ASI Algorithm Study Guide

Printable Reference Booklet

April 3, 2026

Contents

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

ASI Domain Mapping

Aviation Problem	Algorithm
Route aircraft around weather	A*, weighted graph
Track nearby aircraft	Geohash, KD-tree
Schedule maintenance with deps	Topological sort
Optimize fuel across routes	Dijkstra, Bellman-Ford
Real-time position streaming	Sliding window
Airspace network planning	MST, network flow
Signal processing (ADS-B)	FFT, DCT

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.

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))
        groups[key].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.

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


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

    # Prefix pass: result[i] = product of nums[0..i-1]
    prefix = 1
    for i in range(n):
        result[i] = prefix
        prefix *= nums[i]

    # Suffix pass: multiply in product of nums[i+1..n-1]
    suffix = 1
    for i in range(n - 1, -1, -1):
        result[i] *= suffix
        suffix *= nums[i]

    return result
```

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.

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

```
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)
    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]
```

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

    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)
        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]).

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


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:
        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
```

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):
        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

            if not end or right - left < end - start:
                start, end = left, right

            # Invalidate window to search for smaller ones
            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.

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:
            j = stack.pop()
            result[j] = i - j
        stack.append(i)

    return result
```

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.

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:
        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]
```

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:
            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. Aviation: altitude/waypoint lookup tables.

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
        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
        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.

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
            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
```

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).

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 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)
```

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. Aviation: caching decoded METAR/TAF data, recent flight plan lookups.

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)
        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)


# --- 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.

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
    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
```

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


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
    head.next = None
    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
    root.left, root.right = root.right, root.left
    invert_tree(root.left)
    invert_tree(root.right)
    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] = []
        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
    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. ASI relevance: airport code lookup (e.g., "BO" -> ["BOS", "BOI", "BOG"]), flight ID prefix search, geospatial name indexing.

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
            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]
            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):
            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
    if not (lo < root.val < hi):
        return False
    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. ASI relevance: flight route optimization with destination-aware pruning. Use Manhattan distance for grids, great-circle distance for geospatial.

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)
        if (r, c) == goal:
            return _reconstruct_path(came_from, goal)

        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

    for _ in range(num_nodes - 1):
        updated = False
        for u, v, w in edges:
            if dist[u] < INF and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                updated = True
        if not updated:
            break

    # Check for negative cycles
    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.

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 = GraphNode(1)
    ... n2 = GraphNode(2)
    >>> n1.neighbors = [n2]
    ... n2.neighbors = [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

    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]
```

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:
        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)

    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)

    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. Flight routing, network latency, road navigation. For negative weights use Bellman-Ford instead. ASI relevance: core of route optimization in PRESCIENCE.

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)
        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. Aviation: nearby airport/NAVAID search, sector boundary queries. 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)

        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]
        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. Aviation: closest waypoint/airport lookup, collision avoidance in 3D airspace. 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)

        # Only search the far side if the splitting plane is closer than best
        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)
        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. Aviation: minimum-cost ground infrastructure linking airports.

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:
            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
        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))
            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)
        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). Aviation: minimum propagation delay across a network of stations.

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)
        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))

    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. Aviation: air traffic flow management, gate assignment optimization.

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:
        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]
            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
    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):
            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
            ):
                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)

    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]:
            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.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.

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:
        for _ in range(len(queue)):
            word = queue.popleft()
            if word == end_word:
                return level
            for neighbor in _neighbors(word, word_set, visited):
                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":
            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
```

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.

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

```
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):
        prev2, prev1 = prev1, prev1 + prev2
    return prev1
```

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

    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)

    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. Space-optimized to a single row since each cell only depends on the current and previous row.

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):
            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]
```

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. Include both the 2D tabulation (for clarity) and the space-optimized 1D version (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. Aviation: cargo loading optimization, fuel vs payload tradeoff.

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]


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):
        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. Space-optimized to a single row.

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):
            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]
```

Longest Increasing Subsequence — length of LIS

Problem

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

Approach

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.

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:
        pos = bisect.bisect_left(tails, n)
        if pos == len(tails):
            tails.append(n)
        else:
            tails[pos] = n
    return len(tails)
```

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

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.

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. Aviation: multi-stop flight routing.

Complexity

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

Implementation

```
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):
            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
    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
```

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.

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)

    return heap[0]


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)
        elif val > self._heap[0]:
            heapq.heapreplace(self._heap, val)
        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.

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))

    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
```

```
# --- 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.

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. Aviation: runway scheduling with minimum separation times.

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:
                cooldown.append((time + n, cnt))

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

    return time
```

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]
            if c > remaining:
                break
            path.append(c)
            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

Backtracking with a visited set. At each position, try every unused element, mark it used, recurse, then unmark.

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

```
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):
            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
```

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

Backtracking. At each index, decide to include or exclude the element. Append a snapshot of the current path at every node.

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:
        result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()

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

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. Aviation: gate/runway slot allocation.

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

    sorted_iv = sorted(intervals, key=lambda iv: iv[1])
    count = 1
    end = sorted_iv[0][1]

    for start, finish in sorted_iv[1:]:
        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):
        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])
        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. Aviation: airspace reservation merging, calendar/schedule compression.

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:]:
        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.

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


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:
                return strs[0][:i]

    return strs[0]
```

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
        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
        _expand(i, i + 1)  # even length

    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
```

```
    # clamp to 32-bit signed range
    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.

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
    """
    return Counter(s) == Counter(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.

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:
        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
```

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

Recursive: if element is a list, recurse into it; otherwise yield the integer. Also provide an iterative version using an explicit stack (process in reverse to maintain order).

When to Use

Processing recursive/nested data structures — JSON, XML, file trees, AST traversal. ASI relevance: nested geospatial data, hierarchical flight plans.

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


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] = []
    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()
        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

Recursive backtracking: map each digit to its letters, build combinations one digit at a time. At each level pick one letter for the current digit, then recurse on remaining digits.

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

```
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
        for letter in DIGIT_TO_LETTERS[digits[index]]:
            current.append(letter)
            backtrack(index + 1, current)
            current.pop()

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

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

Fast exponentiation (exponentiation by squaring). 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).

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:
        return my_pow(1.0 / x, -n)
    if n == 0:
        return 1.0
    if n % 2 == 0:
        return my_pow(x * x, n // 2)
    return x * my_pow(x, n - 1)
```

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
        solve(disks - 1, src, aux, tgt)
        moves.append((src, tgt))
        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.

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)
    return dp
```

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


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)
    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)
```

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.

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


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
    return result
```

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]
        best = max(best, window_sum)

    return best
```