The DSA Woodshed

Printable Reference Packet

July 23, 2026

Contents

1 Decision Trees & Patterns **4**

1.1 Master Decision Tree . . . 4
1.2 Pattern Recognition Keywords . . . 5
1.3 Data Structure Selection . . . 5
1.4 When Stuck . . . 5
1.5 Common Mistakes . . . 6

2 Arrays & Hashing **7**

2.1 Group Anagrams — group strings that are anagrams of each other . . . 8
2.2 Product of Array Except Self — product of all elements except self . . . 9
2.3 Top K Frequent Elements — return the k most frequent elements . . . 10
2.4 Two Sum — find two indices whose values sum to target . . . 12

3 Two Pointers **13**

3.1 Container With Most Water — max area between two vertical lines . . . 14
3.2 3Sum — find all unique triplets that sum to zero . . . 15
3.3 Trapping Rain Water — total water trapped between bars . . . 16

4 Sliding Window **18**

4.1 Longest substring without repeating characters . . . 19
4.2 Minimum window substring . . . 20

5 Stacks & Queues **22**

5.1 Daily temperatures . . . 23
5.2 Min stack . . . 24
5.3 Valid parentheses . . . 26

6 Searching **27**

6.1 Binary search . . . 28
6.2 Find minimum in rotated sorted array . . . 30
6.3 Search in rotated sorted array . . . 31

7 Sorting **32**

7.1 Merge Sort Inversions — count inversions using merge sort . . . 33
7.2 Quickselect — find the kth smallest element . . . 35

8 Linked Lists **37**

8.1 LRU Cache implementation . . . 38
8.2 Merge two sorted linked lists . . . 40
8.3 Reverse a singly linked list . . . 42

9 Trees **44**

9.1 Invert (mirror) a binary tree . . . 45
9.2 Level-order (BFS) traversal of a binary tree . . . 46
9.3 Maximum depth of a binary tree . . . 47
9.4 Trie (prefix tree) for efficient string operations . . . 48
9.5 Validate binary search tree . . . 51

10 Graphs **52**
10.1 A* pathfinding on a weighted grid . . . 53
10.2 Shortest paths allowing negative edge weights . . . 55
10.3 Deep copy an undirected graph . . . 57
10.4 Determine if all courses can be finished (cycle detection) . . . 59
10.5 Single-source shortest paths with non-negative edge weights . . . 61
10.6 Geohash encoding, decoding, and neighbor lookup . . . 63
10.7 K-dimensional tree for nearest neighbor search . . . 66
10.8 Minimum spanning tree: Kruskal's and Prim's algorithms . . . 69
10.9 Time for a signal to reach all nodes (Dijkstra variant) . . . 71
10.10 Maximum flow via Edmonds-Karp (BFS-based Ford-Fulkerson) . . . 73
10.11 Count islands in a 2D grid of '1's (land) and '0's (water) . . . 75
10.12 Topological ordering of a directed acyclic graph (DAG) . . . 77
10.13 Shortest transformation sequence from beginWord to endWord . . . 79

11 Dynamic Programming **82**
11.1 Climbing Stairs — ways to climb n stairs taking 1 or 2 steps . . . 83
11.2 Coin Change — minimum coins to make amount . . . 85
11.3 Constraint Satisfaction Problem — generic CSP solver with Sudoku example . . . 86
11.4 Edit Distance — minimum operations to convert word1 to word2 . . . 89
11.5 0/1 Knapsack — maximize value within weight capacity . . . 91
11.6 Longest Common Subsequence — length of LCS of two strings . . . 93
11.7 Longest Increasing Subsequence — length of LIS . . . 95
11.8 Traveling Salesman Problem — bitmask DP solution . . . 97

12 Heaps **99**
12.1 Find the kth largest element . . . 100
12.2 Merge k sorted linked lists . . . 102
12.3 Task scheduler with cooldown . . . 104

13 Backtracking **106**
13.1 Combination Sum — find combinations that sum to target . . . 107
13.2 N-Queens — place n queens on n x n board with no attacks . . . 108
13.3 Permutations — generate all permutations of a list . . . 109
13.4 Subsets — generate all subsets of a set . . . 110

14 Greedy **111**
14.1 Interval Scheduling — maximum non-overlapping intervals . . . 112
14.2 Jump Game — can you reach the last index? / minimum jumps . . . 113

14.3 Merge Intervals — merge overlapping intervals . . . 115

15 Strings **116**

15.1 Longest Common Prefix — find the longest common prefix among strings . . . 117
15.2 Longest Palindromic Substring — find the longest palindromic substring . . . 118
15.3 String to Integer (atoi) — convert a string to a 32-bit signed integer . . . 119
15.4 Valid Anagram — check if two strings are anagrams of each other . . . 121
15.5 Valid Palindrome — check if string is a palindrome ignoring non-alnum and case . . . 122

16 Recursion **123**

16.1 Flatten Nested List — recursively flatten arbitrarily deep lists . . . 124
16.2 Generate Parentheses — all valid combinations of n pairs . . . 126
16.3 Letter Combinations of a Phone Number — digit-to-letter mapping . . . 127
16.4 Pow(x, n) — compute x raised to the power n using fast exponentiation . . . 129
16.5 Tower of Hanoi — move n disks between pegs . . . 131

17 Bit Manipulation **132**

17.1 Counting Bits — count 1-bits for all numbers 0..n . . . 133
17.2 Reverse Bits — reverse the bits of a 32-bit unsigned integer . . . 134
17.3 Single Number — find the element appearing once (others twice) . . . 135

18 Math & Number Theory **136**

18.1 Check whether one integer is prime . . . 137
18.2 Sieve of Eratosthenes: all primes up to n . . . 138

19 Patterns **140**

19.1 Sliding window pattern template . . . 141

20 Appendix: Interview Topics **142**

20.1 Concurrency & Parallelism Primitives . . . 143
20.2 Hash Table Internals . . . 146
20.3 Amortized Analysis . . . 149
20.4 Recursion to Iteration Conversion . . . 152
20.5 Fixed-Width & Numeric Pitfalls (C/C++/Java vs Python) . . . 155

Decision Trees & Patterns

Master Decision Tree

```
What does the problem ask for?
|
+-- FIND / SEARCH
|   +-- Input sorted?
|   |   +-- YES --> Binary Search
|   |   +-- Rotated? --> Search Rotated Array
|   |   +-- NO --> Hash Map O(n) lookup
|   +-- Find in a graph?
|       +-- Unweighted --> BFS
|       +-- Weighted (positive) --> Dijkstra
|       +-- Weighted (negative) --> Bellman-Ford
|       +-- With heuristic --> A* Search
|
+-- OPTIMIZE (min cost, max profit, count ways)
|   +-- Overlapping subproblems?
|   |   +-- YES --> Dynamic Programming
|   |   |   +-- 1D: climbing_stairs, coin_change
|   |   |   +-- 2D: edit_distance, longest_common_subseq
|   |   |   +-- Bitmask: traveling_salesman_dp
|   |   +-- NO --> Greedy (merge_intervals, jump_game)
|   +-- Top-K or priority ordering?
|       +-- Kth element --> Min-heap of size k
|       +-- Merge K sorted --> Heap merge
|
+-- GENERATE ALL / ENUMERATE
|   +-- All subsets --> backtracking/subsets
|   +-- All permutations --> backtracking/permutations
|   +-- Combinations to sum --> backtracking/combination_sum
|   +-- Placement with rules --> backtracking/n_queens
|
+-- PROCESS A SEQUENCE (subarray, substring)
|   +-- Contiguous?
|   |   +-- YES --> Sliding Window
|   |   |   +-- Fixed size: standard pattern
|   |   |   +-- Variable: min_window_substring
|   |   +-- NO --> Subsequence DP (LIS, LCS)
|   +-- Next greater/smaller? --> Monotonic Stack
|   +-- Valid nesting? --> Stack (valid_parentheses)
|
+-- GRAPH STRUCTURE
|   +-- Connected components --> DFS/BFS / Union-Find
|   +-- Dependency ordering --> Topological Sort
|   +-- Cycle detection --> DFS coloring / course_schedule
|   +-- Min spanning tree --> Kruskal's MST
|   +-- Maximum flow --> Edmonds-Karp
|
+-- DESIGN A DATA STRUCTURE
    +-- O(1) access + eviction --> LRU Cache
    +-- O(1) min/max retrieval --> Min Stack
    +-- Sorted insert + search --> bisect / SortedList
```

Pattern Recognition Keywords

Keyword	First Try	Fallback
sorted	Binary search, two pointers	–
contiguous subarray	Sliding window	Prefix sums, Kadane's
substring	Sliding window + hash map	DP
parentheses / brackets	Stack	–
next greater / warmer	Monotonic stack	–
shortest path	BFS / Dijkstra	Bellman-Ford, A*
connected / island	DFS/BFS flood fill	Union-Find
dependency / prerequisite	Topological sort	–
cycle	DFS coloring / Union-Find	–
minimum cost / count ways	Dynamic programming	–
all subsets / combinations	Backtracking	Bitmask
merge / overlapping intervals	Sort + greedy sweep	–
kth largest / top k	Heap of size k	Quickselect
design / implement	Choose data structures	–
cache / eviction	Hash map + linked list	–
stream / online	Heap, sliding window	–
frequency / how many	Counter / hash map	–
edit distance / transform	2D DP	BFS (word ladder)
knapsack / subset sum	DP (1D or 2D)	–
XOR / single unique	Bit manipulation	–
nearest point / proximity	KD-tree, geohash	–
visit all cities/nodes	TSP bitmask DP	–

Data Structure Selection

```
Need key -> value?          --> dict
Need "is X in the set?"     --> set
Need min/max repeatedly?    --> heapq
Need FIFO?                  --> collections.deque
Need LIFO?                  --> list (append/pop)
Need sorted insert+search?  --> bisect / SortedList
Need merge/find groups?     --> Union-Find
Need nearest in 2D/3D?      --> KD-tree or geohash
```

When Stuck

```
1. INPUT SIZE? --> Determines max acceptable complexity
2. OUTPUT? --> Boolean=search, Value=optimize, All=backtrack
3. Can I SORT? --> Unlocks two pointers, binary search, greedy
4. HASH MAP? --> Trade space for O(1) lookup
5. OPTIMAL SUBSTRUCTURE + OVERLAPPING? --> DP
   Optimal substructure only? --> Greedy
6. GRAPH STRUCTURE? --> BFS/DFS/Dijkstra
7. INTERVALS / EVENTS? --> Sort by end (schedule) or start (merge)
```

Common Mistakes

Mistake	Fix
list.pop(0) in a loop	Use deque.popleft() – O(1)
s += char in a loop	parts.append(char); ”.join(parts)
Mutable default arg def f(a=[])	Use a=None; a = a or []
Modifying list while iterating	Iterate over a copy
Off-by-one in binary search	Check lo <= hi vs lo < hi
Forgetting to copy in backtrack	result.append(path[:])
@lru_cache with mutable args	Convert lists to tuples first
DFS hitting recursion limit	setrecursionlimit or iterative
Dijkstra with negative weights	Use Bellman-Ford instead
BFS: mark visited late	Mark when ENQUEUING, not dequeuing
// with negatives	-7 // 2 = -4; use int(-7/2) = -3

Arrays & Hashing

Group Anagrams — group strings that are anagrams of each other

Problem

Given a list of strings, group the anagrams together. An anagram is a word formed by rearranging the letters of another.

Approach

Use the sorted characters of each string as a hash key. All anagrams produce the same sorted tuple, so they land in the same bucket in a defaultdict. Alternate group_anagrams_countkey keys on a 26-length character-count vector instead of sorting, trading the sort for a fixed-size linear scan.

When to Use

Grouping items by equivalence class — anagrams, isomorphic strings, etc. Keywords: "group by", "categorize", "bucket by canonical form".

Complexity

```
Time:  O(n * k log k) where n = number of strings, k = max string length
Space:  O(n * k)
```

Implementation

```
from collections import defaultdict


def group_anagrams(strs: list[str]) -> list[list[str]]:
    """Return groups of anagram strings.

    >>> sorted(
    ...     sorted(g)
    ...     for g in group_anagrams(["eat", "tea", "tan", "ate", "nat", "bat"])
    ... )
    [['ate', 'eat', 'tea'], ['bat'], ['nat', 'tan']]
    """
    groups: defaultdict[tuple[str, ...], list[str]] = defaultdict(list)
    for s in strs:
        key = tuple(sorted(s))  # tuple, not list -- the key must be hashable
        groups[key].append(s)
    return list(groups.values())


# --- character-count key variant (avoids sorting each string) ---
def group_anagrams_countkey(strs: list[str]) -> list[list[str]]:
    """Group anagrams using a 26-length lowercase character-count vector.

    Assumes lowercase a-z input (as is typical for this problem). Trades
    the O(k log k) sort for an O(k) count pass per string.

    >>> sorted(
    ...     sorted(g)
    ...     for g in group_anagrams_countkey(["eat", "tea", "tan", "ate", "nat", "bat"])
    ... )
    [['ate', 'eat', 'tea'], ['bat'], ['nat', 'tan']]
    """
    groups: defaultdict[tuple[int, ...], list[str]] = defaultdict(list)
    for s in strs:
        counts = [0] * 26
        for ch in s:
            counts[ord(ch) - ord("a")] += 1  # counts as key sidesteps the sort entirely
        groups[tuple(counts)].append(s)
    return list(groups.values())
```

Product of Array Except Self — product of all elements except self

Problem

Given an integer array, return an array where each element is the product of all other elements. Do not use division.

Approach

Two-pass prefix/suffix products. First pass builds prefix products left-to-right, second pass multiplies in suffix products right-to-left. Uses the output array itself to store prefix, then folds in suffix with a running variable. Alternate product_except_self_accumulate builds the same prefix/suffix products with itertools.accumulate.

When to Use

Prefix/suffix accumulation without division — product, sum, or any associative operation where you need "everything except index i". Also: running totals, range queries without a segment tree.

Complexity

```
Time:  O(n)
Space:  O(1) (output array not counted)
```

Implementation

```
from collections.abc import Sequence
from itertools import accumulate
from operator import mul


def product_except_self(nums: Sequence[int]) -> list[int]:
    """Return array of products of all elements except nums[i].

    >>> product_except_self([1, 2, 3, 4])
    [24, 12, 8, 6]
    """
    n = len(nums)
    result = [1] * n

    # result[] doubles as the prefix table, so no extra O(n) array is needed
    prefix = 1
    for i in range(n):
        result[i] = prefix
        prefix *= nums[i]

    # fold in the suffix product via a running variable, right to left
    suffix = 1
    for i in range(n - 1, -1, -1):
        result[i] *= suffix
        suffix *= nums[i]

    return result


# --- itertools.accumulate prefix/suffix variant (stdlib, O(n) extra space) ---
def product_except_self_accumulate(nums: Sequence[int]) -> list[int]:
    """Return array of products of all elements except nums[i], via accumulate.

    >>> product_except_self_accumulate([1, 2, 3, 4])
    [24, 12, 8, 6]
    """
    n = len(nums)
    prefix = [1, *accumulate(nums, mul)][:n]
    suffix = [1, *accumulate(reversed(nums), mul)][:n][::-1]
    return [p * s for p, s in zip(prefix, suffix, strict=True)]
```

Top K Frequent Elements — return the k most frequent elements

Problem

Given an integer array and an integer k, return the k most frequent elements. The answer is guaranteed to be unique.

Approach

Bucket sort by frequency. Count occurrences, then place each number into a bucket indexed by its frequency. Walk buckets from highest frequency downward until k elements are collected. Alternate top_k_frequent_heap gets the same answer with heapq.nlargest.

When to Use

Frequency counting + selection — "top K", "most common", "least common". Bucket sort avoids O(n log n); see also heaps/kth_largest for streaming variant.

Complexity

```
Time:  O(n)
Space:  O(n)
```

Implementation

```
import heapq
from collections import Counter
from collections.abc import Sequence


def top_k_frequent(nums: Sequence[int], k: int) -> list[int]:
    """Return the *k* most frequent elements in *nums*.

    >>> sorted(top_k_frequent([1, 1, 1, 2, 2, 3], 2))
    [1, 2]
    """
    if k <= 0:
        return []

    count = Counter(nums)
    # bucket index = frequency; frequency is bounded by len(nums), so len(nums)+1 buckets
    buckets: list[list[int]] = [[] for _ in range(len(nums) + 1)]
    for num, freq in count.items():
        buckets[freq].append(num)

    result: list[int] = []
    for i in range(len(buckets) - 1, -1, -1):
        result.extend(buckets[i])
        if len(result) >= k:
            return result[:k]

    return result[:k]


# --- heapq.nlargest one-liner (O(n log k), stdlib) ---
def top_k_frequent_heap(nums: Sequence[int], k: int) -> list[int]:
    """Return the *k* most frequent elements in *nums*, via heapq.nlargest.

    >>> sorted(top_k_frequent_heap([1, 1, 1, 2, 2, 3], 2))
    [1, 2]
    """
    if k <= 0:
        return []

    count = Counter(nums)
```

```
    return heapq.nlargest(k, count.keys(), key=count.__getitem__)
```

Two Sum — find two indices whose values sum to target

Problem

Given an array of integers and a target, return the indices of the two numbers that add up to the target. Each input has exactly one solution and you may not use the same element twice.

Approach

Single-pass hash map. For each element, compute the complement (target - current). If the complement is already in the map, return both indices. Otherwise store current value -> index.

When to Use

Any problem asking "find pair with property X" — hash map for O(1) lookups. Also: complement problems, two-number sum/difference/product.

Complexity

```
Time:  O(n)
Space:  O(n)
```

Implementation

```
from collections.abc import Sequence


def two_sum(nums: Sequence[int], target: int) -> tuple[int, int]:
    """Return indices of two elements that sum to *target*.

    >>> two_sum([2, 7, 11, 15], 9)
    (0, 1)
    """
    seen: dict[int, int] = {}
    for i, n in enumerate(nums):
        complement = target - n
        if complement in seen:
            return (seen[complement], i)
        seen[n] = i  # store after the lookup so we never pair an element with itself

    msg = "no two elements sum to target"
    raise ValueError(msg)
```

Two Pointers

Container With Most Water — max area between two vertical lines

Problem

Given n non-negative integers representing vertical line heights at positions 0..n-1, find the two lines that together with the x-axis form a container holding the most water.

Approach

Start with two pointers at the outermost lines. The area is limited by the shorter line, so always move the pointer at the shorter side inward to try for a taller line.

When to Use

Maximizing area/product with two boundaries — shrink from both ends, always moving the weaker constraint inward. Keywords: "maximize rectangle", "widest pair", "two-boundary optimization".

Complexity

```
Time:  O(n)
Space:  O(1)
```

Implementation

```
from collections.abc import Sequence


def max_area(height: Sequence[int]) -> int:
    """Return the maximum water area between any two lines in *height*.

    >>> max_area([1, 8, 6, 2, 5, 4, 8, 3, 7])
    49
    """
    lo, hi = 0, len(height) - 1
    best = 0

    while lo < hi:
        area = min(height[lo], height[hi]) * (hi - lo)
        best = max(best, area)
        # the shorter line caps the area no matter what -- moving it inward is
        # the only move that can possibly find something taller
        if height[lo] < height[hi]:
            lo += 1
        else:
            hi -= 1

    return best
```

3Sum — find all unique triplets that sum to zero

Problem

Given an integer array, return all unique triplets [a, b, c] such that a + b + c = 0. The solution must not contain duplicate triplets.

Approach

Sort the array. Fix one element and use two pointers on the remaining subarray to find pairs that sum to the negation of the fixed element. Skip duplicate values to avoid repeated triplets.

When to Use

Reducing N-sum to 2-sum on a sorted array — "find triplets/k-tuples with property X". Fix one element, two-pointer scan the rest. Generalizes to k-sum by recursion down to the two-pointer base case.

Complexity

```
Time:  O(n^2)
Space:  O(1) (excluding output)
```

Implementation

```
from collections.abc import Sequence


def three_sum(nums: Sequence[int]) -> list[list[int]]:
    """Return all unique triplets in *nums* that sum to zero.

    >>> three_sum([-1, 0, 1, 2, -1, -4])
    [[-1, -1, 2], [-1, 0, 1]]
    """
    sorted_nums = sorted(nums)
    n = len(sorted_nums)
    result: list[list[int]] = []

    for i in range(n - 2):
        # Skip duplicate fixed element
        if i > 0 and sorted_nums[i] == sorted_nums[i - 1]:
            continue

        lo, hi = i + 1, n - 1
        while lo < hi:
            total = sorted_nums[i] + sorted_nums[lo] + sorted_nums[hi]
            if total < 0:
                lo += 1
            elif total > 0:
                hi -= 1
            else:
                result.append([sorted_nums[i], sorted_nums[lo], sorted_nums[hi]])
                lo += 1
                # Skip duplicate left pointer
                while lo < hi and sorted_nums[lo] == sorted_nums[lo - 1]:
                    lo += 1

    return result
```

Trapping Rain Water — total water trapped between bars

Problem

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water can be trapped after raining.

Approach

Two pointers from both ends. Track left_max and right_max. Move the pointer on the side with the smaller max inward. Water at each position is (current_side_max - height[pointer]). Alternate trap_prefix_suffix precomputes left/right max arrays with itertools.accumulate instead of tracking them with two pointers.

When to Use

Bounded accumulation between barriers — water trapping, histogram area, or any problem where capacity at each position depends on the max values to its left and right. Also: elevation profile analysis.

Complexity

```
Time:  O(n)
Space:  O(1)
```

Implementation

```
from collections.abc import Sequence
from itertools import accumulate


def trap(height: Sequence[int]) -> int:
    """Return total units of water trapped by the elevation map.

    >>> trap([0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1])
    6
    """
    if len(height) < 3:
        return 0

    lo, hi = 0, len(height) - 1
    lo_max = hi_max = 0
    water = 0

    while lo < hi:
        # the smaller running max is the one that actually bounds the water at
        # its pointer -- the far side is guaranteed to be at least as tall
        if height[lo] < height[hi]:
            lo_max = max(lo_max, height[lo])
            water += lo_max - height[lo]
            lo += 1
        else:
            hi_max = max(hi_max, height[hi])
            water += hi_max - height[hi]
            hi -= 1

    return water


# --- prefix/suffix max arrays via itertools.accumulate (O(n) space) ---
def trap_prefix_suffix(height: Sequence[int]) -> int:
    """Return total units of water trapped, via precomputed max arrays.

    >>> trap_prefix_suffix([0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1])
    6
    """
```

```
n = len(height)
if n < 3:
    return 0

left_max = list(accumulate(height, max))
right_max = list(accumulate(reversed(height), max))[::-1]
return sum(min(left_max[i], right_max[i]) - height[i] for i in range(n))
```

Sliding Window

Longest substring without repeating characters

Problem

Given a string s, find the length of the longest substring without repeating characters.

Approach

Sliding window with a dict tracking the last-seen index of each character. When a duplicate is found within the current window, jump the left pointer past the previous occurrence.

When to Use

Longest valid window — "longest substring/subarray without repeats", "maximum window satisfying constraint". Expand right, contract left only when the window becomes invalid. Streaming uniqueness checks.

Complexity

```
Time:  O(n) - single pass through the string
Space:  O(min(n, alphabet_size)) for the last-seen dict
```

Implementation

```
def length_of_longest_substring(s: str) -> int:
    """Return the length of the longest substring with no repeating chars.

    >>> length_of_longest_substring("abcabcbb")
    3
    >>> length_of_longest_substring("bbbbb")
    1
    >>> length_of_longest_substring("")
    0
    """
    seen: dict[str, int] = {}
    left = best = 0

    for right, ch in enumerate(s):
        # seen[ch] may be stale (outside the current window) -- the >= left
        # check stops left from ever moving backward on a stale hit
        if ch in seen and seen[ch] >= left:
            left = seen[ch] + 1
        seen[ch] = right
        best = max(best, right - left + 1)

    return best
```

Minimum window substring

Problem

Given strings s and t, find the minimum window substring of s that contains all characters of t (including duplicates). Return "" if no such window exists.

Approach

Variable-size sliding window with Counter for "need" and "have" tracking. Expand the right pointer to include characters, shrink the left pointer to minimize the window once all characters are satisfied.

When to Use

Minimum window containing all required elements — "smallest substring with all chars", "shortest subarray covering set". Expand right to satisfy, shrink left to minimize. Streaming log/event filtering.

Complexity

```
Time:  O(n) where n = len(s) - each character visited at most twice
Space:  O(k) where k = number of unique characters in t
```

Implementation

```
from collections import Counter


def min_window(s: str, t: str) -> str:
    """Return the minimum window in *s* that contains all chars of *t*.

    >>> min_window("ADOBECODEBANC", "ABC")
    'BANC'
    >>> min_window("a", "a")
    'a'
    >>> min_window("a", "aa")
    ''
    """
    if not t or not s:
        return ""

    need: Counter[str] = Counter(t)
    missing = len(t)
    left = start = end = 0

    for right, ch in enumerate(s, 1):  # right is 1-indexed
        if need[ch] > 0:
            missing -= 1
        need[ch] -= 1

        if missing == 0:
            # Shrink from left while window stays valid
            while need[s[left]] < 0:
                need[s[left]] += 1
                left += 1

            # end starts at 0, and a real window's end is always >= 1 here,
            # so "not end" reliably detects "no window recorded yet"
            if not end or right - left < end - start:
                start, end = left, right

            # deliberately invalidate the window so the loop keeps searching
            # for a smaller one, mirroring the shrink accounting above
            need[s[left]] += 1
            missing += 1
```

```
            left += 1

    return s[start:end]
```

Stacks & Queues

Daily temperatures

Problem

Given an array of daily temperatures, return an array where each element is the number of days you would have to wait until a warmer temperature. If there is no future warmer day, put 0.

Approach

Monotonic decreasing stack of indices. For each temperature, pop all stack entries whose temperature is lower than the current one and record the distance. Alternate daily_temperatures_brute checks each future day directly.

When to Use

Next-greater-element pattern — "next warmer day", "next higher price", "first element greater than X to the right". Monotonic stack scans linearly. Also: stock span, histogram largest rectangle.

Complexity

```
Time:  O(n) - each index pushed and popped at most once
Space:  O(n) - stack in worst case (strictly decreasing input)
```

Implementation

```
from collections.abc import Sequence


def daily_temperatures(temps: Sequence[int]) -> list[int]:
    """Return days until a warmer temperature for each day.

    >>> daily_temperatures([73, 74, 75, 71, 69, 72, 76, 73])
    [1, 1, 4, 2, 1, 1, 0, 0]
    >>> daily_temperatures([30, 40, 50, 60])
    [1, 1, 1, 0]
    """
    result = [0] * len(temps)
    stack: list[int] = []  # indices with decreasing temps

    for i, t in enumerate(temps):
        while stack and temps[stack[-1]] < t:  # current day is warmer than stack top
            j = stack.pop()
            result[j] = i - j  # distance from j's day to the first warmer day found
        stack.append(i)

    return result


# --- brute-force alternate: scan forward for the first warmer day (O(n^2)) ---
def daily_temperatures_brute(temps: Sequence[int]) -> list[int]:
    """Return days until a warmer temperature via a direct forward scan.

    >>> daily_temperatures_brute([73, 74, 75, 71, 69, 72, 76, 73])
    [1, 1, 4, 2, 1, 1, 0, 0]
    """
    n = len(temps)
    return [
        next((j - i for j in range(i + 1, n) if temps[j] > temps[i]), 0)
        for i in range(n)
    ]
```

Min stack

Problem

Design a stack that supports push, pop, top, and retrieving the minimum element, all in O(1) time.

Approach

Store (value, current_min) tuples on the stack. Each entry records the minimum at the time it was pushed, so getMin is always O(1) by reading the top tuple's min field. Alternate MinStackTwoStacks tracks the running minimum in a separate auxiliary stack instead.

When to Use

Augmented data structure for O(1) aggregate queries — "get min/max while supporting push/pop". Pattern: store auxiliary data alongside each element. Useful for real-time monitoring dashboards.

Complexity

```
Time:  O(1) for all operations
Space:  O(n) - one tuple per element
```

Implementation

```
class MinStack:
    """Stack supporting O(1) push, pop, top, and get_min.

    >>> ms = MinStack()
    >>> ms.push(-2)
    >>> ms.push(0)
    >>> ms.push(-3)
    >>> ms.get_min()
    -3
    >>> ms.pop()
    >>> ms.get_min()
    -2
    """

    def __init__(self) -> None:
        self._stack: list[tuple[int, int]] = []

    def push(self, val: int) -> None:
        # store the running min alongside val; a single shared min var would
        # break once that min is popped, since we couldn't recover the prior one
        current_min = min(val, self._stack[-1][1]) if self._stack else val
        self._stack.append((val, current_min))

    def pop(self) -> None:
        if not self._stack:
            msg = "pop from empty stack"
            raise IndexError(msg)
        self._stack.pop()

    def top(self) -> int:
        if not self._stack:
            msg = "top from empty stack"
            raise IndexError(msg)
        return self._stack[-1][0]

    def get_min(self) -> int:
        if not self._stack:
            msg = "get_min from empty stack"
            raise IndexError(msg)
        return self._stack[-1][1]
```

```
# --- two-stack alternate: separate stack tracks the running minimum ---
class MinStackTwoStacks:
    """Stack supporting O(1) push, pop, top, and get_min via an aux min stack.

    >>> ms = MinStackTwoStacks()
    >>> ms.push(-2)
    >>> ms.push(0)
    >>> ms.push(-3)
    >>> ms.get_min()
    -3
    >>> ms.pop()
    >>> ms.get_min()
    -2
    """

    def __init__(self) -> None:
        self._stack: list[int] = []
        self._min_stack: list[int] = []

    def push(self, val: int) -> None:
        self._stack.append(val)
        if not self._min_stack or val <= self._min_stack[-1]:
            # ties must be pushed too, else popping one min desyncs the other
            self._min_stack.append(val)

    def pop(self) -> None:
        if not self._stack:
            msg = "pop from empty stack"
            raise IndexError(msg)
        val = self._stack.pop()
        if self._min_stack and val == self._min_stack[-1]:
            self._min_stack.pop()

    def top(self) -> int:
        if not self._stack:
            msg = "top from empty stack"
            raise IndexError(msg)
        return self._stack[-1]

    def get_min(self) -> int:
        if not self._min_stack:
            msg = "get_min from empty stack"
            raise IndexError(msg)
        return self._min_stack[-1]
```

Valid parentheses

Problem

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. A string is valid if every open bracket is closed by the same type of bracket in the correct order.

Approach

Use a stack. Push opening brackets; on a closing bracket, check that the stack top matches. The string is valid iff the stack is empty at the end.

When to Use

Matching/nesting validation — balanced brackets, HTML tags, expression parsing. Any problem where openers must be closed in LIFO order. Keywords: "valid", "balanced", "nested", "well-formed".

Complexity

```
Time:  O(n) - single pass
Space:  O(n) - stack in worst case (all openers)
```

Implementation

```
def is_valid(s: str) -> bool:
    """Return True if *s* contains valid matched brackets.

    >>> is_valid("()[]{}")
    True
    >>> is_valid("(]")
    False
    >>> is_valid("([)]")
    False
    """
    stack: list[str] = []
    match = {")": "(", "]": "[", "}": "{"}

    for ch in s:
        if ch in match:
            # empty stack means an orphan closer; mismatch means wrong nesting
            if not stack or stack[-1] != match[ch]:
                return False
            stack.pop()
        else:
            stack.append(ch)

    return not stack
```

Searching

Binary search

Problem

Given a sorted array of integers and a target value, return the index of the target if found, otherwise return -1.

Approach

Both iterative and recursive implementations. Maintain lo/hi bounds; compute mid = lo + (hi - lo) // 2 to avoid overflow.

When to Use

Sorted array lookup or any monotonic predicate search — "find target", "first/last occurrence", "search insert position". Foundation for bisect-based optimizations.

Complexity

```
Time:  O(log n)
Space:  O(1) iterative, O(log n) recursive (call stack)
```

Implementation

```
from collections.abc import Sequence


def binary_search(nums: Sequence[int], target: int) -> int:
    """Return the index of *target* in sorted *nums*, or -1 if absent.

    >>> binary_search([1, 3, 5, 7, 9], 5)
    2
    >>> binary_search([1, 3, 5, 7, 9], 4)
    -1
    >>> binary_search([], 1)
    -1
    """
    lo, hi = 0, len(nums) - 1

    while lo <= hi:
        mid = lo + (hi - lo) // 2  # overflow-safe: avoids (lo + hi) overflowing
        if nums[mid] == target:
            return mid
        if nums[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1

    return -1


def binary_search_recursive(nums: Sequence[int], target: int) -> int:
    """Recursive binary search returning index or -1.

    >>> binary_search_recursive([2, 4, 6, 8, 10], 8)
    3
    >>> binary_search_recursive([2, 4, 6, 8, 10], 5)
    -1
    """

    def _helper(lo: int, hi: int) -> int:
        if lo > hi:
            return -1
        mid = lo + (hi - lo) // 2  # overflow-safe midpoint, same as the iterative version
        if nums[mid] == target:
            return mid
```

```
        if nums[mid] < target:
            return _helper(mid + 1, hi)
        return _helper(lo, mid - 1)

    return _helper(0, len(nums) - 1)
```

Find minimum in rotated sorted array

Problem

Given a rotated sorted array of unique elements, find the minimum element.

Approach

Binary search variant. If nums[mid] > nums[hi], the minimum is in the right half; otherwise it is in the left half (including mid). Converge until lo == hi.

When to Use

Pivot finding in a rotated sorted array — "find rotation point", "minimum in rotated". Compare mid vs right boundary to decide which half contains the pivot. See also: search_rotated_array.

Complexity

```
Time:  O(log n)
Space:  O(1)
```

Implementation

```
from collections.abc import Sequence


def find_min(nums: Sequence[int]) -> int:
    """Return the minimum element in a rotated sorted array.

    >>> find_min([3, 4, 5, 1, 2])
    1
    >>> find_min([4, 5, 6, 7, 0, 1, 2])
    0
    >>> find_min([1])
    1
    """
    lo, hi = 0, len(nums) - 1

    while lo < hi:
        mid = lo + (hi - lo) // 2
        if nums[mid] > nums[hi]:
            lo = mid + 1  # min is in the right half
        else:
            hi = mid  # mid could be the min

    return nums[lo]
```

Search in rotated sorted array

Problem

An array sorted in ascending order was rotated at some pivot. Given a target value, return its index or -1. All values are distinct.

Approach

Modified binary search. At each step, determine which half is sorted (compare nums[lo] to nums[mid]). Then check whether the target lies within the sorted half to decide which side to search.

When to Use

Invariant-based binary search — array is sorted but shifted/rotated. Identify which half is sorted, then decide which side to search. Keywords: "rotated sorted array", "shifted sequence", "cyclic order".

Complexity

```
Time:  O(log n)
Space:  O(1)
```

Implementation

```
from collections.abc import Sequence


def search_rotated(nums: Sequence[int], target: int) -> int:
    """Return the index of *target* in a rotated sorted array, or -1.

    >>> search_rotated([4, 5, 6, 7, 0, 1, 2], 0)
    4
    >>> search_rotated([4, 5, 6, 7, 0, 1, 2], 3)
    -1
    >>> search_rotated([1], 1)
    0
    """
    lo, hi = 0, len(nums) - 1

    while lo <= hi:
        mid = lo + (hi - lo) // 2

        if nums[mid] == target:
            return mid

        # Left half is sorted
        if nums[lo] <= nums[mid]:
            if nums[lo] <= target < nums[mid]:
                hi = mid - 1
            else:
                lo = mid + 1
        # Right half is sorted
        else:
            if nums[mid] < target <= nums[hi]:
                lo = mid + 1
            else:
                hi = mid - 1

    return -1
```

Sorting

Merge Sort Inversions — count inversions using merge sort

Problem

Count the number of inversions in an array. An inversion is a pair (i, j) where i < j but nums[i] > nums[j].

Approach

Modified merge sort. During the merge step, when an element from the right half is placed before elements remaining in the left half, those left-half elements all form inversions with it. Alternate count_inversions_brute checks every pair directly.

When to Use

Counting disorder in sequences — "number of inversions", "how far from sorted", "Kendall tau distance". Modified merge sort counts cross-half inversions during the merge step. Also: rank correlation.

Complexity

```
Time:  O(n log n)
Space:  O(n)
```

Implementation

```
from collections.abc import Sequence


def count_inversions(nums: Sequence[int]) -> int:
    """Return the number of inversions in *nums*.

    >>> count_inversions([2, 4, 1, 3, 5])
    3
    >>> count_inversions([5, 4, 3, 2, 1])
    10
    """
    if len(nums) <= 1:
        return 0

    arr = list(nums)
    _, count = _merge_sort(arr, 0, len(arr) - 1)
    return count


def _merge_sort(arr: list[int], lo: int, hi: int) -> tuple[list[int], int]:
    if lo >= hi:
        return arr, 0

    mid = (lo + hi) // 2
    _, left_inv = _merge_sort(arr, lo, mid)
    _, right_inv = _merge_sort(arr, mid + 1, hi)
    split_inv = _merge(arr, lo, mid, hi)

    return arr, left_inv + right_inv + split_inv


def _merge(arr: list[int], lo: int, mid: int, hi: int) -> int:
    left = arr[lo : mid + 1]
    right = arr[mid + 1 : hi + 1]

    inversions = 0
    i = j = 0
    k = lo

    while i < len(left) and j < len(right):
```

```
        if left[i] <= right[j]:
            arr[k] = left[i]
            i += 1
        else:
            arr[k] = right[j]
            inversions += len(left) - i  # every remaining left elem is > right[j]
            j += 1
        k += 1

    while i < len(left):
        arr[k] = left[i]
        i += 1
        k += 1

    while j < len(right):
        arr[k] = right[j]
        j += 1
        k += 1

    return inversions


# --- brute-force alternate: check every pair directly (O(n^2)) ---
def count_inversions_brute(nums: Sequence[int]) -> int:
    """Count inversions by checking every pair directly.

    >>> count_inversions_brute([2, 4, 1, 3, 5])
    3
    >>> count_inversions_brute([5, 4, 3, 2, 1])
    10
    """
    return sum(
        1
        for i in range(len(nums))
        for j in range(i + 1, len(nums))
        if nums[i] > nums[j]
    )
```

Quickselect — find the kth smallest element

Problem

Given an unsorted array and integer k, return the kth smallest element (1-indexed).

Approach

Lomuto partition scheme. Pick a pivot, partition the array so elements less than the pivot come before it. If the pivot lands at position k-1 we are done; otherwise recurse on the correct half. Randomized pivot for expected O(n). Alternate quickselect_heap gets the same answer with heapq.nsmallest.

When to Use

Selection without full sort — "kth smallest/largest", "median", "top K" when you don't need sorted output. O(n) average vs O(n log n) for full sort. See also: heaps/kth_largest for streaming.

Complexity

```
Time:  O(n) average, O(n^2) worst case
Space:  O(1) (in-place partitioning)
```

Implementation

```
import heapq
import random


def quickselect(nums: list[int], k: int) -> int:
    """Return the *k*-th smallest element (1-indexed).

    >>> quickselect([3, 2, 1, 5, 6, 4], 2)
    2
    >>> quickselect([7], 1)
    7
    """
    if k < 1 or k > len(nums):
        msg = f"k={k} out of range for length {len(nums)}"
        raise ValueError(msg)

    def _partition(lo: int, hi: int) -> int:
        # Randomized pivot to avoid worst case
        pivot_idx = random.randint(lo, hi)
        nums[pivot_idx], nums[hi] = nums[hi], nums[pivot_idx]
        pivot = nums[hi]
        store = lo
        for i in range(lo, hi):
            if nums[i] < pivot:
                nums[store], nums[i] = nums[i], nums[store]
                store += 1
        nums[store], nums[hi] = nums[hi], nums[store]
        return store

    lo, hi = 0, len(nums) - 1
    target = k - 1

    while lo <= hi:
        pivot_pos = _partition(lo, hi)
        if pivot_pos == target:
            return nums[pivot_pos]
        if pivot_pos < target:
            lo = pivot_pos + 1
        else:
            hi = pivot_pos - 1
```

```
    # Should not reach here if k is valid
    msg = "quickselect failed"
    raise RuntimeError(msg)


# --- heapq.nsmallest one-liner (O(n log k), stdlib) ---
def quickselect_heap(nums: list[int], k: int) -> int:
    """Return the *k*-th smallest element using heapq.nsmallest (stdlib).

    >>> quickselect_heap([3, 2, 1, 5, 6, 4], 2)
    2
    >>> quickselect_heap([7], 1)
    7
    """
    if k < 1 or k > len(nums):
        msg = f"k={k} out of range for length {len(nums)}"
        raise ValueError(msg)

    return heapq.nsmallest(k, nums)[-1]
```

Linked Lists

LRU Cache implementation

Problem

Design a data structure that follows the Least Recently Used (LRU) eviction policy, supporting get and put in O(1) time.

Approach

OrderedDict version: move accessed keys to end; pop from front on evict. Manual version: doubly linked list + hash map for O(1) removal/insertion.

When to Use

O(1) cache with eviction — "design LRU/LFU cache", bounded-memory caching with recency tracking. Pattern: hash map + doubly linked list.

Complexity

```
Time:  O(1) for both get and put
Space:  O(capacity)
```

Implementation

```
from collections import OrderedDict
from dataclasses import dataclass, field


class LRUCache:
    """LRU Cache using OrderedDict."""

    def __init__(self, capacity: int) -> None:
        if capacity <= 0:
            msg = f"capacity must be positive, got {capacity}"
            raise ValueError(msg)
        self._cache: OrderedDict[int, int] = OrderedDict()
        self._capacity = capacity

    def get(self, key: int) -> int:
        """Return value for key, or -1 if not found. Marks key as recently used."""
        if key not in self._cache:
            return -1
        self._cache.move_to_end(key)  # reads count as "recently used" too
        return self._cache[key]

    def put(self, key: int, value: int) -> None:
        """Insert or update key-value pair. Evicts LRU entry if at capacity."""
        if key in self._cache:
            self._cache.move_to_end(key)
        self._cache[key] = value
        if len(self._cache) > self._capacity:
            self._cache.popitem(last=False)  # last=False: evict the front (oldest)


# --- Manual doubly-linked-list implementation ---
@dataclass
class _DNode:
    """Internal node for the doubly linked list."""

    key: int = 0
    val: int = 0
    prev: _DNode | None = field(default=None, repr=False)
    next: _DNode | None = field(default=None, repr=False)
```

```
class LRUCacheManual:
    """LRU Cache using a doubly linked list + hash map."""

    def __init__(self, capacity: int) -> None:
        if capacity <= 0:
            msg = f"capacity must be positive, got {capacity}"
            raise ValueError(msg)
        self._capacity = capacity
        self._cache: dict[int, _DNode] = {}
        # Sentinel nodes simplify edge-case handling.
        self._head = _DNode()
        self._tail = _DNode()
        self._head.next = self._tail
        self._tail.prev = self._head

    def _remove(self, node: _DNode) -> None:
        """Unlink a node from the doubly linked list."""
        assert node.prev is not None
        assert node.next is not None
        node.prev.next = node.next
        node.next.prev = node.prev

    def _add_to_front(self, node: _DNode) -> None:
        """Insert a node right after the head sentinel (most recent)."""
        assert self._head.next is not None
        node.next = self._head.next
        node.prev = self._head
        self._head.next.prev = node
        self._head.next = node

    def get(self, key: int) -> int:
        """Return value for key, or -1 if not found."""
        if key not in self._cache:
            return -1
        node = self._cache[key]
        self._remove(node)
        self._add_to_front(node)
        return node.val

    def put(self, key: int, value: int) -> None:
        """Insert or update key-value pair. Evicts LRU entry if at capacity."""
        if key in self._cache:
            self._remove(self._cache[key])
        node = _DNode(key, value)
        self._cache[key] = node
        self._add_to_front(node)
        if len(self._cache) > self._capacity:
            assert self._tail.prev is not None
            lru = self._tail.prev
            self._remove(lru)
            del self._cache[lru.key]
```

Merge two sorted linked lists

Problem

Given the heads of two sorted linked lists, merge them into one sorted list built from the nodes of the two input lists.

Approach

Use a dummy head node and compare the fronts of both lists, advancing the smaller one each time. Alternate merge_two_sorted_recu builds the same result through recursive calls instead of an explicit loop.

When to Use

Merge step of merge sort — combining two sorted sequences into one. Building block for merge_k_sorted_lists and external merge sort. Keywords: "merge sorted", "interleave ordered streams".

Complexity

```
Time:  O(n + m)
Space:  O(1) -- only pointer manipulation, no new nodes allocated
```

Implementation

```
from dataclasses import dataclass


@dataclass
class ListNode:
    val: int
    next: ListNode | None = None


def merge_two_sorted(
    l1: ListNode | None,
    l2: ListNode | None,
) -> ListNode | None:
    """Merge two sorted linked lists into one sorted list.

    >>> to_list(merge_two_sorted(from_list([1, 3, 5]), from_list([2, 4, 6])))
    [1, 2, 3, 4, 5, 6]
    """
    dummy = ListNode(0)
    tail = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next

    tail.next = l1 if l1 else l2  # remainder is already sorted; splice it in directly
    return dummy.next


# --- recursive alternate: build the merge through the call stack ---
def merge_two_sorted_recursive(
    l1: ListNode | None,
    l2: ListNode | None,
) -> ListNode | None:
    """Merge two sorted linked lists using recursion.
```

```
    >>> to_list(merge_two_sorted_recursive(from_list([1, 3, 5]), from_list([2, 4, 6])))
    [1, 2, 3, 4, 5, 6]
    """
    if l1 is None:
        return l2
    if l2 is None:
        return l1

    if l1.val <= l2.val:
        l1.next = merge_two_sorted_recursive(l1.next, l2)
        return l1
    l2.next = merge_two_sorted_recursive(l1, l2.next)
    return l2


# --- helpers for testing ---
def from_list(vals: list[int]) -> ListNode | None:
    """Build a linked list from a Python list."""
    dummy = ListNode(0)
    curr = dummy
    for v in vals:
        curr.next = ListNode(v)
        curr = curr.next
    return dummy.next


def to_list(head: ListNode | None) -> list[int]:
    """Collect linked list values into a Python list."""
    result: list[int] = []
    while head:
        result.append(head.val)
        head = head.next
    return result
```

Reverse a singly linked list

Problem

Given the head of a singly linked list, reverse the list and return the new head.

Approach

Iterative: Walk the list with prev/curr pointers, reversing each link. Recursive: Reverse the rest of the list, then point the next node back.

When to Use

In-place linked list reversal — "reverse list", "reverse sublist", pointer manipulation without extra space. Building block for palindrome check, k-group reversal, and reorder-list problems.

Complexity

```
Time:  O(n)
Space:  O(1) iterative, O(n) recursive (call stack)
```

Implementation

```
from dataclasses import dataclass


@dataclass
class ListNode:
    val: int
    next: ListNode | None = None


def reverse_iterative(head: ListNode | None) -> ListNode | None:
    """Reverse a linked list in place using iteration.

    >>> to_list(reverse_iterative(from_list([1, 2, 3])))
    [3, 2, 1]
    """
    prev: ListNode | None = None
    curr = head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt
    return prev


# --- recursive alternate: reverse via the call stack instead of explicit pointers ---
def reverse_recursive(head: ListNode | None) -> ListNode | None:
    """Reverse a linked list using recursion.

    >>> to_list(reverse_recursive(from_list([1, 2, 3])))
    [3, 2, 1]
    """
    if not head or not head.next:
        return head
    new_head = reverse_recursive(head.next)
    head.next.next = head  # point the next node's next back at head
    head.next = None  # head becomes the new tail; must clear its old forward link
    return new_head


# --- helpers for testing ---
```

```
def from_list(vals: list[int]) -> ListNode | None:
    """Build a linked list from a Python list."""
    dummy = ListNode(0)
    curr = dummy
    for v in vals:
        curr.next = ListNode(v)
        curr = curr.next
    return dummy.next


def to_list(head: ListNode | None) -> list[int]:
    """Collect linked list values into a Python list."""
    result: list[int] = []
    while head:
        result.append(head.val)
        head = head.next
    return result
```

Trees

Invert (mirror) a binary tree

Problem

Given the root of a binary tree, invert the tree so that every left child becomes the right child and vice versa.

Approach

Recursive swap: at each node, swap left and right children, then recurse into both subtrees.

When to Use

Tree transformation — "mirror", "invert", "flip". Any problem requiring symmetric restructuring of a tree in-place. Pattern: swap children, then recurse into both subtrees.

Complexity

```
Time:  O(n)
Space:  O(h) where h = height of tree (call stack)
```

Implementation

```
from dataclasses import dataclass


@dataclass
class TreeNode:
    val: int
    left: TreeNode | None = None
    right: TreeNode | None = None


def invert_tree(root: TreeNode | None) -> TreeNode | None:
    """Invert a binary tree in place and return the root.

    >>> t = TreeNode(1, TreeNode(2), TreeNode(3))
    >>> r = invert_tree(t)
    >>> r.left.val, r.right.val  # type: ignore[union-attr]
    (3, 2)
    """
    if not root:
        return None
    # Simultaneous swap (no temp var); recursing afterward still inverts both
    # subtrees correctly since invert_tree treats left/right symmetrically.
    root.left, root.right = root.right, root.left
    invert_tree(root.left)
    invert_tree(root.right)
    # Mutates in place and returns the same root object, so callers can use
    # either the return value or the original reference interchangeably.
    return root
```

Level-order (BFS) traversal of a binary tree

Problem

Given the root of a binary tree, return the level-order traversal as a list of lists, where each inner list contains the values at that depth level.

Approach

BFS with a deque. Process one level at a time by iterating over the current queue length, appending children for the next level.

When to Use

BFS on trees — "level order", "zigzag order", "right side view", any problem requiring per-level processing. Also: shortest-path-like queries on trees, hierarchical data serialization.

Complexity

```
Time:  O(n)
Space:  O(w) where w = max width of the tree (up to n/2)
```

Implementation

```
from collections import deque
from dataclasses import dataclass


@dataclass
class TreeNode:
    val: int
    left: TreeNode | None = None
    right: TreeNode | None = None


def level_order(root: TreeNode | None) -> list[list[int]]:
    """Return level-order traversal as a list of lists.

    >>> level_order(TreeNode(3, TreeNode(9), TreeNode(20, TreeNode(15), TreeNode(7))))
    [[3], [9, 20], [15, 7]]
    """
    if not root:
        return []

    result: list[list[int]] = []
    queue: deque[TreeNode] = deque([root])

    while queue:
        level: list[int] = []
        # range(len(queue)) snapshots this level's size up front -- the loop
        # body pushes next-level nodes into the same queue as it runs.
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(level)

    return result
```

Maximum depth of a binary tree

Problem

Given the root of a binary tree, return its maximum depth (number of nodes along the longest path from root to a leaf).

Approach

DFS recursive: depth = 1 + max(depth(left), depth(right)). Base case: empty tree has depth 0.

When to Use

Recursive tree measurement — "max depth", "height", "diameter", any metric that aggregates over subtrees with a simple recurrence. Foundation for balanced-tree checks and tree diameter computation.

Complexity

```
Time:  O(n)
Space:  O(h) where h = height of tree (call stack)
```

Implementation

```
from dataclasses import dataclass


@dataclass
class TreeNode:
    val: int
    left: TreeNode | None = None
    right: TreeNode | None = None


def max_depth(root: TreeNode | None) -> int:
    """Return the maximum depth of a binary tree.

    >>> max_depth(TreeNode(1, TreeNode(2), TreeNode(3, TreeNode(4), None)))
    3
    """
    if not root:
        return 0
    # Depth counted in nodes (empty tree = 0, a single node = 1), not edges --
    # if asked for edge-count height instead, subtract 1 from the result.
    return 1 + max(max_depth(root.left), max_depth(root.right))
```

Trie (prefix tree) for efficient string operations

Problem

Implement a prefix tree that supports insert, search, prefix matching, delete, and autocomplete operations on a set of strings.

Approach

A tree where each node represents a character. Paths from the root to nodes marked as word-endings form the stored words. Children are stored in a dict keyed by character for O(1) branching.

When to Use

Autocomplete, spell checking, IP routing, prefix matching.

Complexity

```
Time:  O(m) per insert/search/starts_with/delete where m = word length.
Space:  O(N * m) where N = number of words, m = average word length.
```

Implementation

```
from dataclasses import dataclass, field


@dataclass
class TrieNode:
    children: dict[str, TrieNode] = field(default_factory=dict)
    is_end: bool = False


class Trie:
    """Prefix tree supporting insert, search, prefix matching, and autocomplete.

    >>> t = Trie()
    >>> t.insert("apple")
    >>> t.search("apple")
    True
    >>> t.starts_with("app")
    True
    >>> t.search("app")
    False
    """

    def __init__(self) -> None:
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        """Insert a word into the trie.

        >>> t = Trie()
        >>> t.insert("cat")
        >>> t.search("cat")
        True
        """
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.is_end = True

    def search(self, word: str) -> bool:
```

```
        """Return True if the exact word exists in the trie.

        >>> t = Trie()
        >>> t.insert("hello")
        >>> t.search("hello")
        True
        >>> t.search("hell")
        False
        """
        node = self._find_node(word)
        return node is not None and node.is_end

    def starts_with(self, prefix: str) -> bool:
        """Return True if any word in the trie starts with the given prefix.

        >>> t = Trie()
        >>> t.insert("hello")
        >>> t.starts_with("hel")
        True
        >>> t.starts_with("xyz")
        False
        """
        return self._find_node(prefix) is not None

    def delete(self, word: str) -> bool:
        """Remove a word from the trie, cleaning up empty nodes.

        Returns True if the word was found and deleted, False otherwise.

        >>> t = Trie()
        >>> t.insert("cat")
        >>> t.delete("cat")
        True
        >>> t.search("cat")
        False
        """
        return self._delete(self.root, word, 0)

    def autocomplete(self, prefix: str, limit: int = 10) -> list[str]:
        """Return up to `limit` words starting with the given prefix.

        >>> t = Trie()
        >>> for w in ["bar", "bat", "ball"]:
        ...     t.insert(w)
        >>> sorted(t.autocomplete("ba"))
        ['ball', 'bar', 'bat']
        """
        node = self._find_node(prefix)
        if node is None:
            return []
        results: list[str] = []
        self._collect(node, prefix, limit, results)
        return results

    def _find_node(self, prefix: str) -> TrieNode | None:
        """Traverse the trie following the prefix, returning the final node."""
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return None
            node = node.children[ch]
        return node
```

```
def _delete(self, node: TrieNode, word: str, depth: int) -> bool:
    """Recursively delete a word and prune empty branches."""
    if depth == len(word):
        if not node.is_end:
            return False
        # Unmark, don't delete: this node may still be a live prefix of
        # another stored word (e.g. deleting "app" when "apple" exists).
        node.is_end = False
        return True

    ch = word[depth]
    if ch not in node.children:
        return False

    deleted = self._delete(node.children[ch], word, depth + 1)
    if deleted:
        child = node.children[ch]
        # Only prune a child if it's both non-terminal AND childless --
        # pruning a node that ends another word or still has children
        # would corrupt the trie.
        if not child.is_end and not child.children:
            del node.children[ch]
    return deleted

def _collect(
    self,
    node: TrieNode,
    prefix: str,
    limit: int,
    results: list[str],
) -> None:
    """DFS to collect words under a node up to the limit."""
    if len(results) >= limit:
        return
    if node.is_end:
        results.append(prefix)
    for ch in sorted(node.children):
        # Re-check the limit inside the loop too: the recursive call
        # below can fill `results` mid-iteration, not just between calls.
        if len(results) >= limit:
            return
        self._collect(node.children[ch], prefix + ch, limit, results)
```

Validate binary search tree

Problem

Given the root of a binary tree, determine if it is a valid BST. A valid BST requires every node's value to be strictly between the values of its ancestors that constrain it.

Approach

Recursive with min/max bounds. At each node, check that the value is within (lo, hi) and recurse with tightened bounds.

When to Use

Constraint propagation through a tree — "validate BST", "check ordering invariant". Pass tightening bounds (lo, hi) down the recursion. Also: range-constrained tree problems, interval checks.

Complexity

```
Time:  O(n)
Space:  O(h) where h = height of tree (call stack)
```

Implementation

```
from dataclasses import dataclass


@dataclass
class TreeNode:
    val: int
    left: TreeNode | None = None
    right: TreeNode | None = None


def is_valid_bst(
    root: TreeNode | None,
    lo: float = float("-inf"),
    hi: float = float("inf"),
) -> bool:
    """Return True if the binary tree rooted at `root` is a valid BST.

    >>> is_valid_bst(TreeNode(2, TreeNode(1), TreeNode(3)))
    True
    >>> is_valid_bst(TreeNode(2, TreeNode(3), TreeNode(1)))
    False
    """
    if not root:
        return True
    # Strict inequalities: a BST forbids duplicate values, so val == lo or
    # val == hi is a violation, not a boundary pass.
    if not (lo < root.val < hi):
        return False
    # Tighten the bound each recursive call: left subtree's new ceiling is
    # this node's value; right subtree's new floor is this node's value.
    return is_valid_bst(root.left, lo, root.val) and is_valid_bst(
        root.right, root.val, hi
    )
```

Graphs

A* pathfinding on a weighted grid

Problem

Given a 2D grid where each cell has a non-negative traversal cost, find the shortest (least-cost) path from a start cell to a goal cell using A* search with Manhattan distance heuristic.

Approach

Priority queue ordered by f(n) = g(n) + h(n), where g is the cost so far and h is the Manhattan distance heuristic. Expand the node with lowest f; skip already-settled nodes.

When to Use

Shortest path when you have a good heuristic (estimated distance to goal). Better than Dijkstra when goal is known — avoids exploring irrelevant nodes. Use Manhattan distance for grids, great-circle distance for geospatial.

Note: Optimality requires an admissible heuristic (never overestimates the true remaining cost) and, for this no-closed-set form, a consistent one. Manhattan distance satisfies both on a 4-connected grid with per-cell cost >= 1, so the path is optimal when the goal is first popped.

Complexity

```
Time:  O(E log V) with a good heuristic (grid:  E ~ 4V)
Space:  O(V)
```

Implementation

```
import heapq


def manhattan_distance(a: tuple[int, int], b: tuple[int, int]) -> int:
    """Return the Manhattan distance between two grid points."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])


def a_star(
    grid: list[list[int]],
    start: tuple[int, int],
    goal: tuple[int, int],
) -> list[tuple[int, int]] | None:
    """Return the shortest path from *start* to *goal* on *grid*.

    ``grid[r][c]`` is the cost to enter cell (r, c). Returns a list of
    (row, col) coordinates from start to goal inclusive, or ``None`` if
    no path exists.

    >>> a_star([[1, 1, 1], [1, 1, 1], [1, 1, 1]], (0, 0), (2, 2))
    [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)]
    """
    if not grid or not grid[0]:
        return None

    rows, cols = len(grid), len(grid[0])
    sr, sc = start
    gr, gc = goal

    if not (0 <= sr < rows and 0 <= sc < cols):
        return None
    if not (0 <= gr < rows and 0 <= gc < cols):
        return None

    open_heap: list[tuple[int, int, int, int]] = [
        (manhattan_distance(start, goal), 0, sr, sc),
    ]
```

```
    came_from: dict[tuple[int, int], tuple[int, int]] = {}
    g_score: dict[tuple[int, int], int] = {start: 0}

    while open_heap:
        _f, g, r, c = heapq.heappop(open_heap)
        # First pop of goal is optimal only because Manhattan distance is
        # admissible+consistent on this 4-connected, cost>=1 grid.
        if (r, c) == goal:
            return _reconstruct_path(came_from, goal)

        # Stale-heap skip: a cheaper g for (r, c) was already relaxed
        # (heapq has no decrease-key), so this popped entry is obsolete.
        if g > g_score.get((r, c), float("inf")):
            continue

        for dr, dc in ((0, 1), (0, -1), (1, 0), (-1, 0)):
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols:
                ng = g + grid[nr][nc]
                if ng < g_score.get((nr, nc), float("inf")):
                    g_score[(nr, nc)] = ng
                    f = ng + manhattan_distance((nr, nc), goal)
                    heapq.heappush(open_heap, (f, ng, nr, nc))
                    came_from[(nr, nc)] = (r, c)

    return None


def _reconstruct_path(
    came_from: dict[tuple[int, int], tuple[int, int]],
    current: tuple[int, int],
) -> list[tuple[int, int]]:
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path
```

Shortest paths allowing negative edge weights

Problem

Given a weighted directed graph, find the shortest path from a source to all other vertices. Unlike Dijkstra, this handles negative edge weights and can detect negative-weight cycles.

Approach

Relax all edges V-1 times. After V-1 iterations, if any edge can still be relaxed, a negative cycle exists.

When to Use

Shortest paths with negative edge weights — currency arbitrage detection, cost networks with rebates/discounts. Detects negative cycles. Prefer Dijkstra when all weights are non-negative.

Complexity

```
Time:  O(V * E)
Space:  O(V)
```

Implementation

```
from collections.abc import Sequence

INF = float("inf")


class NegativeCycleError(Exception):
    """Raised when a negative-weight cycle is detected."""


def bellman_ford(
    num_nodes: int,
    edges: Sequence[tuple[int, int, float]],
    source: int,
) -> list[float]:
    """Return shortest distances from *source* to all nodes.

    *edges* is a list of (u, v, weight).
    Raises :class:`NegativeCycleError` if a negative cycle is
    reachable from *source*.

    >>> bellman_ford(4, [(0, 1, 1), (1, 2, 3), (0, 2, 10), (2, 3, 2)], 0)
    [0, 1, 4, 6]
    """
    dist: list[float] = [INF] * num_nodes
    dist[source] = 0

    # A shortest path visits at most V-1 edges (no repeated vertices), so
    # V-1 relaxation rounds are guaranteed to converge if no negative cycle.
    for _ in range(num_nodes - 1):
        updated = False
        for u, v, w in edges:
            # dist[u] < INF guard: skip relaxing from a still-unreached node
            # so we never treat "infinity" as a real distance to propagate.
            if dist[u] < INF and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                updated = True
        if not updated:
            break

    # If any edge can still relax after V-1 rounds, that relaxation could
    # only come from a negative-weight cycle reachable from source.
```

```
    for u, v, w in edges:
        if dist[u] < INF and dist[u] + w < dist[v]:
            raise NegativeCycleError(
                "Graph contains a negative-weight cycle reachable from source"
            )

    return dist
```

Deep copy an undirected graph

Problem

Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph. Each node contains a val and a list of its neighbors.

Approach

BFS with a hash map mapping original nodes to their clones. For each node dequeued, iterate its neighbors: clone unseen neighbors, then wire the cloned neighbor into the clone's neighbor list. Also provided: clone_graph_comprehension, which separates "discover every reachable node" (still a BFS loop) from "build the clones and wire their edges" (dict/list comprehensions) instead of doing both in one explicit loop.

When to Use

Graph deep copy — "clone graph", "copy linked structure with cycles". BFS/DFS with a hash map from original to clone prevents revisiting. Also: snapshotting mutable graph state, undo/redo systems.

Complexity

```
Time:  O(V + E)
Space:  O(V)
```

Implementation

```
from collections import deque


class GraphNode:
    """Node in an undirected graph."""

    def __init__(
        self,
        val: int = 0,
        neighbors: list[GraphNode] | None = None,
    ) -> None:
        self.val = val
        self.neighbors: list[GraphNode] = neighbors if neighbors is not None else []

    def __repr__(self) -> str:
        neighbor_vals = [n.val for n in self.neighbors]
        return f"GraphNode({self.val}, neighbors={neighbor_vals})"


def clone_graph(node: GraphNode | None) -> GraphNode | None:
    """Return a deep copy of the graph rooted at *node*.

    >>> n1, n2 = GraphNode(1), GraphNode(2)
    >>> n1.neighbors, n2.neighbors = [n2], [n1]
    >>> c = clone_graph(n1)
    >>> c is not n1 and c is not None and c.val == 1
    True
    """
    if node is None:
        return None

    # GraphNode defines no __eq__/__hash__, so this dict keys on object
    # identity -- the only way to tell two same-valued nodes apart and to
    # stop a cyclic graph from being traversed forever.
    clones: dict[GraphNode, GraphNode] = {node: GraphNode(node.val)}
    queue: deque[GraphNode] = deque([node])

    while queue:
```

```
        current = queue.popleft()
        for neighbor in current.neighbors:
            if neighbor not in clones:
                clones[neighbor] = GraphNode(neighbor.val)
                queue.append(neighbor)
            clones[current].neighbors.append(clones[neighbor])

    return clones[node]


# --- dict/list-comprehension form: discover nodes first, then build declaratively ---
def clone_graph_comprehension(node: GraphNode | None) -> GraphNode | None:
    """Return a deep copy of the graph, built via comprehensions after discovery.

    Same result as clone_graph, but splits the work into two steps: a BFS
    discovery pass that just collects every reachable node (traversal can't
    be a comprehension), then a dict comprehension to create the clones and
    a list comprehension per node to wire up their neighbor lists.

    >>> n1, n2 = GraphNode(1), GraphNode(2)
    >>> n1.neighbors, n2.neighbors = [n2], [n1]
    >>> c = clone_graph_comprehension(n1)
    >>> c is not n1 and c is not None and c.val == 1
    True
    """
    if node is None:
        return None

    all_nodes: list[GraphNode] = []
    seen: set[int] = {id(node)}
    queue: deque[GraphNode] = deque([node])
    while queue:
        current = queue.popleft()
        all_nodes.append(current)
        for neighbor in current.neighbors:
            if id(neighbor) not in seen:
                seen.add(id(neighbor))
                queue.append(neighbor)

    clones = {n: GraphNode(n.val) for n in all_nodes}
    for n in all_nodes:
        clones[n].neighbors = [clones[neighbor] for neighbor in n.neighbors]

    return clones[node]
```

Determine if all courses can be finished (cycle detection)

Problem

There are numCourses courses labeled 0..numCourses-1. Given a list of prerequisite pairs [course, prereq], determine if it is possible to finish all courses (i.e., the prerequisite graph is a DAG).

Approach

BFS topological sort (Kahn's algorithm). If the resulting order contains fewer than numCourses nodes, a cycle exists.

When to Use

Cycle detection in a directed graph — "can all tasks be completed?", "is the dependency graph a DAG?". Topological sort that checks for leftover nodes. See also: topological_sort for the ordering itself.

Complexity

```
Time:  O(V + E)
Space:  O(V + E)
```

Implementation

```
from collections import deque


def can_finish(num_courses: int, prerequisites: list[list[int]]) -> bool:
    """Return True if all courses can be completed.

    >>> can_finish(2, [[1, 0]])
    True
    >>> can_finish(2, [[1, 0], [0, 1]])
    False
    """
    adj: list[list[int]] = [[] for _ in range(num_courses)]
    in_degree = [0] * num_courses

    for course, prereq in prerequisites:
        # Edge direction is prereq -> course: prereq must be visited (taken)
        # before course; reversing this is the most common bug here.
        adj[prereq].append(course)
        in_degree[course] += 1

    queue: deque[int] = deque(i for i in range(num_courses) if in_degree[i] == 0)
    visited = 0

    while queue:
        node = queue.popleft()
        visited += 1
        for neighbor in adj[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # A cycle can only be detected by counting: nodes stuck in a cycle never
    # reach in-degree 0, so BFS finishes early with visited < num_courses.
    return visited == num_courses


def find_order(num_courses: int, prerequisites: list[list[int]]) -> list[int]:
    """Return a valid course order, or [] if impossible.

    >>> find_order(4, [[1, 0], [2, 0], [3, 1], [3, 2]])
    [0, 1, 2, 3]
```

```
    """
    adj: list[list[int]] = [[] for _ in range(num_courses)]
    in_degree = [0] * num_courses

    for course, prereq in prerequisites:
        adj[prereq].append(course)
        in_degree[course] += 1

    queue: deque[int] = deque(i for i in range(num_courses) if in_degree[i] == 0)
    order: list[int] = []

    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in adj[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Same cycle check as can_finish: nodes on a cycle never hit in-degree 0,
    # so a short order here means a cycle exists.
    if len(order) != num_courses:
        return []
    return order
```

Single-source shortest paths with non-negative edge weights

Problem

Given a weighted directed graph and a source vertex, find the shortest path from the source to every other reachable vertex.

Approach

Dijkstra's algorithm using a min-heap (priority queue). Greedily expand the nearest unvisited node and relax its outgoing edges.

When to Use

Shortest path with NON-NEGATIVE weights. Network latency, road navigation, logistics routing. For negative weights use Bellman-Ford instead.

Complexity

```
Time:  O((V + E) log V)
Space:  O(V + E)
```

Implementation

```
import heapq
from collections.abc import Sequence

# Infinity sentinel
INF = float("inf")


def dijkstra(
    num_nodes: int,
    edges: Sequence[tuple[int, int, float]],
    source: int,
) -> list[float]:
    """Return shortest distances from *source* to all nodes.

    *edges* is a list of (u, v, weight) with weight >= 0.
    Unreachable nodes have distance ``float('inf')``.

    >>> dijkstra(4, [(0, 1, 1), (0, 2, 4), (1, 2, 2), (1, 3, 6), (2, 3, 3)], 0)
    [0, 1, 3, 6]
    """
    adj: list[list[tuple[int, float]]] = [[] for _ in range(num_nodes)]
    for u, v, w in edges:
        adj[u].append((v, w))

    dist: list[float] = [INF] * num_nodes
    dist[source] = 0
    heap: list[tuple[float, int]] = [(0, source)]

    while heap:
        d, u = heapq.heappop(heap)
        # Stale-heap skip: a shorter (u, d) was already popped and relaxed
        # earlier (heapq has no decrease-key), so this entry is obsolete.
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(heap, (nd, v))
```

```
    return dist
```

Geohash encoding, decoding, and neighbor lookup

Problem

Encode a (latitude, longitude) pair into a geohash string of a given precision. Decode a geohash back to a bounding box. Find the eight surrounding geohash cells.

Approach

Interleave bits of longitude and latitude ranges, then encode every 5 bits as a base-32 character. Decoding reverses the process. Neighbors are found by decoding to center, nudging into the adjacent cell, and re-encoding.

When to Use

Spatial indexing for proximity queries — "find nearby points", "group by geographic region". Prefix-matching on geohash strings gives fast bounding-box lookups. See also: kd_tree for exact NN.

Implementation

```
_BASE32 = "0123456789bcdefghjkmnpqrstuvwxyz"
_DECODE_MAP: dict[str, int] = {c: i for i, c in enumerate(_BASE32)}


def encode(lat: float, lng: float, precision: int = 12) -> str:
    """Encode latitude/longitude into a geohash string.

    >>> encode(42.6, -5.6, 5)
    'ezs42'
    """
    lat_range = (-90.0, 90.0)
    lng_range = (-180.0, 180.0)
    is_lng = True
    bits = 0
    char_index = 0
    result: list[str] = []

    while len(result) < precision:
        if is_lng:
            mid = (lng_range[0] + lng_range[1]) / 2
            if lng >= mid:
                char_index = (char_index << 1) | 1
                lng_range = (mid, lng_range[1])
            else:
                char_index = char_index << 1
                lng_range = (lng_range[0], mid)
        else:
            mid = (lat_range[0] + lat_range[1]) / 2
            if lat >= mid:
                char_index = (char_index << 1) | 1
                lat_range = (mid, lat_range[1])
            else:
                char_index = char_index << 1
                lat_range = (lat_range[0], mid)

        # Bits accumulate MSB-first (each shift makes room for the next bit);
        # decode() must consume bits in this same MSB-first order.
        is_lng = not is_lng
        bits += 1

        if bits == 5:
            result.append(_BASE32[char_index])
            bits = 0
            char_index = 0
```

```
    return "".join(result)


def decode(geohash: str) -> tuple[tuple[float, float], tuple[float, float]]:
    """Decode a geohash into a bounding box.

    Returns ((lat_min, lat_max), (lng_min, lng_max)).

    >>> lat_range, lng_range = decode("ezs42")
    >>> round(lat_range[0], 1), round(lng_range[0], 1)
    (42.6, -5.6)
    """
    lat_range = [-90.0, 90.0]
    lng_range = [-180.0, 180.0]
    is_lng = True

    for ch in geohash:
        cd = _DECODE_MAP[ch]
        # Mask order (16 down to 1) reads bits MSB-first, matching how
        # encode() packed them into each base32 character.
        for mask in (16, 8, 4, 2, 1):
            if is_lng:
                mid = (lng_range[0] + lng_range[1]) / 2
                if cd & mask:
                    lng_range[0] = mid
                else:
                    lng_range[1] = mid
            else:
                mid = (lat_range[0] + lat_range[1]) / 2
                if cd & mask:
                    lat_range[0] = mid
                else:
                    lat_range[1] = mid
            is_lng = not is_lng

    return (lat_range[0], lat_range[1]), (lng_range[0], lng_range[1])


def decode_center(geohash: str) -> tuple[float, float]:
    """Decode a geohash to its center (lat, lng).

    >>> lat, lng = decode_center("ezs42")
    >>> round(lat, 1), round(lng, 1)
    (42.6, -5.6)
    """
    (lat_min, lat_max), (lng_min, lng_max) = decode(geohash)
    return (lat_min + lat_max) / 2, (lng_min + lng_max) / 2


def neighbor(geohash: str, direction: str) -> str:
    """Return the geohash of the neighbor in *direction*.

    *direction* is one of 'n', 's', 'e', 'w', 'ne', 'nw', 'se', 'sw'.

    >>> neighbor("ezs42", "n")
    'ezs48'
    """
    if not geohash:
        msg = "Cannot compute neighbor of empty geohash"
        raise ValueError(msg)

    precision = len(geohash)
```

```
    (lat_min, lat_max), (lng_min, lng_max) = decode(geohash)
    lat_delta = lat_max - lat_min
    lng_delta = lng_max - lng_min
    center_lat = (lat_min + lat_max) / 2
    center_lng = (lng_min + lng_max) / 2

    dlat = 0.0
    dlng = 0.0
    for ch in direction:
        if ch == "n":
            dlat += lat_delta
        elif ch == "s":
            dlat -= lat_delta
        elif ch == "e":
            dlng += lng_delta
        elif ch == "w":
            dlng -= lng_delta

    new_lat = center_lat + dlat
    new_lng = center_lng + dlng

    # Clamp latitude
    new_lat = max(-89.999999, min(89.999999, new_lat))
    # Wrap longitude
    if new_lng > 180.0:
        new_lng -= 360.0
    elif new_lng < -180.0:
        new_lng += 360.0

    return encode(new_lat, new_lng, precision)


def neighbors(geohash: str) -> dict[str, str]:
    """Return all eight neighbors of a geohash cell.

    >>> sorted(neighbors("ezs42").keys())
    ['e', 'n', 'ne', 'nw', 's', 'se', 'sw', 'w']
    """
    return {
        d: neighbor(geohash, d) for d in ("n", "s", "e", "w", "ne", "nw", "se", "sw")
    }
```

K-dimensional tree for nearest neighbor search

Problem

Given a set of k-dimensional points, build a data structure that supports efficient nearest-neighbor queries.

Approach

Recursively partition points by cycling through dimensions, splitting on the median. For nearest-neighbor queries, traverse the tree pruning branches whose bounding hyperplane is farther than the current best distance.

When to Use

Nearest neighbor in multi-dimensional space — "closest point", "k-nearest neighbors", range search in 2D/3D. See also: geohash_grid for grid-based spatial indexing.

Complexity

`Space:  O(n)`

Implementation

```
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from collections.abc import Sequence

type Point = tuple[float, ...]


class KDNode:
    """A node in a KD-tree."""

    __slots__ = ("axis", "left", "point", "right")

    def __init__(
        self,
        point: Point,
        axis: int,
        left: KDNode | None = None,
        right: KDNode | None = None,
    ) -> None:
        self.point = point
        self.axis = axis
        self.left = left
        self.right = right


class KDTree:
    """K-dimensional tree for nearest neighbor search.

    >>> tree = KDTree([(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)])
    >>> tree.nearest((5, 5))
    (5, 4)
    """

    def __init__(self, points: Sequence[Point]) -> None:
        if not points:
            self.root: KDNode | None = None
            self._k = 0
        else:
            self._k = len(points[0])
            self.root = self._build(list(points), depth=0)
```

```
    def _build(self, points: list[Point], depth: int) -> KDNode | None:
        if not points:
            return None

        axis = depth % self._k
        points.sort(key=lambda p: p[axis])
        mid = len(points) // 2

        return KDNode(
            point=points[mid],
            axis=axis,
            left=self._build(points[:mid], depth + 1),
            right=self._build(points[mid + 1 :], depth + 1),
        )

    def nearest(self, target: Point) -> Point | None:
        """Return the nearest point to *target*, or None if tree is empty."""
        if self.root is None:
            return None

        best: list[tuple[float, Point]] = [(float("inf"), target)]
        self._search(self.root, target, best)
        return best[0][1]

    def _search(
        self,
        node: KDNode | None,
        target: Point,
        best: list[tuple[float, Point]],
    ) -> None:
        if node is None:
            return

        dist = _squared_distance(node.point, target)
        if dist < best[0][0]:
            best[0] = (dist, node.point)

        axis = node.axis
        diff = target[axis] - node.point[axis]

        # Search the side of the splitting plane that contains target first
        near = node.left if diff <= 0 else node.right
        far = node.right if diff <= 0 else node.left

        self._search(near, target, best)

        # Prune: only descend into the far subtree if its splitting plane is
        # closer than the current best -- compare squared distances throughout
        # to avoid an extra sqrt.
        if diff * diff < best[0][0]:
            self._search(far, target, best)

    def range_search(self, target: Point, radius: float) -> list[Point]:
        """Return all points within *radius* of *target*."""
        results: list[Point] = []
        r_sq = radius * radius
        self._range_search(self.root, target, r_sq, results)
        return results

    def _range_search(
        self,
        node: KDNode | None,
```

```
        target: Point,
        r_sq: float,
        results: list[Point],
    ) -> None:
        if node is None:
            return

        dist = _squared_distance(node.point, target)
        if dist <= r_sq:
            results.append(node.point)

        axis = node.axis
        diff = target[axis] - node.point[axis]

        near = node.left if diff <= 0 else node.right
        far = node.right if diff <= 0 else node.left

        self._range_search(near, target, r_sq, results)
        # Same pruning idea as nearest-neighbor search: skip the far subtree
        # unless its splitting plane could still be within the radius.
        if diff * diff <= r_sq:
            self._range_search(far, target, r_sq, results)


def _squared_distance(a: Point, b: Point) -> float:
    return sum((ai - bi) ** 2 for ai, bi in zip(a, b, strict=True))
```

Minimum spanning tree: Kruskal's and Prim's algorithms

Problem

Given an undirected weighted graph, find a subset of edges that connects all vertices with minimum total weight and no cycles.

Approach

Kruskal: Sort edges by weight, greedily add edges that don't form a cycle (checked via Union-Find). Prim: Grow the MST from an arbitrary node using a min-heap.

When to Use

Minimum cost to connect all nodes — cable/road/pipeline routing, network backbone design. Kruskal for sparse graphs, Prim for dense.

Implementation

```
import heapq
from collections.abc import Sequence


class UnionFind:
    """Disjoint-set / Union-Find with path compression and union by rank."""

    def __init__(self, n: int) -> None:
        self.parent = list(range(n))
        self.rank = [0] * n
        self.components = n

    def find(self, x: int) -> int:
        """Find the root of *x* with path compression."""
        while self.parent[x] != x:
            # Path halving: point at the grandparent each step instead of
            # the true root, shortening the path on every future find() too.
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def union(self, x: int, y: int) -> bool:
        """Merge sets containing *x* and *y*. Return False if already same set."""
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        # Union by rank: always hang the lower-rank tree under the
        # higher-rank root, keeping trees shallow.
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        self.components -= 1
        return True


def kruskal(
    num_nodes: int,
    edges: Sequence[tuple[int, int, float]],
) -> list[tuple[int, int, float]]:
    """Return MST edges using Kruskal's algorithm.

    *edges* is a list of (u, v, weight) for an undirected graph.
    Returns the MST as a list of (u, v, weight) edges.
```

```
    >>> kruskal(4, [(0, 1, 1), (1, 2, 2), (0, 2, 3), (2, 3, 4)])
    [(0, 1, 1), (1, 2, 2), (2, 3, 4)]
    """
    sorted_edges = sorted(edges, key=lambda e: e[2])
    uf = UnionFind(num_nodes)
    mst: list[tuple[int, int, float]] = []

    for u, v, w in sorted_edges:
        if uf.union(u, v):
            mst.append((u, v, w))
            # A spanning tree on num_nodes vertices has exactly num_nodes-1
            # edges; stop as soon as we have them instead of scanning the rest.
            if len(mst) == num_nodes - 1:
                break

    return mst


def prim(
    num_nodes: int,
    edges: Sequence[tuple[int, int, float]],
) -> list[tuple[int, int, float]]:
    """Return MST edges using Prim's algorithm.

    *edges* is a list of (u, v, weight) for an undirected graph.

    >>> sorted(
    ...     prim(4, [(0, 1, 1), (1, 2, 2), (0, 2, 3), (2, 3, 4)]), key=lambda e: e[2]
    ... )
    [(0, 1, 1), (1, 2, 2), (2, 3, 4)]
    """
    adj: list[list[tuple[int, float]]] = [[] for _ in range(num_nodes)]
    for u, v, w in edges:
        adj[u].append((v, w))
        adj[v].append((u, w))

    in_mst = [False] * num_nodes
    mst: list[tuple[int, int, float]] = []
    heap: list[tuple[float, int, int]] = [(0, -1, 0)]

    while heap and len(mst) < num_nodes - 1:
        w, frm, to = heapq.heappop(heap)
        # Stale-heap skip: `to` may already be settled via a cheaper edge
        # pushed earlier (heapq has no decrease-key), so ignore leftovers.
        if in_mst[to]:
            continue
        in_mst[to] = True
        if frm != -1:
            mst.append((frm, to, w))
        for neighbor, weight in adj[to]:
            if not in_mst[neighbor]:
                heapq.heappush(heap, (weight, to, neighbor))

    return mst
```

Time for a signal to reach all nodes (Dijkstra variant)

Problem

Given a network of n nodes (1-indexed) and directed weighted edges “times[i] = (u, v, w)“, send a signal from node k. Return the minimum time for all nodes to receive the signal, or -1 if not all nodes are reachable.

Approach

Run Dijkstra from source k. The answer is the maximum shortest distance among all nodes. If any node is unreachable, return -1.

When to Use

"Can a signal/message reach all nodes, and how long?" — broadcast latency, all-nodes reachability. Run Dijkstra, answer is max(dist).

Complexity

```
Time:  O((V + E) log V)
Space:  O(V + E)
```

Implementation

```
import heapq

INF = float("inf")


def network_delay_time(
    times: list[tuple[int, int, int]],
    n: int,
    k: int,
) -> int:
    """Return minimum time for signal from *k* to reach all *n* nodes.

    Nodes are 1-indexed.  Returns -1 if any node is unreachable.

    >>> network_delay_time([(2, 1, 1), (2, 3, 1), (3, 4, 1)], 4, 2)
    2
    """
    adj: list[list[tuple[int, int]]] = [[] for _ in range(n + 1)]
    for u, v, w in times:
        adj[u].append((v, w))

    dist: list[float] = [INF] * (n + 1)
    dist[k] = 0
    heap: list[tuple[float, int]] = [(0, k)]

    while heap:
        d, u = heapq.heappop(heap)
        # Stale-heap skip: a shorter distance to u was already relaxed
        # (heapq has no decrease-key), so this popped entry is obsolete.
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(heap, (nd, v))

    # dist[1:] drops the unused 0-index slot (nodes are 1-indexed); the
    # signal has "reached all nodes" only once the *slowest* one hears it.
    max_dist = max(dist[1:])
```

```
    return int(max_dist) if max_dist < INF else -1
```

Maximum flow via Edmonds-Karp (BFS-based Ford-Fulkerson)

Problem

Given a directed graph with edge capacities, find the maximum flow from a source node to a sink node.

Approach

Edmonds-Karp: repeatedly find augmenting paths using BFS on the residual graph. Each BFS finds the shortest augmenting path, guaranteeing polynomial time.

When to Use

Maximum throughput — "max bandwidth", "maximum matching", "supply chain optimization". Model as source -> sink capacity network.

Complexity

```
Time:  O(V * E^2)
Space:  O(V^2) for the capacity matrix
```

Implementation

```
from collections import deque
from sys import maxsize


def edmonds_karp(
    num_nodes: int,
    edges: list[tuple[int, int, int]],
    source: int,
    sink: int,
) -> int:
    """Return the maximum flow from *source* to *sink*.

    *edges* is a list of (u, v, capacity).

    >>> edmonds_karp(
    ...     4, [(0, 1, 10), (0, 2, 10), (1, 3, 10), (2, 3, 10), (1, 2, 1)], 0, 3
    ... )
    20
    """
    capacity: list[list[int]] = [[0] * num_nodes for _ in range(num_nodes)]
    adj: list[list[int]] = [[] for _ in range(num_nodes)]

    for u, v, cap in edges:
        # += (not =): parallel edges between the same u, v must accumulate
        # capacity rather than one silently overwriting another.
        capacity[u][v] += cap
        adj[u].append(v)
        adj[v].append(u)  # reverse edge for residual graph

    total_flow = 0

    while True:
        parent = _bfs(adj, capacity, source, sink, num_nodes)
        if parent is None:
            break

        # Find bottleneck along the path
        path_flow = maxsize
        node = sink
        while node != source:
            prev = parent[node]
```

```
            path_flow = min(path_flow, capacity[prev][node])
            node = prev

        # Update residual capacities
        node = sink
        while node != source:
            prev = parent[node]
            # Residual update: subtract along the path we used, credit the
            # reverse edge so a later augmenting path can "undo" this flow.
            capacity[prev][node] -= path_flow
            capacity[node][prev] += path_flow
            node = prev

        total_flow += path_flow

    return total_flow


def _bfs(
    adj: list[list[int]],
    capacity: list[list[int]],
    source: int,
    sink: int,
    num_nodes: int,
) -> list[int] | None:
    """BFS to find an augmenting path. Returns parent array or None."""
    parent = [-1] * num_nodes
    # Sentinel: parent[source] = source (not -1) marks source as visited
    # while still distinguishing it from an untouched node.
    parent[source] = source
    queue: deque[int] = deque([source])

    while queue:
        u = queue.popleft()
        for v in adj[u]:
            if parent[v] == -1 and capacity[u][v] > 0:
                parent[v] = u
                if v == sink:
                    return parent
                queue.append(v)

    return None
```

Count islands in a 2D grid of '1's (land) and '0's (water)

Problem

Given an m x n 2D binary grid where '1' represents land and '0' represents water, return the number of islands. An island is surrounded by water and formed by connecting adjacent lands horizontally or vertically.

Approach

BFS flood fill. Iterate every cell; when a '1' is found, increment the island counter and BFS to mark all connected land as visited.

When to Use

Connected components / flood fill — "count islands", "count regions", "label connected areas". BFS/DFS from each unvisited cell. Geospatial: land-use classification, satellite imagery segmentation.

Complexity

```
Time:  O(m * n) -- each cell visited at most once
Space:  O(m * n) -- visited set (or O(min(m, n)) BFS queue)
```

Implementation

```
from collections import deque


def num_islands(grid: list[list[str]]) -> int:
    """Return the number of islands in *grid*.

    >>> num_islands([["1", "1", "0"], ["0", "1", "0"], ["0", "0", "1"]])
    2
    """
    if not grid or not grid[0]:
        return 0

    rows, cols = len(grid), len(grid[0])
    visited: set[tuple[int, int]] = set()
    count = 0

    for r in range(rows):
        for c in range(cols):
            # Cells are the strings "1"/"0", not ints -- a common trap when
            # porting this from a problem statement that shows a numeric grid.
            if grid[r][c] == "1" and (r, c) not in visited:
                count += 1
                _bfs(grid, r, c, rows, cols, visited)

    return count


def _bfs(
    grid: list[list[str]],
    start_r: int,
    start_c: int,
    rows: int,
    cols: int,
    visited: set[tuple[int, int]],
) -> None:
    queue: deque[tuple[int, int]] = deque([(start_r, start_c)])
    visited.add((start_r, start_c))

    while queue:
        cr, cc = queue.popleft()
```

```
for dr, dc in ((0, 1), (0, -1), (1, 0), (-1, 0)):
    nr, nc = cr + dr, cc + dc
    if (
        0 <= nr < rows
        and 0 <= nc < cols
        and grid[nr][nc] == "1"
        and (nr, nc) not in visited
    ):
        # Mark visited on enqueue, not on dequeue -- otherwise the
        # same cell can be pushed multiple times before it's processed.
        visited.add((nr, nc))
        queue.append((nr, nc))
```

Topological ordering of a directed acyclic graph (DAG)

Problem

Given a DAG represented as an adjacency list, return a valid topological ordering of its vertices. If the graph has a cycle, return an empty list.

Approach

1. Kahn's algorithm (BFS): Track in-degrees; repeatedly dequeue nodes with in-degree 0 and decrement neighbors' in-degrees. 2. DFS-based: Post-order DFS; reverse the finish order.

When to Use

Dependency resolution — "build order", "task scheduling with prereqs", "compile order". Any DAG where you need a valid linear ordering. Also: package managers, makefile targets, course planning.

Complexity

```
Time:  O(V + E)
Space:  O(V + E)
```

Implementation

```
from collections import deque


def topological_sort_kahn(
    num_nodes: int,
    edges: list[tuple[int, int]],
) -> list[int]:
    """Return a topological ordering using Kahn's algorithm.

    *edges* is a list of (u, v) meaning u -> v.
    Returns [] if a cycle exists.

    >>> topological_sort_kahn(4, [(0, 1), (0, 2), (1, 3), (2, 3)])
    [0, 1, 2, 3]
    """
    adj: list[list[int]] = [[] for _ in range(num_nodes)]
    in_degree = [0] * num_nodes

    for u, v in edges:
        adj[u].append(v)
        in_degree[v] += 1

    queue: deque[int] = deque(i for i in range(num_nodes) if in_degree[i] == 0)
    order: list[int] = []

    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in adj[node]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Nodes stuck in a cycle never reach in-degree 0, so a short order here
    # is the only signal that a cycle exists.
    if len(order) != num_nodes:
        return []
    return order
```

```
def topological_sort_dfs(
    num_nodes: int,
    edges: list[tuple[int, int]],
) -> list[int]:
    """Return a topological ordering using DFS post-order reversal.

    Returns [] if a cycle exists.

    >>> topological_sort_dfs(4, [(0, 1), (0, 2), (1, 3), (2, 3)])
    [0, 2, 1, 3]
    """
    adj: list[list[int]] = [[] for _ in range(num_nodes)]
    for u, v in edges:
        adj[u].append(v)

    # 0 = unvisited, 1 = in-progress, 2 = done
    state = [0] * num_nodes
    order: list[int] = []
    has_cycle = False

    def dfs(node: int) -> None:
        nonlocal has_cycle
        if has_cycle:
            return
        state[node] = 1
        for neighbor in adj[node]:
            # A back edge to an in-progress (state 1) ancestor closes a
            # cycle; an edge to a done (state 2) node is just a shared
            # descendant and is fine.
            if state[neighbor] == 1:
                has_cycle = True
                return
            if state[neighbor] == 0:
                dfs(neighbor)
        state[node] = 2
        order.append(node)

    for i in range(num_nodes):
        if state[i] == 0:
            dfs(i)

    if has_cycle:
        return []
    # order collected nodes in finish (post-order) time; reversing that
    # gives a valid topological order -- easy to forget this last step.
    order.reverse()
    return order
```

Shortest transformation sequence from beginWord to endWord

Problem

Given two words and a dictionary, find the length of the shortest transformation sequence from beginWord to endWord, such that only one letter can be changed at a time and each intermediate word must exist in the word list.

Approach

BFS level-by-level. At each level, for every word generate all possible one-character mutations and check if they exist in the remaining word set. Also provided: ladder_length_comprehension, which swaps the index-mutation neighbor generator (_neighbors) for a slice-based list comprehension (_neighbors_comprehension) — same BFS, a different way to build the same one-edit neighbor set.

When to Use

BFS for shortest transformation sequence — "minimum edits", "fewest steps to transform X into Y", unweighted shortest path in an implicit graph. Keywords: "word ladder", "gene mutation", "lock combination".

Complexity

```
Time:  O(n * m^2) where n = |word_list|, m = word length
Space:  O(n * m)
```

Implementation

```
from collections import deque


def ladder_length(begin_word: str, end_word: str, word_list: list[str]) -> int:
    """Return the number of words in the shortest transformation sequence.

    Returns 0 if no such sequence exists. The count includes both
    begin_word and end_word.

    >>> ladder_length("hit", "cog", ["hot", "dot", "dog", "lot", "log", "cog"])
    5
    """
    word_set = set(word_list)
    if end_word not in word_set:
        return 0

    queue: deque[str] = deque([begin_word])
    visited: set[str] = {begin_word}
    level = 1

    while queue:
        # range(len(queue)) snapshots the current level's size before the
        # loop starts pushing next-level words into the same queue.
        for _ in range(len(queue)):
            word = queue.popleft()
            if word == end_word:
                return level
            for neighbor in _neighbors(word, word_set, visited):
                # Mark visited on enqueue, not on dequeue, so the same word
                # can't be queued again via a second one-edit path.
                visited.add(neighbor)
                queue.append(neighbor)
        level += 1

    return 0


def _neighbors(
```

```
    word: str,
    word_set: set[str],
    visited: set[str],
) -> list[str]:
    """Generate valid one-edit neighbors of *word*."""
    result: list[str] = []
    word_arr = list(word)
    for i in range(len(word_arr)):
        original = word_arr[i]
        for c in "abcdefghijklmnopqrstuvwxyz":
            # Skip the original letter: that "mutation" isn't an edit at all.
            if c == original:
                continue
            word_arr[i] = c
            candidate = "".join(word_arr)
            if candidate in word_set and candidate not in visited:
                result.append(candidate)
        word_arr[i] = original
    return result


# --- comprehension form: build one-edit neighbors via slicing instead of index-mutation ---
def ladder_length_comprehension(
    begin_word: str, end_word: str, word_list: list[str]
) -> int:
    """Same BFS as ladder_length, but generates neighbors via slice comprehension.

    >>> ladder_length_comprehension(
    ...     "hit", "cog", ["hot", "dot", "dog", "lot", "log", "cog"]
    ... )
    5
    """
    word_set = set(word_list)
    if end_word not in word_set:
        return 0

    queue: deque[str] = deque([begin_word])
    visited: set[str] = {begin_word}
    level = 1

    while queue:
        for _ in range(len(queue)):
            word = queue.popleft()
            if word == end_word:
                return level
            for neighbor in _neighbors_comprehension(word, word_set, visited):
                visited.add(neighbor)
                queue.append(neighbor)
        level += 1

    return 0


def _neighbors_comprehension(
    word: str,
    word_set: set[str],
    visited: set[str],
) -> list[str]:
    """Generate valid one-edit neighbors of *word* via slice comprehension."""
    return [
        candidate
        for i in range(len(word))
```

```
        for c in "abcdefghijklmnopqrstuvwxyz"
        if c != word[i]
        for candidate in (word[:i] + c + word[i + 1 :],)
        if candidate in word_set and candidate not in visited
    ]
```

Dynamic Programming

Climbing Stairs — ways to climb n stairs taking 1 or 2 steps

Problem

You are climbing a staircase with n steps. Each time you can climb 1 or 2 steps. Return the number of distinct ways to reach the top.

Approach

Fibonacci variant. The number of ways to reach step i equals the sum of ways to reach step i-1 (take 1 step) and step i-2 (take 2 steps). Use two rolling variables instead of an array (climb_stairs). An alternate top-down form (climb_stairs_memo) shows the same recurrence via memoized recursion using functools.cache.

When to Use

Counting paths / Fibonacci family — "how many ways to reach step N", "count distinct paths with step choices". Rolling-variable DP when each state depends on a fixed number of previous states.

Complexity

```
Time:  O(n)
Space:  O(1)
```

Implementation

```
import functools


def climb_stairs(n: int) -> int:
    """Return the number of distinct ways to climb *n* stairs.

    >>> climb_stairs(1)
    1
    >>> climb_stairs(5)
    8
    """
    if n < 0:
        msg = f"n must be non-negative, got {n}"
        raise ValueError(msg)
    if n <= 1:
        return 1

    prev2, prev1 = 1, 1
    for _ in range(2, n + 1):
        # tuple assignment evaluates the RHS first, so both rolling
        # variables advance from their *old* values simultaneously
        prev2, prev1 = prev1, prev1 + prev2
    return prev1


# --- top-down memoized alternate (functools.cache instead of a rolling loop) ---
@functools.cache
def climb_stairs_memo(n: int) -> int:
    """Return the number of distinct ways to climb *n* stairs (top-down).

    >>> climb_stairs_memo(1)
    1
    >>> climb_stairs_memo(5)
    8
    """
    if n < 0:
        msg = f"n must be non-negative, got {n}"
        raise ValueError(msg)
    if n <= 1:
```

```
    return 1
return climb_stairs_memo(n - 1) + climb_stairs_memo(n - 2)
```

Coin Change — minimum coins to make amount

Problem

Given an array of coin denominations and a target amount, return the fewest number of coins needed to make that amount. Return -1 if it cannot be made.

Approach

Bottom-up DP. dp[a] holds the minimum coins for amount a. For each amount from 1..target, try every coin and take the min.

When to Use

Classic "minimum cost to reach target" DP. Any problem where you choose from a set of options to reach a goal with minimum steps/cost. Variations: unbounded knapsack, minimum operations.

Complexity

```
Time:  O(amount * len(coins))
Space:  O(amount)
```

Implementation

```
from collections.abc import Sequence


def coin_change(coins: Sequence[int], amount: int) -> int:
    """Return the minimum number of coins to make *amount*, or -1.

    >>> coin_change([1, 5, 10, 25], 30)
    2
    >>> coin_change([2], 3)
    -1
    """
    if amount < 0:
        return -1
    if amount == 0:
        return 0

    # amount + 1 coins is impossible with any real denomination, so it is a
    # safe "unreachable" sentinel that still compares correctly with min()
    dp = [amount + 1] * (amount + 1)
    dp[0] = 0

    for a in range(1, amount + 1):
        for c in coins:
            if c <= a:
                dp[a] = min(dp[a], dp[a - c] + 1)

    # sentinel never got overwritten -> amount is unreachable
    return dp[amount] if dp[amount] <= amount else -1
```

Constraint Satisfaction Problem — generic CSP solver with Sudoku example

Problem

Solve constraint satisfaction problems using backtracking with constraint propagation (arc consistency / forward checking).

Approach

Generic CSP framework: variables have domains, constraints link variable pairs. Backtrack with MRV (Minimum Remaining Values) heuristic and forward checking to prune domains early. Includes a Sudoku solver built on the generic framework.

When to Use

Puzzle solving, scheduling, configuration — "Sudoku", "N-Queens via CSP", "timetable generation", "resource assignment with constraints". Backtracking + forward checking prunes aggressively in practice.

Complexity

```
Time:  Exponential worst case, but constraint propagation prunes
Space:  O(n * d) where n = variables, d = max domain size.
```

Implementation

```
from collections.abc import Callable, Mapping, Sequence

type Constraint[T] = Callable[[T, T], bool]


class CSP[T]:
    """Generic CSP solver using backtracking with forward checking."""

    def __init__(
        self,
        variables: Sequence[str],
        domains: dict[str, list[T]],
        neighbors: Mapping[str, Sequence[str]],
        constraint: Constraint[T],
    ) -> None:
        self.variables = list(variables)
        self.domains = {v: list(d) for v, d in domains.items()}
        self.neighbors = neighbors
        self.constraint = constraint

    def solve(self) -> dict[str, T] | None:
        """Return an assignment satisfying all constraints, or None."""
        assignment: dict[str, T] = {}
        return self._backtrack(assignment)

    def _select_unassigned(self, assignment: dict[str, T]) -> str:
        """MRV heuristic: pick variable with fewest remaining values."""
        unassigned = [v for v in self.variables if v not in assignment]
        return min(unassigned, key=lambda v: len(self.domains[v]))

    def _is_consistent(self, var: str, val: T, assignment: dict[str, T]) -> bool:
        for neighbor in self.neighbors.get(var, []):
            if neighbor in assignment and not self.constraint(
                val, assignment[neighbor]
            ):
                return False
        return True

    def _forward_check(
        self, var: str, val: T, assignment: dict[str, T]
```

```
    ) -> dict[str, list[T]] | None:
        """Prune neighbor domains. Return removed values or None on wipeout."""
        removed: dict[str, list[T]] = {}
        for neighbor in self.neighbors.get(var, []):
            if neighbor in assignment:
                continue
            to_remove: list[T] = []
            for nval in self.domains[neighbor]:
                if not self.constraint(nval, val):
                    to_remove.append(nval)
            if to_remove:
                removed[neighbor] = to_remove
                for rv in to_remove:
                    self.domains[neighbor].remove(rv)
                if not self.domains[neighbor]:
                    # Domain wipeout -- restore and fail
                    for nb, vals in removed.items():
                        self.domains[nb].extend(vals)
                    return None
        return removed

    def _backtrack(self, assignment: dict[str, T]) -> dict[str, T] | None:
        if len(assignment) == len(self.variables):
            return dict(assignment)

        var = self._select_unassigned(assignment)

        for val in list(self.domains[var]):
            if not self._is_consistent(var, val, assignment):
                continue

            assignment[var] = val
            removed = self._forward_check(var, val, assignment)

            if removed is not None:
                result = self._backtrack(assignment)
                if result is not None:
                    return result
                # Restore pruned domains
                for nb, vals in removed.items():
                    self.domains[nb].extend(vals)

            del assignment[var]

        return None


# ---------------------------------------------------------------------------
# Sudoku solver built on the CSP framework
# ---------------------------------------------------------------------------

type Grid = list[list[int]]

_CELLS = [f"R{r}C{c}" for r in range(9) for c in range(9)]


def _sudoku_neighbors() -> dict[str, list[str]]:
    """Build neighbor map: cells sharing a row, column, or 3x3 box."""
    neighbors: dict[str, set[str]] = {cell: set() for cell in _CELLS}
    for r in range(9):
        for c in range(9):
            cell = f"R{r}C{c}"
```

```
            for k in range(9):
                if k != c:
                    neighbors[cell].add(f"R{r}C{k}")
                if k != r:
                    neighbors[cell].add(f"R{k}C{c}")
            br, bc = 3 * (r // 3), 3 * (c // 3)
            for dr in range(3):
                for dc in range(3):
                    other = f"R{br + dr}C{bc + dc}"
                    if other != cell:
                        neighbors[cell].add(other)
    return {k: sorted(v) for k, v in neighbors.items()}


_NEIGHBORS = _sudoku_neighbors()


def solve_sudoku(board: Grid) -> Grid | None:
    """Solve a 9x9 Sudoku board in-place and return it, or None if unsolvable.

    *board* is a 9x9 list of ints where 0 represents an empty cell.

    >>> board = [
    ...     [5, 3, 0, 0, 7, 0, 0, 0, 0],
    ...     [6, 0, 0, 1, 9, 5, 0, 0, 0],
    ...     [0, 9, 8, 0, 0, 0, 0, 6, 0],
    ...     [8, 0, 0, 0, 6, 0, 0, 0, 3],
    ...     [4, 0, 0, 8, 0, 3, 0, 0, 1],
    ...     [7, 0, 0, 0, 2, 0, 0, 0, 6],
    ...     [0, 6, 0, 0, 0, 0, 2, 8, 0],
    ...     [0, 0, 0, 4, 1, 9, 0, 0, 5],
    ...     [0, 0, 0, 0, 8, 0, 0, 7, 9],
    ... ]
    >>> result = solve_sudoku(board)
    >>> result is not None
    True
    """
    domains: dict[str, list[int]] = {}
    variables: list[str] = []

    for r in range(9):
        for c in range(9):
            cell = f"R{r}C{c}"
            if board[r][c] != 0:
                domains[cell] = [board[r][c]]
            else:
                domains[cell] = list(range(1, 10))
            variables.append(cell)

    def not_equal(a: int, b: int) -> bool:
        return a != b

    csp: CSP[int] = CSP(variables, domains, _NEIGHBORS, not_equal)
    solution = csp.solve()

    if solution is None:
        return None

    for r in range(9):
        for c in range(9):
            board[r][c] = solution[f"R{r}C{c}"]
    return board
```

Edit Distance — minimum operations to convert word1 to word2

Problem

Given two strings, return the minimum number of single-character operations (insert, delete, replace) to convert word1 into word2.

Approach

2D DP where “dp[i][j]“ is the edit distance between the first i characters of word1 and the first j characters of word2. edit_distance space-optimizes this to a single rolling row since each cell only depends on the current and previous row; the edit_distance_2d alternate keeps the full table for clarity.

When to Use

String similarity / diff algorithms — "minimum edits", "Levenshtein distance", spell checking, DNA sequence alignment. Foundation for fuzzy matching and diff tools. See also: longest_common_subseq.

Complexity

```
Time:  O(m * n)
Space:  O(n) (space-optimized)
```

Implementation

```
def edit_distance(word1: str, word2: str) -> int:
    """Return the minimum edit distance between *word1* and *word2*.

    >>> edit_distance("horse", "ros")
    3
    >>> edit_distance("intention", "execution")
    5
    """
    m, n = len(word1), len(word2)

    # dp[j] represents the distance for word2[:j]
    dp = list(range(n + 1))

    for i in range(1, m + 1):
        prev = dp[0]
        dp[0] = i
        for j in range(1, n + 1):
            # save the pre-overwrite value: it's dp[i-1][j-1] (diagonal) for
            # the *next* j, since dp[j] is about to become dp[i][j]
            temp = dp[j]
            if word1[i - 1] == word2[j - 1]:
                dp[j] = prev
            else:
                dp[j] = 1 + min(prev, dp[j], dp[j - 1])
            prev = temp

    return dp[n]


# --- full 2D table alternate (explicit rows, no rolling overwrite) ---
def edit_distance_2d(word1: str, word2: str) -> int:
    """Return the minimum edit distance using the full 2D DP table.

    >>> edit_distance_2d("horse", "ros")
    3
    >>> edit_distance_2d("intention", "execution")
    5
    """
    m, n = len(word1), len(word2)
```

```
    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 word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = 1 + min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]
```

0/1 Knapsack — maximize value within weight capacity

Problem

Given n items, each with a weight and value, and a knapsack with a weight capacity W, choose items to maximize total value without exceeding capacity. Each item can be taken at most once.

Approach

Classic DP. knapsack_2d is the full tabulation (for clarity); knapsack is the space-optimized 1D alternate (iterate capacity backwards to avoid reusing items in the same row).

When to Use

Resource allocation with constraints — "maximize value under weight limit", "subset sum", "budget allocation". 0/1 variant for items used at most once.

Complexity

```
Time:  O(n * W)
Space:  O(n * W) for 2D, O(W) for 1D
```

Implementation

```
from collections.abc import Sequence


def knapsack_2d(
    weights: Sequence[int],
    values: Sequence[int],
    capacity: int,
) -> int:
    """Return the maximum value achievable using 2D DP table.

    >>> knapsack_2d([1, 3, 4, 5], [1, 4, 5, 7], 7)
    9
    """
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        w, v = weights[i - 1], values[i - 1]
        for c in range(capacity + 1):
            dp[i][c] = dp[i - 1][c]
            if w <= c:
                dp[i][c] = max(dp[i][c], dp[i - 1][c - w] + v)

    return dp[n][capacity]


# --- space-optimized 1D alternate (rolling array over a 2D table) ---
def knapsack(
    weights: Sequence[int],
    values: Sequence[int],
    capacity: int,
) -> int:
    """Return the maximum value achievable using 1D space optimization.

    >>> knapsack([1, 3, 4, 5], [1, 4, 5, 7], 7)
    9
    """
    dp = [0] * (capacity + 1)

    for w, v in zip(weights, values, strict=True):
```

```
        # capacity walked high-to-low so dp[c - w] still holds last row's
        # (item i-1) value -- forward iteration would let one item get
        # counted twice in the same pass (unbounded, not 0/1)
        for c in range(capacity, w - 1, -1):
            dp[c] = max(dp[c], dp[c - w] + v)

    return dp[capacity]
```

Longest Common Subsequence — length of LCS of two strings

Problem

Given two strings, return the length of their longest common subsequence. A subsequence is a sequence derived by deleting some (or no) characters without changing the relative order.

Approach

2D DP. If characters match, extend the diagonal; otherwise take the max of skipping one character from either string. longest_common_subsequence space-optimizes this to a single row; the longest_common_subsequence_2d alternate keeps the full table.

When to Use

Diff / alignment — "longest common subsequence", "diff two files", DNA sequence alignment. Foundation for unified-diff algorithms. See also: edit_distance for minimum-cost transformation.

Complexity

```
Time:  O(m * n)
Space:  O(min(m, n))
```

Implementation

```
def longest_common_subsequence(text1: str, text2: str) -> int:
    """Return the length of the LCS of *text1* and *text2*.

    >>> longest_common_subsequence("abcde", "ace")
    3
    >>> longest_common_subsequence("abc", "def")
    0
    """
    # Ensure text2 is the shorter string for O(min(m,n)) space
    if len(text1) < len(text2):
        text1, text2 = text2, text1

    m, n = len(text1), len(text2)
    dp = [0] * (n + 1)

    for i in range(1, m + 1):
        prev = 0
        for j in range(1, n + 1):
            # save before overwrite: becomes the diagonal (dp[i-1][j-1])
            # value for the next iteration of j
            temp = dp[j]
            if text1[i - 1] == text2[j - 1]:
                dp[j] = prev + 1
            else:
                dp[j] = max(dp[j], dp[j - 1])
            prev = temp

    return dp[n]


# --- full 2D table alternate (explicit rows, no rolling overwrite) ---
def longest_common_subsequence_2d(text1: str, text2: str) -> int:
    """Return the length of the LCS using the full 2D DP table.

    >>> longest_common_subsequence_2d("abcde", "ace")
    3
    >>> longest_common_subsequence_2d("abc", "def")
    0
    """
```

```
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i - 1] == text2[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]
```

Longest Increasing Subsequence — length of LIS

Problem

Given an integer array, return the length of the longest strictly increasing subsequence.

Approach

length_of_lis: patience sorting with binary search. Maintain a list of "tails" where tails[i] is the smallest tail element for an increasing subsequence of length i+1. For each number, use bisect_left to find its position and either extend or replace. length_of_lis_dp is the classic O(n^2) DP alternate: dp[i] is the LIS length ending at i, built by scanning all earlier elements.

When to Use

Patience sorting / longest chain — "longest increasing subsequence", "longest chain of pairs", "box stacking". Binary search on tails array for O(n log n). Also: longest non-decreasing, envelope nesting.

Complexity

```
Time:  O(n log n)
Space:  O(n)
```

Implementation

```
import bisect
from collections.abc import Sequence


def length_of_lis(nums: Sequence[int]) -> int:
    """Return the length of the longest strictly increasing subsequence.

    >>> length_of_lis([10, 9, 2, 5, 3, 7, 101, 18])
    4
    >>> length_of_lis([0, 1, 0, 3, 2, 3])
    4
    """
    if not nums:
        return 0

    tails: list[int] = []
    for n in nums:
        # bisect_left (not bisect_right) enforces *strictly* increasing:
        # an equal value replaces rather than extends a run
        pos = bisect.bisect_left(tails, n)
        if pos == len(tails):
            tails.append(n)
        else:
            tails[pos] = n
    return len(tails)


# --- O(n^2) DP table alternate (explicit pairwise comparisons) ---
def length_of_lis_dp(nums: Sequence[int]) -> int:
    """Return the length of the longest strictly increasing subsequence.

    >>> length_of_lis_dp([10, 9, 2, 5, 3, 7, 101, 18])
    4
    >>> length_of_lis_dp([0, 1, 0, 3, 2, 3])
    4
    """
    if not nums:
        return 0
```

```
    dp = [1] * len(nums)
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

Traveling Salesman Problem — bitmask DP solution

Problem

Given a weighted adjacency matrix of n cities, find the minimum cost of visiting every city exactly once and returning to the starting city.

Approach

tsp: bitmask DP (Held-Karp algorithm). State is (visited_set, current_city). Use an integer bitmask to represent the set of visited cities. “dp[mask][i]“ = minimum cost to visit the cities in ‘mask‘, ending at city i. tsp_brute_force is the exponential itertools.permutations alternate used to sanity-check small cases.

When to Use

Visit-all-nodes optimization — "shortest route visiting every city", delivery/pickup routing, inspection tours. Bitmask DP (Held-Karp) for exact solution when n <= 20.

Complexity

```
Time:  O(n^2 * 2^n)
Space:  O(n * 2^n)
```

Implementation

```
import itertools
from collections.abc import Sequence

INF = float("inf")


def tsp(dist: Sequence[Sequence[int | float]]) -> int | float:
    """Return the minimum cost tour visiting all cities and returning to start.

    *dist* is an n x n adjacency matrix where ‘‘dist[i][j]‘‘ is the cost
    from city i to city j.

    >>> tsp([[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]])
    80
    """
    n = len(dist)
    if n <= 1:
        return 0

    full_mask = (1 << n) - 1

    # dp[mask][i] = min cost to reach city i having visited cities in mask
    dp: list[list[int | float]] = [[INF] * n for _ in range(1 << n)]
    dp[1][0] = 0  # start at city 0

    for mask in range(1 << n):
        for u in range(n):
            # state unreachable (or u not actually in mask) -- nothing to extend
            if dp[mask][u] == INF:
                continue
            if not (mask & (1 << u)):
                continue
            for v in range(n):
                if mask & (1 << v):
                    continue
                new_mask = mask | (1 << v)
                cost = dp[mask][u] + dist[u][v]
                if cost < dp[new_mask][v]:
                    dp[new_mask][v] = cost
```

```
    # Close the tour: return to city 0. Start at u=1 -- city 0 is the tour's
    # start, not a valid "last stop" to close the loop from.
    result: int | float = INF
    for u in range(1, n):
        cost = dp[full_mask][u] + dist[u][0]
        if cost < result:
            result = cost

    return result


# --- brute-force alternate (itertools.permutations, exponential) ---
def tsp_brute_force(dist: Sequence[Sequence[int | float]]) -> int | float:
    """Return the minimum cost tour by trying every permutation of cities.

    Exponential (O(n!)); only useful as a correctness oracle for small n.

    >>> tsp_brute_force(
    ...     [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
    ... )
    80
    """
    n = len(dist)
    if n <= 1:
        return 0

    best: int | float = INF
    for perm in itertools.permutations(range(1, n)):
        route = (0, *perm, 0)
        cost = sum(dist[route[i]][route[i + 1]] for i in range(len(route) - 1))
        best = min(best, cost)
    return best
```

Heaps

Find the kth largest element

Problem

Given an unsorted array and an integer k, return the kth largest element. Also implement a streaming version that supports adding new values.

Approach

Maintain a min-heap of size k. The heap top is always the kth largest. For each new element larger than the heap top, replace it. Alternate find_kth_largest_nlargest gets the same answer with heapq.nlargest.

When to Use

Streaming top-K — "kth largest in a stream", "maintain running top K". Min-heap of size k keeps only the K largest seen so far. Also: real-time leaderboard, top-K sensor readings in telemetry.

Implementation

```
import heapq
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from collections.abc import Sequence


def find_kth_largest(nums: Sequence[int], k: int) -> int:
    """Return the kth largest element in the array.

    >>> find_kth_largest([3, 2, 1, 5, 6, 4], 2)
    5
    """
    if k <= 0 or k > len(nums):
        msg = f"k={k} out of range for length {len(nums)}"
        raise ValueError(msg)

    heap = list(nums[:k])
    heapq.heapify(heap)

    for n in nums[k:]:
        if n > heap[0]:
            heapq.heapreplace(heap, n)  # pop-then-push in one step; heap stays size k

    return heap[0]


# --- heapq.nlargest one-liner (O(n log k), stdlib) ---
def find_kth_largest_nlargest(nums: Sequence[int], k: int) -> int:
    """Return the kth largest element using heapq.nlargest (stdlib).

    >>> find_kth_largest_nlargest([3, 2, 1, 5, 6, 4], 2)
    5
    """
    if k <= 0 or k > len(nums):
        msg = f"k={k} out of range for length {len(nums)}"
        raise ValueError(msg)

    return heapq.nlargest(k, nums)[-1]


class KthLargest:
    """Stream version: maintain the kth largest over a growing dataset.

    >>> kl = KthLargest(3, [4, 5, 8, 2])
```

```
    >>> kl.add(3)
    4
    >>> kl.add(5)
    5
    >>> kl.add(10)
    5
    """

    def __init__(self, k: int, nums: Sequence[int]) -> None:
        self._k = k
        self._heap: list[int] = []
        for n in nums:
            self.add(n)

    def add(self, val: int) -> int:
        """Add a value and return the current kth largest element."""
        if len(self._heap) < self._k:
            heapq.heappush(self._heap, val)  # still growing to size k: plain push
        elif val > self._heap[0]:
            heapq.heapreplace(self._heap, val)  # at size k: pop-then-push, not push-then-pop
        return self._heap[0]
```

Merge k sorted linked lists

Problem

Given an array of k sorted linked lists, merge them into one sorted linked list.

Approach

Use a min-heap of size k. Push (value, list_index, node) tuples. Pop the smallest, append to result, and push the next node from that list. The list_index breaks ties to avoid comparing nodes. Alternate merge_k_sorted_sorted collects every value and sorts once.

When to Use

K-way merge for external sorting — "merge K sorted lists/arrays/files", combining sorted partitions from distributed systems. Min-heap of size k selects the next smallest element. See also: merge_two_sorted.

Complexity

```
Time:  O(n log k) where n = total nodes across all lists
Space:  O(k) for the heap
```

Implementation

```
import heapq
from dataclasses import dataclass


@dataclass
class ListNode:
    val: int
    next: ListNode | None = None


def merge_k_sorted(lists: list[ListNode | None]) -> ListNode | None:
    """Merge k sorted linked lists into one sorted linked list.

    >>> to_list(
    ...     merge_k_sorted(
    ...         [from_list([1, 4, 5]), from_list([1, 3, 4]), from_list([2, 6])]
    ...     )
    ... )
    [1, 1, 2, 3, 4, 4, 5, 6]
    """
    heap: list[tuple[int, int, ListNode]] = []

    for i, node in enumerate(lists):
        if node:
            heap.append((node.val, i, node))  # index tiebreak: never compares ListNodes

    heapq.heapify(heap)

    dummy = ListNode(0)
    tail = dummy

    while heap:
        _val, idx, node = heapq.heappop(heap)
        tail.next = node
        tail = tail.next
        if node.next:
            heapq.heappush(heap, (node.next.val, idx, node.next))

    return dummy.next
```

```
# --- stdlib alternate: collect all values and sort once (O(n log n), ignores k) ---
def merge_k_sorted_sorted(lists: list[ListNode | None]) -> ListNode | None:
    """Merge k sorted linked lists by collecting values and sorting.

    >>> to_list(
    ...     merge_k_sorted_sorted(
    ...         [from_list([1, 4, 5]), from_list([1, 3, 4]), from_list([2, 6])]
    ...     )
    ... )
    [1, 1, 2, 3, 4, 4, 5, 6]
    """
    vals: list[int] = []
    for node in lists:
        while node:
            vals.append(node.val)
            node = node.next
    vals.sort()

    dummy = ListNode(0)
    tail = dummy
    for v in vals:
        tail.next = ListNode(v)
        tail = tail.next

    return dummy.next


# --- helpers for testing ---
def from_list(vals: list[int]) -> ListNode | None:
    """Build a linked list from a Python list."""
    dummy = ListNode(0)
    curr = dummy
    for v in vals:
        curr.next = ListNode(v)
        curr = curr.next
    return dummy.next


def to_list(head: ListNode | None) -> list[int]:
    """Collect linked list values into a Python list."""
    result: list[int] = []
    while head:
        result.append(head.val)
        head = head.next
    return result
```

Task scheduler with cooldown

Problem

Given a list of tasks (characters) and a cooldown period n, find the minimum number of intervals needed to execute all tasks. The same task must wait at least n intervals before being executed again.

Approach

Use a max-heap (negated counts) to always schedule the most frequent task first. After executing a task, place it in a cooldown queue with the time it becomes available. When that time arrives, push it back onto the heap. Alternate least_interval_formula skips the simulation entirely and computes the answer from the idle-slot math.

When to Use

Scheduling with cooldown constraints — "minimum time to complete all tasks with cooldown", "CPU scheduling", "rate-limited job execution". Max-heap ensures the most frequent task is scheduled first.

Complexity

```
Time:  O(t) where t = total intervals (bounded by len(tasks) * (n+1))
Space:  O(1) -- at most 26 distinct task types
```

Implementation

```
import heapq
from collections import Counter, deque
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from collections.abc import Sequence


def least_interval(tasks: Sequence[str], n: int) -> int:
    """Return the minimum number of intervals to complete all tasks.

    >>> least_interval(["A", "A", "A", "B", "B", "B"], 2)
    8
    """
    if not tasks:
        return 0

    counts = list(Counter(tasks).values())
    max_heap = [-c for c in counts]
    heapq.heapify(max_heap)

    time = 0
    cooldown: deque[tuple[int, int]] = deque()  # (available_time, neg_count)

    while max_heap or cooldown:
        time += 1

        if max_heap:
            cnt = heapq.heappop(max_heap) + 1  # execute one (cnt is negative)
            if cnt:  # task still has instances left; requeue after cooldown
                cooldown.append((time + n, cnt))

        if cooldown and cooldown[0][0] == time:
            heapq.heappush(max_heap, cooldown.popleft()[1])

    return time


# --- formula alternate: idle-slot math instead of simulating the heap ---
```

```
def least_interval_formula(tasks: Sequence[str], n: int) -> int:
    """Return the minimum intervals using the idle-slot formula, no simulation.

    >>> least_interval_formula(["A", "A", "A", "B", "B", "B"], 2)
    8
    """
    if not tasks:
        return 0

    counts = Counter(tasks).values()
    max_count = max(counts)
    # slots after the most frequent task's last-but-one run, plus ties for the max
    num_max = sum(1 for c in counts if c == max_count)
    return max(len(tasks), (max_count - 1) * (n + 1) + num_max)
```

Backtracking

Combination Sum — find combinations that sum to target

Problem

Given an array of distinct positive integers (candidates) and a target integer, return all unique combinations where the chosen numbers sum to target. The same number may be used unlimited times.

Approach

Sort candidates, then backtrack. Start from the current index (not 0) to avoid duplicate combinations. Prune when the candidate exceeds the remaining target.

When to Use

Partition into target — "all combinations summing to T", "coin change enumerate", "ways to split a budget". Unlimited reuse variant; start from current index to avoid duplicate combos. See also: dp/coin_change.

Complexity

```
Time:  O(2^target) (bounded by target / min(candidates))
Space:  O(target / min(candidates)) (recursion depth)
```

Implementation

```
from collections.abc import Sequence


def combination_sum(candidates: Sequence[int], target: int) -> list[list[int]]:
    """Return all combinations of *candidates* that sum to *target*.

    >>> combination_sum([2, 3, 6, 7], 7)
    [[2, 2, 3], [7]]
    """
    result: list[list[int]] = []
    sorted_cands = sorted(candidates)

    def backtrack(start: int, path: list[int], remaining: int) -> None:
        if remaining == 0:
            result.append(path[:])
            return
        for i in range(start, len(sorted_cands)):
            c = sorted_cands[i]
            # sorted ascending -- once one candidate is too big, all later
            # (larger) ones are too, so it's safe to break, not continue
            if c > remaining:
                break
            path.append(c)
            # pass i (not i + 1): the same candidate can be reused
            backtrack(i, path, remaining - c)
            path.pop()

    backtrack(0, [], target)
    return result
```

N-Queens — place n queens on n x n board with no attacks

Problem

Place n queens on an n x n chessboard such that no two queens threaten each other (no shared row, column, or diagonal). Return all distinct solutions.

Approach

Backtracking row by row. Track occupied columns, positive diagonals (r + c), and negative diagonals (r - c) with sets. Only valid placements are explored.

When to Use

Constraint placement — "place N items with mutual exclusion constraints", "non-attacking queens", "conflict-free assignment". Row-by-row backtracking with column/diagonal conflict sets. See also: CSP solver.

Complexity

```
Time:  O(n!)  (with pruning)
Space:  O(n)
```

Implementation

```
def solve_n_queens(n: int) -> list[list[str]]:
    """Return all solutions as lists of strings ('.' and 'Q').

    >>> len(solve_n_queens(4))
    2
    >>> solve_n_queens(1)
    [['Q']]
    """
    result: list[list[str]] = []
    cols: set[int] = set()
    pos_diag: set[int] = set()  # r + c constant on / diagonals
    neg_diag: set[int] = set()  # r - c constant on \\ diagonals
    queens: list[int] = []  # queens[row] = col

    def backtrack(r: int) -> None:
        if r == n:
            board = ["." * c + "Q" + "." * (n - c - 1) for c in queens]
            result.append(board)
            return
        for c in range(n):
            if c in cols or (r + c) in pos_diag or (r - c) in neg_diag:
                continue
            cols.add(c)
            pos_diag.add(r + c)
            neg_diag.add(r - c)
            queens.append(c)
            backtrack(r + 1)
            queens.pop()
            cols.remove(c)
            pos_diag.remove(r + c)
            neg_diag.remove(r - c)

    backtrack(0)
    return result
```

Permutations — generate all permutations of a list

Problem

Given an array of distinct integers, return all possible permutations in any order.

Approach

permutations: backtracking with a visited set. At each position, try every unused element, mark it used, recurse, then unmark. permutations_itertools is the stdlib alternate using itertools.permutations.

When to Use

All orderings — "generate all permutations", "all arrangements", brute-force over orderings for small n. Building block for next-permutation and ranking/unranking algorithms.

Complexity

```
Time:  O(n * n!)
Space:  O(n) (recursion depth + visited set)
```

Implementation

```
import itertools
from collections.abc import Sequence


def permutations(nums: Sequence[int]) -> list[list[int]]:
    """Return all permutations of *nums*.

    >>> sorted(permutations([1, 2, 3]))
    [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    """
    result: list[list[int]] = []
    used = [False] * len(nums)

    def backtrack(path: list[int]) -> None:
        if len(path) == len(nums):
            # snapshot copy -- path keeps mutating after this append,
            # so appending path itself would alias every stored result
            result.append(path[:])
            return
        for i in range(len(nums)):
            if used[i]:
                continue
            used[i] = True
            path.append(nums[i])
            backtrack(path)
            path.pop()
            used[i] = False

    backtrack([])
    return result


# --- stdlib alternate: itertools.permutations ---
def permutations_itertools(nums: Sequence[int]) -> list[list[int]]:
    """Return all permutations of *nums* using itertools.permutations.

    >>> sorted(permutations_itertools([1, 2, 3]))
    [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    """
    return [list(p) for p in itertools.permutations(nums)]
```

Subsets — generate all subsets of a set

Problem

Given an integer array of unique elements, return all possible subsets (the power set). The solution must not contain duplicate subsets.

Approach

subsets: backtracking. At each index, decide to include or exclude the element. Append a snapshot of the current path at every node. subsets_bitmask is the iterative alternate: every integer from 0 to 2^n - 1 encodes one inclusion/exclusion choice per bit.

When to Use

Power set / feature combinations — "generate all subsets", "all combinations of features", "enumerate configurations". Include/exclude decision at each element. Also: feature selection, test coverage sets.

Complexity

```
Time:  O(n * 2^n)
Space:  O(n) (excluding output; recursion depth is n)
```

Implementation

```
from collections.abc import Sequence


def subsets(nums: Sequence[int]) -> list[list[int]]:
    """Return all subsets of *nums*.

    >>> sorted(subsets([1, 2, 3]), key=len)
    [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
    """
    result: list[list[int]] = []

    def backtrack(start: int, path: list[int]) -> None:
        # snapshot copy -- path is reused and mutated across the whole walk
        result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            # i + 1 (not i): each element is used at most once per subset,
            # unlike combination_sum's unlimited-reuse variant
            backtrack(i + 1, path)
            path.pop()

    backtrack(0, [])
    return result


# --- iterative alternate: bitmask enumeration over 2^n choices ---
def subsets_bitmask(nums: Sequence[int]) -> list[list[int]]:
    """Return all subsets of *nums* by enumerating bitmasks.

    >>> sorted(subsets_bitmask([1, 2, 3]), key=len)
    [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
    """
    n = len(nums)
    return [[nums[i] for i in range(n) if mask & (1 << i)] for mask in range(1 << n)]
```

Greedy

Interval Scheduling — maximum non-overlapping intervals

Problem

Given a collection of intervals, find the maximum number of non-overlapping intervals (activity selection problem).

Approach

Sort intervals by end time. Greedily select the next interval whose start time is >= the end time of the last selected interval. This maximizes the number of non-overlapping intervals.

When to Use

Activity selection / resource booking — "max non-overlapping intervals", "minimum removals for no overlap", "room scheduling". Sort by end time, greedily pick earliest-finishing.

Complexity

```
Time:  O(n log n)
Space:  O(1) (excluding input sort)
```

Implementation

```
from collections.abc import Sequence


def max_non_overlapping(intervals: Sequence[Sequence[int]]) -> int:
    """Return the maximum number of non-overlapping intervals.

    >>> max_non_overlapping([[1, 3], [2, 4], [3, 5], [0, 6]])
    2
    """
    if not intervals:
        return 0

    # sort by END time, not start: the interval finishing soonest always
    # leaves the most room for everything after it
    sorted_iv = sorted(intervals, key=lambda iv: iv[1])
    count = 1
    end = sorted_iv[0][1]

    for start, finish in sorted_iv[1:]:
        # >= (not >): touching intervals (start == end) don't overlap
        if start >= end:
            count += 1
            end = finish

    return count


def min_removals(intervals: Sequence[Sequence[int]]) -> int:
    """Return the minimum number of intervals to remove for no overlap.

    Equivalent to len(intervals) - max_non_overlapping(intervals).

    >>> min_removals([[1, 2], [2, 3], [3, 4], [1, 3]])
    1
    """
    return len(intervals) - max_non_overlapping(intervals)
```

Jump Game — can you reach the last index? / minimum jumps

Approach

I: Track the farthest reachable index. If current index exceeds farthest, return False. II: BFS-style greedy. Track current window end and farthest reachable. Increment jumps when reaching the window end.

When to Use

Reachability analysis — "can you reach the end?", "minimum hops to reach target". Track farthest reachable index greedily. Also: network hop-count analysis, coverage verification.

Complexity

```
Time:  O(n)
Space:  O(1)
```

Implementation

```
from collections.abc import Sequence


def can_jump(nums: Sequence[int]) -> bool:
    """Return True if the last index is reachable.

    >>> can_jump([2, 3, 1, 1, 4])
    True
    >>> can_jump([3, 2, 1, 0, 4])
    False
    """
    farthest = 0
    for i, n in enumerate(nums):
        # check BEFORE extending farthest: if this index is already beyond
        # every prior jump's reach, it was never reachable to begin with
        if i > farthest:
            return False
        farthest = max(farthest, i + n)
    return True


def jump_game_ii(nums: Sequence[int]) -> int:
    """Return the minimum number of jumps to reach the last index.

    >>> jump_game_ii([2, 3, 1, 1, 4])
    2
    >>> jump_game_ii([1])
    0
    """
    if len(nums) <= 1:
        return 0

    jumps = 0
    cur_end = 0
    farthest = 0

    for i in range(len(nums) - 1):
        farthest = max(farthest, i + nums[i])
        # i reached the end of the current BFS "layer" -- must jump again
        # to advance to the next layer's reachable window
        if i == cur_end:
            jumps += 1
            cur_end = farthest
            if cur_end >= len(nums) - 1:
                break
```

```
    return jumps
```

Merge Intervals — merge overlapping intervals

Problem

Given an array of intervals where intervals[i] = [start_i, end_i], merge all overlapping intervals and return the non-overlapping result covering the same ranges.

Approach

Sort intervals by start time. Iterate and merge: if the current interval overlaps with the last merged one, extend the end; otherwise append a new interval.

When to Use

Overlapping range consolidation — "merge intervals", "insert interval", "meeting room conflicts". Sort by start, sweep and merge.

Complexity

```
Time:  O(n log n)
Space:  O(n) (for the output)
```

Implementation

```
from collections.abc import Sequence


def merge_intervals(intervals: Sequence[Sequence[int]]) -> list[list[int]]:
    """Return merged non-overlapping intervals.

    >>> merge_intervals([[1, 3], [2, 6], [8, 10], [15, 18]])
    [[1, 6], [8, 10], [15, 18]]
    """
    if not intervals:
        return []

    sorted_iv = sorted(intervals, key=lambda iv: iv[0])
    merged: list[list[int]] = [list(sorted_iv[0])]

    for start, end in sorted_iv[1:]:
        # <= (not <): touching intervals (start == last end) DO merge here,
        # opposite convention from interval_scheduling's non-overlap check
        if start <= merged[-1][1]:
            merged[-1][1] = max(merged[-1][1], end)
        else:
            merged.append([start, end])

    return merged
```

Strings

Longest Common Prefix — find the longest common prefix among strings

Problem

Given a list of strings, return the longest common prefix shared by all of them.

Approach

Vertical scanning: compare character-by-character across all strings. Use the first string as reference; stop when any string differs or is exhausted. Alternate longest_common_prefix_zip does the same columnar scan with zip() and itertools.takewhile.

When to Use

Autocomplete, trie-based search, file path commonality. Simple but frequently asked.

Complexity

```
Time:  O(S) where S = sum of all string lengths
Space:  O(1) -- only stores the prefix end index
```

Implementation

```
from collections.abc import Sequence
from itertools import takewhile


def longest_common_prefix(strs: Sequence[str]) -> str:
    """Return the longest common prefix of all strings in *strs*.

    >>> longest_common_prefix(["flower", "flow", "flight"])
    'fl'
    >>> longest_common_prefix(["dog", "racecar", "car"])
    ''
    """
    if not strs:
        return ""

    for i, ch in enumerate(strs[0]):
        for s in strs[1:]:
            if (
                i >= len(s) or s[i] != ch
            ):  # guard: reference string outran a shorter one
                return strs[0][:i]

    return strs[0]


# --- zip/takewhile columnar variant (stdlib idiom) ---
def longest_common_prefix_zip(strs: Sequence[str]) -> str:
    """Return the longest common prefix of all strings in *strs*, via zip().

    >>> longest_common_prefix_zip(["flower", "flow", "flight"])
    'fl'
    >>> longest_common_prefix_zip(["dog", "racecar", "car"])
    ''
    """
    if not strs:
        return ""

    # zip(*strs) truncates to the shortest string automatically -- no explicit
    # length guard needed, unlike the vertical-scan version above
    columns = takewhile(lambda chars: len(set(chars)) == 1, zip(*strs, strict=False))
    return "".join(chars[0] for chars in columns)
```

Longest Palindromic Substring — find the longest palindromic substring

Problem

Given a string, return the longest substring that is a palindrome.

Approach

Expand around center for each position. Try both odd-length (single center) and even-length (two-char center) expansions. Track the best start and length.

When to Use

Substring search with symmetry constraint. Related to Manacher's algorithm for O(n).

Complexity

```
Time:  O(nš)
Space:  O(1)
```

Implementation

```
def longest_palindromic_substring(s: str) -> str:
    """Return the longest palindromic substring of *s*.

    >>> longest_palindromic_substring("babad") in ("bab", "aba")
    True
    >>> longest_palindromic_substring("cbbd")
    'bb'
    """
    if len(s) < 2:
        return s

    start = 0
    max_len = 1

    def _expand(left: int, right: int) -> None:
        nonlocal start, max_len
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        # loop overshoots by one on both sides before stopping, hence the -1
        length = right - left - 1
        if length > max_len:
            start = left + 1
            max_len = length

    for i in range(len(s)):
        _expand(i, i)  # odd-length palindrome centered on i
        _expand(i, i + 1)  # even-length palindrome centered between i and i+1

    return s[start : start + max_len]
```

String to Integer (atoi) — convert a string to a 32-bit signed integer

Problem

Implement atoi which converts a string to a 32-bit signed integer. Handle leading whitespace, an optional +/- sign, consecutive digits, and clamp the result to [-2^31, 2^31 - 1].

Approach

Linear scan with state tracking: skip whitespace, read optional sign, accumulate digits, clamp on overflow. Stop at first non-digit after sign processing.

When to Use

Parsing problems, state machine patterns. Tests attention to edge cases (whitespace, signs, overflow, trailing non-digits).

Complexity

```
Time:  O(n)
Space:  O(1)
```

Implementation

```
INT_MIN = -(2**31)
INT_MAX = 2**31 - 1


def my_atoi(s: str) -> int:
    """Convert string *s* to a 32-bit signed integer.

    >>> my_atoi("42")
    42
    >>> my_atoi("   -42")
    -42
    >>> my_atoi("4193 with words")
    4193
    >>> my_atoi("-91283472332")
    -2147483648
    """
    n = len(s)
    i = 0

    # skip leading whitespace
    while i < n and s[i] == " ":
        i += 1

    if i == n:
        return 0

    # read optional sign
    sign = 1
    if s[i] in ("+", "-"):
        if s[i] == "-":
            sign = -1
        i += 1

    # accumulate digits
    result = 0
    while i < n and s[i].isdigit():
        result = result * 10 + int(s[i])
        i += 1

    result *= sign  # apply sign before clamping
```

```
    # Python ints never overflow like a fixed-width 32-bit int would --
    # clamp manually to emulate that boundary

    if result < INT_MIN:
        return INT_MIN
    if result > INT_MAX:
        return INT_MAX
    return result
```

Valid Anagram — check if two strings are anagrams of each other

Problem

Given two strings, determine if one is an anagram of the other (same characters, same frequencies, different order allowed).

Approach

Counter comparison — count character frequencies for both strings and check equality. Alternate is_anagram_sorted compares the two strings sorted instead.

When to Use

Equivalence class membership — same chars different order. Group anagrams builds on this. Also: frequency matching, permutation checking.

Complexity

```
Time:  O(n) with Counter, O(n log n) with sort
Space:  O(1) -- bounded by alphabet size (26 for lowercase English)
```

Implementation

```
from collections import Counter


def is_anagram(s: str, t: str) -> bool:
    """Return True if *s* and *t* are anagrams of each other.

    >>> is_anagram("anagram", "nagaram")
    True
    >>> is_anagram("rat", "car")
    False
    """
    # Counter equality also fails on unequal length -- no separate len check needed
    return Counter(s) == Counter(t)


# --- sort-and-compare variant (O(n log n), no extra structures) ---
def is_anagram_sorted(s: str, t: str) -> bool:
    """Return True if *s* and *t* are anagrams of each other, via sorting.

    >>> is_anagram_sorted("anagram", "nagaram")
    True
    >>> is_anagram_sorted("rat", "car")
    False
    """
    return sorted(s) == sorted(t)
```

Valid Palindrome — check if string is a palindrome ignoring non-alnum and case

Problem

Given a string, determine if it is a palindrome considering only alphanumeric characters and ignoring case.

Approach

Two pointers from both ends. Skip non-alphanumeric characters. Compare lowercase characters at each pointer position. Alternate is_palindrome_filtered builds a cleaned list via a comprehension and compares it to its reverse.

When to Use

String validity checks, symmetry problems. Foundation for palindrome family (longest palindromic substring, palindrome partitioning).

Complexity

```
Time:  O(n)
Space:  O(1)
```

Implementation

```
def is_palindrome(s: str) -> bool:
    """Return True if *s* is a palindrome (alphanumeric only, case-insensitive).

    >>> is_palindrome("A man, a plan, a canal: Panama")
    True
    >>> is_palindrome("race a car")
    False
    """
    left, right = 0, len(s) - 1
    while left < right:
        # both skip-loops re-check left < right -- otherwise an all-non-alnum
        # tail could walk a pointer past the other and misread the string
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1
    return True


# --- filter + reverse comprehension variant (O(n) space, stdlib idiom) ---
def is_palindrome_filtered(s: str) -> bool:
    """Return True if *s* is a palindrome, via a filtered comprehension.

    >>> is_palindrome_filtered("A man, a plan, a canal: Panama")
    True
    >>> is_palindrome_filtered("race a car")
    False
    """
    cleaned = [c.lower() for c in s if c.isalnum()]
    return cleaned == cleaned[::-1]
```

Recursion

Flatten Nested List — recursively flatten arbitrarily deep lists

Problem

Given a nested list of integers (can be arbitrarily deep), flatten it into a single list of integers.

Approach

flatten_recursive: recurse into list elements, yield everything else. flatten_iterative is the alternate using an explicit stack instead of the call stack (push in reverse to maintain order).

When to Use

Processing recursive/nested data structures — JSON, XML, file trees, AST traversal.

Complexity

```
Time:  O(n) -- where n = total elements across all nesting levels
Space:  O(d) recursive / O(n) iterative -- d = max depth
```

Implementation

```
from collections.abc import Iterator, Sequence
from typing import Any

# Recursive nesting type -- list[int | list[...]] can't be expressed finitely,
# so we use Sequence[int | list[Any]] for covariant read-only params.
type NestedList = Sequence[int | list[Any]]


def flatten_recursive(nested: NestedList) -> list[int]:
    """Flatten *nested* list of integers recursively.

    >>> flatten_recursive([1, [2, [3, 4], 5], 6])
    [1, 2, 3, 4, 5, 6]
    >>> flatten_recursive([])
    []
    """
    return list(_flatten_helper(nested))


def _flatten_helper(nested: NestedList) -> Iterator[int]:
    for item in nested:
        if isinstance(item, list):
            yield from _flatten_helper(item)
        else:
            yield item


# --- iterative alternate: explicit stack instead of the call stack ---
def flatten_iterative(nested: NestedList) -> list[int]:
    """Flatten *nested* list of integers iteratively using a stack.

    >>> flatten_iterative([1, [2, [3, 4], 5], 6])
    [1, 2, 3, 4, 5, 6]
    >>> flatten_iterative([])
    []
    """
    result: list[int] = []
    # push in reverse so pop() (LIFO) still yields items in original order
    stack: list[int | list[Any]] = list(reversed(nested))
    while stack:
        item = stack.pop()
        if isinstance(item, list):
```

```
            for sub in reversed(item):
                stack.append(sub)
        else:
            result.append(item)
    return result
```

Generate Parentheses — all valid combinations of n pairs

Problem

Given n pairs of parentheses, generate all combinations of well-formed (balanced) parentheses.

Approach

Backtracking: maintain counts of open and close parens placed so far. Add '(' if open $<$ n. Add ')' if close $<$ open. Base case: length == 2*n.

When to Use

Generating all valid structures (expressions, trees, paths). Classic backtracking with pruning. Output count follows Catalan numbers.

Complexity

```
Time:  O(4^n / sqrt(n)) -- nth Catalan number
Space:  O(n) -- recursion depth (excluding output)
```

Implementation

```
def generate_parentheses(n: int) -> list[str]:
    """Return all valid combinations of *n* pairs of parentheses.

    >>> generate_parentheses(1)
    ['()']
    >>> sorted(generate_parentheses(2))
    ['(())', '()()']
    """
    result: list[str] = []

    def backtrack(current: list[str], open_count: int, close_count: int) -> None:
        if len(current) == 2 * n:
            result.append("".join(current))
            return
        if open_count < n:
            current.append("(")
            backtrack(current, open_count + 1, close_count)
            current.pop()  # undo before trying the sibling branch
        # invariant: close must never exceed open, or the prefix is invalid
        if close_count < open_count:
            current.append(")")
            backtrack(current, open_count, close_count + 1)
            current.pop()

    if n > 0:
        backtrack([], 0, 0)
    return result
```

Letter Combinations of a Phone Number — digit-to-letter mapping

Problem

Given a string containing digits from 2-9, return all possible letter combinations that the number could represent on a phone keypad.

Approach

letter_combinations: recursive backtracking that maps each digit to its letters and builds combinations one digit at a time, picking one letter per level then recursing on the rest. letter_combinations_itertools is the stdlib alternate using itertools.product over each digit's letters.

When to Use

Cartesian product generation, combinatorial enumeration. Similar structure to permutations but across different character sets.

Complexity

```
Time:  O(4^n) -- where n = number of digits (worst case:  7 and 9 have 4 letters)
Space:  O(n) -- recursion depth (excluding output)
```

Implementation

```
import itertools

DIGIT_TO_LETTERS: dict[str, str] = {
    "2": "abc",
    "3": "def",
    "4": "ghi",
    "5": "jkl",
    "6": "mno",
    "7": "pqrs",
    "8": "tuv",
    "9": "wxyz",
}


def letter_combinations(digits: str) -> list[str]:
    """Return all letter combinations for the given phone *digits*.

    >>> letter_combinations("23")
    ['ad', 'ae', 'af', 'bd', 'be', 'bf', 'cd', 'ce', 'cf']
    >>> letter_combinations("")
    []
    """
    if not digits:
        return []

    result: list[str] = []

    def backtrack(index: int, current: list[str]) -> None:
        if index == len(digits):
            result.append("".join(current))
            return
        # digits are assumed 2-9 (problem constraint) -- 0/1 aren't mapped
        for letter in DIGIT_TO_LETTERS[digits[index]]:
            current.append(letter)
            backtrack(index + 1, current)
            current.pop()

    backtrack(0, [])
    return result
```

```
# --- stdlib alternate: itertools.product over each digit's letters ---
def letter_combinations_itertools(digits: str) -> list[str]:
    """Return all letter combinations for the given phone *digits*.

    >>> letter_combinations_itertools("23")
    ['ad', 'ae', 'af', 'bd', 'be', 'bf', 'cd', 'ce', 'cf']
    >>> letter_combinations_itertools("")
    []
    """
    if not digits:
        return []

    letter_groups = (DIGIT_TO_LETTERS[d] for d in digits)
    return ["".join(combo) for combo in itertools.product(*letter_groups)]
```

Pow(x, n) — compute x raised to the power n using fast exponentiation

Problem

Implement pow(x, n), which calculates x raised to the power n (i.e., x^n). Handle negative exponents.

Approach

my_pow: fast exponentiation (exponentiation by squaring), recursive. If n is even, pow(x, n) = pow(x*x, n//2). If n is odd, pow(x, n) = x * pow(x, n-1). For negative n, compute pow(1/x, -n). my_pow_iterative is the alternate: the same squaring trick unrolled into a loop that walks n's bits, using O(1) space instead of O(log n) recursion depth.

When to Use

Any "repeated operation" that can be halved — exponentiation, matrix power, modular arithmetic. Foundation of divide-and-conquer thinking.

Complexity

```
Time:  O(log n)
Space:  O(log n) -- recursion stack
```

Implementation

```
def my_pow(x: float, n: int) -> float:
    """Compute x raised to the power n via fast exponentiation.

    >>> my_pow(2.0, 10)
    1024.0
    >>> my_pow(2.0, -2)
    0.25
    >>> my_pow(0.0, 0)
    1.0
    """
    if n < 0:
        # inverting the base here means x == 0 raises ZeroDivisionError,
        # matching the real mathematical domain (0 ** negative is undefined)
        return my_pow(1.0 / x, -n)
    if n == 0:
        return 1.0
    if n % 2 == 0:
        # squaring x while halving n keeps total work at O(log n) calls
        return my_pow(x * x, n // 2)
    return x * my_pow(x, n - 1)


# --- iterative alternate: same squaring trick, O(1) space ---
def my_pow_iterative(x: float, n: int) -> float:
    """Compute x raised to the power n via iterative fast exponentiation.

    >>> my_pow_iterative(2.0, 10)
    1024.0
    >>> my_pow_iterative(2.0, -2)
    0.25
    >>> my_pow_iterative(0.0, 0)
    1.0
    """
    if n < 0:
        x = 1.0 / x
        n = -n

    result = 1.0
    while n > 0:
        if n % 2 == 1:
```

```
            result *= x
        x *= x
        n //= 2
    return result
```

Tower of Hanoi — move n disks between pegs

Problem

Move n disks from a source peg to a target peg using an auxiliary peg. Only one disk may be moved at a time, and a larger disk may never be placed on top of a smaller one.

Approach

Classic recursion: move n-1 disks from source to auxiliary, move the largest disk from source to target, then move n-1 disks from auxiliary to target. Base case: n == 0 (do nothing).

When to Use

Pure recursive decomposition — the canonical example. Demonstrates how recursion breaks problems into identical sub-problems. Also: understanding call stack depth and O(2^n) explosion.

Complexity

```
Time:  O(2^n) -- number of moves
Space:  O(n) -- recursion depth
```

Implementation

```
def tower_of_hanoi(
    n: int,
    source: str = "A",
    target: str = "C",
    auxiliary: str = "B",
) -> list[tuple[str, str]]:
    """Solve Tower of Hanoi for *n* disks, returning moves as (from, to) tuples.

    >>> tower_of_hanoi(1)
    [('A', 'C')]
    >>> tower_of_hanoi(2)
    [('A', 'B'), ('A', 'C'), ('B', 'C')]
    """
    moves: list[tuple[str, str]] = []

    def solve(disks: int, src: str, tgt: str, aux: str) -> None:
        if disks == 0:
            return
        # move n-1 disks out of the way: src -> aux, using tgt as the spare
        solve(disks - 1, src, aux, tgt)
        moves.append((src, tgt))
        # move the same n-1 disks onto the largest one: aux -> tgt, using
        # src (now holding only the largest disk) as the spare
        solve(disks - 1, aux, tgt, src)

    solve(n, source, target, auxiliary)
    return moves
```

Bit Manipulation

Counting Bits — count 1-bits for all numbers 0..n

Problem

Given an integer n, return an array of length n+1 where the i-th element is the number of 1-bits in the binary representation of i.

Approach

DP recurrence: dp[i] = dp[i » 1] + (i & 1). The number of set bits in i equals the bits in i//2 plus whether the least significant bit is set. Alternate counting_bits_listcomp gets the same answer with bin(i).count("1").

When to Use

DP on binary representation — "count set bits for 0..n", "Hamming weight table". Recurrence dp[i] = dp[i»1] + (i&1) builds on previously computed values. Also: popcount lookup tables.

Complexity

```
Time:  O(n)
Space:  O(n)
```

Implementation

```
def counting_bits(n: int) -> list[int]:
    """Return a list of bit counts for 0..n.

    >>> counting_bits(5)
    [0, 1, 1, 2, 1, 2]
    >>> counting_bits(0)
    [0]
    """
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        dp[i] = dp[i >> 1] + (i & 1)  # bits(i) = bits(i//2) + i's own last bit
    return dp


# --- comprehension alternate: bin().count("1") per value (stdlib, O(n * bit-width)) ---
def counting_bits_listcomp(n: int) -> list[int]:
    """Return bit counts for 0..n using bin().count("1") (stdlib).

    >>> counting_bits_listcomp(5)
    [0, 1, 1, 2, 1, 2]
    >>> counting_bits_listcomp(0)
    [0]
    """
    return [bin(i).count("1") for i in range(n + 1)]
```

Reverse Bits — reverse the bits of a 32-bit unsigned integer

Problem

Given a 32-bit unsigned integer, return the integer obtained by reversing all 32 bits.

Approach

Bit-by-bit: extract the lowest bit of n, shift result left, OR the bit in, and shift n right. Repeat 32 times. Also includes a divide-and-conquer approach that swaps groups of bits using masks.

When to Use

Bit-level transformation — "reverse bits", "swap nibbles/bytes", "bit-reversal permutation" (used in FFT). Divide-and-conquer mask approach is constant-time. Also: endianness conversion.

Complexity

```
Time:  O(1) (fixed 32 iterations)
Space:  O(1)
```

Implementation

```
UINT32_BITS = 32


def reverse_bits(n: int) -> int:
    """Reverse the bits of a 32-bit unsigned integer.

    >>> reverse_bits(0b00000010100101000001111010011100)
    964176192
    >>> bin(reverse_bits(0b00000010100101000001111010011100))
    '0b11100101111000001010010100000'
    """
    result = 0
    for _ in range(UINT32_BITS):
        result = (result << 1) | (n & 1)
        n >>= 1
    return result


# --- divide-and-conquer alternate: swap bit groups via bitmasks (O(1), no loop) ---
def reverse_bits_divide_conquer(n: int) -> int:
    """Reverse bits using divide-and-conquer with bitmasks.

    >>> reverse_bits_divide_conquer(0b00000010100101000001111010011100)
    964176192
    """
    n = ((n & 0xFFFF0000) >> 16) | ((n & 0x0000FFFF) << 16)  # swap 16-bit halves
    n = ((n & 0xFF00FF00) >> 8) | ((n & 0x00FF00FF) << 8)
    n = ((n & 0xF0F0F0F0) >> 4) | ((n & 0x0F0F0F0F) << 4)
    n = ((n & 0xCCCCCCCC) >> 2) | ((n & 0x33333333) << 2)
    return ((n & 0xAAAAAAAA) >> 1) | ((n & 0x55555555) << 1)  # swap adjacent bits, finest grain
```

Single Number — find the element appearing once (others twice)

Problem

Given a non-empty array where every element appears twice except for one, find that single element.

Approach

XOR all elements. Since a ^ a = 0 and a ^ 0 = a, all pairs cancel out, leaving only the single element. Alternate single_number_reduce folds the same XOR with functools.reduce.

When to Use

XOR uniqueness — "find the element appearing once while others appear twice". XOR cancels pairs: a ^ a = 0. Extends to "two unique numbers" by splitting on a differing bit. Keywords: "unique", "missing", "duplicate".

Complexity

```
Time:  O(n)
Space:  O(1)
```

Implementation

```
from collections.abc import Sequence
from functools import reduce
from operator import xor


def single_number(nums: Sequence[int]) -> int:
    """Return the element that appears exactly once.

    >>> single_number([4, 1, 2, 1, 2])
    4
    >>> single_number([1])
    1
    """
    result = 0
    for n in nums:
        result ^= n  # a^a=0 cancels pairs, a^0=a preserves the single value
    return result


# --- functools.reduce alternate: fold XOR across the sequence (stdlib idiom) ---
def single_number_reduce(nums: Sequence[int]) -> int:
    """Return the element that appears exactly once, via functools.reduce.

    >>> single_number_reduce([4, 1, 2, 1, 2])
    4
    >>> single_number_reduce([1])
    1
    """
    return reduce(xor, nums, 0)
```

Math & Number Theory

Check whether one integer is prime

Problem

Given an integer n, return whether n is prime. A prime is an integer greater than 1 with no positive divisors other than 1 and itself.

Approach

Reject values below 2 and small composite factors first. Then try possible factors through sqrt(n), checking only numbers of the form 6k - 1 and 6k + 1. This answers one primality query without building a table of every prime up to n.

When to Use

Checking one or a few individual values. Use a sieve instead when many queries share the same upper bound or when every prime up to n is needed.

Complexity

```
Time:  O(sqrt(n))
Space:  O(1)
```

Implementation

```
def is_prime(n: int) -> bool:
    """Return whether *n* is prime.

    >>> is_prime(29)
    True
    >>> is_prime(1)
    False
    """
    if n < 2:
        return False
    if n in (2, 3):
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False

    factor = 5
    while factor * factor <= n:
        if n % factor == 0 or n % (factor + 2) == 0:
            return False
        factor += 6
    return True
```

Sieve of Eratosthenes: all primes up to n

Problem

Given a non-negative integer n, return every prime number less than or equal to n, in increasing order.

Approach

Whiteboard form (sieve_of_eratosthenes): allocate a boolean array of size n + 1, mark 0 and 1 as not prime, then for each candidate p starting at 2, if p is still marked prime, strike every multiple of p starting at p*p (smaller multiples of p already got struck by a smaller prime factor). Collect the indices still marked prime. Pythonic form (sieve_slices): same invariant, but each strike pass is a single slice assignment ("sieve[p * p :: p]") instead of a nested loop, and the surviving indices are collected with a list comprehension over "enumerate".

When to Use

Precomputing all primes up to n for many repeated queries (primality checks, factorization prep, counting primes / pi(n)). Contrast with per-number trial division, which costs O(sqrt(n)) *per query*. The sieve amortizes that cost across every number up to n in one pass.

Complexity

```
Time:  O(n log log n)
Space:  O(n)
```

Implementation

```
from math import isqrt


def sieve_of_eratosthenes(n: int) -> list[int]:
    """Return every prime <= n, in increasing order.

    >>> sieve_of_eratosthenes(10)
    [2, 3, 5, 7]
    >>> sieve_of_eratosthenes(1)
    []
    """
    if n < 2:
        return []

    is_prime = bytearray([1]) * (n + 1)
    is_prime[0] = is_prime[1] = 0  # 0 and 1 are not prime by definition

    for p in range(2, isqrt(n) + 1):  # beyond sqrt(n), no unstruck composite remains
        if is_prime[p]:
            for multiple in range(p * p, n + 1, p):  # below p*p already struck
                is_prime[multiple] = 0

    return [i for i, prime in enumerate(is_prime) if prime]


# Pythonic slice-assignment and comprehension form
def sieve_slices(n: int) -> list[int]:
    """Return every prime <= n via strided slice assignment.

    >>> sieve_slices(10)
    [2, 3, 5, 7]
    >>> sieve_slices(1)
    []
    """
    if n < 2:
        return []
```

```
    is_prime = bytearray([1]) * (n + 1)
    is_prime[0] = is_prime[1] = 0

    for p in range(2, isqrt(n) + 1):
        if is_prime[p]:
            start = p * p
            # bytes(span) must be exactly as long as the strided slice it replaces
            span = len(range(start, n + 1, p))
            is_prime[start::p] = bytes(span)

    return [i for i, prime in enumerate(is_prime) if prime]
```

Patterns

Sliding window pattern template

Implementation

```
from collections.abc import Sequence
from typing import TypeVar

T = TypeVar("T")


def max_sum_subarray(nums: Sequence[int], k: int) -> int:
    """Return the maximum sum of any contiguous subarray of length k.

    Complexity:
        Time:  O(n)
        Space: O(1)

    >>> max_sum_subarray([2, 1, 5, 1, 3, 2], 3)
    9
    """
    if k <= 0 or k > len(nums):
        msg = f"k={k} out of range for length {len(nums)}"
        raise ValueError(msg)

    window_sum = sum(nums[:k])
    best = window_sum

    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]  # add incoming, drop the element k back
        best = max(best, window_sum)

    return best
```

Appendix: Interview Topics

Concurrency & Parallelism Primitives

Concurrency is structure (interleaving independent tasks); parallelism is execution (work at the same instant on multiple cores). Pick the model by bottleneck: I/O-bound work wants concurrency (asyncio or threads), CPU-bound work wants parallelism (processes or subinterpreters). In CPython the GIL serializes bytecode so threads do NOT give CPU parallelism, but they do overlap blocking I/O. The hard problems are shared mutable state (race conditions, atomicity) and liveness (deadlock, starvation); the durable fixes are immutability, message passing over queues, and disciplined lock ordering.

Key Points

- Concurrency vs parallelism: concurrency = dealing with many things at once (structure/interleaving); parallelism = doing many things at once (>=2 cores). You can have concurrency on 1 core.
- I/O-bound -> overlap waits (asyncio/threads). CPU-bound -> use real cores (processes/subinterpreters/native code). Diagnose the bottleneck before choosing.
- Threads: shared address space, cheap to spawn, fast data sharing, but need locks; a crash/segfault in native code can take down the whole process.
- Processes: isolated memory, true parallelism even under the GIL, robust fault isolation; cost = IPC/serialization (pickle) and higher spawn/memory overhead.
- asyncio: single thread, single event loop, cooperative multitasking via await; thousands of sockets cheaply, but ONE blocking call (CPU or sync I/O) stalls every task. Offload blocking work to run_in_executor.
- GIL = one mutex; only one thread executes Python bytecode at a time in CPython. Released around blocking I/O and many C extensions (NumPy), so threads still help I/O-bound code.
- GIL protects the interpreter's internal state and makes individual bytecodes atomic; it does NOT make your multi-step operations atomic (read-modify-write across bytecodes can still race).
- Python 3.12 added per-interpreter GIL (subinterpreters, PEP 684); 3.13 added free-threaded build (PEP 703, no-GIL, opt-in) and stdlib subinterpreters (PEP 734).
- Lock = mutual exclusion (non-reentrant in Python: same thread re-acquiring deadlocks). RLock = reentrant, owner can acquire N times, must release N times.
- Semaphore = counter permitting N concurrent holders (bound a resource pool / limit concurrency). BoundedSemaphore raises if released too many times.
- Condition = lock + wait/notify; wait() in a while-loop on a predicate to handle spurious wakeups and re-checks. Event = one-bit broadcast flag (set/clear/wait) for signaling.
- Coffman's 4 conditions for deadlock (ALL required): mutual exclusion, hold-and-wait, no preemption, circular wait. Break any one to prevent deadlock.
- Deadlock avoidance: global lock ordering (break circular wait), acquire-all-or-none / try-lock with backoff (break hold-and-wait), timeouts (introduce preemption), or avoid shared locks entirely (queues).
- Race condition = correctness depends on timing/interleaving of unsynchronized access to shared mutable state. Atomicity = an operation completes indivisibly (all-or-nothing, no observable midpoint).
- queue.Queue is thread-safe with built-in locking; bounded queue (maxsize) gives backpressure so a fast producer cannot exhaust memory (blocks until a slot frees).
- Futures/pools: concurrent.futures gives ThreadPoolExecutor (I/O) and ProcessPoolExecutor (CPU). A Future is a handle to a pending result; .result() blocks, as_completed() yields in finish order.
- Decision rule: many sockets/high I/O -> asyncio; blocking I/O libs or simple overlap -> threads; CPU crunch in pure Python -> processes (or subinterpreters / release GIL in C / use NumPy).

Execution models in CPython

Model	Parallel CPU?	Best for
Threads	No (GIL)	I/O-bound, blocking libs, shared state
Processes	Yes	CPU-bound, fault isolation
asyncio	No	Massive I/O concurrency, many sockets
Subinterpreters	Yes (3.12+)	CPU + isolation, cheaper than procs

Synchronization primitives

Primitive	Purpose	Note
Lock	Mutual exclusion	Non-reentrant; re-acquire deadlocks
RLock	Reentrant mutex	Acquire N -> release N, same thread
Semaphore	Allow N holders	Bound pools / limit concurrency
Condition	Wait/notify	while-loop the predicate
Event	Broadcast flag	set/clear/wait, one-shot signal
Queue	Thread-safe handoff	maxsize -> backpressure

GIL: protects vs does NOT protect

Protects	Does NOT protect
Interpreter internal state	Your multi-step invariants
Single bytecode atomicity	Read-modify-write (n += 1)
Refcount integrity	Compound/check-then-act logic

Race vs fix: n += 1 is 3 bytecodes (load/add/store), not atomic

```
# RACE: load, add, store can interleave
counter = 0
def bump():
    global counter
    for _ in range(100000):
        counter += 1   # not atomic!

# FIX: serialize the read-modify-write
lock = threading.Lock()
def bump_safe():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1
```

Bounded producer-consumer with backpressure + sentinel shutdown

```
import queue, threading
q = queue.Queue(maxsize=100)   # bounded -> backpressure
DONE = object()

def producer(items):
    for it in items:
        q.put(it)              # blocks if full
    q.put(DONE)

def consumer():
    while True:
        it = q.get()
        if it is DONE:
            q.task_done(); break
        handle(it)
        q.task_done()
```

Deadlock by lock-order inversion -> fix with global ordering

```
# DEADLOCK: T1 locks A then B; T2 locks B then A
# FIX: every thread acquires in one global order
def transfer(a, b, amt):
    first, second = sorted((a, b), key=id)
    with first.lock:
        with second.lock:
            a.bal -= amt; b.bal += amt
```

Futures: pick pool by bottleneck

```
from concurrent.futures import (
    ThreadPoolExecutor, ProcessPoolExecutor, as_completed)

# I/O-bound: threads overlap waits
with ThreadPoolExecutor(max_workers=32) as ex:
    futs = [ex.submit(fetch, u) for u in urls]
    for f in as_completed(futs):
        use(f.result())   # raises if task raised

# CPU-bound: processes use real cores
with ProcessPoolExecutor() as ex:
    results = list(ex.map(crunch, chunks))
```

Condition: guard the predicate in a while loop

```
cond = threading.Condition()
ready = False
def waiter():
    with cond:
        while not ready:        # re-check: spurious wakeups
            cond.wait()
        consume()
def signaler():
    with cond:
        global ready; ready = True
        cond.notify_all()
```

Pitfalls

- Assuming threads speed up CPU-bound Python: the GIL serializes bytecode, so you get little/no speedup (often slower due to contention). Use processes/subinterpreters or drop into C/NumPy.
- Treating n += 1, list.append in a loop, or check-then-act (if k not in d: d[k]=...) as atomic. The GIL guarantees single-bytecode atomicity only, not your compound operation.
- One blocking or CPU-heavy call inside asyncio freezes the entire event loop and every coroutine. Use await for I/O and loop.run_in_executor for blocking/CPU work.
- Calling Lock.acquire() twice in the same thread (or recursive code paths) -> self-deadlock; use RLock when re-entry is genuine.
- Inconsistent lock acquisition order across code paths creates circular wait -> deadlock. Enforce one global ordering (e.g., sort by id) or use try-acquire with timeout + backoff.
- Unbounded queues hide backpressure: a fast producer can exhaust memory. Set maxsize so producers block when consumers fall behind.
- Forgetting to signal consumers to stop (sentinel/poison pill or shutdown event) leaves worker threads blocked on get() forever.
- Sharing non-picklable objects, locks, or large data across ProcessPoolExecutor: state is copied via pickle, not shared; mutations in a child are NOT visible to the parent.
- Swallowing exceptions in futures: an error inside a task surfaces only when you call .result(); if you never read the future, the failure is silent.
- Catching/relying on free-threaded (no-GIL) builds in 3.13 as default: it is opt-in and many C extensions are not yet thread-safe under it; verify before depending on real thread parallelism.

Hash Table Internals

A hash table maps keys to slots in an array via a hash function, giving expected O(1) lookup/insert/delete by trading space for time. Performance hinges on hash quality (uniform spread) and load factor (n/m); collisions are unavoidable (pigeonhole / birthday paradox), so the design choice is how to resolve them: separate chaining (slots hold lists) or open addressing (probe other slots). Resize by doubling keeps amortized O(1) inserts; worst case degrades to O(n) when everything collides into one bucket. Consistent hashing reframes the same problem for distributed sharding so that adding/removing a node moves only K/N keys instead of nearly all.

Key Points

- Hash function goals: deterministic (same key -> same hash every time), uniform (spread keys evenly to minimize collisions), fast, and avalanche (small key change -> big hash change). Map to slot via h(key) mod m.
- Load factor alpha = n/m (entries / buckets). It is the single knob driving collision probability and probe cost; keep it bounded by resizing.
- Separate chaining: each bucket is a list (or tree). Insert prepends O(1); search/delete O(1+alpha). Tolerates alpha > 1, simple deletes, but uses pointers + cache-unfriendly.
- Open addressing: all entries live in the array itself; on collision, probe a sequence until an empty slot. Cache-friendly, no pointers, but needs alpha < 1 and degrades sharply as alpha -> 1.
- Linear probing: try i, i+1, i+2 ... Best cache locality but causes primary clustering (long contiguous runs that grow and merge).
- Quadratic probing: try i, i+1, i+4, i+9 ... Breaks primary clustering but has secondary clustering (same-hash keys share a probe path); needs care so it visits all slots.
- Double hashing: step size = second hash of key. Spreads probe sequences per key, minimizing clustering; near-ideal distribution.
- Robin Hood hashing: on insert, the entry with the larger probe distance (poorer) steals the slot from the richer entry, which keeps probing. Equalizes probe lengths, low variance, great for high load factors.
- Average O(1): with good hash + bounded alpha, expected probes are constant (chaining ~1+alpha; linear probing ~1/(1-alpha)). Worst-case O(n): adversarial or terrible hash collides all keys into one bucket -> linear scan.
- Resize / rehash: when alpha exceeds a threshold, allocate a bigger table (double m) and reinsert (rehash) every key (h mod m changes). One resize is O(n), but doubling amortizes to O(1) per insert.
- Tombstones: in open addressing you cannot blank a deleted slot (it would break probe chains that pass through it). Mark it DELETED; lookups skip past, inserts may reuse. Accumulating tombstones bloats probe length -> rehash to clean.
- Hash flooding / DoS: attacker crafts colliding keys to force O(n) buckets. Mitigation: randomized/keyed hashing (e.g. SipHash) so collisions are unpredictable per process.
- Consistent hashing: place nodes and keys on a hash ring (0..2^k-1); a key belongs to the next node clockwise. Adding/removing a node remaps only ~K/N keys (its arc), not the whole keyspace as plain mod-N would.
- Virtual nodes (vnodes): each physical node owns many ring positions. Smooths load imbalance and, on node loss, spreads its keys across many survivors instead of one neighbor.
- Python dict: open addressing (perturbation probe, not chaining), with a compact two-array layout (dense entries array + sparse index array). Preserves insertion order (guaranteed since 3.7) and saves ~20-25% memory. Resizes (typically ~2/3 full) by ~4x for small dicts, ~2x for large. Keys must be hashable (immutable, ___hash___ + ___eq___).

Chaining vs Open Addressing

Aspect	Separate chaining	Open addressing
Storage	buckets -> lists/trees	entries inline in array
Load factor	works at alpha > 1	needs alpha < 1 (~0.7)
Cache behavior	poor (pointer chasing)	good (contiguous)
Delete	easy (unlink node)	needs tombstones
Degrade mode	long lists	clustering, full table

Collision strategies (open addressing)

Strategy	Probe step	Clustering
Linear	i+1, i+2, ...	primary (runs)
Quadratic	i+1, i+4, i+9	secondary
Double hash	k*h2(key)	minimal
Robin Hood	linear + steal	low variance

Cost summary (good hash)

Op	Average	Worst
lookup	O(1)	O(n)
insert	O(1) amortized	O(n) on resize / all-collide
delete	O(1)	O(n)
resize	O(n) one-time	O(n)

Open addressing: linear probe lookup with tombstones

```
EMPTY, DELETED = object(), object()

def find(tbl, key):
    m = len(tbl)
    i = hash(key) % m
    for _ in range(m):
        slot = tbl[i]
        if slot is EMPTY:
            return None          # stop: chain ends
        if slot is not DELETED and slot.key == key:
            return slot.val      # found
        i = (i + 1) % m          # probe next; skip DELETED
    return None
```

Amortized resize by doubling (chaining)

```
def put(ht, key, val):
    if (ht.n + 1) / ht.m > 0.75:   # load factor threshold
        rehash(ht, ht.m * 2)       # O(n), but rare
    b = hash(key) % ht.m
    ht.buckets[b].upsert(key, val)
    ht.n += 1
# n inserts cost O(n) total -> O(1) each amortized
```

Consistent hashing: key -> node on the ring

```
# ring: sorted list of (pos, node), many vnodes per node
def node_for(ring, key):
    h = hash(key) % RING_SIZE
    for pos, node in ring:        # first vnode clockwise
        if pos >= h:
            return node
    return ring[0][1]             # wrap around
# add/remove a node remaps only ~K/N keys
```

Pitfalls

- Confusing load factor thresholds: chaining stays fine past alpha=1, but open addressing blows up as alpha -> 1 (probe count ~1/(1-alpha)); resize earlier (~0.7).
- Forgetting that rehash changes h mod m, so every key may move to a new bucket; you cannot skip reinsertion on resize.
- In open addressing, physically clearing a deleted slot breaks any probe chain that passed through it -> use tombstones, and periodically rehash to purge them.
- Assuming O(1) is guaranteed: a bad/adversarial hash collapses to O(n). For untrusted input use keyed hashing (SipHash) to prevent hash-flooding DoS.

- Plain hash(key) mod N for sharding remaps almost all keys when N changes; that is exactly what consistent hashing avoids.
- Without virtual nodes, consistent hashing gives uneven load and dumps a failed node's whole arc onto its single clockwise neighbor.
- Mutating a key after insertion (or using a mutable/unhashable key) corrupts lookups because its hash/bucket no longer matches; keys must be immutable.
- Relying on Python set iteration order for determinism: dict preserves insertion order, but set does not.
- Quadratic probing can fail to find an empty slot even when one exists unless table size and probe formula are chosen to cover all slots (e.g. power-of-two size with triangular numbers).

Amortized Analysis

Amortized analysis proves a tight bound on the TOTAL cost of a sequence of n operations, then divides by n. It is a worst-case guarantee over the sequence, not a probabilistic average: no randomness or input distribution is assumed. Use it when an operation is occasionally expensive but the expense is "paid for" by many prior cheap operations. The three tools are aggregate (count total work, divide), accounting/banker (overcharge cheap ops, store credit to fund expensive ones), and potential (define a function Phi of the data-structure state; amortized = actual + delta-Phi).

Key Points

- Definition: amortized cost per op = (worst-case total cost of any sequence of n ops) / n. It bounds the AVERAGE over the sequence in the WORST case.
- Aggregate method: bound total work T(n) for n ops directly, then amortized = T(n)/n. Simplest; gives one cost for all op types.
- Accounting (banker) method: assign each op a chosen amortized charge; cheap ops are overcharged and deposit credit on the structure; expensive ops spend stored credit. Invariant: total credit (bank balance) stays >= 0 at all times -> sum of amortized >= sum of actual -> valid upper bound.
- Potential method: pick Phi(state) >= 0 with Phi(initial)=0. amortized_i = actual_i + Phi(after_i) - Phi(before_i). Telescopes: sum(amortized) = sum(actual) + Phi(final) - Phi(initial) >= sum(actual) when Phi(final) >= 0. Phi is the credit balance made into a formula.
- Dynamic array append: keep capacity; when full, allocate new array of size 2*cap and copy. n appends cost n cheap writes + copies at sizes 1,2,4,...,n. Copies sum <= 2n (geometric series). Total <= 3n -> O(1) amortized per append.
- Why doubling (x2), not grow-by-k: with fixed +k growth you resize every k appends copying ~i elements -> total copy work ~ sum over resizes ~ n^2/(2k) = O(n^2), i.e. O(n) amortized per append. Geometric growth makes resize work a convergent geometric series = O(n) total.
- Growth factor f>1 (any constant) gives O(1) amortized: copy cost sums to n * f/(f-1). f=2 is common; smaller f (e.g. 1.5) wastes less memory but copies more often. Constant factor must be >1, not additive.
- Monotonic stack (daily_temperatures): outer loop over n elements; inner while pops. Each index is pushed exactly once and popped at most once across the WHOLE run, so total push+pop work <= 2n -> O(n), even though one iteration's inner loop can be O(n).
- Sliding window (variable-size, two pointers): right pointer advances n times, left pointer only advances and never resets, so left advances <= n times total. Combined pointer moves <= 2n -> O(n) amortized, despite a nested-looking while.
- Amortized vs average-case: amortized = deterministic worst case over a sequence (no probability). Average-case = expectation over a random input distribution. Worst-case (per-op) = single worst op. A single append can still be O(n); amortized O(1) only describes the run.
- Amortized is NOT a tail/latency guarantee: one operation may spike to O(n) (the resize). For hard real-time / DoD low-jitter needs, prefer worst-case-per-op structures or incremental resizing; amortized throughput hides the spike.
- Banker's intuition map: credit on structure == potential energy Phi. Accounting picks per-op charges; potential picks a global state function. Same proof, two notations.

Three Methods Compared

Method	Core idea	Key check
Aggregate	Total T(n), divide by n	Bound T(n) tightly
Accounting	Overcharge cheap, save credit	Bank balance >= 0 always
Potential	amort=actual+dPhi	Phi>=0, Phi(init)=0

Growth Policy for Append

Policy	Total copy work	Amortized/op
Double (x2)	<= 2n (geometric)	O(1)
x f (f>1)	n*f/(f-1)	O(1)
+k fixed	~ n^2/(2k)	O(n)
+1 each time	~ n^2/2	O(n)

Three Cost Notions

Notion	Over what	Randomness?
Worst-case/op	single op	no
Amortized	sequence of ops	no
Average-case	input distribution	yes (expectation)

Dynamic array append: O(1) amortized via doubling

```
def append(self, x):
    if self.size == self.cap:        # full -> grow
        new = [None] * (2 * self.cap) # geometric x2
        for i in range(self.size):    # copy: cost = old size
            new[i] = self.buf[i]
        self.buf, self.cap = new, 2 * self.cap
    self.buf[self.size] = x           # O(1) common case
    self.size += 1
# n appends: writes n + copies (1+2+4+...+n) <= 3n -> O(1)/op
```

Potential proof sketch for append

```
# Let Phi = 2*size - cap  (>=0 between resizes; =0 right after grow)
# Cheap append: actual=1, dPhi=+2 -> amortized = 3
# Resize at size=cap: actual = cap+1 (copy cap + write 1)
#   before grow Phi = 2*cap - cap = cap
#   after  grow Phi = 2*(cap+1) - 2*cap = 2
#   dPhi = 2 - cap ; amortized = (cap+1) + (2 - cap) = 3
# Every op amortizes to 3 = O(1).  Telescoping sum bounds total.
```

Monotonic stack: each index pushed/popped once -> O(n)

```
def daily_temperatures(t):
    res = [0] * len(t)
    st = []                       # holds indices, decreasing temps
    for i, cur in enumerate(t):
        while st and t[st[-1]] < cur:   # amortized: total pops <= n
            j = st.pop()
            res[j] = i - j
        st.append(i)              # each i pushed exactly once
    return res
# Inner while can run O(n) once, but sum of all pops <= n -> O(n).
```

Sliding window: left never rewinds -> O(n)

```
def longest_unique(s):
    seen, left, best = {}, 0, 0
    for right, c in enumerate(s):      # right: n steps
        if c in seen and seen[c] >= left:
            left = seen[c] + 1         # left only moves forward
        seen[c] = right
        best = max(best, right - left + 1)
    return best
# right advances n; left advances total <= n -> <= 2n moves = O(n).
```

Pitfalls

- Calling amortized O(1) a per-operation guarantee. A single resize append is genuinely O(n); only the n-op run averages to O(1). Say 'amortized' out loud, not just 'O(1)'.
- Conflating amortized with average-case. Amortized uses NO probability and holds for the worst sequence; average-case needs an input distribution and gives an expectation. Interviewers at FFRDCs probe this distinction.
- Claiming append is O(1) amortized under additive (+k) growth. Additive growth is O(n) amortized / O(n^2) total; only multiplicative (constant factor > 1) growth works.
- Potential-method errors: forgetting Phi(initial)=0 or letting Phi go negative. If Phi can dip below its start, sum(amortized)

no longer upper-bounds sum(actual).

- Accounting-method error: leaving the bank balance negative at any point. Each expensive op must be fully prepaid by credit already deposited.
- Stating monotonic-stack/sliding-window as O(n^2) from the visible nested loop. The inner loop's total iterations across the whole run are bounded by n (each element enters/leaves once), so it is O(n).
- Assuming amortized bounds tail latency. For real-time/DoD jitter-sensitive paths, the O(n) resize spike matters; mention incremental-resize or fixed-capacity alternatives.
- Forgetting that pop/delete with shrinking can thrash if you shrink at the same threshold you grow; shrink at <= 1/4 full (not 1/2) to keep both grow and shrink O(1) amortized.

Recursion to Iteration Conversion

Every recursive algorithm can be made iterative by simulating the call stack with an explicit stack: each pushed item is a "frame" holding the arguments plus a marker for where to resume. Tail calls (the recursive call is the last action, nothing happens to its result) are special: they need no stack at all and become a plain loop. The hard cases are non-tail recursion (DFS inorder, quicksort, backtracking) where work remains after a child returns; you encode that pending work as a resume-state on the frame. Convert to iteration when the language lacks tail-call optimization (CPython, default limit ~1000) or when depth can exceed the stack (deep/skewed trees, large inputs) and you risk RecursionError or a native stack overflow.

Key Points

- Recursion == implicit stack of activation frames; iteration with an explicit stack reproduces it exactly. Push args + a resume marker; pop to 'return'.
- Tail call: result of the recursive call is returned unchanged with no pending work. It can reuse the same frame -> rewrite as a loop (reassign args, continue). No stack growth.
- TCO (tail-call optimization) is done by Scheme, Lua, most functional langs. NOT by CPython or Java (by spec); Python author Guido rejected it deliberately. So in Python you MUST hand-convert tails.
- Accumulator trick: turn non-tail linear recursion into tail form by carrying a running result as an extra arg (e.g. factorial(n,acc)), then loop it.
- Preorder DFS is the easy case: visit node, then push children -> a single explicit stack, no resume state. Push right child BEFORE left so left pops first.
- Inorder/postorder need resume state because work happens AFTER a child returns. Use a visited flag, a two-color (gray/white) marker, or split frames into sub-steps.
- Quicksort recursion is two calls; only ONE needs the real stack. Push the larger partition, loop on the smaller (Sedgewick) -> stack depth O(log n) worst case instead of O(n).
- Backtracking (permutations, subsets, N-queens) = DFS over a choice tree. Explicit-stack form pushes (partial_solution, remaining_choices); pop, extend, push children. Must undo state when a branch is exhausted.
- When iteration is REQUIRED: depth > recursion limit, adversarial/deep inputs (a 100k-node linked list or degenerate BST), or hard real-time where stack overflow is unacceptable.
- General template: replace 'call f(x)' with stack.push((state=ENTER, x)); the main loop pops, dispatches on state, and pushes follow-up frames for post-child work. A return value goes into a 'last_result' register the parent frame reads.
- Iterative depth is the same as recursive depth (you moved the stack to the heap), so it trades a fixed OS stack for a growable heap stack and a clear error path, not zero memory.
- BFS is naturally iterative with a QUEUE, not a stack; do not confuse DFS-to-stack with BFS. Swapping stack for queue changes traversal order/semantics.

Recursion shape -> iterative conversion

Shape	Conversion	Stack depth
Tail (last act)	Loop: reassign args, continue	O(1)
Linear non-tail	Add accumulator -> tail -> loop	O(1)
Tree, preorder	Explicit stack, push children	O(h)
Tree, inorder	Stack + visited/state marker	O(h)
Quicksort	Push larger half, loop smaller	O(log n)
Backtracking	Stack of (partial, choices)	O(depth)

When to convert in Python

Trigger	Why
Depth > ~1000	Hits sys.recursionlimit -> RecursionError
Deep/skewed tree	h ~ n, native stack overflow
No TCO	CPython never elides tail frames
Hot path	Explicit stack avoids call overhead

Iterative DFS: preorder and inorder of a binary tree

```
# PREORDER (easy: no resume state)
def preorder(root):
    out, st = [], [root] if root else []
    while st:
        n = st.pop()
        out.append(n.val)
        if n.right: st.append(n.right)  # right first
        if n.left:  st.append(n.left)   # so left pops first
    return out

# INORDER (push-left, then process, then go right)
def inorder(root):
    out, st, cur = [], [], root
    while st or cur:
        while cur:          # dive left
            st.append(cur); cur = cur.left
        cur = st.pop()      # leftmost pending
        out.append(cur.val) # visit
        cur = cur.right     # then right subtree
    return out
```

Iterative quicksort (push larger half, loop smaller -> O(log n) stack)

```
def quicksort(a):
    st = [(0, len(a) - 1)]
    while st:
        lo, hi = st.pop()
        if lo >= hi:
            continue
        p = partition(a, lo, hi)     # a[p] now in place
        left  = (lo, p - 1)
        right = (p + 1, hi)
        # push the LARGER side, recurse-by-loop on smaller
        if (left[1]-left[0]) > (right[1]-right[0]):
            st.append(left); st.append(right)
        else:
            st.append(right); st.append(left)
# partition: standard Lomuto/Hoare; returns pivot index
```

General transform: state on stack frames (resume marker)

```
# recursive: do A; r = rec(child); do B(r); return val
ENTER, AFTER = 0, 1
def iterative(root):
    st = [(ENTER, root)]
    ret = None
    while st:
        state, x = st.pop()
        if state == ENTER:
            # do A
            st.append((AFTER, x))     # frame to resume
            st.append((ENTER, child_of(x)))  # the 'call'
        else:  # AFTER: child returned in 'ret'
            # do B(ret); set ret = val for parent
            ret = combine(x, ret)
    return ret
```

Pitfalls

- Pushing left child before right in preorder reverses output order; push right first so left is processed first (LIFO).
- Forgetting the resume marker on non-tail recursion: you revisit nodes or skip the post-child work (B). Inorder/postorder break silently without a visited/state flag.

- Quicksort that pushes both halves unconditionally can still hit O(n) stack on sorted input; you must loop on the smaller partition to guarantee O(log n).
- Confusing stack vs queue: replacing the DFS stack with a queue gives BFS, a different traversal -> wrong answer if order matters.
- Backtracking without undo: when a branch is exhausted you must restore shared mutable state (board, visited set), or push immutable copies; mutation leaks across branches.
- Iteration does not reduce total depth/memory for tree recursion – it moves the stack to the heap. The win is avoiding the fixed native/CPython limit, not using less space.
- Raising sys.setrecursionlimit too high crashes the interpreter with a C stack overflow (segfault) instead of a catchable RecursionError – convert to iteration instead.
- Only the LAST action being a call makes it a tail call; 'return 1 + f(n-1)' is NOT tail (the add is pending) and needs an accumulator first.
- Empty-input edge cases: guard root is None (DFS) and lo >= hi (quicksort) or you index past the end / loop forever.

Fixed-Width & Numeric Pitfalls (C/C++/Java vs Python)

In C/C++/Java, int is a fixed-width two's-complement type (usually 32-bit, range -2^31 .. 2^31-1) that silently overflows; Python ints are arbitrary precision and never overflow, so bugs that crash a C interview disappear in Python. State this contrast out loud: the same algorithm is safe in Python but UB or wraparound in C/C++ and well-defined wraparound (for int operations) only in Java. Whiteboard defenses: compute midpoints as lo+(hi-lo)/2, accumulate sums/hashes under a prime modulus, and watch INT_MIN, unsigned wrap, negative-number division semantics, and float equality.

Key Points

- Fixed width: C/C++ int and Java int are typically 32-bit two's complement: range -2147483648 .. 2147483647; long/long long is 64-bit (-2^63 .. 2^63-1). Python int is unbounded (bignum) -> never overflows.
- Midpoint bug: mid=(lo+hi)/2 overflows when lo+hi > INT_MAX even though both fit. Fix: mid=lo+(hi-lo)/2. Famous in binary search and mergesort (JDK Arrays.binarySearch had this bug, fixed ~2006).
- lo+(hi-lo)/2 assumes lo<=hi (nonnegative gap). General signed-safe form: lo+((hi-lo)»1) for ints, or use unsigned/wider type. In Java, (lo+hi)»>1 (unsigned shift) also fixes it.
- Sum/product/hash overflow: accumulating into 32-bit overflows fast (sum of n ~10^5 values, or factorials beyond 12!, or rolling hashes). Use 64-bit (long) and/or reduce mod a prime each step.
- Competitive convention: answer mod 1e9+7 (1000000007, prime). Take mod after every add/mul. (a*b)%M needs a,b<M and product <2^63 -> cast to long before multiply: (long)a*b%M.
- Modular negatives: in C/Java, (-5)%3 == -1 (sign follows dividend). To get nonnegative residue: ((x%M)+M)%M. Python's -5%3 == 1 already.
- Signed vs unsigned: C unsigned arithmetic is modular (well-defined wrap); signed overflow is UNDEFINED BEHAVIOR (compiler may assume it never happens). Java has no unsigned; all int ops wrap (defined two's complement).
- Two's complement: most-significant bit is the sign; -x = ~x + 1. Range is asymmetric: there is no positive INT_MIN. |INT_MIN| is not representable.
- INT_MIN trap: -INT_MIN, abs(INT_MIN), and INT_MIN/-1 overflow. In C signed overflow is UB; abs(INT_MIN) is UB. Guard before negating/abs.
- Mixing signed+unsigned in C promotes signed to unsigned: e.g. -1 < (unsigned)1 evaluates as (huge) < 1 -> false. Beware size_t in loops counting down.
- Division/truncation with negatives: C/C++/Java truncate toward ZERO (-7/2 == -3); Python // floors toward NEGATIVE INFINITY (-7//2 == -4). Same for modulo sign.
- Float equality: never use == on floats. Compare with tolerance: fabs(a-b) <= eps (or relative eps for large magnitudes). 0.1+0.2 != 0.3. NaN != NaN is always true.

Language numeric model contrast

Aspect	C/C++	Python / Java
int width	32-bit, overflows	Py: unbounded; Java: 32-bit wraps
signed overflow	Undefined Behavior	Py: n/a; Java: defined wrap
unsigned type	yes, modular wrap	Py/Java: none
a/b negatives	trunc toward 0	Py: floor; Java: trunc 0
-5 % 3	-2	Py: 1; Java: -2
bignum default	no	Py: yes; Java: no (BigInteger)

Common overflow sites and the fix

Site	Bug	Fix
binary search mid	(lo+hi)/2	lo+(hi-lo)/2
prefix sum	32-bit accum	use 64-bit long
modmul	a*b in 32-bit	(long)a*b % M
abs/negate	abs(INT_MIN) UB	guard ==INT_MIN
hash accumulate	wrap to neg	reduce mod prime

Overflow-safe binary search midpoint

```
// BAD: (lo+hi) can overflow int
// int mid = (lo + hi) / 2;

// GOOD (C/C++/Java), assumes lo <= hi:
int mid = lo + (hi - lo) / 2;

// Java alternative, unsigned shift:
int mid2 = (lo + hi) >>> 1;
```

Modular sum/product convention (mod 1e9+7)

```
const long M = 1000000007L;
long acc = 0;
for (int x : a) acc = (acc + x) % M;   // sum
long prod = 1;
for (int x : a) prod = (prod * x) % M;  // x,prod < M, fits 2^63
// nonnegative residue if x could be negative:
long r = ((x % M) + M) % M;
```

INT_MIN / negation guard and float compare

```
// abs(INT_MIN) and -INT_MIN are UB in C (overflow)
if (x == INT_MIN) handle_special(); else y = (x < 0) ? -x : x;

// Float equality with tolerance
bool eq(double a, double b) {
  double eps = 1e-9;
  return fabs(a - b) <= eps * fmax(1.0, fmax(fabs(a), fabs(b)));
}
// NaN check: a != a is true only for NaN
```

Pitfalls

- Saying 'lo+(hi-lo)/2 always works' without noting it needs lo<=hi. If lo>hi or values can be negative, the gap can still overflow; use a wider type or unsigned shift.
- Doing (long) cast AFTER the multiply: (long)(a*b) overflows in 32-bit first. Cast a operand first: (long)a*b.
- Forgetting that Java int wraps silently with NO exception (unlike Python which just grows) -> bugs are invisible until a test value crosses 2^31.
- Assuming -5 % 3 is positive because Python does it; C/C++/Java give -2. Normalize with ((x%M)+M)%M for array indexing / hashing.
- Relying on signed overflow 'wrapping' in C – it is UB; compilers optimize assuming it cannot happen (e.g. x+1 > x always true), producing surprising behavior.
- Mixing signed and unsigned (or using size_t in a countdown loop for(i=n-1; i>=0; ...) with unsigned i) -> infinite loop, since unsigned i>=0 is always true.
- Using == on floats, or comparing accumulated float sums to an exact constant. Also forgetting NaN breaks ordering: NaN compares false to everything, so sorts/min/max misbehave.
- Truncation vs floor: off-by-one in index math (e.g. negative modular bucket) when porting Python // logic to C/Java division.
- Char/byte sign: in C, char may be signed; reading bytes >127 can go negative. Use unsigned char for byte values and hashing.