: for email in account[1:]:
accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; dsu=DSU{'parent':[..., ...,
    ...], 'size':[..., ..., ...], 'count':3}; owner={'a':0}; i=0; account=['Ana', 'a', 'b'];
    email='b'

L11 merge_accounts: owner[email] = i
accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; dsu=DSU{'parent':[..., ...,
    ...], 'size':[..., ..., ...], 'count':3}; owner={'a':0, 'b':0}; i=0; account=['Ana', 'a',
```

```
    'b']; email='b'

L6 merge_accounts: for i, account in enumerate(accounts):
accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; dsu=DSU{'parent':[..., ...,
    ...], 'size':[..., ..., ...], 'count':3}; owner={'a':0, 'b':0}; i=1; account=['Ana', 'b',
    'c']; email='b'

L9 merge_accounts: dsu.union(i, owner[email])
accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; dsu=DSU{'parent':[..., ...,
    ...], 'size':[..., ..., ...], 'count':2}; owner={'a':0, 'b':0}; i=1; account=['Ana', 'b',
    'c']; email='b'

... 14 intermediate state changes omitted ...

L15 <genexpr>: return sorted([accounts[root][0]] + sorted(emails)
.0=dict_itemiterator; accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]

L16 <genexpr>: for root, emails in groups.items())
.0=dict_itemiterator; accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; root=1;
    emails=['a', 'b', 'c']

L16 <genexpr>: for root, emails in groups.items())
.0=dict_itemiterator; accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; root=2;
    emails=['z']

L16 <genexpr>: for root, emails in groups.items())
.0=dict_itemiterator; accounts=[['Ana', 'a', 'b'], ['Ana', 'b', 'c'], ['Ana', 'z']]; root=2;
    emails=['z']
```

Result: [['Ana', 'a', 'b', 'c'], ['Ana', 'z']]. Shared email b merges only the first two accounts.

```
merge_accounts([])

L5 merge_accounts: dsu, owner = DSU(len(accounts)), {}
accounts=[]; dsu=DSU{'parent':[], 'size':[], 'count':0}; owner={}

L12 merge_accounts: groups = defaultdict(list)
accounts=[]; dsu=DSU{'parent':[], 'size':[], 'count':0}; owner={}; groups={}

L15 <genexpr>: return sorted([accounts[root][0]] + sorted(emails)
.0=dict_itemiterator; accounts=[]
```

Result: []. No accounts produce no groups.

Why the solution is correct

Every union is justified by a shared email, and every shared-email occurrence creates the necessary union. DSU transitivity therefore matches exactly the identity relation. Grouping distinct email keys by final roots includes every email once in its correct merged account.

Time and space

O(A+T*alpha(A)+T log T) coarse time for A accounts and T email occurrences, including output sorting; O(A+T) auxiliary/output storage. String comparison costs depend on lengths.

Common mistakes

Merging by name combines unrelated people. Grouping before all unions are complete can split accounts whose connection appears later.

Reasoning card

State: owner maps each email to an account containing it. DSU components represent all shared-email links processed so far.

Invariant: All emails in one root component belong to one merged account.

Decision to rehearse: Union account identities through shared keys, then group by root.

Finish: Prefix the consistent name, sort emails, and sort records

Transfer exercise

Remove the consistent-name guarantee and define how conflicting names should be surfaced instead of silently choosing one.

205. Kruskal Minimum Spanning Tree

Bridge | Module 21

Problem

Find a minimum spanning tree of an undirected weighted graph with n vertices and edges (u,v,weight). Negative weights, parallel edges and loops are allowed. Return (total_weight,selected_edges), or None if disconnected. Sort candidates by (weight,u,v,input_index) for deterministic ties. For n=0 return (0,[]).

Examples

Example 1. The two cheapest noncycling edges span the graph.

```
kruskal_mst(3,[(0,2,5),(1,2,2),(0,1,1)])
=> (3, [(0, 1, 1), (1, 2, 2)])
```

Example 2. Vertex 2 cannot be connected.

```
kruskal_mst(3,[(0,1,1)])
=> None
```

Example 3. The empty graph uses the explicit empty result.

```
kruskal_mst(0,[])
=> (0, [])
```

Why this mechanism works

Kruskal considers the cheapest remaining edge and accepts it only when it joins different components. DSU makes that cycle test efficient. The selected edges remain a forest, growing toward a spanning tree. Sorting determines which equal-cost optimum is returned.

State and boundaries

selected is an acyclic forest. DSU components are its connected components; total sums exactly its selected edge weights.

Protocol

1. Sort indexed edges by the specified key.

2. Start singleton components.

3. Accept an edge only when union succeeds.

4. Accumulate its weight and original tuple.

5. Stop after n-1 selections.

6. Return None if too few edges can be selected.

Visualization

```
0--1 weight 1; 1--2 weight 2; 0--2 weight 5
accept 1, accept 2 -> total 3
the heavier triangle edge is unnecessary
```

Python implementation

```
from kata_201 import DSU

def kruskal_mst(n, edges):
    if n == 0:
        return (0, [])
    ordered = sorted(enumerate(edges), key=lambda item:
                     (item[1][2], item[1][0], item[1][1], item[0]))
    dsu, selected, total = DSU(n), [], 0
    for _, (u, v, weight) in ordered:
        if dsu.union(u, v):
            selected.append((u, v, weight))
            total += weight
            if len(selected) == n - 1:
                break
    return (total, selected) if len(selected) == n - 1 else None
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
kruskal_mst(3,[(0,2,5),(1,2,2),(0,1,1)])

L4 kruskal_mst: if n == 0:
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]

L7 <lambda>: (item[1][2], item[1][0], item[1][1], item[0]))
item=(0, (0, 2, 5))

L7 <lambda>: (item[1][2], item[1][0], item[1][1], item[0]))
item=(1, (1, 2, 2))

L7 <lambda>: (item[1][2], item[1][0], item[1][1], item[0]))
item=(2, (0, 1, 1))

L6 kruskal_mst: ordered = sorted(enumerate(edges), key=lambda item:
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]; ordered=[(2, (..., ..., ...)), (1, (..., ...,
    ...)), (0, (..., ..., ...))]

L8 kruskal_mst: dsu, selected, total = DSU(n), [], 0
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]; ordered=[(2, (..., ..., ...)), (1, (..., ...,
    ...)), (0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
    'count':3}; selected=[]; total=0

L9 kruskal_mst: for _, (u, v, weight) in ordered:
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]; ordered=[(2, (..., ..., ...)), (1, (..., ...,
    ...)), (0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
```

```
    'count':3}; selected=[]; total=0; _=2; u=0; v=1; weight=1

L10 kruskal_mst: if dsu.union(u, v):
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]; ordered=[(2, (..., ..., ...)), (1, (..., ...,
    ...)), (0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
    'count':2}; selected=[]; total=0; _=2; u=0; v=1; weight=1

... 2 intermediate state changes omitted ...

L9 kruskal_mst: for _, (u, v, weight) in ordered:
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]; ordered=[(2, (..., ..., ...)), (1, (..., ...,
    ...)), (0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
    'count':2}; selected=[(0, 1, 1)]; total=1; _=1; u=1; v=2; weight=2

L10 kruskal_mst: if dsu.union(u, v):
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]; ordered=[(2, (..., ..., ...)), (1, (..., ...,
    ...)), (0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
    'count':1}; selected=[(0, 1, 1)]; total=1; _=1; u=1; v=2; weight=2

L11 kruskal_mst: selected.append((u, v, weight))
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]; ordered=[(2, (..., ..., ...)), (1, (..., ...,
    ...)), (0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
    'count':1}; selected=[(0, 1, 1), (1, 2, 2)]; total=1; _=1; u=1; v=2; weight=2

L12 kruskal_mst: total += weight
n=3; edges=[(0, 2, 5), (1, 2, 2), (0, 1, 1)]; ordered=[(2, (..., ..., ...)), (1, (..., ...,
    ...)), (0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...], 'size':[..., ..., ...],
    'count':1}; selected=[(0, 1, 1), (1, 2, 2)]; total=3; _=1; u=1; v=2; weight=2
```

Result: (3, [(0, 1, 1), (1, 2, 2)]). The two cheapest noncycling edges span the graph.

`kruskal_mst(3,[(0,1,1)])`

```
L4 kruskal_mst: if n == 0:
n=3; edges=[(0, 1, 1)]

L7 <lambda>: (item[1][2], item[1][0], item[1][1], item[0]))
item=(0, (0, 1, 1))

L6 kruskal_mst: ordered = sorted(enumerate(edges), key=lambda item:
n=3; edges=[(0, 1, 1)]; ordered=[(0, (..., ..., ...))]

L8 kruskal_mst: dsu, selected, total = DSU(n), [], 0
n=3; edges=[(0, 1, 1)]; ordered=[(0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...],
    'size':[..., ..., ...], 'count':3}; selected=[]; total=0

L9 kruskal_mst: for _, (u, v, weight) in ordered:
n=3; edges=[(0, 1, 1)]; ordered=[(0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...],
    'size':[..., ..., ...], 'count':3}; selected=[]; total=0; _=0; u=0; v=1; weight=1

L10 kruskal_mst: if dsu.union(u, v):
n=3; edges=[(0, 1, 1)]; ordered=[(0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...],
    'size':[..., ..., ...], 'count':2}; selected=[]; total=0; _=0; u=0; v=1; weight=1

L11 kruskal_mst: selected.append((u, v, weight))
n=3; edges=[(0, 1, 1)]; ordered=[(0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...],
    'size':[..., ..., ...], 'count':2}; selected=[(0, 1, 1)]; total=0; _=0; u=0; v=1; weight=1

L12 kruskal_mst: total += weight
n=3; edges=[(0, 1, 1)]; ordered=[(0, (..., ..., ...))]; dsu=DSU{'parent':[..., ..., ...],
    'size':[..., ..., ...], 'count':2}; selected=[(0, 1, 1)]; total=1; _=0; u=0; v=1; weight=1
```

Result: None. Vertex 2 cannot be connected.

Why the solution is correct

For any accepted edge, its endpoints lie in different current components. It is a minimum-weight available edge crossing an appropriate component cut; the cut exchange argument shows it can belong to an optimum spanning tree extending the current forest. Repeating safe choices yields an MST when all vertices connect.

Time and space

O(E log E+V+E*alpha(V)) time and O(V+E) storage including sorting and output.

Common mistakes

Choosing the globally cheapest n-1 edges without cycle checks can leave vertices disconnected. Negative weights do not invalidate MST algorithms.

Reasoning card

State: selected is an acyclic forest. DSU components are its connected components; total sums exactly its selected edge weights.

Invariant: Accepted edges form a minimum-cost forest without cycles.

Decision to rehearse: Sort edges and accept only those joining distinct components.

Finish: Return None if too few edges can be selected

Transfer exercise

Prove the cut exchange step by adding the chosen edge to an optimum tree and removing a no-cheaper edge from the resulting cycle.

206. Prim Minimum Spanning Tree

Bridge | Module 21

Problem

Return the total weight of an MST for an undirected weighted graph given as n and edges (u,v,weight), or None if disconnected. Negative weights and repeated edges are allowed. Use Prim's algorithm starting at vertex zero; n=0 returns zero. Only total cost is required.

Examples

Example 1. The frontier updates after adding vertex 1.

```
prim_mst(3,[(0,1,1),(0,2,5),(1,2,2)])
=> 3
```

Example 2. No edge crosses from vertex 0 to vertex 1.

```
prim_mst(2,[])
=> None
```

Example 3. Negative edge weights are valid connection costs.

```
prim_mst(2,[(0,1,-3)])
=> -3
```

Why this mechanism works

Grow one connected tree rather than a forest. A min-heap stores edges leaving visited vertices; the cheapest edge reaching an unseen vertex safely extends the tree across its current cut. Edges whose destination is already visited are stale candidates and must be skipped.

State and boundaries

seen is the current tree's vertex set. Heap entries are candidate (edge_weight,destination) pairs from that set, plus the initial zero-cost root entry.

Protocol

1. Build undirected weighted adjacency.

2. Seed (0,0).

3. Pop the cheapest candidate.

4. Skip an already included destination.

5. Include the new vertex, add its edge cost, and push edges to unseen neighbors.

6. Return total only if every vertex was included.

Visualization

```
tree {0}; frontier edges weight 1 to 1, weight 5 to 2
take 1; new edge weight 2 to 2
take 2 -> total 3; stale weight-5 entry is ignored
```

Python implementation

```
import heapq

def prim_mst(n, edges):
    if n == 0:
        return 0
    adj = [[] for _ in range(n)]
    for u, v, weight in edges:
        adj[u].append((v, weight))
        adj[v].append((u, weight))
    heap, seen, total = [(0, 0)], set(), 0
    while heap:
        weight, u = heapq.heappop(heap)
        if u in seen:
            continue
        seen.add(u)
        total += weight
        for v, cost in adj[u]:
            if v not in seen:
                heapq.heappush(heap, (cost, v))
    return total if len(seen) == n else None
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
prim_mst(3,[(0,1,1),(0,2,5),(1,2,2)])

L4 prim_mst: if n == 0:
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]

L6 prim_mst: adj = [[] for _ in range(n)]
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; _=0

L6 prim_mst: adj = [[] for _ in range(n)]
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; _=1

L6 prim_mst: adj = [[] for _ in range(n)]
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; _=2

L6 prim_mst: adj = [[] for _ in range(n)]
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; adj=[[], [], []]

L7 prim_mst: for u, v, weight in edges:
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; adj=[[], [], []]; u=0; v=1; weight=1

L8 prim_mst: adj[u].append((v, weight))
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; adj=[[(..., ...)], [], []]; u=0; v=1; weight=1
```

```
L9 prim_mst: adj[v].append((u, weight))
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; adj=[[(..., ...)], [(..., ...)], []]; u=0; v=1;
    weight=1

... 21 intermediate state changes omitted ...

L16 prim_mst: total += weight
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; adj=[[(..., ...), (..., ...)], [(..., ...),
    (..., ...)], [(..., ...), (..., ...)]]; u=2; v=2; weight=2; heap=[(5, 2)]; seen={0, 1, 2};
     total=3; cost=2

L17 prim_mst: for v, cost in adj[u]:
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; adj=[[(..., ...), (..., ...)], [(..., ...),
    (..., ...)], [(..., ...), (..., ...)]]; u=2; v=0; weight=2; heap=[(5, 2)]; seen={0, 1, 2};
     total=3; cost=5

L17 prim_mst: for v, cost in adj[u]:
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; adj=[[(..., ...), (..., ...)], [(..., ...),
    (..., ...)], [(..., ...), (..., ...)]]; u=2; v=1; weight=2; heap=[(5, 2)]; seen={0, 1, 2};
     total=3; cost=2

L12 prim_mst: weight, u = heapq.heappop(heap)
n=3; edges=[(0, 1, 1), (0, 2, 5), (1, 2, 2)]; adj=[[(..., ...), (..., ...)], [(..., ...),
    (..., ...)], [(..., ...), (..., ...)]]; u=2; v=1; weight=5; heap=[]; seen={0, 1, 2};
    total=3; cost=2
```

Result: 3. The frontier updates after adding vertex 1.

```
prim_mst(2,[])

L4 prim_mst: if n == 0:
n=2; edges=[]

L6 prim_mst: adj = [[] for _ in range(n)]
n=2; edges=[]; _=0

L6 prim_mst: adj = [[] for _ in range(n)]
n=2; edges=[]; _=1

L6 prim_mst: adj = [[] for _ in range(n)]
n=2; edges=[]; adj=[[], []]

L10 prim_mst: heap, seen, total = [(0, 0)], set(), 0
n=2; edges=[]; adj=[[], []]; heap=[(0, 0)]; seen=set(); total=0

L12 prim_mst: weight, u = heapq.heappop(heap)
n=2; edges=[]; adj=[[], []]; heap=[]; seen=set(); total=0; u=0; weight=0

L15 prim_mst: seen.add(u)
n=2; edges=[]; adj=[[], []]; heap=[]; seen={0}; total=0; u=0; weight=0
```

Result: None. No edge crosses from vertex 0 to vertex 1.

Why the solution is correct

After the root, each accepted entry is a minimum-weight edge crossing from the current tree to an unseen vertex: every such edge has been offered to the heap. The MST cut property makes this extension safe. If the heap empties before all vertices are included, no edge connects the remaining components.

Time and space

O(V+E log(E+2)) time and O(V+E) space for this lazy edge-heap implementation, including adjacency.

Common mistakes

Adding a stale edge's weight before checking seen double-counts vertices. Heap priorities are individual edge weights, unlike Dijkstra's accumulated path distances.

Reasoning card

State: seen is the current tree's vertex set. Heap entries are candidate (edge_weight,destination) pairs from that set, plus the initial zero-cost root entry.

Invariant: Visited nodes are connected by the chosen tree edges.

Decision to rehearse: Heap exposes the cheapest edge crossing into an unvisited node.

Finish: Return total only if every vertex was included

Transfer exercise

Record parent edges to return the actual tree, then compare total cost with Kruskal on the same graph.

207. Dijkstra Shortest Paths

Bridge | Module 21

Problem

Compute shortest distances from source in a directed weighted adjacency list, where each entry is (neighbor,nonnegative_weight). Vertices and source are valid indices; parallel edges and zero weights are allowed. Return a list with None for unreachable vertices. Negative weights are outside the contract.

Examples

Example 1. A later proposal improves vertex 1 while vertex 3 remains unreachable.

```
dijkstra([[(1,10),(2,1)],[],[(1,2)],[]],0)
=> [0, 3, 1, None]
```

Example 2. Zero-cost edges are permitted.

```
dijkstra([[(1,0)],[]],0)
=> [0, 0]
```

Example 3. The source has a zero-edge route to itself.

```
dijkstra([[]],0)
=> [0]
```

Why this mechanism works

Use a min-heap of proposed total path distances. Relaxing an edge offers a better route to its destination. Since Python's heap has no decrease-key operation, push the new proposal and leave old entries to be skipped later. Nonnegative edges ensure the smallest current proposal cannot be improved through a farther unsettled vertex.

State and boundaries

dist[v] is the best route length discovered so far. A heap entry is actionable only when its distance still equals dist[v].

Protocol

1. Initialize infinity everywhere except source zero.
2. Pop the minimum proposal.
3. Skip a stale distance.
4. Relax each outgoing edge with current distance+weight.
5. Push every strict improvement.

6. Convert remaining infinities to None.

Visualization

```
0->1 cost 10; 0->2 cost 1; 2->1 cost 2
proposals for 1: 10, then 3
pop 3 and finalize; later 10 is stale
```

Python implementation

```
import heapq

def dijkstra(adj, source):
    dist = [float('inf')] * len(adj)
    dist[source] = 0
    heap = [(0, source)]
    while heap:
        distance, u = heapq.heappop(heap)
        if distance != dist[u]:
            continue
        for v, weight in adj[u]:
            candidate = distance + weight
            if candidate < dist[v]:
                dist[v] = candidate
                heapq.heappush(heap, (candidate, v))
    return [None if value == float('inf') else value for value in dist]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
dijkstra([[(1,10),(2,1)],[],[(1,2)],[]],0)

L4 dijkstra: dist = [float('inf')] * len(adj)
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[inf, inf, inf, inf]

L5 dijkstra: dist[source] = 0
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf]

L6 dijkstra: heap = [(0, source)]
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf];
    heap=[(0, 0)]

L8 dijkstra: distance, u = heapq.heappop(heap)
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf];
    heap=[]; distance=0; u=0

L11 dijkstra: for v, weight in adj[u]:
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf];
    heap=[]; distance=0; u=0; v=1; weight=10

L12 dijkstra: candidate = distance + weight
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf];
    heap=[]; distance=0; u=0; v=1; weight=10; candidate=10

L14 dijkstra: dist[v] = candidate
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 10, inf, inf];
    heap=[]; distance=0; u=0; v=1; weight=10; candidate=10
```

```
L15 dijkstra: heapq.heappush(heap, (candidate, v))
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 10, inf, inf];
    heap=[(10, 1)]; distance=0; u=0; v=1; weight=10; candidate=10

... 12 intermediate state changes omitted ...

L16 dijkstra: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 3, 1, inf]; heap=[];
    distance=10; u=1; v=1; weight=2; candidate=3; value=3

L16 dijkstra: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 3, 1, inf]; heap=[];
    distance=10; u=1; v=1; weight=2; candidate=3; value=1

L16 dijkstra: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 3, 1, inf]; heap=[];
    distance=10; u=1; v=1; weight=2; candidate=3; value=inf

L16 dijkstra: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 3, 1, inf]; heap=[];
    distance=10; u=1; v=1; weight=2; candidate=3
```

Result: [0, 3, 1, None]. A later proposal improves vertex 1 while vertex 3 remains unreachable.

```
dijkstra([[(1,0)],[]],0)

L4 dijkstra: dist = [float('inf')] * len(adj)
adj=[[(..., ...)], []]; source=0; dist=[inf, inf]

L5 dijkstra: dist[source] = 0
adj=[[(..., ...)], []]; source=0; dist=[0, inf]

L6 dijkstra: heap = [(0, source)]
adj=[[(..., ...)], []]; source=0; dist=[0, inf]; heap=[(0, 0)]

L8 dijkstra: distance, u = heapq.heappop(heap)
adj=[[(..., ...)], []]; source=0; dist=[0, inf]; heap=[]; distance=0; u=0

L11 dijkstra: for v, weight in adj[u]:
adj=[[(..., ...)], []]; source=0; dist=[0, inf]; heap=[]; distance=0; u=0; v=1; weight=0

L12 dijkstra: candidate = distance + weight
adj=[[(..., ...)], []]; source=0; dist=[0, inf]; heap=[]; distance=0; u=0; v=1; weight=0;
    candidate=0

L14 dijkstra: dist[v] = candidate
adj=[[(..., ...)], []]; source=0; dist=[0, 0]; heap=[]; distance=0; u=0; v=1; weight=0;
    candidate=0

L15 dijkstra: heapq.heappush(heap, (candidate, v))
adj=[[(..., ...)], []]; source=0; dist=[0, 0]; heap=[(0, 1)]; distance=0; u=0; v=1; weight=0;
    candidate=0

L8 dijkstra: distance, u = heapq.heappop(heap)
adj=[[(..., ...)], []]; source=0; dist=[0, 0]; heap=[]; distance=0; u=1; v=1; weight=0;
    candidate=0

L16 dijkstra: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...)], []]; source=0; dist=[0, 0]; heap=[]; distance=0; u=1; v=1; weight=0;
    candidate=0; value=0
```

```
L16 dijkstra: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...)], []]; source=0; dist=[0, 0]; heap=[]; distance=0; u=1; v=1; weight=0;
    candidate=0
```

Result: [0, 0]. Zero-cost edges are permitted.

Why the solution is correct

Every finite proposal is the length of an actual path. At a nonstale minimum pop, any competing route through an unsettled vertex must already be at least that large because remaining edge weights are nonnegative. Thus the popped distance is final. Relaxations then preserve all candidate shortest routes.

Time and space

O(V+E log(E+2)) time and O(V+E) auxiliary space for a lazy heap that can hold multiple proposals per vertex. With simple graphs this is commonly written O((V+E) log V).

Common mistakes

Using edge weight alone as the priority runs an MST-style rule, not shortest paths. A permanent visited mark at first enqueue is too early. Negative edges break the finalization proof.

Reasoning card

State: dist[v] is the best route length discovered so far. A heap entry is actionable only when its distance still equals dist[v].

Invariant: A popped non-stale distance is final.

Decision to rehearse: Heap exposes the unsettled node with smallest known distance.

Finish: Convert remaining infinities to None

Transfer exercise

Return predecessor pointers alongside distances, and explain how equal-cost ties affect the chosen path.

208. Network Delay Time

Interview | Module 21

Problem

Given n>=1 network nodes labeled 1 through n, directed edges (u,v,delay) with nonnegative delays, and a starting node, return the earliest time by which every node receives a signal. Return -1 if any node is unreachable. Signal propagation may branch simultaneously.

Examples

Example 1. The last arrival is at node 4 after two time units.

```
network_delay(4,[(2,1,1),(2,3,1),(3,4,1)],2)
=> 2
```

Example 2. Directed edges do not let node 2 send backward to node 1.

```
network_delay(2,[(1,2,1)],2)
=> -1
```

Example 3. The source already has the signal.

```
network_delay(1,[],1)
=> 0
```

Why this mechanism works

Each node receives the signal at its shortest travel time from the source. Parallel propagation means total completion time is the maximum of those shortest distances, not their sum. Translate one-based labels at the boundary and reuse Dijkstra's nonnegative-weight protocol.

State and boundaries

The zero-based adjacency models the same directed network. distances[v] is the earliest arrival time at node v+1.

Protocol

1. Convert each label to zero-based adjacency.

2. Run Dijkstra from start-1.

3. Return -1 if any distance is None.

4. Otherwise return the maximum distance.

Visualization

```
source 2 -> 1 in 1; source 2 -> 3 in 1; 3 -> 4 in 1
arrival times [1,0,1,2]
all reached at time max=2
```

Python implementation

```
from kata_207 import dijkstra

def network_delay(n, edges, start):
    adj = [[] for _ in range(n)]
    for u, v, delay in edges:
        adj[u - 1].append((v - 1, delay))
    distances = dijkstra(adj, start - 1)
    return -1 if any(value is None for value in distances) else max(distances)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
network_delay(4,[(2,1,1),(2,3,1),(3,4,1)],2)

L4 network_delay: adj = [[] for _ in range(n)]
n=4; edges=[(2, 1, 1), (2, 3, 1), (3, 4, 1)]; start=2; _=0

L4 network_delay: adj = [[] for _ in range(n)]
n=4; edges=[(2, 1, 1), (2, 3, 1), (3, 4, 1)]; start=2; _=1

L4 network_delay: adj = [[] for _ in range(n)]
n=4; edges=[(2, 1, 1), (2, 3, 1), (3, 4, 1)]; start=2; _=2

L4 network_delay: adj = [[] for _ in range(n)]
n=4; edges=[(2, 1, 1), (2, 3, 1), (3, 4, 1)]; start=2; _=3

L4 network_delay: adj = [[] for _ in range(n)]
n=4; edges=[(2, 1, 1), (2, 3, 1), (3, 4, 1)]; start=2; adj=[[], [], [], []]

L5 network_delay: for u, v, delay in edges:
n=4; edges=[(2, 1, 1), (2, 3, 1), (3, 4, 1)]; start=2; adj=[[], [], [], []]; u=2; v=1; delay=1

L6 network_delay: adj[u - 1].append((v - 1, delay))
n=4; edges=[(2, 1, 1), (2, 3, 1), (3, 4, 1)]; start=2; adj=[[], [(..., ...)], [], []]; u=2;
    v=1; delay=1

L5 network_delay: for u, v, delay in edges:
n=4; edges=[(2, 1, 1), (2, 3, 1), (3, 4, 1)]; start=2; adj=[[], [(..., ...)], [], []]; u=2;
    v=3; delay=1

... 5 intermediate state changes omitted ...

L8 <genexpr>: return -1 if any(value is None for value in distances) else max(distances)
.0=list_iterator; value=0

L8 <genexpr>: return -1 if any(value is None for value in distances) else max(distances)
.0=list_iterator; value=1

L8 <genexpr>: return -1 if any(value is None for value in distances) else max(distances)
.0=list_iterator; value=2
```

```
L8 <genexpr>: return -1 if any(value is None for value in distances) else max(distances)
.0=list_iterator; value=2
```

Result: 2. The last arrival is at node 4 after two time units.

```
network_delay(2,[(1,2,1)],2)

L4 network_delay: adj = [[] for _ in range(n)]
n=2; edges=[(1, 2, 1)]; start=2; _=0

L4 network_delay: adj = [[] for _ in range(n)]
n=2; edges=[(1, 2, 1)]; start=2; _=1

L4 network_delay: adj = [[] for _ in range(n)]
n=2; edges=[(1, 2, 1)]; start=2; adj=[[], []]

L5 network_delay: for u, v, delay in edges:
n=2; edges=[(1, 2, 1)]; start=2; adj=[[], []]; u=1; v=2; delay=1

L6 network_delay: adj[u - 1].append((v - 1, delay))
n=2; edges=[(1, 2, 1)]; start=2; adj=[[(..., ...)], []]; u=1; v=2; delay=1

L7 network_delay: distances = dijkstra(adj, start - 1)
n=2; edges=[(1, 2, 1)]; start=2; adj=[[(..., ...)], []]; u=1; v=2; delay=1; distances=[None,
    0]

L8 <genexpr>: return -1 if any(value is None for value in distances) else max(distances)
.0=list_iterator; value=None
```

Result: -1. Directed edges do not let node 2 send backward to node 1.

Why the solution is correct

Dijkstra gives the earliest achievable arrival at every reachable node. All nodes have received the signal exactly when the last of these arrivals occurs, so their maximum is necessary and sufficient. An unreachable node can never receive the signal.

Time and space

O(n+E log(E+2)) time and O(n+E) space, inherited from adjacency construction and lazy-heap Dijkstra.

Common mistakes

Summing arrival times incorrectly serializes concurrent propagation. Forgetting to convert both endpoints and the start mixes label conventions.

Reasoning card

State: The zero-based adjacency models the same directed network. distances[v] is the earliest arrival time at node v+1.

Invariant: Each finalized distance is the earliest arrival time.

Decision to rehearse: Run Dijkstra and take the maximum finite shortest distance.

Finish: Otherwise return the maximum distance

Transfer exercise

Allow several initial sources by initializing each at distance zero in the same heap.

209. Cheapest Flight Within K Stops

Interview | Module 21

Problem

Return the cheapest price from source to destination using at most k intermediate stops in a directed flight graph. n vertices use zero-based labels; flights are (u,v,nonnegative_price), and k>=0. At most k+1 flight edges are allowed. Return -1 if impossible; source==destination costs zero.

Examples

Example 1. No intermediate stop permits only the direct flight.

```
cheapest_flight(3,[(0,1,1),(1,2,1),(0,2,10)],0,2,0)
=> 10
```

Example 2. One intermediate stop permits the cheaper two-edge route.

```
cheapest_flight(3,[(0,1,1),(1,2,1),(0,2,10)],0,2,1)
=> 2
```

Example 3. No budget can create a missing route.

```
cheapest_flight(2,[],0,1,3)
=> -1
```

Why this mechanism works

The edge budget is part of the state. A cheap route with too many edges cannot replace a more expensive route with budget remaining. Use repeated relaxation layers: layer j stores the best cost using at most j edges. Read only the previous layer and write a fresh copy to prevent one pass from consuming several flights.

State and boundaries

dist holds costs using at most the completed number of edges. next_dist begins as a copy, retaining routes that use fewer edges.

Protocol

1. Initialize source cost zero.

2. Repeat k+1 layers.

3. Copy the previous costs.

4. Relax every flight using only old dist values.

5. Replace dist after the full edge scan.

6. Return destination cost or -1.

Visualization

```
0->1 cost 1, 1->2 cost 1, 0->2 cost 10
k=0 permits one edge -> 10
k=1 permits two edges -> 2
fresh layers enforce the difference
```

Python implementation

```
def cheapest_flight(n, flights, source, destination, k):
    dist = [float('inf')] * n
    dist[source] = 0
    for _ in range(k + 1):
        next_dist = dist.copy()
        for u, v, price in flights:
            next_dist[v] = min(next_dist[v], dist[u] + price)
        dist = next_dist
    return -1 if dist[destination] == float('inf') else dist[destination]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
cheapest_flight(3,[(0,1,1),(1,2,1),(0,2,10)],0,2,0)

L2 cheapest_flight: dist = [float('inf')] * n
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[inf, inf,
     inf]

L3 cheapest_flight: dist[source] = 0
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, inf,
    inf]

L4 cheapest_flight: for _ in range(k + 1):
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, inf,
    inf]; _=0

L5 cheapest_flight: next_dist = dist.copy()
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, inf,
    inf]; _=0; next_dist=[0, inf, inf]

L6 cheapest_flight: for u, v, price in flights:
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, inf,
    inf]; _=0; next_dist=[0, inf, inf]; u=0; v=1; price=1

L7 cheapest_flight: next_dist[v] = min(next_dist[v], dist[u] + price)
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, inf,
    inf]; _=0; next_dist=[0, 1, inf]; u=0; v=1; price=1

L6 cheapest_flight: for u, v, price in flights:
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, inf,
    inf]; _=0; next_dist=[0, 1, inf]; u=1; v=2; price=1

L6 cheapest_flight: for u, v, price in flights:
```

```
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, inf,
    inf]; _=0; next_dist=[0, 1, inf]; u=0; v=2; price=10

L7 cheapest_flight: next_dist[v] = min(next_dist[v], dist[u] + price)
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, inf,
    inf]; _=0; next_dist=[0, 1, 10]; u=0; v=2; price=10

L8 cheapest_flight: dist = next_dist
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=0; dist=[0, 1,
    10]; _=0; next_dist=[0, 1, 10]; u=0; v=2; price=10
```

Result: 10. No intermediate stop permits only the direct flight.

```
cheapest_flight(3,[(0,1,1),(1,2,1),(0,2,10)],0,2,1)

L2 cheapest_flight: dist = [float('inf')] * n
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[inf, inf,
     inf]

L3 cheapest_flight: dist[source] = 0
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, inf,
    inf]

L4 cheapest_flight: for _ in range(k + 1):
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, inf,
    inf]; _=0

L5 cheapest_flight: next_dist = dist.copy()
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, inf,
    inf]; _=0; next_dist=[0, inf, inf]

L6 cheapest_flight: for u, v, price in flights:
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, inf,
    inf]; _=0; next_dist=[0, inf, inf]; u=0; v=1; price=1

L7 cheapest_flight: next_dist[v] = min(next_dist[v], dist[u] + price)
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, inf,
    inf]; _=0; next_dist=[0, 1, inf]; u=0; v=1; price=1

L6 cheapest_flight: for u, v, price in flights:
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, inf,
    inf]; _=0; next_dist=[0, 1, inf]; u=1; v=2; price=1

L6 cheapest_flight: for u, v, price in flights:
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, inf,
    inf]; _=0; next_dist=[0, 1, inf]; u=0; v=2; price=10

... 4 intermediate state changes omitted ...

L6 cheapest_flight: for u, v, price in flights:
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, 1,
    10]; _=1; next_dist=[0, 1, 10]; u=1; v=2; price=1

L7 cheapest_flight: next_dist[v] = min(next_dist[v], dist[u] + price)
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, 1,
    10]; _=1; next_dist=[0, 1, 2]; u=1; v=2; price=1

L6 cheapest_flight: for u, v, price in flights:
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, 1,
    10]; _=1; next_dist=[0, 1, 2]; u=0; v=2; price=10
```

```
L8 cheapest_flight: dist = next_dist
n=3; flights=[(0, 1, 1), (1, 2, 1), (0, 2, 10)]; source=0; destination=2; k=1; dist=[0, 1, 2];
     _=1; next_dist=[0, 1, 2]; u=0; v=2; price=10
```

Result: 2. One intermediate stop permits the cheaper two-edge route.

Why the solution is correct

Any route with at most j edges either uses at most j-1 edges or extends such a route by one final flight. The copy and edge relaxations consider exactly these alternatives. Fresh-layer reads enforce the budget regardless of flight input order. Induction proves the final bounded cost optimal.

Time and space

O((k+1)*(V+E)) time, including each distance-array copy, and O(V) auxiliary space.

Common mistakes

In-place relaxation may chain several flights within one layer. Confusing stops with edges removes or adds one legal flight.

Reasoning card

State: dist holds costs using at most the completed number of edges. next_dist begins as a copy, retaining routes that use fewer edges.

Invariant: Cost state includes both destination and resources used.

Decision to rehearse: Use layered relaxation/Bellman-Ford or state (node,stops) in a heap.

Finish: Return destination cost or -1

Transfer exercise

Return the selected flights by storing a predecessor for every budget layer, rather than one predecessor per vertex.

210. Zero-One BFS

Interview | Module 21

Problem

Compute shortest distances from a valid source in a directed adjacency list whose weights are only zero or one. Return None for unreachable vertices. Parallel edges and cycles are allowed. Use a deque instead of a general priority heap.

Examples

Example 1. The zero-cost detour improves the initially proposed cost of vertex 1.

```
zero_one_bfs([[(1,1),(2,0)],[],[(1,0)],[]],0)
=> [0, 0, 0, None]
```

Example 2. A zero self-loop cannot cause a strict improvement.

```
zero_one_bfs([[(0,0),(1,1)],[]],0)
=> [0, 1]
```

Example 3. The source distance is zero.

```
zero_one_bfs([[]],0)
=> [0]
```

Why this mechanism works

From a minimum-distance vertex, a zero edge stays in the current distance bucket and belongs at the deque's front; a one edge belongs at the back. This maintains the required distance order with only two adjacent distance levels. Keep distance-tagged entries so an old proposal can be skipped after a later improvement.

State and boundaries

dist records best discovered costs. Nonstale deque entries are processed in nondecreasing distance; zero-cost improvements go before one-cost work.

Protocol

1. Initialize source at zero.
2. Pop from the deque front.
3. Skip stale entries.
4. Relax each outgoing edge.
5. Push an improved zero-edge route to the front, or a one-edge route to the back.
6. Convert unreachable infinities to None.

Visualization

```
0->1 weight 1; 0->2 weight 0; 2->1 weight 0
deque front favors 2 at distance 0
then improves 1 from 1 to 0 before stale work is used
```

Python implementation

```
from collections import deque

def zero_one_bfs(adj, source):
    dist = [float('inf')] * len(adj)
    dist[source] = 0
    queue = deque([(0, source)])
    while queue:
        distance, u = queue.popleft()
        if distance != dist[u]:
            continue
        for v, weight in adj[u]:
            candidate = distance + weight
            if candidate < dist[v]:
                dist[v] = candidate
                entry = (candidate, v)
                if weight == 0:
                    queue.appendleft(entry)
                else:
                    queue.append(entry)
    return [None if value == float('inf') else value for value in dist]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
zero_one_bfs([[(1,1),(2,0)],[],[(1,0)],[]],0)

L4 zero_one_bfs: dist = [float('inf')] * len(adj)
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[inf, inf, inf, inf]

L5 zero_one_bfs: dist[source] = 0
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf]

L6 zero_one_bfs: queue = deque([(0, source)])
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf];
    queue=deque([(0, 0)])

L8 zero_one_bfs: distance, u = queue.popleft()
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf];
    queue=deque([]); distance=0; u=0

L11 zero_one_bfs: for v, weight in adj[u]:
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf];
    queue=deque([]); distance=0; u=0; v=1; weight=1

L12 zero_one_bfs: candidate = distance + weight
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, inf, inf, inf];
    queue=deque([]); distance=0; u=0; v=1; weight=1; candidate=1

L14 zero_one_bfs: dist[v] = candidate
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 1, inf, inf];
```

```
    queue=deque([]); distance=0; u=0; v=1; weight=1; candidate=1

L15 zero_one_bfs: entry = (candidate, v)
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 1, inf, inf];
    queue=deque([]); distance=0; u=0; v=1; weight=1; candidate=1; entry=(1, 1)

... 12 intermediate state changes omitted ...

L8 zero_one_bfs: distance, u = queue.popleft()
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 0, 0, inf];
    queue=deque([]); distance=1; u=1; v=1; weight=0; candidate=0; entry=(0, 1)

L20 zero_one_bfs: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 0, 0, inf];
    queue=deque([]); distance=1; u=1; v=1; weight=0; candidate=0; entry=(0, 1); value=0

L20 zero_one_bfs: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 0, 0, inf];
    queue=deque([]); distance=1; u=1; v=1; weight=0; candidate=0; entry=(0, 1); value=inf

L20 zero_one_bfs: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], [], [(..., ...)], []]; source=0; dist=[0, 0, 0, inf];
    queue=deque([]); distance=1; u=1; v=1; weight=0; candidate=0; entry=(0, 1)
```

Result: [0, 0, 0, None]. The zero-cost detour improves the initially proposed cost of vertex 1.

`zero_one_bfs([[(0,0),(1,1)],[]],0)`

```
L4 zero_one_bfs: dist = [float('inf')] * len(adj)
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[inf, inf]

L5 zero_one_bfs: dist[source] = 0
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, inf]

L6 zero_one_bfs: queue = deque([(0, source)])
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, inf]; queue=deque([(0, 0)])

L8 zero_one_bfs: distance, u = queue.popleft()
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, inf]; queue=deque([]); distance=0; u=0

L11 zero_one_bfs: for v, weight in adj[u]:
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, inf]; queue=deque([]); distance=0; u=0;
     v=0; weight=0

L12 zero_one_bfs: candidate = distance + weight
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, inf]; queue=deque([]); distance=0; u=0;
     v=0; weight=0; candidate=0

L11 zero_one_bfs: for v, weight in adj[u]:
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, inf]; queue=deque([]); distance=0; u=0;
     v=1; weight=1; candidate=0

L12 zero_one_bfs: candidate = distance + weight
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, inf]; queue=deque([]); distance=0; u=0;
     v=1; weight=1; candidate=1

... 3 intermediate state changes omitted ...

L8 zero_one_bfs: distance, u = queue.popleft()
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, 1]; queue=deque([]); distance=1; u=1;
    v=1; weight=1; candidate=1; entry=(1, 1)

L20 zero_one_bfs: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, 1]; queue=deque([]); distance=1; u=1;
```

```
    v=1; weight=1; candidate=1; entry=(1, 1); value=0

L20 zero_one_bfs: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, 1]; queue=deque([]); distance=1; u=1;
    v=1; weight=1; candidate=1; entry=(1, 1); value=1

L20 zero_one_bfs: return [None if value == float('inf') else value for value in dist]
adj=[[(..., ...), (..., ...)], []]; source=0; dist=[0, 1]; queue=deque([]); distance=1; u=1;
    v=1; weight=1; candidate=1; entry=(1, 1)
```

Result: [0, 1]. A zero self-loop cannot cause a strict improvement.

Why the solution is correct

The front/back rule preserves the ordering that Dijkstra needs because every new distance is current or current+1. Therefore a nonstale minimum removal has its optimal distance. Each vertex's outgoing edges are processed when finalized, and strict improvements prevent endless zero-cycle scheduling.

Time and space

O(V+E) time and O(V+E) auxiliary space with lazy distance-tagged proposals; each edge can create at most one proposal when its source is processed.

Common mistakes

Appending zero-cost edges at the back loses the distance-order proof. A visited-on-enqueue rule rejects later zero-cost improvements. Weights outside {0,1} require a different frontier.

Reasoning card

State: dist records best discovered costs. Nonstale deque entries are processed in nondecreasing distance; zero-cost improvements go before one-cost work.

Invariant: Deque ordering preserves nondecreasing tentative distance.

Decision to rehearse: Push weight-0 transitions to deque front and weight-1 to back.

Finish: Convert unreachable infinities to None

Transfer exercise

Generalize to small integer weights with cyclic distance buckets, stating how many buckets are required.

Module 22
Greedy Algorithms and Kadane-Style State

Recognize when one compact frontier or a locally safe choice summarizes all useful history.

A greedy rule needs an exchange or dominance argument, not just a plausible choice. Earliest finishing intervals leave room for later choices; a failed gas-station prefix rules out several starts at once. Some one-pass recurrences also prepare the next chapter: distinguish the best answer ending here from the best answer anywhere.

The ten-step progression

211. Best Single Stock Transaction

212. Maximum Subarray Sum

213. Maximum Product Subarray

214. Maximum Circular Subarray

215. Jump Game Reachability

216. Minimum Jumps to End

217. Gas Station Circuit

218. Maximum Number of Non-overlapping Intervals

219. Partition Labels

220. Task Scheduler with Cooldown

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

211. Best Single Stock Transaction

Foundation | Module 22

Problem

Given daily nonnegative prices, return the maximum profit from at most one purchase followed by a sale on a later day. Return zero if no profitable transaction exists or fewer than two days are supplied. Do not mutate prices; no short selling is allowed.

Examples

Example 1. Buy at 1 before selling at 6.

```
stock_profit([7,1,5,3,6,4])
=> 5
```

Example 2. Skipping a transaction is better than a loss.

```
stock_profit([7,5,3])
=> 0
```

Example 3. There is no legal buy/sell pair.

```
stock_profit([])
=> 0
```

Why this mechanism works

Fix the sale day. The best compatible purchase is the cheapest earlier day, so the entire earlier prefix can be summarized by its minimum. Compare today's possible profit before adding today's price to that prefix. The best answer may also remain zero by choosing no transaction.

State and boundaries

lowest is the minimum price strictly before the current day. best is the greatest legal profit considered so far, including no trade.

Protocol

1. Initialize lowest to infinity and best to zero.
2. Compare today's sale price minus lowest with best.
3. Update lowest with today's price.
4. Return best.

Visualization

```
prices 7,1,5,3,6,4
cheapest earlier price before 6 is 1
profit 6-1=5; later 4 does not improve it
```

Python implementation

```
def stock_profit(prices):
    lowest, best = float('inf'), 0
    for price in prices:
        best = max(best, price - lowest)
        lowest = min(lowest, price)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
stock_profit([7,1,5,3,6,4])

L2 stock_profit: lowest, best = float('inf'), 0
prices=[7, 1, 5, 3, 6, 4]; lowest=inf; best=0

L3 stock_profit: for price in prices:
prices=[7, 1, 5, 3, 6, 4]; lowest=inf; best=0; price=7

L5 stock_profit: lowest = min(lowest, price)
prices=[7, 1, 5, 3, 6, 4]; lowest=7; best=0; price=7

L3 stock_profit: for price in prices:
prices=[7, 1, 5, 3, 6, 4]; lowest=7; best=0; price=1

L5 stock_profit: lowest = min(lowest, price)
prices=[7, 1, 5, 3, 6, 4]; lowest=1; best=0; price=1

L3 stock_profit: for price in prices:
prices=[7, 1, 5, 3, 6, 4]; lowest=1; best=0; price=5

L4 stock_profit: best = max(best, price - lowest)
prices=[7, 1, 5, 3, 6, 4]; lowest=1; best=4; price=5

L3 stock_profit: for price in prices:
prices=[7, 1, 5, 3, 6, 4]; lowest=1; best=4; price=3

L3 stock_profit: for price in prices:
prices=[7, 1, 5, 3, 6, 4]; lowest=1; best=4; price=6

L4 stock_profit: best = max(best, price - lowest)
prices=[7, 1, 5, 3, 6, 4]; lowest=1; best=5; price=6

L3 stock_profit: for price in prices:
prices=[7, 1, 5, 3, 6, 4]; lowest=1; best=5; price=4
```

Result: 5. Buy at 1 before selling at 6.

```
stock_profit([7,5,3])

L2 stock_profit: lowest, best = float('inf'), 0
prices=[7, 5, 3]; lowest=inf; best=0
```

```
L3 stock_profit: for price in prices:
prices=[7, 5, 3]; lowest=inf; best=0; price=7

L5 stock_profit: lowest = min(lowest, price)
prices=[7, 5, 3]; lowest=7; best=0; price=7

L3 stock_profit: for price in prices:
prices=[7, 5, 3]; lowest=7; best=0; price=5

L5 stock_profit: lowest = min(lowest, price)
prices=[7, 5, 3]; lowest=5; best=0; price=5

L3 stock_profit: for price in prices:
prices=[7, 5, 3]; lowest=5; best=0; price=3

L5 stock_profit: lowest = min(lowest, price)
prices=[7, 5, 3]; lowest=3; best=0; price=3
```

Result: 0. Skipping a transaction is better than a loss.

Why the solution is correct

For each sale day, choosing any earlier price above the prefix minimum cannot improve profit. The loop evaluates that day's optimal legal purchase and retains the maximum across all sale days. The zero initial result covers every unprofitable or empty case.

Time and space

O(n) time and O(1) auxiliary state.

Common mistakes

Using the global minimum without chronology may buy after selling. Summing all positive daily changes solves the multiple-transaction variant.

Reasoning card

State: lowest is the minimum price strictly before the current day. best is the greatest legal profit considered so far, including no trade.

Invariant: Minimum price is from an earlier or current day.

Decision to rehearse: Maintain lowest prior price and best profit.

Finish: Return best

Transfer exercise

Return purchase and sale indices with an explicit tie rule while preserving the same prefix-minimum summary.

212. Maximum Subarray Sum

Foundation | Module 22

Problem

Return the largest sum of a nonempty contiguous subarray of an integer list. Negative values are allowed. Return None for empty input. Do not mutate nums; selecting the empty subarray is not permitted for nonempty input.

Examples

Example 1. The interval [4,-1,2,1] gives the largest sum.

```
max_subarray([-2,1,-3,4,-1,2,1,-5,4])
=> 6
```

Example 2. A nonempty all-negative answer is the least negative singleton.

```
max_subarray([-5,-2,-9])
=> -2
```

Example 3. No nonempty subarray exists.

```
max_subarray([])
=> None
```

Why this mechanism works

Consider the best subarray ending exactly at the current position. It either starts here or extends the best ending at the previous position. A negative previous sum is harmful and should be discarded. Keep this local ending state separate from the best answer anywhere.

State and boundaries

ending is the largest sum of a nonempty subarray ending at the last processed index. best is the largest such sum over all processed endpoints.

Protocol

1. Handle empty input.
2. Initialize both states to the first value.
3. Set ending=max(current,ending+current).
4. Update best from ending.
5. Return best.

Visualization

```
[-2,1,-3,4,-1,2,1,-5,4]
at 4: restart rather than extend -2
extend 4,-1,2,1 -> sum 6
```

Python implementation

```
def max_subarray(nums):
    if not nums:
        return None
    ending = best = nums[0]
    for i in range(1, len(nums)):
        ending = max(nums[i], ending + nums[i])
        best = max(best, ending)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
max_subarray([-2,1,-3,4,-1,2,1,-5,4])

L2 max_subarray: if not nums:
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]

L4 max_subarray: ending = best = nums[0]
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=-2; best=-2

L5 max_subarray: for i in range(1, len(nums)):
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=-2; best=-2; i=1

L6 max_subarray: ending = max(nums[i], ending + nums[i])
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=1; best=-2; i=1

L7 max_subarray: best = max(best, ending)
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=1; best=1; i=1

L5 max_subarray: for i in range(1, len(nums)):
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=1; best=1; i=2

L6 max_subarray: ending = max(nums[i], ending + nums[i])
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=-2; best=1; i=2

L5 max_subarray: for i in range(1, len(nums)):
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=-2; best=1; i=3

... 10 intermediate state changes omitted ...

L5 max_subarray: for i in range(1, len(nums)):
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=6; best=6; i=7

L6 max_subarray: ending = max(nums[i], ending + nums[i])
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=1; best=6; i=7

L5 max_subarray: for i in range(1, len(nums)):
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=1; best=6; i=8
```

```
L6 max_subarray: ending = max(nums[i], ending + nums[i])
nums=[-2, 1, -3, 4, -1, 2, 1, -5, ...]; ending=5; best=6; i=8
```

Result: 6. The interval [4,-1,2,1] gives the largest sum.

```
max_subarray([-5,-2,-9])

L2 max_subarray: if not nums:
nums=[-5, -2, -9]

L4 max_subarray: ending = best = nums[0]
nums=[-5, -2, -9]; ending=-5; best=-5

L5 max_subarray: for i in range(1, len(nums)):
nums=[-5, -2, -9]; ending=-5; best=-5; i=1

L6 max_subarray: ending = max(nums[i], ending + nums[i])
nums=[-5, -2, -9]; ending=-2; best=-5; i=1

L7 max_subarray: best = max(best, ending)
nums=[-5, -2, -9]; ending=-2; best=-2; i=1

L5 max_subarray: for i in range(1, len(nums)):
nums=[-5, -2, -9]; ending=-2; best=-2; i=2

L6 max_subarray: ending = max(nums[i], ending + nums[i])
nums=[-5, -2, -9]; ending=-9; best=-2; i=2
```

Result: -2. A nonempty all-negative answer is the least negative singleton.

Why the solution is correct

Every subarray ending at the current index is either the singleton or a previous ending subarray plus the current value. Extending the maximum previous ending sum is optimal among extensions. Taking the maximum with the singleton gives the exact state, and the global maximum covers every endpoint.

Time and space

O(n) time and O(1) auxiliary state.

Common mistakes

Initializing best to zero incorrectly returns an empty-subarray result on all-negative input. Updating best from the raw current value misses multi-element runs.

Reasoning card

State: ending is the largest sum of a nonempty subarray ending at the last processed index. best is the largest such sum over all processed endpoints.

Invariant: ending_here is the best nonempty subarray ending exactly here.

Decision to rehearse: At each value, extend the previous best-ending-here or restart.

Finish: Return best

Transfer exercise

Track the candidate start index whenever the recurrence restarts, then return an optimal half-open interval.

213. Maximum Product Subarray

Foundation | Module 22

Problem

Return the maximum product of a nonempty contiguous subarray of integers, or None for empty input. Zeros and negative values are allowed. Do not mutate nums. Products use Python's exact integers.

Examples

Example 1. The negative minimum becomes the best positive product.

```
max_product_subarray([-2,3,-4])
=> 24
```

Example 2. Zero is better than either negative singleton.

```
max_product_subarray([-2,0,-1])
=> 0
```

Example 3. The empty list has no nonempty product interval.

```
max_product_subarray([])
=> None
```

Why this mechanism works

Multiplication by a negative value reverses order: the smallest previous ending product can become the largest new one. Therefore preserve both extremes. At each position compare starting anew with extending either old extreme, computing both new states before overwriting the old ones.

State and boundaries

high and low are the maximum and minimum products of nonempty subarrays ending at the previous index. best is the maximum over all completed endpoints.

Protocol

1. Initialize both extremes and best from the first value.

2. Form candidates current, old_high*current, old_low*current.

3. Assign new high and low from their maximum and minimum.

4. Update best.

Visualization

```
[-2,3,-4]
after 3: high=3, low=-6
multiply by -4: old low becomes 24
new maximum comes from the former minimum
```

Python implementation

```
def max_product_subarray(nums):
    if not nums:
        return None
    high = low = best = nums[0]
    for i in range(1, len(nums)):
        value = nums[i]
        candidates = (value, high * value, low * value)
        high, low = max(candidates), min(candidates)
        best = max(best, high)
    return best
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
max_product_subarray([-2,3,-4])

L2 max_product_subarray: if not nums:
nums=[-2, 3, -4]

L4 max_product_subarray: high = low = best = nums[0]
nums=[-2, 3, -4]; high=-2; low=-2; best=-2

L5 max_product_subarray: for i in range(1, len(nums)):
nums=[-2, 3, -4]; high=-2; low=-2; best=-2; i=1

L6 max_product_subarray: value = nums[i]
nums=[-2, 3, -4]; high=-2; low=-2; best=-2; i=1; value=3

L7 max_product_subarray: candidates = (value, high * value, low * value)
nums=[-2, 3, -4]; high=-2; low=-2; best=-2; i=1; value=3; candidates=(3, -6, -6)

L8 max_product_subarray: high, low = max(candidates), min(candidates)
nums=[-2, 3, -4]; high=3; low=-6; best=-2; i=1; value=3; candidates=(3, -6, -6)

L9 max_product_subarray: best = max(best, high)
nums=[-2, 3, -4]; high=3; low=-6; best=3; i=1; value=3; candidates=(3, -6, -6)

L5 max_product_subarray: for i in range(1, len(nums)):
nums=[-2, 3, -4]; high=3; low=-6; best=3; i=2; value=3; candidates=(3, -6, -6)

L6 max_product_subarray: value = nums[i]
nums=[-2, 3, -4]; high=3; low=-6; best=3; i=2; value=-4; candidates=(3, -6, -6)

L7 max_product_subarray: candidates = (value, high * value, low * value)
nums=[-2, 3, -4]; high=3; low=-6; best=3; i=2; value=-4; candidates=(-4, -12, 24)

L8 max_product_subarray: high, low = max(candidates), min(candidates)
nums=[-2, 3, -4]; high=24; low=-12; best=3; i=2; value=-4; candidates=(-4, -12, 24)
```

```
L9 max_product_subarray: best = max(best, high)
nums=[-2, 3, -4]; high=24; low=-12; best=24; i=2; value=-4; candidates=(-4, -12, 24)
```

Result: 24. The negative minimum becomes the best positive product.

```
max_product_subarray([-2,0,-1])

L2 max_product_subarray: if not nums:
nums=[-2, 0, -1]

L4 max_product_subarray: high = low = best = nums[0]
nums=[-2, 0, -1]; high=-2; low=-2; best=-2

L5 max_product_subarray: for i in range(1, len(nums)):
nums=[-2, 0, -1]; high=-2; low=-2; best=-2; i=1

L6 max_product_subarray: value = nums[i]
nums=[-2, 0, -1]; high=-2; low=-2; best=-2; i=1; value=0

L7 max_product_subarray: candidates = (value, high * value, low * value)
nums=[-2, 0, -1]; high=-2; low=-2; best=-2; i=1; value=0; candidates=(0, 0, 0)

L8 max_product_subarray: high, low = max(candidates), min(candidates)
nums=[-2, 0, -1]; high=0; low=0; best=-2; i=1; value=0; candidates=(0, 0, 0)

L9 max_product_subarray: best = max(best, high)
nums=[-2, 0, -1]; high=0; low=0; best=0; i=1; value=0; candidates=(0, 0, 0)

L5 max_product_subarray: for i in range(1, len(nums)):
nums=[-2, 0, -1]; high=0; low=0; best=0; i=2; value=0; candidates=(0, 0, 0)

L6 max_product_subarray: value = nums[i]
nums=[-2, 0, -1]; high=0; low=0; best=0; i=2; value=-1; candidates=(0, 0, 0)

L7 max_product_subarray: candidates = (value, high * value, low * value)
nums=[-2, 0, -1]; high=0; low=0; best=0; i=2; value=-1; candidates=(-1, 0, 0)

L8 max_product_subarray: high, low = max(candidates), min(candidates)
nums=[-2, 0, -1]; high=0; low=-1; best=0; i=2; value=-1; candidates=(-1, 0, 0)
```

Result: 0. Zero is better than either negative singleton.

Why the solution is correct

Every ending subarray is a singleton or an extension. Multiplication by a fixed value maps the range of previous products monotonically or reverse-monotonically, so its extrema arise from the old high or low. Considering those and the singleton yields both exact new extremes, including zero resets.

Time and space

O(n) arithmetic operations and O(1) integer variables. Exact product bit lengths can grow with n, so arithmetic time and bit storage are not constant for unbounded integers.

Common mistakes

Keeping only the maximum loses negative products that later become positive. Updating high before computing low can use the same element twice.

Reasoning card

State: high and low are the maximum and minimum products of nonempty subarrays ending at the previous index. best is the maximum over all completed endpoints.

Invariant: Two ending states cover all sign possibilities.

Decision to rehearse: Track both maximum and minimum ending here because negatives swap roles.

Finish: Update best

Transfer exercise

Track the intervals associated with both extremes, paying attention to which old interval each new candidate extends.

214. Maximum Circular Subarray

Foundation | Module 22

Problem

Return the maximum sum of a nonempty circular contiguous subarray, using each list position at most once. The interval may wrap from the end to the beginning. Return None for empty input. Negative values are allowed.

Examples

Example 1. The optimum crosses the array boundary.

```
max_circular_subarray([5,-3,5])
=> 10
```

Example 2. The empty complement is rejected by the negative guard.

```
max_circular_subarray([-3,-2,-5])
=> -2
```

Example 3. One position may be used once, not on repeated laps.

```
max_circular_subarray([7])
=> 7
```

Why this mechanism works

An optimum either does not wrap, handled by ordinary Kadane, or wraps and therefore excludes one middle interval. Maximizing a wrapped sum is total minus the minimum nonempty subarray sum. If every value is negative, excluding the entire array would create an illegal empty answer, so return the ordinary maximum in that case.

State and boundaries

high_end/low_end summarize maximum/minimum sums ending here. best/worst are their global extrema; total is the full processed sum.

Protocol

1. Initialize extrema from the first value.
2. Update both Kadane recurrences and total.
3. If best<0 return best.
4. Otherwise return max(best,total-worst).

Visualization

```
[5,-3,5], total=7, minimum middle=-3
wrapped selection: last 5 + first 5
7-(-3)=10
```

Python implementation

```
def max_circular_subarray(nums):
    if not nums:
        return None
    high_end = low_end = best = worst = total = nums[0]
    for i in range(1, len(nums)):
        value = nums[i]
        high_end = max(value, high_end + value)
        low_end = min(value, low_end + value)
        best, worst = max(best, high_end), min(worst, low_end)
        total += value
    return best if best < 0 else max(best, total - worst)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
max_circular_subarray([5,-3,5])

L2 max_circular_subarray: if not nums:
nums=[5, -3, 5]

L4 max_circular_subarray: high_end = low_end = best = worst = total = nums[0]
nums=[5, -3, 5]; high_end=5; low_end=5; best=5; worst=5; total=5

L5 max_circular_subarray: for i in range(1, len(nums)):
nums=[5, -3, 5]; high_end=5; low_end=5; best=5; worst=5; total=5; i=1

L6 max_circular_subarray: value = nums[i]
nums=[5, -3, 5]; high_end=5; low_end=5; best=5; worst=5; total=5; i=1; value=-3

L7 max_circular_subarray: high_end = max(value, high_end + value)
nums=[5, -3, 5]; high_end=2; low_end=5; best=5; worst=5; total=5; i=1; value=-3

L8 max_circular_subarray: low_end = min(value, low_end + value)
nums=[5, -3, 5]; high_end=2; low_end=-3; best=5; worst=5; total=5; i=1; value=-3

L9 max_circular_subarray: best, worst = max(best, high_end), min(worst, low_end)
nums=[5, -3, 5]; high_end=2; low_end=-3; best=5; worst=-3; total=5; i=1; value=-3

L10 max_circular_subarray: total += value
nums=[5, -3, 5]; high_end=2; low_end=-3; best=5; worst=-3; total=2; i=1; value=-3

... 2 intermediate state changes omitted ...

L7 max_circular_subarray: high_end = max(value, high_end + value)
nums=[5, -3, 5]; high_end=7; low_end=-3; best=5; worst=-3; total=2; i=2; value=5

L8 max_circular_subarray: low_end = min(value, low_end + value)
nums=[5, -3, 5]; high_end=7; low_end=2; best=5; worst=-3; total=2; i=2; value=5

L9 max_circular_subarray: best, worst = max(best, high_end), min(worst, low_end)
nums=[5, -3, 5]; high_end=7; low_end=2; best=7; worst=-3; total=2; i=2; value=5
```

```
L10 max_circular_subarray: total += value
nums=[5, -3, 5]; high_end=7; low_end=2; best=7; worst=-3; total=7; i=2; value=5
```

Result: 10. The optimum crosses the array boundary.

```
max_circular_subarray([-3,-2,-5])

L2 max_circular_subarray: if not nums:
nums=[-3, -2, -5]

L4 max_circular_subarray: high_end = low_end = best = worst = total = nums[0]
nums=[-3, -2, -5]; high_end=-3; low_end=-3; best=-3; worst=-3; total=-3

L5 max_circular_subarray: for i in range(1, len(nums)):
nums=[-3, -2, -5]; high_end=-3; low_end=-3; best=-3; worst=-3; total=-3; i=1

L6 max_circular_subarray: value = nums[i]
nums=[-3, -2, -5]; high_end=-3; low_end=-3; best=-3; worst=-3; total=-3; i=1; value=-2

L7 max_circular_subarray: high_end = max(value, high_end + value)
nums=[-3, -2, -5]; high_end=-2; low_end=-3; best=-3; worst=-3; total=-3; i=1; value=-2

L8 max_circular_subarray: low_end = min(value, low_end + value)
nums=[-3, -2, -5]; high_end=-2; low_end=-5; best=-3; worst=-3; total=-3; i=1; value=-2

L9 max_circular_subarray: best, worst = max(best, high_end), min(worst, low_end)
nums=[-3, -2, -5]; high_end=-2; low_end=-5; best=-2; worst=-5; total=-3; i=1; value=-2

L10 max_circular_subarray: total += value
nums=[-3, -2, -5]; high_end=-2; low_end=-5; best=-2; worst=-5; total=-5; i=1; value=-2

... 2 intermediate state changes omitted ...

L7 max_circular_subarray: high_end = max(value, high_end + value)
nums=[-3, -2, -5]; high_end=-5; low_end=-5; best=-2; worst=-5; total=-5; i=2; value=-5

L8 max_circular_subarray: low_end = min(value, low_end + value)
nums=[-3, -2, -5]; high_end=-5; low_end=-10; best=-2; worst=-5; total=-5; i=2; value=-5

L9 max_circular_subarray: best, worst = max(best, high_end), min(worst, low_end)
nums=[-3, -2, -5]; high_end=-5; low_end=-10; best=-2; worst=-10; total=-5; i=2; value=-5

L10 max_circular_subarray: total += value
nums=[-3, -2, -5]; high_end=-5; low_end=-10; best=-2; worst=-10; total=-10; i=2; value=-5
```

Result: -2. The empty complement is rejected by the negative guard.

Why the solution is correct

Every proper wrapped interval is the complement of a nonempty ordinary interval, so the minimum excluded sum gives its largest value. Nonwrapped candidates are covered separately. The all-negative guard prevents the only case where the invalid empty complement would exceed every legal answer; otherwise a legal ordinary answer handles that boundary.

Time and space

O(n) time and O(1) auxiliary numeric state.

Common mistakes

Returning total-worst unconditionally gives zero on all-negative input. Allowing repeated laps changes the problem and can make positive totals unbounded.

Reasoning card

State: high_end/low_end summarize maximum/minimum sums ending here. best/worst are their global extrema; total is the full processed sum.

Invariant: Wrapped optimum is the complement of a minimum middle segment.

Decision to rehearse: Compare ordinary max with total sum minus minimum subarray.

Finish: Otherwise return max(best,total-worst)

Transfer exercise

Return the wrapped interval as either one ordinary range or two ranges at the array ends.

215. Jump Game Reachability

Bridge | Module 22

Problem

Given nonnegative jump limits nums, return whether the last index is reachable from index zero. From i, any forward jump of length at most nums[i] is allowed. A singleton succeeds without jumping; empty input returns false because it has no start or destination.

Examples

Example 1. Reach extends from the early prefix to the final index.

```
can_jump([2,3,1,1,4])
=> True
```

Example 2. The zero at the boundary prevents crossing to index 4.

```
can_jump([3,2,1,0,4])
=> False
```

Example 3. The starting position is already the destination.

```
can_jump([0])
=> True
```

Why this mechanism works

If every position through farthest is reachable, a reachable index i can extend this prefix through i+nums[i]. Only that boundary matters; enumerating individual jump choices adds no information. Encountering an index beyond the boundary proves that no earlier reachable position can bridge the gap.

State and boundaries

farthest is the greatest index reachable using processed positions. All indices from zero through it are reachable.

Protocol

1. Reject empty input.
2. Scan indices from zero.
3. Reject when i>farthest.
4. Extend farthest with i+nums[i].
5. Succeed once the final index is covered.

Visualization

```
[3,2,1,0,4]
reachable frontier stays at index 3
index 4 lies beyond it -> failure
```

Python implementation

```
def can_jump(nums):
    if not nums:
        return False
    farthest = 0
    for i, jump in enumerate(nums):
        if i > farthest:
            return False
        farthest = max(farthest, i + jump)
        if farthest >= len(nums) - 1:
            return True
    return False
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
can_jump([2,3,1,1,4])

L2 can_jump: if not nums:
nums=[2, 3, 1, 1, 4]

L4 can_jump: farthest = 0
nums=[2, 3, 1, 1, 4]; farthest=0

L5 can_jump: for i, jump in enumerate(nums):
nums=[2, 3, 1, 1, 4]; farthest=0; i=0; jump=2

L8 can_jump: farthest = max(farthest, i + jump)
nums=[2, 3, 1, 1, 4]; farthest=2; i=0; jump=2

L5 can_jump: for i, jump in enumerate(nums):
nums=[2, 3, 1, 1, 4]; farthest=2; i=1; jump=3

L8 can_jump: farthest = max(farthest, i + jump)
nums=[2, 3, 1, 1, 4]; farthest=4; i=1; jump=3
```

Result: True. Reach extends from the early prefix to the final index.

```
can_jump([3,2,1,0,4])

L2 can_jump: if not nums:
nums=[3, 2, 1, 0, 4]

L4 can_jump: farthest = 0
nums=[3, 2, 1, 0, 4]; farthest=0

L5 can_jump: for i, jump in enumerate(nums):
nums=[3, 2, 1, 0, 4]; farthest=0; i=0; jump=3

L8 can_jump: farthest = max(farthest, i + jump)
nums=[3, 2, 1, 0, 4]; farthest=3; i=0; jump=3
```

```
L5 can_jump: for i, jump in enumerate(nums):
nums=[3, 2, 1, 0, 4]; farthest=3; i=1; jump=2

L5 can_jump: for i, jump in enumerate(nums):
nums=[3, 2, 1, 0, 4]; farthest=3; i=2; jump=1

L5 can_jump: for i, jump in enumerate(nums):
nums=[3, 2, 1, 0, 4]; farthest=3; i=3; jump=0

L5 can_jump: for i, jump in enumerate(nums):
nums=[3, 2, 1, 0, 4]; farthest=3; i=4; jump=4
```

Result: False. The zero at the boundary prevents crossing to index 4.

Why the solution is correct

The reachable-prefix property holds at zero. From a reachable position all intermediate forward destinations up to its jump limit are allowed, so taking the maximum preserves a contiguous reachable prefix. If the scan reaches a gap, no processed position can cross it and no unprocessed position is reachable to help.

Time and space

O(n) time and O(1) auxiliary state.

Common mistakes

Updating reach from an unreachable position invents a path. Treating nums[i] as an exact jump length violates the at-most contract.

Reasoning card

State: farthest is the greatest index reachable using processed positions. All indices from zero through it are reachable.

Invariant: Every index <= farthest is potentially reachable.

Decision to rehearse: Maintain the farthest reachable frontier.

Finish: Succeed once the final index is covered

Transfer exercise

Allow backward jumps too. Explain why one forward frontier no longer summarizes all reachable states.

216. Minimum Jumps to End

Bridge | Module 22

Problem

Return the fewest jumps needed to reach the last index from zero, with nonnegative at-most jump limits. The curriculum's primary inputs are reachable; this implementation additionally returns -1 for unreachable or empty input. A singleton needs zero jumps.

Examples

Example 1. The route 0->1->4 uses two jumps.

```
min_jumps([2,3,1,1,4])
=> 2
```

Example 2. No departure is required.

```
min_jumps([0])
=> 0
```

Example 3. The second frontier cannot expand.

```
min_jumps([1,0,2])
=> -1
```

Why this mechanism works

Compress BFS into index ranges. Every index through end is reachable within the current jump budget. While scanning that layer, collect the farthest destination available for one additional jump. Increase the jump count only when the current layer boundary is exhausted.

State and boundaries

end is the farthest index reachable with jumps jumps. farthest accumulates the next layer's boundary from scanned reachable positions.

Protocol

1. Handle empty/singleton cases.
2. Scan only positions before the destination.
3. Extend farthest.
4. At i==end reject a nonexpanding frontier.
5. Otherwise add one jump and move end to farthest.

6. Return when end covers the destination.

Visualization

```
[2,3,1,1,4]
zero-jump boundary 0
after first layer: boundary 2, jumps 1
scan indices 1..2 -> boundary 4, jumps 2
```

Python implementation

```
def min_jumps(nums):
    if not nums:
        return -1
    if len(nums) == 1:
        return 0
    jumps = end = farthest = 0
    for i in range(len(nums) - 1):
        farthest = max(farthest, i + nums[i])
        if i == end:
            if farthest == end:
                return -1
            jumps += 1
            end = farthest
            if end >= len(nums) - 1:
                return jumps
    return -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
min_jumps([2,3,1,1,4])

L2 min_jumps: if not nums:
nums=[2, 3, 1, 1, 4]

L6 min_jumps: jumps = end = farthest = 0
nums=[2, 3, 1, 1, 4]; jumps=0; end=0; farthest=0

L7 min_jumps: for i in range(len(nums) - 1):
nums=[2, 3, 1, 1, 4]; jumps=0; end=0; farthest=0; i=0

L8 min_jumps: farthest = max(farthest, i + nums[i])
nums=[2, 3, 1, 1, 4]; jumps=0; end=0; farthest=2; i=0

L12 min_jumps: jumps += 1
nums=[2, 3, 1, 1, 4]; jumps=1; end=0; farthest=2; i=0

L13 min_jumps: end = farthest
nums=[2, 3, 1, 1, 4]; jumps=1; end=2; farthest=2; i=0

L7 min_jumps: for i in range(len(nums) - 1):
nums=[2, 3, 1, 1, 4]; jumps=1; end=2; farthest=2; i=1

L8 min_jumps: farthest = max(farthest, i + nums[i])
nums=[2, 3, 1, 1, 4]; jumps=1; end=2; farthest=4; i=1
```

```
L7 min_jumps: for i in range(len(nums) - 1):
nums=[2, 3, 1, 1, 4]; jumps=1; end=2; farthest=4; i=2

L12 min_jumps: jumps += 1
nums=[2, 3, 1, 1, 4]; jumps=2; end=2; farthest=4; i=2

L13 min_jumps: end = farthest
nums=[2, 3, 1, 1, 4]; jumps=2; end=4; farthest=4; i=2
```

Result: 2. The route 0->1->4 uses two jumps.

```
min_jumps([0])

L2 min_jumps: if not nums:
nums=[0]
```

Result: 0. No departure is required.

Why the solution is correct

The first frontier represents zero jumps. Scanning every position in a layer discovers exactly all destinations obtainable with one additional jump. Advancing layers therefore follows minimum jump count just as BFS does. A frontier that cannot expand proves the destination unreachable.

Time and space

O(n) time and O(1) auxiliary state.

Common mistakes

Increasing jumps at every index counts explored positions instead of BFS layers. Processing the final destination as another departure can add an unnecessary jump.

Reasoning card

State: end is the farthest index reachable with jumps jumps. farthest accumulates the next layer's boundary from scanned reachable positions.

Invariant: One jump count corresponds to one completed frontier range.

Decision to rehearse: Treat current reachable range as a BFS layer and track next farthest.

Finish: Return when end covers the destination

Transfer exercise

Record which index extends each frontier furthest to reconstruct one minimum-jump route.

217. Gas Station Circuit

Bridge | Module 22

Problem

Given equally sized nonnegative gas and cost lists, return a station index from which one full clockwise circuit is possible with an initially empty, unlimited-capacity tank. At station i receive gas[i], then spend cost[i] to reach the next station. Return -1 if impossible or empty. Use the candidate-reset rule for deterministic selection.

Examples

Example 1. Station 3 begins a feasible circuit after the failed prefix.

```
gas_station([1,2,3,4,5],[3,4,5,1,2])
=> 3
```

Example 2. The total available fuel is insufficient.

```
gas_station([2,3,4],[3,4,3])
=> -1
```

Example 3. A zero-cost singleton circuit is feasible.

```
gas_station([0],[0])
=> 0
```

Why this mechanism works

Track global surplus separately from the current candidate's tank. If the candidate fails at i, every station between that candidate and i also fails: their skipped prefix had nonnegative surplus, so starting later cannot repair the negative remainder to i. Discard that entire block and try i+1.

State and boundaries

total is the whole processed surplus. tank is the surplus from candidate through the current station; every earlier candidate in discarded blocks has been disproved.

Protocol

1. Reject empty input.

2. Add gas[i]-cost[i] to total and tank.

3. When tank<0 move candidate to i+1 and reset tank.

4. Return candidate if total>=0, otherwise -1.

Visualization

```
net [-2,-2,-2,3,3]
failed blocks move candidate to 1,2,3
total=0; start 3 survives the suffix and wraparound
```

Python implementation

```
def gas_station(gas, cost):
    if not gas:
        return -1
    total = tank = candidate = 0
    for i in range(len(gas)):
        surplus = gas[i] - cost[i]
        total += surplus
        tank += surplus
        if tank < 0:
            candidate = i + 1
            tank = 0
    return candidate if total >= 0 else -1
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
gas_station([1,2,3,4,5],[3,4,5,1,2])

L2 gas_station: if not gas:
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]

L4 gas_station: total = tank = candidate = 0
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=0; tank=0; candidate=0

L5 gas_station: for i in range(len(gas)):
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=0; tank=0; candidate=0; i=0

L6 gas_station: surplus = gas[i] - cost[i]
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=0; tank=0; candidate=0; i=0; surplus=-2

L7 gas_station: total += surplus
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=-2; tank=0; candidate=0; i=0; surplus=-2

L8 gas_station: tank += surplus
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=-2; tank=-2; candidate=0; i=0; surplus=-2

L10 gas_station: candidate = i + 1
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=-2; tank=-2; candidate=1; i=0; surplus=-2

L11 gas_station: tank = 0
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=-2; tank=0; candidate=1; i=0; surplus=-2

... 13 intermediate state changes omitted ...

L8 gas_station: tank += surplus
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=-3; tank=3; candidate=3; i=3; surplus=3

L5 gas_station: for i in range(len(gas)):
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=-3; tank=3; candidate=3; i=4; surplus=3

L7 gas_station: total += surplus
```

```
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=0; tank=3; candidate=3; i=4; surplus=3

L8 gas_station: tank += surplus
gas=[1, 2, 3, 4, 5]; cost=[3, 4, 5, 1, 2]; total=0; tank=6; candidate=3; i=4; surplus=3
```

Result: 3. Station 3 begins a feasible circuit after the failed prefix.

```
gas_station([2,3,4],[3,4,3])

L2 gas_station: if not gas:
gas=[2, 3, 4]; cost=[3, 4, 3]

L4 gas_station: total = tank = candidate = 0
gas=[2, 3, 4]; cost=[3, 4, 3]; total=0; tank=0; candidate=0

L5 gas_station: for i in range(len(gas)):
gas=[2, 3, 4]; cost=[3, 4, 3]; total=0; tank=0; candidate=0; i=0

L6 gas_station: surplus = gas[i] - cost[i]
gas=[2, 3, 4]; cost=[3, 4, 3]; total=0; tank=0; candidate=0; i=0; surplus=-1

L7 gas_station: total += surplus
gas=[2, 3, 4]; cost=[3, 4, 3]; total=-1; tank=0; candidate=0; i=0; surplus=-1

L8 gas_station: tank += surplus
gas=[2, 3, 4]; cost=[3, 4, 3]; total=-1; tank=-1; candidate=0; i=0; surplus=-1

L10 gas_station: candidate = i + 1
gas=[2, 3, 4]; cost=[3, 4, 3]; total=-1; tank=-1; candidate=1; i=0; surplus=-1

L11 gas_station: tank = 0
gas=[2, 3, 4]; cost=[3, 4, 3]; total=-1; tank=0; candidate=1; i=0; surplus=-1

... 5 intermediate state changes omitted ...

L5 gas_station: for i in range(len(gas)):
gas=[2, 3, 4]; cost=[3, 4, 3]; total=-2; tank=0; candidate=2; i=2; surplus=-1

L6 gas_station: surplus = gas[i] - cost[i]
gas=[2, 3, 4]; cost=[3, 4, 3]; total=-2; tank=0; candidate=2; i=2; surplus=1

L7 gas_station: total += surplus
gas=[2, 3, 4]; cost=[3, 4, 3]; total=-1; tank=0; candidate=2; i=2; surplus=1

L8 gas_station: tank += surplus
gas=[2, 3, 4]; cost=[3, 4, 3]; total=-1; tank=1; candidate=2; i=2; surplus=1
```

Result: -1. The total available fuel is insufficient.

Why the solution is correct

A negative total makes any full circuit impossible. Each failed block eliminates all its possible starts by the prefix-surplus argument. The final candidate lies after a minimum cumulative-prefix boundary, so all subsequent cyclic partial sums are nonnegative when total is nonnegative. It therefore completes the circuit.

Time and space

O(n) time and O(1) auxiliary state.

Common mistakes

Resetting total with tank discards the global feasibility condition. A finite tank capacity changes the surplus argument.

Reasoning card

State: total is the whole processed surplus. tank is the surplus from candidate through the current station; every earlier candidate in discarded blocks has been disproved.

Invariant: Total surplus determines global feasibility.

Decision to rehearse: If running tank fails at i, no start since the current candidate can work.

Finish: Return candidate if total>=0, otherwise -1

Transfer exercise

Return every feasible start using prefix sums and a sliding minimum over a doubled surplus array.

218. Maximum Number of Non-overlapping Intervals

Interview | Module 22

Problem

Select as many mutually nonoverlapping half-open intervals [start,end) as possible and return their endpoint tuples in selection order. Every interval satisfies start<end. Touching endpoints are compatible. Sort by (end,start,input_index) to break ties deterministically; leave input unchanged.

Examples

Example 1. Earliest finishes permit three compatible choices.

```
select_intervals([(1,4),(2,3),(3,5),(5,6)])
=> [(2, 3), (3, 5), (5, 6)]
```

Example 2. No intervals can be selected.

```
select_intervals([])
=> []
```

Example 3. Identical nonempty intervals overlap, so only one is chosen.

```
select_intervals([(0,2),(0,2)])
=> [(0, 2)]
```

Why this mechanism works

The interval finishing earliest leaves at least as much room for future intervals as any alternative first choice. After selecting it, solve the same problem among intervals starting at or after its end. Sorting once allows this repeated exchange argument to become one scan.

State and boundaries

selected is a compatible sequence. last_end is its final endpoint; an interval is available exactly when start>=last_end.

Protocol

1. Sort indexed intervals by finishing time and tie keys.

2. Begin with no chosen endpoint.

3. Accept each interval whose start is compatible.

4. Update last_end only on acceptance.

5. Return the selected sequence.

Visualization

```
[1,4), [2,3), [3,5), [5,6)
choose [2,3), then [3,5), then [5,6)
earliest start [1,4) would block the first two
```

Python implementation

```
def select_intervals(intervals):
    ordered = sorted(enumerate(intervals), key=lambda item:
                     (item[1][1], item[1][0], item[0]))
    selected, last_end = [], None
    for _, (start, end) in ordered:
        if last_end is None or start >= last_end:
            selected.append((start, end))
            last_end = end
    return selected
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
select_intervals([(1,4),(2,3),(3,5),(5,6)])

L3 <lambda>: (item[1][1], item[1][0], item[0]))
item=(0, (1, 4))

L3 <lambda>: (item[1][1], item[1][0], item[0]))
item=(1, (2, 3))

L3 <lambda>: (item[1][1], item[1][0], item[0]))
item=(2, (3, 5))

L3 <lambda>: (item[1][1], item[1][0], item[0]))
item=(3, (5, 6))

L2 select_intervals: ordered = sorted(enumerate(intervals), key=lambda item:
intervals=[(1, 4), (2, 3), (3, 5), (5, 6)]; ordered=[(1, (..., ...)), (0, (..., ...)), (2,
    (..., ...)), (3, (..., ...))]

L4 select_intervals: selected, last_end = [], None
intervals=[(1, 4), (2, 3), (3, 5), (5, 6)]; ordered=[(1, (..., ...)), (0, (..., ...)), (2,
    (..., ...)), (3, (..., ...))]; selected=[]; last_end=None

L5 select_intervals: for _, (start, end) in ordered:
intervals=[(1, 4), (2, 3), (3, 5), (5, 6)]; ordered=[(1, (..., ...)), (0, (..., ...)), (2,
    (..., ...)), (3, (..., ...))]; selected=[]; last_end=None; _=1; start=2; end=3

L7 select_intervals: selected.append((start, end))
intervals=[(1, 4), (2, 3), (3, 5), (5, 6)]; ordered=[(1, (..., ...)), (0, (..., ...)), (2,
    (..., ...)), (3, (..., ...))]; selected=[(2, 3)]; last_end=None; _=1; start=2; end=3

... 4 intermediate state changes omitted ...

L8 select_intervals: last_end = end
intervals=[(1, 4), (2, 3), (3, 5), (5, 6)]; ordered=[(1, (..., ...)), (0, (..., ...)), (2,
    (..., ...)), (3, (..., ...))]; selected=[(2, 3), (3, 5)]; last_end=5; _=2; start=3; end=5
```

```
L5 select_intervals: for _, (start, end) in ordered:
intervals=[(1, 4), (2, 3), (3, 5), (5, 6)]; ordered=[(1, (..., ...)), (0, (..., ...)), (2,
    (..., ...)), (3, (..., ...))]; selected=[(2, 3), (3, 5)]; last_end=5; _=3; start=5; end=6

L7 select_intervals: selected.append((start, end))
intervals=[(1, 4), (2, 3), (3, 5), (5, 6)]; ordered=[(1, (..., ...)), (0, (..., ...)), (2,
    (..., ...)), (3, (..., ...))]; selected=[(2, 3), (3, 5), (5, 6)]; last_end=5; _=3;
    start=5; end=6

L8 select_intervals: last_end = end
intervals=[(1, 4), (2, 3), (3, 5), (5, 6)]; ordered=[(1, (..., ...)), (0, (..., ...)), (2,
    (..., ...)), (3, (..., ...))]; selected=[(2, 3), (3, 5), (5, 6)]; last_end=6; _=3;
    start=5; end=6
```

Result: [(2, 3), (3, 5), (5, 6)]. Earliest finishes permit three compatible choices.

```
select_intervals([])

L2 select_intervals: ordered = sorted(enumerate(intervals), key=lambda item:
intervals=[]; ordered=[]

L4 select_intervals: selected, last_end = [], None
intervals=[]; ordered=[]; selected=[]; last_end=None
```

Result: []. No intervals can be selected.

Why the solution is correct

Take an optimum solution's first interval. Replacing it with the earliest-finishing available interval cannot overlap any later selected interval, because its end is no later. Thus an optimum exists containing the greedy choice. Applying the same exchange to the remaining suffix proves maximum cardinality.

Time and space

O(n log n) time and O(n) sorting/output storage.

Common mistakes

Choosing earliest start can select a long interval that blocks many short ones. Using start>last_end incorrectly rejects touching half-open intervals.

Reasoning card

State: selected is a compatible sequence. last_end is its final endpoint; an interval is available exactly when start>=last_end.

Invariant: Chosen intervals leave maximal room for future choices.

Decision to rehearse: Sort by earliest finishing time.

Finish: Return the selected sequence

Transfer exercise

Assign each interval a value and maximize total value. Explain why earliest finish alone no longer proves optimality.

219. Partition Labels

Interview | Module 22

Problem

Partition a string into the maximum number of contiguous nonempty parts such that each distinct character appears in only one part. Return the part lengths from left to right. Matching is exact; empty input returns [].

Examples

Example 1. Each character remains inside exactly one of the three parts.

```
partition_labels("ababcbacadefegdehijhklij")
=> [9, 7, 8]
```

Example 2. Distinct characters permit every immediate cut.

```
partition_labels("abc")
=> [1, 1, 1]
```

Example 3. There are no nonempty parts.

```
partition_labels("")
=> []
```

Why this mechanism works

Any character encountered in the current part forces that part to extend through the character's final occurrence. Keep the farthest of these obligations. When the scan reaches that boundary, every character seen in the part is complete, so closing immediately is safe and maximizes opportunities for later parts.

State and boundaries

end is the farthest last occurrence among characters in the current part. start is that part's first index.

Protocol

1. Record each character's last index.
2. Scan the string.
3. Extend end with the current character's last index.
4. At i==end append i-start+1.
5. Begin a fresh part at i+1.

Visualization

```
ababcbaca | defegde | hijhklij
first part's obligations extend through index 8
lengths 9,7,8
```

Python implementation

```
def partition_labels(text):
    last = {char: i for i, char in enumerate(text)}
    output, start, end = [], 0, 0
    for i, char in enumerate(text):
        end = max(end, last[char])
        if i == end:
            output.append(i - start + 1)
            start = i + 1
    return output
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
partition_labels("ababcbacadefegdehijhklij")

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='ababcbacadefegdehijhklij'; i=0; char='a'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='ababcbacadefegdehijhklij'; i=1; char='b'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='ababcbacadefegdehijhklij'; i=2; char='a'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='ababcbacadefegdehijhklij'; i=3; char='b'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='ababcbacadefegdehijhklij'; i=4; char='c'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='ababcbacadefegdehijhklij'; i=5; char='b'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='ababcbacadefegdehijhklij'; i=6; char='a'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='ababcbacadefegdehijhklij'; i=7; char='c'

... 50 intermediate state changes omitted ...

L4 partition_labels: for i, char in enumerate(text):
text='ababcbacadefegdehijhklij'; last={'a':8, 'b':5, 'c':7, 'd':14, 'e':15, ...}; output=[9,
    7]; start=16; end=23; i=22; char='i'

L4 partition_labels: for i, char in enumerate(text):
text='ababcbacadefegdehijhklij'; last={'a':8, 'b':5, 'c':7, 'd':14, 'e':15, ...}; output=[9,
    7]; start=16; end=23; i=23; char='j'

L7 partition_labels: output.append(i - start + 1)
text='ababcbacadefegdehijhklij'; last={'a':8, 'b':5, 'c':7, 'd':14, 'e':15, ...}; output=[9,
```

```
    7, 8]; start=16; end=23; i=23; char='j'

L8 partition_labels: start = i + 1
text='ababcbacadefegdehijhklij'; last={'a':8, 'b':5, 'c':7, 'd':14, 'e':15, ...}; output=[9,
    7, 8]; start=24; end=23; i=23; char='j'
```

Result: [9, 7, 8]. Each character remains inside exactly one of the three parts.

```
partition_labels("abc")

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='abc'; i=0; char='a'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='abc'; i=1; char='b'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='abc'; i=2; char='c'

L2 partition_labels: last = {char: i for i, char in enumerate(text)}
text='abc'; last={'a':0, 'b':1, 'c':2}

L3 partition_labels: output, start, end = [], 0, 0
text='abc'; last={'a':0, 'b':1, 'c':2}; output=[]; start=0; end=0

L4 partition_labels: for i, char in enumerate(text):
text='abc'; last={'a':0, 'b':1, 'c':2}; output=[]; start=0; end=0; i=0; char='a'

L7 partition_labels: output.append(i - start + 1)
text='abc'; last={'a':0, 'b':1, 'c':2}; output=[1]; start=0; end=0; i=0; char='a'

L8 partition_labels: start = i + 1
text='abc'; last={'a':0, 'b':1, 'c':2}; output=[1]; start=1; end=0; i=0; char='a'

... 4 intermediate state changes omitted ...

L4 partition_labels: for i, char in enumerate(text):
text='abc'; last={'a':0, 'b':1, 'c':2}; output=[1, 1]; start=2; end=1; i=2; char='c'

L5 partition_labels: end = max(end, last[char])
text='abc'; last={'a':0, 'b':1, 'c':2}; output=[1, 1]; start=2; end=2; i=2; char='c'

L7 partition_labels: output.append(i - start + 1)
text='abc'; last={'a':0, 'b':1, 'c':2}; output=[1, 1, 1]; start=2; end=2; i=2; char='c'

L8 partition_labels: start = i + 1
text='abc'; last={'a':0, 'b':1, 'c':2}; output=[1, 1, 1]; start=3; end=2; i=2; char='c'
```

Result: [1, 1, 1]. Distinct characters permit every immediate cut.

Why the solution is correct

No legal cut can occur before end because it would split some character's occurrences. At end, every character in the part has its final occurrence inside it, so the cut is valid. Taking every earliest legal cut cannot reduce later choices and yields the maximum number of parts.

Time and space

Expected O(n) time and O(u) auxiliary map space for u distinct characters, plus O(n) worst-case output lengths.

Common mistakes

Using the first occurrence instead of the last closes parts too early. Returning endpoints instead of lengths changes the requested output.

Reasoning card

State: end is the farthest last occurrence among characters in the current part. start is that part's first index.

Invariant: Segment closes only when current index reaches its required farthest end.

Decision to rehearse: Extend current segment to the farthest last occurrence seen.

Finish: Begin a fresh part at i+1

Transfer exercise

Return the substrings themselves and account for the total copying cost.

220. Task Scheduler with Cooldown

Interview | Module 22

Problem

Given unit-duration tasks identified by hashable labels and cooldown n>=0, return the minimum number of time slots needed to execute them all. Equal labels require at least n intervening slots, which may contain other tasks or idle time. Tasks may be reordered freely; empty input returns zero.

Examples

Example 1. Two idle slots are unavoidable between the repeated labels.

```
task_scheduler(list("AAABBB"),2)
=> 8
```

Example 2. No cooldown requires only the six execution slots.

```
task_scheduler(list("AAABBB"),0)
=> 6
```

Example 3. No work requires no time.

```
task_scheduler([],3)
=> 0
```

Example 4. Other labels fill all required gaps without idling.

```
task_scheduler(list("AABBCC"),1)
=> 6
```

Why this mechanism works

Let f be the largest frequency and c the number of labels occurring f times. The first f-1 occurrences of a most frequent label require frames of at least n+1 slots, followed by a final group of c tied labels. This gives a spacing lower bound. The task count gives another lower bound; other labels either fill the frames or make the schedule longer without idles.

State and boundaries

frequencies records task multiplicities. maximum=f and tied=c determine the unavoidable cooldown frame length.

Protocol

1. Count labels and handle empty input.

2. Find the maximum frequency.

3. Count labels tied at that maximum.

4. Compute (f-1)*(n+1)+c.

5. Return the larger of that bound and the number of tasks.

Visualization

```
A A A B B B, n=2
one optimum: A B _ A B _ A B
f=3,c=2 -> (3-1)*3+2 = 8 slots
```

Python implementation

```
from collections import Counter

def task_scheduler(tasks, n):
    if not tasks:
        return 0
    counts = Counter(tasks)
    maximum = max(counts.values())
    tied = sum(value == maximum for value in counts.values())
    return max(len(tasks), (maximum - 1) * (n + 1) + tied)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
task_scheduler(list("AAABBB"),2)

L4 task_scheduler: if not tasks:
tasks=['A', 'A', 'A', 'B', 'B', 'B']; n=2

L6 task_scheduler: counts = Counter(tasks)
tasks=['A', 'A', 'A', 'B', 'B', 'B']; n=2; counts={'A':3, 'B':3}

L7 task_scheduler: maximum = max(counts.values())
tasks=['A', 'A', 'A', 'B', 'B', 'B']; n=2; counts={'A':3, 'B':3}; maximum=3

L8 <genexpr>: tied = sum(value == maximum for value in counts.values())
.0=dict_valueiterator; value=3; maximum=3

L8 <genexpr>: tied = sum(value == maximum for value in counts.values())
.0=dict_valueiterator; value=3; maximum=3

L8 <genexpr>: tied = sum(value == maximum for value in counts.values())
.0=dict_valueiterator; value=3; maximum=3

L8 task_scheduler: tied = sum(value == maximum for value in counts.values())
tasks=['A', 'A', 'A', 'B', 'B', 'B']; n=2; counts={'A':3, 'B':3}; maximum=3; tied=2
```

Result: 8. Two idle slots are unavoidable between the repeated labels.

```
task_scheduler(list("AAABBB"),0)

L4 task_scheduler: if not tasks:
tasks=['A', 'A', 'A', 'B', 'B', 'B']; n=0

L6 task_scheduler: counts = Counter(tasks)
tasks=['A', 'A', 'A', 'B', 'B', 'B']; n=0; counts={'A':3, 'B':3}

L7 task_scheduler: maximum = max(counts.values())
tasks=['A', 'A', 'A', 'B', 'B', 'B']; n=0; counts={'A':3, 'B':3}; maximum=3

L8 <genexpr>: tied = sum(value == maximum for value in counts.values())
.0=dict_valueiterator; value=3; maximum=3

L8 <genexpr>: tied = sum(value == maximum for value in counts.values())
.0=dict_valueiterator; value=3; maximum=3

L8 <genexpr>: tied = sum(value == maximum for value in counts.values())
.0=dict_valueiterator; value=3; maximum=3

L8 task_scheduler: tied = sum(value == maximum for value in counts.values())
tasks=['A', 'A', 'A', 'B', 'B', 'B']; n=0; counts={'A':3, 'B':3}; maximum=3; tied=2
```

Result: 6. No cooldown requires only the six execution slots.

Why the solution is correct

Any schedule must fit every task and separate repeated maximum-frequency labels, establishing both lower bounds. Arrange maximum-frequency labels in successive frames and distribute lower-frequency labels among the gaps. If enough tasks fill all gaps, task count suffices; otherwise remaining gaps are idle, attaining the frame bound.

Time and space

Expected O(T+U) time and O(U) auxiliary frequency storage for T tasks and U distinct labels.

Common mistakes

Using only one most frequent label undercounts the final tied group. Cooldown n means n intervening slots, so equal-label start positions differ by at least n+1.

Reasoning card

State: frequencies records task multiplicities. maximum=f and tied=c determine the unavoidable cooldown frame length.

Invariant: Unavailable tasks remain blocked until their release time.

Decision to rehearse: Use frequency geometry or a max-heap plus cooldown queue.

Finish: Return the larger of that bound and the number of tasks

Transfer exercise

Construct a concrete schedule with a max-heap and cooldown queue, and verify its length against this formula.

Module 23
Dynamic Programming I: One-Dimensional State

Define one subproblem precisely, derive its recurrence, set base cases, and choose a valid computation order.

Define one subproblem before optimizing its storage. Prefix length, exact amount and endpoint index describe different states and lead to different recurrences. Base cases are part of the model, especially the empty prefix. Only compress a table after you can identify which previous answers every update must still read.

The ten-step progression

221. Fibonacci with Rolling State

222. Climbing Stairs

223. Minimum Cost Climbing Stairs

224. House Robber

225. House Robber II

226. Decode Ways

227. Word Break

228. Minimum Coin Change

229. Coin-Change Combination Count

230. Longest Increasing Subsequence

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

221. Fibonacci with Rolling State

Foundation | Module 23

Problem

Return Fibonacci number F(n) for integer n>=0, with F(0)=0, F(1)=1 and F(n)=F(n-1)+F(n-2). Use a constant number of integer variables rather than a table. Python integers preserve the exact result.

Examples

Example 1. Seven pair advances leave F(7) in previous.

```
fibonacci(7)
=> 13
```

Example 2. No update changes the defining base value.

```
fibonacci(0)
=> 0
```

Example 3. One update reaches the second base value.

```
fibonacci(1)
=> 1
```

Why this mechanism works

The recurrence needs only two consecutive answers. Define the rolling state before coding its assignment: at iteration i, previous is F(i) and current is F(i+1). Advancing both simultaneously retains the old pair long enough to compute the next one.

State and boundaries

Before iteration i, (previous,current)=(F(i),F(i+1)). The loop advances this pair exactly n times.

Protocol

1. Initialize (0,1).

2. Repeat n times.

3. Assign previous=current and current=old_previous+old_current simultaneously.

4. Return previous.

Visualization

```
i=0: (0,1)
i=1: (1,1)
i=2: (1,2)
i=3: (2,3) -> previous is F(3)
```

Python implementation

```
def fibonacci(n):
    previous, current = 0, 1
    for _ in range(n):
        previous, current = current, previous + current
    return previous
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
fibonacci(7)

L2 fibonacci: previous, current = 0, 1
n=7; previous=0; current=1

L3 fibonacci: for _ in range(n):
n=7; previous=0; current=1; _=0

L4 fibonacci: previous, current = current, previous + current
n=7; previous=1; current=1; _=0

L3 fibonacci: for _ in range(n):
n=7; previous=1; current=1; _=1

L4 fibonacci: previous, current = current, previous + current
n=7; previous=1; current=2; _=1

L3 fibonacci: for _ in range(n):
n=7; previous=1; current=2; _=2

L4 fibonacci: previous, current = current, previous + current
n=7; previous=2; current=3; _=2

L3 fibonacci: for _ in range(n):
n=7; previous=2; current=3; _=3

... 3 intermediate state changes omitted ...

L3 fibonacci: for _ in range(n):
n=7; previous=5; current=8; _=5

L4 fibonacci: previous, current = current, previous + current
n=7; previous=8; current=13; _=5

L3 fibonacci: for _ in range(n):
n=7; previous=8; current=13; _=6

L4 fibonacci: previous, current = current, previous + current
n=7; previous=13; current=21; _=6
```

Result: 13. Seven pair advances leave F(7) in previous.

```
fibonacci(0)

L2 fibonacci: previous, current = 0, 1
n=0; previous=0; current=1
```

Result: 0. No update changes the defining base value.

Why the solution is correct

The initial pair is the two defining base values. The Fibonacci recurrence transforms a correct consecutive pair into the next consecutive pair, preserving the invariant. After n advances previous is exactly F(n). The fixed iteration count establishes termination.

Time and space

O(n) additions and O(1) integer variables. Exact Fibonacci values have O(n) bits, so bit storage is O(n) and total elementary bit work for these additions is O(n^2), not literal constant-byte space.

Common mistakes

Sequentially overwriting previous before forming the sum adds the new current twice. Returning current gives F(n+1) under this invariant.

Reasoning card

State: Before iteration i, (previous,current)=(F(i),F(i+1)). The loop advances this pair exactly n times.

Invariant: previous and current are consecutive Fibonacci values.

Decision to rehearse: Update consecutive states simultaneously.

Finish: Return previous

Transfer exercise

Derive fast doubling identities to reduce the number of arithmetic operations to O(log n).

222. Climbing Stairs

Foundation | Module 23

Problem

Count distinct ordered ways to climb exactly n steps using moves of one or two steps, for n>=0. Different move orders count separately. There is one way to climb zero steps: take no moves. Return an exact integer.

Examples

Example 1. The three ordered move sequences are distinct.

```
climb_stairs(3)
=> 3
```

Example 2. The empty move sequence is one valid route.

```
climb_stairs(0)
=> 1
```

Example 3. Either two single steps or one double step reaches the top.

```
climb_stairs(2)
=> 2
```

Why this mechanism works

Classify routes by their final move. A one-step finish extends any route to n-1; a two-step finish extends any route to n-2. These classes are disjoint and exhaustive, so their counts add. Base counts are ways(0)=1 and ways(1)=1.

State and boundaries

Before computing step i>=2, previous=ways(i-2) and current=ways(i-1).

Protocol

1. Return one for n<=1.
2. Initialize the two base counts.
3. For steps 2 through n, advance to (current,previous+current).
4. Return current.

Visualization

```
n=3: [1,1,1], [1,2], [2,1]
last move 1: extend two routes to step 2
last move 2: extend one route to step 1
```

Python implementation

```
def climb_stairs(n):
    if n <= 1:
        return 1
    previous = current = 1
    for _ in range(2, n + 1):
        previous, current = current, previous + current
    return current
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
climb_stairs(3)

L2 climb_stairs: if n <= 1:
n=3

L4 climb_stairs: previous = current = 1
n=3; previous=1; current=1

L5 climb_stairs: for _ in range(2, n + 1):
n=3; previous=1; current=1; _=2

L6 climb_stairs: previous, current = current, previous + current
n=3; previous=1; current=2; _=2

L5 climb_stairs: for _ in range(2, n + 1):
n=3; previous=1; current=2; _=3

L6 climb_stairs: previous, current = current, previous + current
n=3; previous=2; current=3; _=3
```

Result: 3. The three ordered move sequences are distinct.

```
climb_stairs(0)

L2 climb_stairs: if n <= 1:
n=0
```

Result: 1. The empty move sequence is one valid route.

Why the solution is correct

Removing the final move gives a unique shorter route in exactly one of the two predecessor classes. Conversely, appending that move creates a valid route. The sum recurrence therefore counts every route once. Rolling updates preserve the last two complete answers until n is reached.

Time and space

O(n) additions and O(1) integer variables; exact counts grow to O(n) bits, so constant-variable storage is not constant bit storage.

Common mistakes

Using ways(0)=0 removes routes that begin with a two-step move. Counting unordered numbers of ones and twos ignores distinct move sequences.

Reasoning card

State: Before computing step i>=2, previous=ways(i-2) and current=ways(i-1).

Invariant: Rolling states are the complete answers for recent step counts.

Decision to rehearse: Ways to reach i come from i-1 and i-2.

Finish: Return current

Transfer exercise

Allow moves from a supplied positive step-size set and define the corresponding sum over predecessor states.

223. Minimum Cost Climbing Stairs

Foundation | Module 23

Problem

Given nonnegative costs for stair indices 0 through n-1, reach position n at minimum cost. You may start at position 0 or 1 for free; when leaving stair i, pay cost[i] and move one or two positions. Empty and singleton lists therefore cost zero because the top can be a free starting position.

Examples

Example 1. Start at stair 1, pay 15, and jump to the top.

```
min_stair_cost([10,15,20])
=> 15
```

Example 2. The top is already a free starting position.

```
min_stair_cost([])
=> 0
```

Example 3. Starting at position 1 is starting at the top.

```
min_stair_cost([5])
=> 0
```

Why this mechanism works

Define dp[i] as the cheapest cost to reach position i before paying to leave it. This makes dp[0]=dp[1]=0. To reach i>=2, either leave i-1 after paying its cost or leave i-2 after paying its cost. The destination top has no fee of its own.

State and boundaries

previous and current are dp[i-2] and dp[i-1] before computing dp[i]. Costs belong to departure stairs.

Protocol

1. Initialize the free start states to zero.

2. For each position 2 through n compute min(previous+cost[i-2],current+cost[i-1]).

3. Roll the states.

4. Return the cost at position n.

Visualization

```
cost [10,15,20]
dp[0]=dp[1]=0
dp[2]=min(10,15)=10
dp[3]=min(15,30)=15
```

Python implementation

```
def min_stair_cost(cost):
    previous = current = 0
    for i in range(2, len(cost) + 1):
        next_cost = min(previous + cost[i - 2], current + cost[i - 1])
        previous, current = current, next_cost
    return current
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
min_stair_cost([10,15,20])

L2 min_stair_cost: previous = current = 0
cost=[10, 15, 20]; previous=0; current=0

L3 min_stair_cost: for i in range(2, len(cost) + 1):
cost=[10, 15, 20]; previous=0; current=0; i=2

L4 min_stair_cost: next_cost = min(previous + cost[i - 2], current + cost[i - 1])
cost=[10, 15, 20]; previous=0; current=0; i=2; next_cost=10

L5 min_stair_cost: previous, current = current, next_cost
cost=[10, 15, 20]; previous=0; current=10; i=2; next_cost=10

L3 min_stair_cost: for i in range(2, len(cost) + 1):
cost=[10, 15, 20]; previous=0; current=10; i=3; next_cost=10

L4 min_stair_cost: next_cost = min(previous + cost[i - 2], current + cost[i - 1])
cost=[10, 15, 20]; previous=0; current=10; i=3; next_cost=15

L5 min_stair_cost: previous, current = current, next_cost
cost=[10, 15, 20]; previous=10; current=15; i=3; next_cost=15
```

Result: 15. Start at stair 1, pay 15, and jump to the top.

```
min_stair_cost([])

L2 min_stair_cost: previous = current = 0
cost=[]; previous=0; current=0
```

Result: 0. The top is already a free starting position.

Why the solution is correct

Every route into i has one of the two allowed departure stairs. Optimal predecessor costs plus their departure fees consider all legal final moves, so the minimum is exact. The ascending order supplies both predecessor answers before use.

Time and space

O(n) time and O(1) auxiliary numeric state.

Common mistakes

Mixing a pay-on-arrival state with pay-on-departure transitions double-charges or omits a stair. The top is position n, outside the cost list.

Reasoning card

State: previous and current are dp[i-2] and dp[i-1] before computing dp[i]. Costs belong to departure stairs.

Invariant: Each DP state is minimum cost to reach its position.

Decision to rehearse: Current cost depends on the cheaper of the prior two reachable states.

Finish: Return the cost at position n

Transfer exercise

Return one cheapest route by retaining which predecessor attained each position's minimum.

224. House Robber

Foundation | Module 23

Problem

Return the maximum sum obtainable from a list of nonnegative house values without choosing adjacent positions. Choosing no houses is allowed, so empty input returns zero. Values belong to distinct positions even when equal. Do not mutate nums.

Examples

Example 1. Choosing positions 0,2,4 obeys nonadjacency.

```
rob_houses([2,7,9,3,1])
=> 12
```

Example 2. The empty selection has sum zero.

```
rob_houses([])
=> 0
```

Example 3. Both endpoint houses can be chosen in a line.

```
rob_houses([5,1,5])
=> 10
```

Why this mechanism works

For the current house, every legal optimum either skips it or takes it. Skipping preserves the best answer through the previous house. Taking excludes that previous house and adds the current value to the best through two positions back. These alternatives describe complete legal selections, not just local value comparisons.

State and boundaries

previous is the best for the prefix excluding the most recent house; current is the best for the entire processed prefix.

Protocol

1. Initialize both empty-prefix states to zero.

2. Compute max(current,previous+value) for each house.

3. Roll previous to old current and current to the new optimum.

4. Return current.

Visualization

```
[2,7,9,3,1]
prefix best: 2,7,11,11,12
final selection indices 0,2,4 gives 2+9+1=12
```

Python implementation

```
def rob_houses(nums):
    previous = current = 0
    for value in nums:
        previous, current = current, max(current, previous + value)
    return current
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
rob_houses([2,7,9,3,1])

L2 rob_houses: previous = current = 0
nums=[2, 7, 9, 3, 1]; previous=0; current=0

L3 rob_houses: for value in nums:
nums=[2, 7, 9, 3, 1]; previous=0; current=0; value=2

L4 rob_houses: previous, current = current, max(current, previous + value)
nums=[2, 7, 9, 3, 1]; previous=0; current=2; value=2

L3 rob_houses: for value in nums:
nums=[2, 7, 9, 3, 1]; previous=0; current=2; value=7

L4 rob_houses: previous, current = current, max(current, previous + value)
nums=[2, 7, 9, 3, 1]; previous=2; current=7; value=7

L3 rob_houses: for value in nums:
nums=[2, 7, 9, 3, 1]; previous=2; current=7; value=9

L4 rob_houses: previous, current = current, max(current, previous + value)
nums=[2, 7, 9, 3, 1]; previous=7; current=11; value=9

L3 rob_houses: for value in nums:
nums=[2, 7, 9, 3, 1]; previous=7; current=11; value=3

L4 rob_houses: previous, current = current, max(current, previous + value)
nums=[2, 7, 9, 3, 1]; previous=11; current=11; value=3

L3 rob_houses: for value in nums:
nums=[2, 7, 9, 3, 1]; previous=11; current=11; value=1

L4 rob_houses: previous, current = current, max(current, previous + value)
nums=[2, 7, 9, 3, 1]; previous=11; current=12; value=1
```

Result: 12. Choosing positions 0,2,4 obeys nonadjacency.

```
rob_houses([])

L2 rob_houses: previous = current = 0
nums=[]; previous=0; current=0
```

Result: 0. The empty selection has sum zero.

Why the solution is correct

Partition legal selections by whether they include the current house. The skip and take recurrences give the optimum within each class, and their maximum gives the optimum over the full prefix. The zero bases allow the first house and the empty selection correctly.

Time and space

O(n) time and O(1) auxiliary state.

Common mistakes

Choosing the locally largest house can block two smaller houses whose sum is larger. Updating previous too early can permit adjacent selections.

Reasoning card

State: previous is the best for the prefix excluding the most recent house; current is the best for the entire processed prefix.

Invariant: Best-through-current summarizes all legal selections so far.

Decision to rehearse: Compare skipping current versus taking it plus best two positions back.

Finish: Return current

Transfer exercise

Allow negative values and require at least one house. Identify which base cases and state meanings must change.

225. House Robber II

Bridge | Module 23

Problem

Return the maximum nonadjacent sum of nonnegative house values arranged in a circle, where first and last positions are adjacent. Empty input returns zero and a singleton may be chosen once. Do not mutate nums.

Examples

Example 1. The two endpoint houses conflict around the circle.

```
rob_circle([2,3,2])
=> 3
```

Example 2. Positions 0 and 2 form the best legal selection.

```
rob_circle([1,2,3,1])
=> 4
```

Example 3. The only house can be selected once.

```
rob_circle([7])
=> 7
```

Why this mechanism works

A legal circular selection cannot contain both endpoints. Therefore it belongs to at least one of two linear cases: exclude the last house, or exclude the first. Solve each range using the previous kata's take/skip recurrence and choose the larger result. A singleton needs special handling because both ranges would otherwise be empty.

State and boundaries

linear(left,right) returns the optimum for the half-open noncircular range. The two calls cover all legal circular selections.

Protocol

1. Handle zero and one house.

2. Solve [0,n-1) with rolling take/skip state.

3. Solve [1,n) the same way.

4. Return the larger result.

Visualization

```
[2,3,2] around a circle
exclude last -> [2,3] best 3
exclude first -> [3,2] best 3
choosing both 2s is forbidden
```

Python implementation

```
def rob_circle(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]
    def linear(left, right):
        previous = current = 0
        for i in range(left, right):
            previous, current = current, max(current, previous + nums[i])
        return current
    return max(linear(0, len(nums) - 1), linear(1, len(nums)))
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
rob_circle([2,3,2])

L2 rob_circle: if not nums:
nums=[2, 3, 2]

L7 linear: previous = current = 0
left=0; right=2; previous=0; current=0; nums=[2, 3, 2]

L8 linear: for i in range(left, right):
left=0; right=2; previous=0; current=0; nums=[2, 3, 2]; i=0

L9 linear: previous, current = current, max(current, previous + nums[i])
left=0; right=2; previous=0; current=2; nums=[2, 3, 2]; i=0

L8 linear: for i in range(left, right):
left=0; right=2; previous=0; current=2; nums=[2, 3, 2]; i=1

L9 linear: previous, current = current, max(current, previous + nums[i])
left=0; right=2; previous=2; current=3; nums=[2, 3, 2]; i=1

L7 linear: previous = current = 0
left=1; right=3; previous=0; current=0; nums=[2, 3, 2]

L8 linear: for i in range(left, right):
left=1; right=3; previous=0; current=0; nums=[2, 3, 2]; i=1

L9 linear: previous, current = current, max(current, previous + nums[i])
left=1; right=3; previous=0; current=3; nums=[2, 3, 2]; i=1

L8 linear: for i in range(left, right):
left=1; right=3; previous=0; current=3; nums=[2, 3, 2]; i=2
```

```
L9 linear: previous, current = current, max(current, previous + nums[i])
left=1; right=3; previous=3; current=3; nums=[2, 3, 2]; i=2
```

Result: 3. The two endpoint houses conflict around the circle.

```
rob_circle([1,2,3,1])

L2 rob_circle: if not nums:
nums=[1, 2, 3, 1]

L7 linear: previous = current = 0
left=0; right=3; previous=0; current=0; nums=[1, 2, 3, 1]

L8 linear: for i in range(left, right):
left=0; right=3; previous=0; current=0; nums=[1, 2, 3, 1]; i=0

L9 linear: previous, current = current, max(current, previous + nums[i])
left=0; right=3; previous=0; current=1; nums=[1, 2, 3, 1]; i=0

L8 linear: for i in range(left, right):
left=0; right=3; previous=0; current=1; nums=[1, 2, 3, 1]; i=1

L9 linear: previous, current = current, max(current, previous + nums[i])
left=0; right=3; previous=1; current=2; nums=[1, 2, 3, 1]; i=1

L8 linear: for i in range(left, right):
left=0; right=3; previous=1; current=2; nums=[1, 2, 3, 1]; i=2

L9 linear: previous, current = current, max(current, previous + nums[i])
left=0; right=3; previous=2; current=4; nums=[1, 2, 3, 1]; i=2

... 3 intermediate state changes omitted ...

L8 linear: for i in range(left, right):
left=1; right=4; previous=0; current=2; nums=[1, 2, 3, 1]; i=2

L9 linear: previous, current = current, max(current, previous + nums[i])
left=1; right=4; previous=2; current=3; nums=[1, 2, 3, 1]; i=2

L8 linear: for i in range(left, right):
left=1; right=4; previous=2; current=3; nums=[1, 2, 3, 1]; i=3

L9 linear: previous, current = current, max(current, previous + nums[i])
left=1; right=4; previous=3; current=3; nums=[1, 2, 3, 1]; i=3
```

Result: 4. Positions 0 and 2 form the best legal selection.

Why the solution is correct

Every circular solution excludes at least one endpoint and is therefore legal in one of the two ranges. Conversely, every nonadjacent selection inside either range is legal on the circle because the excluded endpoint breaks the wraparound conflict. Taking the larger linear optimum is thus exact.

Time and space

O(n) time over two scans and O(1) auxiliary state; index ranges avoid allocating slices.

Common mistakes

Running the linear algorithm once can select both endpoints. Treating a singleton as two mutually adjacent distinct houses wrongly removes its value.

Reasoning card

State: linear(left,right) returns the optimum for the half-open noncircular range. The two calls cover all legal circular selections.

Invariant: Every valid circular solution belongs to at least one case.

Decision to rehearse: Split into two linear cases: exclude first or exclude last.

Finish: Return the larger result

Transfer exercise

Return a selected index set from the winning linear case, translating its indices back to the original circle.

226. Decode Ways

Bridge | Module 23

Problem

Count decodings of an ASCII digit string under 1->A through 26->Z. Codes have no leading zero; '0' alone is invalid, while '10' and '20' are valid. Return zero for an empty message under this public contract. Different partitions into valid codes count separately.

Examples

Example 1. Three valid code partitions produce three decodings.

```
decode_ways("226")
=> 3
```

Example 2. A leading zero cannot begin any code.

```
decode_ways("06")
=> 0
```

Example 3. Only the two-digit code is valid.

```
decode_ways("10")
=> 1
```

Example 4. The public empty-message result is explicitly zero.

```
decode_ways("")
=> 0
```

Why this mechanism works

Use prefix length as the state. A decoding of the first i digits ends with either a valid one-digit code or a valid two-digit code. Their predecessor prefix counts add because the final code lengths distinguish the classes. The internal empty-prefix count is one even though the public empty-message result is defined as zero.

State and boundaries

Before processing length i, previous=dp[i-2] and current=dp[i-1]. dp[0]=1 represents one way to decode an already-consumed empty prefix.

Protocol

1. Handle empty text.

2. Initialize dp[0]=1 and dp[1] from the first digit.

3. For each later prefix add current for a nonzero final digit.

4. Add previous for a final pair between 10 and 26.

5. Roll states.

Visualization

```
'226' -> 2|2|6, 22|6, 2|26
prefix counts: dp[0]=1, dp[1]=1, dp[2]=2, dp[3]=3
```

Python implementation

```
def decode_ways(text):
    if not text:
        return 0
    previous, current = 1, int(text[0] != '0')
    for i in range(2, len(text) + 1):
        ways = current if text[i - 1] != '0' else 0
        if 10 <= int(text[i - 2:i]) <= 26:
            ways += previous
        previous, current = current, ways
    return current
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
decode_ways("226")

L2 decode_ways: if not text:
text='226'

L4 decode_ways: previous, current = 1, int(text[0] != '0')
text='226'; previous=1; current=1

L5 decode_ways: for i in range(2, len(text) + 1):
text='226'; previous=1; current=1; i=2

L6 decode_ways: ways = current if text[i - 1] != '0' else 0
text='226'; previous=1; current=1; i=2; ways=1

L8 decode_ways: ways += previous
text='226'; previous=1; current=1; i=2; ways=2

L9 decode_ways: previous, current = current, ways
text='226'; previous=1; current=2; i=2; ways=2

L5 decode_ways: for i in range(2, len(text) + 1):
text='226'; previous=1; current=2; i=3; ways=2

L8 decode_ways: ways += previous
text='226'; previous=1; current=2; i=3; ways=3

L9 decode_ways: previous, current = current, ways
text='226'; previous=2; current=3; i=3; ways=3
```

Result: 3. Three valid code partitions produce three decodings.

```
decode_ways("06")

L2 decode_ways: if not text:
text='06'

L4 decode_ways: previous, current = 1, int(text[0] != '0')
text='06'; previous=1; current=0

L5 decode_ways: for i in range(2, len(text) + 1):
text='06'; previous=1; current=0; i=2

L6 decode_ways: ways = current if text[i - 1] != '0' else 0
text='06'; previous=1; current=0; i=2; ways=0

L9 decode_ways: previous, current = current, ways
text='06'; previous=0; current=0; i=2; ways=0
```

Result: 0. A leading zero cannot begin any code.

Why the solution is correct

Removing the final valid code maps each decoding uniquely to a predecessor decoding. The one- and two-digit validity tests include exactly the permitted codes, so the recurrence counts all decodings without duplication. Ascending prefix length supplies complete predecessor counts.

Time and space

O(n) small-substring checks and additions, using O(1) integer variables. Exact count bit lengths can grow with n.

Common mistakes

Treating zero as a standalone code admits invalid messages. Setting internal dp[0] to zero removes valid two-digit decodings such as '10'.

Reasoning card

State: Before processing length i, previous=dp[i-2] and current=dp[i-1]. dp[0]=1 represents one way to decode an already-consumed empty prefix.

Invariant: dp[i] counts decodings of the first i characters.

Decision to rehearse: Add ways from valid one-digit and two-digit predecessors.

Finish: Roll states

Transfer exercise

Allow '*' as an unknown digit and count the valid one- and two-character interpretations for each transition.

227. Word Break

Bridge | Module 23

Problem

Return whether text can be split into a sequence of nonempty dictionary words. Words may be reused, matching is exact, and duplicate dictionary entries are consolidated. Ignore empty dictionary entries. Empty text returns true because the empty sequence is a valid segmentation.

Examples

Example 1. The cut after four characters joins two dictionary words.

```
word_break("leetcode",["leet","code"])
=> True
```

Example 2. Every candidate segmentation leaves an invalid remainder.

```
word_break("catsandog",["cats","dog","sand","and","cat"])
=> False
```

Example 3. The empty sequence segments the empty string.

```
word_break("",[])
=> True
```

Why this mechanism works

Let dp[i] mean the prefix ending before i is segmentable. To set it true, find an earlier reachable cut start whose following substring is a dictionary word. Bound candidate word length by the dictionary's longest word to avoid checking impossible long pieces.

State and boundaries

dp[i] is true exactly when text[:i] can be segmented. dp[0] is the reachable empty prefix.

Protocol

1. Build a set of nonempty words and their maximum length.
2. Set dp[0]=true.
3. For each end, try eligible earlier starts.
4. If dp[start] and text[start:end] is a word, mark end and stop its search.
5. Return dp[n].

Visualization

```
text 'leetcode', dictionary {leet,code}
dp[0]=true -> dp[4]=true via 'leet'
dp[4]=true -> dp[8]=true via 'code'
```

Python implementation

```
def word_break(text, dictionary):
    words = {word for word in dictionary if word}
    limit = min(len(text), max(map(len, words), default=0))
    dp = [False] * (len(text) + 1)
    dp[0] = True
    for end in range(1, len(text) + 1):
        for start in range(max(0, end - limit), end):
            if dp[start] and text[start:end] in words:
                dp[end] = True
                break
    return dp[-1]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
word_break("leetcode",["leet","code"])

L2 word_break: words = {word for word in dictionary if word}
text='leetcode'; dictionary=['leet', 'code']; word='leet'

L2 word_break: words = {word for word in dictionary if word}
text='leetcode'; dictionary=['leet', 'code']; word='code'

L2 word_break: words = {word for word in dictionary if word}
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}

L3 word_break: limit = min(len(text), max(map(len, words), default=0))
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4

L4 word_break: dp = [False] * (len(text) + 1)
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4; dp=[False,
    False, False, False, False, False, False, False, ...]

L5 word_break: dp[0] = True
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4; dp=[True,
    False, False, False, False, False, False, False, ...]

L6 word_break: for end in range(1, len(text) + 1):
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4; dp=[True,
    False, False, False, False, False, False, False, ...]; end=1

L7 word_break: for start in range(max(0, end - limit), end):
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4; dp=[True,
    False, False, False, False, False, False, False, ...]; end=1; start=0

... 22 intermediate state changes omitted ...

L7 word_break: for start in range(max(0, end - limit), end):
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4; dp=[True,
    False, False, False, True, False, False, False, ...]; end=7; start=5
```

```
L7 word_break: for start in range(max(0, end - limit), end):
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4; dp=[True,
    False, False, False, True, False, False, False, ...]; end=7; start=6

L6 word_break: for end in range(1, len(text) + 1):
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4; dp=[True,
    False, False, False, True, False, False, False, ...]; end=8; start=6

L7 word_break: for start in range(max(0, end - limit), end):
text='leetcode'; dictionary=['leet', 'code']; words={'code', 'leet'}; limit=4; dp=[True,
    False, False, False, True, False, False, False, ...]; end=8; start=4
```

Result: True. The cut after four characters joins two dictionary words.

```
word_break("catsandog",["cats","dog","sand","and","cat"])

L2 word_break: words = {word for word in dictionary if word}
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; word='cats'

L2 word_break: words = {word for word in dictionary if word}
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; word='dog'

L2 word_break: words = {word for word in dictionary if word}
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; word='sand'

L2 word_break: words = {word for word in dictionary if word}
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; word='and'

L2 word_break: words = {word for word in dictionary if word}
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; word='cat'

L2 word_break: words = {word for word in dictionary if word}
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; words={'and', 'cat',
    'cats', 'dog', 'sand'}

L3 word_break: limit = min(len(text), max(map(len, words), default=0))
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; words={'and', 'cat',
    'cats', 'dog', 'sand'}; limit=4

L4 word_break: dp = [False] * (len(text) + 1)
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; words={'and', 'cat',
    'cats', 'dog', 'sand'}; limit=4; dp=[False, False, False, False, False, False, False,
    False, ...]

... 29 intermediate state changes omitted ...

L7 word_break: for start in range(max(0, end - limit), end):
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; words={'and', 'cat',
    'cats', 'dog', 'sand'}; limit=4; dp=[True, False, False, True, True, False, False, True,
    ...]; end=9; start=5

L7 word_break: for start in range(max(0, end - limit), end):
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; words={'and', 'cat',
    'cats', 'dog', 'sand'}; limit=4; dp=[True, False, False, True, True, False, False, True,
    ...]; end=9; start=6

L7 word_break: for start in range(max(0, end - limit), end):
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; words={'and', 'cat',
    'cats', 'dog', 'sand'}; limit=4; dp=[True, False, False, True, True, False, False, True,
    ...]; end=9; start=7
```

```
L7 word_break: for start in range(max(0, end - limit), end):
text='catsandog'; dictionary=['cats', 'dog', 'sand', 'and', 'cat']; words={'and', 'cat',
    'cats', 'dog', 'sand'}; limit=4; dp=[True, False, False, True, True, False, False, True,
    ...]; end=9; start=8
```

Result: False. Every candidate segmentation leaves an invalid remainder.

Why the solution is correct

Any successful transition appends a valid word to a valid prefix, so it constructs a segmentation. Conversely, the final word of any segmentation begins at a reachable earlier cut examined by the loop. The maximum-length bound cannot remove a dictionary word. Induction over end proves every state exact.

Time and space

O(n*L^2+S) worst-case Python time, including substring creation/hashing, for maximum relevant word length L<=n and total dictionary characters S; O(n+S) storage. Without slicing costs the transition count is O(n*L).

Common mistakes

A greedy longest-word choice can block a later valid segmentation. Claiming O(n^2) while repeatedly creating length-O(n) substrings overlooks Python's copying cost.

Reasoning card

State: dp[i] is true exactly when text[:i] can be segmented. dp[0] is the reachable empty prefix.

Invariant: dp[i] means prefix s[:i] is segmentable.

Decision to rehearse: A position is reachable if an earlier reachable position forms a valid word.

Finish: Return dp[n]

Transfer exercise

Use a trie to scan forward from reachable cuts, avoiding substring allocation at every candidate boundary.

228. Minimum Coin Change

Interview | Module 23

Problem

Given positive integer coin denominations and a nonnegative amount, return the minimum number of coins summing exactly to that amount, with unlimited reuse. Duplicate denominations are ignored. Return -1 if unreachable and zero for amount zero. Do not mutate the inputs.

Examples

Example 1. Two coins of value 3 beat the greedy choice.

```
min_coin_change([1,3,4],6)
=> 2
```

Example 2. No exact odd amount is reachable.

```
min_coin_change([2],3)
=> -1
```

Example 3. The empty selection reaches zero.

```
min_coin_change([],0)
=> 0
```

Why this mechanism works

Define dp[x] as the optimum for exact amount x. Every nonempty solution has a final coin c and an optimal solution for x-c before it; replacing a nonoptimal prefix would improve the whole solution. Positive coins ensure the predecessor amount is smaller, so ascending amounts form a valid computation order.

State and boundaries

dp[0]=0; an infinity sentinel marks amounts not yet known reachable. At amount x, every dp[x-c] read is already complete.

Protocol

1. Deduplicate positive coins.

2. Initialize exact-amount states to infinity except zero.

3. Scan amounts upward.

4. For every fitting coin minimize dp[x] with dp[x-coin]+1.

5. Convert an unreachable final state to -1.

Visualization

```
coins [1,3,4], amount 6
greedy 4+1+1 uses 3 coins
dp[6] can extend dp[3]=1 with coin 3 -> 2 coins
```

Python implementation

```
def min_coin_change(coins, amount):
    values = set(coins)
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for total in range(1, amount + 1):
        for coin in values:
            if coin <= total:
                dp[total] = min(dp[total], dp[total - coin] + 1)
    return -1 if dp[amount] == float('inf') else dp[amount]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
min_coin_change([1,3,4],6)

L2 min_coin_change: values = set(coins)
coins=[1, 3, 4]; amount=6; values={1, 3, 4}

L3 min_coin_change: dp = [float('inf')] * (amount + 1)
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[inf, inf, inf, inf, inf, inf, inf]

L4 min_coin_change: dp[0] = 0
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, inf, inf, inf, inf, inf, inf]

L5 min_coin_change: for total in range(1, amount + 1):
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, inf, inf, inf, inf, inf, inf]; total=1

L6 min_coin_change: for coin in values:
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, inf, inf, inf, inf, inf, inf]; total=1;
    coin=1

L8 min_coin_change: dp[total] = min(dp[total], dp[total - coin] + 1)
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, 1, inf, inf, inf, inf, inf]; total=1;
    coin=1

L6 min_coin_change: for coin in values:
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, 1, inf, inf, inf, inf, inf]; total=1;
    coin=3

L6 min_coin_change: for coin in values:
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, 1, inf, inf, inf, inf, inf]; total=1;
    coin=4

... 24 intermediate state changes omitted ...

L8 min_coin_change: dp[total] = min(dp[total], dp[total - coin] + 1)
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, 1, 2, 1, 1, 2, 3]; total=6; coin=1

L6 min_coin_change: for coin in values:
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, 1, 2, 1, 1, 2, 3]; total=6; coin=3
```

```
L8 min_coin_change: dp[total] = min(dp[total], dp[total - coin] + 1)
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, 1, 2, 1, 1, 2, 2]; total=6; coin=3

L6 min_coin_change: for coin in values:
coins=[1, 3, 4]; amount=6; values={1, 3, 4}; dp=[0, 1, 2, 1, 1, 2, 2]; total=6; coin=4
```

Result: 2. Two coins of value 3 beat the greedy choice.

```
min_coin_change([2],3)

L2 min_coin_change: values = set(coins)
coins=[2]; amount=3; values={2}

L3 min_coin_change: dp = [float('inf')] * (amount + 1)
coins=[2]; amount=3; values={2}; dp=[inf, inf, inf, inf]

L4 min_coin_change: dp[0] = 0
coins=[2]; amount=3; values={2}; dp=[0, inf, inf, inf]

L5 min_coin_change: for total in range(1, amount + 1):
coins=[2]; amount=3; values={2}; dp=[0, inf, inf, inf]; total=1

L6 min_coin_change: for coin in values:
coins=[2]; amount=3; values={2}; dp=[0, inf, inf, inf]; total=1; coin=2

L5 min_coin_change: for total in range(1, amount + 1):
coins=[2]; amount=3; values={2}; dp=[0, inf, inf, inf]; total=2; coin=2

L8 min_coin_change: dp[total] = min(dp[total], dp[total - coin] + 1)
coins=[2]; amount=3; values={2}; dp=[0, inf, 1, inf]; total=2; coin=2

L5 min_coin_change: for total in range(1, amount + 1):
coins=[2]; amount=3; values={2}; dp=[0, inf, 1, inf]; total=3; coin=2
```

Result: -1. No exact odd amount is reachable.

Why the solution is correct

Each candidate corresponds to a legal last coin appended to an exact smaller amount. Every legal solution has such a last coin, so minimizing across them considers an optimum. The base represents taking no coins, and positive denominations guarantee acyclic dependencies.

Time and space

O(A*C) transition time and O(A+C) storage for amount A and C distinct denominations, plus denomination-set construction. This is pseudopolynomial in the numeric amount.

Common mistakes

A greedy largest-coin choice fails for arbitrary denominations, such as [1,3,4] at amount 6. Zero coins break the smaller-state rationale.

Reasoning card

State: dp[0]=0; an infinity sentinel marks amounts not yet known reachable. At amount x, every dp[x-c] read is already complete.

Invariant: dp[x] is the minimum coins required for exactly x.

Decision to rehearse: Build exact-amount answers from smaller amounts.

Finish: Convert an unreachable final state to -1

Transfer exercise

Store the last coin chosen for each reachable amount to reconstruct one minimum-coin combination.

229. Coin-Change Combination Count

Interview | Module 23

Problem

Count unordered combinations of positive integer coin denominations that sum to a nonnegative amount, allowing unlimited reuse. Duplicate denominations do not create distinct coin types. Return one for amount zero, including when no denominations exist. The order of chosen coins does not distinguish combinations.

Examples

Example 1. The four combinations are five ones; three ones plus two; one plus two twos; and five.

```
coin_combinations([1,2,5],5)
=> 4
```

Example 2. No exact combination exists.

```
coin_combinations([2],3)
=> 0
```

Example 3. The empty combination is counted once despite duplicate input types.

```
coin_combinations([1,1],0)
=> 1
```

Why this mechanism works

Make coin type the outer loop. After processing a type, ways[x] counts combinations using only types seen so far. Ascending amounts permit multiple copies of the current coin; retaining the existing ways[x] preserves combinations using none. This fixes a canonical type order and avoids counting permutations.

State and boundaries

Before a coin pass, ways uses earlier types. During its ascending pass, ways[x-coin] already allows the current type.

Protocol

1. Sort distinct denominations.

2. Initialize ways[0]=1 and other counts zero.

3. For each coin scan amounts from coin upward.

4. Add ways[amount-coin] into ways[amount].

5. Return the target count.

Visualization

```
coins [1,2], amount 3
after coin 1: ways [1,1,1,1]
after coin 2: ways [1,1,2,2]
3 = 1+1+1 or 1+2, not separately 2+1
```

Python implementation

```
def coin_combinations(coins, amount):
    ways = [0] * (amount + 1)
    ways[0] = 1
    for coin in sorted(set(coins)):
        for total in range(coin, amount + 1):
            ways[total] += ways[total - coin]
    return ways[amount]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
coin_combinations([1,2,5],5)

L2 coin_combinations: ways = [0] * (amount + 1)
coins=[1, 2, 5]; amount=5; ways=[0, 0, 0, 0, 0, 0]

L3 coin_combinations: ways[0] = 1
coins=[1, 2, 5]; amount=5; ways=[1, 0, 0, 0, 0, 0]

L4 coin_combinations: for coin in sorted(set(coins)):
coins=[1, 2, 5]; amount=5; ways=[1, 0, 0, 0, 0, 0]; coin=1

L5 coin_combinations: for total in range(coin, amount + 1):
coins=[1, 2, 5]; amount=5; ways=[1, 0, 0, 0, 0, 0]; coin=1; total=1

L6 coin_combinations: ways[total] += ways[total - coin]
coins=[1, 2, 5]; amount=5; ways=[1, 1, 0, 0, 0, 0]; coin=1; total=1

L5 coin_combinations: for total in range(coin, amount + 1):
coins=[1, 2, 5]; amount=5; ways=[1, 1, 0, 0, 0, 0]; coin=1; total=2

L6 coin_combinations: ways[total] += ways[total - coin]
coins=[1, 2, 5]; amount=5; ways=[1, 1, 1, 0, 0, 0]; coin=1; total=2

L5 coin_combinations: for total in range(coin, amount + 1):
coins=[1, 2, 5]; amount=5; ways=[1, 1, 1, 0, 0, 0]; coin=1; total=3

... 12 intermediate state changes omitted ...

L5 coin_combinations: for total in range(coin, amount + 1):
coins=[1, 2, 5]; amount=5; ways=[1, 1, 2, 2, 3, 1]; coin=2; total=5

L6 coin_combinations: ways[total] += ways[total - coin]
coins=[1, 2, 5]; amount=5; ways=[1, 1, 2, 2, 3, 3]; coin=2; total=5

L4 coin_combinations: for coin in sorted(set(coins)):
coins=[1, 2, 5]; amount=5; ways=[1, 1, 2, 2, 3, 3]; coin=5; total=5
```

```
L6 coin_combinations: ways[total] += ways[total - coin]
coins=[1, 2, 5]; amount=5; ways=[1, 1, 2, 2, 3, 4]; coin=5; total=5
```

Result: 4. The four combinations are five ones; three ones plus two; one plus two twos; and five.

```
coin_combinations([2],3)

L2 coin_combinations: ways = [0] * (amount + 1)
coins=[2]; amount=3; ways=[0, 0, 0, 0]

L3 coin_combinations: ways[0] = 1
coins=[2]; amount=3; ways=[1, 0, 0, 0]

L4 coin_combinations: for coin in sorted(set(coins)):
coins=[2]; amount=3; ways=[1, 0, 0, 0]; coin=2

L5 coin_combinations: for total in range(coin, amount + 1):
coins=[2]; amount=3; ways=[1, 0, 0, 0]; coin=2; total=2

L6 coin_combinations: ways[total] += ways[total - coin]
coins=[2]; amount=3; ways=[1, 0, 1, 0]; coin=2; total=2

L5 coin_combinations: for total in range(coin, amount + 1):
coins=[2]; amount=3; ways=[1, 0, 1, 0]; coin=2; total=3
```

Result: 0. No exact combination exists.

Why the solution is correct

Combinations split into those using zero copies of the current coin and those using at least one. The old state counts the former; removing one current coin maps the latter to the already-updated smaller amount. These disjoint classes cover every combination exactly once.

Time and space

O(C*A+C log C) arithmetic operations and O(A+C) integer storage; exact counts can require multiple machine words.

Common mistakes

Putting amount outside and coins inside counts different orders separately. Descending amount order would restrict each coin type to one use.

Reasoning card

State: Before a coin pass, ways uses earlier types. During its ascending pass, ways[x-coin] already allows the current type.

Invariant: ways[x] uses only coin types processed so far.

Decision to rehearse: Loop coin types outside to avoid counting different orders separately.

Finish: Return the target count

Transfer exercise

Count ordered coin sequences by intentionally reversing the loop hierarchy, and demonstrate the changed answer on amount 3 with coins 1 and 2.

230. Longest Increasing Subsequence

Interview | Module 23

Problem

Return the length of the longest strictly increasing subsequence of nums. A subsequence preserves index order but may skip positions. Equal values cannot extend it. Empty input returns zero. First implement the quadratic ending-at-index DP, then derive the binary-search tails optimization.

Examples

Example 1. The direct DP finds a length-four subsequence such as 2,3,7,18.

```
lis_quadratic([10,9,2,5,3,7,101,18])
=> 4
```

Example 2. Lower bound prevents equal values from extending the length.

```
lis_length([3,1,2,2,4])
=> 3
```

Example 3. Both formulations use zero for empty input.

```
(lis_quadratic([]),lis_length([]))
=> (0, 0)
```

Why this mechanism works

For each endpoint i, consider every earlier smaller value j and extend the best subsequence ending there. This yields a direct, inspectable recurrence. The optimized view groups subsequences by length instead: retain the smallest possible ending value for each length, because a smaller tail is at least as easy to extend later.

State and boundaries

dp[i] is the LIS length ending exactly at i. In the optimized version, tails[length-1] is the smallest achievable tail for that subsequence length among processed positions.

Protocol

1. Initialize each ending-at-index length to one.

2. Extend from every earlier strictly smaller value.

3. Take the maximum for the reference DP.

4. For optimized search, find the first tail >=current.

5. Replace it, or append when no such tail exists.

Visualization

```
nums [3,1,2,2,4]
tails [3] -> [1] -> [1,2] -> [1,2] -> [1,2,4]
the repeated 2 replaces; it does not extend
```

From the recurrence to the optimized solution

Start with the state you can prove: dp[i] answers a question about one fixed endpoint. For [3,1,2,2,4], the table becomes [1,1,2,2,3]. The final answer is max(dp), because the optimum may end anywhere.

Now ask which information an unseen future value needs. For subsequences of equal length, only their smallest ending value matters for future extension. A length-two subsequence ending at 2 is at least as useful as one ending at 8. This is a dominance argument: discard the worse tail, not a potentially larger length.

The tails array is strictly increasing. A length-(k+1) subsequence contains a length-k prefix with a smaller endpoint, so the minimum tail for length k is smaller than the minimum tail for length k+1. Binary search is therefore valid. The first tail >=x is the length whose ending value x can improve; every earlier tail is strictly smaller than x and can be extended by it. Equality replaces instead of appending.

Remember the boundary with the earlier binary-search chapter: bisect_left finds the first index where tail>=x. It is the same false-then-true predicate, expressed through a standard-library function. If every tail is smaller, the returned index is len(tails), the valid insertion boundary for a new longest length.

Do not infer that the current tails entries belong to the same historical route. They summarize separate best endpoints. To recover a route, preserve predecessor indices when each input value changes a tail.

Python implementation

```
from bisect import bisect_left

def lis_quadratic(nums):
    dp = [1] * len(nums)
    for i in range(len(nums)):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp, default=0)

def lis_length(nums):
    tails = []
    for value in nums:
        i = bisect_left(tails, value)
        if i == len(tails):
            tails.append(value)
        else:
            tails[i] = value
    return len(tails)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
lis_quadratic([10,9,2,5,3,7,101,18])

L4 lis_quadratic: dp = [1] * len(nums)
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 1, 1, 1, 1, 1]

L5 lis_quadratic: for i in range(len(nums)):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 1, 1, 1, 1, 1]; i=0

L5 lis_quadratic: for i in range(len(nums)):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 1, 1, 1, 1, 1]; i=1

L6 lis_quadratic: for j in range(i):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 1, 1, 1, 1, 1]; i=1; j=0

L5 lis_quadratic: for i in range(len(nums)):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 1, 1, 1, 1, 1]; i=2; j=0

L6 lis_quadratic: for j in range(i):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 1, 1, 1, 1, 1]; i=2; j=1

L5 lis_quadratic: for i in range(len(nums)):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 1, 1, 1, 1, 1]; i=3; j=1

L6 lis_quadratic: for j in range(i):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 1, 1, 1, 1, 1]; i=3; j=0

... 34 intermediate state changes omitted ...

L6 lis_quadratic: for j in range(i):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 2, 2, 3, 4, 3]; i=7; j=4

L6 lis_quadratic: for j in range(i):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 2, 2, 3, 4, 3]; i=7; j=5
```

```
L8 lis_quadratic: dp[i] = max(dp[i], dp[j] + 1)
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 2, 2, 3, 4, 4]; i=7; j=5

L6 lis_quadratic: for j in range(i):
nums=[10, 9, 2, 5, 3, 7, 101, 18]; dp=[1, 1, 1, 2, 2, 3, 4, 4]; i=7; j=6
```

Result: 4. The direct DP finds a length-four subsequence such as 2,3,7,18.

```
lis_length([3,1,2,2,4])

L12 lis_length: tails = []
nums=[3, 1, 2, 2, 4]; tails=[]

L13 lis_length: for value in nums:
nums=[3, 1, 2, 2, 4]; tails=[]; value=3

L14 lis_length: i = bisect_left(tails, value)
nums=[3, 1, 2, 2, 4]; tails=[]; value=3; i=0

L16 lis_length: tails.append(value)
nums=[3, 1, 2, 2, 4]; tails=[3]; value=3; i=0

L13 lis_length: for value in nums:
nums=[3, 1, 2, 2, 4]; tails=[3]; value=1; i=0

L18 lis_length: tails[i] = value
nums=[3, 1, 2, 2, 4]; tails=[1]; value=1; i=0

L13 lis_length: for value in nums:
nums=[3, 1, 2, 2, 4]; tails=[1]; value=2; i=0

L14 lis_length: i = bisect_left(tails, value)
nums=[3, 1, 2, 2, 4]; tails=[1]; value=2; i=1

L16 lis_length: tails.append(value)
nums=[3, 1, 2, 2, 4]; tails=[1, 2]; value=2; i=1

L13 lis_length: for value in nums:
nums=[3, 1, 2, 2, 4]; tails=[1, 2]; value=4; i=1

L14 lis_length: i = bisect_left(tails, value)
nums=[3, 1, 2, 2, 4]; tails=[1, 2]; value=4; i=2

L16 lis_length: tails.append(value)
nums=[3, 1, 2, 2, 4]; tails=[1, 2, 4]; value=4; i=2
```

Result: 3. Lower bound prevents equal values from extending the length.

Why the solution is correct

The endpoint recurrence considers every possible penultimate index, so it gives exact ending lengths. For tails, replacing the first value >=current preserves feasible lengths while improving their extension opportunities; appending means current extends the longest known length. The tails length therefore equals the optimum, although its entries need not form one actual subsequence.

Time and space

Reference: O(n^2) time and O(n) space. Optimized: O(n log(n+1)) time and O(n) space. Both leave input unchanged.

Common mistakes

Using upper_bound instead of lower_bound changes strict increasing to nondecreasing. Returning tails as the actual subsequence is not justified by its invariant.

Reasoning card

State: dp[i] is the LIS length ending exactly at i. In the optimized version, tails[length-1] is the smallest achievable tail for that subsequence length among processed positions.

Invariant: dp[i] is the best increasing subsequence ending exactly at i.

Decision to rehearse: Start with O(n^2) ending-at-i state, then study tails/binary-search optimization.

Finish: Replace it, or append when no such tail exists

Transfer exercise

Reconstruct an actual LIS using predecessor indices and the input index associated with each current tail.

Module 24
Dynamic Programming II: Grids, Knapsack, and Strings

Give every DP axis a precise meaning and derive transitions from neighbouring or smaller states.

Every DP axis needs a sentence. Grid coordinates, item/capacity pairs, two prefix lengths and interval endpoints encode different dependency graphs. Loop order is a topological order of those states. In one-dimensional knapsack compression, reversing capacity order is what preserves the previous item layer.

The ten-step progression

231. Unique Grid Paths

232. Minimum Path Sum

233. Unique Paths with Obstacles

234. Zero-One Knapsack

235. Partition Equal Subset Sum

236. Target Sum Ways

237. Longest Common Subsequence

238. Edit Distance

239. Interleaving String

240. Longest Palindromic Subsequence

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

231. Unique Grid Paths

Foundation | Module 24

Problem

Count paths from the top-left cell to the bottom-right cell of a rows-by-cols grid, moving only one cell right or down. Dimensions are nonnegative. Return zero if either dimension is zero; a one-cell grid has one zero-move path. Return an exact integer.

Examples

Example 1. The final table entry counts six right/down routes.

```
unique_paths(3,3)
=> 6
```

Example 2. Taking no moves is one complete path.

```
unique_paths(1,1)
=> 1
```

Example 3. There is no starting cell.

```
unique_paths(0,4)
=> 0
```

Why this mechanism works

Name both axes before filling the table: dp[r][c] counts paths ending at that cell. The final move comes from above or from the left, and these route classes are disjoint. Boundary cells have only one predecessor; the origin needs an explicit seed of one.

State and boundaries

dp[r][c] is the complete count for exactly cell (r,c). Row-major order completes its top and left predecessors before use.

Protocol

1. Handle zero dimensions.
2. Allocate a zero table and seed the origin with one.
3. Scan cells in row-major order, skipping the origin.
4. Add the top count if present and the left count if present.
5. Return the destination count.

Visualization

```
3x3 path-count table
1 1 1
1 2 3
1 3 6
each interior value = top + left
```

Python implementation

```
def unique_paths(rows, cols):
    if rows == 0 or cols == 0:
        return 0
    dp = [[0] * cols for _ in range(rows)]
    dp[0][0] = 1
    for r in range(rows):
        for c in range(cols):
            if r == c == 0:
                continue
            dp[r][c] = (dp[r - 1][c] if r else 0) + (dp[r][c - 1] if c else 0)
    return dp[-1][-1]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
unique_paths(3,3)

L2 unique_paths: if rows == 0 or cols == 0:
rows=3; cols=3

L4 unique_paths: dp = [[0] * cols for _ in range(rows)]
rows=3; cols=3; _=0

L4 unique_paths: dp = [[0] * cols for _ in range(rows)]
rows=3; cols=3; _=1

L4 unique_paths: dp = [[0] * cols for _ in range(rows)]
rows=3; cols=3; _=2

L4 unique_paths: dp = [[0] * cols for _ in range(rows)]
rows=3; cols=3; dp=[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

L5 unique_paths: dp[0][0] = 1
rows=3; cols=3; dp=[[1, 0, 0], [0, 0, 0], [0, 0, 0]]

L6 unique_paths: for r in range(rows):
rows=3; cols=3; dp=[[1, 0, 0], [0, 0, 0], [0, 0, 0]]; r=0

L7 unique_paths: for c in range(cols):
rows=3; cols=3; dp=[[1, 0, 0], [0, 0, 0], [0, 0, 0]]; r=0; c=0

... 14 intermediate state changes omitted ...

L7 unique_paths: for c in range(cols):
rows=3; cols=3; dp=[[1, 1, 1], [1, 2, 3], [1, 0, 0]]; r=2; c=1

L10 unique_paths: dp[r][c] = (dp[r - 1][c] if r else 0) + (dp[r][c - 1] if c else 0)
rows=3; cols=3; dp=[[1, 1, 1], [1, 2, 3], [1, 3, 0]]; r=2; c=1
```

```
L7 unique_paths: for c in range(cols):
rows=3; cols=3; dp=[[1, 1, 1], [1, 2, 3], [1, 3, 0]]; r=2; c=2

L10 unique_paths: dp[r][c] = (dp[r - 1][c] if r else 0) + (dp[r][c - 1] if c else 0)
rows=3; cols=3; dp=[[1, 1, 1], [1, 2, 3], [1, 3, 6]]; r=2; c=2
```

Result: 6. The final table entry counts six right/down routes.

```
unique_paths(1,1)

L2 unique_paths: if rows == 0 or cols == 0:
rows=1; cols=1

L4 unique_paths: dp = [[0] * cols for _ in range(rows)]
rows=1; cols=1; _=0

L4 unique_paths: dp = [[0] * cols for _ in range(rows)]
rows=1; cols=1; dp=[[0]]

L5 unique_paths: dp[0][0] = 1
rows=1; cols=1; dp=[[1]]

L6 unique_paths: for r in range(rows):
rows=1; cols=1; dp=[[1]]; r=0

L7 unique_paths: for c in range(cols):
rows=1; cols=1; dp=[[1]]; r=0; c=0
```

Result: 1. Taking no moves is one complete path.

Why the solution is correct

The origin's empty path is the base. Every path into another cell ends with exactly one of the two permitted moves. Removing that move maps it uniquely to a path counted by the corresponding predecessor. Summing their complete counts therefore counts every route once.

Time and space

O(rows*cols) additions and table entries. Exact counts can require many bits; the table is pedagogically explicit rather than space-compressed.

Common mistakes

Seeding every cell with one and also adding predecessors overcounts. A zero-sized grid has no origin to seed.

Reasoning card

State: dp[r][c] is the complete count for exactly cell (r,c). Row-major order completes its top and left predecessors before use.

Invariant: dp[r][c] counts paths to exactly that cell.

Decision to rehearse: Each cell receives paths from top and left.

Finish: Return the destination count

Transfer exercise

Compress the table to one row, explaining which old and new row values each update reads.

232. Minimum Path Sum

Foundation | Module 24

Problem

Return the minimum sum of visited cell values on a right/down path from the top-left to the bottom-right of a rectangular integer grid. Include both endpoints. Values may be negative because movement is acyclic. Return None for an empty grid or an empty row. Do not mutate the grid.

Examples

Example 1. The cheapest route follows the top row and then the right column.

```
min_path_sum([[1,3,1],[1,5,1],[4,2,1]])
=> 7
```

Example 2. Negative cells are valid in this acyclic movement graph.

```
min_path_sum([[2,-5],[3,1]])
=> -2
```

Example 3. No endpoint-to-endpoint path exists.

```
min_path_sum([])
=> None
```

Why this mechanism works

Replace path counting's sum over alternatives with a minimum over costs. The final move must come from above or left, then pay the current cell. Use infinity for absent predecessors so a boundary cannot take a nonexistent free route. Seed the origin with its actual value.

State and boundaries

dp[r][c] is the cheapest cost including cell (r,c). Both candidate predecessor states are already complete in row-major order.

Protocol

1. Handle empty input.
2. Initialize a cost table and set the origin.
3. For each other cell read top/left costs, using infinity when absent.
4. Add the current cell to their minimum.
5. Return the destination cost.

Visualization

```
grid:       best costs:
1 3 1       1 4 5
1 5 1       2 7 6
4 2 1       6 8 7
```

Python implementation

```
def min_path_sum(grid):
    if not grid or not grid[0]:
        return None
    rows, cols = len(grid), len(grid[0])
    dp = [[0] * cols for _ in range(rows)]
    dp[0][0] = grid[0][0]
    for r in range(rows):
        for c in range(cols):
            if r == c == 0:
                continue
            top = dp[r - 1][c] if r else float('inf')
            left = dp[r][c - 1] if c else float('inf')
            dp[r][c] = grid[r][c] + min(top, left)
    return dp[-1][-1]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
min_path_sum([[1,3,1],[1,5,1],[4,2,1]])

L2 min_path_sum: if not grid or not grid[0]:
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]

L4 min_path_sum: rows, cols = len(grid), len(grid[0])
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3

L5 min_path_sum: dp = [[0] * cols for _ in range(rows)]
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; _=0

L5 min_path_sum: dp = [[0] * cols for _ in range(rows)]
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; _=1

L5 min_path_sum: dp = [[0] * cols for _ in range(rows)]
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; _=2

L5 min_path_sum: dp = [[0] * cols for _ in range(rows)]
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; dp=[[0, 0, 0], [0, 0, 0], [0, 0, 0]]

L6 min_path_sum: dp[0][0] = grid[0][0]
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; dp=[[1, 0, 0], [0, 0, 0], [0, 0, 0]]

L7 min_path_sum: for r in range(rows):
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; dp=[[1, 0, 0], [0, 0, 0], [0, 0, 0]];
    r=0

... 30 intermediate state changes omitted ...

L8 min_path_sum: for c in range(cols):
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; dp=[[1, 4, 5], [2, 7, 6], [6, 8, 0]];
```

```
    r=2; c=2; top=7; left=6

L11 min_path_sum: top = dp[r - 1][c] if r else float('inf')
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; dp=[[1, 4, 5], [2, 7, 6], [6, 8, 0]];
    r=2; c=2; top=6; left=6

L12 min_path_sum: left = dp[r][c - 1] if c else float('inf')
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; dp=[[1, 4, 5], [2, 7, 6], [6, 8, 0]];
    r=2; c=2; top=6; left=8

L13 min_path_sum: dp[r][c] = grid[r][c] + min(top, left)
grid=[[1, 3, 1], [1, 5, 1], [4, 2, 1]]; rows=3; cols=3; dp=[[1, 4, 5], [2, 7, 6], [6, 8, 7]];
    r=2; c=2; top=6; left=8
```

Result: 7. The cheapest route follows the top row and then the right column.

```
min_path_sum([[2,-5],[3,1]])

L2 min_path_sum: if not grid or not grid[0]:
grid=[[2, -5], [3, 1]]

L4 min_path_sum: rows, cols = len(grid), len(grid[0])
grid=[[2, -5], [3, 1]]; rows=2; cols=2

L5 min_path_sum: dp = [[0] * cols for _ in range(rows)]
grid=[[2, -5], [3, 1]]; rows=2; cols=2; _=0

L5 min_path_sum: dp = [[0] * cols for _ in range(rows)]
grid=[[2, -5], [3, 1]]; rows=2; cols=2; _=1

L5 min_path_sum: dp = [[0] * cols for _ in range(rows)]
grid=[[2, -5], [3, 1]]; rows=2; cols=2; dp=[[0, 0], [0, 0]]

L6 min_path_sum: dp[0][0] = grid[0][0]
grid=[[2, -5], [3, 1]]; rows=2; cols=2; dp=[[2, 0], [0, 0]]

L7 min_path_sum: for r in range(rows):
grid=[[2, -5], [3, 1]]; rows=2; cols=2; dp=[[2, 0], [0, 0]]; r=0

L8 min_path_sum: for c in range(cols):
grid=[[2, -5], [3, 1]]; rows=2; cols=2; dp=[[2, 0], [0, 0]]; r=0; c=0

... 9 intermediate state changes omitted ...

L8 min_path_sum: for c in range(cols):
grid=[[2, -5], [3, 1]]; rows=2; cols=2; dp=[[2, -3], [5, 0]]; r=1; c=1; top=2; left=inf

L11 min_path_sum: top = dp[r - 1][c] if r else float('inf')
grid=[[2, -5], [3, 1]]; rows=2; cols=2; dp=[[2, -3], [5, 0]]; r=1; c=1; top=-3; left=inf

L12 min_path_sum: left = dp[r][c - 1] if c else float('inf')
grid=[[2, -5], [3, 1]]; rows=2; cols=2; dp=[[2, -3], [5, 0]]; r=1; c=1; top=-3; left=5

L13 min_path_sum: dp[r][c] = grid[r][c] + min(top, left)
grid=[[2, -5], [3, 1]]; rows=2; cols=2; dp=[[2, -3], [5, -2]]; r=1; c=1; top=-3; left=5
```

Result: -2. Negative cells are valid in this acyclic movement graph.

Why the solution is correct

Every route into a nonorigin cell has one of the two legal predecessor cells. Replacing its prefix with the cheapest route to that predecessor cannot increase cost, so the recurrence considers an optimum route. Negative values cause no cycle or repeated relaxation because coordinates only increase.

Time and space

O(R*C) time and O(R*C) auxiliary table space.

Common mistakes

Using zero for missing predecessors permits paths entering from outside the grid. Omitting the origin's value changes the cost contract.

Reasoning card

State: dp[r][c] is the cheapest cost including cell (r,c). Both candidate predecessor states are already complete in row-major order.

Invariant: Each state is the cheapest cost to reach its cell.

Decision to rehearse: Current cost equals cell value plus min(top,left).

Finish: Return the destination cost

Transfer exercise

Store the winning predecessor direction at each cell and reconstruct a minimum-cost route.

233. Unique Paths with Obstacles

Foundation | Module 24

Problem

Count right/down paths through a rectangular zero/one grid, where one means blocked and zero means open. Paths run from top-left to bottom-right. A blocked endpoint yields zero; empty input yields zero. Do not mutate the obstacle grid.

Examples

Example 1. Only the two routes around the central block remain.

```
paths_with_obstacles([[0,0,0],[0,1,0],[0,0,0]])
=> 2
```

Example 2. The blocked origin erases the initial path seed.

```
paths_with_obstacles([[1]])
=> 0
```

Example 3. An open singleton has one zero-move route.

```
paths_with_obstacles([[0]])
=> 1
```

Why this mechanism works

Use the same predecessor recurrence as unique paths, but a blocked cell contributes no paths regardless of its neighbors. In a one-row table, dp[c] still holds the old row's count from above, while dp[c-1] already holds the current row's count from the left. Clearing an obstacle must happen before combining anything.

State and boundaries

Before updating column c, dp[c] is the top predecessor count; after updating, it is the current cell count. The left entry is already current-row state.

Protocol

1. Handle empty input.
2. Seed dp[0]=1 and other columns zero.
3. Scan rows left to right.
4. Set dp[c]=0 at an obstacle.
5. Otherwise add dp[c-1] when c>0.
6. Return the last column count.

Visualization

```
0 0 0
0 1 0   center count is forced to zero
0 0 0   two boundary routes survive
```

Python implementation

```
def paths_with_obstacles(grid):
    if not grid or not grid[0]:
        return 0
    dp = [0] * len(grid[0])
    dp[0] = 1
    for row in grid:
        for c, blocked in enumerate(row):
            if blocked:
                dp[c] = 0
            elif c:
                dp[c] += dp[c - 1]
    return dp[-1]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
paths_with_obstacles([[0,0,0],[0,1,0],[0,0,0]])

L2 paths_with_obstacles: if not grid or not grid[0]:
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]

L4 paths_with_obstacles: dp = [0] * len(grid[0])
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[0, 0, 0]

L5 paths_with_obstacles: dp[0] = 1
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 0, 0]

L6 paths_with_obstacles: for row in grid:
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 0, 0]; row=[0, 0, 0]

L7 paths_with_obstacles: for c, blocked in enumerate(row):
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 0, 0]; row=[0, 0, 0]; c=0; blocked=0

L7 paths_with_obstacles: for c, blocked in enumerate(row):
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 0, 0]; row=[0, 0, 0]; c=1; blocked=0

L11 paths_with_obstacles: dp[c] += dp[c - 1]
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 1, 0]; row=[0, 0, 0]; c=1; blocked=0

L7 paths_with_obstacles: for c, blocked in enumerate(row):
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 1, 0]; row=[0, 0, 0]; c=2; blocked=0

... 8 intermediate state changes omitted ...

L7 paths_with_obstacles: for c, blocked in enumerate(row):
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 0, 1]; row=[0, 0, 0]; c=1; blocked=0

L11 paths_with_obstacles: dp[c] += dp[c - 1]
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 1, 1]; row=[0, 0, 0]; c=1; blocked=0

L7 paths_with_obstacles: for c, blocked in enumerate(row):
```

```
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 1, 1]; row=[0, 0, 0]; c=2; blocked=0

L11 paths_with_obstacles: dp[c] += dp[c - 1]
grid=[[0, 0, 0], [0, 1, 0], [0, 0, 0]]; dp=[1, 1, 2]; row=[0, 0, 0]; c=2; blocked=0
```

Result: 2. Only the two routes around the central block remain.

```
paths_with_obstacles([[1]])

L2 paths_with_obstacles: if not grid or not grid[0]:
grid=[[1]]

L4 paths_with_obstacles: dp = [0] * len(grid[0])
grid=[[1]]; dp=[0]

L5 paths_with_obstacles: dp[0] = 1
grid=[[1]]; dp=[1]

L6 paths_with_obstacles: for row in grid:
grid=[[1]]; dp=[1]; row=[1]

L7 paths_with_obstacles: for c, blocked in enumerate(row):
grid=[[1]]; dp=[1]; row=[1]; c=0; blocked=1

L9 paths_with_obstacles: dp[c] = 0
grid=[[1]]; dp=[0]; row=[1]; c=0; blocked=1
```

Result: 0. The blocked origin erases the initial path seed.

Why the solution is correct

The seed represents one path entering the open origin and is erased if that origin is blocked. At each open cell, old-top plus new-left counts all legal arrivals. At a blocked cell, zero correctly prevents every route from continuing through it. Row-major updates preserve these meanings.

Time and space

O(R*C) additions and O(C) integer storage; exact counts may exceed machine-word size.

Common mistakes

Skipping an obstacle without clearing dp[c] leaves stale paths from above. Updating right to left would read the wrong version of the left predecessor.

Reasoning card

State: Before updating column c, dp[c] is the top predecessor count; after updating, it is the current cell count. The left entry is already current-row state.

Invariant: Blocked states are unreachable.

Decision to rehearse: Set obstacle states before combining predecessors.

Finish: Return the last column count

Transfer exercise

Allow at most one obstacle removal by adding a second state for whether that removal has already been used.

234. Zero-One Knapsack

Foundation | Module 24

Problem

Given equally sized weight and value lists and nonnegative integer capacity, return the maximum total value of a subset whose weight is at most capacity. Weights are positive integers; values are nonnegative. Each item index may be used at most once, and the empty subset is allowed.

Examples

Example 1. The first two items fit together for value seven.

```
zero_one_knapsack([2,3,4],[3,4,5],5)
=> 7
```

Example 2. The sole item cannot be used twice.

```
zero_one_knapsack([2],[3],4)
=> 3
```

Example 3. The empty subset is the only choice.

```
zero_one_knapsack([],[],5)
=> 0
```

Why this mechanism works

For each item and capacity, compare skipping it with taking it plus the best smaller-capacity result from before this item. A one-dimensional table can preserve that older layer by scanning capacities downward. Ascending order would read a value already improved by the same item and permit repeated use.

State and boundaries

Before an item pass, dp[c] is the best value using earlier items within capacity c. Descending updates keep dp[c-weight] in that previous-item layer.

Protocol

1. Initialize every capacity to zero.

2. For each item scan capacity down to its weight.

3. Set dp[c]=max(dp[c],dp[c-weight]+value).

4. Return dp[capacity].

Visualization

```
one item weight 2,value 3; capacity 4
descending: dp[4] reads old dp[2]=0 -> 3
ascending would make dp[2]=3, then dp[4]=6 illegally
```

Python implementation

```
def zero_one_knapsack(weights, values, capacity):
    dp = [0] * (capacity + 1)
    for weight, value in zip(weights, values):
        for c in range(capacity, weight - 1, -1):
            dp[c] = max(dp[c], dp[c - weight] + value)
    return dp[capacity]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
zero_one_knapsack([2,3,4],[3,4,5],5)

L2 zero_one_knapsack: dp = [0] * (capacity + 1)
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 0, 0, 0, 0]

L3 zero_one_knapsack: for weight, value in zip(weights, values):
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 0, 0, 0, 0]; weight=2; value=3

L4 zero_one_knapsack: for c in range(capacity, weight - 1, -1):
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 0, 0, 0, 0]; weight=2; value=3; c=5

L5 zero_one_knapsack: dp[c] = max(dp[c], dp[c - weight] + value)
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 0, 0, 0, 3]; weight=2; value=3; c=5

L4 zero_one_knapsack: for c in range(capacity, weight - 1, -1):
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 0, 0, 0, 3]; weight=2; value=3; c=4

L5 zero_one_knapsack: dp[c] = max(dp[c], dp[c - weight] + value)
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 0, 0, 3, 3]; weight=2; value=3; c=4

L4 zero_one_knapsack: for c in range(capacity, weight - 1, -1):
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 0, 0, 3, 3]; weight=2; value=3; c=3

L5 zero_one_knapsack: dp[c] = max(dp[c], dp[c - weight] + value)
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 0, 3, 3, 3]; weight=2; value=3; c=3

... 9 intermediate state changes omitted ...

L3 zero_one_knapsack: for weight, value in zip(weights, values):
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 3, 4, 4, 7]; weight=4; value=5; c=3

L4 zero_one_knapsack: for c in range(capacity, weight - 1, -1):
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 3, 4, 4, 7]; weight=4; value=5; c=5

L4 zero_one_knapsack: for c in range(capacity, weight - 1, -1):
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 3, 4, 4, 7]; weight=4; value=5; c=4

L5 zero_one_knapsack: dp[c] = max(dp[c], dp[c - weight] + value)
weights=[2, 3, 4]; values=[3, 4, 5]; capacity=5; dp=[0, 0, 3, 4, 5, 7]; weight=4; value=5; c=4
```

Result: 7. The first two items fit together for value seven.

```
zero_one_knapsack([2],[3],4)

L2 zero_one_knapsack: dp = [0] * (capacity + 1)
weights=[2]; values=[3]; capacity=4; dp=[0, 0, 0, 0, 0]

L3 zero_one_knapsack: for weight, value in zip(weights, values):
weights=[2]; values=[3]; capacity=4; dp=[0, 0, 0, 0, 0]; weight=2; value=3

L4 zero_one_knapsack: for c in range(capacity, weight - 1, -1):
weights=[2]; values=[3]; capacity=4; dp=[0, 0, 0, 0, 0]; weight=2; value=3; c=4

L5 zero_one_knapsack: dp[c] = max(dp[c], dp[c - weight] + value)
weights=[2]; values=[3]; capacity=4; dp=[0, 0, 0, 0, 3]; weight=2; value=3; c=4

L4 zero_one_knapsack: for c in range(capacity, weight - 1, -1):
weights=[2]; values=[3]; capacity=4; dp=[0, 0, 0, 0, 3]; weight=2; value=3; c=3

L5 zero_one_knapsack: dp[c] = max(dp[c], dp[c - weight] + value)
weights=[2]; values=[3]; capacity=4; dp=[0, 0, 0, 3, 3]; weight=2; value=3; c=3

L4 zero_one_knapsack: for c in range(capacity, weight - 1, -1):
weights=[2]; values=[3]; capacity=4; dp=[0, 0, 0, 3, 3]; weight=2; value=3; c=2

L5 zero_one_knapsack: dp[c] = max(dp[c], dp[c - weight] + value)
weights=[2]; values=[3]; capacity=4; dp=[0, 0, 3, 3, 3]; weight=2; value=3; c=2
```

Result: 3. The sole item cannot be used twice.

Why the solution is correct

A legal optimum either excludes the current item or includes it once with an earlier-item solution fitting the remaining capacity. The recurrence compares exactly those cases. Descending iteration ensures the take branch reads a state that cannot already contain this item, preserving zero-one usage.

Time and space

O(N*C) time and O(C) auxiliary storage for N items and capacity C. This is pseudopolynomial in capacity, not polynomial in its bit length.

Common mistakes

Ascending capacities change the problem to unlimited item reuse. Initializing unreachable exact weights is unnecessary here because states mean at-most capacity, not exact fill.

Reasoning card

State: Before an item pass, dp[c] is the best value using earlier items within capacity c. Descending updates keep dp[c-weight] in that previous-item layer.

Invariant: dp[c] uses only items processed so far.

Decision to rehearse: Choose skip versus take; reverse one-dimensional capacity iteration.

Finish: Return dp[capacity]

Transfer exercise

Require exact total weight rather than at-most capacity and revise unreachable-state initialization accordingly.

235. Partition Equal Subset Sum

Bridge | Module 24

Problem

Return whether a list of nonnegative integers can be divided into two disjoint subsets covering every input occurrence and having equal sums. Empty subsets are permitted, so [] returns true. Duplicate and zero values are distinct occurrences. Do not mutate nums.

Examples

Example 1. A subset reaches half the total.

```
equal_partition([1,5,11,5])
=> True
```

Example 2. The even total alone does not guarantee a reachable half.

```
equal_partition([1,2,5])
=> False
```

Example 3. Two empty subsets both sum to zero.

```
equal_partition([])
=> True
```

Why this mechanism works

Equal halves must each sum to half the total, so an odd total is impossible. For an even total, the problem is zero-one subset-sum reachability. A boolean capacity table and descending updates ensure each occurrence is used at most once; the complement automatically forms the other half.

State and boundaries

reachable[s] means some subset of processed occurrences sums exactly to s. reachable[0] is always true via the empty subset.

Protocol

1. Compute total and reject odd parity.
2. Set target=total//2.
3. Initialize only sum zero reachable.
4. For each value update sums downward with reachable[s-value].
5. Return target reachability.

Visualization

```
[1,5,11,5], total 22, target 11
subset [11] and complement [1,5,5]
one reachable half is sufficient evidence
```

Python implementation

```
def equal_partition(nums):
    total = sum(nums)
    if total % 2:
        return False
    target = total // 2
    reachable = [False] * (target + 1)
    reachable[0] = True
    for value in nums:
        for amount in range(target, value - 1, -1):
            reachable[amount] = reachable[amount] or reachable[amount - value]
    return reachable[target]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
equal_partition([1,5,11,5])

L2 equal_partition: total = sum(nums)
nums=[1, 5, 11, 5]; total=22

L5 equal_partition: target = total // 2
nums=[1, 5, 11, 5]; total=22; target=11

L6 equal_partition: reachable = [False] * (target + 1)
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[False, False, False, False, False, False,
    False, False, ...]

L7 equal_partition: reachable[0] = True
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, False, False, False, False, False,
    False, False, ...]

L8 equal_partition: for value in nums:
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, False, False, False, False, False,
    False, False, ...]; value=1

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, False, False, False, False, False,
    False, False, ...]; value=1; amount=11

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, False, False, False, False, False,
    False, False, ...]; value=1; amount=10

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, False, False, False, False, False,
    False, False, ...]; value=1; amount=9

... 24 intermediate state changes omitted ...

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, True, False, False, False, True,
```

```
    True, False, ...]; value=5; amount=8

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, True, False, False, False, True,
    True, False, ...]; value=5; amount=7

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, True, False, False, False, True,
    True, False, ...]; value=5; amount=6

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 5, 11, 5]; total=22; target=11; reachable=[True, True, False, False, False, True,
    True, False, ...]; value=5; amount=5
```

Result: True. A subset reaches half the total.

```
equal_partition([1,2,5])

L2 equal_partition: total = sum(nums)
nums=[1, 2, 5]; total=8

L5 equal_partition: target = total // 2
nums=[1, 2, 5]; total=8; target=4

L6 equal_partition: reachable = [False] * (target + 1)
nums=[1, 2, 5]; total=8; target=4; reachable=[False, False, False, False, False]

L7 equal_partition: reachable[0] = True
nums=[1, 2, 5]; total=8; target=4; reachable=[True, False, False, False, False]

L8 equal_partition: for value in nums:
nums=[1, 2, 5]; total=8; target=4; reachable=[True, False, False, False, False]; value=1

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 2, 5]; total=8; target=4; reachable=[True, False, False, False, False]; value=1;
    amount=4

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 2, 5]; total=8; target=4; reachable=[True, False, False, False, False]; value=1;
    amount=3

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 2, 5]; total=8; target=4; reachable=[True, False, False, False, False]; value=1;
    amount=2

... 5 intermediate state changes omitted ...

L10 equal_partition: reachable[amount] = reachable[amount] or reachable[amount - value]
nums=[1, 2, 5]; total=8; target=4; reachable=[True, True, False, True, False]; value=2;
    amount=3

L9 equal_partition: for amount in range(target, value - 1, -1):
nums=[1, 2, 5]; total=8; target=4; reachable=[True, True, False, True, False]; value=2;
    amount=2

L10 equal_partition: reachable[amount] = reachable[amount] or reachable[amount - value]
nums=[1, 2, 5]; total=8; target=4; reachable=[True, True, True, True, False]; value=2;
    amount=2

L8 equal_partition: for value in nums:
nums=[1, 2, 5]; total=8; target=4; reachable=[True, True, True, True, False]; value=5;
    amount=2
```

Result: False. The even total alone does not guarantee a reachable half.

Why the solution is correct

Any equal partition supplies a subset of sum total/2, and any such subset leaves a complement of the same sum. Each update either skips the occurrence or includes it once using an earlier-layer sum, so reachable tracks exactly all possible subset sums.

Time and space

O(n*T) time and O(T) auxiliary storage for target T=total/2; pseudopolynomial in the numeric sum.

Common mistakes

Ascending sums can reuse an occurrence. Negative input values invalidate the nonnegative bounded-capacity representation.

Reasoning card

State: reachable[s] means some subset of processed occurrences sums exactly to s. reachable[0] is always true via the empty subset.

Invariant: Reachable sums reflect processed items without reuse.

Decision to rehearse: Reduce to reachability of total/2.

Finish: Return target reachability

Transfer exercise

Use a bitset with bits |= bits<<value to represent reachable sums, then explain why the whole shift reads the old bitset.

236. Target Sum Ways

Bridge | Module 24

Problem

Count assignments of either '+' or '-' to every occurrence in a list of nonnegative integers so the signed sum equals target. Equal values at different indices represent separate choices. For zero, +0 and -0 count as distinct assignments. Empty input has one assignment reaching zero and none reaching another target.

Examples

Example 1. Exactly one of the five occurrences must receive a minus sign.

```
target_sum_ways([1,1,1,1,1],3)
=> 5
```

Example 2. The zero can receive either sign.

```
target_sum_ways([0,1],1)
=> 2
```

Example 3. The empty sign assignment reaches zero.

```
target_sum_ways([],0)
=> 1
```

Why this mechanism works

A signed-sum dictionary expresses the state directly without a parity transformation. After each occurrence, every existing sum branches to sum+value and sum-value. Build a fresh dictionary because both branches belong to the next prefix length. When value is zero they reach the same key but must both add their counts.

State and boundaries

counts[s] is the number of sign assignments for the processed prefix producing exactly signed sum s.

Protocol

1. Seed counts={0:1}.
2. Create an empty next-layer counter per value.
3. Add each old count to both plus and minus destinations.
4. Replace the layer only after all old states are processed.
5. Return the target count, defaulting to zero.

Visualization

```
start {0:1}
value 0 -> +0 and -0 -> {0:2}
value 1 -> {1:2,-1:2}
two assignments reach target 1
```

Python implementation

```
from collections import defaultdict

def target_sum_ways(nums, target):
    counts = {0: 1}
    for value in nums:
        next_counts = defaultdict(int)
        for total, count in counts.items():
            next_counts[total + value] += count
            next_counts[total - value] += count
        counts = dict(next_counts)
    return counts.get(target, 0)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
target_sum_ways([1,1,1,1,1],3)

L4 target_sum_ways: counts = {0: 1}
nums=[1, 1, 1, 1, 1]; target=3; counts={0:1}

L5 target_sum_ways: for value in nums:
nums=[1, 1, 1, 1, 1]; target=3; counts={0:1}; value=1

L6 target_sum_ways: next_counts = defaultdict(int)
nums=[1, 1, 1, 1, 1]; target=3; counts={0:1}; value=1; next_counts={}

L7 target_sum_ways: for total, count in counts.items():
nums=[1, 1, 1, 1, 1]; target=3; counts={0:1}; value=1; next_counts={}; total=0; count=1

L8 target_sum_ways: next_counts[total + value] += count
nums=[1, 1, 1, 1, 1]; target=3; counts={0:1}; value=1; next_counts={1:1}; total=0; count=1

L9 target_sum_ways: next_counts[total - value] += count
nums=[1, 1, 1, 1, 1]; target=3; counts={0:1}; value=1; next_counts={1:1, -1:1}; total=0;
    count=1

L10 target_sum_ways: counts = dict(next_counts)
nums=[1, 1, 1, 1, 1]; target=3; counts={1:1, -1:1}; value=1; next_counts={1:1, -1:1}; total=0;
     count=1

L6 target_sum_ways: next_counts = defaultdict(int)
nums=[1, 1, 1, 1, 1]; target=3; counts={1:1, -1:1}; value=1; next_counts={}; total=0; count=1

... 45 intermediate state changes omitted ...

L7 target_sum_ways: for total, count in counts.items():
nums=[1, 1, 1, 1, 1]; target=3; counts={4:1, 2:4, 0:6, -2:4, -4:1}; value=1; next_counts={5:1,
     3:5, 1:10, -1:10, -3:4}; total=-4; count=1

L8 target_sum_ways: next_counts[total + value] += count
```

```
nums=[1, 1, 1, 1, 1]; target=3; counts={4:1, 2:4, 0:6, -2:4, -4:1}; value=1; next_counts={5:1,
     3:5, 1:10, -1:10, -3:5}; total=-4; count=1

L9 target_sum_ways: next_counts[total - value] += count
nums=[1, 1, 1, 1, 1]; target=3; counts={4:1, 2:4, 0:6, -2:4, -4:1}; value=1; next_counts={5:1,
     3:5, 1:10, -1:10, -3:5, ...}; total=-4; count=1

L10 target_sum_ways: counts = dict(next_counts)
nums=[1, 1, 1, 1, 1]; target=3; counts={5:1, 3:5, 1:10, -1:10, -3:5, ...}; value=1;
    next_counts={5:1, 3:5, 1:10, -1:10, -3:5, ...}; total=-4; count=1
```

Result: 5. Exactly one of the five occurrences must receive a minus sign.

```
target_sum_ways([0,1],1)

L4 target_sum_ways: counts = {0: 1}
nums=[0, 1]; target=1; counts={0:1}

L5 target_sum_ways: for value in nums:
nums=[0, 1]; target=1; counts={0:1}; value=0

L6 target_sum_ways: next_counts = defaultdict(int)
nums=[0, 1]; target=1; counts={0:1}; value=0; next_counts={}

L7 target_sum_ways: for total, count in counts.items():
nums=[0, 1]; target=1; counts={0:1}; value=0; next_counts={}; total=0; count=1

L8 target_sum_ways: next_counts[total + value] += count
nums=[0, 1]; target=1; counts={0:1}; value=0; next_counts={0:1}; total=0; count=1

L9 target_sum_ways: next_counts[total - value] += count
nums=[0, 1]; target=1; counts={0:1}; value=0; next_counts={0:2}; total=0; count=1

L10 target_sum_ways: counts = dict(next_counts)
nums=[0, 1]; target=1; counts={0:2}; value=0; next_counts={0:2}; total=0; count=1

L5 target_sum_ways: for value in nums:
nums=[0, 1]; target=1; counts={0:2}; value=1; next_counts={0:2}; total=0; count=1

... 1 intermediate state changes omitted ...

L7 target_sum_ways: for total, count in counts.items():
nums=[0, 1]; target=1; counts={0:2}; value=1; next_counts={}; total=0; count=2

L8 target_sum_ways: next_counts[total + value] += count
nums=[0, 1]; target=1; counts={0:2}; value=1; next_counts={1:2}; total=0; count=2

L9 target_sum_ways: next_counts[total - value] += count
nums=[0, 1]; target=1; counts={0:2}; value=1; next_counts={1:2, -1:2}; total=0; count=2

L10 target_sum_ways: counts = dict(next_counts)
nums=[0, 1]; target=1; counts={1:2, -1:2}; value=1; next_counts={1:2, -1:2}; total=0; count=2
```

Result: 2. The zero can receive either sign.

Why the solution is correct

Every full assignment has a unique shorter assignment and final sign. The two transitions enumerate exactly those alternatives and sum multiplicities when different assignments reach the same sum. Fresh layers ensure each input occurrence receives exactly one sign.

Time and space

O(n*(S+1)) coarse time and O(S+1) dictionary states for S=sum(nums), since sums lie in [-S,S]. Counts themselves may need O(n) bits. Sparse layers can be smaller.

Common mistakes

Using a set loses the number of assignments. Assigning rather than adding the two zero transitions undercounts by a factor of two per zero.

Reasoning card

State: counts[s] is the number of sign assignments for the processed prefix producing exactly signed sum s.

Invariant: Each state counts assignments for a processed prefix.

Decision to rehearse: Use signed-sum state or transform to subset-sum counting.

Finish: Return the target count, defaulting to zero

Transfer exercise

Derive the subset-count transformation P=(total+target)/2 and state its parity, range and zero-handling conditions.

237. Longest Common Subsequence

Bridge | Module 24

Problem

Return the length of the longest common subsequence of two strings. Characters must retain order within each string but need not be adjacent. Matching is exact and case-sensitive. If either input is empty, return zero. The task asks for length, not a reconstructed subsequence.

Examples

Example 1. Skipping unmatched characters preserves the common order.

```
lcs_length("abcde","ace")
=> 3
```

Example 2. No character can be matched.

```
lcs_length("abc","def")
=> 0
```

Example 3. An empty prefix contributes no matches.

```
lcs_length("","abc")
=> 0
```

Why this mechanism works

Let each axis represent a prefix length, with row zero and column zero as empty prefixes. If the final characters match, a common subsequence can extend the diagonal answer by one. Otherwise at least one final character must be omitted, so take the better answer after dropping either side.

State and boundaries

dp[i][j] is the LCS length for first[:i] and second[:j]. Smaller prefix pairs are solved before the current pair.

Protocol

1. Allocate an (m+1)-by-(n+1) zero table.

2. Scan positive prefix lengths.

3. On equal final characters use diagonal+1.

4. Otherwise use max(top,left).

5. Return dp[m][n].

Visualization

```
first abcde, second ace
match a, skip b, match c, skip d, match e
subsequence 'ace' has length 3 without requiring adjacency
```

Python implementation

```
def lcs_length(first, second):
    m, n = len(first), len(second)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if first[i - 1] == second[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp[m][n]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
lcs_length("abcde","ace")

L2 lcs_length: m, n = len(first), len(second)
first='abcde'; second='ace'; m=5; n=3

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abcde'; second='ace'; m=5; n=3; _=0

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abcde'; second='ace'; m=5; n=3; _=1

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abcde'; second='ace'; m=5; n=3; _=2

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abcde'; second='ace'; m=5; n=3; _=3

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abcde'; second='ace'; m=5; n=3; _=4

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abcde'; second='ace'; m=5; n=3; _=5

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abcde'; second='ace'; m=5; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
     0], [0, 0, 0, 0], [0, 0, 0, 0]]

... 31 intermediate state changes omitted ...

L5 lcs_length: for j in range(1, n + 1):
first='abcde'; second='ace'; m=5; n=3; dp=[[0, 0, 0, 0], [0, 1, 1, 1], [0, 1, 1, 1], [0, 1, 2,
     2], [0, 1, 2, 2], [0, 1, 0, 0]]; i=5; j=2

L9 lcs_length: dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
```

```
first='abcde'; second='ace'; m=5; n=3; dp=[[0, 0, 0, 0], [0, 1, 1, 1], [0, 1, 1, 1], [0, 1, 2,
     2], [0, 1, 2, 2], [0, 1, 2, 0]]; i=5; j=2

L5 lcs_length: for j in range(1, n + 1):
first='abcde'; second='ace'; m=5; n=3; dp=[[0, 0, 0, 0], [0, 1, 1, 1], [0, 1, 1, 1], [0, 1, 2,
     2], [0, 1, 2, 2], [0, 1, 2, 0]]; i=5; j=3

L7 lcs_length: dp[i][j] = dp[i - 1][j - 1] + 1
first='abcde'; second='ace'; m=5; n=3; dp=[[0, 0, 0, 0], [0, 1, 1, 1], [0, 1, 1, 1], [0, 1, 2,
     2], [0, 1, 2, 2], [0, 1, 2, 3]]; i=5; j=3
```

Result: 3. Skipping unmatched characters preserves the common order.

```
lcs_length("abc","def")

L2 lcs_length: m, n = len(first), len(second)
first='abc'; second='def'; m=3; n=3

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abc'; second='def'; m=3; n=3; _=0

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abc'; second='def'; m=3; n=3; _=1

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abc'; second='def'; m=3; n=3; _=2

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abc'; second='def'; m=3; n=3; _=3

L3 lcs_length: dp = [[0] * (n + 1) for _ in range(m + 1)]
first='abc'; second='def'; m=3; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
    0]]

L4 lcs_length: for i in range(1, m + 1):
first='abc'; second='def'; m=3; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
    0]]; i=1

L5 lcs_length: for j in range(1, n + 1):
first='abc'; second='def'; m=3; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
    0]]; i=1; j=1

... 6 intermediate state changes omitted ...

L4 lcs_length: for i in range(1, m + 1):
first='abc'; second='def'; m=3; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
    0]]; i=3; j=3

L5 lcs_length: for j in range(1, n + 1):
first='abc'; second='def'; m=3; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
    0]]; i=3; j=1

L5 lcs_length: for j in range(1, n + 1):
first='abc'; second='def'; m=3; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
    0]]; i=3; j=2

L5 lcs_length: for j in range(1, n + 1):
first='abc'; second='def'; m=3; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0,
    0]]; i=3; j=3
```

Result: 0. No character can be matched.

Why the solution is correct

Empty prefixes share only an empty subsequence. Matching endpoints admit an optimum ending with that common character, reducing to the two shorter prefixes. With unequal endpoints, no common subsequence can use both as its final matched pair, so an optimum omits at least one and is covered by top or left.

Time and space

O(m*n) time and O(m*n) auxiliary table space.

Common mistakes

Resetting to zero on mismatch computes longest common substring, a different problem. Table indices are prefix lengths, so character indices are one smaller.

Reasoning card

State: dp[i][j] is the LCS length for first[:i] and second[:j]. Smaller prefix pairs are solved before the current pair.

Invariant: dp[i][j] describes prefixes of lengths i and j.

Decision to rehearse: Match uses diagonal +1; mismatch uses max(top,left).

Finish: Return dp[m][n]

Transfer exercise

Backtrack through the table to reconstruct one LCS, with an explicit tie rule when top and left are equal.

238. Edit Distance

Interview | Module 24

Problem

Return the minimum number of single-character insertions, deletions and replacements needed to transform source into target. Each operation costs one; matching characters costs zero. Strings use exact Python-character comparison. Empty strings are permitted; transposition is not an allowed single operation.

Examples

Example 1. One replacement and two deletions attain the optimum.

```
edit_distance("horse","ros")
=> 3
```

Example 2. Three insertions build the target from empty.

```
edit_distance("","abc")
=> 3
```

Example 3. Matching characters require no edits.

```
edit_distance("same","same")
=> 0
```

Why this mechanism works

Use prefix lengths again, but now the state is a minimum transformation cost. Deleting the source's final character comes from above, inserting the target's final character from the left, and replacing from the diagonal. Equal final characters simply inherit the diagonal cost.

State and boundaries

dp[i][j] is the minimum edit cost from source[:i] to target[:j]. Row zero counts insertions; column zero counts deletions.

Protocol

1. Initialize dp[i][0]=i and dp[0][j]=j.
2. Scan positive prefix pairs.
3. Copy the diagonal on equal final characters.
4. Otherwise add one to min(top,left,diagonal).
5. Return the full-prefix cost.

Visualization

```
horse -> rorse (replace h by r)
rorse -> rose  (delete the extra r)
rose  -> ros   (delete e)
three edits
```

Python implementation

```
def edit_distance(source, target):
    m, n = len(source), len(target)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if source[i - 1] == target[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
    return dp[m][n]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
edit_distance("horse","ros")

L2 edit_distance: m, n = len(source), len(target)
source='horse'; target='ros'; m=5; n=3

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source='horse'; target='ros'; m=5; n=3; _=0

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source='horse'; target='ros'; m=5; n=3; _=1

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source='horse'; target='ros'; m=5; n=3; _=2

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source='horse'; target='ros'; m=5; n=3; _=3

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source='horse'; target='ros'; m=5; n=3; _=4

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source='horse'; target='ros'; m=5; n=3; _=5

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source='horse'; target='ros'; m=5; n=3; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0,
    0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

... 49 intermediate state changes omitted ...

L9 edit_distance: for j in range(1, n + 1):
source='horse'; target='ros'; m=5; n=3; dp=[[0, 1, 2, 3], [1, 1, 2, 3], [2, 2, 1, 2], [3, 2,
```

```
    2, 2], [4, 3, 3, 2], [5, 4, 0, 0]]; i=5; j=2

L13 edit_distance: dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
source='horse'; target='ros'; m=5; n=3; dp=[[0, 1, 2, 3], [1, 1, 2, 3], [2, 2, 1, 2], [3, 2,
    2, 2], [4, 3, 3, 2], [5, 4, 4, 0]]; i=5; j=2

L9 edit_distance: for j in range(1, n + 1):
source='horse'; target='ros'; m=5; n=3; dp=[[0, 1, 2, 3], [1, 1, 2, 3], [2, 2, 1, 2], [3, 2,
    2, 2], [4, 3, 3, 2], [5, 4, 4, 0]]; i=5; j=3

L13 edit_distance: dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1])
source='horse'; target='ros'; m=5; n=3; dp=[[0, 1, 2, 3], [1, 1, 2, 3], [2, 2, 1, 2], [3, 2,
    2, 2], [4, 3, 3, 2], [5, 4, 4, 3]]; i=5; j=3
```

Result: 3. One replacement and two deletions attain the optimum.

```
edit_distance("","abc")

L2 edit_distance: m, n = len(source), len(target)
source=''; target='abc'; m=0; n=3

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source=''; target='abc'; m=0; n=3; _=0

L3 edit_distance: dp = [[0] * (n + 1) for _ in range(m + 1)]
source=''; target='abc'; m=0; n=3; dp=[[0, 0, 0, 0]]

L4 edit_distance: for i in range(m + 1):
source=''; target='abc'; m=0; n=3; dp=[[0, 0, 0, 0]]; i=0

L6 edit_distance: for j in range(n + 1):
source=''; target='abc'; m=0; n=3; dp=[[0, 0, 0, 0]]; i=0; j=0

L6 edit_distance: for j in range(n + 1):
source=''; target='abc'; m=0; n=3; dp=[[0, 0, 0, 0]]; i=0; j=1

L7 edit_distance: dp[0][j] = j
source=''; target='abc'; m=0; n=3; dp=[[0, 1, 0, 0]]; i=0; j=1

L6 edit_distance: for j in range(n + 1):
source=''; target='abc'; m=0; n=3; dp=[[0, 1, 0, 0]]; i=0; j=2

L7 edit_distance: dp[0][j] = j
source=''; target='abc'; m=0; n=3; dp=[[0, 1, 2, 0]]; i=0; j=2

L6 edit_distance: for j in range(n + 1):
source=''; target='abc'; m=0; n=3; dp=[[0, 1, 2, 0]]; i=0; j=3

L7 edit_distance: dp[0][j] = j
source=''; target='abc'; m=0; n=3; dp=[[0, 1, 2, 3]]; i=0; j=3
```

Result: 3. Three insertions build the target from empty.

Why the solution is correct

The boundaries are the only possible transformations to or from an empty string. For a mismatching final alignment, the last edit is an insertion, deletion or replacement, and each predecessor represents its optimal remaining work. Minimizing those exhaustive alternatives gives the optimal cost; matching endpoints need no final edit.

Time and space

O(m*n) time and O(m*n) auxiliary space for source and target lengths m,n.

Common mistakes

Omitting the empty-prefix row and column makes insertions or deletions at the beginning awkward and error-prone. Treating a swap as one operation computes a different distance.

Reasoning card

State: dp[i][j] is the minimum edit cost from source[:i] to target[:j]. Row zero counts insertions; column zero counts deletions.

Invariant: State is minimum edits between two prefixes.

Decision to rehearse: Equal characters use diagonal; mismatch adds one to best predecessor operation.

Finish: Return the full-prefix cost

Transfer exercise

Assign different insertion, deletion and replacement costs, and revise the boundaries and transition costs explicitly.

239. Interleaving String

Interview | Module 24

Problem

Return whether target is an interleaving of first and second, using every character of each source exactly once while preserving each source's internal order. Matching is exact. Characters may be taken from either source at each step. All three empty strings form a valid interleaving.

Examples

Example 1. Alternating source choices preserves both internal orders.

```
is_interleaving("ab","cd","acbd")
=> True
```

Example 2. The target places d before c, violating the second source order.

```
is_interleaving("ab","cd","adbc")
=> False
```

Example 3. The seeded empty state is the full result.

```
is_interleaving("","","")
=> True
```

Why this mechanism works

Two consumed counts determine the target position automatically: after i characters from first and j from second, exactly i+j target characters have been formed. A state is reachable if its last character came from a reachable predecessor in either source and matches the corresponding target character. Ambiguous equal characters require considering both alternatives.

State and boundaries

dp[i][j] means target[:i+j] can be formed from first[:i] and second[:j]. dp[0][0] is true.

Protocol

1. Reject a total-length mismatch.

2. Seed the empty-prefix state.

3. Scan all prefix pairs.

4. Accept a transition from above if the next first character matches.

5. Accept one from the left if the next second character matches.

6. Return the full-prefix state.

Visualization

```
first ab, second cd, target acbd
(0,0)->(1,0)->(1,1)->(2,1)->(2,2)
target progress is always i+j
```

Python implementation

```
def is_interleaving(first, second, target):
    m, n = len(first), len(second)
    if m + n != len(target):
        return False
    dp = [[False] * (n + 1) for _ in range(m + 1)]
    dp[0][0] = True
    for i in range(m + 1):
        for j in range(n + 1):
            if i:
                dp[i][j] |= dp[i - 1][j] and first[i - 1] == target[i + j - 1]
            if j:
                dp[i][j] |= dp[i][j - 1] and second[j - 1] == target[i + j - 1]
    return dp[m][n]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
is_interleaving("ab","cd","acbd")

L2 is_interleaving: m, n = len(first), len(second)
first='ab'; second='cd'; target='acbd'; m=2; n=2

L5 is_interleaving: dp = [[False] * (n + 1) for _ in range(m + 1)]
first='ab'; second='cd'; target='acbd'; m=2; n=2; _=0

L5 is_interleaving: dp = [[False] * (n + 1) for _ in range(m + 1)]
first='ab'; second='cd'; target='acbd'; m=2; n=2; _=1

L5 is_interleaving: dp = [[False] * (n + 1) for _ in range(m + 1)]
first='ab'; second='cd'; target='acbd'; m=2; n=2; _=2

L5 is_interleaving: dp = [[False] * (n + 1) for _ in range(m + 1)]
first='ab'; second='cd'; target='acbd'; m=2; n=2; dp=[[False, False, False], [False, False,
    False], [False, False, False]]

L6 is_interleaving: dp[0][0] = True
first='ab'; second='cd'; target='acbd'; m=2; n=2; dp=[[True, False, False], [False, False,
    False], [False, False, False]]

L7 is_interleaving: for i in range(m + 1):
first='ab'; second='cd'; target='acbd'; m=2; n=2; dp=[[True, False, False], [False, False,
    False], [False, False, False]]; i=0

L8 is_interleaving: for j in range(n + 1):
first='ab'; second='cd'; target='acbd'; m=2; n=2; dp=[[True, False, False], [False, False,
    False], [False, False, False]]; i=0; j=0

... 10 intermediate state changes omitted ...
```

```
L8 is_interleaving: for j in range(n + 1):
first='ab'; second='cd'; target='acbd'; m=2; n=2; dp=[[True, False, False], [True, True,
    False], [False, False, False]]; i=2; j=1

L10 is_interleaving: dp[i][j] |= dp[i - 1][j] and first[i - 1] == target[i + j - 1]
first='ab'; second='cd'; target='acbd'; m=2; n=2; dp=[[True, False, False], [True, True,
    False], [False, True, False]]; i=2; j=1

L8 is_interleaving: for j in range(n + 1):
first='ab'; second='cd'; target='acbd'; m=2; n=2; dp=[[True, False, False], [True, True,
    False], [False, True, False]]; i=2; j=2

L12 is_interleaving: dp[i][j] |= dp[i][j - 1] and second[j - 1] == target[i + j - 1]
first='ab'; second='cd'; target='acbd'; m=2; n=2; dp=[[True, False, False], [True, True,
    False], [False, True, True]]; i=2; j=2
```

Result: True. Alternating source choices preserves both internal orders.

```
is_interleaving("ab","cd","adbc")

L2 is_interleaving: m, n = len(first), len(second)
first='ab'; second='cd'; target='adbc'; m=2; n=2

L5 is_interleaving: dp = [[False] * (n + 1) for _ in range(m + 1)]
first='ab'; second='cd'; target='adbc'; m=2; n=2; _=0

L5 is_interleaving: dp = [[False] * (n + 1) for _ in range(m + 1)]
first='ab'; second='cd'; target='adbc'; m=2; n=2; _=1

L5 is_interleaving: dp = [[False] * (n + 1) for _ in range(m + 1)]
first='ab'; second='cd'; target='adbc'; m=2; n=2; _=2

L5 is_interleaving: dp = [[False] * (n + 1) for _ in range(m + 1)]
first='ab'; second='cd'; target='adbc'; m=2; n=2; dp=[[False, False, False], [False, False,
    False], [False, False, False]]

L6 is_interleaving: dp[0][0] = True
first='ab'; second='cd'; target='adbc'; m=2; n=2; dp=[[True, False, False], [False, False,
    False], [False, False, False]]

L7 is_interleaving: for i in range(m + 1):
first='ab'; second='cd'; target='adbc'; m=2; n=2; dp=[[True, False, False], [False, False,
    False], [False, False, False]]; i=0

L8 is_interleaving: for j in range(n + 1):
first='ab'; second='cd'; target='adbc'; m=2; n=2; dp=[[True, False, False], [False, False,
    False], [False, False, False]]; i=0; j=0

... 7 intermediate state changes omitted ...

L7 is_interleaving: for i in range(m + 1):
first='ab'; second='cd'; target='adbc'; m=2; n=2; dp=[[True, False, False], [True, False,
    False], [False, False, False]]; i=2; j=2

L8 is_interleaving: for j in range(n + 1):
first='ab'; second='cd'; target='adbc'; m=2; n=2; dp=[[True, False, False], [True, False,
    False], [False, False, False]]; i=2; j=0

L8 is_interleaving: for j in range(n + 1):
first='ab'; second='cd'; target='adbc'; m=2; n=2; dp=[[True, False, False], [True, False,
    False], [False, False, False]]; i=2; j=1
```

```
L8 is_interleaving: for j in range(n + 1):
first='ab'; second='cd'; target='adbc'; m=2; n=2; dp=[[True, False, False], [True, False,
    False], [False, False, False]]; i=2; j=2
```

Result: False. The target places d before c, violating the second source order.

Why the solution is correct

Any interleaving's last character comes from exactly one source, so removing it gives one of the two predecessor states. Conversely, appending a matching source character to a valid predecessor preserves both source orders. Their boolean union therefore captures all possible interleavings.

Time and space

O(m*n+m+n) time and O((m+1)*(n+1)) table space, including empty-source boundaries.

Common mistakes

Greedily choosing first whenever both sources match can block a later valid route. A third DP axis is redundant because i+j determines target progress.

Reasoning card

State: dp[i][j] means target[:i+j] can be formed from first[:i] and second[:j]. dp[0][0] is true.

Invariant: Consumed counts determine the corresponding position in s3.

Decision to rehearse: State records how many characters consumed from each source.

Finish: Return the full-prefix state

Transfer exercise

Count distinct source-choice sequences instead of returning a boolean, noting that equal character values can hide different choices.

240. Longest Palindromic Subsequence

Interview | Module 24

Problem

Return the length of the longest palindromic subsequence of text. Characters may be skipped but their order must remain. Matching is exact and case-sensitive. Empty input returns zero; every singleton has length one.

Examples

Example 1. Skipping a yields the length-four palindrome bbbb.

```
longest_palindromic_subsequence("bbbab")
=> 4
```

Example 2. The two middle b characters form the best subsequence.

```
longest_palindromic_subsequence("cbbd")
=> 2
```

Example 3. No character can be selected.

```
longest_palindromic_subsequence("")
=> 0
```

Why this mechanism works

Use an interval rather than two independent prefix axes. If both endpoints match, they can surround a longest palindromic subsequence of the interior. If they differ, an optimum cannot use both as its outermost pair, so discard one endpoint and take the better interval. Fill shorter dependencies before larger intervals.

State and boundaries

dp[left][right] is the optimum within the inclusive interval text[left:right+1]. Diagonal states are one; the empty interior of a matching pair contributes zero.

Protocol

1. Handle empty text.

2. Scan left indices downward.

3. Set dp[left][left]=1.

4. Scan right indices upward.

5. On matching ends use inner+2, with a zero inner for adjacent ends.

6. Otherwise use max(drop-left,drop-right).

7. Return the full interval.

Visualization

```
text bbbab
choose indices 0,1,2,4 -> 'bbbb'
subsequence length 4; the skipped 'a' need not remain
```

Python implementation

```
def longest_palindromic_subsequence(text):
    n = len(text)
    if n == 0:
        return 0
    dp = [[0] * n for _ in range(n)]
    for left in range(n - 1, -1, -1):
        dp[left][left] = 1
        for right in range(left + 1, n):
            if text[left] == text[right]:
                inner = dp[left + 1][right - 1] if right - left > 1 else 0
                dp[left][right] = inner + 2
            else:
                dp[left][right] = max(dp[left + 1][right], dp[left][right - 1])
    return dp[0][-1]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
longest_palindromic_subsequence("bbbab")

L2 longest_palindromic_subsequence: n = len(text)
text='bbbab'; n=5

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='bbbab'; n=5; _=0

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='bbbab'; n=5; _=1

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='bbbab'; n=5; _=2

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='bbbab'; n=5; _=3

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='bbbab'; n=5; _=4

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='bbbab'; n=5; dp=[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0,
     0, 0, 0, 0]]

L6 longest_palindromic_subsequence: for left in range(n - 1, -1, -1):
```

```
text='bbbab'; n=5; dp=[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0,
     0, 0, 0, 0]]; left=4

... 31 intermediate state changes omitted ...

L13 longest_palindromic_subsequence: dp[left][right] = max(dp[left + 1][right], dp[left][right
     - 1])
text='bbbab'; n=5; dp=[[1, 2, 3, 3, 0], [0, 1, 2, 2, 3], [0, 0, 1, 1, 3], [0, 0, 0, 1, 1], [0,
     0, 0, 0, 1]]; left=0; right=3; inner=1

L8 longest_palindromic_subsequence: for right in range(left + 1, n):
text='bbbab'; n=5; dp=[[1, 2, 3, 3, 0], [0, 1, 2, 2, 3], [0, 0, 1, 1, 3], [0, 0, 0, 1, 1], [0,
     0, 0, 0, 1]]; left=0; right=4; inner=1

L10 longest_palindromic_subsequence: inner = dp[left + 1][right - 1] if right - left > 1 else
    0
text='bbbab'; n=5; dp=[[1, 2, 3, 3, 0], [0, 1, 2, 2, 3], [0, 0, 1, 1, 3], [0, 0, 0, 1, 1], [0,
     0, 0, 0, 1]]; left=0; right=4; inner=2

L11 longest_palindromic_subsequence: dp[left][right] = inner + 2
text='bbbab'; n=5; dp=[[1, 2, 3, 3, 4], [0, 1, 2, 2, 3], [0, 0, 1, 1, 3], [0, 0, 0, 1, 1], [0,
     0, 0, 0, 1]]; left=0; right=4; inner=2
```

Result: 4. Skipping a yields the length-four palindrome bbbb.

`longest_palindromic_subsequence("cbbd")`

```
L2 longest_palindromic_subsequence: n = len(text)
text='cbbd'; n=4

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='cbbd'; n=4; _=0

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='cbbd'; n=4; _=1

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='cbbd'; n=4; _=2

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='cbbd'; n=4; _=3

L5 longest_palindromic_subsequence: dp = [[0] * n for _ in range(n)]
text='cbbd'; n=4; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]

L6 longest_palindromic_subsequence: for left in range(n - 1, -1, -1):
text='cbbd'; n=4; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]; left=3

L7 longest_palindromic_subsequence: dp[left][left] = 1
text='cbbd'; n=4; dp=[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 1]]; left=3

... 15 intermediate state changes omitted ...

L8 longest_palindromic_subsequence: for right in range(left + 1, n):
text='cbbd'; n=4; dp=[[1, 1, 0, 0], [0, 1, 2, 2], [0, 0, 1, 1], [0, 0, 0, 1]]; left=0;
    right=2; inner=0

L13 longest_palindromic_subsequence: dp[left][right] = max(dp[left + 1][right], dp[left][right
     - 1])
text='cbbd'; n=4; dp=[[1, 1, 2, 0], [0, 1, 2, 2], [0, 0, 1, 1], [0, 0, 0, 1]]; left=0;
    right=2; inner=0

L8 longest_palindromic_subsequence: for right in range(left + 1, n):
text='cbbd'; n=4; dp=[[1, 1, 2, 0], [0, 1, 2, 2], [0, 0, 1, 1], [0, 0, 0, 1]]; left=0;
```

```
    right=3; inner=0

L13 longest_palindromic_subsequence: dp[left][right] = max(dp[left + 1][right], dp[left][right
     - 1])
text='cbbd'; n=4; dp=[[1, 1, 2, 2], [0, 1, 2, 2], [0, 0, 1, 1], [0, 0, 0, 1]]; left=0;
    right=3; inner=0
```

Result: 2. The two middle b characters form the best subsequence.

Why the solution is correct

Singleton and empty intervals have the stated bases. For matching endpoints, an optimum can include that matching pair around an optimum interior; a solution omitting an endpoint can be adjusted to include the pair without reducing length. For unequal endpoints at least one is omitted, and the two smaller intervals cover both possibilities.

Time and space

O(n^2) time and O(n^2) auxiliary table space.

Common mistakes

A palindromic substring requires adjacency and has a different recurrence. Reading dp[left+1][right-1] before that row is complete violates dependency order.

Reasoning card

State: dp[left][right] is the optimum within the inclusive interval text[left:right+1]. Diagonal states are one; the empty interior of a matching pair contributes zero.

Invariant: dp[l][r] is the answer for substring l..r.

Decision to rehearse: Matching ends use inner interval +2; otherwise drop one end.

Finish: Return the full interval

Transfer exercise

Reconstruct one palindrome by following interval decisions, storing matched endpoints on opposite sides of the output.

Module 25
Bit Manipulation, Mathematics, and Designed Data Structures

Use algebraic identities and maintain multi-operation invariants in stateful classes.

Finish with algebraic summaries and persistent multi-operation state. XOR cancels paired information; Euclid preserves a mathematical identity. RandomizedSet, LRU and TimeMap combine structures whose invariants must agree after every operation. The final timestamp search returns to the half-open boundary discipline developed earlier.

The ten-step progression

241. Single Number with XOR

242. Missing Number with XOR

243. Count Set Bits

244. Counting Bits from 0 to N

245. Reverse 32 Bits

246. Greatest Common Divisor and LCM

247. Sieve of Eratosthenes

248. Design Randomized Set

249. Design LRU Cache

250. Time-Based Key-Value Store

Module mastery check

Reconstruct at least eight solutions without looking. For three randomly chosen exercises, explain every branch and mutation, then revisit them after a delay. Record the exact failure category: model, state, ingest, repair, commit, boundary, restore, schedule, contract or complexity.

241. Single Number with XOR

Foundation | Module 25

Problem

Given a nonempty integer list in which exactly one value occurs once and every other distinct value occurs exactly twice, return the single value. Negative integers are allowed. Use XOR accumulation and leave the input unchanged. The occurrence guarantee is a precondition, not something this algorithm verifies.

Examples

Example 1. Both repeated values cancel.

```
single_number([4,1,2,1,2])
=> 4
```

Example 2. The identity works for negative integers too.

```
single_number([-3,5,5])
=> -3
```

Example 3. The sole value can be zero.

```
single_number([0])
=> 0
```

Why this mechanism works

XOR is associative and commutative, with x XOR x=0 and x XOR 0=x. Consequently each paired value cancels regardless of where its occurrences appear. The unmatched value survives. This is an algebraic summary of a prefix, rather than a frequency table.

State and boundaries

answer equals the XOR of all processed values. Its final value is determined by cancellation under the occurrence contract.

Protocol

1. Initialize answer to zero.
2. XOR each value into answer.
3. Return the accumulator.

Visualization

```
4 XOR 1 XOR 2 XOR 1 XOR 2
= 4 XOR (1 XOR 1) XOR (2 XOR 2)
= 4 XOR 0 XOR 0 = 4
```

Python implementation

```
def single_number(nums):
    answer = 0
    for value in nums:
        answer ^= value
    return answer
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
single_number([4,1,2,1,2])

L2 single_number: answer = 0
nums=[4, 1, 2, 1, 2]; answer=0

L3 single_number: for value in nums:
nums=[4, 1, 2, 1, 2]; answer=0; value=4

L4 single_number: answer ^= value
nums=[4, 1, 2, 1, 2]; answer=4; value=4

L3 single_number: for value in nums:
nums=[4, 1, 2, 1, 2]; answer=4; value=1

L4 single_number: answer ^= value
nums=[4, 1, 2, 1, 2]; answer=5; value=1

L3 single_number: for value in nums:
nums=[4, 1, 2, 1, 2]; answer=5; value=2

L4 single_number: answer ^= value
nums=[4, 1, 2, 1, 2]; answer=7; value=2

L3 single_number: for value in nums:
nums=[4, 1, 2, 1, 2]; answer=7; value=1

L4 single_number: answer ^= value
nums=[4, 1, 2, 1, 2]; answer=6; value=1

L3 single_number: for value in nums:
nums=[4, 1, 2, 1, 2]; answer=6; value=2

L4 single_number: answer ^= value
nums=[4, 1, 2, 1, 2]; answer=4; value=2
```

Result: 4. Both repeated values cancel.

```
single_number([-3,5,5])

L2 single_number: answer = 0
nums=[-3, 5, 5]; answer=0

L3 single_number: for value in nums:
```

```
nums=[-3, 5, 5]; answer=0; value=-3

L4 single_number: answer ^= value
nums=[-3, 5, 5]; answer=-3; value=-3

L3 single_number: for value in nums:
nums=[-3, 5, 5]; answer=-3; value=5

L4 single_number: answer ^= value
nums=[-3, 5, 5]; answer=-8; value=5

L4 single_number: answer ^= value
nums=[-3, 5, 5]; answer=-3; value=5
```

Result: -3. The identity works for negative integers too.

Why the solution is correct

The loop invariant follows directly from appending one more XOR term. At completion, associativity and commutativity allow equal terms to be paired and canceled, leaving precisely the sole unpaired value. Python's integer XOR identities also hold for negative integers.

Time and space

O(n) XOR operations and O(1) integer variables; with b-bit values, bit-level work is O(n*b) and the accumulator uses O(b) bits.

Common mistakes

XOR does not prove that the promised frequency pattern holds. Replacing XOR with OR prevents equal values from canceling.

Reasoning card

State: answer equals the XOR of all processed values. Its final value is determined by cancellation under the occurrence contract.

Invariant: Accumulator equals XOR of the processed prefix.

Decision to rehearse: Use x^x=0 and x^0=x.

Finish: Return the accumulator

Transfer exercise

Find two single values when every other value appears twice by partitioning on one set bit of their combined XOR.

242. Missing Number with XOR

Foundation | Module 25

Problem

Given a length-n list of distinct integers drawn from 0 through n inclusive, return the one missing value. Empty input returns zero. The values satisfy the stated range and uniqueness guarantees. Do not mutate or sort the input.

Examples

Example 1. The missing interior range value remains after cancellation.

```
missing_number([3,0,1])
=> 2
```

Example 2. The omitted value may be the upper endpoint n.

```
missing_number([0,1])
=> 2
```

Example 3. The expected range contains only zero.

```
missing_number([])
=> 0
```

Why this mechanism works

XOR the complete expected range and the actual values. Every present value occurs once on each side and cancels. Use the loop's index to supply expected values 0 through n-1, and seed the accumulator with n to include the final expected value.

State and boundaries

After processing index i, answer contains n XOR all expected indices through i XOR all actual values through i.

Protocol

1. Initialize answer to len(nums).

2. For each index and value XOR both into answer.

3. Return the unmatched expected value.

Visualization

```
actual [3,0,1]; expected 0,1,2,3
cancel 0,1,3 on both sides
unmatched expected value 2 remains
```

Python implementation

```
def missing_number(nums):
    answer = len(nums)
    for i, value in enumerate(nums):
        answer ^= i ^ value
    return answer
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
missing_number([3,0,1])

L2 missing_number: answer = len(nums)
nums=[3, 0, 1]; answer=3

L3 missing_number: for i, value in enumerate(nums):
nums=[3, 0, 1]; answer=3; i=0; value=3

L4 missing_number: answer ^= i ^ value
nums=[3, 0, 1]; answer=0; i=0; value=3

L3 missing_number: for i, value in enumerate(nums):
nums=[3, 0, 1]; answer=0; i=1; value=0

L4 missing_number: answer ^= i ^ value
nums=[3, 0, 1]; answer=1; i=1; value=0

L3 missing_number: for i, value in enumerate(nums):
nums=[3, 0, 1]; answer=1; i=2; value=1

L4 missing_number: answer ^= i ^ value
nums=[3, 0, 1]; answer=2; i=2; value=1
```

Result: 2. The missing interior range value remains after cancellation.

```
missing_number([0,1])

L2 missing_number: answer = len(nums)
nums=[0, 1]; answer=2

L3 missing_number: for i, value in enumerate(nums):
nums=[0, 1]; answer=2; i=0; value=0

L3 missing_number: for i, value in enumerate(nums):
nums=[0, 1]; answer=2; i=1; value=1
```

Result: 2. The omitted value may be the upper endpoint n.

Why the solution is correct

At completion, the expected terms cover 0..n exactly once. All actual terms are distinct members of that range, so every present number cancels with its expected term. The only remaining term is the missing number, including zero when the input is empty.

Time and space

O(n) XOR operations and O(1) integer variables; indices require O(log(n+1)) bits.

Common mistakes

Starting at zero without also XORing n omits one expected value. Duplicate or out-of-range inputs invalidate cancellation's interpretation.

Reasoning card

State: After processing index i, answer contains n XOR all expected indices through i XOR all actual values through i.

Invariant: Paired values cancel, leaving the missing one.

Decision to rehearse: XOR all expected indices and actual values.

Finish: Return the unmatched expected value

Transfer exercise

Compare the XOR method with n*(n+1)//2-sum(nums), discussing fixed-width overflow versus Python's arbitrary-precision arithmetic.

243. Count Set Bits

Foundation | Module 25

Problem

Return the number of one bits in the binary representation of a nonnegative integer n. Zero has zero set bits. Negative integers are outside this unbounded-width contract; a fixed-width interpretation would require an explicit mask.

Examples

Example 1. Binary 101100 contains three one bits.

```
count_set_bits(44)
=> 3
```

Example 2. No loop iteration is needed.

```
count_set_bits(0)
=> 0
```

Example 3. A power of two has one set bit.

```
count_set_bits(16)
=> 1
```

Why this mechanism works

Subtracting one flips the lowest set bit to zero and all lower zero bits to one. ANDing that result with the original removes exactly the lowest set bit and leaves all higher set bits unchanged. Count how many such removals are needed to reach zero.

State and boundaries

count is the number of original set bits removed. The current n contains exactly the remaining original set bits.

Protocol

1. Initialize count to zero.
2. While n is nonzero, assign n=n AND (n-1).
3. Increment count.
4. Return the number of removals.

Visualization

```
n     = 101100
n-1   = 101011
AND   = 101000
the lowest one bit was removed
```

Python implementation

```
def count_set_bits(n):
    count = 0
    while n:
        n &= n - 1
        count += 1
    return count
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
count_set_bits(44)

L2 count_set_bits: count = 0
n=44; count=0

L4 count_set_bits: n &= n - 1
n=40; count=0

L5 count_set_bits: count += 1
n=40; count=1

L4 count_set_bits: n &= n - 1
n=32; count=1

L5 count_set_bits: count += 1
n=32; count=2

L4 count_set_bits: n &= n - 1
n=0; count=2

L5 count_set_bits: count += 1
n=0; count=3
```

Result: 3. Binary 101100 contains three one bits.

```
count_set_bits(0)

L2 count_set_bits: count = 0
n=0; count=0
```

Result: 0. No loop iteration is needed.

Why the solution is correct

For every positive n, the AND identity clears exactly one one-bit. Thus the invariant is preserved and the number of remaining set bits strictly decreases. When n reaches zero, every original one-bit has been removed exactly once, so count is correct.

Time and space

O(k) bitwise operations for k set bits, using a constant number of integer variables. For arbitrary b-bit Python integers, each operation may cost O(b), giving O(k*b) coarse bit work.

Common mistakes

Right-shifting a negative Python integer preserves the sign indefinitely; negative-width assumptions also break this loop. Counting decimal digits does not measure binary set bits.

Reasoning card

State: count is the number of original set bits removed. The current n contains exactly the remaining original set bits.

Invariant: Each iteration removes exactly one set bit.

Decision to rehearse: Repeatedly clear the lowest set bit with n &= n-1.

Finish: Return the number of removals

Transfer exercise

Count bits of a signed 32-bit value by first masking with (1<<32)-1, and explain the resulting representation.

244. Counting Bits from 0 to N

Foundation | Module 25

Problem

Return a list of set-bit counts for every integer from zero through n inclusive, where n>=0. Entry i must describe i. Use previously computed smaller answers instead of independently recounting each number.

Examples

Example 1. Each entry reuses the count after removing its low bit.

```
counting_bits(5)
=> [0, 1, 1, 2, 1, 2]
```

Example 2. The inclusive range still contains zero.

```
counting_bits(0)
=> [0]
```

Example 3. Binary 10 has one set bit, like binary 1.

```
counting_bits(2)
=> [0, 1, 1]
```

Why this mechanism works

Right shift removes the least significant bit, leaving a smaller integer whose count is already known. Add back the removed bit, which is either zero or one. Ascending integer order therefore forms a dynamic program over binary representations.

State and boundaries

counts[i] is the number of one bits in i. Every dependency i>>1 is smaller than i for i>=1.

Protocol

1. Allocate n+1 zero entries.

2. For i from 1 through n, set counts[i]=counts[i>>1]+(i&1).

3. Return the full table.

Visualization

```
i=5 is binary 101
i>>1=2 is binary 10, count 1
i&1=1 -> count[5]=1+1=2
```

Python implementation

```
def counting_bits(n):
    counts = [0] * (n + 1)
    for i in range(1, n + 1):
        counts[i] = counts[i >> 1] + (i & 1)
    return counts
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
counting_bits(5)

L2 counting_bits: counts = [0] * (n + 1)
n=5; counts=[0, 0, 0, 0, 0, 0]

L3 counting_bits: for i in range(1, n + 1):
n=5; counts=[0, 0, 0, 0, 0, 0]; i=1

L4 counting_bits: counts[i] = counts[i >> 1] + (i & 1)
n=5; counts=[0, 1, 0, 0, 0, 0]; i=1

L3 counting_bits: for i in range(1, n + 1):
n=5; counts=[0, 1, 0, 0, 0, 0]; i=2

L4 counting_bits: counts[i] = counts[i >> 1] + (i & 1)
n=5; counts=[0, 1, 1, 0, 0, 0]; i=2

L3 counting_bits: for i in range(1, n + 1):
n=5; counts=[0, 1, 1, 0, 0, 0]; i=3

L4 counting_bits: counts[i] = counts[i >> 1] + (i & 1)
n=5; counts=[0, 1, 1, 2, 0, 0]; i=3

L3 counting_bits: for i in range(1, n + 1):
n=5; counts=[0, 1, 1, 2, 0, 0]; i=4

L4 counting_bits: counts[i] = counts[i >> 1] + (i & 1)
n=5; counts=[0, 1, 1, 2, 1, 0]; i=4

L3 counting_bits: for i in range(1, n + 1):
n=5; counts=[0, 1, 1, 2, 1, 0]; i=5

L4 counting_bits: counts[i] = counts[i >> 1] + (i & 1)
n=5; counts=[0, 1, 1, 2, 1, 2]; i=5
```

Result: [0, 1, 1, 2, 1, 2]. Each entry reuses the count after removing its low bit.

```
counting_bits(0)

L2 counting_bits: counts = [0] * (n + 1)
n=0; counts=[0]
```

Result: [0]. The inclusive range still contains zero.

Why the solution is correct

The binary digits of i consist of the digits of i>>1 followed by its low bit. Their set-bit counts add exactly as the recurrence states. The zero base is correct, and ascending order guarantees the smaller count is complete.

Time and space

O(n) word-level operations and O(n) output entries. Integer indices and operations have logarithmic bit width under an arbitrary-precision analysis.

Common mistakes

Allocating only n entries omits the inclusive endpoint. Adding one unconditionally overcounts even integers.

Reasoning card

State: counts[i] is the number of one bits in i. Every dependency i>>1 is smaller than i for i>=1.

Invariant: Each count is derived from a previously computed integer.

Decision to rehearse: Reuse a smaller solved state such as count[i>>1] + (i&1).

Finish: Return the full table

Transfer exercise

Use counts[i]=counts[i&(i-1)]+1 and connect its dependency to the previous kata's bit-clearing loop.

245. Reverse 32 Bits

Bridge | Module 25

Problem

Reverse the order of exactly 32 bits of an unsigned integer n satisfying 0<=n<2^32. Return the resulting unsigned integer. Leading zeros are part of the fixed-width input and become trailing zeros. Do not stop early merely because the remaining source value becomes zero.

Examples

Example 1. The least significant bit moves to the highest position.

```
reverse_32_bits(1)
=> 2147483648
```

Example 2. All 32 positions remain zero.

```
reverse_32_bits(0)
=> 0
```

Example 3. The complete fixed-width reversal includes leading zeros.

```
reverse_32_bits(43261596)
=> 964176192
```

Why this mechanism works

Read the source from its least significant end and append each bit to a growing output by shifting the output left. After k steps, the low k bits of the original source have been read in reverse order. Completing all 32 steps moves the first-read bit into its required most significant position.

State and boundaries

After k iterations, answer is the reversal of the original low k bits within k positions; n has discarded those k bits by right shifts.

Protocol

1. Initialize answer to zero.
2. Repeat exactly 32 times.
3. Shift answer left and append n&1.
4. Shift n right.
5. Return answer.

Visualization

```
32-bit source 000...0001
first bit read is 1; then 31 zeros
output 1000...0000 = 2147483648
```

Python implementation

```
def reverse_32_bits(n):
    answer = 0
    for _ in range(32):
        answer = (answer << 1) | (n & 1)
        n >>= 1
    return answer
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
reverse_32_bits(1)

L2 reverse_32_bits: answer = 0
n=1; answer=0

L3 reverse_32_bits: for _ in range(32):
n=1; answer=0; _=0

L4 reverse_32_bits: answer = (answer << 1) | (n & 1)
n=1; answer=1; _=0

L5 reverse_32_bits: n >>= 1
n=0; answer=1; _=0

L3 reverse_32_bits: for _ in range(32):
n=0; answer=1; _=1

L4 reverse_32_bits: answer = (answer << 1) | (n & 1)
n=0; answer=2; _=1

L3 reverse_32_bits: for _ in range(32):
n=0; answer=2; _=2

L4 reverse_32_bits: answer = (answer << 1) | (n & 1)
n=0; answer=4; _=2

... 54 intermediate state changes omitted ...

L3 reverse_32_bits: for _ in range(32):
n=0; answer=536870912; _=30

L4 reverse_32_bits: answer = (answer << 1) | (n & 1)
n=0; answer=1073741824; _=30

L3 reverse_32_bits: for _ in range(32):
n=0; answer=1073741824; _=31

L4 reverse_32_bits: answer = (answer << 1) | (n & 1)
n=0; answer=2147483648; _=31
```

Result: 2147483648. The least significant bit moves to the highest position.

```
reverse_32_bits(0)

L2 reverse_32_bits: answer = 0
n=0; answer=0

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=0

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=1

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=2

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=3

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=4

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=5

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=6

... 21 intermediate state changes omitted ...

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=28

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=29

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=30

L3 reverse_32_bits: for _ in range(32):
n=0; answer=0; _=31
```

Result: 0. All 32 positions remain zero.

Why the solution is correct

Each update appends the next source bit to the reversed processed prefix, preserving the k-bit invariant. After 32 iterations all source positions, including leading zeros, have been consumed; original bit i occupies output position 31-i.

Time and space

O(32)=O(1) time and O(1) storage under the fixed-width contract. Generalizing to width w uses O(w) bit-processing iterations.

Common mistakes

Looping only while n is nonzero loses leading-zero positions. Reversing a decimal string does not reverse the fixed-width binary representation.

Reasoning card

State: After k iterations, answer is the reversal of the original low k bits within k positions; n has discarded those k bits by right shifts.

Invariant: Processed source bits occupy their final reversed output positions.

Decision to rehearse: Read one bit at a time and place it in mirrored position.

Finish: Return answer

Transfer exercise

Reverse twice and prove the original 32-bit value is restored; use that property for randomized verification.

246. Greatest Common Divisor and LCM

Bridge | Module 25

Problem

Return (gcd,lcm) for any two integers a and b, using nonnegative results. Define gcd(0,0)=0 and lcm(a,0)=0. Implement Euclid's algorithm directly; compute LCM after dividing by GCD to reduce intermediate magnitude.

Examples

Example 1. Euclid preserves common divisors while shrinking the second operand.

```
gcd_lcm(48,18)
=> (6, 144)
```

Example 2. Results are normalized to nonnegative magnitudes.

```
gcd_lcm(-6,8)
=> (2, 24)
```

Example 3. The explicit zero convention avoids division by zero.

```
gcd_lcm(0,0)
=> (0, 0)
```

Why this mechanism works

A divisor common to a and b is also common to b and a mod b, and the converse holds. Repeatedly replace the pair with that smaller equivalent problem until the second value is zero. Normalize signs first. For nonzero inputs, gcd*lcm=abs(a*b).

State and boundaries

The current pair has the same GCD as the original magnitudes. saved_a and saved_b retain those magnitudes for the later LCM calculation.

Protocol

1. Take absolute values and save them.

2. While b is nonzero replace (a,b) with (b,a%b).

3. The remaining a is the GCD.

4. Return LCM zero if either original was zero.

5. Otherwise compute (saved_a//gcd)*saved_b.

Visualization

```
gcd(48,18): (48,18)->(18,12)->(12,6)->(6,0)
gcd=6; lcm=(48//6)*18=144
```

Python implementation

```
def gcd_lcm(a, b):
    saved_a, saved_b = abs(a), abs(b)
    a, b = saved_a, saved_b
    while b:
        a, b = b, a % b
    gcd = a
    lcm = 0 if saved_a == 0 or saved_b == 0 else (saved_a // gcd) * saved_b
    return (gcd, lcm)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
gcd_lcm(48,18)

L2 gcd_lcm: saved_a, saved_b = abs(a), abs(b)
a=48; b=18; saved_a=48; saved_b=18

L5 gcd_lcm: a, b = b, a % b
a=18; b=12; saved_a=48; saved_b=18

L5 gcd_lcm: a, b = b, a % b
a=12; b=6; saved_a=48; saved_b=18

L5 gcd_lcm: a, b = b, a % b
a=6; b=0; saved_a=48; saved_b=18

L6 gcd_lcm: gcd = a
a=6; b=0; saved_a=48; saved_b=18; gcd=6

L7 gcd_lcm: lcm = 0 if saved_a == 0 or saved_b == 0 else (saved_a // gcd) * saved_b
a=6; b=0; saved_a=48; saved_b=18; gcd=6; lcm=144
```

Result: (6, 144). Euclid preserves common divisors while shrinking the second operand.

```
gcd_lcm(-6,8)

L2 gcd_lcm: saved_a, saved_b = abs(a), abs(b)
a=-6; b=8; saved_a=6; saved_b=8

L3 gcd_lcm: a, b = saved_a, saved_b
a=6; b=8; saved_a=6; saved_b=8

L5 gcd_lcm: a, b = b, a % b
a=8; b=6; saved_a=6; saved_b=8

L5 gcd_lcm: a, b = b, a % b
a=6; b=2; saved_a=6; saved_b=8

L5 gcd_lcm: a, b = b, a % b
a=2; b=0; saved_a=6; saved_b=8

L6 gcd_lcm: gcd = a
```

```
a=2; b=0; saved_a=6; saved_b=8; gcd=2

L7 gcd_lcm: lcm = 0 if saved_a == 0 or saved_b == 0 else (saved_a // gcd) * saved_b
a=2; b=0; saved_a=6; saved_b=8; gcd=2; lcm=24
```

Result: (2, 24). Results are normalized to nonnegative magnitudes.

Why the solution is correct

The remainder identity preserves exactly the common divisors, while 0<=a%b<b makes the second operand strictly decrease. At b=0, the nonnegative first operand is the GCD. Dividing one original factor by that GCD and multiplying by the other implements the standard GCD-LCM identity without unnecessary intermediate growth.

Time and space

O(log(max(|a|,|b|)+1)) remainder steps and a constant number of integers. Actual arithmetic cost depends on operand bit lengths; division-before-multiplication reduces but does not universally prevent fixed-width overflow.

Common mistakes

Computing a*b//gcd before handling both zeros divides by zero. Returning a negative GCD conflicts with the normalized contract.

Reasoning card

State: The current pair has the same GCD as the original magnitudes. saved_a and saved_b retain those magnitudes for the later LCM calculation.

Invariant: gcd(a,b) is preserved by each transition.

Decision to rehearse: Replace (a,b) with (b,a%b) until b=0.

Finish: Otherwise compute (saved_a//gcd)*saved_b

Transfer exercise

Extend Euclid to return coefficients x,y satisfying a*x+b*y=gcd(a,b).

247. Sieve of Eratosthenes

Bridge | Module 25

Problem

Return all prime integers less than or equal to n in ascending order, for n>=0. A prime is an integer greater than one with no positive divisors other than one and itself. Return [] when n<2. Use the Sieve of Eratosthenes.

Examples

Example 1. All and only primes up to the inclusive endpoint survive.

```
sieve_primes(20)
=> [2, 3, 5, 7, 11, 13, 17, 19]
```

Example 2. Neither zero nor one is prime.

```
sieve_primes(1)
=> []
```

Example 3. The first prime survives without any marking pass.

```
sieve_primes(2)
=> [2]
```

Why this mechanism works

Initially treat values at least two as candidates. For each still-unmarked p through sqrt(n), mark its multiples. Begin at p*p because smaller multiples p*k with k<p already have a smaller prime factor and were handled earlier. Every composite has a prime factor no larger than its square root.

State and boundaries

prime[x] remains true only if no processed prime factor divides x. When p is reached unmarked, no smaller prime divides it, so p is prime.

Protocol

1. Handle n<2.

2. Allocate true flags and clear zero and one.

3. For candidate p while p*p<=n, if unmarked then clear multiples from p*p.

4. Return every remaining true index.

Visualization

```
p=2 marks 4,6,8,10,...
p=3 starts at 9; 6 was already handled by 2
unmarked values begin 2,3,5,7,11,...
```

Python implementation

```
def sieve_primes(n):
    if n < 2:
        return []
    prime = [True] * (n + 1)
    prime[0] = prime[1] = False
    p = 2
    while p * p <= n:
        if prime[p]:
            for multiple in range(p * p, n + 1, p):
                prime[multiple] = False
        p += 1
    return [value for value in range(2, n + 1) if prime[value]]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
sieve_primes(20)

L2 sieve_primes: if n < 2:
n=20

L4 sieve_primes: prime = [True] * (n + 1)
n=20; prime=[True, True, True, True, True, True, True, True, ...]

L5 sieve_primes: prime[0] = prime[1] = False
n=20; prime=[False, False, True, True, True, True, True, True, ...]

L6 sieve_primes: p = 2
n=20; prime=[False, False, True, True, True, True, True, True, ...]; p=2

L9 sieve_primes: for multiple in range(p * p, n + 1, p):
n=20; prime=[False, False, True, True, True, True, True, True, ...]; p=2; multiple=4

L10 sieve_primes: prime[multiple] = False
n=20; prime=[False, False, True, True, False, True, True, True, ...]; p=2; multiple=4

L9 sieve_primes: for multiple in range(p * p, n + 1, p):
n=20; prime=[False, False, True, True, False, True, True, True, ...]; p=2; multiple=6

L10 sieve_primes: prime[multiple] = False
n=20; prime=[False, False, True, True, False, True, False, True, ...]; p=2; multiple=6

... 30 intermediate state changes omitted ...

L12 sieve_primes: return [value for value in range(2, n + 1) if prime[value]]
n=20; prime=[False, False, True, True, False, True, False, True, ...]; p=5; multiple=18;
    value=18

L12 sieve_primes: return [value for value in range(2, n + 1) if prime[value]]
n=20; prime=[False, False, True, True, False, True, False, True, ...]; p=5; multiple=18;
    value=19
```

```
L12 sieve_primes: return [value for value in range(2, n + 1) if prime[value]]
n=20; prime=[False, False, True, True, False, True, False, True, ...]; p=5; multiple=18;
    value=20

L12 sieve_primes: return [value for value in range(2, n + 1) if prime[value]]
n=20; prime=[False, False, True, True, False, True, False, True, ...]; p=5; multiple=18
```

Result: [2, 3, 5, 7, 11, 13, 17, 19]. All and only primes up to the inclusive endpoint survive.

```
sieve_primes(1)

L2 sieve_primes: if n < 2:
n=1
```

Result: []. Neither zero nor one is prime.

Why the solution is correct

Only proper multiples of confirmed primes are removed, so primes are never removed. Every composite x has a prime factor p<=sqrt(x)<=sqrt(n); it is marked during that p pass. Therefore the remaining flags correspond exactly to primes, and the final ascending scan orders them.

Time and space

O(n log log n) marking time plus O(n) initialization/output scan, and O(n) auxiliary flags.

Common mistakes

Leaving one marked true incorrectly calls it prime. Starting at p rather than p*p erases the prime itself.

Reasoning card

State: prime[x] remains true only if no processed prime factor divides x. When p is reached unmarked, no smaller prime divides it, so p is prime.

Invariant: Unmarked values below the current frontier have correct primality status.

Decision to rehearse: Mark multiples starting from p*p.

Finish: Return every remaining true index

Transfer exercise

Implement a segmented sieve for a large interval using base primes only through the square root of its upper bound.

248. Design Randomized Set

Interview | Module 25

Problem

Implement RandomizedSet for distinct hashable values with insert(value), remove(value), and getRandom(). Insert/remove return whether membership changed. getRandom returns a uniformly selected current member and raises IndexError when empty. Operations should take expected amortized constant time; cryptographic randomness is not required.

Examples

Example 1. After removal only 2 can be sampled.

```
run_operations(RandomizedSet,[("insert",1),("insert",2),("remove",1),("getRandom",),("
    remove",1)])
=> [True, True, True, 2, False]
```

Example 2. Duplicate insertion leaves membership unchanged.

```
run_operations(RandomizedSet,[("insert",7),("insert",7),("remove",7)])
=> [True, False, True]
```

Example 3. The moved final value receives the vacated index.

```
store=RandomizedSet(); store.insert(1); store.insert(2); store.insert(3);
    store.remove(2)
(store.values,store.positions)
=> ([1, 3], {1: 0, 3: 1})
```

Why this mechanism works

A list gives constant-time random indexing but slow arbitrary deletion. A dictionary locates any value's index. Delete by moving the list's final value into the removed slot, updating that moved value's index, then popping the last slot. Set semantics impose no order to preserve.

State and boundaries

values contains each member once, and positions[value] is exactly its current list index. Both structures represent the same membership after every operation.

Protocol

1. Insert absent values at the list end and map their index.

2. For removal locate the slot.

3. Move the final value into it and update that value's mapping.

4. Pop the list end and delete the removed mapping.

5. Choose a uniform random list entry.

Visualization

```
values [A,B,C], positions A:0 B:1 C:2
remove B -> move C into slot 1 -> [A,C]
repair C:1, delete B
```

Python implementation

```
import random
from structures import run_operations

class RandomizedSet:
    def __init__(self):
        self.values = []
        self.positions = {}

    def insert(self, value):
        if value in self.positions:
            return False
        self.positions[value] = len(self.values)
        self.values.append(value)
        return True

    def remove(self, value):
        if value not in self.positions:
            return False
        index = self.positions[value]
        last = self.values[-1]
        self.values[index] = last
        self.positions[last] = index
        self.values.pop()
        del self.positions[value]
        return True

    def getRandom(self):
        return random.choice(self.values)
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(RandomizedSet,[("insert",1),("insert",2),("remove",1),("getRandom",),("
    remove",1)])

L6 __init__: self.values = []
self=RandomizedSet{'values':[]}

L7 __init__: self.positions = {}
self=RandomizedSet{'values':[], 'positions':{}}

L10 insert: if value in self.positions:
self=RandomizedSet{'values':[], 'positions':{}}; value=1
```

```
L12 insert: self.positions[value] = len(self.values)
self=RandomizedSet{'values':[], 'positions':{...:...}}; value=1

L13 insert: self.values.append(value)
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=1

L10 insert: if value in self.positions:
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=2

L12 insert: self.positions[value] = len(self.values)
self=RandomizedSet{'values':[...], 'positions':{...:..., ...:...}}; value=2

L13 insert: self.values.append(value)
self=RandomizedSet{'values':[..., ...], 'positions':{...:..., ...:...}}; value=2

... 3 intermediate state changes omitted ...

L23 remove: self.values.pop()
self=RandomizedSet{'values':[...], 'positions':{...:..., ...:...}}; value=1; index=0; last=2

L24 remove: del self.positions[value]
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=1; index=0; last=2

L28 getRandom: return random.choice(self.values)
self=RandomizedSet{'values':[...], 'positions':{...:...}}

L17 remove: if value not in self.positions:
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=1
```

Result: [True, True, True, 2, False]. After removal only 2 can be sampled.

```
run_operations(RandomizedSet,[("insert",7),("insert",7),("remove",7)])

L6 __init__: self.values = []
self=RandomizedSet{'values':[]}

L7 __init__: self.positions = {}
self=RandomizedSet{'values':[], 'positions':{}}

L10 insert: if value in self.positions:
self=RandomizedSet{'values':[], 'positions':{}}; value=7

L12 insert: self.positions[value] = len(self.values)
self=RandomizedSet{'values':[], 'positions':{...:...}}; value=7

L13 insert: self.values.append(value)
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=7

L10 insert: if value in self.positions:
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=7

L17 remove: if value not in self.positions:
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=7

L19 remove: index = self.positions[value]
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=7; index=0

L20 remove: last = self.values[-1]
self=RandomizedSet{'values':[...], 'positions':{...:...}}; value=7; index=0; last=7

L23 remove: self.values.pop()
self=RandomizedSet{'values':[], 'positions':{...:...}}; value=7; index=0; last=7
```

```
L24 remove: del self.positions[value]
self=RandomizedSet{'values':[], 'positions':{}}; value=7; index=0; last=7
```

Result: [True, False, True]. Duplicate insertion leaves membership unchanged.

Why the solution is correct

Insertion adds one new aligned entry. Swap-delete removes exactly the requested member and repairs the sole other index that may change; removing the final element also works because its temporary mapping is deleted afterward. Every member occupies one list slot, so uniform index selection gives uniform member selection.

Time and space

Expected amortized O(1) per operation using hashing, dynamic-array append/pop and random indexing; O(n) stored entries. Hash/equality costs depend on the value type.

Common mistakes

Forgetting to update the moved value's index corrupts later removals. Sampling dictionary iteration positions requires linear work unless a separate indexable structure is maintained.

Reasoning card

State: values contains each member once, and positions[value] is exactly its current list index. Both structures represent the same membership after every operation.

Invariant: Map indices and array positions agree after every operation.

Decision to rehearse: Combine an array with value-to-index mapping; swap-delete.

Finish: Choose a uniform random list entry

Transfer exercise

Allow duplicates with a value-to-set-of-indices map, and state how sampling should weight repeated occurrences.

249. Design LRU Cache

Interview | Module 25

Problem

Implement an LRU cache with nonnegative capacity, integer keys and values, get(key), and put(key,value). get returns -1 for a miss; a hit becomes most recent. put returns None, updates or inserts as most recent, and evicts the least recent key when over capacity. Capacity zero stores nothing.

Examples

Example 1. Reading key 1 makes key 2 the eviction victim.

```
run_operations(LRUCache,[("put",1,10),("put",2,20),("get",1),("put",3,30),("get",2),("
    get",3)],2)
=> [None, None, 10, None, -1, 30]
```

Example 2. Updating an existing key replaces its value without adding a duplicate node.

```
run_operations(LRUCache,[("put",1,10),("put",1,11),("get",1)],1)
=> [None, None, 11]
```

Example 3. A zero-capacity cache immediately evicts an insertion.

```
run_operations(LRUCache,[("put",1,10),("get",1)],0)
=> [None, -1]
```

Why this mechanism works

A dictionary finds an entry immediately, while a doubly linked list tracks recency and supports removing that known entry without searching. Sentinel nodes avoid separate empty/head/tail cases. Keep the least recent real node after head and the most recent before tail. Hits and updates move their nodes to that end.

State and boundaries

nodes maps exactly cached keys to their list nodes. The list contains each cached node once, ordered from least to most recent, with mutually consistent prev/next links.

Protocol

1. Initialize empty linked sentinels and map.

2. On a hit detach the node and append it before tail.

3. On put update an existing node or allocate a new one at the recent end.

4. If capacity is exceeded, detach head.next and delete its map entry.

Visualization

```
head <-> least recent <-> ... <-> most recent <-> tail
get(key): detach its node, append before tail
overflow: remove head.next from both list and dictionary
```

Python implementation

```
from structures import run_operations

class CacheNode:
    def __init__(self, key=None, value=None):
        self.key, self.value = key, value
        self.prev = self.next = None

class LRUCache:
    def __init__(self, capacity):
        self.capacity, self.nodes = capacity, {}
        self.head, self.tail = CacheNode(), CacheNode()
        self.head.next, self.tail.prev = self.tail, self.head

    def _detach(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def _append_recent(self, node):
        previous = self.tail.prev
        previous.next, node.prev = node, previous
        node.next, self.tail.prev = self.tail, node

    def get(self, key):
        if key not in self.nodes:
            return -1
        node = self.nodes[key]
        self._detach(node)
        self._append_recent(node)
        return node.value

    def put(self, key, value):
        if key in self.nodes:
            node = self.nodes[key]
            node.value = value
            self._detach(node)
        else:
            node = CacheNode(key, value)
            self.nodes[key] = node
        self._append_recent(node)
        if len(self.nodes) > self.capacity:
            oldest = self.head.next
            self._detach(oldest)
            del self.nodes[oldest.key]
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(LRUCache,[("put",1,10),("put",2,20),("get",1),("put",3,30),("get",2),("
    get",3)],2)

L10 __init__: self.capacity, self.nodes = capacity, {}
self=LRUCache{'capacity':2, 'nodes':{}}; capacity=2

L5 __init__: self.key, self.value = key, value
self=CacheNode{'key':None, 'value':None}; key=None; value=None

L6 __init__: self.prev = self.next = None
self=CacheNode{'key':None, 'value':None, 'prev':None, 'next':None}; key=None; value=None

L5 __init__: self.key, self.value = key, value
self=CacheNode{'key':None, 'value':None}; key=None; value=None

L6 __init__: self.prev = self.next = None
self=CacheNode{'key':None, 'value':None, 'prev':None, 'next':None}; key=None; value=None

L11 __init__: self.head, self.tail = CacheNode(), CacheNode()
self=LRUCache{'capacity':2, 'nodes':{}, 'head':CacheNode..., 'tail':CacheNode...}; capacity=2

L32 put: if key in self.nodes:
self=LRUCache{'capacity':2, 'nodes':{}, 'head':CacheNode..., 'tail':CacheNode...}; key=1;
    value=10

L5 __init__: self.key, self.value = key, value
self=CacheNode{'key':1, 'value':10}; key=1; value=10

... 33 intermediate state changes omitted ...

L24 get: if key not in self.nodes:
self=LRUCache{'capacity':2, 'nodes':{...:..., ...:...}, 'head':CacheNode...,
    'tail':CacheNode...}; key=3

L26 get: node = self.nodes[key]
self=LRUCache{'capacity':2, 'nodes':{...:..., ...:...}, 'head':CacheNode...,
    'tail':CacheNode...}; key=3; node=CacheNode{'key':3, 'value':30, 'prev':CacheNode...,
    'next':CacheNode...}

L15 _detach: node.prev.next = node.next
self=LRUCache{'capacity':2, 'nodes':{...:..., ...:...}, 'head':CacheNode...,
    'tail':CacheNode...}; node=CacheNode{'key':3, 'value':30, 'prev':CacheNode...,
    'next':CacheNode...}

L19 _append_recent: previous = self.tail.prev
self=LRUCache{'capacity':2, 'nodes':{...:..., ...:...}, 'head':CacheNode...,
    'tail':CacheNode...}; node=CacheNode{'key':3, 'value':30, 'prev':CacheNode...,
    'next':CacheNode...}; previous=CacheNode{'key':1, 'value':10, 'prev':CacheNode...,
    'next':CacheNode...}
```

Result: [None, None, 10, None, -1, 30]. Reading key 1 makes key 2 the eviction victim.

```
run_operations(LRUCache,[("put",1,10),("put",1,11),("get",1)],1)

L10 __init__: self.capacity, self.nodes = capacity, {}
self=LRUCache{'capacity':1, 'nodes':{}}; capacity=1

L5 __init__: self.key, self.value = key, value
self=CacheNode{'key':None, 'value':None}; key=None; value=None

L6 __init__: self.prev = self.next = None
self=CacheNode{'key':None, 'value':None, 'prev':None, 'next':None}; key=None; value=None

L5 __init__: self.key, self.value = key, value
```

```
self=CacheNode{'key':None, 'value':None}; key=None; value=None

L6 __init__: self.prev = self.next = None
self=CacheNode{'key':None, 'value':None, 'prev':None, 'next':None}; key=None; value=None

L11 __init__: self.head, self.tail = CacheNode(), CacheNode()
self=LRUCache{'capacity':1, 'nodes':{}, 'head':CacheNode..., 'tail':CacheNode...}; capacity=1

L32 put: if key in self.nodes:
self=LRUCache{'capacity':1, 'nodes':{}, 'head':CacheNode..., 'tail':CacheNode...}; key=1;
    value=10

L5 __init__: self.key, self.value = key, value
self=CacheNode{'key':1, 'value':10}; key=1; value=10

... 12 intermediate state changes omitted ...

L24 get: if key not in self.nodes:
self=LRUCache{'capacity':1, 'nodes':{...:...}, 'head':CacheNode..., 'tail':CacheNode...};
    key=1

L26 get: node = self.nodes[key]
self=LRUCache{'capacity':1, 'nodes':{...:...}, 'head':CacheNode..., 'tail':CacheNode...};
    key=1; node=CacheNode{'key':1, 'value':11, 'prev':CacheNode..., 'next':CacheNode...}

L15 _detach: node.prev.next = node.next
self=LRUCache{'capacity':1, 'nodes':{...:...}, 'head':CacheNode..., 'tail':CacheNode...};
    node=CacheNode{'key':1, 'value':11, 'prev':CacheNode..., 'next':CacheNode...}

L19 _append_recent: previous = self.tail.prev
self=LRUCache{'capacity':1, 'nodes':{...:...}, 'head':CacheNode..., 'tail':CacheNode...};
    node=CacheNode{'key':1, 'value':11, 'prev':CacheNode..., 'next':CacheNode...};
    previous=CacheNode{'key':None, 'value':None, 'prev':None, 'next':CacheNode...}
```

Result: [None, None, 11]. Updating an existing key replaces its value without adding a duplicate node.

Why the solution is correct

Detaching and appending changes only recency, preserving all other entries and links. Every successful access or update becomes most recent by construction. New entries also enter there. Thus head.next is always the least recently used entry, and removing it implements exactly the required eviction policy.

Time and space

Expected O(1) time per get/put with hash lookup and constant pointer changes; O(capacity) stored entries. The sentinels use constant extra space.

Common mistakes

Updating a value without refreshing recency violates the access policy. Deleting only from the list or only from the map leaves inconsistent membership. Sentinel nodes must never be returned as cached data.

Reasoning card

State: nodes maps exactly cached keys to their list nodes. The list contains each cached node once, ordered from least to most recent, with mutually consistent prev/next links.

Invariant: Order runs from least recent to most recent and contains exactly cached keys.

Decision to rehearse: Combine hash lookup with recency ordering.

Finish: If capacity is exceeded, detach head.next and delete its map entry

Transfer exercise

Add expiration times. Identify why recency ordering alone cannot efficiently find the next expiring entry.

250. Time-Based Key-Value Store

Interview | Module 25

Problem

Implement TimeMap with set(key,value,timestamp) and get(key,timestamp), for string keys/values and integer timestamps. Sets for each individual key arrive in nondecreasing timestamp order; equal timestamps are allowed and the latest set wins. get returns the value with the greatest stored timestamp <=query, or '' when none exists. set returns None.

Examples

Example 1. Upper-bound search chooses the latest eligible version at each query time.

```
run_operations(TimeMap,[("set","x","early",1),("set","x","new",4),("get","x",3),("get"
    ,"x",4),("get","x",0)])
=> [None, None, 'early', 'new', '']
```

Example 2. The later set wins at an equal timestamp.

```
run_operations(TimeMap,[("set","x","a",4),("set","x","b",4),("get","x",4)])
=> [None, None, 'b']
```

Example 3. Unknown keys have empty histories.

```
run_operations(TimeMap,[("get","missing",10)])
=> ['']
```

Why this mechanism works

Append-only sorted histories make each query an upper-bound search. Find the first timestamp greater than the query; the previous entry is the newest eligible value. Searching timestamps directly avoids constructing a temporary timestamp list on every get and naturally handles repeated timestamps by selecting the last eligible occurrence.

State and boundaries

Each history is ordered by timestamp and insertion order within ties. During search, [left,right) contains the first-greater boundary; indices below left are eligible and indices at or above right are too late.

Protocol

1. Append each set to its key history, enforcing nondecreasing time.

2. On get use an empty history for unknown keys.

3. Binary-search the first timestamp >query.

4. Move right=mid for a too-late entry, otherwise left=mid+1.

5. Return history[left-1]'s value if left>0, else ''.

Visualization

```
history times [1,4,4,9], query 4
first timestamp >4 is index 3
answer index 2: latest value stored at time 4
query 0 gives boundary 0 -> no eligible value
```

Python implementation

```
from structures import run_operations

class TimeMap:
    def __init__(self):
        self.histories = {}

    def set(self, key, value, timestamp):
        history = self.histories.setdefault(key, [])
        if history and timestamp < history[-1][0]:
            raise ValueError('timestamps must be nondecreasing per key')
        history.append((timestamp, value))

    def get(self, key, timestamp):
        history = self.histories.get(key, ())
        left, right = 0, len(history)
        while left < right:
            mid = (left + right) // 2
            if history[mid][0] > timestamp:
                right = mid
            else:
                left = mid + 1
        return history[left - 1][1] if left else ''
```

Worked execution

State changes after statements. L refers to a line in the downloadable solution. Long states and traces are explicitly abbreviated.

```
run_operations(TimeMap,[("set","x","early",1),("set","x","new",4),("get","x",3),("get"
    ,"x",4),("get","x",0)])

L5 __init__: self.histories = {}
self=TimeMap{'histories':{}}

L8 set: history = self.histories.setdefault(key, [])
self=TimeMap{'histories':{...:...}}; key='x'; value='early'; timestamp=1; history=[]

L11 set: history.append((timestamp, value))
self=TimeMap{'histories':{...:...}}; key='x'; value='early'; timestamp=1; history=[(1,
    'early')]

L8 set: history = self.histories.setdefault(key, [])
self=TimeMap{'histories':{...:...}}; key='x'; value='new'; timestamp=4; history=[(1, 'early')]
```

```
L11 set: history.append((timestamp, value))
self=TimeMap{'histories':{...:...}}; key='x'; value='new'; timestamp=4; history=[(1, 'early'),
     (4, 'new')]

L14 get: history = self.histories.get(key, ())
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=3; history=[(1, 'early'), (4, 'new')]

L15 get: left, right = 0, len(history)
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=3; history=[(1, 'early'), (4, 'new')];
     left=0; right=2

L17 get: mid = (left + right) // 2
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=3; history=[(1, 'early'), (4, 'new')];
     left=0; right=2; mid=1

... 9 intermediate state changes omitted ...

L17 get: mid = (left + right) // 2
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=0; history=[(1, 'early'), (4, 'new')];
     left=0; right=2; mid=1

L19 get: right = mid
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=0; history=[(1, 'early'), (4, 'new')];
     left=0; right=1; mid=1

L17 get: mid = (left + right) // 2
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=0; history=[(1, 'early'), (4, 'new')];
     left=0; right=1; mid=0

L19 get: right = mid
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=0; history=[(1, 'early'), (4, 'new')];
     left=0; right=0; mid=0
```

Result: [None, None, 'early', 'new', '']. Upper-bound search chooses the latest eligible version at each query time.

```
run_operations(TimeMap,[("set","x","a",4),("set","x","b",4),("get","x",4)])

L5 __init__: self.histories = {}
self=TimeMap{'histories':{}}

L8 set: history = self.histories.setdefault(key, [])
self=TimeMap{'histories':{...:...}}; key='x'; value='a'; timestamp=4; history=[]

L11 set: history.append((timestamp, value))
self=TimeMap{'histories':{...:...}}; key='x'; value='a'; timestamp=4; history=[(4, 'a')]

L8 set: history = self.histories.setdefault(key, [])
self=TimeMap{'histories':{...:...}}; key='x'; value='b'; timestamp=4; history=[(4, 'a')]

L11 set: history.append((timestamp, value))
self=TimeMap{'histories':{...:...}}; key='x'; value='b'; timestamp=4; history=[(4, 'a'), (4,
    'b')]

L14 get: history = self.histories.get(key, ())
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=4; history=[(4, 'a'), (4, 'b')]

L15 get: left, right = 0, len(history)
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=4; history=[(4, 'a'), (4, 'b')];
    left=0; right=2

L17 get: mid = (left + right) // 2
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=4; history=[(4, 'a'), (4, 'b')];
    left=0; right=2; mid=1
```

```
L21 get: left = mid + 1
self=TimeMap{'histories':{...:...}}; key='x'; timestamp=4; history=[(4, 'a'), (4, 'b')];
    left=2; right=2; mid=1
```

Result: [None, None, 'b']. The later set wins at an equal timestamp.

Why the solution is correct

Sorted timestamps create an eligible-then-too-late predicate boundary. Each branch discards only indices whose relation to that boundary is known, preserving the half-open candidate interval and reducing its width. At convergence, left-1 is exactly the last eligible entry, including the latest insertion among equal timestamps.

Time and space

Expected amortized O(1) set and O(log(h+1)) get for a key with h versions, with O(total_versions) storage. Dictionary access assumes expected constant time.

Common mistakes

Returning the lower-bound entry can select a future value or the first tie. Building [time for time,value in history] inside get makes the query linear. Out-of-order sets violate the append-only search invariant.

Reasoning card

State: Each history is ordered by timestamp and insertion order within ties. During search, [left,right) contains the first-greater boundary; indices below left are eligible and indices at or above right are too late.

Invariant: Each key history remains timestamp-sorted.

Decision to rehearse: Append in time order and binary-search each key history.

Finish: Return history[left-1]'s value if left>0, else ''

Transfer exercise

Support out-of-order timestamps. Compare sorted-list insertion, balanced search trees and batched sorting in terms of write/read costs.

A. Diagnose the implementation gap

When an attempt fails, record the smallest failing input and the first state that no longer matches its meaning. Avoid writing only "off-by-one" or "need more practice." Those labels do not identify the decision to rehearse.

Model: you solved a different problem, such as a substring instead of a subsequence, a walk instead of a simple path, or ordered sequences instead of unordered combinations. Rewrite the output contract and one distinguishing example.

State: a variable mixed two meanings, such as height and diameter, current-window validity and historical maximum, or a previous DP layer and a current one. Rename the quantities and write their meanings at one specific point in the loop.

Timing: you stored the current element before a lookup that required earlier elements, committed before repairing a window, or marked a graph vertex too late. Write the exact ingest, update, repair and commit order, then trace two adjacent iterations.

Boundary: an inclusive endpoint was treated as exclusive, a binary-search update did not shrink the interval, or a leaf was confused with one missing child. Draw the smallest empty, singleton and two-candidate states.

Ownership: a recursive branch did not undo its mutation, a result retained a mutable path reference, or a linked-list rewrite lost the next node. Name the owner of every changed object and the point at which it is restored or transferred.

Schedule: a DP state read an unfinished predecessor, an in-place relaxation reused one item, or a graph algorithm used the wrong frontier. Draw the dependency arrows and choose a processing order that respects them.

Cost: the algorithm was correct but hidden slicing, list insertion, hashing or repeated subtree work made the claimed complexity wrong. Count the actual operations in this language, including output materialization.

B. A repeatable interview rehearsal

Use a blank editor and a problem statement with its topic hidden. First restate the contract aloud. Ask only questions that change the model: duplicates, order, mutation, endpoint semantics, missing answers or constraints. Work one example manually before choosing the mechanism.

State a simple baseline, then identify its repeated work. Derive the improved state from that waste. Write a protocol and one invariant sentence before implementation. Translate the protocol in order rather than typing a remembered code shape and hoping its variables acquire meaning.

During implementation, keep the state sentence visible. When adding a branch, say what it proves and why the discarded alternatives are safe. For recursion, say what the child returns and which mutations belong to the current frame.

Validate with a normal example, the smallest legal input, an equality/tie case and an adversarial case tied to the proof. Explain time, auxiliary space, output size and recursion depth. If time remains, discuss one changed constraint that would invalidate the mechanism.

After the rehearsal, score five abilities from zero to two: contract clarity, derivation, executable translation, verification and explanation. Zero means missing, one means achieved with a hint, and two means independent. Use the pattern of scores to choose the next practice action. The score is a personal learning instrument, not a hiring prediction.

C. Retrieval map

Prefix evidence: scans and maps, 001-020. Ask what the processed prefix can summarize and whether the current item belongs in that summary before a query.

Ordered frontiers: sorting, two pointers and windows, 021-050. Ask what ordering or contiguity permits you to discard, compact or repair.

Boundary algebra: prefixes, differences and sweeps, 051-060 and 091-100. Ask whether an interval can be represented by two boundary events or two prefix states.

Unfinished candidates: stacks, monotonic structures, queues and heaps, 061-090. Ask why a candidate is removed: resolved, dominated, expired or selected.

References and search intervals: linked lists and binary search, 101-120. Ask which objects or candidate positions remain reachable after the next update.

Coordinates and traversal: matrices and grid search, 121-140. Ask which neighbors exist, when discovery is marked and whether updates need original cell state.

Calls and choices: recursion, backtracking and trees, 141-180. Ask what one call promises, what it owns and whether its state must be restored.

Connectivity and dependency order: graphs, DAGs, DSU and weighted frontiers, 181-210. Ask whether the task is reachability, enumeration, ordering, connection cost or route cost.

Optimal substructure: greedy arguments and DP, 211-240. Ask whether one choice dominates alternatives, or whether several complete predecessor answers must be compared.

Persistent invariants: bits, mathematics and designed structures, 241-250. Ask which algebraic identity or cross-structure relationship survives every operation.

D. Glossary of working terms

Invariant: a statement true at a named point before and after every iteration or operation. It must connect maintained state to the problem, not merely describe the code.

Candidate interval: the positions in which a sought result or boundary can still lie. A half-open interval [left,right) includes left and excludes right; its width is right-left.

Monotone predicate: a boolean condition whose value changes at most once over an ordered search domain. It justifies searching for a first true or first false boundary.

Commit point: the moment maintained state is valid and complete enough to update or emit an answer. A repaired window and a completed recursive path have different commit conditions.

Frontier: discovered work waiting to be processed. FIFO, LIFO, minimum-priority and monotonic frontiers imply different ordering guarantees.

Dominance: one candidate is at least as useful as another for every possible future continuation, allowing the weaker candidate to be discarded safely.

Amortized cost: the total cost of a sequence divided across its operations. A particular operation may be expensive even when the average per operation is constant.

State axis: one dimension needed to identify a subproblem, such as prefix length, remaining budget, vertex or capacity. Omitting a necessary axis merges situations with different futures.

Pseudopolynomial: polynomial in a numeric value such as capacity, but potentially exponential in the number of bits used to encode that value.

Output-sensitive work: runtime or memory that necessarily depends on how many answers or output characters are produced. Enumerating exponentially many subsets cannot take linear total time.

Aliasing: two references designate the same mutable object. Saving a path without copying it and allocating matrix rows by repeated references are common accidental forms.

Stable ordering: equal-key records retain their earlier relative order. It is a property of how ties are handled, not simply of whether the final keys are sorted.

E. Running the examples and keeping editions aligned

Use Python 3.10 or newer. The solution files themselves use only the standard library. Download the complete solution bundle and keep its files together: some later lessons deliberately import an earlier kata or the shared structures module.

From the solutions directory, start Python and import the named function or class, then paste an example call. List and tree examples use the constructors and serializers printed in the next appendix. Stateful examples use run_operations: the first argument is a class, the second is a sequence of method-name/argument tuples, and any remaining arguments initialize the class. The returned list includes None for mutation methods.

The package README gives exact rebuild commands. tools/check.py regenerates all Python files and executes every published example. tests/independent_checks.py exercises bounded independent oracles and stateful invariants. tools/build_book.py generates the chapter Markdown, website and PDF from the verified lesson records.

For a correction, edit the relevant authoring/module_NN.py record, not just a generated Python or HTML file. Re-run verification and rebuild every edition. Preserve kata IDs so links, review notes and progress records remain meaningful.

Printed code may wrap long lines for page layout; the downloadable Python preserves original line breaks. Trace line numbers refer to those downloadable files. Original text diagrams are schematic rather than to scale. Local browser progress stays on that browser and is not a cross-device learning record.

F. Shared Python structures

These constructors, serializers and operation helpers make the book examples reproducible. Keep structures.py beside the kata files.

```
"""Shared interview data types and small example adapters used by the book."""
from collections import deque

class ListNode:
    def __init__(self, val=0, next=None, random=None):
        self.val, self.next, self.random = val, next, random

def build_list(values):
    dummy = ListNode()
    tail = dummy
    for value in values:
        tail.next = ListNode(value)
        tail = tail.next
    return dummy.next

def list_values(head):
    output, seen = [], set()
    while head is not None:
        if head in seen:
            raise ValueError('Cannot serialize a cyclic chain as a finite list')
        seen.add(head)
        output.append(head.val)
        head = head.next
    return output

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val, self.left, self.right = val, left, right

def build_tree(values):
    if not values or values[0] is None:
        return None
    root = TreeNode(values[0])
    queue = deque([root])
    i = 1
    while queue and i < len(values):
        node = queue.popleft()
        for side in ('left', 'right'):
            if i < len(values) and values[i] is not None:
                child = TreeNode(values[i])
                setattr(node, side, child)
                queue.append(child)
            i += 1
    return root

def tree_values(root):
    if root is None:
        return []
    output, queue = [], deque([root])
    while queue:
```

```
        node = queue.popleft()
        if node is None:
            output.append(None)
        else:
            output.append(node.val)
            queue.extend((node.left, node.right))
    while output and output[-1] is None:
        output.pop()
    return output

class GraphNode:
    def __init__(self, val=0, neighbors=None):
        self.val = val
        self.neighbors = [] if neighbors is None else neighbors

def run_operations(factory, operations, *constructor_args):
    """Return the result of every named operation, including None for mutations."""
    instance = factory(*constructor_args)
    output = []
    for operation in operations:
        method, *args = operation
        output.append(getattr(instance, method)(*args))
    return output
```