LeetCode Notes

Cheat Sheet & Common Patterns

Based on Steven S. Skiena, *The Algorithm Design Manual*, 3rd ed., Springer, 2020

October 2026

Contents

Algorithm Design Cheat Sheet	2
Complexity Analysis	11
Combinatorics	23
Binary Search	33
Two Pointers & Sliding Window	42
Bit Manipulation	53
Sorting	61
Linked Lists	72
Stacks & Queues	82
Intervals	91
Heaps & Priority Queues	99
Trees	109
Graphs	120
Trie & Union Find	142
Combinatorial Search	151
Greedy Algorithms	166
Dynamic Programming	174
References	190

Algorithm Design Cheat Sheet

A quick reference for designing a solution, in the order you need it: read the **constraints** to get a complexity budget, match the **problem signals** to a technique, pick the **algorithm**, then choose the **data structures** that make its operations cheap, and finally check the **Python costs** that can silently break the budget.

This page collects the lookup tables. Each section links to the chapter that explains the material in depth.

From Constraints to Complexity

Budget roughly 10^7–10^8 simple operations per second in Python. The input size usually tells you the intended complexity:

n up to	Target complexity	Typical techniques
10	$O(n!)$, $O(n \cdot n!)$	Permutations, brute force
20	$O(2^n)$, $O(n \cdot 2^n)$	Subsets, bitmask DP, backtracking
40	$O(2^{n/2})$	Meet in the middle
500	$O(n^3)$	Floyd–Warshall, interval DP
5,000	$O(n^2)$	2-D DP, all pairs
10^5–10^6	$O(n \log n)$, $O(n)$	Sorting, heaps, binary search, two pointers, hashing, BFS / DFS
10^9 and beyond	$O(\log n)$, $O(\sqrt{n})$, $O(1)$	Binary search on the answer, math

See: Complexity Analysis, Combinatorics.

Problem Signals

If the problem mentions…	Think of
Sorted input, “find a pair / triplet”	Two pointers, binary search
“Minimum possible maximum”, “at least / at most” with large bounds	Binary search on the answer
“Longest / shortest subarray or substring such that…”	Sliding window (monotone condition) or prefix sums
“Subarray sum equals k”, negative numbers	Prefix sums + hash map
“Next greater / smaller”, “span”, histogram	Monotonic stack
“k largest / smallest / closest”, “median of a stream”	Heap(s)
Overlapping ranges, meetings, “minimum rooms”	Sort by endpoint, sweep line
“All combinations / permutations / subsets”, $n \leq 20$	Backtracking

If the problem mentions…	Think of
“Number of ways”, “minimum cost”, “longest …” with choices	Dynamic programming
“Shortest path”, “fewest steps”, grid moves	BFS (unweighted) / Dijkstra (weighted)
Prerequisites, ordering with dependencies	Topological sort
“Connected”, “same group”, merging accounts	Union-find or DFS
Prefixes, dictionary of words, word search	Trie
“Appears once / twice”, XOR, subsets as masks	Bit manipulation
“Modulo $10^9 + 7$”, “how many ways” with large n	Combinatorics + DP

Algorithm Quick Reference

Design paradigms

Paradigm	Use when	Correctness comes from	Chapter
Sorting first	Pairs, duplicates, nearness, intervals	Order makes neighbors meaningful	Sorting
Two pointers / sliding window	Pairs or subarrays with a monotone condition	An elimination argument	Two Pointers
Divide and conquer	The problem splits into independent halves	Induction; master theorem for cost	Complexity
Greedy	A local choice provably belongs to some optimum	Exchange argument, stays-ahead, or matching a lower bound	Greedy
Dynamic programming	Optimal / counting problems with overlapping subproblems	A correct recurrence over few states	Dynamic Programming
Backtracking	Enumerate or search all configurations; small n	Exhaustive enumeration + pruning	Combinatorial Search
Graph modeling	Relationships, states, dependencies	A classical algorithm on a well-designed graph	Graphs

Sorting and searching

Task	Algorithm	Time	Notes
Sort anything	Timsort (`sorted`, `list.sort`)	$O(n \log n)$	Stable; $O(n)$ on sorted runs
Sort small integer keys	Counting / radix sort	$O(n + k)$	Beats the comparison lower bound
Search sorted data	Binary search (`bisect`)	$O(\log n)$	Also: first / last occurrence, insertion point

Task	Algorithm	Time	Notes
Minimize a value with a monotone feasibility test	Binary search on the answer	$O(\text{check} \cdot \log R)$	Range R of possible answers
k-th smallest	Quickselect	$O(n)$ expected	Or a size-k heap, $O(n \log k)$
Merge k sorted lists	Heap of heads	$O(n \log k)$	
Count inversions	Mergesort with counting	$O(n \log n)$	

Graphs

Problem	Algorithm	Time	Requirements
Reachability, connected components	BFS / DFS	$O(V + E)$	—
Shortest path, unweighted	BFS	$O(V + E)$	—
Shortest path, weights 0 / 1	0-1 BFS (deque)	$O(V + E)$	—
Shortest path, DAG	Relax in topological order	$O(V + E)$	Acyclic; any weights
Shortest path, non-negative weights	Dijkstra (heap)	$O(E \log V)$	No negative edges
Shortest path, negative weights / at most k edges	Bellman–Ford	$O(VE)$ / $O(kE)$	Detects negative cycles
All-pairs shortest paths	Floyd–Warshall	$O(V^3)$	$V \lesssim 400$
Minimum spanning tree	Kruskal (union-find) / Prim (heap)	$O(E \log E)$ / $O(E \log V)$	Connected, undirected
Ordering with dependencies	Topological sort (Kahn / DFS)	$O(V + E)$	Acyclic
Cycle detection	DFS (3 colors) / union-find	$O(V + E)$	Directed / undirected
Two-coloring (bipartite?)	BFS / DFS coloring	$O(V + E)$	—
Strongly connected components	Kosaraju / Tarjan	$O(V + E)$	Directed
Bridges, articulation points	DFS low-link	$O(V + E)$	Undirected

See: Sorting, Binary Search, Graphs.

Which Structure Do I Need?

Start from the operations the algorithm performs, and how often; pick the simplest structure that makes all of them fast.

I need to…	Use	Cost
Look up, insert, delete by exact key	Hash map / set	$O(1)$ expected
Count occurrences	`Counter`	$O(1)$ expected per update

I need to…	Use	Cost
Repeatedly take the smallest (or largest) item, with insertions	Binary heap	$O(\log n)$
Keep the k best items of a stream	Size-k heap	$O(\log k)$ per item
Insert, delete, **and** ask ordered questions (successor, k-th, range)	Balanced BST / `SortedList`	$O(\log n)$
Answer "what's nearest to x?" on fixed data	Sorted array + `bisect`	$O(\log n)$
Process in LIFO order / match nested structure	Stack	$O(1)$
Process in FIFO order / explore by distance	Queue (`deque`)	$O(1)$
Find the next greater / smaller element for all items	Monotonic stack	$O(n)$ total
Max / min of every sliding window	Monotonic deque	$O(n)$ total
Merge groups and ask "same group?"	Union-find	$O(\alpha(n))$
Query words by prefix	Trie	$O(L)$
Range sums with updates	Fenwick tree	$O(\log n)$
Evict the least recently used item	Hash map + doubly linked list (`OrderedDict`)	$O(1)$
Represent a subset of ≤ 64 items compactly	Bitmask (`int`)	$O(1)$

Take-Home Lesson

Data structure design is a balancing act. The fastest structure for operations A *and* B is often not the fastest for A alone or B alone. Picking the wrong structure can be disastrous; picking the very best one matters much less, because several good choices usually perform similarly.

Dictionary Operations: The Master Table

Rows are data structures; columns are operations on n stored items. In each column, the **best** cost is in bold. Unless noted, costs are worst case.

Conventions: *Delete*, *successor*, and *predecessor* are given a reference (index or node) to the item, so they don't include the cost of finding it. *Successor / predecessor* mean the next item in **sorted** order, not physical order. h is the height of a tree.

Structure	Access i-th	Search by key	Insert	Delete	Min / Max	Successor / Predecessor	Build from n items	Iterate in sorted order
Unsorted array (`list`)	$O(1)$	$O(n)$	$O(1)$ [1]	$O(1)$ [2]	$O(n)$	$O(n)$	$O(n)$	$O(n \log n)$
Sorted array	$O(1)$	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n \log n)$	$O(n)$

Structure	Access i-th	Search by key	Insert	Delete	Min / Max	Successor / Predecessor	Build from n items	Iterate in sorted order
Singly linked list	$O(n)$	$O(n)$	$O(1)$	$O(n)$ [3]	$O(n)$	$O(n)$	$O(n)$	$O(n \log n)$
Doubly linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log n)$
Sorted doubly linked list	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(n \log n)$	$O(n)$
Hash table (`dict`, `set`)	—	$O(1)$ [4]	$O(1)$ [4]	$O(1)$ [4]	$O(n)$	$O(n)$	$O(n)$ [4]	$O(n \log n)$
Binary search tree (unbalanced)	$O(h)$ [5]	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$	$O(nh)$	$O(n)$
Balanced BST (AVL, red–black, `SortedList`)	$O(\log n)$ [5]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$ [6]	$O(\log n)$	$O(n \log n)$	$O(n)$
Binary heap (`heapq`)	—	$O(n)$	$O(\log n)$	$O(\log n)$ [7]	$O(1)$ min only	$O(n)$	$O(n)$	$O(n \log n)$

1. Amortized: append to the end of a dynamic array. Inserting at a *specific* position costs $O(n)$.
2. Overwrite the deleted item with the last one and shrink. Order is not preserved; if it must be, delete costs $O(n)$.
3. The *predecessor's* pointer must change, and finding it takes $O(n)$. Copying the successor's value into the node gives $O(1)$ except for the last node (LC 237).
4. Expected, assuming a good hash function. The worst case is $O(n)$ per operation.
5. By rank only if each node stores its subtree size (an order-statistic tree). `sortedcontainers.SortedList` supports indexing.
6. $O(1)$ if a pointer to the extreme element is cached and maintained.
7. Delete-min. Deleting an arbitrary item needs its index; `heapq` doesn't track indices, so use lazy deletion instead.

Reading the table. No row is bold everywhere, which is exactly the point:

- **Hash tables** win at exact-key lookup and update, and are useless for anything involving order.
- **Sorted arrays** win at order queries (min, successor, sorted iteration, binary search) and lose at updates.
- **Heaps** win at "give me the minimum" and nothing else — which is often all you need.
- **Balanced BSTs** are never the fastest at any single operation, but they are the only structure with **no bad column**. Use one when you need both updates *and* ordered queries.

See: Linked Lists, Heaps, Trees.

Containers and Specialized Structures

Structures built for a narrower set of operations, and the best cost for each.

Structure	Python	Operations and costs	Use it when
Stack	`list`	push, pop, peek: $O(1)$ [1]	Nesting, undo, DFS, "most recent unresolved item"
Queue	`collections.deque`	enqueue, dequeue, peek: $O(1)$	BFS, FIFO processing, fairness
Deque	`collections.deque`	push / pop at either end: $O(1)$; index: $O(n)$	Sliding windows, 0-1 BFS
Monotonic stack / deque	`list` / `deque`	$O(1)$ amortized per element	Next greater / smaller element, sliding window max / min
Priority queue	`heapq`	push, pop-min: $O(\log n)$; peek: $O(1)$; heapify: $O(n)$	Repeatedly extract the best item; top-k; Dijkstra, Prim
Two heaps	`heapq` $\times 2$	insert: $O(\log n)$; median: $O(1)$	Running median, split at an order statistic
Union-find	custom	find, union: $O(\alpha(n))$ amortized [2]	Dynamic connectivity, grouping, Kruskal
Trie	nested `dict`	insert, search, prefix query: $O(L)$ for a key of length L	Prefix queries, autocomplete, word search on grids
Bit vector	`int`	member, add, remove: $O(1)$; union, intersection: $O(n/w)$ [3]	Small fixed universe; sets as dictionary keys
Counter / multiset	`collections.Counter`	add, remove, count: $O(1)$ expected	Frequencies, anagrams, window contents
Ordered map / multiset	`sortedcontainers.SortedLi`	add, remove, rank, successor: $O(\log n)$	Sliding window median, "closest value" queries, events

1. Amortized for `list.append` / `list.pop()`.
2. α is the inverse Ackermann function: at most 4 for any realistic n.
3. w is the machine word size; Python integers are arbitrary precision, so large unions cost time proportional to the size of the integer.

See: Stacks & Queues, Heaps, Trie & Union Find, Bit Manipulation.

Range Queries

Structures for questions about contiguous ranges of an array: sums, minimums, counts. The right choice depends on whether the array **changes** between queries. Best cost in each column in bold.

Structure	Build	Point update	Range query	Range update	Supports
Prefix-sum array	$O(n)$	$O(n)$	$O(1)$	$O(n)$	Sums (any invertible operation)
Difference array	$O(n)$	$O(1)$	$O(n)$ [1]	$O(1)$	Many range additions, one final read

Structure	Build	Point update	Range query	Range update	Supports
Sparse table	$O(n \log n)$	—	$O(1)$	—	Min, max, gcd (idempotent operations), static arrays
Fenwick tree (BIT)	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$ [2]	Sums with updates
Segment tree	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$ [3]	Any associative operation, with updates

1. Reading values requires a prefix sum over the differences.
2. Using the range-update / point-query variant, or two trees for range update and range query.
3. With lazy propagation.

Rule of thumb: static array → prefix sums (sums) or sparse table (min / max). Updates interleaved with queries → Fenwick tree (sums) or segment tree (anything else). Offline batch of range additions → difference array.

A Fenwick tree is short enough to write from memory. Index i is responsible for the range ending at i whose length is the lowest set bit of i (see Bit Manipulation):

```python
class FenwickTree:
    """Point update and prefix sum in O(log n). Indices are 0-based externally."""

    def __init__(self, n: int) -> None:
        self.tree = [0] * (n + 1)

    def add(self, i: int, delta: int) -> None:
        i += 1
        while i < len(self.tree):
            self.tree[i] += delta
            i += i & -i                     # move to the next range covering i

    def prefix_sum(self, i: int) -> int:
        """Sum of the first i elements (indices 0 .. i-1)."""
        total = 0
        while i > 0:
            total += self.tree[i]
            i -= i & -i                     # drop the range just added
        return total

    def range_sum(self, lo: int, hi: int) -> int:
        """Sum of indices lo .. hi-1."""
        return self.prefix_sum(hi) - self.prefix_sum(lo)
```

See: Prefix Sums, Intervals: Difference Arrays.

Python Cost Model

A built-in call is a subroutine, not a step: many one-liners hide a loop. All figures are for CPython.

Built-in operations

Container	Operation	Cost
`list`	index `a[i]`, assign `a[i] = x`, `len(a)`	$O(1)$
	`append(x)`, `pop()`	$O(1)$ amortized
	`pop(0)`, `insert(0, x)`, `insert(i, x)`, `del a[i]`	$O(n)$
	`x in a`, `a.index(x)`, `a.count(x)`, `a.remove(x)`	$O(n)$
	`min(a)`, `max(a)`, `sum(a)`	$O(n)$
	slice `a[i:j]`, `a.copy()`, `a + b`	$O(j-i)$, $O(n)$, $O(n+m)$
	`a.sort()`, `sorted(a)`	$O(n \log n)$
`dict` / `set`	`d[k]`, `k in d`, `d[k] = v`, `del d[k]`, `s.add(x)`	$O(1)$ average, $O(n)$ worst [1]
	iteration, copy, `set(a)`, `list(d.keys())`	$O(n)$
	`s & t`, s | t	$O(\min(n, m))$, $O(n+m)$
`collections.deque`	`append`, `appendleft`, `pop`, `popleft`	$O(1)$
	index `q[i]`	$O(n)$ (fast only near the ends)
`heapq`	`heappush`, `heappop`	$O(\log n)$
	`heapify`	$O(n)$
	`h[0]` (peek)	$O(1)$
`str`	index, `len`	$O(1)$
	`s + t`, slicing, `s.replace`, `s.split`	$O(n)$
	`"".join(parts)`	O(total length)
`int`	arithmetic on "small" integers	$O(1)$
	arithmetic on k-digit integers	grows with k (arbitrary precision)

1. Plus the cost of hashing the key: $O(k)$ for a string or tuple of length k.

Performance traps

Looks cheap	Actually costs	Use instead
`list.pop(0)`, `list.insert(0, x)`	$O(n)$	`deque.popleft()`, `deque.appendleft(x)`
`x in some_list`	$O(n)$	`x in some_set`
`s += piece` in a loop	$O(n^2)$ total	`"".join(pieces)`
`arr[1:]` in recursion	$O(n)$ per call, $O(n^2)$ total	Pass indices
`sorted()` inside a loop	$O(n \log n)$ per iteration	Sort once, or use a heap / `SortedList`
`min(arr)` / `max(arr)` in a loop	$O(n)$ per iteration	Heap, monotonic deque, or running value
Deep recursion ($>$ ~1000 levels)	`RecursionError`	Explicit stack, or `sys.setrecursionlimit`

Looks cheap	Actually costs	Use instead
`heapq` with non-comparable items on ties	`TypeError`	`(priority, counter, item)` tuples

See: Complexity Analysis: the RAM model.

> **Source**
>
> This cheat sheet is based on Steven S. Skiena, *The Algorithm Design Manual*, 3rd ed. (Springer, 2020). See References.

Complexity Analysis

An algorithm is a procedure that takes *any* valid instance of a problem and produces the correct output. Two questions follow immediately: **is it correct?** and **how much does it cost?** This chapter is about the second question — how to reason about cost in a way that is independent of the machine, the language, and the particular input you happened to test on.

The goal is not mathematical rigor for its own sake. It is to develop the reflex of looking at a piece of code, or at a problem's constraints, and knowing within seconds whether an idea is fast enough to be worth writing.

The RAM Model of Computation

To compare algorithms without running them, we need an abstract machine to count steps on. The standard choice is the **Random Access Machine (RAM)**:

- Each *simple* operation (`+`, `*`, `-`, `=`, `if`, a function call) takes exactly one step.
- Loops and subroutines are **not** simple operations. They are compositions of simple operations, and their cost depends on how many times they iterate.
- Each memory access takes one step, and memory is unlimited.

Under this model, the running time of an algorithm on an instance is the number of steps it executes. The model is *wrong* in almost every detail — multiplication costs more than addition, cache misses cost a hundred times more than cache hits — and yet it is extraordinarily useful, for the same reason a flat-Earth model is useful for planning a walk to the grocery store: the errors are irrelevant at the scale of the question being asked.

> **⚠ Python breaks the RAM model in sneaky places**
>
> The single most common source of complexity bugs in Python is a line that *looks* like one step but is secretly a loop. `x in my_list` scans the list, `my_list.pop(0)` shifts every element, `arr[i:j]` copies the slice, `s + t` builds a new string, and `min(arr)` reads everything.
> Apply the RAM model faithfully: **a built-in call is a subroutine, not a step.** The Python cost model in the cheat sheet lists the cost of each common built-in and the usual fixes.

> **Take-Home Lesson**
>
> Algorithms can be understood and compared in a language- and machine-independent way. The RAM model gives us a common currency — the number of simple steps — for that comparison.

Best, Worst, and Average Case

Running time depends on more than the input *size*. Sorting an already-sorted array may be much faster than sorting a reversed one. So for each size n there is not one running time but a whole distribution of them, one per possible input.

Three numerical functions summarize that distribution:

- **Worst case** — the *maximum* number of steps over all inputs of size n.
- **Best case** — the *minimum* number of steps over all inputs of size n.
- **Average case** (or expected case) — the mean number of steps over some distribution of inputs of size n.

The worst case is almost always what we care about. The best case is typically meaningless (every algorithm is fast when the answer is at index 0). The average case is useful but requires assuming a probability distribution over inputs, which is hard to justify. The worst case is a *guarantee*: an adversary cannot make the algorithm slower than this.

When average case *does* matter

Two important exceptions: hash tables ($O(1)$ expected, $O(n)$ worst) and randomized quicksort ($O(n \log n)$ expected, $O(n^2)$ worst). When the randomness is inside the *algorithm* (a random pivot, a randomly seeded hash function), not assumed about the input, the expected bound holds for **every** input — nearly as good as a worst-case guarantee. Python randomizes the pivot only if you do, and seeds hashes only for `str` and `bytes`: `hash(5) == 5`, so carefully chosen integer keys can still force collisions.

The Big Oh Notation

Exact step counts are ugly. Selection sort might take $T(n) = \frac{1}{2}n^2 + \frac{3}{2}n - 4$ steps under one reasonable counting and $n^2 - n + 7$ under another. The difference is noise; what matters is that both grow like n^2. Big Oh notation is the tool for throwing away the noise.

Formally, for functions f, g from positive integers to positive reals:

Notation	Meaning	Formal definition
$f(n) = O(g(n))$	g is an **upper bound** on f	$\exists\, c, n_0$ such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$
$f(n) = \Omega(g(n))$	g is a **lower bound** on f	$\exists\, c, n_0$ such that $f(n) \ge c \cdot g(n)$ for all $n \ge n_0$
$f(n) = \Theta(g(n))$	g is a **tight bound** on f	$f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$

Two details in the definition do all the work:

1. **The constant** c lets us ignore multiplicative factors. $1000n$ and $0.001n$ are both $\Theta(n)$.
2. **The threshold** n_0 lets us ignore small inputs. We only care about behavior as n grows large.

!!! info “In interviews, ”O” usually means ”Θ”” Technically $3n = O(n^3)$ is true — n^3 is an upper bound on $3n$, just a very loose one. In everyday usage (and in interviews), when someone says an algorithm “is $O(n)$” they mean the tight bound $\Theta(n)$. Always give the tightest bound you can justify.

Stop and Think: Back to the definition

Problem: Is $2^{n+1} = \Theta(2^n)$? Is $2^{2n} = \Theta(2^n)$?
Solution: For the first, $2^{n+1} = 2 \cdot 2^n$. Pick $c = 2$ and the upper bound holds; pick $c = 1$ and the lower bound holds. So yes, $2^{n+1} = \Theta(2^n)$ — adding a constant to an exponent multiplies by a constant.
For the second, $2^{2n} = 4^n = 2^n \cdot 2^n$. For $2^{2n} \le c \cdot 2^n$ we would need $2^n \le c$ for all large n, which is impossible for any fixed c. So $2^{2n} \neq O(2^n)$. *Multiplying* an exponent changes the growth class. This is why $O(2^n)$ and $O(4^n)$ are genuinely different complexities, whereas $O(\log_2 n)$ and $O(\log_{10} n)$ are not.

Take-Home Lesson

Big Oh notation and worst-case analysis are *simplifications*, and deliberately so. They discard exactly the information that doesn't matter for choosing between algorithms, and keep exactly the information that does.

Growth Rates and Dominance

Why are we allowed to be so cavalier about constants? Because the gap *between* growth classes is so enormous that constants become irrelevant. The table below assumes roughly 10^7 simple operations per second, a realistic figure for pure Python (compiled languages are 10–100× faster, which shifts every row a little but changes none of the conclusions).

n	$\lg n$	n	$n \lg n$	n^2	n^3	2^n	$n!$
10	< 1 µs	1 µs	3 µs	10 µs	100 µs	100 µs	0.36 s
20	< 1 µs	2 µs	9 µs	40 µs	0.8 ms	0.1 s	7,700 years
30	< 1 µs	3 µs	15 µs	90 µs	2.7 ms	1.8 min	—
50	< 1 µs	5 µs	28 µs	0.25 ms	12.5 ms	3.6 years	—
100	< 1 µs	10 µs	66 µs	1 ms	0.1 s	10^{15} years	—
10^3	1 µs	0.1 ms	1 ms	0.1 s	1.7 min	—	—
10^4	1 µs	1 ms	13 ms	10 s	1.2 days	—	—
10^5	2 µs	10 ms	0.17 s	17 min	3 years	—	—
10^6	2 µs	0.1 s	2 s	1.2 days	—	—	—
10^9	3 µs	1.7 min	50 min	3,200 years	—	—	—

Read it row by row and the lessons jump out:

- For $n = 10$, **everything is fast**. Small inputs never distinguish algorithms.
- $n!$ algorithms die at $n \approx 12$; 2^n algorithms at $n \approx 25$–30.
- Quadratic algorithms are fine up to a few thousand, painful at 10^5, hopeless at 10^6.
- $n \log n$ and linear algorithms stay practical on essentially any input that fits in memory.
- A logarithmic algorithm barely notices n at all.

The dominance pecking order

We say g **dominates** f (written $g \gg f$) when $f(n) = O(g(n))$ but not the other way around. The handful of classes that cover nearly every algorithm you will write, from fastest-growing to slowest:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \sqrt{n} \gg \log n \gg 1$$

Filling in the gaps with some classes that show up in more advanced analyses:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg \log^2 n \gg \log n \gg \log \log n \gg \alpha(n) \gg 1$$

where $\alpha(n)$ is the inverse Ackermann function — the "practically constant" factor in Union-Find (see Trie & Union Find).

Class	Typical source
1	Arithmetic, hash lookup, array indexing
$\log n$	Halving the search space: binary search, balanced BST ops, heap push/pop
n	Looking at each element a constant number of times
$n \log n$	Sorting; n operations on a heap or BST; divide-and-conquer with linear merge
n^2	Looking at all **pairs**: nested loops, simple DP on two sequences
n^3	Looking at all **triples**: Floyd–Warshall, interval DP, naive matrix multiplication
2^n	Enumerating all **subsets**
$n!$	Enumerating all **orderings** (permutations)

Counting these objects exactly — subsets of size k, orderings with repeats, trees, partitions — is the subject of Combinatorics.

> **Take-Home Lesson**
>
> A small set of growth classes suffices for almost every algorithm. Learn to recognize them by their *structure* — pairs, triples, subsets, orderings, halving — and you can estimate complexity before writing a line of code.

Working with the Big Oh

The algebra of Big Oh reduces to two rules, plus one warning.

Adding functions: the larger one wins.

$$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

Two consecutive loops over n items is $O(n) + O(n) = O(n)$. A sort followed by a linear scan is $O(n \log n) + O(n) = O(n \log n)$. This is why we "drop non-dominant terms": $n^2 + n \log n + 100n = \Theta(n^2)$.

Multiplying functions: they multiply.

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

A loop running n times whose body costs $O(\log n)$ is $O(n \log n)$. Multiplying by a positive *constant* does nothing: $O(c \cdot f(n)) = O(f(n))$.

Warning: different inputs get different variables.

```
def common_elements(a: list[int], b: list[int]) -> list[int]:
    b_set = set(b)                        # O(len(b))
    return [x for x in a if x in b_set]   # O(len(a))
# Total: O(a + b), NOT O(n)
```

```python
def all_pairs(a: list[int], b: list[int]) -> list[tuple[int, int]]:
    return [(x, y) for x in a for y in b]
# Total: O(a * b), NOT O(n^2)
```

Collapsing a and b into a single n hides real information. If a has 10 elements and b has 10^6, $O(a \cdot b)$ is perfectly fine while "$O(n^2)$" sounds alarming. The same applies to grids ($O(m \cdot n)$ for an $m \times n$ grid), graphs ($O(V + E)$), and strings with an alphabet ($O(n \cdot \Sigma)$).

Reasoning About Efficiency

The fastest way to learn analysis is to do it on real code. Here are four classic examples, each teaching one technique.

Selection sort: counting nested loops exactly

```python
def selection_sort(a: list[int]) -> None:
    n = len(a)
    for i in range(n):
        smallest = i
        for j in range(i + 1, n):
            if a[j] < a[smallest]:
                smallest = j
        a[i], a[smallest] = a[smallest], a[i]
```

The inner loop runs $n - i - 1$ times for each i. Summing:

$$\sum_{i=0}^{n-1}(n - i - 1) = (n-1) + (n-2) + \cdots + 1 + 0 = \frac{n(n-1)}{2} = \Theta(n^2)$$

A faster argument, useful when the exact sum is ugly: to show $\Theta(n^2)$, show both bounds. **Upper:** each loop runs at most n times, so at most $n \cdot n = n^2$ steps. **Lower:** for the first $n/2$ values of i, the inner loop runs at least $n/2$ times, so at least $(n/2)(n/2) = n^2/4$ steps. Done — no summation formula required.

Insertion sort: when best and worst case differ

```python
def insertion_sort(a: list[int]) -> None:
    for i in range(1, len(a)):
        j = i
        while j > 0 and a[j] < a[j - 1]:
            a[j], a[j - 1] = a[j - 1], a[j]
            j -= 1
```

The `while` loop stops as soon as the element is in place. On a sorted array it never iterates, so the total is $\Theta(n)$. On a reversed array it goes all the way to the front every time: $\Theta(n^2)$. The worst case is $\Theta(n^2)$, and that is what we report — but the gap explains why insertion sort is used inside Timsort for small or nearly-sorted runs.

Counting what actually happens

For `while` loops, the number of iterations is not written in the code. You have to argue about it: *what quantity changes each iteration, and how far can it go?* Here, `j` decreases by one each step and cannot go below zero, so the inner loop runs at most i times.

String pattern matching: two variables, one loop nest

```python
def find_substring(text: str, pattern: str) -> int:
    n, m = len(text), len(pattern)
    for i in range(n - m + 1):
        j = 0
        while j < m and text[i + j] == pattern[j]:
            j += 1
        if j == m:
            return i
    return -1
```

The outer loop runs $n-m+1$ times and the inner loop at most m times, so the worst case is $O((n-m+1)\cdot m) \leq O(nm)$. Is that bound tight? Try `text = "aaaa...a"` and `pattern = "aa...ab"`: every alignment matches $m-1$ characters before failing, so yes. Cleverer algorithms (KMP, Z-function, Rabin–Karp) bring this down to $O(n+m)$.

Matrix multiplication: three nested loops

```python
def matmul(A: list[list[float]], B: list[list[float]]) -> list[list[float]]:
    x, y, z = len(A), len(B), len(B[0])      # A is x*y, B is y*z
    C = [[0.0] * z for _ in range(x)]
    for i in range(x):
        for j in range(z):
            for k in range(y):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

Three independent loops multiply: $\Theta(xyz)$, or $\Theta(n^3)$ for square matrices.

Summations

Loops in code become summations in analysis. Two families of sums cover almost everything.

Sums of powers: $\sum i^p = \Theta(n^{p+1})$

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2) \qquad \sum_{i=1}^{n} i^2 = \Theta(n^3)$$

The big-picture fact: summing a polynomial of degree p gives a polynomial of degree $p+1$. The constant ($\frac{1}{2}$, $\frac{1}{3}$, …) rarely matters.

Geometric series: the "free lunch" of analysis

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

- If $a > 1$ the sum is dominated by its **largest term**: $1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2 \cdot 2^k$.
- If $a < 1$ the sum is bounded by a **constant**: $1 + \frac{1}{2} + \frac{1}{4} + \cdots < 2$, no matter how many terms.

This is one of the most powerful tools in analysis: *a sum of a logarithmic number of geometrically shrinking terms costs only as much as the largest term*. It is the reason why:

- Appending to a dynamic array is $O(1)$ amortized (the resize costs $1 + 2 + 4 + \cdots + n < 2n$).
- Building a heap bottom-up (`heapq.heapify`) is $O(n)$, not $O(n \log n)$.
- A recursion that does $O(n)$ work and then recurses on *one* half (quickselect, on average) is $O(n)$ total: $n + n/2 + n/4 + \cdots < 2n$.

The harmonic series: where the log comes from

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \Theta(\log n)$$

When a $\log n$ appears unexpectedly in an analysis, the harmonic series is usually why. A classic example:

```python
def count_divisor_pairs(n: int) -> int:
    count = 0
    for i in range(1, n + 1):
        for j in range(i, n + 1, i):   # multiples of i: n / i of them
            count += 1
    return count
```

This looks like it might be quadratic, but the inner loop runs n/i times, so the total is $\sum_{i=1}^{n} n/i = n \cdot H(n) = \Theta(n \log n)$. The same shape appears in the Sieve of Eratosthenes (which is even better, $\Theta(n \log \log n)$, because it only iterates over primes i).

Logarithms

A logarithm is an inverse exponential: $b^x = y \iff x = \log_b y$. Exponentials grow terrifyingly fast; logarithms grow refreshingly slowly. $\log_2(10^9) \approx 30$.

> **Take-Home Lesson**
>
> Logarithms arise whenever things are repeatedly **halved** or **doubled**. If you see a quantity multiplied or divided by a constant at each step, expect a log.

Binary search. Each comparison discards half of the remaining candidates. The number of steps is the number of times you can halve n before reaching 1, which is exactly $\log_2 n$. Twenty comparisons find any item among a million.

Trees. A binary tree of height h has at most 2^h leaves. So a tree holding n leaves must have height at least $\log_2 n$ — and a *balanced* tree achieves it. Short trees can hold an enormous number of items, which is why trees underlie so many fast data structures. A d-ary tree has height $\log_d n$.

Bits. There are 2^w bit strings of length w. To give n items distinct labels you need $w = \lceil \log_2 n \rceil$ bits. This is why an integer n has $\Theta(\log n)$ digits, and why iterating over the bits of n costs $O(\log n)$.

Fast exponentiation. Computing a^n by $n-1$ multiplications is wasteful. Since $a^n = (a^{n/2})^2$ for even n and $a \cdot (a^{\lfloor n/2 \rfloor})^2$ for odd n, we halve the exponent with at most two multiplications:

```python
def fast_pow(a: int, n: int, mod: int | None = None) -> int:
    """Compute a**n (optionally modulo mod) in O(log n) multiplications."""
    result = 1
    while n > 0:
        if n & 1:                # current bit of n is set
            result *= a
            if mod:
                result %= mod
        a *= a                   # a, a^2, a^4, a^8, ...
        if mod:
            a %= mod
        n >>= 1
    return result
```

(Python's built-in `pow(a, n, mod)` does exactly this.)

Properties worth memorizing

Identity	Consequence
$\log_a(xy) = \log_a x + \log_a y$	Log of a product is a sum of logs
$\log_a n^b = b \log_a n$	$\log(n^2) = 2\log n = \Theta(\log n)$
$\log_a n = \dfrac{\log_b n}{\log_b a}$	**The base does not matter**: changing it is a constant factor
$\log(n!) = \Theta(n \log n)$	Why comparison sorting needs $\Omega(n \log n)$ comparisons

The base-change rule is why we write $O(\log n)$ without a base. But note that the base *inside* an exponent matters: $2^{\log_2 n} = n$ whereas $2^{\log_4 n} = \sqrt{n}$.

Recurrences and Divide and Conquer

Recursive algorithms are analyzed with **recurrence relations**, which express the cost on an input of size n in terms of the cost on smaller inputs.

Recurrence	Solution	Example
$T(n) = T(n-1) + O(1)$	$O(n)$	Recursive linear scan, factorial
$T(n) = T(n-1) + O(n)$	$O(n^2)$	Selection sort as recursion; quicksort worst case
$T(n) = T(n/2) + O(1)$	$O(\log n)$	Binary search
$T(n) = T(n/2) + O(n)$	$O(n)$	Quickselect (expected) — geometric series
$T(n) = 2T(n/2) + O(1)$	$O(n)$	Tree traversal
$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$	Merge sort

Recurrence	Solution	Example
$T(n) = 2T(n-1) + O(1)$	$O(2^n)$	Naive Fibonacci (roughly), subset enumeration

The master theorem

For recurrences of the form $T(n) = a\,T(n/b) + f(n)$ — split into a subproblems of size n/b, then combine in $f(n)$ time — picture the **recursion tree**. Level i has a^i nodes, each of size n/b^i. The tree has $\log_b n$ levels and $a^{\log_b n} = n^{\log_b a}$ leaves. The answer depends on where the work concentrates:

Case	Condition	Result	Intuition
1	$f(n) = O(n^{\log_b a - \epsilon})$	$\Theta(n^{\log_b a})$	**Leaves dominate**: too many subproblems
2	$f(n) = \Theta(n^{\log_b a})$	$\Theta(n^{\log_b a} \log n)$	**Every level costs the same**: multiply by the depth
3	$f(n) = \Omega(n^{\log_b a + \epsilon})$	$\Theta(f(n))$	**Root dominates**: the combine step is the bottleneck

Algorithm	a, b, $f(n)$	$n^{\log_b a}$	Case	Result
Binary search	1, 2, 1	1	2	$\Theta(\log n)$
Merge sort	2, 2, n	n	2	$\Theta(n \log n)$
Tree traversal	2, 2, 1	n	1	$\Theta(n)$
Karatsuba multiplication	3, 2, n	$n^{1.585}$	1	$\Theta(n^{1.585})$
Strassen	7, 2, n^2	$n^{2.807}$	1	$\Theta(n^{2.807})$

Divide as evenly as possible

Fast exponentiation and merge sort both work because they split the problem into halves. A split of $n-1$ and 1 (as in quicksort with a bad pivot) turns $T(n) = 2T(n/2)+n$ into $T(n) = T(n-1)+n = \Theta(n^2)$. Balance is what makes divide and conquer fast.

Amortized Analysis

Sometimes a single operation can be expensive, but expensive operations are *rare enough* that the average over any sequence is cheap. The **amortized cost** is the total cost of a sequence of operations divided by its length, in the worst case over all sequences. Unlike average-case analysis, no probability is involved — it is a worst-case guarantee on the *total*.

Dynamic arrays

A Python `list` stores its elements in a contiguous block with some spare capacity. When `append` finds the block full, it allocates a larger block and copies everything over — an $O(n)$ operation. Is `append` therefore $O(n)$?

Suppose the capacity doubles on each resize. Over n appends, resizes happen at sizes $1, 2, 4, \ldots, n$, copying

$$1 + 2 + 4 + \cdots + n < 2n$$

elements in total — a geometric series. Adding the n ordinary writes, n appends cost less than $3n$ steps: $O(1)$ **amortized** per append. (CPython grows by roughly 12.5% rather than 100%, but any constant growth factor > 1 gives a geometric series and the same conclusion. Growing by a constant *amount* would not: $k + 2k + 3k + \cdots$ is quadratic.)

Aggregate counting: "each element is touched a constant number of times"

The most useful amortized argument in practice is simple bookkeeping: instead of bounding each loop iteration, bound how many times **each element** can be processed over the entire run.

```python
def next_greater(nums: list[int]) -> list[int]:
    result = [-1] * len(nums)
    stack: list[int] = []                  # indices, decreasing values
    for i, x in enumerate(nums):
        while stack and nums[stack[-1]] < x:
            result[stack.pop()] = x
        stack.append(i)
    return result
```

A `while` inside a `for` looks quadratic. But every index is pushed exactly once and popped at most once, so across the *whole* run the `while` loop body executes at most n times. Total: $O(n)$. The same argument proves that sliding-window two-pointer algorithms are linear: `left` and `right` each move forward at most n times.

> **Take-Home Lesson**
>
> When a nested loop looks quadratic, ask: *can the inner loop really run n times on every outer iteration, or is it consuming a shared budget?* If each element can be pushed, popped, or passed over only once, the total is linear.

Space Complexity

Space is analyzed with the same tools. Two conventions to be explicit about:

- **Auxiliary space** is the extra memory beyond the input. When people say "$O(1)$ space," they mean auxiliary space.
- **Output space** is usually not counted — returning a list of n answers doesn't make an algorithm "use $O(n)$ space" in the usual sense.

The call stack counts

Each active recursive call holds a stack frame. Recursion depth d costs $O(d)$ space even if the function allocates nothing.

Recursion	Depth	Space
Binary search (recursive)	$\log n$	$O(\log n)$
DFS on a balanced tree	$\log n$	$O(\log n)$
DFS on a skewed tree / linked list	n	$O(n)$

Recursion	Depth	Space
DFS on a graph	up to V	$O(V)$
Quicksort (recurse on the smaller side, loop on the larger)	$\log n$	$O(\log n)$

Python's recursion limit

CPython's default recursion limit is 1000. A recursive DFS on a path graph or a degenerate tree of 10,000 nodes raises `RecursionError`. Either raise the limit with `sys.setrecursionlimit`, or convert the recursion into an explicit stack — which is always possible and uses the same asymptotic space.

Estimation: From Constraints to Complexity

On LeetCode, the constraints *tell you* the intended complexity. Budget roughly 10^7–10^8 simple operations for Python in a one- or two-second time limit, then read the growth-rate table backwards: find the fastest-growing class that still fits the budget at the given n.

- $n \leq 20$ admits $2^n \approx 10^6$, so exponential search over subsets is intended.
- $n \leq 5{,}000$ admits $n^2 = 2.5 \times 10^7$, but not n^3.
- $n = 10^5$ rules out $n^2 = 10^{10}$ and leaves $n \log n \approx 1.7 \times 10^6$.

The full lookup, with the techniques that typically hit each bound, is the constraints table in the cheat sheet.

Estimation is also a sanity check for your own design. If you have derived an $O(n^2)$ algorithm and $n = 10^5$, stop: you need a better idea, not a better implementation.

Take-Home Lesson

Programs working on large inputs must run in linear or near-linear time. Once you commit to that requirement, the constraint itself usually points toward the right technique.

Common Mistakes

1. **Confusing additive and multiplicative progress.**

```
i = 1
while i < n:
    i += 2      # O(n): additive steps
i = 1
while i < n:
    i *= 2      # O(log n): multiplicative steps
```

2. **Hidden linear work inside a loop.** `pop(0)`, `x in list`, slicing (including `solve(arr[1:])` in recursion), and string concatenation inside a loop turn $O(n)$ into $O(n^2)$. See the performance traps for the fixes.
3. **Forgetting the call stack.** A recursive solution with no explicit data structures still uses $O(\text{depth})$ space.
4. **Assuming nested loops mean $O(n^2)$.** The inner loop may share a budget with the outer one (amortized

$O(n)$) or shrink geometrically (harmonic, $O(n \log n)$). Count what actually happens.

5. **Reporting $O(n)$ for a two-input problem.** Use $O(m + n)$, $O(V + E)$, $O(m \cdot n)$ — every independent size gets its own variable.
6. **Treating hash operations as always $O(1)$.** They are $O(1)$ *expected*. Also, hashing a key costs time proportional to the key: hashing a string of length k is $O(k)$, and using a `tuple` of length k as a dict key is $O(k)$ per lookup.

> **Source**
>
> The Common Patterns chapters are based on Steven S. Skiena, *The Algorithm Design Manual*, 3rd ed. (Springer, 2020). See References.

Combinatorics

Counting is the quiet partner of algorithm design. You count to know whether exhaustive search is feasible (2^{20} subsets: yes; 20! permutations: no). You count to analyze algorithms (how many nodes does this search tree have?). Many problems simply *are* counting problems — "how many ways…" — usually solved with dynamic programming. And recognizing which combinatorial object a problem is really about is often the key step toward a solution.

This chapter collects the counting tools that come up again and again: the basic rules, the standard formulas and where they come from, the important number families, how to generate and index combinatorial objects, and the probability needed to reason about randomized algorithms.

Modeling: Which Object Is This?

Most algorithmic problems are secretly about a handful of fundamental structures. The words in the problem statement are often a strong hint:

Object	Keywords that suggest it	Typical count
Permutation	arrangement, ordering, tour, sequence, schedule, ranking	$n!$
Subset	selection, group, committee, cluster, collection, packing	2^n, or $\binom{n}{k}$
Tree	hierarchy, ancestor, dominance, taxonomy, nesting	Catalan numbers (binary trees)
Graph	network, relationship, connection, circuit, web	$2^{\binom{n}{2}}$ labeled graphs
String	text, pattern, label, sequence of symbols	$\lvert\Sigma\rvert^n$
Partition	split into groups, clustering, coloring	Stirling / Bell numbers

Take-Home Lesson

Modeling your problem in terms of well-defined structures — permutations, subsets, trees, graphs, strings — is the most important single step toward a solution. Once you know which object you're dealing with, you know how to count it, how to enumerate it, and which algorithms already exist for it.

The Basic Counting Rules

Almost every counting argument is built from a few rules.

Rule	Statement	Example
Sum rule	If choices come from disjoint cases, add the counts	A password is 4 or 5 digits: $10^4 + 10^5$
Product rule	If a choice is a sequence of independent steps, multiply	A 3-letter code: 26^3
Complement	Count what you *don't* want, subtract from the total	Strings of length n with at least one a: $26^n - 25^n$
Division rule	If every outcome is counted exactly k times, divide by k	Seatings of n people at a round table: $n!/n = (n-1)!$
Bijection	If two sets can be matched one-to-one, they have the same size	Subsets of n items $\leftrightarrow$ n-bit strings: 2^n
Inclusion–exclusion	$\lvert A \cup B \rvert = \lvert A \rvert + \lvert B \rvert - \lvert A \cap B \rvert$	Numbers ≤ 100 divisible by 2 or 3: $50 + 33 - 16 = 67$

The **bijection** rule deserves special mention: much of combinatorial cleverness is finding a way to see an unfamiliar set as a familiar one in disguise. Stars and bars, below, is a bijection. So is the observation that subsets are binary strings (see Bit Manipulation).

Permutations and Combinations

Choose r from n…	Order matters	Order doesn't matter
without repetition	$P(n,r) = \dfrac{n!}{(n-r)!}$	$\dbinom{n}{r} = \dfrac{n!}{r!\,(n-r)!}$
with repetition	n^r	$\dbinom{n+r-1}{r}$

Permutations without repetition. By the product rule: n choices for the first position, $n-1$ for the second, …, $n-r+1$ for the last. Gold, silver, and bronze among 8 athletes: $8 \cdot 7 \cdot 6 = 336$.

Combinations. Each unordered selection of r items appears $r!$ times among the ordered ones, so divide (the division rule): $\binom{n}{r} = P(n,r)/r!$. A 3-person committee from 10 people: $\binom{10}{3} = 120$.

With repetition, ordered. r independent choices of n options each: n^r. A 4-digit PIN: 10^4.

Multiset permutations. Arrangements of n items with n_1 identical copies of one kind, n_2 of another, …: $\dfrac{n!}{n_1!\,n_2!\cdots n_k!}$. Swapping identical items doesn't create a new arrangement, so divide out those swaps. The letters of MISSISSIPPI (1 M, 4 I, 4 S, 2 P) have $\dfrac{11!}{1!\,4!\,4!\,2!} = 34{,}650$ arrangements.

Stars and bars: combinations with repetition

How many ways to buy 5 donuts from 3 flavors? Write any purchase as a row of 5 stars (donuts) and 2 bars (dividers between flavors):

```
★★|★|★★     =  2 of flavor A, 1 of flavor B, 2 of flavor C
★★★★★||     =  5 of flavor A
```

Every purchase is exactly one arrangement of 5 stars and 2 bars, and vice versa — a bijection. So the answer is the number of ways to choose which 2 of the 7 positions hold bars: $\binom{7}{2} = 21$. In general, r items from n types: $\binom{n+r-1}{n-1} = \binom{n+r-1}{r}$.

The same argument counts the solutions of $x_1 + x_2 + \cdots + x_n = r$ in non-negative integers, which is how it appears in practice.

Binomial Coefficients

$\binom{n}{k}$ is the most important family of counting numbers. Computing it from factorials is fine in Python (arbitrary precision), but in fixed-width languages the intermediate factorials overflow long before the answer does. The robust way is **Pascal's triangle**:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \qquad \binom{n}{0} = \binom{n}{n} = 1$$

Why it's true (a combinatorial proof, which is often clearer than algebra): fix one particular item. Every k-subset either **contains** it — then the other $k-1$ items come from the remaining $n-1$ — or it **doesn't** — then all k come from the remaining $n-1$. The cases don't overlap and cover everything, so add them. This is exactly the take-or-skip recurrence of dynamic programming, and Pascal's triangle is its table.

```python
def pascal(n: int) -> list[list[int]]:
    rows = [[1]]
    for _ in range(n):
        prev = rows[-1]
        rows.append([1] + [prev[i] + prev[i + 1] for i in range(len(prev) - 1)] + [1])
    return rows
```

In Python, `math.comb(n, k)` computes a single coefficient exactly and quickly.

Lattice paths. The number of monotone paths through an $m \times n$ grid (LC 62, Unique Paths) is $\binom{m+n-2}{m-1}$: a path is a sequence of $m-1$ downs and $n-1$ rights, and choosing the positions of the downs determines it. The DP and the formula compute the same Pascal's triangle.

Useful identities

Identity	Formula	Why
Symmetry	$\binom{n}{k} = \binom{n}{n-k}$	Choosing what to include = choosing what to exclude
Row sum	$\sum_k \binom{n}{k} = 2^n$	Counting all subsets by size
Vandermonde	$\binom{m+n}{r} = \sum_k \binom{m}{k}\binom{n}{r-k}$	Choose r from two groups: k from the first, the rest from the second
Hockey stick	$\sum_{i=r}^{n} \binom{i}{r} = \binom{n+1}{r+1}$	Classify $(r+1)$-subsets of $\{0..n\}$ by their largest element
Binomial theorem	$(x+y)^n = \sum_k \binom{n}{k} x^k y^{n-k}$	Choose which factors contribute an x

Counting modulo a prime

When answers are huge, problems ask for them modulo $p = 10^9 + 7$. Precompute factorials and their inverses once; then each $\binom{n}{k}$ costs $O(1)$. Since p is prime, Fermat's little theorem gives the inverse: $a^{-1} \equiv a^{p-2} \pmod{p}$.

```
MOD = 10**9 + 7


def binomial_table(limit: int):
    fact = [1] * (limit + 1)
    for i in range(1, limit + 1):
        fact[i] = fact[i - 1] * i % MOD
    inv_fact = [1] * (limit + 1)
    inv_fact[limit] = pow(fact[limit], MOD - 2, MOD)     # Fermat inverse, O(log MOD)
    for i in range(limit, 0, -1):
        inv_fact[i - 1] = inv_fact[i] * i % MOD

    def comb_mod(n: int, k: int) -> int:
        if k < 0 or k > n:
            return 0
        return fact[n] * inv_fact[k] % MOD * inv_fact[n - k] % MOD

    return comb_mod
```

Important Number Families

Catalan numbers

$$C_0 = 1, \qquad C_n = \sum_{i=0}^{n-1} C_i \, C_{n-1-i}, \qquad C_n = \frac{1}{n+1}\binom{2n}{n}$$

Values: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …

Catalan numbers count structures that **split into two independent smaller structures of the same kind**. For balanced parentheses: a string of n pairs starts with (and its matching) encloses a balanced string of i pairs, followed by a balanced string of the remaining $n - 1 - i$. Sum over i — that's the recurrence. The same decomposition, "root plus left part plus right part," appears in all of these:

Structure of size n	Counted by C_n
Balanced strings of n pairs of parentheses (LC 22)	C_n
Structurally distinct BSTs with n keys (LC 96)	C_n
Full binary trees with $n + 1$ leaves	C_n
Ways to parenthesize a product of $n + 1$ factors	C_n
Triangulations of a convex polygon with $n + 2$ sides	C_n
Monotone lattice paths that never cross the diagonal	C_n

```
def num_trees(n: int) -> int:
    """LC 96: C_n, via the root-splits-into-two recurrence."""
```

```
    catalan = [1] + [0] * n
    for m in range(1, n + 1):
        catalan[m] = sum(catalan[i] * catalan[m - 1 - i] for i in range(m))
    return catalan[n]
```

The closed form comes from the **reflection principle**: of the $\binom{2n}{n}$ lattice paths from $(0,0)$ to (n,n), the ones that cross the diagonal can be put in bijection with all paths to $(n-1, n+1)$ — reflect each bad path after its first crossing — so the good ones number $\binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n}$.

More families worth recognizing

Family	Counts	Recurrence
Fibonacci F_n	Tilings of a $1 \times n$ strip with 1×1 and 1×2 tiles; ways to climb n stairs by 1s and 2s	$F_n = F_{n-1} + F_{n-2}$
Derangements D_n	Permutations with no fixed point	$D_n = (n-1)(D_{n-1} + D_{n-2})$, $D_n \approx n!/e$
Stirling numbers (2nd kind) $S(n,k)$	Partitions of n labeled items into k non-empty groups	$S(n,k) = k\,S(n-1,k) + S(n-1,k-1)$
Bell numbers B_n	All partitions of n labeled items	$B_n = \sum_k S(n,k)$
Integer partitions $p(n)$	Ways to write n as an unordered sum of positive integers	Coin-change counting with coins $1 \dots n$

Each recurrence comes from asking **what happens to the last item**. For Stirling numbers: item n either joins one of the k groups formed by the others ($k \cdot S(n-1,k)$) or sits alone in a new group ($S(n-1,k-1)$). For derangements: item n goes to some position i ($n-1$ choices); then either item i goes to position n (swap, leaving D_{n-2}) or it doesn't (effectively a derangement of $n-1$). It's the same reasoning as a DP recurrence, because it *is* one.

Generating and Indexing Combinatorial Objects

Ranking and unranking

To enumerate, sample, or index a family of objects, it helps to put them in a fixed order and define two inverse functions:

- **rank**(object): its position in the order, from 0 to $N-1$.
- **unrank**(position): the object at that position.

With these, "the next object" is `unrank(rank(x) + 1)`, "a uniformly random object" is `unrank(random integer)`, and a set of objects can be stored as a bit vector over ranks.

For permutations in lexicographic order, the first element splits the $n!$ permutations into n blocks of $(n-1)!$ each, so

$$\text{rank}(p) = (\text{index of } p_1 \text{ among remaining items}) \cdot (n-1)! + \text{rank}(p_2, \dots, p_n)$$

Unranking reverses this: divide by $(n-1)!$ to find the first element, and recurse on the remainder. This is exactly **Permutation Sequence** (LC 60), the k-th permutation:

```python
from math import factorial


def get_permutation(n: int, k: int) -> str:
    """LC 60: the k-th (1-indexed) permutation of 1..n in lexicographic order."""
    items = [str(i) for i in range(1, n + 1)]
    k -= 1                                          # to a 0-indexed rank
    out = []
    for size in range(n, 0, -1):
        block = factorial(size - 1)                 # permutations per first choice
        i, k = divmod(k, block)
        out.append(items.pop(i))
    return "".join(out)


def rank_permutation(p: list[int]) -> int:
    items = sorted(p)
    rank = 0
    for i, x in enumerate(p):
        idx = items.index(x)
        rank += idx * factorial(len(p) - 1 - i)
        items.pop(idx)
    return rank
```

(The digits i chosen along the way are the permutation's **factorial number system** representation.)

Next permutation (LC 31)

To step to the next permutation in lexicographic order without ranking: find the longest non-increasing suffix — it's already the *last* arrangement of those elements. The element just before it, the **pivot**, must increase by as little as possible: swap it with the smallest suffix element larger than it, then reverse the suffix to make it the *first* arrangement.

```python
def next_permutation(nums: list[int]) -> None:
    i = len(nums) - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1                                      # nums[i+1:] is non-increasing
    if i >= 0:
        j = len(nums) - 1
        while nums[j] <= nums[i]:
            j -= 1                                  # smallest element > pivot in the suffix
        nums[i], nums[j] = nums[j], nums[i]
    nums[i + 1:] = reversed(nums[i + 1:])           # suffix back to ascending order
```

Amortized $O(1)$ per step, and it handles duplicates correctly. This is how C++'s `std::next_permutation` works. Python's `itertools.permutations`, by contrast, permutes *positions*, so it repeats arrangements when the input contains duplicates.

Random permutations: the Fisher–Yates shuffle

Generating a uniformly random permutation is a small problem that people often get wrong. The correct algorithm (LC 384) is two lines: for each position i from left to right, swap in a random element from positions i **through the end**.

```python
import random


def shuffle(a: list) -> None:
    for i in range(len(a) - 1):
        j = random.randint(i, len(a) - 1)           # i..n-1, NOT 0..n-1
        a[i], a[j] = a[j], a[i]
```

Why uniform? Position 0 receives each element with probability $1/n$; given that, position 1 receives each remaining element with probability $1/(n-1)$; and so on. Every permutation arises with probability $\frac{1}{n} \cdot \frac{1}{n-1} \cdots = \frac{1}{n!}$.

Stop and Think: Why is the naive shuffle biased?

Problem: What's wrong with swapping each position with a random position from the **whole** array, `j = random.randint(0, n - 1)`?
Solution: It makes n independent choices of n options, so it has n^n equally likely execution paths. Each path produces one permutation, so a permutation's probability is (number of paths producing it)$/n^n$. For the result to be uniform, each of the $n!$ permutations would need exactly $n^n/n!$ paths — but $n!$ doesn't divide n^n for $n \geq 3$: the factor $n-1$ of $n!$ shares no prime factor with n^n (for $n = 3$: 6 doesn't divide 27). So some permutations must be more likely than others, no matter how good the random number generator is.

The Pigeonhole Principle

> *If n items are placed into $m < n$ boxes, some box contains at least two items. More generally, some box contains at least $\lceil n/m \rceil$ items.*

It sounds too obvious to be useful, but it proves existence without construction:

- **Find the Duplicate Number** (LC 287): $n+1$ values from $\{1, \ldots, n\}$ must include a repeat. Treating the array as a function $i \mapsto \text{nums}[i]$, that repeat is the entrance to a cycle, found by Floyd's algorithm.
- **Maximum Gap** (LC 164): with n numbers spanning a range R, some gap is at least $R/(n-1)$, so the answer never lies inside a bucket of that width (see Sorting).
- **Hashing:** with more keys than buckets, collisions are guaranteed; the question is only how many.
- **Prefix sums mod k:** among any $k+1$ prefix sums, two agree modulo k, so some non-empty subarray has a sum divisible by k.

Probability Essentials

Randomized algorithms (quicksort, hashing, sampling) are analyzed with a small amount of probability.

- A **sample space** is the set of possible outcomes; an **event** is a subset of it. With equally likely outcomes, $P(E) = |E|/|S|$ — probability is counting.
- **Complement:** $P(E) = 1 - P(\bar{E})$, often the easier side to compute.
- **Union:** $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.
- **Independence:** A and B are independent if $P(A \cap B) = P(A)\,P(B)$. Independent events let you multiply.
- **Conditional probability:** $P(A \mid B) = P(A \cap B)/P(B)$, and **Bayes' theorem** reverses it: $P(B \mid A) = P(A \mid$

$B)\ P(B)/P(A)$.

The birthday paradox. With n people and 365 days, the probability that all birthdays differ is $\prod_{i=0}^{n-1}(1 - i/365)$, which drops below $1/2$ at just $n = 23$. The same computation says that hashing about $\sqrt{m}$ keys into m buckets already makes a collision likely — which is why hash tables must handle collisions rather than hope to avoid them.

Linearity of expectation

The **expected value** of a random quantity is its probability-weighted average. The single most useful fact about it:

$$E[X + Y] = E[X] + E[Y] \qquad \text{always — even when } X \text{ and } Y \text{ are dependent.}$$

This lets you compute complicated expectations by breaking a quantity into simple **indicator** pieces (1 if something happens, 0 if not), whose expectations are just probabilities:

- **Fixed points of a random permutation:** position i keeps its element with probability $1/n$. Summing over the n positions, the expected number of fixed points is exactly 1 — for every n.
- **Coupon collector:** collecting all n coupons, when each draw is uniformly random, takes $n\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) = nH_n \approx n \ln n$ draws on average. After i distinct coupons, each draw is new with probability $(n - i)/n$, so that phase takes $n/(n - i)$ draws on average; add the phases.
- **Randomized quicksort:** elements i and j (in sorted order) are compared iff one of them is the first pivot chosen among $i, \ldots, j$, which has probability $2/(j - i + 1)$. Summing over all pairs gives about $2n \ln n$ comparisons.

Sampling from a stream: reservoir sampling

To pick a uniformly random element from a stream of unknown length in one pass with $O(1)$ memory (LC 382, LC 398): keep the i-th element with probability $1/i$, replacing the current choice.

```
def reservoir_sample(stream) -> object:
    chosen = None
    for i, x in enumerate(stream, start=1):
        if random.randrange(i) == 0:                  # probability 1/i
            chosen = x
    return chosen
```

Why uniform? Element i is chosen at step i with probability $1/i$, then survives each later step j with probability $(j - 1)/j$. The product telescopes: $\frac{1}{i} \cdot \frac{i}{i+1} \cdot \frac{i+1}{i+2} \cdots \frac{n-1}{n} = \frac{1}{n}$.

Weighted sampling (LC 528): pick index i with probability proportional to w_i. Build prefix sums of the weights, draw a uniform number in $[0, \text{total})$, and binary search for the first prefix sum exceeding it.

```
import bisect
import itertools


class WeightedPicker:
    def __init__(self, weights: list[int]) -> None:
        self.prefix = list(itertools.accumulate(weights))

    def pick_index(self) -> int:
        target = random.random() * self.prefix[-1]
        return bisect.bisect_right(self.prefix, target)
```

From Counts to Code

The counts above are exactly the sizes of the search spaces in Combinatorial Search, which is why they give the running times of the corresponding algorithms:

Object	Count	Generate with	Time to generate all
All subsets	2^n	Choose-next backtracking, or bitmasks	$O(n \cdot 2^n)$
k-subsets	$\binom{n}{k}$	Choose-next backtracking with a size limit	$O(k\binom{n}{k})$
Permutations	$n!$	Backtracking with `used` flags, or next-permutation	$O(n \cdot n!)$
Multiset permutations	$\frac{n!}{n_1!\cdots n_k!}$	Sort + skip equal siblings	$O(n \cdot \text{count})$
Balanced parentheses	$C_n \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$	Backtracking with open/close counts	$O(n \cdot C_n)$
Solutions of $x_1 + \cdots + x_n = r$	$\binom{n+r-1}{r}$	Unbounded-knapsack-style recursion	$O(n \cdot \text{count})$

When a problem only asks *how many*, don't generate — count with a formula or a DP.

Common Mistakes

1. **Confusing ordered and unordered counts.** "How many ways to pick 3 people" ($\binom{n}{3}$) vs. "to pick a president, VP, and treasurer" ($P(n, 3)$). Ask whether swapping two choices produces a different outcome.
2. **Double counting.** Counting "at least one" as "pick one, then anything" overcounts outcomes with several. Use the complement, or inclusion–exclusion.
3. **Forgetting the modulus at every step.** Reduce after each multiplication and addition, not only at the end. And never *divide* modulo p — multiply by the modular inverse.
4. **Overflowing factorials.** Irrelevant in Python, fatal in Java and C++. Use Pascal's triangle or modular arithmetic.
5. **The naive shuffle.** Swapping with a random position from the whole array is biased. Use Fisher–Yates: random position from i to the end.
6. **Assuming independence.** Multiplying probabilities is valid only for independent events. Linearity of expectation, by contrast, never needs independence.

Practice Problems

Problem	Idea
Unique Paths (LC 62)	$\binom{m+n-2}{m-1}$ or Pascal DP
Pascal's Triangle (LC 118)	Pascal's rule
Unique Binary Search Trees (LC 96)	Catalan numbers
Generate Parentheses (LC 22)	Enumerate Catalan structures
Permutation Sequence (LC 60)	Unranking via factorials
Next Permutation (LC 31)	Pivot, swap, reverse suffix
Shuffle an Array (LC 384)	Fisher–Yates
Linked List Random Node (LC 382)	Reservoir sampling
Random Pick with Weight (LC 528)	Prefix sums + binary search
Find the Duplicate Number (LC 287)	Pigeonhole + cycle detection
Count Sorted Vowel Strings (LC 1641)	Stars and bars: $\binom{n+4}{4}$
Count Vowels Permutation (LC 1220)	Counting DP
Number of Music Playlists (LC 920)	Counting DP with inclusion–exclusion flavor

Binary Search

Binary search is the mother of all divide-and-conquer algorithms. Its power is easy to underestimate: it finds any item among a billion in about thirty comparisons. But the real reason it deserves a chapter is that "search a sorted array" is only its most obvious application. Binary search works on *anything* with a monotone yes/no structure — the boundary inside a sorted array, the minimum feasible capacity of a ship, the square root of a number, the first bad commit in a history.

This chapter develops one template, proves why it is correct, and then shows that almost every binary search problem is that template with a different predicate plugged in.

Twenty Questions

In the game of twenty questions, one player thinks of a word and the other asks yes/no questions until they guess it. The second player has a guaranteed winning strategy: open a dictionary to the middle, and ask "does your word come before *this* one alphabetically?" Every answer eliminates half of the remaining words. A dictionary has at most a few hundred thousand entries, and $2^{20} \approx 10^6$, so twenty questions always suffice.

That is binary search: each comparison with the middle element halves the candidate set, so n candidates are reduced to one in $\lceil \log_2 n \rceil$ steps.

n	Linear scan (worst case)	Binary search (worst case)
10^3	1,000	10
10^6	1,000,000	20
10^9	1,000,000,000	30

The price is a precondition: the data must be *ordered* so that a single comparison tells you which half to discard. Paying $O(n \log n)$ once to sort, then $O(\log n)$ per query, is one of the most common trades in algorithm design.

The Classic Algorithm

```python
def binary_search(nums: list[int], target: int) -> int:
    """Return an index i with nums[i] == target, or -1 if absent."""
    lo, hi = 0, len(nums) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if nums[mid] == target:
            return mid
        if nums[mid] < target:
            lo = mid + 1           # target can only be in nums[mid+1 .. hi]
        else:
```

```
            hi = mid - 1           # target can only be in nums[lo .. mid-1]
    return -1
```

Why it is correct. The loop maintains an *invariant*: if `target` is anywhere in `nums`, it is in `nums[lo..hi]`. Initially this is the whole array, so the invariant holds. Each branch discards only elements that provably cannot equal `target` (they are all strictly smaller or strictly larger, because the array is sorted), so the invariant is preserved. The window shrinks by at least one each iteration, so the loop terminates; when it does with `lo > hi`, the window is empty, and by the invariant `target` is not present.

Time: $O(\log n)$. **Space:** $O(1)$ iteratively, $O(\log n)$ if written recursively.

> **Take-Home Lesson**
>
> Every correct binary search rests on an invariant about what lies inside, left of, and right of the current window. If you can't state the invariant, you can't trust the off-by-ones. Write it down as a comment before writing the loop.

The Real Pattern: Finding a Boundary

The classic algorithm answers "is `target` here?" But most problems ask a different question: *where does something start?* The first element $\geq x$, the first day a condition holds, the smallest capacity that works.

All of these are the same problem. Suppose we have a **predicate** $P(i)$ over indices lo … hi that is **monotone**: once it becomes true, it stays true.

```
index:  0     1     2     3     4     5     6     7
P(i):   F     F     F     F     T     T     T     T
                              ^
                        first True — this is what we want
```

The following template finds the first index where P is true. Memorize this one; everything else in the chapter is a variation.

```
from typing import Callable


def first_true(lo: int, hi: int, pred: Callable[[int], bool]) -> int:
    """Smallest i in [lo, hi) with pred(i) True, or hi if there is none.

    Requires pred to be monotone: False ... False True ... True.
    """
    while lo < hi:
        mid = (lo + hi) // 2
        if pred(mid):
            hi = mid           # mid might be the answer; keep it
        else:
            lo = mid + 1       # mid is definitely not the answer
    return lo
```

Invariant: everything left of `lo` is False; everything at or right of `hi` is True (treating `hi` as a virtual True sentinel when it is past the end). The search space is the half-open interval [lo, hi). Each iteration shrinks it strictly, because `mid < hi` always holds when `lo < hi`. When `lo == hi`, the False region and the True region meet: `lo` is the boundary.

Notice what this template does *not* have: an equality check, a `result` variable, a `<=` loop condition, or a `mid - 1`. Every one of those is a place to make an off-by-one mistake, and none of them is needed.

Lower bound, upper bound, and counting

With `first_true`, the standard bounds are one-liners:

```
def lower_bound(nums: list[int], x: int) -> int:
    """First index i with nums[i] >= x (== bisect.bisect_left)."""
    return first_true(0, len(nums), lambda i: nums[i] >= x)


def upper_bound(nums: list[int], x: int) -> int:
    """First index i with nums[i] > x (== bisect.bisect_right)."""
    return first_true(0, len(nums), lambda i: nums[i] > x)
```

These two boundaries bracket the block of elements equal to `x`, which gives several classic problems for free:

Question	Answer
Is `x` present?	`lo = lower_bound(nums, x)`; check `lo < len(nums) and nums[lo] == x`
First occurrence of `x` (LC 34)	`lower_bound(nums, x)` if present
Last occurrence of `x` (LC 34)	`upper_bound(nums, x) - 1` if present
How many times does `x` occur?	`upper_bound(nums, x) - lower_bound(nums, x)`
How many elements are $< x$?	`lower_bound(nums, x)`
How many elements lie in $[a, b]$?	`upper_bound(nums, b) - lower_bound(nums, a)`
Where should `x` be inserted to keep order? (LC 35)	`lower_bound(nums, x)`

Stop and Think: Counting occurrences

Problem: Count how many times a key k occurs in a sorted array of n elements.
A tempting solution: binary search for any copy of k, then walk left and right until the values change. This costs $O(\log n + s)$, where s is the number of copies. That looks fine — until the array is a million copies of k, and the "binary search" algorithm quietly becomes a linear scan.
The fix: don't search for k; search for the *boundaries* of its block. Two boundary searches cost $O(\log n)$ no matter how large the block is: `upper_bound(nums, k) - lower_bound(nums, k)`.
The lesson generalizes: when an answer is a contiguous range, search for its **endpoints**, not for a member.

Use the standard library

Python's `bisect` module implements exactly these searches in C:

```
import bisect

nums = [1, 3, 3, 3, 5, 8]
bisect.bisect_left(nums, 3)    # 1 — first index with nums[i] >= 3
bisect.bisect_right(nums, 3)   # 4 — first index with nums[i] >  3
bisect.insort(nums, 4)         # insert keeping order: O(log n) search + O(n) shift

# Python 3.10+: search by key without building a separate list
events = [(1, "a"), (4, "b"), (9, "c")]
```

```
bisect.bisect_left(events, 5, key=lambda e: e[0])   # 2
```

In an interview, `bisect` is almost always acceptable — but be ready to write `first_true` by hand, since the interesting problems need a custom predicate.

Recognizing a Monotone Predicate

The skill that separates binary search experts from everyone else is spotting a monotone predicate in data that is **not sorted**. Here are three examples where the array is not sorted, yet a clean F…FT…T structure exists.

Rotated sorted array — find the minimum (LC 153)

A sorted array rotated at an unknown pivot, e.g. `[4, 5, 6, 7, 0, 1, 2]`. The predicate $P(i) = \text{nums}[i] \le \text{nums}[-1]$ is False on the left part (values larger than the last element) and True on the right part (the rotated-in smaller values):

```
nums:  4  5  6  7  0  1  2
P(i):  F  F  F  F  T  T  T      P(i) = nums[i] <= nums[-1]
```

```
def find_min_rotated(nums: list[int]) -> int:
    i = first_true(0, len(nums), lambda i: nums[i] <= nums[-1])
    return nums[i]
```

Rotated sorted array — search (LC 33)

With the pivot found, the array is two sorted runs, and ordinary binary search on the correct run finishes the job. An alternative single-pass approach uses the observation that **at least one half around `mid` is always sorted**, and a sorted half lets you decide in $O(1)$ whether the target lies in it:

```
def search_rotated(nums: list[int], target: int) -> int:
    lo, hi = 0, len(nums) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if nums[mid] == target:
            return mid
        if nums[lo] <= nums[mid]:                    # left half is sorted
            if nums[lo] <= target < nums[mid]:
                hi = mid - 1
            else:
                lo = mid + 1
        else:                                        # right half is sorted
            if nums[mid] < target <= nums[hi]:
                lo = mid + 1
            else:
                hi = mid - 1
    return -1
```

With duplicates (LC 81), this code returns wrong answers

The test `nums[lo] <= nums[mid]` is how the code decides which half is sorted. With duplicates it can pick the wrong half. Search for `0` in `[1, 0, 1, 1, 1]`: `nums[lo] = nums[mid] = 1`, so the code assumes the left half `[1, 0, 1]` is sorted. It then rules out the target there and moves `lo` past the `0`. When `nums[lo] == nums[mid] == nums[hi]`, either half could be the sorted one. What you *do* know is that `nums[lo]` and `nums[hi]` both equal `nums[mid]`, which was already checked and isn't the target. So drop those two elements and try again:

```
if nums[lo] == nums[mid] == nums[hi]:
    lo, hi = lo + 1, hi - 1
    continue
```

Each such step removes only two elements, so the worst case becomes $O(n)$. No algorithm can do better. In `[1, 1, 1, 1, 0, 1, 1]`, every probe that lands on a `1` says nothing about where the `0` is, so in the worst case you must look at every position.

Peak element (LC 162)

Find any index with `nums[i] > nums[i-1]` and `nums[i] > nums[i+1]` (with $-\infty$ outside the array; adjacent values differ). The array can be anything, yet the predicate $P(i) = \text{nums}[i] > \text{nums}[i+1]$ ("we are going downhill") has a useful property: if P(mid) is False, we are going uphill, so *some* peak must exist to the right (the slope can't go up forever); if True, some peak exists at `mid` or to its left.

```
def find_peak_element(nums: list[int]) -> int:
    n = len(nums)
    return first_true(0, n - 1, lambda i: nums[i] > nums[i + 1])
```

Here the predicate isn't globally monotone, but the invariant "a peak exists in [lo, hi]" is preserved by each step, and that is all correctness requires.

Why `nums[i + 1]` never goes out of bounds

The search range is `[0, n - 1)`, and `first_true` only evaluates `pred(mid)` with `mid < hi`, so `i ≤ n - 2` and `i + 1 ≤ n - 1` always. The last index is never tested: if no `True` is found, `first_true` returns `hi = n - 1`, which is correct because the array was increasing all the way and nums$[n] = -\infty$. For `n == 1` the loop doesn't run and the answer is `0`.
This depends on the half-open `lo < hi` loop. With the closed-interval `lo <= hi` style, `mid` can reach `n - 1`: on `[1, 2]` the second iteration has `lo = hi = mid = 1` and reads `nums[2]`. That loop also never ends once `lo == hi` and `hi = mid` is taken.

Search a 2D matrix (LC 74)

If each row is sorted and each row starts after the previous row ends, the matrix *is* a sorted array of length mn stored row by row. Map index k to `matrix[k // n][k % n]` and run the ordinary search in $O(\log mn)$.

If instead rows and columns are sorted independently (LC 240), no such flattening exists. Start at the top-right corner: if the value is too large, the entire column is too large (move left); if too small, the entire row is too small (move down). Each step discards a row or a column, giving $O(m + n)$.

Binary Search on the Answer

This is the single most important binary search pattern in interviews. The array being searched is not in the input at all — it is the **range of possible answers**.

The pattern applies when a problem asks for the *minimum* (or maximum) value of something, and it is much easier to **check** whether a given candidate works than to **compute** the optimum directly. If feasibility is monotone — any value larger than a feasible one is also feasible — then:

```
candidate:  1    2    3    4    5    6    7    8
feasible?:  F    F    F    T    T    T    T    T
                          ^
                 minimum feasible answer
```

and the answer is `first_true(lo, hi + 1, feasible)`. The total cost is $O(C \cdot \log R)$, where C is the cost of one feasibility check and R is the size of the answer range.

Koko eating bananas (LC 875)

Koko has `piles` of bananas and `h` hours. At speed k she needs $\lceil p/k \rceil$ hours for a pile of size p. Find the minimum k that finishes in time.

Computing the optimal k directly is awkward. Checking a *given* k is trivial — sum the hours. And if speed k works, any faster speed also works. That is all we need.

```python
def min_eating_speed(piles: list[int], h: int) -> int:
    def can_finish(k: int) -> bool:
        return sum((p + k - 1) // k for p in piles) <= h   # ceil(p / k)

    return first_true(1, max(piles) + 1, can_finish)
```

Bounds: speed 1 is the slowest meaningful speed; speed `max(piles)` finishes every pile in one hour, and since `h >= len(piles)` it always works. **Time:** $O(n \log M)$ with $M = $ `max(piles)`.

Capacity to ship packages within D days (LC 1011)

Packages must ship **in order**; each day the ship loads consecutive packages up to its capacity. Find the minimum capacity to ship everything in `days` days.

Feasibility of a capacity c is a **greedy** check: load each day as full as possible. (Greedy is optimal here because leaving room on an earlier day can never help a later day.)

```python
def ship_within_days(weights: list[int], days: int) -> int:
    def fits(capacity: int) -> bool:
        needed, load = 1, 0
        for w in weights:
            if load + w > capacity:
                needed += 1
                load = 0
            load += w
        return needed <= days

    return first_true(max(weights), sum(weights) + 1, fits)
```

Bounds matter: the capacity must be at least `max(weights)` or the heaviest package never ships; `sum(weights)` ships everything in one day. Starting `lo` below `max(weights)` makes the greedy check silently wrong.

Split array largest sum (LC 410)

Split `nums` into `k` contiguous parts to minimize the largest part sum. This is the *same problem* as LC 1011 in disguise — "capacity" is "largest part sum," "days" is "number of parts" — and the same code solves it. Recognizing that two problems are really one is the whole game.

A catalog of answer-space searches

Problem	Search over	Feasibility check
Koko Eating Bananas (LC 875)	eating speed	total hours $\le h$
Ship Packages (LC 1011)	ship capacity	greedy day count $\le D$
Split Array Largest Sum (LC 410)	max part sum	greedy part count $\le k$
Minimum Days to Make Bouquets (LC 1482)	day	count adjacent bloomed runs
Magnetic Force Between Balls (LC 1552)	min distance (maximize)	greedy placement count $\ge m$
Kth Smallest in Sorted Matrix (LC 378)	value	count of entries $\le$ value $\ge k$
Find the Smallest Divisor (LC 1283)	divisor	sum of ceilings $\le$ threshold

Maximizing instead of minimizing

When the question asks for the *largest* feasible value, feasibility is monotone in the other direction (T…TF…F). Don't write a second template — flip the predicate and step back one:

```
# Largest x in [lo, hi] with ok(x) True, given ok is True...True False...False
largest = first_true(lo, hi + 1, lambda x: not ok(x)) - 1
```

Take-Home Lesson

If you can't see how to *compute* an optimal value but can easily *verify* a candidate, and verification is monotone in the candidate, you have a binary search. Checking is often dramatically easier than optimizing.

One-Sided (Exponential) Search

Suppose the array is unbounded — a stream of 0s followed by an endless run of 1s — and we want the transition point p. Without a known upper bound, we can't pick a midpoint.

The fix is to probe at exponentially increasing positions $1, 2, 4, 8, 16, \ldots$ until we hit a 1. That takes $\lceil \log_2 p \rceil$ probes and produces a window $[2^{j-1}, 2^j]$ that contains p; an ordinary binary search inside it takes $\log_2 p$ more. Total: at most $2 \log_2 p$ comparisons, **independent of the total size** of the data.

```
def exponential_search(pred: Callable[[int], bool]) -> int:
    """First i >= 0 with pred(i) True, for a monotone pred with no known upper bound."""
    if pred(0):
        return 0
    hi = 1
    while not pred(hi):
```

```
        hi *= 2
    return first_true(hi // 2 + 1, hi + 1, pred)
```

Exponential search shines when the answer is likely to be **near the start**: the cost depends on where the answer is, not on how much data there is. LC 702 (Search in a Sorted Array of Unknown Size) is the direct application; Timsort's "galloping mode" uses the same idea to merge runs quickly.

Bisection on Real Numbers

Binary search doesn't need integers. To compute $\sqrt{n}$, note that $f(x) = x^2 - n$ is increasing, negative at $x = 0$, and non-negative at $x = \max(1, n)$. Halving the interval that brackets the sign change is the **bisection method**, and it finds a root of any continuous function once you know two points where its sign differs.

```
def real_sqrt(n: float) -> float:
    lo, hi = 0.0, max(1.0, n)
    for _ in range(100):          # each iteration halves the interval
        mid = (lo + hi) / 2
        if mid * mid < n:
            lo = mid
        else:
            hi = mid
    return lo
```

> **Iterate a fixed number of times**
>
> With floats, a loop like `while hi - lo > 1e-9` can spin forever if the tolerance is below the floating-point spacing at that magnitude. A fixed iteration count is simpler and always terminates: 100 halvings shrink any interval by $2^{100} \approx 10^{30}$, far more precision than a `float` holds.

For the integer version (LC 69, $\lfloor\sqrt{n}\rfloor$), search for the first x with $x^2 > n$ and subtract one:

```
def my_sqrt(n: int) -> int:
    return first_true(0, n + 2, lambda x: x * x > n) - 1
```

Divide and Conquer Beyond the Array

Binary search is best understood not as an array algorithm but as a **question-asking strategy**: every yes/no answer should split the remaining possibilities in half. That view explains two further applications.

Debugging by bisection. A test passed at commit A and fails at commit B, a thousand commits later. Checking out the midpoint and running the test tells you which half contains the culprit; ten rounds find it. `git bisect` automates exactly this. The same strategy works for commenting out halves of a failing function, or halves of a failing input file.

Binary search is inherently sequential — each question depends on the previous answer. When each question is expensive (a lab experiment that takes a week, a build that takes an hour), that's a problem. If you're allowed to ask *arbitrary subsets* rather than halves, you can ask all $\log_2 n$ questions at once: label the n candidates with the binary numbers $0 \ldots n-1$, and let question j be "is the answer among the candidates whose bit j is set?" The $\log_2 n$ answers spell out the answer's label directly. The number of questions is the

same; the rounds collapse to one.

Common Mistakes

1. **No invariant.** Most binary search bugs come from mixing templates: a `<=` loop with `hi = mid`, or a `<` loop with `hi = len(nums) - 1`. Pick one template (the half-open `first_true` is recommended), state its invariant, and never mix.
2. **Infinite loop from rounding.** In a `lo < hi` loop, `mid = (lo + hi) // 2` rounds *down*, so `mid < hi` always and `hi = mid` makes progress. But `lo = mid` does **not** make progress when `hi == lo + 1`. The `first_true` template avoids this by only ever assigning `lo = mid + 1`.
3. **Wrong answer-space bounds.** The initial range must contain the answer. Ship capacity below `max(weights)` is not merely infeasible — it makes the greedy check return wrong results.
4. **Non-monotone predicate.** Binary search on the answer only works if feasibility is monotone. Before coding, argue explicitly: "if x works, why does $x + 1$ work?"
5. **Linear work hiding in the check.** The total cost is (cost of check) $\times \log R$. A check that sorts, copies, or slices can dominate. Also note that `bisect.insort` is $O(n)$ per insertion because of the shift, even though the search is $O(\log n)$.
6. **Integer overflow in other languages.** `(lo + hi) / 2` overflows in Java, C, and C++ when `lo + hi` exceeds $2^{31} - 1$ — a bug that famously sat in the JDK's own binary search for nearly a decade. Python integers don't overflow, but write `lo + (hi - lo) // 2` if you're translating.

Practice Problems

Problem	Idea
Binary Search (LC 704)	Classic search
Search Insert Position (LC 35)	Lower bound
First and Last Position of Element (LC 34)	Lower and upper bound
Find Minimum in Rotated Sorted Array (LC 153)	Predicate `nums[i] <= nums[-1]`
Search in Rotated Sorted Array (LC 33)	One half is always sorted
Find Peak Element (LC 162)	Follow the uphill slope
Search a 2D Matrix (LC 74)	Flatten to 1-D
Search a 2D Matrix II (LC 240)	Staircase from the top-right corner
Time Based Key-Value Store (LC 981)	`bisect_right` on timestamps
Koko Eating Bananas (LC 875)	Search on the answer
Capacity to Ship Packages (LC 1011)	Search on the answer + greedy check
Split Array Largest Sum (LC 410)	Same problem as LC 1011
Median of Two Sorted Arrays (LC 4)	Binary search on the partition point

Two Pointers & Sliding Window

A huge number of array and string problems ask about **pairs** of positions: two elements that sum to a target, two walls that hold the most water, the start and end of the best subarray. An array of n elements has $\binom{n}{2} \approx n^2/2$ pairs, so brute force is quadratic — fine for $n = 1{,}000$, hopeless for $n = 10^5$.

Two pointers and sliding windows are techniques for examining only $O(n)$ of those pairs **while provably not missing the answer**. That "provably" is the whole point. Each technique rests on an argument that, once some pair has been examined, a whole row or column of other pairs can be discarded without looking at them. If you understand the argument, you can apply the technique to problems you've never seen; if you only memorize the template, you'll apply it to problems where it silently gives wrong answers.

Sorting as a Building Block

Many pair problems become easy once the data is sorted. Sorting costs $O(n \log n)$, which is almost never the bottleneck, and it buys you structure: equal elements become adjacent, close elements become neighbors, and moving a pointer changes a value in a *known direction*.

Problem	Brute force	After sorting
Are there duplicates?	Compare all pairs, $O(n^2)$	Duplicates are adjacent: one scan, $O(n)$
Closest pair of numbers	All pairs, $O(n^2)$	Closest pair is adjacent: one scan
Most frequent element (mode)	Count each, $O(n^2)$	Equal elements form runs: one scan
Two elements summing to t	All pairs, $O(n^2)$	Two pointers, $O(n)$
Do two sets intersect?	All pairs, $O(nm)$	Merge-style scan, $O(n + m)$

Take-Home Lesson

Sorting the data is one of the first things to try in the quest for efficiency. A problem that looks quadratic on unsorted input is often linear on sorted input, and paying $O(n \log n)$ to get there is a good deal.

Stop and Think: Sorting vs. hashing

Problem: Hash tables give $O(1)$ expected lookups. For which of the problems in the table above can hashing replace sorting and get $O(n)$ expected time?

Solution:

- **Duplicates, intersection, mode, two-sum:** yes. Insert elements into a `set` or `Counter` and look each one up. Expected $O(n)$.
- **Closest pair:** no. Hashing scatters close values into unrelated buckets, destroying exactly the order information the problem needs.

The general rule: hashing answers "**is this exact value present?**" Sorting answers that *and* "**what is near this value?**" When a problem involves order, ranges, or nearness, you need sorting (or a tree); when it involves only equality, hashing usually wins.

Pattern 1: Opposite-Direction Pointers

Two Sum on a sorted array (LC 167)

Given a sorted array, find two elements summing to `target`.

```python
def two_sum_sorted(nums: list[int], target: int) -> tuple[int, int] | None:
    lo, hi = 0, len(nums) - 1
    while lo < hi:
        s = nums[lo] + nums[hi]
        if s == target:
            return lo, hi
        if s < target:
            lo += 1
        else:
            hi -= 1
    return None
```

Why is it correct? Imagine the $n \times n$ grid of all pairs (i, j) with $i < j$. Brute force checks every cell. The two-pointer algorithm starts at the corner $(0, n-1)$ and argues away an entire row or column with every step:

- If `nums[lo] + nums[hi] < target`, then `nums[lo] + nums[j] < target` for **every** `j <= hi`, because the array is sorted and `nums[j] <= nums[hi]`. So `lo` cannot be part of any solution among the remaining candidates — eliminate the whole row and advance `lo`.
- Symmetrically, if the sum is too large, `hi` pairs too large with every remaining `i`, so eliminate the column.

Each step discards a full row or column, and there are only n of each, so the loop runs at most n times: $O(n)$ time, $O(1)$ space. On *unsorted* input, use a hash map instead (LC 1) — same $O(n)$, but $O(n)$ space.

> **Take-Home Lesson**
>
> A two-pointer algorithm is only as good as its **elimination argument**: "when I move this pointer, every pair I skip is provably not the answer." Before using the pattern, say the argument out loud. If you can't, the pattern probably doesn't apply.

3Sum (LC 15)

Find all unique triplets summing to zero. Fix the smallest element `nums[i]`, and the rest of the problem is Two Sum on the suffix with target `-nums[i]`:

```python
def three_sum(nums: list[int]) -> list[list[int]]:
    nums.sort()
    n = len(nums)
    result = []
    for i in range(n - 2):
        if nums[i] > 0:
            break                          # smallest element positive: no zero sum
        if i > 0 and nums[i] == nums[i - 1]:
            continue                       # same first element: same triplets
        lo, hi = i + 1, n - 1
        while lo < hi:
            s = nums[i] + nums[lo] + nums[hi]
            if s < 0:
                lo += 1
```

```
            elif s > 0:
                hi -= 1
            else:
                result.append([nums[i], nums[lo], nums[hi]])
                lo += 1
                while lo < hi and nums[lo] == nums[lo - 1]:
                    lo += 1                # skip duplicate second elements
    return result
```

Time: $O(n^2)$ — n iterations of an $O(n)$ scan; the sort is dominated. **Space:** $O(1)$ beyond the output (plus the sort).

This reduction — "fix one element, solve the smaller problem on the rest" — generalizes: k-Sum costs $O(n^{k-1})$. For 3Sum, $O(n^2)$ is believed to be essentially optimal; the "3SUM conjecture" is used to argue that many geometric problems can't be solved in subquadratic time either.

Container with most water (LC 11)

Choose two walls to maximize `min(h[i], h[j]) * (j - i)`.

```
def max_area(height: list[int]) -> int:
    lo, hi = 0, len(height) - 1
    best = 0
    while lo < hi:
        best = max(best, min(height[lo], height[hi]) * (hi - lo))
        if height[lo] < height[hi]:
            lo += 1
        else:
            hi -= 1
    return best
```

The array isn't sorted, but an elimination argument still exists. Suppose `height[lo] < height[hi]`. Any container using wall `lo` with some other wall `j < hi` is *narrower* than the current one, and its height is still at most `height[lo]`. So every remaining pair involving `lo` is no better than the one we just measured — `lo` can be discarded. The shorter wall is always the one to move.

Trapping rain water (LC 42)

Water above position i is $\min(\text{maxLeft}_i, \text{maxRight}_i) - h_i$: it fills up to the lower of the tallest walls on either side. The direct solution precomputes both maxima for every position:

```
def trap(height: list[int]) -> int:
    n = len(height)
    left_max = [0] * n                 # left_max[i] = tallest wall in height[:i+1]
    right_max = [0] * n                # right_max[i] = tallest wall in height[i:]
    for i in range(n):
        left_max[i] = max(height[i], left_max[i - 1] if i > 0 else 0)
    for i in range(n - 1, -1, -1):
        right_max[i] = max(height[i], right_max[i + 1] if i < n - 1 else 0)
    return sum(min(left_max[i], right_max[i]) - height[i] for i in range(n))
```

$O(n)$ time and space. The two-pointer version drops both arrays, keeping only one running maximum per side, for $O(1)$ space:

```python
def trap(height: list[int]) -> int:
    lo, hi = 0, len(height) - 1
    left_max = right_max = water = 0
    while lo < hi:
        if height[lo] < height[hi]:
            left_max = max(left_max, height[lo])
            water += left_max - height[lo]
            lo += 1
        else:
            right_max = max(right_max, height[hi])
            water += right_max - height[hi]
            hi -= 1
    return water
```

Why can we settle position `lo` knowing only `left_max`? Its water depends on min(maxLeft, maxRight), and we don't know the true maxRight yet. We don't need to; we only need to know that it is *at least* `left_max`. That follows from an invariant: whenever `left_max` was raised to some `height[lo']`, it happened in the `if` branch, where `height[lo'] < height[hi']` for a wall `hi'` at or right of the current `hi`. So some wall on the right is taller than `left_max`, the minimum is `left_max`, and the water at `lo` is exactly `left_max - height[lo]`. The `else` branch is symmetric.

> **Correctness needs care, even for short code**
>
> Trapping rain water is 12 lines, and the argument for why it works is longer than the code. That is normal. Reasonable-looking pointer algorithms are easy to write and easy to get wrong; the way to be sure is an invariant, not a few passing test cases.

Palindromes (LC 125) and reversals

Two pointers from the ends, comparing and moving inward, check whether a sequence is a palindrome or reverse it in place, in $O(n)$ time and $O(1)$ space. Expand-around-center (two pointers moving *outward* from each of the $2n - 1$ possible centers) finds the longest palindromic substring in $O(n^2)$ (LC 5).

Pattern 2: Same-Direction Pointers

Reader and writer: in-place compaction

A `read` pointer scans every element; a `write` pointer marks where the next *kept* element goes. Everything before `write` is the finished output.

```python
def remove_duplicates(nums: list[int]) -> int:
    """LC 26: dedupe a sorted array in place; return the new length."""
    write = 0
    for read in range(len(nums)):
        if write == 0 or nums[read] != nums[write - 1]:
            nums[write] = nums[read]
            write += 1
    return write


def move_zeroes(nums: list[int]) -> None:
    """LC 283: move zeros to the end, preserving the order of the rest."""
```

```
    write = 0
    for read in range(len(nums)):
        if nums[read] != 0:
            nums[write], nums[read] = nums[read], nums[write]
            write += 1
```

Invariant: `nums[:write]` is exactly the output for `nums[:read]`. Since `write <= read`, we never overwrite an element we haven't read yet. The same pattern is the partition step in quicksort (see Sorting).

Merging two sorted sequences

Two pointers, one per sorted input, repeatedly take the smaller head. This is the merge step of merge sort, and it also answers set questions on sorted inputs in $O(n+m)$: intersection, union, difference, "do these sets overlap?"

```
def intersect_sorted(a: list[int], b: list[int]) -> list[int]:
    i = j = 0
    out = []
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            i += 1              # a[i] is smaller than everything left in b
        elif a[i] > b[j]:
            j += 1
        else:
            out.append(a[i])
            i += 1
            j += 1
    return out
```

When merging into an array that has spare room at its end (LC 88), fill it **from the back** so you never overwrite unread elements.

Fast and slow

Pointers moving at different speeds find the middle of a list, detect cycles, and locate cycle entrances. These are covered in Linked Lists; the same idea finds the duplicate in LC 287, where the array is interpreted as a linked list `i -> nums[i]`.

Pattern 3: Sliding Window

A **subarray** (or substring) is determined by its endpoints $[l, r]$, so there are $\Theta(n^2)$ of them. A sliding window enumerates only $O(n)$ of them: it moves `r` forward one step at a time, and for each `r`, moves `l` forward only as far as necessary.

When does it work?

The sliding window needs one structural property. For a "find the longest valid window" problem:

> *If a window is valid, every window inside it is also valid.*

Equivalently: extending a window can only make it *less* valid, and shrinking it can only make it *more* valid. Under this property, when `r` advances and the window becomes invalid, some prefix of the window must be

dropped — and since `l` only has to move right to restore validity, and never back, `l` and `r` each move at most n times.

Condition	Monotone?	Sliding window?
"At most k distinct characters"	Yes — sub-windows have fewer distinct chars	Yes
"No repeated characters"	Yes	Yes
"Sum $\leq k$", all elements ≥ 0	Yes — removing elements can't increase the sum	Yes
"Sum $= k$", elements may be negative	**No** — removing a negative increases the sum	No: use prefix sums
"Exactly k distinct"	**No** — a sub-window may have fewer than k	Not directly: use at-most(k) − at-most($k-1$)

The template

```
def longest_valid_window(s: str) -> int:
    window = {}          # state describing s[left..right]
    left = best = 0
    for right, ch in enumerate(s):
        # 1. extend: add s[right] to the window state
        window[ch] = window.get(ch, 0) + 1

        # 2. shrink: restore validity by dropping from the left
        while window[ch] > 1:           # <- the validity condition goes here
            window[s[left]] -= 1
            left += 1

        # 3. record: s[left..right] is now the longest valid window ending at right
        best = max(best, right - left + 1)
    return best
```

This instance solves **Longest Substring Without Repeating Characters (LC 3)**: the window is valid when no character appears twice, and the only character that can have become duplicated is the one just added.

Why is it $O(n)$ when there's a `while` inside a `for`? Amortization: `left` only increases, and it can't pass `right`, so across the whole run the inner loop body executes at most n times in total. (See Complexity Analysis.)

Shortest window: Minimum Window Substring (LC 76)

For "shortest valid window" problems, the logic flips: extend until the window becomes valid, then shrink **while it stays valid**, recording the answer inside the shrink loop.

```
from collections import Counter


def min_window(s: str, t: str) -> str:
    need = Counter(t)
    missing = len(t)                # characters of t not yet covered
    left = 0
    best_start, best_len = 0, float("inf")
    for right, ch in enumerate(s):
```

```
        if need[ch] > 0:
            missing -= 1
        need[ch] -= 1                # may go negative: surplus copies
        while missing == 0:          # window is valid: try to shrink
            if right - left + 1 < best_len:
                best_start, best_len = left, right - left + 1
            need[s[left]] += 1
            if need[s[left]] > 0:    # just dropped a needed character
                missing += 1
            left += 1
    return "" if best_len == float("inf") else s[best_start:best_start + best_len]
```

The `missing` counter makes the validity check $O(1)$ instead of comparing two whole counters each step.

Longest repeating character replacement (LC 424)

A window can be made uniform with at most k replacements iff `window_size - count_of_most_frequent_char <= k`.

```
def character_replacement(s: str, k: int) -> int:
    count: dict[str, int] = {}
    left = max_freq = 0
    for right, ch in enumerate(s):
        count[ch] = count.get(ch, 0) + 1
        max_freq = max(max_freq, count[ch])
        if (right - left + 1) - max_freq > k:
            count[s[left]] -= 1
            left += 1
    return len(s) - left
```

Two subtleties make this shorter than the template. First, `max_freq` is never decreased when the window shrinks, so it may be stale. That's harmless: the answer only improves when a window beats the best `max_freq` seen so far, and a stale (too high) value only prevents shrinking in cases where no improvement was possible. Second, the window never shrinks — it slides at constant size when invalid — so its final size is the answer.

Counting subarrays: the "at most" trick

To count subarrays satisfying a monotone condition, note that for each `right`, **every** start in `[left, right]` gives a valid window, contributing `right - left + 1` subarrays. For non-monotone "exactly k" conditions, subtract two monotone counts:

```
def subarrays_with_k_distinct(nums: list[int], k: int) -> int:
    """LC 992: number of subarrays with exactly k distinct values."""
    def at_most(k: int) -> int:
        count: dict[int, int] = {}
        left = total = 0
        for right, x in enumerate(nums):
            count[x] = count.get(x, 0) + 1
            while len(count) > k:
                count[nums[left]] -= 1
                if count[nums[left]] == 0:
                    del count[nums[left]]
                left += 1
            total += right - left + 1
```

```
        return total

    return at_most(k) - at_most(k - 1)
```

Fixed-size windows

When the window size k is given, there's no shrinking decision at all: add `nums[i]`, remove `nums[i - k]`, and update the answer once the window is full. Maximum average subarray (LC 643), anagram search (LC 438, LC 567 — compare character counts of the window to the pattern), and any "every window of size k" statistic follow this shape.

Sliding window maximum (LC 239)

Maintaining a *sum* under add/remove is trivial; maintaining a *maximum* is not, since removing the current maximum requires knowing the next one. A **monotonic deque** stores indices whose values are decreasing from front to back. A new element evicts every smaller element from the back — those can never be a window's maximum again, since the newcomer is larger *and* will stay in the window longer.

```
from collections import deque


def max_sliding_window(nums: list[int], k: int) -> list[int]:
    dq: deque[int] = deque()          # indices; nums[dq] strictly decreasing
    result = []
    for i, x in enumerate(nums):
        while dq and nums[dq[-1]] <= x:
            dq.pop()                  # dominated: older and not larger
        dq.append(i)
        if dq[0] <= i - k:
            dq.popleft()              # front fell out of the window
        if i >= k - 1:
            result.append(nums[dq[0]])
    return result
```

Each index is appended once and popped at most once: $O(n)$ total. More on monotonic structures in Stacks & Queues.

When the Window Breaks: Negative Numbers

With negative numbers, the window's key property fails: shrinking can *decrease* the sum, extending can *increase* it. We need different tools.

Largest subrange: three algorithms for one problem

A hedge fund had these monthly returns, and wants to advertise its best stretch:

$$[-17,\ 5,\ 3,\ -10,\ 6,\ 1,\ 4,\ -3,\ 8,\ 1,\ -13,\ 4]$$

The year was a loss overall, but months 5 through 10 (`[6, 1, 4, -3, 8, 1]`) gained 17 — the **maximum subarray sum** (LC 53). This problem is a good lesson in how algorithm design improves on brute force step by step.

$O(n^2)$ **— try every start.** For each start, extend the end and keep a running sum.

$O(n \log n)$ **— divide and conquer.** Split the array in half. The best subarray lies entirely in the left half, entirely in the right half, or **straddles the middle**. The first two are recursive calls. The straddling one is the best subarray *ending* at the middle (a linear sweep leftward) plus the best one *starting* just after it (a linear sweep rightward). Linear work plus two half-size calls: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$.

$O(n)$ **— Kadane's algorithm.** Let B_i be the best sum of a subarray *ending exactly at* i. Either it extends the best subarray ending at $i-1$, or it starts fresh at i:

$$B_i = \max(a_i,\ B_{i-1} + a_i)$$

```python
def max_subarray(nums: list[int]) -> int:
    best = cur = nums[0]
    for x in nums[1:]:
        cur = max(x, cur + x)     # extend, or start over if the prefix hurts
        best = max(best, cur)
    return best
```

That recurrence is a one-dimensional dynamic program (see Dynamic Programming), and a good example of how asking "what is the best answer **ending here**?" turns a quadratic search into a single pass.

Prefix sums

Define $P_0 = 0$ and $P_j = a_0 + \cdots + a_{j-1}$. Then the sum of any subarray is a difference of two prefix sums:

$$\sum_{k=i}^{j-1} a_k = P_j - P_i$$

This turns questions about subarrays into questions about **pairs of prefix sums**, and pairs are something we know how to handle:

- **Range sum queries** (LC 303): precompute P, answer each query in $O(1)$.
- **Maximum subarray**: maximize $P_j - P_i$ over $i < j$ — track the minimum prefix seen so far. (This is also Best Time to Buy and Sell Stock, LC 121, where prices play the role of prefix sums.)
- **Count subarrays summing to** k (LC 560): for each j, count earlier i with $P_i = P_j - k$. A hash map of prefix-sum counts does it in one pass:

```python
def subarray_sum(nums: list[int], k: int) -> int:
    seen = {0: 1}                 # prefix sum -> how many times seen
    prefix = count = 0
    for x in nums:
        prefix += x
        count += seen.get(prefix - k, 0)
        seen[prefix] = seen.get(prefix, 0) + 1
    return count
```

The same idea handles "subarray sum divisible by k" (LC 974: key on `prefix % k`) and "longest subarray with equal 0s and 1s" (LC 525: map 0 to -1 and store the *first* index of each prefix sum). In 2-D, prefix sums over rectangles give $O(1)$ submatrix sums (LC 304).

Recognizing the Pattern

Signal in the problem	Technique
Sorted array, find a pair with a target sum/difference	Opposite pointers
Find triplets / quadruplets	Sort, fix one element, two pointers on the rest
Choose two boundaries to maximize a width × height	Opposite pointers; move the limiting side
Remove / compact elements in place	Reader–writer pointers
Two sorted inputs	Merge-style pointers
Longest / shortest substring or subarray satisfying a **monotone** condition	Variable sliding window
Every window of size k	Fixed sliding window
Max / min over each window	Monotonic deque
"Exactly k"	at-most(k) − at-most($k-1$)
Subarray sums with negative numbers	Prefix sums + hash map
Best subarray sum	Kadane (DP on "best ending here")

Common Mistakes

1. **Using a sliding window on a non-monotone condition.** The classic failure: "subarray sum equals k" with negative numbers. Check the property before coding: *is every sub-window of a valid window valid?*
2. **Two pointers without an elimination argument.** Moving "the pointer that seems right" gives plausible code that fails on edge cases. Justify each move.
3. **Forgetting to skip duplicates** in k-Sum problems, producing repeated triplets — or skipping them before the first use, losing valid answers.
4. **Off-by-one in window size.** The window $[l, r]$ has size `r - l + 1`. The count of subarrays ending at `r` with start in $[l, r]$ is also `r - l + 1`.
5. **Recomputing window state from scratch.** `len(set(s[left:right + 1]))` inside the loop is $O(k)$ per step, making the whole algorithm $O(nk)$. Update state incrementally as elements enter and leave.
6. **Missing the empty prefix.** In prefix-sum-plus-hash-map problems, initialize the map with `{0: 1}` (or `{0: -1}` for index-based variants); otherwise subarrays starting at index 0 are never counted.

Practice Problems

Problem	Technique
Two Sum II (LC 167)	Opposite pointers
Valid Palindrome (LC 125)	Opposite pointers
3Sum (LC 15)	Sort + fix one + two pointers

Problem	Technique
Container With Most Water (LC 11)	Move the shorter wall
Trapping Rain Water (LC 42)	Two pointers with running maxima
Remove Duplicates from Sorted Array (LC 26)	Reader–writer
Merge Sorted Array (LC 88)	Merge from the back
Longest Substring Without Repeating Characters (LC 3)	Variable window
Longest Repeating Character Replacement (LC 424)	Variable window
Permutation in String (LC 567)	Fixed window + counts
Minimum Window Substring (LC 76)	Shrinking window
Subarrays with K Different Integers (LC 992)	at-most trick
Sliding Window Maximum (LC 239)	Monotonic deque
Maximum Subarray (LC 53)	Kadane
Subarray Sum Equals K (LC 560)	Prefix sums + hash map

Bit Manipulation

An integer is two things at once: a number, and a fixed-length array of bits. Most code only uses the first view. Bit manipulation uses the second, and it gets two things in return:

1. **A compact set representation.** A 64-bit integer is a set over a 64-element universe, and union, intersection, and complement are single machine instructions.
2. **Free parallelism.** One `&` operates on every bit at once. Algorithms that process a whole word per step can be dozens of times faster than their element-by-element versions.

Bit tricks have a reputation for being clever puzzles. The goal of this chapter is to show that the handful you actually need all follow from a few simple facts about binary arithmetic, so that you can derive them rather than memorize them.

Bits and Logarithms

There are 2 bit strings of length 1, 4 of length 2, and 2^w of length w. Turning that around: to give n different things distinct labels, you need $w = \lceil \log_2 n \rceil$ bits. This one fact explains a lot:

- A number n has $\lfloor \log_2 n \rfloor + 1$ bits, so **looping over the bits of n costs $O(\log n)$**, not $O(1)$ and not $O(n)$.
- A set of n items has 2^n subsets, and a subset is exactly an n-bit string: bit i says whether item i is in it. So **integers $0 \dots 2^n - 1$ are the subsets of n items**.
- $\log_2$ of the number of possibilities is the number of yes/no questions needed to identify one of them — binary search, decision trees, and information-theoretic lower bounds are all the same idea.

The Operators

Operator	Meaning	5 **op** 3 (**`101` op `011`**)
`a & b`	AND: 1 where **both** are 1 — set intersection	`001` = 1
`a | b`	OR: 1 where **either** is 1 — set union	`111` = 7
`a ^ b`	XOR: 1 where they **differ** — symmetric difference	`110` = 6
`~a`	NOT: flip every bit — set complement	−6 in Python (see below)
`a << k`	shift left: multiply by 2^k	`5 << 1` = 10
`a >> k`	shift right: floor-divide by 2^k	`5 >> 1` = 2

> **Precedence trap**
>
> In Python, **arithmetic** binds tighter than shifts, and shifts bind tighter than `&`, `^`, `|`. So `1 << n - 1` means `1 << (n - 1)`, and `x & y + 1` means `x & (y + 1)`. (Comparisons bind looser than all of them,

> so `x & 1 == 0` does mean `(x & 1) == 0` — unlike in C, where it's a classic bug.) Always parenthesize mixed expressions.

Negative numbers: two's complement

Hardware stores a w-bit signed integer so that ordinary addition "just works" modulo 2^w. The representation of $-x$ is the bit pattern that, added to x, overflows to zero:

$$-x \equiv \sim x + 1 \pmod{2^w}$$

Why? $x + \sim x$ is all ones ($= 2^w - 1$), so $x + (\sim x + 1) = 2^w \equiv 0$. From this identity come two of the most useful tricks below.

Python integers have unlimited width. Conceptually, a negative Python int is two's complement with an *infinite* run of leading 1 bits. So `~5 == -6`, `-1` is "all ones forever," and `x >> k` on a negative `x` rounds toward $-\infty$. When a problem asks you to emulate 32-bit behavior, mask explicitly:

```python
MASK32 = 0xFFFF_FFFF


def to_signed32(x: int) -> int:
    """Interpret the low 32 bits of x as a signed 32-bit integer."""
    x &= MASK32
    return x - (1 << 32) if x >> 31 else x
```

The Algebra of XOR

XOR is addition modulo 2 on each bit independently — addition **without carries**. That makes it behave like `+` in a group:

Property	Identity
Identity element	$a \oplus 0 = a$
Every element is its own inverse	$a \oplus a = 0$
Commutative, associative	Order and grouping don't matter
Cancellation	$a \oplus b = c \iff a = b \oplus c$

The key consequence: XOR-ing a collection of values cancels every value that appears an even number of times. What survives is the XOR of the values appearing an odd number of times. This single fact solves a whole family of problems:

```python
from functools import reduce
from operator import xor


def single_number(nums: list[int]) -> int:
    """LC 136: every element appears twice except one."""
    return reduce(xor, nums, 0)
```

```python
def missing_number(nums: list[int]) -> int:
    """LC 268: nums contains n distinct values from 0..n; find the absent one."""
    n = len(nums)
    return reduce(xor, range(n + 1), 0) ^ reduce(xor, nums, 0)
```

In `missing_number`, every present value appears twice across the two XORs and cancels; the missing one appears once. (The sum formula $n(n+1)/2 - \sum \text{nums}$ also works, but in fixed-width languages it can overflow; XOR cannot.)

Addition from XOR and AND

Since XOR is addition without carries, and a carry occurs exactly where both bits are 1:

$$a + b = (a \oplus b) + 2 \cdot (a \,\&\, b)$$

Repeating until there are no carries left gives addition using only bitwise operations (LC 371). In Python, the infinite-width integers mean negative numbers never run out of carries, so we must work within 32 bits:

```python
def get_sum(a: int, b: int) -> int:
    """LC 371: a + b without + or -, with 32-bit semantics."""
    while b & MASK32:
        a, b = a ^ b, (a & b) << 1
    return to_signed32(a)
```

Single-Bit Operations

Everything here uses a **mask** with a single 1 at position i: `1 << i`.

```python
def get_bit(x: int, i: int) -> int:
    return (x >> i) & 1

def set_bit(x: int, i: int) -> int:
    return x | (1 << i)

def clear_bit(x: int, i: int) -> int:
    return x & ~(1 << i)

def toggle_bit(x: int, i: int) -> int:
    return x ^ (1 << i)
```

The lowest set bit

Subtracting 1 from x turns its lowest 1 bit into 0 and all the 0s below it into 1s (the borrow propagates up to the first 1):

```
x      = 1011 0100
x - 1  = 1011 0011     lowest 1 became 0; trailing 0s became 1s
x & (x-1) = 1011 0000  -> lowest set bit REMOVED
```

And since $-x = \sim x + 1 = \sim (x - 1)$, the negation flips every bit *above* the lowest 1 while leaving it and the zeros below it intact:

```
x      = 1011 0100
-x     = 0100 1100     (as two's complement)
x & -x = 0000 0100     -> lowest set bit ISOLATED
```

Expression	Effect	Typical use
`x & (x - 1)`	Clears the lowest set bit	Popcount loop; power-of-two test
`x & -x`	Isolates the lowest set bit	Fenwick trees; splitting by a differing bit
`(x & (x - 1)) == 0`	True iff x has at most one set bit	`x > 0 and (x & (x - 1)) == 0` tests powers of two

Counting bits

```python
def popcount(x: int) -> int:
    """Kernighan's method: one iteration per SET bit, not per bit."""
    count = 0
    while x:
        x &= x - 1
        count += 1
    return count
```

In practice use the built-ins: `x.bit_count()` (Python 3.10+) for the number of 1s, and `x.bit_length()` for the position of the highest 1 plus one — that is, $\lfloor \log_2 x \rfloor + 1$ for $x > 0$.

To count bits for **every** number from 0 to n (LC 338), don't call popcount n times. Dropping the last bit of i gives $i \gg 1$, whose answer is already known:

```python
def count_bits(n: int) -> list[int]:
    bits = [0] * (n + 1)
    for i in range(1, n + 1):
        bits[i] = bits[i >> 1] + (i & 1)
    return bits
```

That's a tiny dynamic program: each answer is built from a smaller, already-solved one.

Bit Vectors: Integers as Sets

When the universe is small and fixed — the 26 lowercase letters, the 9 digits of a Sudoku row, the $n \leq 20$ items of a search problem — an integer is an excellent set.

Set operation	Bit vector	Python `set`
$\{\}$, universe U	`0`, `(1 << n) - 1`	`set()`, `set(range(n))`
$i \in S$	`S >> i & 1`	`i in S`
$S \cup \{i\}$, $S \setminus \{i\}$	`S | 1 << i`, `S & ~(1 << i)`	`S | {i}`, `S - {i}`
$S \cup T$, $S \cap T$, $S \setminus T$	`S | T`, `S & T`, `S & ~T`	`S | T`, `S & T`, `S - T`
$S \subseteq T$	`(S & ~T) == 0`	`S <= T`
$\lvert S \rvert$	`S.bit_count()`	`len(S)`

Set operation	Bit vector	Python `set`
Hashable key for memoization	Already an `int`	Needs `frozenset`

Bit vectors are compact and every operation is a single word operation for $n \leq 64$. Their weakness is sparse sets over large universes: listing the members of a nearly empty bit vector still means scanning all n positions. For large, sparse universes, use a hash set.

Bit vectors in practice

- **Valid Sudoku (LC 36):** one 9-bit mask per row, column, and box.
- **Maximum Product of Word Lengths (LC 318):** precompute each word's 26-bit letter mask; two words share no letters iff `(mask1 & mask2) == 0`. That turns an $O(L)$ comparison into $O(1)$.
- **Memoizing on a set of visited items:** a bitmask is a hashable `int`, so it works directly as a `dict` key or `lru_cache` argument.

Word-level parallelism: bitsets as a speedup

Python's unbounded integers make a surprisingly powerful trick available. Take **Partition Equal Subset Sum (LC 416)**: can the numbers be split into two halves of equal sum? It's enough to find one subset summing to half the total, because the remaining numbers then make up the other half. The standard DP tracks the set of reachable subset sums. Represent that set as the bits of one big integer — bit s is 1 iff sum s is reachable. Adding a number x to every reachable sum is a *shift*:

```
def can_partition(nums: list[int]) -> bool:
    total = sum(nums)
    if total % 2:
        return False
    reachable = 1                          # only sum 0 is reachable
    for x in nums:
        reachable |= reachable << x        # old sums, plus old sums + x
    return bool((reachable >> (total // 2)) & 1)
```

The DP is the same $O(n \cdot S)$ in principle, but each step processes S sums with a handful of word-sized operations implemented in C, so it typically runs an order of magnitude or more faster than a list-of-booleans version.

Generating Subsets

Every subset-search problem needs a systematic way to visit all 2^n subsets. Three orders are useful.

Binary counting

Counting from 0 to $2^n - 1$ visits every n-bit string exactly once, and hence every subset exactly once. This bijection between integers and subsets is the key to all subset-generation problems.

```
def all_subsets(items: list[int]) -> list[list[int]]:
    n = len(items)
    return [
        [items[i] for i in range(n) if mask >> i & 1]
        for mask in range(1 << n)
    ]
```

Time: $O(n \cdot 2^n)$. Remember the practical limit from Complexity Analysis: $2^{20} \approx 10^6$ is comfortable, $2^{30} \approx 10^9$ is not.

The bijection also gives **ranking** (subset → integer: set the bits of its members) and **unranking** (integer → subset: read off its bits). So "the next subset" is just `mask + 1`, and "a random subset" is `random.getrandbits(n)`.

Gray code: minimum-change order

In a **Gray code**, consecutive subsets differ by exactly one element. That matters when the cost of evaluating a subset can be *updated* incrementally: moving to the next subset is then one insertion or deletion instead of a full recomputation.

The recursive construction is elegant: take the Gray code for $n-1$ bits, follow it with a **reflected** (reversed) copy with bit $n-1$ set. The two halves meet at a single flipped bit, and the reflection guarantees the rest stays one-change-apart. The closed form is even shorter:

```python
def gray_code(n: int) -> list[int]:
    """LC 89: the i-th Gray code is i XOR (i >> 1)."""
    return [i ^ (i >> 1) for i in range(1 << n)]
```

Submasks of a mask

To iterate over every subset of a given set `mask` (not of the whole universe), use `sub = (sub - 1) & mask`. Subtracting 1 decrements the number, and the `& mask` throws away bits outside `mask` — so this counts down through exactly the submasks, in decreasing order:

```python
def submasks(mask: int) -> list[int]:
    result = []
    sub = mask
    while True:
        result.append(sub)
        if sub == 0:
            break
        sub = (sub - 1) & mask
    return result
```

If you iterate over the submasks of *every* mask of n bits, the total work is not 4^n but 3^n: each element is independently either outside the mask, in the mask but not the submask, or in both. This bound is what makes "DP over subsets of subsets" feasible for $n \leq 15$ or so.

> **Take-Home Lesson**
>
> Every subset of $\{0, \ldots, n-1\}$ *is* an integer in $[0, 2^n)$. Once you internalize that, enumerating subsets is a `for` loop, a set of visited items is a dictionary key, and "all subsets" dynamic programming (see Dynamic Programming) is ordinary DP with integer states.

Classic Problems

Single Number III (LC 260): two unique elements

Every element appears twice except **two**, a and b. XOR-ing everything gives $a \oplus b \neq 0$. Any set bit of $a \oplus b$ is a position where a and b differ — so it partitions the input into two groups, each containing exactly one

of them (and pairs of duplicates, which stay together and cancel).

```
def single_number_iii(nums: list[int]) -> list[int]:
    both = reduce(xor, nums, 0)
    diff = both & -both                  # any bit where a and b differ
    a = 0
    for x in nums:
        if x & diff:
            a ^= x
    return [a, both ^ a]
```

Single Number II (LC 137): everything appears three times

XOR cancels pairs, not triples. Generalize instead: look at each bit position independently. Each triplicate contributes 0 or 3 to that position's count of 1s, so the count **modulo 3** is the unique element's bit.

```
def single_number_ii(nums: list[int]) -> int:
    result = 0
    for i in range(32):
        if sum((x >> i) & 1 for x in nums) % 3:
            result |= 1 << i
    return to_signed32(result)
```

This works for any repetition count k: replace 3 with k. There is a slicker $O(1)$-state solution that encodes the count mod 3 in two bitmasks (`ones`, `twos`), but the counting version is easier to get right and to explain.

Reverse bits (LC 190)

Shift bits out of the bottom of `n` and into the bottom of the result:

```
def reverse_bits(n: int) -> int:
    result = 0
    for _ in range(32):
        result = (result << 1) | (n & 1)
        n >>= 1
    return result
```

Bitwise AND of a range (LC 201)

The AND of all integers in $[m, n]$ keeps only the **common binary prefix** of m and n: below the first position where they differ, the range passes through both a 0 and a 1 in every lower bit. Strip low bits until they agree:

```
def range_bitwise_and(m: int, n: int) -> int:
    shift = 0
    while m != n:
        m >>= 1
        n >>= 1
        shift += 1
    return m << shift
```

Maximum XOR of two numbers (LC 421)

For each number, greedily prefer the opposite bit at each position from the top down. A binary trie over the bits answers "is there a number with this prefix?" in $O(\text{bits})$. See Trie & Union Find.

Common Mistakes

1. **Operator precedence.** `1 << n - 1` is `1 << (n - 1)` and `x & y + 1` is `x & (y + 1)`, because arithmetic binds tighter than shifts and bitwise operators. `x & 1 == 0` happens to work in Python, but the same line is a bug in C and Java. Parenthesize everything.
2. **Forgetting Python's infinite width.** `~x` is `-x - 1`, not a 32-bit complement. Negative numbers have infinitely many leading 1s, so loops like `while x: x >>= 1` never terminate for negative `x`. Mask with `0xFFFFFFFF` whenever a problem assumes fixed width.
3. **Assuming bit loops are $O(1)$.** Iterating over the bits of n is $O(\log n)$. Iterating over all 2^n masks is exponential — respect the $n \leq 20$ rule.
4. **Using a bit vector on a sparse, large universe.** A 10^6-bit integer representing a 3-element set wastes memory and makes iteration slow. Use a `set`.
5. **Reaching for bit tricks when clarity is cheaper.** `x % 2` and `x // 2` are as fast as `x & 1` and `x >> 1` in Python, and read better. Use bits when they express the idea (sets, XOR cancellation, masks), not as micro-optimization.

Practice Problems

Problem	Idea
Single Number (LC 136)	XOR cancellation
Missing Number (LC 268)	XOR with the full range
Number of 1 Bits (LC 191)	`x & (x - 1)`
Counting Bits (LC 338)	`bits[i >> 1] + (i & 1)`
Reverse Bits (LC 190)	Shift out, shift in
Sum of Two Integers (LC 371)	XOR + carries, 32-bit masking
Single Number II (LC 137)	Per-bit count mod 3
Single Number III (LC 260)	Split by a differing bit
Subsets (LC 78)	Binary counting
Gray Code (LC 89)	`i ^ (i >> 1)`
Bitwise AND of Numbers Range (LC 201)	Common prefix
Maximum Product of Word Lengths (LC 318)	Letter masks
Partition Equal Subset Sum (LC 416)	Bitset of reachable sums
Maximum XOR of Two Numbers (LC 421)	Binary trie, greedy by bit

Sorting

Sorting is the most thoroughly studied problem in computer science, and for good reason. It is the basic building block of countless algorithms: once data is sorted, searching becomes logarithmic, duplicates become adjacent, and closest pairs become neighbors (see Two Pointers & Sliding Window). A rule of thumb worth adopting: **when stuck on an array problem, ask what would become easy if the input were sorted.**

Sorting is also the best showcase of algorithm design paradigms. Every major technique produces a good sorting algorithm: incremental insertion gives insertion sort, data structures give heapsort, divide and conquer gives mergesort, randomization gives quicksort, and bucketing gives distribution sort. Studying them together is studying algorithm design in miniature.

In practice, you will almost never implement a sort — Python's built-in is excellent. What you *will* do is (1) specify precisely what "sorted" means for your data, (2) reuse the *ideas* inside the sorting algorithms (merging, partitioning, bucketing) to solve other problems, and (3) recognize when you can beat $O(n \log n)$.

Pragmatics: What Does "Sorted" Mean?

Before choosing an algorithm, answer four questions about the order you want.

Increasing or decreasing? Pass `reverse=True`, or negate a numeric key.

Sort by what key? Records are usually sorted by a field, and the rest of the record must travel with it. Python's `key` function extracts the field; it is called once per element (n calls, not one per comparison), so even an expensive key is affordable.

What happens to ties? A sort is **stable** if elements with equal keys keep their original relative order. Python's sort is guaranteed stable. Stability matters whenever you sort in several passes, or when the input order carries meaning (e.g., arrival time).

How do non-numeric values compare? Tuples compare lexicographically, strings compare by code point (so `"Z" < "a"`), and custom orders need a custom key or comparator.

```
people = [("alice", 30), ("bob", 25), ("carol", 30), ("dave", 25)]
words = ["banana", "Apple", "cherry"]

sorted(people, key=lambda p: p[1])                # by age; ties keep input order
sorted(people, key=lambda p: (-p[1], p[0]))       # age descending, then name ascending
sorted(words, key=str.lower)                      # case-insensitive

# Multi-pass: because the sort is stable, sort by the SECONDARY key first
people.sort(key=lambda p: p[0])                   # secondary: name
people.sort(key=lambda p: p[1], reverse=True)     # primary: age descending
```

When a key isn't enough: comparators

Some orders can't be expressed as a key. **Largest Number (LC 179)** asks for the arrangement of numbers forming the largest concatenation: `[3, 30, 34, 5, 9]` → `"9534330"`. The right order between two numbers a and b depends on both: put a first iff `a + b > b + a` as strings.

```
from functools import cmp_to_key


def largest_number(nums: list[int]) -> str:
    strs = [str(x) for x in nums]

    def compare(a: str, b: str) -> int:
        if a + b > b + a:
            return -1           # a goes first
        if a + b < b + a:
            return 1
        return 0

    strs.sort(key=cmp_to_key(compare))
    result = "".join(strs)
    return "0" if result[0] == "0" else result
```

A comparator must define a consistent order

Sorting algorithms assume the comparison is **transitive**: if a precedes b and b precedes c, then a precedes c. An inconsistent comparator doesn't raise an error; it silently produces garbage. (The concatenation order above *is* transitive, but that requires proof — it's not obvious!) When writing a comparator, ask whether it is really comparing some underlying quantity.

Incremental Insertion: Insertion Sort

The simplest design strategy is **incremental**: solve the problem for the first $n-1$ items, then add the n-th. For sorting, that means inserting each new element into its place in an already-sorted prefix.

```
def insertion_sort(a: list[int]) -> None:
    for i in range(1, len(a)):
        x = a[i]
        j = i - 1
        while j >= 0 and a[j] > x:
            a[j + 1] = a[j]          # shift larger elements right
            j -= 1
        a[j + 1] = x
```

$O(n^2)$ in the worst case, but $O(n + d)$ where d is the number of **inversions** (out-of-order pairs). On nearly-sorted data it is linear, which is why hybrid sorts use it for small or nearly-sorted pieces.

The incremental idea gets faster with a better data structure. Insert into a balanced search tree instead of an array, and each insertion costs $O(\log n)$; an in-order traversal then reads out the sorted order. That is $O(n \log n)$ sorting by incremental insertion.

Heapsort: Selection Sort with the Right Data Structure

Selection sort repeatedly finds the smallest remaining element. Finding it by scanning costs $O(n)$ per step, $O(n^2)$ total. But "repeatedly extract the minimum" is exactly the operation a **priority queue** supports — a

binary heap does it in $O(\log n)$. Same algorithm, better data structure: $O(n \log n)$.

```
import heapq


def heapsort(a: list[int]) -> list[int]:
    heap = a[:]
    heapq.heapify(heap)                                  # O(n)
    return [heapq.heappop(heap) for _ in range(len(heap))]
```

Heapsort is worst-case $O(n \log n)$, can run in place with $O(1)$ extra space, and is not stable. Heaps get a full treatment in Heaps.

> **Take-Home Lesson**
>
> Selection sort and heapsort are the *same* algorithm. The difference between $O(n^2)$ and $O(n \log n)$ is entirely the data structure used to find the minimum. Many "new" algorithms are old algorithms with a better data structure plugged in.

Mergesort: Divide and Conquer

Split the array in half, sort each half recursively, and **merge** the two sorted halves. The smallest remaining element overall is always at the front of one of the two halves, so merging is a linear two-pointer scan.

```
def merge_sort(a: list[int]) -> list[int]:
    if len(a) <= 1:
        return a
    mid = len(a) // 2
    left, right = merge_sort(a[:mid]), merge_sort(a[mid:])
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:          # <= keeps the sort stable
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged
```

Analysis. Picture the recursion tree. Level k has 2^k subproblems of size $n/2^k$; merging them all costs $O(n)$ because every element appears in exactly one subproblem per level. There are $\log_2 n$ levels. Total: $\Theta(n \log n)$ in **every** case — best, average, and worst. Formally, $T(n) = 2T(n/2) + \Theta(n)$, case 2 of the master theorem.

Properties:

- **Stable**, as long as the merge takes from the left on ties.
- **Needs $O(n)$ auxiliary space** for arrays — merging in place without a buffer would overwrite unmerged elements.
- **Ideal for linked lists**: merging two lists just relinks nodes, with no extra space and no need for random access (LC 148).
- **Ideal for external sorting**: data too large for memory is sorted in chunks, then the sorted chunks are

merged by streaming them sequentially.

Piggybacking on the merge: counting inversions

Mergesort's real value in interviews is that the merge step sees every "cross" pair — one element from the left half, one from the right — in sorted order. Any quantity summed over cross pairs can be computed during the merge for free.

Count inversions: how many pairs $i < j$ have $a_i > a_j$? When the merge takes `right[j]` before the remaining `left[i:]`, then `right[j]` is smaller than every one of those `len(left) - i` elements, all of which were originally to its left.

```
def count_inversions(a: list[int]) -> int:
    def sort_count(lo: int, hi: int) -> int:
        if hi - lo <= 1:
            return 0
        mid = (lo + hi) // 2
        count = sort_count(lo, mid) + sort_count(mid, hi)
        merged, i, j = [], lo, mid
        while i < mid and j < hi:
            if a[i] <= a[j]:
                merged.append(a[i])
                i += 1
            else:
                merged.append(a[j])
                count += mid - i          # a[j] jumps ahead of a[i:mid]
                j += 1
        merged += a[i:mid] + a[j:hi]
        a[lo:hi] = merged
        return count

    return sort_count(0, len(a))
```

Brute force is $O(n^2)$; this is $O(n \log n)$. The same technique solves Reverse Pairs (LC 493) and Count of Smaller Numbers After Self (LC 315).

LC 315 wants the count **per element**, so two things change. Sort the *indices* rather than the values, so each count can be credited to the element's original position. And count from the left element's point of view: when `left[i]` is placed, the `j - mid` right elements already placed are exactly the smaller ones to its right.

```
def count_smaller(nums: list[int]) -> list[int]:
    counts = [0] * len(nums)
    idx = list(range(len(nums)))                # sorted by value as the merges proceed

    def sort(lo: int, hi: int) -> None:
        if hi - lo <= 1:
            return
        mid = (lo + hi) // 2
        sort(lo, mid)
        sort(mid, hi)
        merged, i, j = [], lo, mid
        while i < mid or j < hi:
            if j == hi or (i < mid and nums[idx[i]] <= nums[idx[j]]):
                counts[idx[i]] += j - mid       # right elements already placed are smaller
                merged.append(idx[i])
                i += 1
```

```
            else:
                merged.append(idx[j])
                j += 1
        idx[lo:hi] = merged

    sort(0, len(nums))
    return counts
```

Taking the left element on ties (`<=`) keeps equal values out of the count, so only strictly smaller elements are counted.

Quicksort: Sorting by Randomization

Pick a **pivot** element p. Partition the array into elements less than p, then p, then elements greater than or equal to p. Now p is in its final position, and no element will ever cross from one side to the other — so sort each side independently, recursively.

```
import random


def quicksort(a: list[int], lo: int = 0, hi: int | None = None) -> None:
    if hi is None:
        hi = len(a) - 1

    if lo >= hi:
        return

    p = partition(a, lo, hi)
    quicksort(a, lo, p - 1)
    quicksort(a, p + 1, hi)


def partition(a: list[int], lo: int, hi: int) -> int:
    """Lomuto partition around a random pivot; returns the pivot's final index."""
    r = random.randint(lo, hi)
    a[r], a[hi] = a[hi], a[r]            # move the random pivot to the end
    pivot = a[hi]
    first_high = lo                      # a[lo:first_high] < pivot

    for i in range(lo, hi):
        if a[i] < pivot:
            a[i], a[first_high] = a[first_high], a[i]
            first_high += 1
    a[first_high], a[hi] = a[hi], a[first_high]
    return first_high
```

The partition maintains three regions: `a[lo:first_high]` is less than the pivot, `a[first_high:i]` is at least the pivot, and `a[i:hi]` is unexplored. It's the reader–writer pattern from the two-pointers chapter.

How fast is it?

Like mergesort, quicksort does $O(n)$ work per level of recursion, so its running time is $O(n \cdot h)$ where h is the height of the recursion tree. Everything depends on the pivots.

- **Best case:** every pivot is the median, the tree has height $\log_2 n$, and quicksort is $O(n \log n)$.
- **Worst case:** every pivot is the smallest or largest element, each level peels off just one element, the height is n, and quicksort is $\Theta(n^2)$ — it has turned into selection sort.

The expected case. Call a pivot *good enough* if it lies in the middle half of the sorted order (between the 25th and 75th percentiles). Half of all elements are good enough, so a random pivot is good enough with probability 1/2. A good-enough pivot leaves at most $3n/4$ elements on the bigger side. Along any root-to-leaf path, the subproblem shrinks by a factor of 3/4 at least every other level on average, so it reaches size 1 after about $2\log_{4/3} n = O(\log n)$ levels. Hence $O(n \log n)$ expected. (A careful analysis gives about $2n \ln n \approx 1.39\, n \log_2 n$ comparisons.)

Why randomize?

With a *fixed* pivot rule, such as "always use the last element," there is a specific input that forces the worst case — and for the last-element rule, that input is an already-sorted array, which is extremely common in practice.

Choosing the pivot **at random** changes the nature of the guarantee. There is no longer a bad *input*; there are only bad *coin flips*, and they are astronomically unlikely. Randomized quicksort runs in $O(n \log n)$ expected time **on every input**.

Take-Home Lesson

Randomization makes an algorithm's performance depend on its own coin flips rather than on the input. It is a powerful way to defeat worst-case inputs, and it often produces algorithms far simpler than their deterministic counterparts. When quicksort "goes quadratic," nothing mysterious happens — you just lost a lottery you were overwhelmingly likely to win.

Stop and Think: Nuts and bolts

Problem: You have n bolts of different widths and their n matching nuts. You can try a nut on a bolt and learn whether the nut is too large, too small, or a match — but you cannot compare two nuts or two bolts directly. Match every bolt to its nut efficiently.
Solution: Trying each bolt against every remaining nut is $O(n^2)$. To do better, emulate quicksort. Pick a random bolt b and test every nut against it, partitioning the nuts into smaller-than-b, larger-than-b, and its match m. Now use m as a pivot to partition the **bolts** the same way. In $2n - 1$ tests we've matched one pair and split the problem into two independent subproblems, exactly as quicksort does. Expected time: $O(n \log n)$.
Remarkably, no simple *deterministic* $O(n \log n)$ algorithm is known for this problem. Randomization makes the bad cases disappear, leaving a short and elegant algorithm.

Many duplicates: three-way partitioning

If the array is all copies of one value, the partition above puts everything on one side: quadratic again. The fix is to partition into **three** regions: less than, equal to, and greater than the pivot. Equal elements are finished and never recursed on. This is the **Dutch national flag** problem (LC 75, Sort Colors):

```
def three_way_partition(a: list[int], pivot: int) -> tuple[int, int]:
    """Rearrange a into  < pivot | == pivot | > pivot ; return the middle bounds."""
    lt, i, gt = 0, 0, len(a)      # a[:lt] < p, a[lt:i] == p, a[gt:] > p
    while i < gt:
        if a[i] < pivot:
            a[lt], a[i] = a[i], a[lt]
            lt += 1
```

```
            i += 1
        elif a[i] > pivot:
            gt -= 1
            a[gt], a[i] = a[i], a[gt]   # don't advance i: new a[i] is unexamined
        else:
            i += 1
    return lt, gt
```

Is quicksort really quick?

Mergesort, heapsort, and quicksort are all $\Theta(n \log n)$, so asymptotic analysis can't rank them. In practice a well-implemented quicksort is typically two to three times faster than the others: its inner loop is tiny and it scans memory sequentially, which caches love. When algorithms tie asymptotically, only experiments can break the tie.

Selection: The k-th Smallest Element

To find the median, or the k-th largest element (LC 215), sorting costs $O(n \log n)$ — but it does far more work than needed. **Quickselect** partitions like quicksort, then recurses into **only the side containing position** k:

```
def quickselect(a: list[int], k: int) -> int:
    """Return the k-th smallest element (0-indexed). Rearranges a."""
    lo, hi = 0, len(a) - 1
    while True:
        p = partition(a, lo, hi)
        if p == k:
            return a[p]
        if p < k:
            lo = p + 1
        else:
            hi = p - 1


def find_kth_largest(nums: list[int], k: int) -> int:
    return quickselect(nums[:], len(nums) - k)
```

With good pivots, the work is $n + n/2 + n/4 + \cdots < 2n$ — a geometric series — so quickselect runs in $O(n)$ expected time. (The worst case is $O(n^2)$; the "median of medians" algorithm guarantees $O(n)$ deterministically, but it's rarely worth the complexity.) For the k largest elements when $k \ll n$, a size-k heap in $O(n \log k)$ is often simpler; see Heaps.

The Lower Bound: Why $n \log n$?

Mergesort, heapsort, and quicksort all take $\Theta(n \log n)$. Is a linear-time sort possible?

Not if the algorithm learns about the input only by **comparing** pairs of elements. Any such algorithm can be drawn as a **decision tree**: each internal node is a comparison, each branch an outcome, and each leaf a final ordering. The algorithm must reach a *different* leaf for each of the $n!$ input permutations — if two permutations led to the same leaf, the same rearrangement would be applied to both, and at least one would

come out unsorted. A binary tree with $n!$ leaves has height at least $\log_2(n!)$, and

$$\log_2(n!) = \sum_{i=1}^{n} \log_2 i \;\geq\; \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$$

So every comparison sort needs $\Omega(n \log n)$ comparisons in the worst case. Mergesort and heapsort are **optimal**. The same argument gives $\Omega(n \log n)$ lower bounds for problems that sorting solves, such as element uniqueness in the comparison model.

The only way around the bound is to **not compare**: to use the values themselves as addresses. That's what distribution sorts do.

Distribution Sorts: Beating $n \log n$

To sort names for a phone book, you might first split them into 26 piles by first letter. Every name in pile J belongs after every name in pile I, so each pile can be sorted independently and the piles concatenated. This is **bucketing**, and when keys are small integers or evenly distributed, it sorts in linear time.

Counting sort

When keys are integers in a small range $[0, k)$, count occurrences of each key and read them back in order. $O(n + k)$ time.

```python
def counting_sort(a: list[int], k: int) -> list[int]:
    """Sort integers in range [0, k)."""
    counts = [0] * k
    for x in a:
        counts[x] += 1
    result = []
    for value, c in enumerate(counts):
        result.extend([value] * c)
    return result
```

For sorting *records* by an integer key, the stable version computes prefix sums of the counts to find each key's starting position, then places records in input order. That stable version is what radix sort needs.

Radix sort

To sort d-digit numbers, run a **stable** counting sort on each digit from least to most significant. Stability is essential: after sorting by digit i, ties are broken by the previous passes on lower digits. Total: $O(d \cdot (n + b))$ for base b. For 32-bit integers with base 2^{16}, that's two linear passes.

Bucket sort

For real numbers roughly **uniformly distributed** in $[0, 1)$, create n buckets, put x in bucket $\lfloor nx \rfloor$, sort each (tiny) bucket, and concatenate. Expected $O(n)$.

> **⚠ Bucketing bets on the distribution**
>
> Bucket-based methods are fast when the data is spread out as expected and terrible when it isn't. If most keys land in one bucket, you're back to sorting everything with the fallback algorithm. Real data is often wildly non-uniform: surnames cluster regionally, timestamps cluster at business hours, and

> adversarial inputs cluster deliberately. Balanced trees and comparison sorts give guarantees on *every* input; bucketing gives them only on the inputs you expected.

Bucketing in interviews

Distribution-sort ideas appear more often than distribution sorts themselves:

- **Top K Frequent Elements (LC 347):** a frequency is an integer in $[1, n]$, so bucket elements by frequency and read buckets from the top: $O(n)$.
- **H-Index (LC 274):** citations above n can be capped at n, then counting sort.
- **Maximum Gap (LC 164):** with n numbers spread over range R, the maximum gap is at least $R/(n-1)$ by the pigeonhole principle. With buckets of that width, the maximum gap can't occur *inside* a bucket, only *between* adjacent non-empty buckets. So only each bucket's min and max matter: $O(n)$.
- **Sort Characters by Frequency (LC 451):** 26 or 128 possible characters make any key small.

Maximum Gap is the one with fiddly details. Rounding the bucket width *down* keeps it at or below the smallest possible maximum gap. A gap inside a bucket is then smaller than the answer, so it can be ignored:

```python
def maximum_gap(nums: list[int]) -> int:
    lo, hi = min(nums), max(nums)
    if len(nums) < 2 or lo == hi:
        return 0
    width = max(1, (hi - lo) // (len(nums) - 1))
    count = (hi - lo) // width + 1
    mins, maxs = [float("inf")] * count, [float("-inf")] * count
    for x in nums:
        b = (x - lo) // width
        mins[b], maxs[b] = min(mins[b], x), max(maxs[b], x)
    best, prev_max = 0, lo
    for b in range(count):
        if mins[b] != float("inf"):                # skip empty buckets
            best = max(best, mins[b] - prev_max)
            prev_max = maxs[b]
    return best
```

Python's Built-in Sort: Timsort

`list.sort()` and `sorted()` use **Timsort**, a hybrid of mergesort and insertion sort designed for real-world data:

- It scans for existing sorted runs ("natural runs") and merges them, so it is $O(n)$ **on already-sorted or reverse-sorted input** and fast on partially sorted input.
- Short runs are extended with binary insertion sort.
- When one run keeps "winning" during a merge, it switches to galloping (exponential search) to skip ahead.
- It is **stable**, worst-case $O(n \log n)$, and uses $O(n)$ auxiliary space.

Use it. A hand-written sort in pure Python is typically 10–100× slower than the built-in, which runs in C.

Comparison Table

Algorithm	Best	Average	Worst	Extra space	Stable	Notes
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	$O(n +$ inversions); great for small / nearly sorted
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Minimal number of swaps
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Guaranteed bound, in place
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Linked lists, external sorting, inversions
Quicksort (random pivot)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Fastest in practice
Timsort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Python's built-in
Counting sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Yes*	Integer keys in $[0, k)$
Radix sort	$O(d(n + b))$	$O(d(n + b))$	$O(d(n + b))$	$O(n + b)$	Yes	d digits in base b
Bucket sort	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$	Yes*	Assumes uniform distribution

* When implemented with the stable placement pass.

Common Mistakes

1. **Sorting when you don't need order.** Sorting is $O(n \log n)$; a hash set answers "have I seen this?" in $O(n)$ total. Sort when the problem involves *order* or *nearness*; hash when it involves only *equality*.
2. **Forgetting that sorting destroys positions.** If you need original indices afterward, sort `range(n)` by key, or sort `(value, index)` pairs.
3. **Mutating instead of copying.** `list.sort()` sorts in place and returns `None`; `x = arr.sort()` sets `x` to `None`. Use `sorted(arr)` for a new list.
4. **Inconsistent comparators.** A comparator that isn't transitive produces an unspecified order without any error.
5. **Deterministic pivots.** Quicksort or quickselect with a first- or last-element pivot is quadratic on sorted input — a common test case. Randomize.

6. **Ignoring duplicates in quicksort.** Two-way partitioning degrades to $O(n^2)$ on arrays with few distinct values; use three-way partitioning.
7. **Assuming bucketing is always linear.** It is linear only when the keys are small integers or evenly distributed.

Practice Problems

Problem	Idea
Sort Colors (LC 75)	Three-way partition
Largest Number (LC 179)	Custom comparator
Sort List (LC 148)	Mergesort on a linked list
Kth Largest Element in an Array (LC 215)	Quickselect or size-k heap
Top K Frequent Elements (LC 347)	Bucket by frequency
H-Index (LC 274)	Counting sort with capping
Maximum Gap (LC 164)	Pigeonhole buckets
Count of Smaller Numbers After Self (LC 315)	Mergesort with counting
Reverse Pairs (LC 493)	Mergesort with counting
Wiggle Sort II (LC 324)	Median via quickselect + three-way partition
Sort an Array (LC 912)	Implement mergesort / randomized quicksort

Linked Lists

Data structures come in two fundamental flavors. **Contiguous** structures — arrays, dynamic arrays, heaps, hash tables — live in a single block of memory. **Linked** structures — lists, trees, graph adjacency lists — are separate chunks of memory tied together by pointers. Nearly every data structure you will ever use is built from one of these two, or both.

The linked list is the simplest linked structure, and it is worth studying less for its own sake (in Python, you'll rarely choose one) than for what it teaches: how to reason about pointer manipulation, how to exploit recursive structure, and how the choice between contiguous and linked storage makes some operations cheap and others expensive.

Contiguous vs. Linked

	Array (contiguous)	**Linked list**
Access the i-th element	$O(1)$: compute the address	$O(i)$: follow i pointers
Insert / delete at a **known position**	$O(n)$: shift elements	$O(1)$: rewire a couple of pointers
Insert / delete at the front	$O(n)$	$O(1)$
Append at the end	$O(1)$ amortized (dynamic array)	$O(1)$ with a tail pointer
Memory overhead	None beyond the data	One or two pointers per element
Cache behavior	Excellent: sequential memory	Poor: every hop may be a cache miss
Growth	Occasional $O(n)$ copy on resize	Never needs to move existing elements

Arrays win on random access, space, and locality. Linked lists win when you need to **splice**: insert or remove elements in the middle, given a reference to where, without disturbing anything else. They also let you move large records around by moving pointers instead of copying the records themselves.

> **Python's `list` is an array**
>
> Python's `list` is a dynamic array, and `collections.deque` is a doubly linked list of fixed-size blocks, giving $O(1)$ operations at both ends. In real Python code, explicit node-based linked lists appear mainly inside other structures — the LRU cache below, `OrderedDict`'s internals — where $O(1)$ removal of a node you already hold is exactly what's needed.

The same operations, different costs

It's illuminating to compare how each list variant supports the basic dictionary operations. Assume we're handed a reference to the node x for delete / successor / predecessor.

Operation	**Singly, unsorted**	**Singly, sorted**	**Doubly, unsorted**	**Doubly, sorted**
Search(k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Operation	Singly, unsorted	Singly, sorted	Doubly, unsorted	Doubly, sorted
Insert(x)	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Delete(x)	$O(n)$ †	$O(n)$ †	$O(1)$	$O(1)$
Successor(x)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Predecessor(x)	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Minimum / Maximum	$O(n)$	$O(1)$	$O(n)$	$O(1)$

A few entries deserve explanation:

- **Sorting a list doesn't speed up search.** Binary search needs the middle element in $O(1)$, and a list can't provide it.
- **Deletion in a singly linked list needs the predecessor**, since it's the predecessor's `next` that must change. Given only x, finding the predecessor takes a linear scan. A doubly linked list stores it directly.
- † **Deleting from a singly linked list in $O(1)$ is possible with a trick:** instead of removing node x, copy its *successor's* value into x and remove the successor. This fails for the last node, and it changes which node object holds which value — which matters if other code holds references to nodes. It's LC 237, Delete Node in a Linked List.

> **Take-Home Lesson**
>
> Data structure design is a balancing act between operations. The fastest structure for operations A *and* B is often not the fastest for A alone or B alone. Pick representations by listing the operations your algorithm actually performs, and how often.

Lists Are Recursive Objects

Remove the first node of a linked list and what remains is — a smaller linked list. This recursive view is often the cleanest way to design list algorithms: handle the head, recurse on the rest, and trust induction for correctness.

```
from __future__ import annotations

from dataclasses import dataclass


@dataclass
class ListNode:
    val: int = 0
    next: ListNode | None = None


def length(head: ListNode | None) -> int:
    return 0 if head is None else 1 + length(head.next)


def merge_two_lists(a: ListNode | None, b: ListNode | None) -> ListNode | None:
    """LC 21: merge two sorted lists by relinking their nodes."""
    if a is None or b is None:
        return a or b
```

```
    if a.val <= b.val:
        a.next = merge_two_lists(a.next, b)
        return a
    b.next = merge_two_lists(a, b.next)
    return b
```

`merge_two_lists` is correct by induction: the smaller head must come first, and assuming the recursive call correctly merges what remains, attaching it after that head gives a correctly merged list.

The recursive versions are elegant but use $O(n)$ stack space, and Python's recursion limit (about 1000) makes them fail on long lists. In practice, write the iterative version — with the recursive argument in mind.

Core Techniques

1. The dummy (sentinel) head

Many list algorithms have an annoying special case: the head itself might change (it gets deleted, or a new node goes before it). A **dummy node** placed before the head turns the head into an ordinary node, so the special case disappears.

```
def merge_two_lists_iter(a: ListNode | None, b: ListNode | None) -> ListNode | None:
    dummy = tail = ListNode()
    while a and b:
        if a.val <= b.val:
            tail.next, a = a, a.next
        else:
            tail.next, b = b, b.next
        tail = tail.next
    tail.next = a or b
    return dummy.next


def remove_elements(head: ListNode | None, val: int) -> ListNode | None:
    """LC 203: delete every node with the given value."""
    dummy = ListNode(0, head)
    prev = dummy
    while prev.next:
        if prev.next.val == val:
            prev.next = prev.next.next      # splice out; don't advance prev
        else:
            prev = prev.next
    return dummy.next
```

Note how `remove_elements` walks with `prev` rather than the current node: in a singly linked list, you delete a node by standing on its predecessor.

2. Rewire carefully: save before you overwrite

Every pointer manipulation bug has the same shape: a `next` field is overwritten while it was still the only path to the rest of the list. Before assigning to `x.next`, ask: *is anything reachable only through the old value?* If so, save it first.

Reversal is the canonical example:

```python
def reverse_list(head: ListNode | None) -> ListNode | None:
    """LC 206."""
    prev = None
    curr = head
    while curr:
        nxt = curr.next          # 1. save the rest of the list
        curr.next = prev         # 2. reverse this one pointer
        prev, curr = curr, nxt   # 3. advance both
    return prev
```

Invariant: `prev` is the head of the already-reversed prefix; `curr` is the head of the untouched suffix. Each iteration moves one node from the suffix to the front of the prefix. When `curr` is `None`, `prev` is the whole reversed list.

When in doubt, draw three boxes and arrows on paper and trace one iteration. Pointer code is much easier to get right by picture than by reasoning in your head.

3. Runners: two pointers at different speeds

Arrays let you jump to the middle; lists don't. But two pointers moving at different speeds can compute positions in a single pass.

Find the middle. `slow` moves one step, `fast` moves two. When `fast` reaches the end, `slow` has gone half as far.

```python
def middle_node(head: ListNode) -> ListNode:
    """LC 876: second middle for even lengths."""
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

The *n*-th node from the end (LC 19). Advance `fast` *n* steps first, then move both at the same speed. When `fast` falls off the end, `slow` is *n* nodes behind it. Starting both at a dummy node leaves `slow` at the *predecessor* of the target, which is exactly where you need to stand to delete it.

```python
def remove_nth_from_end(head: ListNode | None, n: int) -> ListNode | None:
    dummy = ListNode(0, head)
    slow = fast = dummy
    for _ in range(n):
        fast = fast.next
    while fast.next:
        slow, fast = slow.next, fast.next
    slow.next = slow.next.next
    return dummy.next
```

Cycle Detection: Floyd's Tortoise and Hare

A list might loop back on itself. A `set` of visited nodes detects this in $O(n)$ space, but two runners do it in $O(1)$.

Detection (LC 141). Move `slow` one step and `fast` two. If there's no cycle, `fast` reaches the end. If there is, both eventually enter the cycle, and from then on `fast` gains exactly one node per step on `slow` — so the

gap between them shrinks by one each step and must hit zero. They meet within one lap.

Finding the entrance (LC 142). Let a be the distance from the head to the cycle's entrance, L the cycle length, and suppose they meet b steps past the entrance. When they meet, `slow` has walked $a + b$ and `fast` has walked twice that. The extra distance `fast` covered is a whole number of laps:

$$2(a+b) - (a+b) = a + b = kL \quad \Longrightarrow \quad a = kL - b$$

So from the meeting point, walking a more steps means walking $kL - b$ steps — which lands exactly on the entrance (you're b past it; $L - b$ more steps completes the lap, and the extra $(k-1)$ laps change nothing). A pointer starting from the head also reaches the entrance after a steps. Walk both at the same speed; they meet at the entrance.

```
def detect_cycle(head: ListNode | None) -> ListNode | None:
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast:
            finder = head
            while finder is not slow:
                finder, slow = finder.next, slow.next
            return finder
    return None
```

The same algorithm works on any function iterated from a starting point, $x \to f(x) \to f(f(x)) \to \cdots$, which is how it solves Find the Duplicate Number (LC 287) with $f(i) = \text{nums}[i]$ and Happy Number (LC 202).

Composing Primitives

Harder list problems are usually a few of the primitives above, glued together. Recognizing the pieces is most of the work.

Reorder list (LC 143): middle + reverse + merge

Rearrange $L_0 \to L_1 \to \cdots \to L_n$ into $L_0 \to L_n \to L_1 \to L_{n-1} \to \cdots$.

```
def reorder_list(head: ListNode | None) -> None:
    if not head or not head.next:
        return
    # 1. split at the middle
    slow, fast = head, head.next
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
    second, slow.next = slow.next, None
    # 2. reverse the second half
    second = reverse_list(second)
    # 3. interleave the two halves
    first = head
    while second:
        first_rest, second_rest = first.next, second.next   # save both before rewiring
        first.next = second
        second.next = first_rest
```

```
        first, second = first_rest, second_rest
```

The same split-and-reverse approach checks whether a list is a palindrome in $O(1)$ space (LC 234): reverse the second half and compare it against the first.

Reverse a sublist (LC 92)

Reverse nodes at positions `left` through `right`. Walk a pointer to the node *before* the sublist, then repeatedly take the node right after the sublist's current first node and move it to the front:

```
def reverse_between(head: ListNode | None, left: int, right: int) -> ListNode | None:
    dummy = ListNode(0, head)
    before = dummy
    for _ in range(left - 1):
        before = before.next
    first = before.next                  # will end up as the sublist's tail
    for _ in range(right - left):
        moved = first.next
        first.next = moved.next
        moved.next = before.next
        before.next = moved
    return dummy.next
```

Reverse in groups of k (LC 25)

The same move-to-front, applied to successive blocks of k. Two details: first check that k nodes remain (a short final block stays as is), and after reversing, the block's original first node is its new tail — the `before` of the next block.

```
def reverse_k_group(head: ListNode | None, k: int) -> ListNode | None:
    dummy = ListNode(0, head)
    before = dummy
    while True:
        probe = before
        for _ in range(k):                   # are there k nodes left?
            probe = probe.next
            if probe is None:
                return dummy.next
        first = before.next
        for _ in range(k - 1):               # move-to-front, as in LC 92
            moved = first.next
            first.next = moved.next
            moved.next = before.next
            before.next = moved
        before = first
```

$O(n)$ time: each node is probed once and moved once. $O(1)$ space.

Sort a list (LC 148): mergesort

Mergesort is the natural sort for linked lists: splitting uses the middle-finding runners, merging relinks nodes with no extra space, and nothing needs random access. See Sorting.

```
def sort_list(head: ListNode | None) -> ListNode | None:
    if not head or not head.next:
```

```
        return head
    slow, fast = head, head.next             # slow stops at the end of the first half
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
    second, slow.next = slow.next, None      # cut, or the halves stay connected
    return merge_two_lists_iter(sort_list(head), sort_list(second))
```

$O(n \log n)$ time, $O(\log n)$ recursion depth. Starting `fast` at `head.next` matters: with `fast = head`, a two-node list splits into two nodes and nothing, and the recursion never ends.

Merge k sorted lists (LC 23)

Merging lists pairwise, one after another, costs $O(nk)$ for n total nodes. Two better approaches, both $O(n \log k)$: keep the k current heads in a **min-heap** and repeatedly pop the smallest (see Heaps), or merge lists in pairs like the levels of mergesort. With nodes, push `(node.val, i, node)`: nodes aren't orderable, and the list index `i` breaks ties before Python ever compares two nodes.

More Classic Problems

Intersection of two lists (LC 160)

Two lists merge at some node and share a tail. Let the unshared parts have lengths a and b, and the shared tail c. A pointer that walks list A and then switches to list B travels $a + c + b$ before reaching the intersection a second time; one that walks B then A travels $b + c + a$. Same distance — so they arrive at the intersection at the same step (or both reach `None` together if there is none).

```
def get_intersection_node(a: ListNode | None, b: ListNode | None) -> ListNode | None:
    p, q = a, b
    while p is not q:
        p = p.next if p else b
        q = q.next if q else a
    return p
```

Add two numbers (LC 2)

Digits stored in reverse order are exactly the order in which grade-school addition processes them, least significant first. Walk both lists together, carrying as you go; a dummy head collects the result.

```
def add_two_numbers(l1: ListNode | None, l2: ListNode | None) -> ListNode | None:
    dummy = tail = ListNode()
    carry = 0
    while l1 or l2 or carry:
        total = carry + (l1.val if l1 else 0) + (l2.val if l2 else 0)
        carry, digit = divmod(total, 10)
        tail.next = ListNode(digit)
        tail = tail.next
        l1 = l1.next if l1 else None
        l2 = l2.next if l2 else None
    return dummy.next
```

Copy a list with random pointers (LC 138)

Each node has an extra `random` pointer to an arbitrary node. The difficulty: when copying node x, x's random target may not have been copied yet. The clean fix is a **hash map from original node to copy**: one pass creates all the copies, a second pass wires `next` and `random` through the map. $O(n)$ time and space. (An $O(1)$-extra-space version interleaves each copy right after its original, uses `x.next.random = x.random.next`, then unweaves the lists.)

Design: The LRU Cache (LC 146)

An LRU (least-recently-used) cache with capacity c supports `get(key)` and `put(key, value)` in $O(1)$, evicting the least recently used key when full.

No single structure does this:

- A **hash map** finds a key in $O(1)$, but knows nothing about recency.
- A **list ordered by recency** can move an item to the front and evict from the back in $O(1)$ — *if* you already hold a reference to the node, and *if* it's doubly linked so that the node can unlink itself.

Combine them: the hash map stores, for each key, a pointer to its node in a doubly linked recency list. The map provides the reference; the list provides the $O(1)$ splicing.

```python
class _Node:
    __slots__ = ("key", "val", "prev", "next")

    def __init__(self, key: int = 0, val: int = 0):
        self.key, self.val = key, val
        self.prev: _Node | None = None
        self.next: _Node | None = None


class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.nodes: dict[int, _Node] = {}
        self.head, self.tail = _Node(), _Node()   # sentinels: head.next is most recent
        self.head.next, self.tail.prev = self.tail, self.head

    def _unlink(self, node: _Node) -> None:
        node.prev.next, node.next.prev = node.next, node.prev

    def _push_front(self, node: _Node) -> None:
        node.prev, node.next = self.head, self.head.next
        self.head.next.prev = node
        self.head.next = node

    def get(self, key: int) -> int:
        node = self.nodes.get(key)
        if node is None:
            return -1
        self._unlink(node)
        self._push_front(node)
        return node.val
```

```
    def put(self, key: int, value: int) -> None:
        if key in self.nodes:
            node = self.nodes[key]
            node.val = value
            self._unlink(node)
        else:
            if len(self.nodes) == self.capacity:
                lru = self.tail.prev
                self._unlink(lru)
                del self.nodes[lru.key]            # why nodes store their key
            node = _Node(key, value)
            self.nodes[key] = node
        self._push_front(node)
```

Two sentinels (`head` and `tail`) mean `_unlink` and `_push_front` never check for `None` — no special cases for the first or last element.

> **Take-Home Lesson**
>
> Building algorithms around the right combination of data structures leads to both clean code and good performance. When no single structure supports all the operations you need, look for two that each support half, and link them with pointers.

In production Python, `collections.OrderedDict` implements exactly this design: `move_to_end(key)` and `popitem(last=False)` give an LRU cache in a few lines. In an interview, be ready to build it by hand.

Common Mistakes

1. **Losing the rest of the list.** Overwriting `node.next` before saving it. Always save first.
2. **Forgetting the head can change.** Deleting the first node, inserting before it, or reversing changes which node is the head. Use a dummy node, and return `dummy.next`.
3. **Null-pointer errors in runners.** `fast.next.next` fails if `fast.next` is `None`. The condition `while fast and fast.next` protects both hops.
4. **Comparing values instead of identities.** Two different nodes can hold equal values. Use `is` to compare nodes (intersection, cycle detection), `==` only for values.
5. **Leaving a cycle behind.** After splitting a list, set the end of the first half to `None`; otherwise later code will traverse into the second half or loop forever.
6. **Recursion depth.** A recursive solution on a 10^5-node list will exceed Python's recursion limit. Prefer iteration.

Practice Problems

Problem	Technique
Reverse Linked List (LC 206)	Three-pointer reversal
Merge Two Sorted Lists (LC 21)	Dummy head

Problem	Technique
Remove Nth Node From End (LC 19)	Runners with a gap
Middle of the Linked List (LC 876)	Slow / fast
Linked List Cycle II (LC 142)	Floyd + entrance argument
Delete Node in a Linked List (LC 237)	Copy successor into the node
Palindrome Linked List (LC 234)	Middle + reverse + compare
Reorder List (LC 143)	Middle + reverse + merge
Reverse Linked List II (LC 92)	Move-to-front within a sublist
Reverse Nodes in k-Group (LC 25)	Repeated sublist reversal
Intersection of Two Linked Lists (LC 160)	Switch heads, equal distances
Add Two Numbers (LC 2)	Grade-school addition with carry
Copy List with Random Pointer (LC 138)	Hash map old → new
Sort List (LC 148)	Mergesort
Merge k Sorted Lists (LC 23)	Min-heap of heads
LRU Cache (LC 146)	Hash map + doubly linked list

Stacks & Queues

A **container** is a data structure that stores items and hands them back **without regard to their content**. What distinguishes one container from another is the *order* in which items come back out:

- A **stack** returns the most recently inserted item: **last in, first out** (LIFO).
- A **queue** returns the least recently inserted item: **first in, first out** (FIFO).

That one decision — which item comes out next — turns out to shape entire families of algorithms. Stacks are the natural companion of recursion, nesting, and "the most recent unresolved thing." Queues are the natural companion of fairness, simulation, and exploring things in the order they were discovered. And when neither order is what you need, you want a priority queue, which retrieves by content after all (see Heaps).

The Two Orders

LIFO shows up wherever things *nest*. People packed into a crowded elevator leave in LIFO order. Function calls return in LIFO order: the most recently called function finishes first. Parentheses close in LIFO order: the most recently opened one must close first. Undo histories, browser back buttons, and depth-first search all live on stacks.

FIFO shows up wherever things *wait their turn*. It is the fair order for serving requests: it minimizes the maximum time anyone waits. (Interestingly, the *average* waiting time is the same under FIFO and LIFO; what FIFO buys is that nobody is starved.) Breadth-first search, task schedulers, and buffered streams all live on queues.

> **Take-Home Lesson**
>
> If retrieval order doesn't matter to your algorithm, use a stack: it is the simplest and fastest container. Choose a queue when fairness or discovery order matters — most notably in breadth-first search, where FIFO order is exactly what makes shortest paths come out right.

In Python

```
from collections import deque

stack: list[int] = []
stack.append(1)          # push       O(1) amortized
stack.append(2)
top = stack[-1]          # peek       O(1)
stack.pop()              # pop        O(1)

queue: deque[int] = deque()
queue.append(1)          # enqueue    O(1)
queue.append(2)
front = queue[0]         # peek       O(1)
queue.popleft()          # dequeue    O(1)
```

> ⚠ **Never use `list.pop(0)` as a queue**
>
> `list.pop(0)` shifts every remaining element: $O(n)$ per dequeue, $O(n^2)$ for a BFS over n nodes. `deque.popleft()` is $O(1)$. This single substitution is the most common performance bug in Python BFS code.

Implementing Containers

Both containers can be built on arrays or on linked lists; the question is whether you know a size bound in advance.

A queue from two stacks (LC 232)

A stack reverses order; reversing twice restores it. Push onto an `inbox` stack. To dequeue, pop from an `outbox` stack — and when it's empty, pour the entire inbox into it, which reverses the inbox so its oldest element ends up on top.

```
class MyQueue:
    def __init__(self) -> None:
        self.inbox: list[int] = []
        self.outbox: list[int] = []

    def push(self, x: int) -> None:
        self.inbox.append(x)

    def pop(self) -> int:
        self.peek()
        return self.outbox.pop()

    def peek(self) -> int:
        if not self.outbox:
            while self.inbox:
                self.outbox.append(self.inbox.pop())
        return self.outbox[-1]

    def empty(self) -> bool:
        return not self.inbox and not self.outbox
```

A single `pop` can cost $O(n)$ when it triggers the pour. But each element is moved from inbox to outbox **at most once** in its lifetime — pushed once, poured once, popped once. So n operations cost $O(n)$ in total: $O(1)$ **amortized** per operation. (This is the aggregate method from Complexity Analysis.)

A circular buffer (LC 622)

With a fixed capacity, a queue fits in an array whose ends wrap around. Keep the index of the front and the current size; the back is `(front + size) % capacity`.

```
class CircularQueue:
    def __init__(self, capacity: int) -> None:
        self.buf = [0] * capacity
        self.front = 0
        self.size = 0
```

```
    def enqueue(self, x: int) -> bool:
        if self.size == len(self.buf):
            return False
        self.buf[(self.front + self.size) % len(self.buf)] = x
        self.size += 1
        return True

    def dequeue(self) -> int | None:
        if self.size == 0:
            return None
        x = self.buf[self.front]
        self.front = (self.front + 1) % len(self.buf)
        self.size -= 1
        return x
```

Circular buffers are how fixed-size queues are built in practice: no allocation after construction, excellent locality, and $O(1)$ everything. They're the natural structure for "last k events" problems like moving averages and rate limiters.

Stacks and Nesting

Matching brackets (LC 20)

Why is a stack the right structure for checking brackets? Because the rule is exactly LIFO: a closing bracket must match **the most recently opened bracket that is still unclosed**. The stack holds the unclosed brackets, most recent on top.

```
def is_valid(s: str) -> bool:
    pairs = {")": "(", "]": "[", "}": "{"}
    stack: list[str] = []
    for ch in s:
        if ch in pairs:
            if not stack or stack.pop() != pairs[ch]:
                return False
        else:
            stack.append(ch)
    return not stack          # anything left open is an error
```

When there's only one bracket type, the stack degenerates into a **counter** (depth).

Longest Valid Parentheses (LC 32)

Find the longest well-formed substring. Push *indices* rather than characters, and keep one extra index at the bottom of the stack: the position just before the current valid run (initially `-1`, later the last unmatched `)`). Above it sit the indices of unmatched `(`.

```
def longest_valid_parentheses(s: str) -> int:
    stack = [-1]                    # bottom: index just before the current valid run
    best = 0
    for i, ch in enumerate(s):
        if ch == "(":
            stack.append(i)
```

```
        else:
            stack.pop()
            if stack:
                best = max(best, i - stack[-1])
            else:
                stack.append(i)      # unmatched ')': no valid run can cross it
    return best
```

After a `)` pops its match, the new top is the last index that is still unmatched, so everything after it up to `i` is balanced: a valid run of length `i - stack[-1]`. If the pop empties the stack, the `)` had no match, and it becomes the new bottom. $O(n)$ time and space.

Nested state: Decode String (LC 394)

`"3[a2[c]]"` decodes to `"accaccacc"`. Each `[` begins a new, nested context; each `]` ends the current one and splices its result into the enclosing context. Push the enclosing context when entering, pop it when leaving:

```
def decode_string(s: str) -> str:
    stack: list[tuple[str, int]] = []      # (text before '[', repeat count)
    current, num = "", 0
    for ch in s:
        if ch.isdigit():
            num = num * 10 + int(ch)
        elif ch == "[":
            stack.append((current, num))
            current, num = "", 0
        elif ch == "]":
            prefix, repeat = stack.pop()
            current = prefix + current * repeat
        else:
            current += ch
    return current
```

This is exactly what a recursive-descent parser does implicitly with the call stack. Any recursive algorithm can be converted into an iterative one with an explicit stack of "saved contexts" — and vice versa.

Expression evaluation

In **reverse Polish notation** (LC 150), operators follow their operands: `2 1 + 3 *` means $(2 + 1) \times 3$. A stack evaluates it directly — push numbers; on an operator, pop two operands and push the result. There's no need for parentheses or precedence rules, which is why stack machines (the JVM, CPython's bytecode interpreter) use it.

```
def eval_rpn(tokens: list[str]) -> int:
    stack: list[int] = []
    for tok in tokens:
        if tok in {"+", "-", "*", "/"}:
            b, a = stack.pop(), stack.pop()
            if tok == "+":
                stack.append(a + b)
            elif tok == "-":
                stack.append(a - b)
            elif tok == "*":
                stack.append(a * b)
            else:
```

```
                stack.append(int(a / b))     # truncate toward zero
            else:
                stack.append(int(tok))
        return stack[0]
```

For ordinary **infix** expressions with `+ - * /` (LC 227), keep a stack of *terms* to be summed at the end. `+x` and `-x` push a new term; `*x` and `/x` bind tighter, so they modify the top term immediately:

```
def calculate(s: str) -> int:
    stack: list[int] = []
    num, op = 0, "+"
    for i, ch in enumerate(s):
        if ch.isdigit():
            num = num * 10 + int(ch)
        if ch in "+-*/" or i == len(s) - 1:
            if op == "+":
                stack.append(num)
            elif op == "-":
                stack.append(-num)
            elif op == "*":
                stack.append(stack.pop() * num)
            else:
                stack.append(int(stack.pop() / num))
            num, op = 0, ch
    return sum(stack)
```

With parentheses and only `+ -` (LC 224), push the running result and the sign in front of the `(`, start a fresh result inside, and combine when you see `)` — the same "save the enclosing context" idea as Decode String:

```
def calculate_with_parens(s: str) -> int:
    stack: list[tuple[int, int]] = []        # (result before '(', sign in front of '(')
    result, num, sign = 0, 0, 1
    for ch in s:
        if ch.isdigit():
            num = num * 10 + int(ch)
        elif ch in "+-":
            result += sign * num
            num, sign = 0, (1 if ch == "+" else -1)
        elif ch == "(":
            stack.append((result, sign))
            result, sign = 0, 1
        elif ch == ")":
            result += sign * num
            prev, prev_sign = stack.pop()
            result, num = prev + prev_sign * result, 0
    return result + sign * num
```

Unary minus needs no special case: `-(2+3)` adds `+0`, then applies sign `-1` to the parenthesized group.

Augmented Stacks: Min Stack (LC 155)

Support `push`, `pop`, `top`, and `get_min`, all in $O(1)$.

The trouble with tracking a single minimum is that popping the minimum leaves you not knowing the next

one. The fix exploits LIFO order: the elements *below* any stack entry never change while that entry is on the stack. So each entry can permanently record **the minimum of itself and everything below it**.

```
class MinStack:
    def __init__(self) -> None:
        self.stack: list[tuple[int, int]] = []     # (value, min of stack up to here)

    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:
        self.stack.pop()

    def top(self) -> int:
        return self.stack[-1][0]

    def get_min(self) -> int:
        return self.stack[-1][1]
```

The idea generalizes to any associative aggregate (max, sum, gcd): augment each entry with the aggregate of the stack beneath it. Combined with the two-stacks queue above, it even gives a **queue** with $O(1)$ amortized min — an alternative to the monotonic deque.

The Monotonic Stack

Many problems ask, for every element, about the **nearest element to one side that is larger (or smaller)**: the next warmer day, the previous taller building, the first smaller bar to the left. Brute force scans outward from each element: $O(n^2)$.

The monotonic stack answers all of these queries in a single $O(n)$ pass. Scan left to right, keeping a stack of indices whose values are **decreasing** from bottom to top. These are the elements still waiting for their "next greater" answer. When a new element x arrives, every waiting element smaller than x has just found its answer — x — so pop them all. Then push x to wait its turn.

```
def next_greater(nums: list[int]) -> list[int]:
    """For each i, the next value to the right that is greater than nums[i], or -1."""
    answer = [-1] * len(nums)
    stack: list[int] = []                  # indices; nums[stack] decreasing
    for i, x in enumerate(nums):
        while stack and nums[stack[-1]] < x:
            answer[stack.pop()] = x
        stack.append(i)
    return answer
```

Why is this correct? Two facts. First, the stack really is decreasing: before any index is pushed, every smaller value on top of the stack has been popped. Second, when x pops an index j, x is truly the *nearest* greater element to j's right: every element between j and x was pushed on top of j and was not greater than it (otherwise it would have popped j itself).

Complexity: each index is pushed once and popped at most once, so the `while` loop runs at most n times in total. $O(n)$ time, despite the nested loop.

The four variants

Question for each element	Scan direction	Keep stack	Pop while
Next greater (to the right)	left → right	decreasing	`nums[top] < x`
Next smaller (to the right)	left → right	increasing	`nums[top] > x`
Previous greater (to the left)	left → right	decreasing	`nums[top] <= x`; then the top is the answer
Previous smaller (to the left)	left → right	increasing	`nums[top] >= x`; then the top is the answer

Two ways to read the same pass: the element being **popped** learns its *next* greater/smaller (it's the newcomer), and the **newcomer**, after popping, learns its *previous* greater/smaller (it's whatever remains on top).

Circular arrays (LC 503): iterate over indices $0 \dots 2n-1$ and use `i % n`, pushing only during the first pass. **Daily Temperatures** (LC 739) records `i - popped_index` instead of a value.

Largest rectangle in a histogram (LC 84)

The largest rectangle must be exactly as tall as *some* bar i — the shortest bar it spans. For bar i, the widest such rectangle extends left and right until it hits a strictly shorter bar. So the answer for bar i needs its **previous smaller** and **next smaller** elements, which one increasing stack provides:

```python
def largest_rectangle_area(heights: list[int]) -> int:
    stack: list[int] = []                    # indices; heights increasing
    best = 0
    for i, h in enumerate(heights + [0]):    # sentinel 0 flushes the stack
        while stack and heights[stack[-1]] >= h:
            height = heights[stack.pop()]
            left = stack[-1] if stack else -1    # previous smaller bar
            best = max(best, height * (i - left - 1))   # i is the next smaller bar
        stack.append(i)
    return best
```

When a bar is popped, both of its boundaries are known at once: the newcomer `i` is the first bar to its right that's shorter, and the new stack top is the first bar to its left that's shorter. The appended sentinel `0` forces every remaining bar off the stack at the end. **Maximal Rectangle** (LC 85) reduces to this problem once per row of a binary matrix.

Greedy with a monotonic stack

Remove K Digits (LC 402): delete k digits from a number to make it as small as possible. Greedy insight: a digit followed by a *smaller* digit should be removed, since the smaller one then shifts into a more significant position. Scanning left to right with an increasing stack implements exactly that:

```python
def remove_k_digits(num: str, k: int) -> str:
    stack: list[str] = []
    for d in num:
        while k and stack and stack[-1] > d:
            stack.pop()
            k -= 1
        stack.append(d)
```

```
    stack = stack[:len(stack) - k]           # still need removals: drop the largest tail
    return "".join(stack).lstrip("0") or "0"
```

Remove Duplicate Letters (LC 316, same as LC 1081) uses the same stack with two extra rules: every letter must appear exactly once, so a letter already on the stack is skipped, and a larger letter on top may be popped only if it **occurs again later** — otherwise it would be lost for good.

```
def remove_duplicate_letters(s: str) -> str:
    last = {ch: i for i, ch in enumerate(s)}
    stack: list[str] = []
    on_stack: set[str] = set()
    for i, ch in enumerate(s):
        if ch in on_stack:
            continue
        while stack and stack[-1] > ch and last[stack[-1]] > i:
            on_stack.remove(stack.pop())
        stack.append(ch)
        on_stack.add(ch)
    return "".join(stack)
```

Counting contributions

Sum of Subarray Minimums (LC 907): add up min over all $\Theta(n^2)$ subarrays. Flip the question: for each element, *in how many subarrays is it the minimum?* If L is the distance to its previous smaller element and R the distance to its next smaller-or-equal element, it is the minimum of exactly $L \cdot R$ subarrays (choose a start in L ways, an end in R ways). (Breaking ties asymmetrically — strict on one side, non-strict on the other — ensures a subarray with repeated minimums is counted exactly once.)

Both distances come out of a single pass, exactly as in the histogram: when `j` is popped, the newcomer `i` is its next smaller-or-equal element and the new top is its previous strictly smaller one.

```
def sum_subarray_mins(arr: list[int]) -> int:
    stack: list[int] = []                        # indices; arr strictly increasing
    total = 0
    for i, x in enumerate(arr + [0]):            # sentinel (arr[i] >= 1) flushes the stack
        while stack and arr[stack[-1]] >= x:
            j = stack.pop()
            left = stack[-1] if stack else -1
            total += arr[j] * (j - left) * (i - j)
        stack.append(i)
    return total % (10**9 + 7)
```

> **Take-Home Lesson**
>
> When a problem asks about "the nearest larger/smaller element," or about every subarray's max/min, think monotonic stack. The key insight is that an element dominated by a newer one can never be the answer for anything that comes later — so it can be discarded forever.

Queues in Algorithms

The queue's starring role is **breadth-first search**: vertices are explored in the order they're discovered, which guarantees that each is first reached by a shortest path. That story belongs to Graphs and Trees.

The **monotonic deque** is the queue-flavored sibling of the monotonic stack: it keeps candidates in decreasing order, evicts dominated ones from the back, and evicts expired ones from the front. It gives $O(n)$ sliding window maximum (LC 239) — see Two Pointers & Sliding Window.

Common Mistakes

1. **`list.pop(0)` as a dequeue.** $O(n)$ per call. Use `collections.deque`.
2. **Popping or peeking an empty stack.** Check `if stack` first. In bracket matching, a closing bracket with an empty stack is invalid input, not a crash.
3. **Forgetting leftovers.** After the scan, a non-empty stack means unclosed brackets (LC 20) or elements that never found a next greater (they keep the default answer).
4. **Storing values when you need indices.** Monotonic stack problems usually need distances or positions (daily temperatures, histogram widths). Push indices; look values up.
5. **Wrong strictness in monotonic stacks.** `<` vs `<=` decides how equal elements are treated. Decide deliberately, especially in counting problems where duplicates can be double-counted.
6. **Integer division semantics.** Python's `//` rounds toward $-\infty$ (`-7 // 2 == -4`); many problems expect truncation toward zero. Use `int(a / b)`.

Practice Problems

Problem	Technique
Valid Parentheses (LC 20)	Stack of open brackets
Longest Valid Parentheses (LC 32)	Index stack with a base index
Implement Queue using Stacks (LC 232)	Inbox / outbox, amortized $O(1)$
Design Circular Queue (LC 622)	Array with wrap-around
Min Stack (LC 155)	Augmented entries
Evaluate Reverse Polish Notation (LC 150)	Operand stack
Basic Calculator II (LC 227)	Stack of terms
Basic Calculator (LC 224)	Save context on `(`
Decode String (LC 394)	Stack of enclosing contexts
Asteroid Collision (LC 735)	Stack simulation
Daily Temperatures (LC 739)	Next greater, store distance
Next Greater Element II (LC 503)	Circular: iterate twice
Online Stock Span (LC 901)	Previous greater, online
Largest Rectangle in Histogram (LC 84)	Previous / next smaller
Maximal Rectangle (LC 85)	Histogram per row
Remove K Digits (LC 402)	Greedy increasing stack
Remove Duplicate Letters (LC 316)	Greedy stack, pop only if seen later
Sum of Subarray Minimums (LC 907)	Contribution counting

Intervals

Intervals are everywhere: meetings on a calendar, jobs on a machine, reservations for a room, the lifetime of a process, the span of a gene on a chromosome. Interval problems look varied — merge these, schedule those, count how many overlap — but almost all of them are solved by the same two moves:

1. **Sort** the intervals by an endpoint (and choosing *which* endpoint is the real decision).
2. **Sweep** from left to right, maintaining a small amount of state.

This chapter is also the right place to confront one of the central lessons of algorithm design. Interval problems invite **greedy** algorithms, and greedy algorithms invite *plausible-looking wrong answers*. We will see several of those, learn how to break them with counterexamples, and learn how to prove the correct one right.

Foundations

Representation

An interval is a pair `[start, end]`. Before writing any code, decide whether endpoints are **inclusive** or not, because it changes the overlap test:

- **Closed** $[s, e]$: `[1, 3]` and `[3, 5]` share the point 3, so they overlap. (LC 56 uses this convention.)
- **Half-open** $[s, e)$: `[1, 3)` and `[3, 5)` don't overlap — a meeting ending at 3 doesn't conflict with one starting at 3. (Meeting rooms use this convention.)

Half-open intervals are usually the better model for time: the length is `e - s`, and adjacent intervals tile without overlapping.

When do two intervals overlap?

The positive condition is easy to get wrong, so derive it from the negative. Two intervals are **disjoint** exactly when one ends before the other starts: $a_e \le b_s$ or $b_e \le a_s$ (half-open). Negating:

$$\text{overlap}(a, b) \iff a_s < b_e \ \text{ and } \ b_s < a_e$$

For closed intervals, use $\le$ in both places. The overlapping region, when it exists, is $[\max(a_s, b_s),\ \min(a_e, b_e)]$.

Only three ways for two intervals to meet

Up to symmetry, two intervals are either **disjoint**, **partially overlapping**, or **nested** (one contains the other). That short list is useful when testing an idea: any claim about pairs of intervals can be checked against all three cases, and any counterexample to an interval algorithm can be built by adding a third interval to one of these three pictures.

Which endpoint to sort by?

Goal	Sort by	Why
Merge overlapping intervals, compute a union	**start**	Anything that could merge with the current block starts before it ends
Select the most non-overlapping intervals	**end**	Finishing early leaves the most room for the rest
Count simultaneous overlaps	**both endpoints as events**	Only the moments where something starts or ends matter

Merging Intervals (LC 56)

Sort by start. Walk through the intervals, maintaining the current merged block. An interval either overlaps the block (extend it) or starts after it ends (the block is finished; start a new one).

```
def merge(intervals: list[list[int]]) -> list[list[int]]:
    intervals.sort(key=lambda iv: iv[0])
    merged: list[list[int]] = []
    for start, end in intervals:
        if merged and start <= merged[-1][1]:            # overlaps the current block
            merged[-1][1] = max(merged[-1][1], end)      # nested intervals: keep the larger end
        else:
            merged.append([start, end])
    return merged
```

Why is sorting by start enough? Invariant: `merged` holds disjoint blocks covering exactly the intervals seen so far, and nothing unseen starts before the last block's start. If the next interval starts after the last block ends, then *every* later interval starts even later — none of them can reach back to touch the finished block. So only the last block ever needs checking.

The `max` handles the **nested** case: `[1, 10]` followed by `[2, 3]` must not shrink the block to end at 3.

Time: $O(n \log n)$ for the sort; the sweep is $O(n)$.

Insert into a sorted list (LC 57)

When the intervals are already sorted and disjoint, inserting a new one needs no sort. Three phases in one pass: copy the intervals that end before the new one starts, absorb every interval that overlaps it, copy the rest.

```
def insert(intervals: list[list[int]], new: list[int]) -> list[list[int]]:
    result = []
    i, n = 0, len(intervals)
    start, end = new
    while i < n and intervals[i][1] < start:          # entirely before
        result.append(intervals[i])
        i += 1
    while i < n and intervals[i][0] <= end:           # overlaps: absorb
        start = min(start, intervals[i][0])
        end = max(end, intervals[i][1])
        i += 1
    result.append([start, end])
```

```
    result.extend(intervals[i:])                    # entirely after
    return result
```

$O(n)$. (Binary search could locate the overlap region in $O(\log n)$, but building the output list is $O(n)$ anyway.)

Intersecting two interval lists (LC 986)

Given two sorted lists of disjoint intervals, report all pairwise intersections. Two pointers, one per list: the intersection of the current pair is `[max(starts), min(ends)]` if non-empty. Then advance whichever interval **ends first** — it can't intersect anything further in the other list.

```
def interval_intersection(a: list[list[int]], b: list[list[int]]) -> list[list[int]]:
    i = j = 0
    result = []
    while i < len(a) and j < len(b):
        lo = max(a[i][0], b[j][0])
        hi = min(a[i][1], b[j][1])
        if lo <= hi:
            result.append([lo, hi])
        if a[i][1] < b[j][1]:
            i += 1
        else:
            j += 1
    return result
```

Interval Scheduling: The Movie Star Problem

Here is a problem worth thinking about slowly.

> *You are a movie star with offers for n film projects. Each project occupies a fixed interval of days, and you can't be on two sets at once. Every film pays the same. Which projects should you accept to star in **as many films as possible**?*

In other words: given n intervals, find the largest subset of mutually non-overlapping ones. Before reading on, try to find an algorithm.

Three plausible ideas, three failures

Idea 1: earliest start first. Take the job that starts first, discard everything that conflicts with it, repeat. After all, no other job can use those early days.

It fails when the earliest job is very long. One epic that starts first and runs all year blocks every other offer:

```
[====================== epic ======================]
  [a]   [b]   [c]   [d]   [e]
earliest-start picks 1 film; the optimum is 5
```

Idea 2: shortest job first. The problem with the epic was its length, so prefer short jobs.

It fails when a short job straddles two longer ones:

```
[=== x ===]   [=== y ===]
       [ s ]
shortest-first picks s (1 film); the optimum is x and y (2 films)
```

The loss here is at most half the optimum — but "at most half as good" is still wrong.

Idea 3: fewest conflicts first. Take the interval that overlaps the fewest others; surely that sacrifices the least. This one is harder to break, which is exactly why it's dangerous.

Stop and Think: Break the fewest-conflicts heuristic

Problem: Find an instance where repeatedly choosing the interval with the fewest overlaps gives a suboptimal answer.
Solution: Start from four disjoint intervals in a row — the optimal answer, $T_1 \dots T_4$. Add a short interval M that bridges T_2 and T_3; it conflicts with just those two. Now make T_2 and T_3 look *worse* than M by piling three identical intervals across the T_1/T_2 boundary and three across the T_3/T_4 boundary:

```
T1 [0,2]   T2 [3,5]   T3 [6,8]   T4 [9,11]
       3 x [1.5,3.5]      3 x [7.5,9.5]
               M [4.5,6.5]
```

Conflict counts: M has 2, T_1 and T_4 have 3, T_2 and T_3 have 4, each stacked interval has 4. The heuristic grabs M, which eliminates T_2 and T_3, and can then get at most one interval from each side: 3 films. The optimum, $T_1 \dots T_4$, is 4.
Notice the recipe: find the weakness ("a low-conflict interval can still block good ones"), then use **ties and piles of identical items** to steer the heuristic into it.

Take-Home Lesson

Reasonable-looking algorithms are easily wrong. Correctness must be demonstrated, not assumed — and the fastest way to demolish a wrong idea is a **small counterexample**. Think small (two or three intervals), think exhaustively (try all three pairwise configurations), seek extremes (very long vs. very short), and go for ties.

The algorithm that works: earliest finish first

Consider the interval x that **ends first**. Every other interval that conflicts with x must still be running just before x ends (it can't end earlier, by the choice of x), so all of them overlap each other at that moment — at most one of them can be in *any* solution. And among them, x is the one that frees up the timeline earliest. So choosing x can never hurt.

```python
def max_non_overlapping(intervals: list[list[int]]) -> int:
    """Largest number of pairwise non-overlapping half-open intervals."""
    count = 0
    last_end = float("-inf")
    for start, end in sorted(intervals, key=lambda iv: iv[1]):
        if start >= last_end:              # compatible with everything chosen
            count += 1
            last_end = end
    return count
```

Proof by exchange. Let $G = g_1, g_2, \dots$ be the greedy picks and $O = o_1, o_2, \dots$ any optimal solution, both sorted by end time. Greedy picked g_1 as the earliest-ending interval of all, so $\text{end}(g_1) \le \text{end}(o_1)$. Replace o_1 with g_1 in O: it still doesn't conflict with o_2, since g_1 ends no later than o_1 did. Repeating the argument, greedy "stays ahead" at every step — its k-th pick ends no later than the optimum's k-th — so if the optimum

had a $(k+1)$-th interval, it would also have been available to greedy. Hence $|G| \geq |O|$.

Time: $O(n \log n)$.

This exchange argument is *the* template for proving greedy algorithms correct: show that the greedy choice can be swapped into any optimal solution without making it worse. See Greedy for more.

Problems that are secretly interval scheduling

Problem	Reduction
Non-overlapping Intervals (LC 435): fewest removals to eliminate overlaps	`n - max_non_overlapping(intervals)`
Minimum Arrows to Burst Balloons (LC 452)	Same greedy (closed intervals, so compare with `>`). Each chosen interval's end is where an arrow goes; the number of disjoint intervals equals the number of arrows needed
Maximum Length of Pair Chain (LC 646)	Exactly interval scheduling

The arrows problem is a nice duality: the maximum number of pairwise-disjoint intervals equals the minimum number of points that **stab** every interval. Greedy finds both at once.

Partitioning: How Many Rooms? (LC 253)

Now keep *all* the intervals, and ask for the minimum number of rooms (machines, colors) so that no two overlapping meetings share a room.

Lower bound: if some moment has d meetings in progress, you need at least d rooms. Call the maximum such d over all moments the **depth**.

Upper bound: the depth is always enough. Process meetings by start time, and reuse any room whose meeting has ended. A new room is opened only when all current rooms are busy — that is, at a moment when that many meetings overlap. So the number of rooms never exceeds the depth.

So the answer is simply **the maximum number of simultaneous intervals**, which a sweep over endpoints computes directly:

```python
def min_meeting_rooms(intervals: list[list[int]]) -> int:
    events = []
    for start, end in intervals:
        events.append((start, 1))      # a meeting begins
        events.append((end, -1))       # a meeting ends
    events.sort()                      # at equal times, -1 sorts before +1
    rooms = best = 0
    for _, delta in events:
        rooms += delta
        best = max(best, rooms)
    return best
```

The tuple sort puts an end event `(t, -1)` before a start event `(t, 1)` at the same time — exactly right for half-open intervals, where a room freed at time t can be reused at time t. For closed intervals, you'd want the opposite order.

If you need to know **which** room each meeting gets, simulate the greedy assignment with a min-heap of end times:

```
import heapq


def min_meeting_rooms_heap(intervals: list[list[int]]) -> int:
    ends: list[int] = []                          # end times of rooms in use
    for start, end in sorted(intervals):
        if ends and ends[0] <= start:
            heapq.heapreplace(ends, end)          # reuse the room that frees up first
        else:
            heapq.heappush(ends, end)             # open a new room
    return len(ends)
```

Take-Home Lesson

For many optimization problems, the smartest first step is to find a simple **lower bound** on the answer, then look for an algorithm that meets it. When a greedy algorithm matches the lower bound, it's optimal — no exchange argument needed.

The Sweep Line

The events technique generalizes: turn each interval into a "+1 at start, −1 at end" pair of events, sort the events, and sweep while maintaining a running state. It answers questions like:

- Maximum overlap, and *when* it occurs.
- Total length covered by the union of intervals (add up the stretches where the count is positive).
- Whether any point is covered by more than k intervals (LC 731, 732).
- The skyline of a set of buildings (LC 218), with a max-heap of active heights as the state.

The skyline (LC 218)

Sweep the building edges left to right, keeping a max-heap of the buildings standing at the current x. The skyline changes exactly where the tallest active height changes. Removing an arbitrary building from a heap is awkward, so removal is **lazy**: each heap entry carries its right edge, and expired entries are popped only when they reach the top.

```
def get_skyline(buildings: list[list[int]]) -> list[list[int]]:
    events = sorted([(left, -h, right) for left, right, h in buildings]
                    + [(right, 0, 0) for _, right, _ in buildings])
    live = [(0, float("inf"))]                    # (-height, right edge); the ground never ends
    skyline = [[0, 0]]
    for x, neg_h, right in events:
        while live[0][1] <= x:                    # drop buildings that ended at or before x
            heapq.heappop(live)
        if neg_h:
            heapq.heappush(live, (neg_h, right))
        if -live[0][0] != skyline[-1][1]:
            skyline.append([x, -live[0][0]])
    return skyline[1:]
```

The event order handles ties: at the same x, starts (negative key) come before ends (key 0), and taller starts

come first. A building that starts exactly where another ends therefore never creates a false dip to the lower height. $O(n \log n)$.

Difference arrays: sweep lines on integer grids

When coordinates are small integers, skip the sort: record `+x` at the start and `-x` just past the end in an array, then take a prefix sum. Each prefix sum value is the total active at that position.

```python
def car_pooling(trips: list[list[int]], capacity: int) -> bool:
    """LC 1094: trips[i] = [passengers, from, to]; can one car carry them all?"""
    delta = [0] * 1001
    for passengers, start, end in trips:
        delta[start] += passengers
        delta[end] -= passengers           # they get off at `end`
    load = 0
    for change in delta:
        load += change
        if load > capacity:
            return False
    return True
```

$O(n + C)$ for coordinate range C — a counting-sort version of the sweep. Corporate Flight Bookings (LC 1109) is the same trick.

When Greedy Fails: Weighted Scheduling

Change the movie problem slightly: each film now pays a **different** fee, and you want to maximize total earnings (LC 1235, Maximum Profit in Job Scheduling). Earliest-finish-first breaks immediately — it would happily take a 1-dollar job that blocks a million-dollar one.

The fix is **dynamic programming** over the jobs sorted by end time. For job i, either skip it, or take it and add the best total over jobs that end by the time i starts. That "latest compatible job" is found by binary search.

```python
import bisect


def job_scheduling(start: list[int], end: list[int], profit: list[int]) -> int:
    jobs = sorted(zip(end, start, profit))
    ends = [e for e, _, _ in jobs]
    best = [0] * (len(jobs) + 1)          # best[i] = max profit using the first i jobs
    for i, (e, s, p) in enumerate(jobs):
        k = bisect.bisect_right(ends, s, 0, i)    # jobs[:k] all end by time s
        best[i + 1] = max(best[i], best[k] + p)
    return best[-1]
```

Time: $O(n \log n)$. The pattern — sort, then DP where each item looks back to its last compatible predecessor — recurs throughout scheduling problems. See Dynamic Programming.

> **Narrowing the problem**
>
> Scheduling is extremely sensitive to the exact problem statement. Unweighted intervals: greedy, $O(n \log n)$. Weighted intervals: DP, $O(n \log n)$. But allow each job to occupy *several* disjoint intervals (a film that shoots in March and again in June), and the problem becomes NP-complete — no efficient

> algorithm is known at all. When a problem seems hard, check whether the instances you actually need to handle belong to a simpler special case. Restricting the input until an efficient algorithm exists is a legitimate and powerful design technique.

Common Mistakes

1. **Closed vs. half-open confusion.** `start <= prev_end` merges `[1, 3]` and `[3, 5]`; `start < prev_end` doesn't. Decide which the problem intends before writing the comparison — including in event sorting.
2. **Forgetting nested intervals when merging.** `merged[-1][1] = end` instead of `max(merged[-1][1], end)` shrinks a block when a short interval sits inside a long one.
3. **Sorting by the wrong endpoint.** Merging wants start order; selecting the most disjoint intervals wants end order. Earliest-*start*-first selection is a classic wrong answer.
4. **Wrong tie-breaking in sweeps.** Whether an end and a start at the same instant overlap is decided by event order. Encode it deliberately.
5. **Trusting a greedy rule without proof.** Before coding a greedy interval algorithm, spend a minute hunting for a counterexample with two or three intervals. If you can't find one, try to sketch the exchange argument.
6. **Mutating the input.** `intervals.sort()` and editing `merged[-1]` in place modify lists the caller may still use. Copy if that matters.

Practice Problems

Problem	Technique
Merge Intervals (LC 56)	Sort by start, extend block
Insert Interval (LC 57)	Three-phase linear scan
Interval List Intersections (LC 986)	Two pointers, advance the earlier end
Meeting Rooms (LC 252)	Sort, check adjacent pairs
Meeting Rooms II (LC 253)	Sweep line or heap of end times
Non-overlapping Intervals (LC 435)	Earliest finish first
Minimum Number of Arrows (LC 452)	Earliest finish first (stabbing)
Maximum Length of Pair Chain (LC 646)	Earliest finish first
Remove Covered Intervals (LC 1288)	Sort by start asc, end desc
Car Pooling (LC 1094)	Difference array
My Calendar II (LC 731)	Sweep / overlap counting
Employee Free Time (LC 759)	Merge, then report gaps
The Skyline Problem (LC 218)	Sweep line + max-heap
Maximum Profit in Job Scheduling (LC 1235)	Sort by end + DP + binary search

Heaps & Priority Queues

Many algorithms need to process items in order of importance: the cheapest edge, the earliest deadline, the closest unexplored vertex, the next event in a simulation. If all the items were known in advance, you could sort them once. But usually they aren't — new items keep arriving *while* you are processing old ones, and re-sorting after each arrival would be ruinous.

A **priority queue** is the data structure for this situation. It supports three operations:

- **Insert** an item with a key.
- **Find-min**: return the item with the smallest key.
- **Delete-min**: remove and return that item.

(Or max instead of min — the choice is symmetric.) The **binary heap** implements all three in $O(\log n)$ or better, with a representation so compact it needs no pointers at all.

> **Take-Home Lesson**
>
> Building algorithms around the right abstract data type — dictionaries, priority queues — leads to both clean structure and good performance. Many famous algorithms (Dijkstra, Prim, Huffman coding, heapsort, event simulation) are "a priority queue plus a loop."

How Should a Priority Queue Be Built?

Before looking at heaps, consider the obvious alternatives. The table has one subtlety: find-min can be $O(1)$ for *every* structure, by caching a pointer to the current minimum and updating it on each insert and delete (a delete must then find the new minimum, which is where the real cost goes).

	Unsorted array	Sorted array	Balanced BST	Binary heap
Insert	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
Find-min	$O(1)$ (cached)	$O(1)$	$O(1)$ (cached)	$O(1)$
Delete-min	$O(n)$	$O(1)$ (min stored at the end)	$O(\log n)$	$O(\log n)$
Build from n items	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

A balanced search tree matches the heap asymptotically, and supports more operations besides (search, successor, delete arbitrary item). So why heaps? Because they are dramatically simpler, use no pointers, build in linear time, and are several times faster in practice. When all you need is "give me the minimum," a heap is the right tool.

The Binary Heap

A heap maintains a **partial order**: weaker than sorted order (so it's cheap to maintain) but stronger than no order (so the minimum is always easy to find).

> ***Heap property:*** *every node's key is less than or equal to the keys of its children.*

So the root holds the minimum, every root-to-leaf path is sorted, and — importantly — *nothing* is promised about the order between siblings or cousins.

An implicit tree

A heap is a **complete** binary tree: every level is full except possibly the last, which is filled from the left. Complete trees can be stored in an array with no pointers: put the root at index 0 and lay out the levels left to right.

```
          1                    index:  0  1  2  3  4  5  6  7
        /   \                  value:  1  3  2  7  4  5  9  8
       3     2
      / \   / \                children of i:  2i + 1, 2i + 2
     7   4 5   9               parent of i:    (i - 1) // 2
    /
   8
```

Because the tree is complete, an n-element heap has height $\lfloor \log_2 n \rfloor$. That's what makes every operation logarithmic.

The price of the implicit representation is inflexibility: you can't store an arbitrary tree shape without wasting space, and you can't move a subtree by changing one pointer. Fine for heaps — fatal for search trees.

Stop and Think: Who's where in the heap?

Problem: How do you efficiently search a heap for a particular key k?
Solution: You can't. A heap is not a search tree: knowing that k is larger than a node tells you nothing about which of its two subtrees to explore. About half of the elements are leaves, and the heap property says nothing about their relative order. Searching takes $O(n)$. If you need both "find the minimum" and "find key k" quickly, you need a balanced BST — or a heap *plus* a hash map from key to position.

Insert: sift up

Put the new element in the first free slot (the end of the array). The shape is right, but the new element might be smaller than its parent. Swap it upward until it isn't. Each swap fixes the parent–child pair and moves the problem one level up: $O(\log n)$.

Delete-min: sift down

Remove the root, move the last element into its place, and swap it downward with its **smaller** child until it's no larger than both children. Swapping with the smaller child is essential: that child becomes the parent of the other, so it had better be the smaller of the two. Again $O(\log n)$.

```python
class MinHeap:
    def __init__(self) -> None:
        self.a: list[int] = []

    def __len__(self) -> int:
```

```
        return len(self.a)

    def peek(self) -> int:
        return self.a[0]

    def push(self, x: int) -> None:
        a = self.a
        a.append(x)
        i = len(a) - 1
        while i > 0 and a[(i - 1) // 2] > a[i]:          # sift up
            parent = (i - 1) // 2
            a[i], a[parent] = a[parent], a[i]
            i = parent

    def pop(self) -> int:
        a = self.a
        top = a[0]
        last = a.pop()
        if a:
            a[0] = last
            self._sift_down(0)
        return top

    def _sift_down(self, i: int) -> None:
        a, n = self.a, len(self.a)
        while True:
            smallest = i
            for child in (2 * i + 1, 2 * i + 2):
                if child < n and a[child] < a[smallest]:
                    smallest = child
            if smallest == i:
                return
            a[i], a[smallest] = a[smallest], a[i]
            i = smallest
```

Building a heap in linear time

Inserting n items one at a time costs $O(n \log n)$. But there's a faster way. Dump all the items into the array in any order. The leaves — the last half of the array — are already valid one-element heaps. Walk backward through the remaining nodes, sifting each one down; when you reach node i, both of its subtrees are already valid heaps, which is exactly the situation sift-down repairs.

```
def build_heap(items: list[int]) -> MinHeap:
    h = MinHeap()
    h.a = list(items)
    for i in range(len(h.a) // 2 - 1, -1, -1):
        h._sift_down(i)
    return h
```

$n/2$ calls to an $O(\log n)$ procedure suggests $O(n \log n)$. But that bound is loose: sift-down costs time proportional to the node's **height**, and most nodes are near the bottom. About $n/4$ nodes have height 1, $n/8$ have height 2, and in general $n/2^{h+1}$ have height h:

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot h \;=\; \frac{n}{2} \sum_{h \ge 0} \frac{h}{2^h} \;\le\; \frac{n}{2} \cdot 2 \;=\; n$$

The h in the numerator is crushed by the 2^h in the denominator, and the sum converges to a constant. **Heap construction is $O(n)$.** It's another free lunch courtesy of a geometric-flavored series.

Heapsort

Build a heap, then delete-min n times. This is just **selection sort with a better data structure**: selection sort repeatedly finds the minimum by scanning in $O(n)$; heapsort finds it in $O(\log n)$. Worst-case $O(n \log n)$, and it can run in place. See Sorting.

Python's heapq

`heapq` implements a **min-heap** as functions operating on a plain `list`:

```
import heapq

h: list[int] = []
heapq.heappush(h, 5)             # O(log n)
heapq.heappush(h, 1)
heapq.heappush(h, 3)
h[0]                             # peek at the min: 1, O(1)
heapq.heappop(h)                 # remove and return the min: 1, O(log n)

data = [9, 4, 7, 1]
heapq.heapify(data)              # in place, O(n)

heapq.heappushpop(h, 2)          # push, then pop: faster than the two calls
heapq.heapreplace(h, 8)          # pop, then push: heap size unchanged

heapq.nsmallest(2, [9, 4, 7, 1])            # [1, 4], O(n log k)
heapq.nlargest(2, [9, 4, 7, 1])             # [9, 7]
list(heapq.merge([1, 4, 9], [2, 3, 10]))    # lazy k-way merge of sorted inputs
```

Max-heaps

`heapq` is a min-heap. The standard trick for a max-heap is to **negate the keys** on the way in and out:

```
maxh: list[int] = []
for x in [3, 1, 4, 1, 5]:
    heapq.heappush(maxh, -x)
largest = -heapq.heappop(maxh)     # 5
```

(Python 3.14 also adds `heapq.heappush_max`, `heappop_max`, and `heapify_max`, which avoid the negation.)

Ordering by key, and ties

Push tuples: they compare lexicographically, so `(priority, item)` orders by priority. If two priorities tie, Python compares the items — which crashes if the items aren't comparable (e.g., `ListNode` objects). Insert a **tiebreaker** that is always unique:

```
from itertools import count

tiebreak = count()
```

```
tasks: list[tuple[int, int, object]] = []
heapq.heappush(tasks, (2, next(tiebreak), object()))
heapq.heappush(tasks, (2, next(tiebreak), object()))   # no comparison of objects needed
```

A monotonically increasing counter also makes the heap FIFO among equal priorities — a stable priority queue.

Pattern 1: The k Best of n

To find the k largest elements, keep a **min-heap of size** k holding the best k seen so far. Its root is the *smallest* of those k — the one to evict when something better arrives.

```
def find_kth_largest(nums: list[int], k: int) -> int:
    """LC 215."""
    heap = nums[:k]
    heapq.heapify(heap)
    for x in nums[k:]:
        if x > heap[0]:
            heapq.heapreplace(heap, x)     # evict the weakest of the current top k
    return heap[0]
```

Invariant: after processing a prefix, `heap` holds the k largest elements of that prefix. **Time:** $O(n \log k)$, **space:** $O(k)$ — and it works on a stream, never holding more than k items.

It may seem backward that "k largest" uses a **min**-heap. The heap exists to answer one question quickly: *which of my current candidates should be thrown out?* — and that's the smallest one.

Approach for k-th largest	Time	Space	Notes
Sort	$O(n \log n)$	$O(n)$	Simplest
Size-k min-heap	$O(n \log k)$	$O(k)$	Streams; best when $k \ll n$
Heapify all, pop k times	$O(n + k \log n)$	$O(n)$	
Quickselect	$O(n)$ expected	$O(1)$	Fastest; see Sorting

The same pattern handles K Closest Points (LC 973; a max-heap of size k on distance, via negation), Top K Frequent Elements (LC 347; heap over `Counter` items), and Kth Largest Element in a Stream (LC 703).

Pattern 2: Merging Sorted Streams

To merge k sorted lists with n total elements (LC 23), keep a heap of the k current heads. Pop the smallest, emit it, and push its successor from the same list. The heap never holds more than k items.

```
def merge_k_sorted(lists: list[list[int]]) -> list[int]:
    heap = [(lst[0], i, 0) for i, lst in enumerate(lists) if lst]
    heapq.heapify(heap)
    result = []
    while heap:
```

```
        val, i, j = heapq.heappop(heap)
        result.append(val)
        if j + 1 < len(lists[i]):
            heapq.heappush(heap, (lists[i][j + 1], i, j + 1))
    return result
```

Time: $O(n \log k)$. The list index `i` in each tuple doubles as a tiebreaker.

Smallest Range Covering Elements from K Lists (LC 632) is a variation: the heap holds one element per list, and the range is `[heap min, running max]`. The minimum is the only element worth advancing. Advancing any other element can only raise the maximum, while the minimum stays the same. Once the list holding the minimum runs out, no later range can include that list, so the search stops.

```
def smallest_range(lists: list[list[int]]) -> list[int]:
    heap = [(lst[0], i, 0) for i, lst in enumerate(lists)]
    heapq.heapify(heap)
    hi = max(lst[0] for lst in lists)
    best = [heap[0][0], hi]
    while True:
        lo, i, j = heapq.heappop(heap)
        if hi - lo < best[1] - best[0]:
            best = [lo, hi]
        if j + 1 == len(lists[i]):
            return best
        hi = max(hi, lists[i][j + 1])
        heapq.heappush(heap, (lists[i][j + 1], i, j + 1))
```

Pattern 3: Best-First Enumeration of a Frontier

Some problems ask for the k smallest of a set of candidates that is far too large to build — but that has **order structure**: you can tell which candidates must come after which.

An airline wanted the cheapest valid two-leg fare between two cities. Each leg had its own sorted list of fares, X and Y, and a black-box rule checker decided whether a particular combination was legal. The cheapest *legal* fare might be far down the list of $|X| \cdot |Y|$ combinations, and building and sorting all of them was too slow when the answer was usually near the top.

The key observation: if both lists are sorted, the pair (i, j) is never more expensive than $(i+1, j)$ or $(i, j+1)$. So there's no need to consider those two until (i, j) has been examined. Start the priority queue with just $(0, 0)$. Each time you pop a pair, push its two successors. Pairs come out in exactly increasing order of cost, and you stop as soon as you find what you want — having generated only a thin **frontier** of candidates.

This is **Find K Pairs with Smallest Sums** (LC 373). One detail: pair (i, j) can be reached from both $(i-1, j)$ and $(i, j-1)$. Either keep a `seen` set, or use a trick that gives every pair a unique parent: seed the heap with $(i, 0)$ for every i, and only ever step to $(i, j+1)$.

```
def k_smallest_pairs(xs: list[int], ys: list[int], k: int) -> list[list[int]]:
    if not xs or not ys:
        return []
    heap = [(x + ys[0], i, 0) for i, x in enumerate(xs[:k])]
    heapq.heapify(heap)
    result = []
    while heap and len(result) < k:
```

```
        _, i, j = heapq.heappop(heap)
        result.append([xs[i], ys[j]])
        if j + 1 < len(ys):
            heapq.heappush(heap, (xs[i] + ys[j + 1], i, j + 1))
    return result
```

Time: $O(k \log k)$, independent of the lengths of the input lists. **Kth Smallest Element in a Sorted Matrix** (LC 378) and **Ugly Number II** (LC 264) use the same idea.

> **Take-Home Lesson**
>
> When the candidate set is huge but partially ordered — every candidate has "successors" that can't be better — don't generate it. Expand it lazily from the best candidate, with a priority queue holding the frontier. This is the same idea as Dijkstra's algorithm, and it lets you stop the moment you have your answer.

Pattern 4: Two Heaps

To maintain the **median** of a growing stream (LC 295), split the elements into a lower half and an upper half:

- `low`: a **max**-heap holding the smaller half.
- `high`: a **min**-heap holding the larger half.

Maintain two invariants: every element of `low` is at most every element of `high`, and the sizes differ by at most one (with `low` allowed the extra element). Then the median is at the top of `low`, or the average of both tops.

```
class MedianFinder:
    def __init__(self) -> None:
        self.low: list[int] = []    # max-heap via negation
        self.high: list[int] = []   # min-heap

    def add_num(self, x: int) -> None:
        # Route through low so the ordering invariant holds, then rebalance sizes.
        heapq.heappush(self.high, -heapq.heappushpop(self.low, -x))
        if len(self.high) > len(self.low):
            heapq.heappush(self.low, -heapq.heappop(self.high))

    def find_median(self) -> float:
        if len(self.low) > len(self.high):
            return -self.low[0]
        return (-self.low[0] + self.high[0]) / 2
```

`add_num` is $O(\log n)$ and `find_median` is $O(1)$. The same two-heap split answers any "running k-th order statistic" question, and appears in scheduling problems like IPO (LC 502), where one heap holds projects you can't afford yet (keyed by cost) and another holds those you can (keyed by profit). Costs never change, so a sorted list with a pointer can replace the first heap:

```
def find_maximized_capital(k: int, w: int, profits: list[int], capital: list[int]) -> int:
    projects = sorted(zip(capital, profits))     # unaffordable ones, cheapest first
    affordable: list[int] = []                   # max-heap of profits via negation
    i = 0
```

```
    for _ in range(k):
        while i < len(projects) and projects[i][0] <= w:
            heapq.heappush(affordable, -projects[i][1])
            i += 1
        if not affordable:
            break
        w -= heapq.heappop(affordable)          # take the most profitable affordable project
    return w
```

Capital only grows, so a project that becomes affordable stays affordable. Taking the most profitable one each time is therefore safe: whatever you could afford with less capital, you can still afford with more.

Pattern 5: Greedy Scheduling

Greedy algorithms repeatedly take "the best available option," and a priority queue is how you find it fast.

Reorganize String (LC 767): rearrange characters so no two adjacent ones are equal. Greedily place the most frequent remaining character that differs from the previous one. A max-heap of `(-count, char)` gives that character in $O(\log \Sigma)$; hold back the character just used for one step so it can't be picked twice in a row.

```
from collections import Counter


def reorganize_string(s: str) -> str:
    heap = [(-c, ch) for ch, c in Counter(s).items()]
    heapq.heapify(heap)
    result: list[str] = []
    held: tuple[int, str] | None = None          # the char used last step
    while heap:
        count, ch = heapq.heappop(heap)
        result.append(ch)
        if held:
            heapq.heappush(heap, held)
        held = (count + 1, ch) if count + 1 < 0 else None
    return "".join(result) if len(result) == len(s) else ""
```

Other greedy-with-heap problems: Task Scheduler (LC 621), Meeting Rooms II (a heap of end times — see Intervals), Minimum Cost to Connect Sticks (LC 1167, which is Huffman coding), and, most importantly, **Dijkstra's shortest paths** and **Prim's minimum spanning tree** (see Graphs).

Lazy Deletion

`heapq` can't delete an arbitrary element or change a key. The standard workaround is **lazy deletion**: leave stale entries in the heap, and discard them when they surface at the top.

```
def top_valid(heap: list[tuple[int, int]], is_stale) -> tuple[int, int] | None:
    while heap and is_stale(heap[0]):
        heapq.heappop(heap)
    return heap[0] if heap else None
```

To "update" a key, push a new entry and mark the old one stale (for example, by checking against a dictionary of current values). Each entry is pushed once and popped at most once, so the amortized cost stays $O(\log n)$ per operation; the heap may just grow larger than the live set. Dijkstra's algorithm in Python is almost always written this way.

Stop and Think: Is the k-th smallest at least x?

Problem: Given an array-based min-heap of n elements and a value x, decide whether the k-th smallest element is $\geq x$, in $O(k)$ time — independent of n.
Two correct but slow ideas: pop k times, $O(k \log n)$. Or note the k-th smallest must lie in the top k levels and scan them, $O(\min(n, 2^k))$.
The $O(k)$ idea: you don't need to *find* the k-th smallest, only to count elements below x. Do a DFS from the root that only descends through nodes $< x$ (by the heap property, everything below a node $\geq x$ is also $\geq x$), and stop as soon as k such nodes have been found. Each counted node causes at most two children to be examined, so at most $2k + 1$ nodes are visited.

```python
def kth_smallest_at_least(heap: list[int], k: int, x: int) -> bool:
    count = 0
    stack = [0]
    while stack and count < k:
        i = stack.pop()
        if i < len(heap) and heap[i] < x:
            count += 1
            stack.extend((2 * i + 1, 2 * i + 2)) # Note: dfs will be performed on right
↪ then left children
    return count < k          # fewer than k elements are below x
```

Common Mistakes

1. **Min vs. max confusion.** `heapq` is a min-heap. For "k largest," use a min-heap of size k; for a max-heap, negate keys.
2. **Uncomparable ties.** `(priority, node)` tuples crash when priorities tie and nodes don't define `<`. Add a counter between them.
3. **Treating the heap list as sorted.** Only `h[0]` is meaningful. `h[1]` is *not* the second smallest; `h[-1]` is *not* the largest.
4. **Mutating keys in place.** Changing an element's priority inside the list silently breaks the heap property. Push a new entry and lazily delete the old one.
5. **Building with repeated pushes.** n calls to `heappush` cost $O(n \log n)$; `heapify` costs $O(n)$.
6. **Using `nsmallest`/`nlargest` for k close to n.** For large k they're slower than just sorting; for $k = 1$, use `min`/`max`.

Practice Problems

Problem	Pattern
Kth Largest Element in an Array (LC 215)	Size-k min-heap (or quickselect)
Kth Largest Element in a Stream (LC 703)	Size-k min-heap
Last Stone Weight (LC 1046)	Max-heap simulation
K Closest Points to Origin (LC 973)	Size-k max-heap
Top K Frequent Elements (LC 347)	Heap over counts
Merge k Sorted Lists (LC 23)	k-way merge
Find K Pairs with Smallest Sums (LC 373)	Best-first frontier
Kth Smallest Element in a Sorted Matrix (LC 378)	Best-first frontier
Find Median from Data Stream (LC 295)	Two heaps
IPO (LC 502)	Two heaps, greedy
Reorganize String (LC 767)	Greedy max-heap with hold-back
Task Scheduler (LC 621)	Greedy max-heap / counting
Smallest Range Covering Elements from K Lists (LC 632)	k-way merge + running max
Sliding Window Median (LC 480)	Two heaps + lazy deletion

Trees

Trees model **hierarchy**: a family tree, a file system, an organization chart, the parse of an expression, the decisions of a search. Whenever a problem talks about ancestors and descendants, parents and children, containment or dominance, there is probably a tree in it.

But the real reason trees deserve careful study is that they are the purest example of a **recursive object**. A binary tree is either empty, or a root together with two smaller binary trees. Almost every tree algorithm is written by trusting that definition: solve the problem for the two subtrees, then combine. If you become fluent at that one move, the vast majority of tree problems become short exercises.

Recursion Is Induction

Recursion can feel like magic the first time you see it: the function calls itself on a smaller input and somehow gets the right answer. Mathematical induction can feel the same way: you assume the claim for smaller cases and somehow prove it for the general one. They feel alike because **they are the same thing**. A recursive function is correct if:

1. **Base case:** it returns the right answer on the smallest inputs (the empty tree, a leaf).
2. **Inductive step:** *assuming* the recursive calls return correct answers for the subtrees, the function combines them into the correct answer for the whole tree.

That's the entire method. You never trace the recursion all the way down; you trust the recursive calls exactly the way an inductive proof trusts the induction hypothesis.

```python
from __future__ import annotations

from dataclasses import dataclass


@dataclass
class TreeNode:
    val: int = 0
    left: TreeNode | None = None
    right: TreeNode | None = None


def max_depth(root: TreeNode | None) -> int:
    """LC 104. Base: the empty tree has depth 0. Step: 1 + the deeper subtree."""
    if root is None:
        return 0
    return 1 + max(max_depth(root.left), max_depth(root.right))
```

> **Take-Home Lesson**
>
> To write a tree function, answer two questions: *what should it return for an empty tree?* and *given correct answers for the left and right subtrees, how do I build the answer for the whole tree?* Don't trace the recursion; trust it.

Strengthening the hypothesis

Sometimes the answers from the subtrees aren't *enough* to build the answer for the tree. Then the fix is the same as in an induction proof that gets stuck: **prove something stronger**. Make the recursive function return more information than the problem asked for.

Balanced Binary Tree (LC 110). A tree is height-balanced if every node's subtrees differ in height by at most one. The definition transcribes directly:

```
def is_balanced_naive(root: TreeNode | None) -> bool:
    if root is None:
        return True
    return (abs(max_depth(root.left) - max_depth(root.right)) <= 1
            and is_balanced_naive(root.left) and is_balanced_naive(root.right))
```

It's correct, but every node recomputes the heights of its whole subtree, which is $O(n^2)$ on a path-shaped tree. The trouble is that the recursive call answers only "is this subtree balanced?", and the parent also needs its height. So return both. A convenient encoding returns the height, or -1 for "unbalanced":

```
def is_balanced(root: TreeNode | None) -> bool:
    def height(node: TreeNode | None) -> int:
        """Height of node's subtree, or -1 if it isn't balanced."""
        if node is None:
            return 0
        lh, rh = height(node.left), height(node.right)
        if lh < 0 or rh < 0 or abs(lh - rh) > 1:
            return -1
        return 1 + max(lh, rh)

    return height(root) >= 0
```

The strengthened version is one pass, $O(n)$.

Diameter of Binary Tree (LC 543). The longest path between any two nodes either passes through the root or lies entirely inside one subtree. If it passes through a node, it's the deepest path down the left plus the deepest path down the right. So the recursive function returns **depth** (what the parent needs) while recording the best **diameter** seen (what the problem asks for):

```
def diameter_of_binary_tree(root: TreeNode | None) -> int:
    best = 0

    def depth(node: TreeNode | None) -> int:
        nonlocal best
        if node is None:
            return 0
        left, right = depth(node.left), depth(node.right)
        best = max(best, left + right)       # longest path bending at this node
        return 1 + max(left, right)          # longest path going down from here

    depth(root)
    return best
```

Binary Tree Maximum Path Sum (LC 124) has exactly the same shape, with sums instead of lengths — and one twist: a branch with a negative sum should be dropped, so each child contributes `max(0, gain)`.

```python
def max_path_sum(root: TreeNode) -> int:
    best = root.val

    def gain(node: TreeNode | None) -> int:
        nonlocal best
        if node is None:
            return 0
        left = max(0, gain(node.left))
        right = max(0, gain(node.right))
        best = max(best, node.val + left + right)   # path bending here
        return node.val + max(left, right)          # path continuing to the parent

    gain(root)
    return best
```

This **"return one thing, record another"** pattern is the single most important idea for hard tree problems. The value returned to the parent must describe a path that can be *extended* upward (so it can use only one branch); the value recorded globally can *bend* at the current node.

Traversals

Visiting every node is the foundation of everything else. The three depth-first orders differ only in *when* the node itself is processed relative to its subtrees:

Order	Sequence	Typical use
Pre-order	node, left, right	Copying or serializing a tree; top-down propagation
In-order	left, node, right	BST in sorted order
Post-order	left, right, node	Anything needing both subtrees' answers first: height, deletion, evaluation

```python
def inorder(root: TreeNode | None) -> list[int]:
    out: list[int] = []

    def visit(node: TreeNode | None) -> None:
        if node:
            visit(node.left)
            out.append(node.val)       # move this line up for pre-order, down for post-order
            visit(node.right)

    visit(root)
    return out
```

All three are $O(n)$ time and $O(h)$ space for the recursion stack, where h is the height.

Iterative traversals

Recursion uses the call stack implicitly; an explicit stack does the same job and avoids Python's recursion limit on deep trees. Pre-order is simplest — push the right child before the left so the left is processed first:

```
def preorder_iter(root: TreeNode | None) -> list[int]:
    out, stack = [], [root] if root else []
    while stack:
        node = stack.pop()
        out.append(node.val)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
    return out
```

In-order must first dive left as far as possible, remembering the path, then process and turn right:

```
def inorder_iter(root: TreeNode | None) -> list[int]:
    out: list[int] = []
    stack: list[TreeNode] = []
    node = root
    while node or stack:
        while node:                    # dive left, remembering the path
            stack.append(node)
            node = node.left
        node = stack.pop()             # leftmost unvisited node
        out.append(node.val)
        node = node.right              # then its right subtree
    return out
```

Post-order is easiest as "reverse of a modified pre-order": visit node, right, left, then reverse the output.

(Morris traversal achieves $O(1)$ extra space by temporarily threading right pointers back to ancestors. It's clever, rarely necessary, and modifies the tree during the walk.)

Level-Order Traversal (BFS)

Visiting nodes level by level uses a **queue** instead of a stack. Processing the queue in batches — one batch per level — gives per-level results:

```
from collections import deque


def level_order(root: TreeNode | None) -> list[list[int]]:
    """LC 102."""
    if root is None:
        return []
    levels, queue = [], deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):          # exactly the nodes of this level
            node = queue.popleft()
            level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        levels.append(level)
    return levels
```

Many problems are one-line variations: the **right side view** (LC 199) is the last element of each level; **zigzag**

order (LC 103) reverses every other level; **minimum depth** (LC 111) is the level of the first leaf BFS reaches — and BFS finds it without exploring deeper levels, which DFS can't promise.

Top-Down vs. Bottom-Up

There are two ways to move information through a tree recursion, and recognizing which one a problem needs is half the solution.

Top-down (pre-order): pass information **down** as arguments — the depth so far, the sum so far, the allowed range of values. Each node receives context from its ancestors.

```python
def has_path_sum(root: TreeNode | None, target: int) -> bool:
    """LC 112: is there a root-to-leaf path summing to target?"""
    if root is None:
        return False
    remaining = target - root.val
    if root.left is None and root.right is None:
        return remaining == 0
    return has_path_sum(root.left, remaining) or has_path_sum(root.right, remaining)
```

Bottom-up (post-order): return information **up** as results — height, subtree size, subtree sum, "is this subtree valid." Each node combines what its children report. `max_depth`, `is_balanced`, and `diameter` above are all bottom-up.

Some problems need both: pass the running state down, and collect results as the recursion unwinds. **Path Sum II** (LC 113) passes the current path down, appends at each node, and **undoes** the append on the way back up — the backtracking pattern (see Combinatorial Search).

Path Sum III (LC 437) counts downward paths (not necessarily from the root) summing to a target. Along any root-to-node path, this is exactly "count subarrays with sum k" — so it's solved with the prefix-sum-plus-hash-map idea from Two Pointers & Sliding Window, passing the prefix counts down and removing each prefix as the recursion leaves its node:

```python
def path_sum_iii(root: TreeNode | None, target: int) -> int:
    counts = {0: 1}                    # prefix sums on the current root-to-node path

    def dfs(node: TreeNode | None, prefix: int) -> int:
        if node is None:
            return 0
        prefix += node.val
        found = counts.get(prefix - target, 0)
        counts[prefix] = counts.get(prefix, 0) + 1
        found += dfs(node.left, prefix) + dfs(node.right, prefix)
        counts[prefix] -= 1            # leaving this node: its prefix is no longer on the path
        return found

    return dfs(root, 0)
```

Binary Search Trees

Sorted arrays offer fast search ($O(\log n)$ by binary search) but slow updates ($O(n)$ to shift elements). Linked lists offer fast updates but slow search. Binary search needs quick access to the element in the middle of the remaining range — and a **binary search tree** provides it by storing the median of each range at the top, as a linked structure with two pointers per node.

> ***BST property:*** *for every node x, all keys in x's left subtree are less than x, and all keys in its right subtree are greater.*

Note the word *all*. It's not enough for each node to be bigger than its left child: every key in the entire left subtree must be smaller.

Search, minimum, successor

Search compares the key with the root and descends into the only subtree that could contain it. The minimum is the leftmost node; the maximum is the rightmost. All take $O(h)$.

```python
def search_bst(root: TreeNode | None, key: int) -> TreeNode | None:
    node = root
    while node and node.val != key:
        node = node.left if key < node.val else node.right
    return node


def insert_bst(root: TreeNode | None, key: int) -> TreeNode:
    """Insert where an unsuccessful search for key falls off the tree."""
    if root is None:
        return TreeNode(key)
    if key < root.val:
        root.left = insert_bst(root.left, key)
    else:
        root.right = insert_bst(root.right, key)
    return root
```

An **in-order traversal of a BST visits keys in sorted order** — this follows directly from the BST property, by induction. It makes the k-th smallest element (LC 230) an in-order traversal that stops after k nodes, and validation a check that the in-order sequence is strictly increasing.

Validation (LC 98)

The classic bug is to check only that each node is between its two children. That misses violations deeper down:

```
      5
     / \
    1   6        each node is fine relative to its children,
       / \       but 3 is in 5's RIGHT subtree and 3 < 5
      3   7
```

The fix is to pass **bounds** down (top-down): every node in the right subtree of 5 must be greater than 5.

```python
def is_valid_bst(root: TreeNode | None) -> bool:
    def valid(node: TreeNode | None, lo: float, hi: float) -> bool:
        if node is None:
```

```
            return True
        if not lo < node.val < hi:
            return False
        return valid(node.left, lo, node.val) and valid(node.right, node.val, hi)

    return valid(root, float("-inf"), float("inf"))
```

Deletion (LC 450)

Deleting a node has three cases:

1. **No children:** just remove it.
2. **One child:** replace it with that child. The child's subtree already sits entirely on the correct side of the parent, so the BST property survives.
3. **Two children:** the tricky case. Replace the node's key with its **in-order successor** — the smallest key in its right subtree, which is the leftmost node there. That successor has no left child, so deleting it from the right subtree falls into case 1 or 2.

```
def delete_node(root: TreeNode | None, key: int) -> TreeNode | None:
    if root is None:
        return None
    if key < root.val:
        root.left = delete_node(root.left, key)
    elif key > root.val:
        root.right = delete_node(root.right, key)
    else:
        if root.left is None:
            return root.right
        if root.right is None:
            return root.left
        successor = root.right
        while successor.left:
            successor = successor.left
        root.val = successor.val
        root.right = delete_node(root.right, successor.val)
    return root
```

How good are binary search trees?

Every operation costs $O(h)$. The height h ranges from $\log_2 n$ (perfectly balanced) to n (a path) — and which one you get depends entirely on the **insertion order**, which the data structure doesn't control. Insert keys in sorted order, a very common situation, and each new key becomes the right child of the previous one: the BST degenerates into a linked list with $O(n)$ operations.

If the keys arrive in **random** order, the tree is good: the average depth of a node is about $2 \ln n \approx 1.39 \log_2 n$ — only 39% more than in a perfectly balanced tree — and the height is $O(\log n)$ with high probability. (The analysis is the same as for randomized quicksort: the root of a random BST plays the role of the first pivot.) But "random order" is an assumption about the user, not a guarantee.

Balanced search trees (AVL trees, red–black trees, B-trees, skip lists) restructure themselves slightly on each update to guarantee $h = O(\log n)$, so every operation is $O(\log n)$ in the worst case. You should know they exist and what they guarantee; you will almost never need to implement one. For algorithm analysis, assume a balanced tree is available as a black box.

Balanced trees in Python

Python has no balanced BST in its standard library. The usual substitutes: `bisect` on a sorted list (fast search, $O(n)$ insert — often fine in practice because the shifting is a fast `memmove`), `heapq` if you only need the minimum, or the third-party `sortedcontainers.SortedList` ($O(\log n)$-ish everything), which LeetCode makes available.

Take-Home Lesson

Picking the *wrong* data structure can be disastrous — an unbalanced BST on sorted input is a linked list in disguise. Picking the *very best* one is usually less critical, because several good choices perform similarly. Aim first to avoid the disaster.

Stop and Think: Sorting with a search tree

Problem: You have a balanced BST supporting insert, delete, search, minimum, maximum, successor, and predecessor, each in $O(\log n)$. How can you sort n numbers in $O(n \log n)$ using only (a) insert and in-order traversal? (b) minimum, successor, and insert? (c) minimum, insert, and delete?
Solution: Every version begins by inserting all n items: $O(n \log n)$. Then:

- (a) An in-order traversal lists the keys in sorted order: $O(n)$.
- (b) Start at the minimum and call successor $n - 1$ times: $O(n \log n)$.
- (c) Repeatedly find and delete the minimum: $O(n \log n)$. This is heapsort's strategy, with a tree standing in for the heap.

In every case, building the structure is the bottleneck. And (c) shows that a balanced BST can do everything a priority queue can.

Lowest Common Ancestor

The **lowest common ancestor** (LCA) of nodes p and q is the deepest node that has both of them as descendants (a node counts as its own descendant).

In a BST (LC 235), the BST property steers the search. If both p and q are smaller than the current node, the LCA is in the left subtree; if both are larger, it's in the right. Otherwise they split here (or one of them *is* here), so this is the LCA.

```python
def lca_bst(root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
    node = root
    while True:
        if p.val < node.val and q.val < node.val:
            node = node.left
        elif p.val > node.val and q.val > node.val:
            node = node.right
        else:
            return node
```

In a general binary tree (LC 236), there are no values to steer by, so ask each subtree bottom-up: *does it contain p or q?* The recursive function returns p or q if it finds one, the LCA if it finds both, and `None` otherwise.

```python
def lca(root: TreeNode | None, p: TreeNode, q: TreeNode) -> TreeNode | None:
    if root is None or root is p or root is q:
        return root
```

```
    left = lca(root.left, p, q)
    right = lca(root.right, p, q)
    if left and right:          # p and q found in different subtrees: we're the split point
        return root
    return left or right        # pass up whichever was found (or None)
```

Why is it correct when p is an ancestor of q? The recursion returns p as soon as it reaches p, without looking for q below it. That's the right answer: q is guaranteed to exist in the tree, and if it isn't found elsewhere, it must be under p.

Building and Serializing Trees

Why one traversal isn't enough

The pre-order sequence `[1, 2, 3]` could come from five different three-node trees. A single traversal loses the shape. Two traversals can pin it down: **pre-order + in-order** determine a binary tree with distinct values uniquely. Pre-order tells you the root (its first element); finding that root in the in-order sequence splits everything else into the left and right subtrees (LC 105).

```
def build_tree(preorder: list[int], inorder: list[int]) -> TreeNode | None:
    index = {v: i for i, v in enumerate(inorder)}
    next_pre = 0

    def build(lo: int, hi: int) -> TreeNode | None:
        """Build the subtree whose in-order values are inorder[lo:hi]."""
        nonlocal next_pre
        if lo >= hi:
            return None
        root = TreeNode(preorder[next_pre])
        next_pre += 1
        mid = index[root.val]
        root.left = build(lo, mid)          # left subtree comes next in pre-order
        root.right = build(mid + 1, hi)
        return root

    return build(0, len(inorder))
```

The hash map avoids an $O(n)$ search for each root, making the whole construction $O(n)$. (Pre-order + post-order is *not* enough in general: a node with a single child can't tell whether it's a left or right child.)

Serialization (LC 297)

To write a tree to a string and read it back, a single traversal *is* enough — if you also record the **empty subtrees**. With null markers, the pre-order sequence describes the shape completely:

```
def serialize(root: TreeNode | None) -> str:
    out: list[str] = []

    def walk(node: TreeNode | None) -> None:
        if node is None:
            out.append("#")
            return
```

```
        out.append(str(node.val))
        walk(node.left)
        walk(node.right)

    walk(root)
    return ",".join(out)


def deserialize(data: str) -> TreeNode | None:
    tokens = iter(data.split(","))

    def build() -> TreeNode | None:
        tok = next(tokens)
        if tok == "#":
            return None
        node = TreeNode(int(tok))
        node.left = build()
        node.right = build()
        return node

    return build()
```

The deserializer reads tokens in exactly the order the serializer wrote them, so no indices or lengths are needed. Serialization also gives a neat solution for finding duplicate subtrees (LC 652): two subtrees are identical exactly when their serializations are equal.

Common Mistakes

1. **Checking only immediate children for BST validity.** Pass bounds down, or check that the in-order sequence is strictly increasing.
2. **Recomputing heights at every node.** Calling `max_depth` inside another recursion makes the algorithm $O(n^2)$ on skewed trees. Return the height alongside the other result.
3. **Confusing "returned" and "recorded" values.** In diameter and max-path-sum problems, the parent can only extend a one-branch path; the bent path through a node is a candidate answer, not a return value.
4. **Forgetting the backtracking step.** When passing a mutable path or counter down, undo the change after the recursive calls return.
5. **Leaf vs. `None` confusion.** "Root-to-leaf" means stopping at nodes with no children, not at `None` — otherwise a node with one child counts as a path end.
6. **Deep recursion.** A 10^5-node skewed tree exceeds Python's default recursion limit. Use an explicit stack or BFS when depth can be large.
7. **Assuming a BST is balanced.** $O(\log n)$ holds only for balanced trees. Sorted insertions produce $O(n)$ height.

Practice Problems

Problem	Idea
Maximum Depth of Binary Tree (LC 104)	Bottom-up height
Invert Binary Tree (LC 226)	Swap children recursively
Same Tree (LC 100) / Subtree of Another Tree (LC 572)	Simultaneous recursion
Binary Tree Level Order Traversal (LC 102)	BFS in batches
Binary Tree Right Side View (LC 199)	Last node per level
Balanced Binary Tree (LC 110)	Return height or -1
Diameter of Binary Tree (LC 543)	Return depth, record diameter
Binary Tree Maximum Path Sum (LC 124)	Return gain, record bent path
Path Sum II (LC 113)	Top-down with backtracking
Path Sum III (LC 437)	Prefix sums on the root path
Validate Binary Search Tree (LC 98)	Bounds passed down
Kth Smallest Element in a BST (LC 230)	In-order, stop at k
Delete Node in a BST (LC 450)	Successor replacement
Lowest Common Ancestor of a BST (LC 235)	Steer by values
Lowest Common Ancestor of a Binary Tree (LC 236)	Bottom-up search
Construct Tree from Preorder and Inorder (LC 105)	Root from pre-order, split by in-order
Serialize and Deserialize Binary Tree (LC 297)	Pre-order with null markers
Count Good Nodes in Binary Tree (LC 1448)	Top-down running max

Graphs

A graph $G = (V, E)$ is a set of **vertices** together with a set of **edges** connecting pairs of them. That's all — and that's exactly why graphs are so powerful. Road networks, social networks, the web, circuits, course prerequisites, the moves of a puzzle, the states of a game, the pixels of an image: all of them are graphs once you decide what the vertices and edges are.

This chapter has two goals. The first is to master a small toolkit of classical algorithms — breadth-first search, depth-first search, topological sort, minimum spanning trees, shortest paths — each of which runs in (nearly) linear time. The second, and more important, is to learn to **model**: to look at a problem that doesn't mention graphs at all and see which graph, and which classical question about it, the problem is secretly asking.

> **Take-Home Lesson**
>
> Designing novel graph algorithms is very hard, so don't do it. Instead, design **graphs** that let you use classical algorithms. Most graph interview problems are one of a dozen standard algorithms applied to a cleverly chosen graph.

Flavors of Graphs

The first step in any graph problem is to identify what kind of graph you have, because it determines which algorithms apply.

Property	Question to ask	Why it matters
Directed vs. undirected	Does an edge (x, y) imply (y, x)?	Prerequisites, one-way streets, and "follows" relations are directed. Cycle detection and connectivity work differently.
Weighted vs. unweighted	Do edges have costs?	Unweighted shortest paths: BFS. Weighted: Dijkstra and friends.
Sparse vs. dense	Is E closer to V or to V^2?	Chooses the representation, and between some algorithms.
Cyclic vs. acyclic	Can you return to where you started?	Directed acyclic graphs (DAGs) admit topological order, which makes many hard problems easy. Trees are connected acyclic undirected graphs.
Implicit vs. explicit	Is the graph given, or generated by rules?	Grids, puzzle states, and word ladders are graphs you never build in full; you generate neighbors on demand.
Simple vs. non-simple	Self-loops (x, x)? Repeated edges?	Many implementations quietly assume a simple graph.

Representations

	Adjacency list	Adjacency matrix	Edge list
Space	$O(V + E)$	$O(V^2)$	$O(E)$
Is (x, y) an edge?	$O(\deg x)$	$O(1)$	$O(E)$
Iterate over neighbors of x	$O(\deg x)$	$O(V)$	$O(E)$
Best for	Almost everything	Dense graphs, Floyd–Warshall	Kruskal, Bellman–Ford

Take-Home Lesson

Adjacency lists are the right data structure for most applications of graphs. Traversal algorithms spend their time iterating over neighbors, and adjacency lists make that proportional to the number of edges that actually exist.

In Python, an adjacency list is a `list` of lists (vertices numbered $0 \dots n-1$) or a `defaultdict(list)` (arbitrary hashable labels):

```
from collections import defaultdict


def build_adjacency_list(n: int, edges: list[list[int]], directed: bool = False) ->
↪  list[list[int]]:
    graph: list[list[int]] = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        if not directed:
            graph[v].append(u)
    return graph


def build_weighted(edges: list[tuple[str, str, int]]) -> dict[str, list[tuple[str, int]]]:
    graph: dict[str, list[tuple[str, int]]] = defaultdict(list)
    for u, v, w in edges:
        graph[u].append((v, w))
    return graph
```

A **grid** is an implicit graph: each cell is a vertex, and its neighbors are computed from coordinates, never stored.

```
DIRECTIONS = [(0, 1), (1, 0), (0, -1), (-1, 0)]


def grid_neighbors(r: int, c: int, rows: int, cols: int):
    for dr, dc in DIRECTIONS:
        nr, nc = r + dr, c + dc
        if 0 <= nr < rows and 0 <= nc < cols:
            yield nr, nc
```

Traversal: The Master Algorithm

Visiting every vertex and edge systematically is the foundation of nearly every graph algorithm. The key idea is to **mark** vertices so that we never get lost going around in circles. Each vertex passes through three states:

1. **Undiscovered** — not yet seen.
2. **Discovered** — seen, but its edges haven't all been explored.
3. **Processed** — all its edges have been explored.

Keep the discovered-but-unprocessed vertices in a container. Repeatedly take a vertex out, and for each edge leading to an undiscovered vertex, mark that vertex discovered and add it to the container.

Why does this visit everything reachable? Suppose some reachable vertex is never visited, and let u be the **first** unvisited vertex on a path from the start. Its predecessor v on that path *was* visited — but when v was processed, it would have discovered u. Contradiction.

Every traversal takes $O(V + E)$ time: each vertex enters the container once, and each edge is examined once from each endpoint. That's optimal — you can't even *read* the graph faster. Anything you can do with one or two traversals is linear time.

The remarkable thing is that **the order of exploration is decided entirely by the container**:

- A **queue** (FIFO) explores the oldest discoveries first. Exploration radiates outward in rings from the start: **breadth-first search**.
- A **stack** (LIFO) explores the newest discoveries first. Exploration plunges down a path and backs up only when stuck: **depth-first search**.

For some problems the choice doesn't matter (reachability, counting components). For others it's everything: BFS finds shortest paths; DFS reveals cycles and dependency order.

Breadth-First Search

```
from collections import deque


def bfs(graph: list[list[int]], start: int) -> tuple[list[int], list[int]]:
    """Return (dist, parent): hop distance from start and the BFS tree (-1 = none)."""
    n = len(graph)
    dist = [-1] * n
    parent = [-1] * n
    dist[start] = 0
    queue = deque([start])
    while queue:
        u = queue.popleft()
        for v in graph[u]:
            if dist[v] == -1:              # undiscovered
                dist[v] = dist[u] + 1
                parent[v] = u
                queue.append(v)
    return dist, parent


def reconstruct_path(parent: list[int], target: int) -> list[int]:
    path = []
```

```
    while target != -1:
        path.append(target)
        target = parent[target]
    return path[::-1]
```

> ⚠ **Mark when you *enqueue*, not when you dequeue**
>
> Marking a vertex as seen only when it's popped lets the same vertex be enqueued many times — once from each neighbor that discovers it before it's processed. On dense graphs and grids, this can blow up both time and memory.

Shortest paths in unweighted graphs

BFS discovers vertices in order of increasing distance: all vertices at distance 1, then all at distance 2, and so on. So the first time a vertex is discovered, it's by a **shortest path**, and the parent pointers form a shortest-path tree rooted at the start. Two caveats: the tree gives shortest paths *from the start vertex only*, and only when **all edges have equal weight**.

Shortest Path in Binary Matrix (LC 1091) is BFS on an 8-connected grid; **Minimum Knight Moves** (LC 1197) is BFS on the implicit graph of knight moves.

Multi-source BFS

To compute each vertex's distance to the **nearest** of several sources, put all the sources in the queue at distance 0 and run a single BFS. It's as if a super-source were connected to all of them with zero-length edges. **Rotting Oranges** (LC 994), **Walls and Gates** (LC 286), and **01 Matrix** (LC 542) all work this way.

```
def oranges_rotting(grid: list[list[int]]) -> int:
    rows, cols = len(grid), len(grid[0])
    queue = deque((r, c) for r in range(rows) for c in range(cols) if grid[r][c] == 2)
    fresh = sum(row.count(1) for row in grid)
    minutes = 0
    while queue and fresh:
        minutes += 1
        for _ in range(len(queue)):                  # one ring = one minute
            r, c = queue.popleft()
            for nr, nc in grid_neighbors(r, c, rows, cols):
                if grid[nr][nc] == 1:
                    grid[nr][nc] = 2
                    fresh -= 1
                    queue.append((nr, nc))
    return -1 if fresh else minutes
```

Two-coloring: is the graph bipartite? (LC 785)

A graph is **bipartite** if its vertices can be colored with two colors so that every edge joins different colors. Color the start vertex arbitrarily; every neighbor is then forced to take the other color, and so on outward. BFS propagates the forced colors; if some edge ever joins two vertices of the same color, no two-coloring exists. (Equivalently: a graph is bipartite iff it has no odd-length cycle.)

```
def is_bipartite(graph: list[list[int]]) -> bool:
    color = [-1] * len(graph)
    for start in range(len(graph)):              # the graph may be disconnected
        if color[start] != -1:
```

```
            continue
        color[start] = 0
        queue = deque([start])
        while queue:
            u = queue.popleft()
            for v in graph[u]:
                if color[v] == -1:
                    color[v] = 1 - color[u]
                    queue.append(v)
                elif color[v] == color[u]:
                    return False
    return True
```

Implicit graphs: searching a state space

Many puzzles are shortest-path problems on a graph that's never written down. The vertices are **states**, the edges are **moves**, and BFS generates neighbors on the fly.

Open the Lock (LC 752): a state is a 4-digit combination; each move turns one wheel up or down, giving 8 neighbors. The whole graph has 10^4 vertices — small enough for BFS, though no one would build it explicitly.

```
def open_lock(deadends: list[str], target: str) -> int:
    dead = set(deadends)
    if "0000" in dead:
        return -1
    seen = {"0000"}
    queue = deque([("0000", 0)])
    while queue:
        state, moves = queue.popleft()
        if state == target:
            return moves
        for i in range(4):
            d = int(state[i])
            for step in (1, -1):
                nxt = state[:i] + str((d + step) % 10) + state[i + 1:]
                if nxt not in seen and nxt not in dead:
                    seen.add(nxt)
                    queue.append((nxt, moves + 1))
    return -1
```

Word Ladder (LC 127) is the same idea with words as states and single-letter changes as moves. When the state space is large, **bidirectional BFS** — searching from both ends and stopping when the frontiers meet — explores roughly $2b^{d/2}$ states instead of b^d.

Depth-First Search

DFS is BFS with a stack instead of a queue — and since recursion *is* a stack, DFS has a beautifully short recursive form:

```
def dfs_order(graph: list[list[int]], start: int) -> list[int]:
    visited: set[int] = set()
    order: list[int] = []
```

```
    def dfs(u: int) -> None:
        visited.add(u)
        order.append(u)                  # "process vertex early": pre-order
        for v in graph[u]:
            if v not in visited:
                dfs(v)
        # "process vertex late" would go here: post-order

    dfs(start)
    return order
```

DFS is essentially the same procedure as **backtracking** (see Combinatorial Search): advance whenever possible, back up only when every option is exhausted.

> **Recursion depth**
>
> Recursive DFS on a long path of 10^5 vertices exceeds Python's recursion limit. For large graphs, either use an explicit stack or `sys.setrecursionlimit` (and be aware that very deep recursion can still crash the interpreter).

Why DFS is special: entry/exit times and edge classes

Give DFS a clock that ticks each time a vertex is entered or exited. If x is an ancestor of y in the DFS tree, then x is entered before y and exited after y — the intervals [entry, exit] nest like parentheses.

More importantly, DFS sorts the edges of an **undirected** graph into just two classes:

- **Tree edges**, which discover new vertices.
- **Back edges**, which lead to an **ancestor** in the DFS tree.

There are no "cross edges" between unrelated branches. Why? If u has an edge to v and v is still undiscovered when u is explored, DFS will discover v *from* u — making it a tree edge. By the time DFS leaves u, everything reachable from u has been explored.

> **Take-Home Lesson**
>
> DFS organizes vertices by entry and exit times, and edges into tree and back edges. That organization — not the traversal itself — is where DFS gets its power: cycle detection, topological sorting, articulation points, and strongly connected components all come from it.

Connected components and flood fill

Starting a traversal from each still-unvisited vertex partitions the graph into its **connected components**. A surprising number of problems reduce to counting or labeling components. **Number of Islands** (LC 200) is exactly this on a grid:

```
def num_islands(grid: list[list[str]]) -> int:
    rows, cols = len(grid), len(grid[0])

    def sink(r: int, c: int) -> None:
        stack = [(r, c)]                        # iterative: islands can be large
        grid[r][c] = "0"
        while stack:
            r, c = stack.pop()
            for nr, nc in grid_neighbors(r, c, rows, cols):
```

```
                if grid[nr][nc] == "1":
                    grid[nr][nc] = "0"          # mark on discovery
                    stack.append((nr, nc))

    islands = 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == "1":
                islands += 1
                sink(r, c)
    return islands
```

Reverse the question. In **Surrounded Regions** (LC 130), rather than asking "is this region surrounded?", flood-fill from every border `O` and mark what's reachable; everything unmarked is surrounded. In **Pacific Atlantic Water Flow** (LC 417), rather than simulating water flowing downhill from every cell, flow *uphill* from each ocean's border and intersect the two reachable sets. Starting from the targets is often dramatically simpler than starting from every source.

Cycle detection

A lot depends on whether a graph has cycles. A topological order exists only in a DAG, a tree is a connected graph with no cycle, and a cycle in a "waits for" graph is a deadlock. Directed and undirected graphs use the same test: **a graph has a cycle iff DFS finds a back edge**. The only difference is how you recognize a back edge.

Undirected graphs. An undirected DFS has no cross edges (see above), so every edge that doesn't discover a new vertex leads to an ancestor. A cycle therefore exists iff DFS reaches an **already-visited vertex other than its parent**. The parent is excluded because every undirected edge appears in both adjacency lists. From v you always see the edge back to the u that discovered it, but that's the tree edge you just walked, not a cycle. The test is correct in both directions:

- **Back edge ⇒ cycle.** If (u, v) is a back edge, v is an ancestor of u. The tree path $v \to \ldots \to u$ plus the edge (u, v) forms a cycle.
- **No back edge ⇒ no cycle.** If every edge is a tree edge, the graph is a forest.

```
def has_cycle_undirected(graph: list[list[int]]) -> bool:
    visited = [False] * len(graph)

    def dfs(current: int, parent: int) -> bool:
        visited[current] = True
        for neighbor in graph[current]:
            if neighbor == parent:
                continue                        # the tree edge we arrived on
            if visited[neighbor]:
                return True                     # back edge to an ancestor
            if dfs(neighbor, current):
                return True
        return False

    for start in range(len(graph)):             # the graph may be disconnected
        if not visited[start] and dfs(start, -1):
            return True
    return False
```

The test skips the parent by **vertex**, which assumes a simple graph. In a multigraph, two parallel edges between u and v form a cycle of length 2, but `neighbor == parent` skips both copies. If the input can

contain duplicate edges, skip the parent only once, or identify edges by index instead of by endpoint. Self-loops are caught: u is visited and isn't its own parent.

Two shortcuts avoid writing a DFS at all:

- **Counting.** A forest with V vertices and C components has exactly $V - C$ edges, and any additional edge closes a cycle. So a simple undirected graph has a cycle iff $E > V - C$. **Graph Valid Tree** (LC 261) reduces to "$E = V - 1$ and the graph is connected."
- **Union-find.** When edges arrive one at a time, an edge closes a cycle iff its endpoints are already in the same component. Kruskal's algorithm uses this test to reject edges, and **Redundant Connection** (LC 684) asks for exactly the edge that closes the first cycle. See Trie & Union Find.

Directed graphs. The undirected test fails here: in the graph $a \to b$, $a \to c$, $c \to b$, DFS reaches b a second time, but there's no cycle. A directed DFS sorts edges into **four** classes, not two. Each edge `current → neighbor` is classified by the state of `neighbor` at the moment DFS explores the edge:

State of `neighbor`	Edge class	Meaning
White: undiscovered	Tree edge	DFS discovers `neighbor` from `current`
Gray: discovered, not finished	**Back edge**	`neighbor` is an ancestor, still on the recursion stack
Black: finished	Forward or cross edge	`neighbor` is a finished descendant, or belongs to a finished branch

Only back edges signal a cycle:

- **Back edge ⇒ cycle.** A gray `neighbor` is an ancestor of `current`. The tree path `neighbor → … → current` plus the edge `current → neighbor` forms a cycle.
- **Cycle ⇒ back edge.** Let v be the first vertex of the cycle that DFS discovers. At that moment, every other vertex on the cycle is white and reachable from v, so DFS discovers all of them before it finishes v. That includes the vertex u whose edge closes the cycle, $u \to v$. When DFS explores that edge, v is still gray, so it's a back edge.
- **Edges to black vertices are harmless.** A finished vertex has already explored everything reachable from it. If it could reach back into the current path, that exploration would already have found a back edge.

Three colors are the minimum needed to tell these classes apart. A single `visited` set merges gray and black, and so reports forward and cross edges as cycles. Gray vertices are exactly the current DFS path, so keeping that path on a list makes it easy to return the **cycle itself** rather than just a yes or no:

```
WHITE, GRAY, BLACK = 0, 1, 2                    # undiscovered, on the current path, finished


def find_cycle_directed(graph: list[list[int]]) -> list[int] | None:
    """Return the vertices of a directed cycle in order, or None if the graph is a DAG."""
    color = [WHITE] * len(graph)
    path: list[int] = []                        # the gray vertices, from the DFS root down

    def dfs(current: int) -> list[int] | None:
        color[current] = GRAY
        path.append(current)
        for neighbor in graph[current]:
            if color[neighbor] == GRAY:         # back edge: the cycle runs from neighbor to
             ↪  current
```

```
                return path[path.index(neighbor):]
            if color[neighbor] == WHITE:
                cycle = dfs(neighbor)
                if cycle is not None:
                    return cycle
            # BLACK: forward or cross edge, harmless
        path.pop()
        color[current] = BLACK
        return None

    for start in range(len(graph)):              # the graph may be disconnected
        if color[start] == WHITE:
            cycle = dfs(start)
            if cycle is not None:
                return cycle
    return None
```

If you only need a yes or no, drop `path` and return booleans. The same DFS also gives you a topological order for free: record each vertex as it turns black, then reverse the list (see Via DFS: reverse post-order). The colors also answer **Find Eventual Safe States** (LC 802), which asks for the vertices from which no cycle can be reached. A vertex is safe iff its DFS turns it black without ever meeting a gray vertex.

Without recursion, Kahn's algorithm detects directed cycles too: a cycle exists iff some vertices never reach in-degree zero (see Via in-degrees: Kahn's algorithm). Not every leftover vertex is on a cycle, though; some are only downstream of one. **Course Schedule** (LC 207) is solved either way.

Situation	Method	Example
Undirected, whole graph given	DFS that skips the parent, or $E > V - C$	Graph Valid Tree (LC 261)
Undirected, edges arrive one at a time	Union-find	Redundant Connection (LC 684)
Directed, yes/no or a topological order	Three-color DFS, or Kahn's algorithm	Course Schedule (LC 207)
Directed, which vertices can reach a cycle	Three-color DFS	Find Eventual Safe States (LC 802)
Every vertex has out-degree 1 (linked list, functional graph)	Floyd's tortoise and hare, $O(1)$ space	Linked List Cycle (LC 141), Find the Duplicate Number (LC 287)
Is there an *odd* cycle?	Two-coloring	Is Graph Bipartite? (LC 785)

Weak points: bridges and articulation vertices

An **articulation vertex** is one whose removal disconnects the graph; a **bridge** is such an edge. They are single points of failure in a network. DFS finds all of them in one pass, tracking two numbers per vertex:

- discovery_order(v): when DFS first reaches v.
- low(v): the smallest discovery order reachable from v's subtree by going down tree edges and then up **at most one** back edge. It measures how high v's subtree can climb without using the edge to v's parent.

Because an undirected DFS has no cross edges, deleting a vertex u can only cut off **subtrees of its children**: a child's subtree stays attached only if one of its back edges climbs above u. Leaves of the DFS tree are never articulation vertices, since deleting one leaves the rest of the tree intact. For every other vertex, one of three cases applies:

1. **Root case.** The root is an articulation vertex iff it has **two or more children** in the DFS tree. With no cross edges, the only connection between its subtrees is the root itself. A root with one child can be deleted safely. The `low` test below doesn't work at the root: nothing sits above it, so low(v) $\geq$ discovery_order(root) always holds.
2. **Parent case.** If low(v) = discovery_order(u) for some child v, then v's subtree can climb back to u but no higher. Deleting u strands that subtree, so u is an articulation vertex (unless it's the root, which case 1 handles).
3. **Bridge case.** If low(v) = discovery_order(v), then nothing below v even reaches u, so (u, v) is a bridge. Both of its endpoints are articulation vertices, except for one that is a leaf (v with no children) or a root with only this child (u).

Cases 2 and 3 combine into one test: a **non-root** u is an articulation vertex iff some child v has low(v) $\geq$ discovery_order(u). That covers v in case 3 too, since v's own children satisfy the test against v. A vertex can pass the test for several children, so record the results in a boolean array and not a list:

```
def articulation_vertices(n: int, edges: list[list[int]]) -> list[int]:
    graph = build_adjacency_list(n, edges)
    discovery_order = [-1] * n            # -1 = unvisited
    low = [0] * n                         # min discovery_order reachable with ≤ 1 back edge
    is_articulation = [False] * n
    discovery_time = 0

    def dfs(current: int, parent: int) -> None:
        nonlocal discovery_time
        discovery_order[current] = low[current] = discovery_time
        discovery_time += 1
        children = 0
        for neighbor in graph[current]:
            if neighbor == parent:
                continue
            if discovery_order[neighbor] == -1:      # tree edge
                children += 1
                dfs(neighbor, current)
                low[current] = min(low[current], low[neighbor])
                if parent != -1 and low[neighbor] >= discovery_order[current]:
                    is_articulation[current] = True  # parent or bridge case
            else:                                    # back edge
                low[current] = min(low[current], discovery_order[neighbor])
        if parent == -1 and children >= 2:
            is_articulation[current] = True          # root case

    for s in range(n):
        if discovery_order[s] == -1:
            dfs(s, -1)
    return [vertex for vertex in range(n) if is_articulation[vertex]]
```

Bridges come from the same DFS with a stricter test: a tree edge (u, v) is a bridge iff low(v) $>$ discovery_order(u), meaning no back edge from v's subtree reaches u or any of its ancestors. **Critical Connections** (LC 1192) asks for exactly these:

```
def critical_connections(n: int, connections: list[list[int]]) -> list[list[int]]:
    graph = build_adjacency_list(n, connections)
    discovery_order = [-1] * n                       # -1 = unvisited
    low = [0] * n                                    # min discovery_order reachable with ≤ 1
 ↪  back edge
```

```
    bridges = []
    discovery_time = 0

    def dfs(current: int, parent: int) -> None:
        nonlocal discovery_time
        discovery_order[current] = low[current] = discovery_time
        discovery_time += 1
        for neighbor in graph[current]:
            if neighbor == parent:
                continue
            if discovery_order[neighbor] == -1:      # tree edge
                dfs(neighbor, current)
                low[current] = min(low[current], low[neighbor])
                if low[neighbor] > discovery_order[current]:
                    bridges.append([current, neighbor])
            else:                                    # back edge
                low[current] = min(low[current], discovery_order[neighbor])

    for s in range(n):
        if discovery_order[s] == -1:
            dfs(s, -1)
    return bridges
```

LC 1192 allows 10^5 vertices, so on a long chain this recursion exceeds Python's default limit (see the recursion-depth warning above).

Topological Sorting

A **topological order** of a directed graph lists its vertices so that every edge points **forward**, from earlier to later. If edges mean "must happen before," a topological order is a valid schedule. It exists **iff the graph is a DAG**: a cycle would require going forward forever and still returning to the start.

Topological sorting is the most important operation on DAGs, because it lets you process each vertex after everything that points to it. Shortest paths, longest paths, and counting paths in a DAG all become a single left-to-right pass in topological order — even with negative weights.

Via DFS: reverse post-order

List the vertices in the order DFS *finishes* them, then reverse. Why is that a topological order? Consider any edge (x, y) at the moment DFS explores it from x:

- If y is undiscovered, DFS finishes y before returning to finish x. ✓
- If y is already finished, it finished before x will. ✓
- If y is discovered but unfinished, it's an ancestor on the stack — a back edge, which means a cycle, which is impossible in a DAG.

In every legal case, y finishes before x, so after reversing, x comes before y. In code, it's a plain DFS that records each vertex as it finishes:

```
def topological_sort_dfs(graph: list[list[int]]) -> list[int]:
    """Assumes a DAG; use the three-color DFS from Cycle detection if it may not be one."""
    visited = [False] * len(graph)
    finished: list[int] = []
```

```
    def dfs(current: int) -> None:
        visited[current] = True
        for neighbor in graph[current]:
            if not visited[neighbor]:
                dfs(neighbor)
        finished.append(current)                # post-order: every successor is already in

    for start in range(len(graph)):
        if not visited[start]:
            dfs(start)
    return finished[::-1]
```

Via in-degrees: Kahn's algorithm

A vertex with no incoming edges can safely go first. Remove it, which may free up others, and repeat. If vertices remain but none has in-degree zero, they contain a cycle.

```
def topological_sort(n: int, edges: list[list[int]]) -> list[int] | None:
    """Kahn's algorithm. edges[i] = [u, v] means u must come before v."""
    graph: list[list[int]] = [[] for _ in range(n)]
    indegree = [0] * n
    for u, v in edges:
        graph[u].append(v)
        indegree[v] += 1
    queue = deque(u for u in range(n) if indegree[u] == 0)
    order = []
    while queue:
        u = queue.popleft()
        order.append(u)
        for v in graph[u]:
            indegree[v] -= 1
            if indegree[v] == 0:
                queue.append(v)
    return order if len(order) == n else None      # None: there's a cycle
```

Course Schedule (LC 207) asks whether the order exists; **Course Schedule II** (LC 210) asks for it. Kahn's algorithm also answers "what's the minimum number of semesters?" (process the queue level by level, as in BFS) and "is the order unique?" (the queue never holds more than one vertex).

Stop and Think: Ordering the fragments

Problem: A DNA sequencing experiment produces fragments. For some pairs you know that fragment a must lie to the left of fragment b. Find a left-to-right ordering consistent with all the constraints.
Solution: Make each fragment a vertex, and add a directed edge $a \to b$ for each "left of" constraint. A consistent ordering is exactly a topological sort of this graph; if the graph has a cycle, the constraints are contradictory. **Alien Dictionary** (LC 269) is the same problem: compare adjacent words to extract "letter x comes before letter y" edges, then topologically sort the letters.

Strongly connected components

A directed graph is **strongly connected** if every vertex can reach every other. Its **strongly connected components** (SCCs) are the maximal strongly connected pieces. Contracting each SCC to a single vertex gives the **condensation**, which is always a DAG: a cycle through two components would make them one component.

Key idea: a DFS started in a sink component visits exactly that component. A sink component has no

edges leaving it, so the DFS cannot escape, and strong connectivity means it reaches every vertex inside. Starting each DFS in a sink component therefore produces one SCC per search tree. Once a component is marked, the DFS ignores it, so another component becomes a sink, and the process repeats. The only problem is finding a vertex in a sink component without already knowing the components. Finish times solve it.

> ***Finish-time lemma.*** *Write $f(C)$ for the latest finish time of any vertex in component C. If the condensation has an edge $C \to C'$, then $f(C) > f(C')$.*

Proof: consider which of the two components DFS enters first.

- **C first, at vertex x.** At that moment every vertex of C and C' is undiscovered and reachable from x, so all of them become descendants of x and finish before x.
- **C' first.** C is not reachable from C': together with the edge $C \to C'$, that would put both on a cycle. So DFS finishes all of C' before it discovers any vertex of C.

So the vertex that finishes **last** lies in a **source** component, one with no incoming edges. We need a sink, and reversing every edge provides one. The reversed graph G^R has the **same SCCs** (a reversed cycle is still a cycle), but every condensation edge points the other way, so sources of G are sinks of G^R.

Kosaraju's algorithm therefore makes two passes:

1. DFS on G, recording vertices in the order they finish.
2. DFS on G^R, starting from unassigned vertices in **decreasing** finish order. Each search tree is one SCC.

Pass 2 stays correct after the first component. Let x be the unassigned vertex with the latest finish, and let C be its component. In G^R, the edges leaving C lead to components C' that have an edge $C' \to C$ in G. By the lemma, $f(C') > f(C)$, so C' has already been assigned. The search from x therefore reaches only C, plus already-assigned vertices that it skips. Both passes and building G^R take $O(V + E)$.

Stop and Think: Why not skip the reversal?

Problem: The lemma says the last vertex to finish lies in a source component. Doesn't that make the vertex that finishes **first** lie in a sink component? Then pass 2 could run on G itself, in increasing finish order.
Solution: No. The lemma compares only the *latest* finish time of each component; it says nothing about the earliest. Take the edges $0 \to 1$, $1 \to 0$, $0 \to 2$, and run DFS from 0, visiting 1 first. The finish order is $1, 2, 0$. Vertex 1 finishes first, but its component $\{0, 1\}$ is a source: a DFS from 1 on G reaches 2 as well and merges two SCCs. Only the latest finish is reliable, which is why the algorithm reverses the graph rather than the order.

```python
def strongly_connected_components(graph: list[list[int]]) -> list[list[int]]:
    n = len(graph)
    finished: list[int] = []
    seen = [False] * n

    for s in range(n):                              # pass 1: finish order on G
        if seen[s]:
            continue
        seen[s] = True
        stack = [(s, iter(graph[s]))]
        while stack:
            u, it = stack[-1]
            v = next(it, None)
            if v is None:                           # all neighbors explored: u finishes
                stack.pop()
                finished.append(u)
```

```
            elif not seen[v]:
                seen[v] = True
                stack.append((v, iter(graph[v])))

    reverse: list[list[int]] = [[] for _ in range(n)]
    for u in range(n):
        for v in graph[u]:
            reverse[v].append(u)

    comp = [-1] * n
    components = []
    for s in reversed(finished):                    # pass 2: DFS on reversed G
        if comp[s] != -1:
            continue
        comp[s] = len(components)
        members, stack = [], [s]
        while stack:
            u = stack.pop()
            members.append(u)
            for v in reverse[u]:
                if comp[v] == -1:
                    comp[v] = comp[s]
                    stack.append(v)
        components.append(members)
    return components
```

Pass 1 needs true finish times, so it cannot use the plain stack from flood fill: popping a vertex there says nothing about whether its descendants are done. Instead, each stack entry holds a vertex together with an iterator over its unexplored neighbors. A vertex finishes when its iterator runs out, which is exactly when the recursive call would return. Pass 2 only collects reachable vertices, so the plain stack is enough there.

Tarjan's algorithm finds the same components in a single DFS, using low-link values like those in Weak points. Kosaraju's two passes are easier to get right in an interview.

Minimum Spanning Trees

A **spanning tree** of a connected graph is a subset of $V - 1$ edges connecting all the vertices without a cycle. The **minimum spanning tree** (MST) is the one of least total weight: the cheapest way to connect everything — cities with roads, houses with cable, points with wire.

Two greedy algorithms find it. Greedy algorithms are usually suspect (see Intervals), so both come with a proof. The key fact behind both:

> ***Cut property.*** *Split the vertices into two groups. The lightest edge crossing between them belongs to some MST.*

Proof by exchange: take an MST that doesn't contain the lightest crossing edge e. Adding e creates a cycle, and that cycle must cross the split a second time through some edge f. Since e is the lightest crossing edge, $w(e) \le w(f)$. Swapping f for e gives a spanning tree no heavier than before — so an MST containing e exists.

Prim's algorithm

Grow a single tree from any start vertex. Repeatedly add the **cheapest edge leaving the tree** — the lightest edge across the cut (tree, everything else). A heap holds the candidate edges; stale entries for vertices already in the tree are skipped.

```
import heapq


def prim_mst_weight(n: int, graph: list[list[tuple[int, int]]]) -> int:
    """graph[u] = [(v, w), ...] for an undirected connected graph."""
    in_tree = [False] * n
    heap = [(0, 0)]                                  # (edge weight, vertex)
    total = 0
    while heap:
        w, u = heapq.heappop(heap)
        if in_tree[u]:
            continue                                 # stale entry
        in_tree[u] = True
        total += w
        for v, wv in graph[u]:
            if not in_tree[v]:
                heapq.heappush(heap, (wv, v))
    return total
```

$O(E \log E)$: this is the **lazy** variant, which pushes every edge leaving the tree instead of decreasing keys, so the heap can hold $O(E)$ entries. Since $E < V^2$, $\log E < 2 \log V$, so this matches the $O(E \log V)$ of the decrease-key version.

For **dense** graphs — such as **Min Cost to Connect All Points** (LC 1584), where every pair of points is an edge — $E \approx V^2/2$, and the heap only adds overhead. Drop it: keep, for every vertex outside the tree, the cheapest edge connecting it to the tree, and find the next vertex with a linear scan.

```
from typing import Callable


def prim_dense(n: int, cost: Callable[[int, int], int]) -> int:
    """MST weight of the complete graph on n vertices, with edge weights cost(u, v)."""
    in_tree = [False] * n
    cheapest = [float("inf")] * n                    # lightest edge from the tree to each vertex
    cheapest[0] = 0
    total = 0
    for _ in range(n):
        u = min((v for v in range(n) if not in_tree[v]), key=lambda v: cheapest[v])
        in_tree[u] = True
        total += cheapest[u]
        for v in range(n):                           # u joined the tree: update the other vertices
            if not in_tree[v]:
                cheapest[v] = min(cheapest[v], cost(u, v))
    return total
```

V rounds of $O(V)$ work: $O(V^2)$ time and $O(V)$ space, with edge weights computed on demand rather than stored. For LC 1584, `cost` is the Manhattan distance between `points[u]` and `points[v]`.

Kruskal's algorithm

Sort all edges by weight. Take each edge in order, **unless it would close a cycle**. Each accepted edge is the lightest one leaving its endpoint's component, so the cut property applies. To check "would this close a cycle?" — are the endpoints already in the same component? — use a **union-find** structure (see Trie & Union Find), which answers it in nearly constant time.

```
def kruskal_mst_weight(n: int, edges: list[tuple[int, int, int]]) -> int:
    """edges = [(u, v, w), ...] for an undirected connected graph."""
    parent = list(range(n))

    def find(x: int) -> int:
        while parent[x] != x:
            parent[x] = parent[parent[x]]            # path halving
            x = parent[x]
        return x

    total = used = 0
    for u, v, w in sorted(edges, key=lambda edge: edge[2]):
        ru, rv = find(u), find(v)
        if ru != rv:
            parent[ru] = rv
            total += w
            used += 1
            if used == n - 1:
                break
    return total
```

$O(E \log E)$, dominated by the sort. Kruskal is usually the simpler choice for sparse graphs given as an edge list.

Variations fall out of the same algorithms: a **maximum** spanning tree is an MST on negated weights; the **minimax path** (minimize the heaviest edge on a path) follows the MST; and stopping Kruskal early at k components gives **single-linkage clustering**.

Shortest Paths

BFS finds shortest paths only when every edge costs the same. With weights, the shortest path might use many cheap edges rather than a few expensive ones.

Dijkstra's algorithm

Dijkstra's algorithm is Prim's algorithm with a different notion of "cheapest." Prim adds the vertex attached by the **lightest edge**; Dijkstra adds the vertex with the smallest **total distance from the source**. It maintains tentative distances and repeatedly finalizes the closest unfinalized vertex, relaxing its outgoing edges.

```
def dijkstra(graph: list[list[tuple[int, int]]], source: int) -> list[float]:
    dist = [float("inf")] * len(graph)
    dist[source] = 0
    heap = [(0, source)]
    while heap:
        d, u = heapq.heappop(heap)
        if d > dist[u]:
```

```
            continue                        # stale entry: u already finalized with a shorter d
        for v, w in graph[u]:
            if d + w < dist[v]:             # relax edge (u, v)
                dist[v] = d + w
                heapq.heappush(heap, (dist[v], v))
    return dist
```

Time: $O((V+E)\log E)$, which is $O((V+E)\log V)$ since $\log E < 2\log V$. Python's `heapq` can't decrease a key, so this uses **lazy deletion**: push a new entry on every improvement and skip entries that are out of date when popped. The heap can therefore hold up to $O(E)$ entries rather than V.

Why it's correct: when u is popped with distance d, every other path to u must leave the set of finalized vertices through some vertex whose tentative distance is already $\geq d$ (otherwise *it* would have been popped first), and continuing from there can only add more. That argument needs one assumption: **no negative edges**. A negative edge could make a longer-looking detour cheaper after u was already finalized. (If the bank paid you to walk through its lobby, the shortest way to your neighbor's house might involve walking through it forever.)

Dijkstra adapts to many "shortest" notions, as long as extending a path never makes it better:

- **Path With Minimum Effort** (LC 1631): the cost of a path is its *maximum* step, so relax with `max(d, w)` instead of `d + w`.
- **Path with Maximum Probability** (LC 1514): maximize a product of probabilities — use a max-heap, or Dijkstra on $-\log p$.
- **Swim in Rising Water** (LC 778): minimize the maximum elevation along a path.

Stop and Think: Costs on vertices instead of edges

Problem: In a directed graph, each **vertex** has a cost, and a path's cost is the sum of the costs of the vertices on it. Find the cheapest path from s to t.
Solution: You could modify Dijkstra's code to add vertex costs. But it's cleaner to leave the algorithm alone and **change the graph**: give each directed edge (u, v) the weight cost(v). Every path's edge weight now equals the sum of its vertex costs (plus cost(s), a constant). Plain Dijkstra does the job. This is "design graphs, not algorithms" in miniature.

Negative edges and limited hops: Bellman–Ford

Bellman–Ford relaxes *every* edge, $V-1$ times. After round k, every shortest path using at most k edges has been found; since shortest paths (without negative cycles) use at most $V-1$ edges, that's enough. If a V-th round still improves something, there's a **negative cycle**. $O(VE)$.

The round structure is useful in its own right. **Cheapest Flights Within K Stops** (LC 787) wants the cheapest path with at most $k+1$ edges — exactly $k+1$ rounds of Bellman–Ford, relaxing from a snapshot of the previous round so that each round extends paths by only one edge:

```
def find_cheapest_price(n: int, flights: list[list[int]], src: int, dst: int, k: int) -> int:
    dist = [float("inf")] * n
    dist[src] = 0
    for _ in range(k + 1):
        prev = dist[:]                      # distances using one fewer edge
        for u, v, w in flights:
            if prev[u] + w < dist[v]:
                dist[v] = prev[u] + w
    return -1 if dist[dst] == float("inf") else int(dist[dst])
```

All pairs: Floyd–Warshall

To get the distance between **every** pair of vertices, number the vertices and let $D_k[i][j]$ be the shortest path from i to j whose intermediate vertices all come from $\{0, \dots, k\}$. Allowing vertex k as a new intermediate either helps or it doesn't:

$$D_k[i][j] = \min\left(D_{k-1}[i][j],\ D_{k-1}[i][k] + D_{k-1}[k][j]\right)$$

This is dynamic programming (see Dynamic Programming). The full table would be 3-D, one $V \times V$ layer per k. But a single 2-D table updated in place is enough. Round k reads only row k and column k, and those don't change during round k: $D[i][k] = \min(D[i][k],\ D[i][k] + D[k][k])$, and $D[k][k] = 0$. So the code is three nested loops over one table:

```python
def floyd_warshall(n: int, edges: list[tuple[int, int, int]]) -> list[list[float]]:
    INF = float("inf")
    dist = [[0 if i == j else INF for j in range(n)] for i in range(n)]
    for u, v, w in edges:
        dist[u][v] = min(dist[u][v], w)      # parallel edges: keep the lightest
    for k in range(n):                       # k MUST be the outermost loop
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
```

$O(V^3)$. On sparse graphs, running Dijkstra from every vertex is asymptotically faster ($O(VE \log V)$), but on dense graphs that becomes $O(V^3 \log V)$, and Floyd–Warshall's tight loops win for V up to a few hundred. It handles negative edges, and detects negative cycles: one exists iff some $D[i][i] < 0$ at the end. It also computes **transitive closure** — which vertices can reach which — as a byproduct: i reaches j iff $D[i][j] < \infty$.

Which shortest-path algorithm?

Situation	Algorithm	Time
Unweighted (or all weights equal)	BFS	$O(V + E)$
Weights 0 or 1	0-1 BFS (deque: push 0-edges front, 1-edges back)	$O(V + E)$
DAG, any weights	Relax edges in topological order	$O(V + E)$
Non-negative weights	Dijkstra	$O(E \log V)$
Negative edges, or at most k edges	Bellman–Ford	$O(VE)$ or $O(kE)$
All pairs, $V \lesssim 400$	Floyd–Warshall	$O(V^3)$

Network Flow and Matching (Briefly)

The problem. You're given a directed graph in which each edge (u, v) is a pipe with a **capacity** $c(u, v)$, plus a **source** s and a **sink** t. A **flow** assigns each edge an amount $f(u, v)$ subject to two rules:

- **Capacity:** $0 \le f(u, v) \le c(u, v)$. No pipe carries more than it can hold.

- **Conservation:** at every vertex except s and t, flow in equals flow out. Material is neither created nor stored along the way.

The **value** of the flow is the net amount leaving s, which equals the amount arriving at t. The **maximum flow** problem asks for the largest possible value: how much can be shipped from s to t per unit of time?

Why greedy fails. The natural idea is to find any path from s to t with spare capacity, push as much as it allows, and repeat. Take four unit-capacity edges $s \to a$, $s \to b$, $a \to t$, $b \to t$, plus a unit edge $a \to b$. If the first path found is $s \to a \to b \to t$, it uses up $s \to a$ and $b \to t$, and no other path remains: flow 1. The answer is 2, using $s \to a \to t$ and $s \to b \to t$. The first path routed its flow badly, and greedy can't take that back.

The fix: allow undoing. Search for paths in the **residual graph**. It has a forward edge wherever capacity is left, and a **reverse** edge $v \to u$ wherever (u, v) carries flow. Following a reverse edge cancels flow already sent. In the example, the residual graph contains $s \to b \to a \to t$. Pushing one unit along it cancels the flow on $a \to b$ and reroutes both units correctly. Such a path is an **augmenting path**, and repeating "find one, push along it" until none is left is the **Ford–Fulkerson** method. Finding each path with BFS (the shortest one) is **Edmonds–Karp**, which runs in $O(VE^2)$.

```python
def max_flow(n: int, edges: list[tuple[int, int, int]], source: int, sink: int) -> int:
    """Edmonds-Karp. edges = [(u, v, capacity), ...], directed."""
    residual = [defaultdict(int) for _ in range(n)]  # residual[u][v] = capacity left on u → v
    for u, v, capacity in edges:
        residual[u][v] += capacity
        residual[v][u] += 0                          # make sure the reverse edge exists
    flow = 0
    while True:
        parent = [-1] * n                            # BFS for a shortest augmenting path
        parent[source] = source
        queue = deque([source])
        while queue and parent[sink] == -1:
            current = queue.popleft()
            for neighbor, left in residual[current].items():
                if left > 0 and parent[neighbor] == -1:
                    parent[neighbor] = current
                    queue.append(neighbor)
        if parent[sink] == -1:
            return flow                              # no augmenting path: flow is maximum
        bottleneck, v = float("inf"), sink           # the smallest capacity left on the path
        while v != source:
            bottleneck = min(bottleneck, residual[parent[v]][v])
            v = parent[v]
        v = sink                                     # push it: forward edges shrink, reverse
↪   edges grow
        while v != source:
            residual[parent[v]][v] -= bottleneck
            residual[v][parent[v]] += bottleneck
            v = parent[v]
        flow += bottleneck
```

The reverse entry `residual[v][u]` starts at 0 and grows by however much flow crosses $u \to v$. That's what lets a later BFS find a path through it and cancel that flow.

Why the result is optimal: max-flow = min-cut. A **cut** splits the vertices into a side containing s and a side containing t. Its capacity is the total capacity of the edges crossing from the s side to the t side. Every unit of flow has to cross every cut, so **any flow ≤ any cut**. When no augmenting path is left, the vertices still reachable from s in the residual graph form a cut whose edges are all saturated, so the flow equals that

cut's capacity. Both are therefore optimal. The same computation answers "what is the cheapest set of edges whose removal disconnects s from t?"

Where it shows up: bipartite matching. To pair workers with jobs they're qualified for, as many pairs as possible: add a source with an edge to every worker, an edge from each worker to each job they can do, and an edge from every job to a sink, all with capacity 1. Each unit of flow is one worker–job pair, and capacity 1 stops anyone from being used twice. So the maximum flow is the maximum matching, and augmenting paths here become the "try to reassign someone" step of matching algorithms. Flow problems are uncommon in interviews, but they're worth recognizing in "assign X to Y" and "how many edges or vertices must be cut" problems, because spotting the reduction is usually the whole solution.

Design Graphs, Not Algorithms

The hardest part of most graph problems is noticing that they are graph problems, and choosing the vertices and edges so that a standard algorithm answers the question. Some examples:

Problem	Vertices	Edges	Algorithm
Word Ladder (LC 127)	Words	Differ by one letter	BFS
Open the Lock (LC 752)	4-digit combinations	One wheel turn	BFS
Bus Routes (LC 815)	**Routes** (not stops!)	Routes sharing a stop	BFS from routes containing the source
Evaluate Division (LC 399)	Variables	$a \xrightarrow{a/b} b$, $b \xrightarrow{b/a} a$	DFS multiplying weights
Cheapest Flights Within K Stops (LC 787)	(city, stops used)	Flights	Dijkstra / Bellman–Ford
Shortest Path to Get All Keys (LC 864)	(cell, keys held as a bitmask)	Grid moves	BFS
Reconstruct Itinerary (LC 332)	Airports	Tickets	Eulerian path (Hierholzer)
Course Schedule (LC 207)	Courses	Prerequisites	Topological sort

Two recurring tricks stand out. First, **the obvious vertices are not always the right ones**: in Bus Routes, making each route a vertex turns "fewest buses" into plain BFS. Second, **when a constraint depends on history, put the history into the state**: "at most k stops" or "holding these keys" become part of the vertex, and the enlarged graph is searched with the ordinary algorithm.

Eulerian paths: Reconstruct Itinerary (LC 332)

An **Eulerian path** uses every *edge* exactly once. In a directed graph whose edges are all connected, one exists iff every vertex has in-degree equal to out-degree, except possibly a start with one extra outgoing edge and an end with one extra incoming edge. LC 332 guarantees that a path exists and asks for the lexicographically smallest one.

Walking greedily (always the smallest unused ticket) can get stuck early. From `JFK` with tickets to `KUL` and `NRT`, and `NRT` → `JFK`, greedy flies to `KUL` and dead-ends with tickets left. **Hierholzer's algorithm** fixes this by emitting airports in **post-order**. An airport is added to the route only once it has no unused tickets left, so it must come after everything still to be visited. Dead-end branches are therefore emitted first and end up at the end of the reversed route:

```python
def find_itinerary(tickets: list[list[str]]) -> list[str]:
    graph: dict[str, list[str]] = defaultdict(list)
    for src, dst in sorted(tickets, reverse=True):
        graph[src].append(dst)                  # reverse-sorted, so pop() gives the smallest
    route, stack = [], ["JFK"]
    while stack:
        if graph[stack[-1]]:
            stack.append(graph[stack[-1]].pop())  # use the smallest unused ticket
        else:
            route.append(stack.pop())           # stuck: this airport goes last among what's left
    return route[::-1]
```

Each ticket is pushed and popped once: $O(E \log E)$ for the sort, $O(E)$ for the walk.

Common Mistakes

1. **Marking visited on dequeue in BFS.** Vertices get enqueued repeatedly; mark them when first discovered.
2. **Using a plain visited set for directed cycle detection.** An edge to a finished vertex is not a cycle. Track "on the current DFS path" separately (gray vs. black).
3. **Using BFS on weighted graphs.** BFS minimizes the number of edges, not their total weight.
4. **Dijkstra with negative edges.** It silently returns wrong answers. Use Bellman–Ford, or topological order if the graph is a DAG.
5. **Forgetting disconnected graphs.** A single traversal only covers one component. Loop over all vertices when you need the whole graph (components, bipartiteness, cycle detection, topological sort).
6. **Building huge implicit graphs explicitly.** Generate neighbors on demand. For grids, check bounds instead of storing edges.
7. **Deep recursion.** Recursive DFS on large graphs hits Python's recursion limit. Use an explicit stack when depth can reach the thousands.
8. **Wrong loop order in Floyd–Warshall.** The intermediate vertex k must be the **outer** loop.

Practice Problems

Problem	Technique
Number of Islands (LC 200)	Flood fill / components
Clone Graph (LC 133)	DFS/BFS with old → new map
Max Area of Island (LC 695)	Flood fill with size
Pacific Atlantic Water Flow (LC 417)	Reverse flood fill from both oceans
Surrounded Regions (LC 130)	Flood fill from the border
Rotting Oranges (LC 994)	Multi-source BFS
Shortest Path in Binary Matrix (LC 1091)	BFS on a grid

Problem	Technique
Is Graph Bipartite? (LC 785)	BFS two-coloring
Open the Lock (LC 752)	BFS on an implicit state graph
Word Ladder (LC 127)	BFS, bidirectional BFS
Course Schedule I & II (LC 207, 210)	Topological sort
Find Eventual Safe States (LC 802)	Three-color DFS cycle detection
Alien Dictionary (LC 269)	Build the graph, then topological sort
Redundant Connection (LC 684)	Union-find cycle detection
Critical Connections in a Network (LC 1192)	Bridges via low-link
Min Cost to Connect All Points (LC 1584)	Prim's MST
Network Delay Time (LC 743)	Dijkstra
Path With Minimum Effort (LC 1631)	Dijkstra with max instead of sum
Cheapest Flights Within K Stops (LC 787)	Bellman–Ford with $k + 1$ rounds
Find the City With the Smallest Number of Neighbors (LC 1334)	Floyd–Warshall
Bus Routes (LC 815)	BFS over routes
Reconstruct Itinerary (LC 332)	Eulerian path

Trie & Union Find

This chapter covers two specialized data structures. Neither is as general as a hash table or a balanced tree, but each does one job so well that it turns otherwise awkward problems into short, fast code:

- A **trie** stores a set of strings so that everything sharing a prefix shares storage and search effort. It answers prefix questions — "which words start with `pre`?" — in time proportional to the prefix, independent of how many words are stored.
- A **union-find** (disjoint set) structure maintains a partition of elements into groups under merging, answering "are these two in the same group?" in effectively constant time.

Both illustrate a recurring lesson of data structure design: when the obvious structures each make one of your operations fast and another slow, look for a structure that balances them.

Part 1: Tries

Why Another Dictionary?

Suppose you store a set of n words and need to answer queries about a string q of length k.

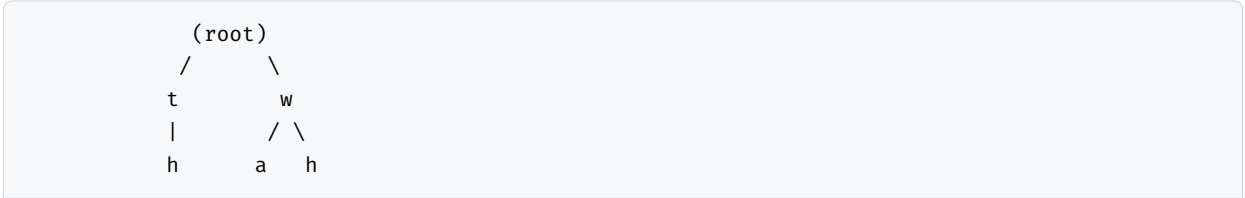
Structure	Is q a word?	Do any words start with q?	List words starting with q
Hash set	$O(k)$ expected	$O(n \cdot k)$: check every word	$O(n \cdot k)$
Sorted list + binary search	$O(k \log n)$	$O(k \log n)$	$O(k \log n + \text{output})$
Trie	$O(k)$	$O(k)$	$O(k + \text{output size})$

A hash set is great at exact lookups but useless for prefixes: hashing scatters `car`, `card`, and `care` to unrelated buckets. A sorted list keeps words with a common prefix together, which is why it handles prefix queries — but every comparison costs up to k character checks.

There's a subtler advantage too. Often queries arrive **incrementally**: first `c`, then `ca`, then `car`, each extending the last by one character (a user typing, a DFS extending a path on a board). A hash set must hash each new string from scratch, costing $O(k)$ per query. A trie just takes **one more step** from where the previous query ended: $O(1)$ per character.

Structure

A trie is a tree in which each edge is labeled by a character, and the path from the root to a node spells out a string. Words with a common prefix share the path for that prefix and branch at their first differing character. Nodes that end a word are marked.



```
          |      |   |
          e*     s*  e
         / \         |
        i   r        n*
        |   |
        r*  e*
                  words: the, their, there, was, when   (* = end of word)
```

Searching for q walks down from the root following q's characters. If an edge is missing, q isn't in the set — and neither is any word with q as a prefix. Otherwise, the search ends after exactly $|q|$ steps, no matter how many words are stored.

Implementation (LC 208)

```python
class TrieNode:
    __slots__ = ("children", "word")

    def __init__(self) -> None:
        self.children: dict[str, TrieNode] = {}
        self.word: str | None = None             # the word ending here, if any


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

    def insert(self, word: str) -> None:
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.word = word

    def _walk(self, s: str) -> TrieNode | None:
        node = self.root
        for ch in s:
            node = node.children.get(ch)
            if node is None:
                return None
        return node

    def search(self, word: str) -> bool:
        node = self._walk(word)
        return node is not None and node.word is not None

    def starts_with(self, prefix: str) -> bool:
        return self._walk(prefix) is not None
```

Each operation is $O(k)$ for a string of length k. Space is O(total characters inserted) in the worst case (no shared prefixes), and much less when prefixes overlap. A `dict` of children works for any alphabet; a fixed array of 26 slots is faster in compiled languages but wastes space on sparse nodes.

Marking word ends with the word itself rather than a boolean costs one reference per word, and a search that reaches the node gets the word back without rebuilding it — Word Search II below relies on this.

Searching with Wildcards (LC 211)

When the query contains `.` (matches any single character), the walk branches: at a `.`, try every child. The trie still prunes aggressively — a branch dies the moment its prefix isn't present.

```
class WordDictionary(Trie):                    # words are added with Trie.insert
    def search(self, word: str) -> bool:
        def match(node: TrieNode, i: int) -> bool:
            if i == len(word):
                return node.word is not None
            ch = word[i]
            if ch == ".":
                return any(match(child, i + 1) for child in node.children.values())
            child = node.children.get(ch)
            return child is not None and match(child, i + 1)

        return match(self.root, 0)
```

Word Search II (LC 212): Walking a Trie and a Grid Together

Find all dictionary words that can be traced on a letter grid by moving between adjacent cells. The naive approach runs a separate board search for each word — hopeless for thousands of words.

The key insight is the incremental-query advantage from above. A DFS on the board extends its current path by **one letter at a time**, so walk the trie **in lockstep** with it. Each board step is one trie step, and the moment the path's prefix is absent from the trie, that entire branch of the board search is abandoned. One DFS from each cell finds *all* words at once.

```
def grid_neighbors(r: int, c: int, rows: int, cols: int):
    for nr, nc in ((r + 1, c), (r - 1, c), (r, c + 1), (r, c - 1)):
        if 0 <= nr < rows and 0 <= nc < cols:
            yield nr, nc


def find_words(board: list[list[str]], words: list[str]) -> list[str]:
    trie = Trie()
    for word in words:
        trie.insert(word)
    rows, cols = len(board), len(board[0])
    found: list[str] = []

    def dfs(r: int, c: int, parent: TrieNode) -> None:
        letter = board[r][c]
        node = parent.children[letter]
        if node.word is not None:
            found.append(node.word)
            node.word = None                    # report each word only once
        board[r][c] = "#"                       # mark as used on this path
        for nr, nc in grid_neighbors(r, c, rows, cols):
            if board[nr][nc] in node.children:
                dfs(nr, nc, node)
        board[r][c] = letter                    # backtrack
        if not node.children:
            del parent.children[letter]         # prune: nothing left below this prefix

    for r in range(rows):
```

```
        for c in range(cols):
            if board[r][c] in trie.root.children:
                dfs(r, c, trie.root)
    return found
```

Two refinements make a big difference in practice: removing each word from the trie once it's found (so it isn't reported twice), and **pruning** trie nodes that have become empty, so later searches don't walk into exhausted branches.

> **Take-Home Lesson**
>
> A trie lets a search that grows one character at a time take one step per character, and it lets you abandon a search as soon as the current prefix leads nowhere. Whenever a backtracking search builds strings and checks them against a dictionary, drive the search with a trie.

More Trie Applications

- **Autocomplete / Search Suggestions** (LC 1268): walk to the prefix's node, then DFS (in alphabetical child order) to collect completions. For heavy traffic, cache the top suggestions at each node.
- **Replace Words** (LC 648): for each word, walk the trie and stop at the first node marked as a root word.
- **Longest Word in Dictionary** (LC 720): a word counts only if every prefix is also a word — a DFS that only descends through word-ending nodes.
- **Binary tries for XOR** (LC 421): store numbers as fixed-width bit strings, most significant bit first. To maximize `x ^ y`, walk the trie from the top, at each level preferring the child with the **opposite** bit of `x` — a greedy choice that's correct because a higher bit outweighs all lower bits combined.

```
def find_maximum_xor(nums: list[int]) -> int:
    bits = max(nums).bit_length()
    root: dict[int, dict] = {}                  # binary trie: bit -> child
    for x in nums:                              # insert every number, high bit first
        node = root
        for i in reversed(range(bits)):
            node = node.setdefault((x >> i) & 1, {})
    best = 0
    for x in nums:                             # greedily take the opposite bit when possible
        node, xor = root, 0
        for i in reversed(range(bits)):
            bit = (x >> i) & 1
            opposite = bit ^ 1
            if opposite in node:
                xor |= 1 << i
                node = node[opposite]
            else:
                node = node[bit]
        best = max(best, xor)
    return best
```

Suffix Tries: Every Substring Is a Prefix

A **suffix trie** stores all suffixes of a string S. Its magic is a simple observation: **every substring of S is a prefix of some suffix**. So "is q a substring of S?" becomes a walk of $|q|$ steps from the root.

The catch: S has n suffixes of average length $n/2$, so the naive suffix trie has $\Theta(n^2)$ nodes. **Suffix trees** compress each unbranching path into a single edge labeled by a pair of indices into S, which brings the

size down to $O(n)$, and clever algorithms build them in linear time. **Suffix arrays** — the sorted list of suffix start positions — offer most of the same power in a simpler, more memory-friendly package. You'll rarely implement either in an interview, but knowing they exist is valuable: they turn many quadratic string problems (longest repeated substring, longest common substring of two strings) into linear or near-linear ones.

Part 2: Union-Find

Dynamic Connectivity

Many problems maintain a **partition** of elements into disjoint groups — connected components of a growing graph, accounts belonging to the same person, variables forced equal — and repeatedly ask two things:

- **Find:** which group is x in? (Equivalently: are x and y in the same group?)
- **Union:** merge the groups containing x and y.

The two obvious representations each make one operation fast and the other slow:

Representation	Same group?	Merge two groups
Label array: `group[x]`	$O(1)$	$O(n)$: relabel one group's elements
Graph of the merges, traversed on demand	$O(n)$: run BFS/DFS	$O(1)$: add an edge
Union-find	$O(\alpha(n))$ amortized	$O(\alpha(n))$ amortized

Backward Trees

Union-find represents each group as a tree in which every node points to its **parent**, and the root names the group. There are no child pointers at all — just one `parent` array.

- **Find(x):** follow parent pointers from x until reaching a root (a node that is its own parent).
- **Union(x, y):** find both roots; if they differ, make one root point to the other.

Both cost $O(\text{height})$, so everything depends on keeping the trees short. Two simple rules do that.

Union by size. Always hang the **smaller** tree under the root of the larger one. Every node in the smaller tree gets one step deeper; nobody in the larger tree does. Why does this keep trees short? A node gets deeper only when its tree is merged into one at least as large — so its tree's size at least **doubles** every time the node's depth increases. A size can double at most $\log_2 n$ times, so every tree has height $O(\log n)$.

Path compression. During a find, after locating the root, repoint every node on the path directly at it. Future finds on those nodes take one step.

With both rules, a sequence of m operations costs $O(m \cdot \alpha(n))$, where α is the **inverse Ackermann function** — which grows so slowly that $\alpha(n) \leq 4$ for any n that could fit in this universe. For all practical purposes, union-find operations are constant time.

```
class UnionFind:
    def __init__(self, n: int) -> None:
        self.parent = list(range(n))
        self.size = [1] * n
        self.components = n
```

```
    def find(self, x: int) -> int:
        root = x
        while self.parent[root] != root:
            root = self.parent[root]
        while self.parent[x] != root:           # path compression
            next_x = self.parent[x]
            self.parent[x] = root
            x = next_x
        return root

    def union(self, x: int, y: int) -> bool:
        """Merge the groups of x and y. Return False if they were already together."""
        rx, ry = self.find(x), self.find(y)
        if rx == ry:
            return False
        if self.size[rx] < self.size[ry]:       # union by size
            rx, ry = ry, rx
        self.parent[ry] = rx
        self.size[rx] += self.size[ry]
        self.components -= 1
        return True

    def connected(self, x: int, y: int) -> bool:
        return self.find(x) == self.find(y)
```

`union` returning whether a merge happened is a small design choice that pays off constantly: "this edge connected two already-connected vertices" is exactly what cycle detection and Kruskal's algorithm need to know.

Applications

Components and cycles

- **Number of Connected Components** (LC 323) and **Number of Provinces** (LC 547): union every edge; the answer is `components`.
- **Redundant Connection** (LC 684): a tree plus one extra edge. Process edges in order; the first edge whose `union` returns `False` closes a cycle.
- **Graph Valid Tree** (LC 261): exactly $n - 1$ edges, and no `union` fails.

```
def find_redundant_connection(edges: list[list[int]]) -> list[int]:
    uf = UnionFind(len(edges) + 1)
    for u, v in edges:
        if not uf.union(u, v):
            return [u, v]
    return []
```

Grouping by shared keys: Accounts Merge (LC 721)

Accounts sharing any email belong to the same person. Rather than comparing accounts pairwise, union each account with the **first account that claimed each email**. Then gather emails by root.

```
from collections import defaultdict


def accounts_merge(accounts: list[list[str]]) -> list[list[str]]:
    uf = UnionFind(len(accounts))
```

```
    owner: dict[str, int] = {}                  # email -> first account that listed it
    for i, account in enumerate(accounts):
        for email in account[1:]:               # account[0] is the name
            if email in owner:
                uf.union(i, owner[email])
            else:
                owner[email] = i
    groups: dict[int, list[str]] = defaultdict(list)
    for email, i in owner.items():
        groups[uf.find(i)].append(email)
    return [[accounts[root][0]] + sorted(emails) for root, emails in groups.items()]
```

Equivalence constraints: Satisfiability of Equality Equations (LC 990)

Given constraints like `a==b` and `b!=c`, decide whether they can all hold. Equality is an equivalence relation, so union all `==` pairs first; then every `!=` pair must lie in different groups.

```
def letter_index(ch: str) -> int:
    return ord(ch) - ord("a")


def equations_possible(equations: list[str]) -> bool:
    uf = UnionFind(26)
    for x, op, _, y in equations:               # "a==b" unpacks to "a", "=", "=", "b"
        if op == "=":
            uf.union(letter_index(x), letter_index(y))
    for x, op, _, y in equations:
        if op == "!" and uf.connected(letter_index(x), letter_index(y)):
            return False
    return True
```

Kruskal's minimum spanning tree

Sort edges by weight and add each one whose endpoints are in different components — the union-find test is exactly "would this edge close a cycle?" See Graphs.

Online connectivity: Number of Islands II (LC 305)

Land cells appear one at a time on a grid; after each addition, report the number of islands. Re-running a flood fill after each addition costs $O(mn)$ per step. With union-find, each new cell starts as its own island and unions with its land neighbors: $O(\alpha(mn))$ per step.

```
def num_islands2(m: int, n: int, positions: list[list[int]]) -> list[int]:
    def cell_id(r: int, c: int) -> int:
        return r * n + c

    uf = UnionFind(m * n)
    land = [[False] * n for _ in range(m)]
    islands = 0
    result: list[int] = []
    for r, c in positions:
        if not land[r][c]:                      # the same cell may be added twice
            land[r][c] = True
            islands += 1
            for nr, nc in grid_neighbors(r, c, m, n):
                if land[nr][nc] and uf.union(cell_id(r, c), cell_id(nr, nc)):
                    islands -= 1                # two islands just merged into one
```

```
            result.append(islands)
        return result
```

`uf.components` would count water cells too, so the island count is tracked separately: +1 for each new cell, and −1 for each `union` that actually merges two islands.

This is where union-find truly beats BFS/DFS: when **edges arrive over time** and connectivity questions are interleaved with them.

Union-Find vs. Traversal

Situation	Better tool
Static graph, count or label components once	Either; DFS/BFS is just as fast and needs no extra structure
Edges arrive incrementally; queries interleaved	**Union-find**
Need actual paths, distances, or traversal order	**BFS / DFS**
Need to **delete** edges or split groups	Neither directly — union-find can't split. If all operations are known in advance, process them **in reverse**, turning deletions into unions
Directed reachability	**DFS / BFS** (union-find is for symmetric relations)

Common Mistakes

Tries

1. **Confusing "prefix exists" with "word exists."** `starts_with("app")` can be true while `search("app")` is false. The end-of-word flag is what distinguishes them.
2. **Rebuilding strings during DFS.** Concatenating `path + ch` at each step costs $O(k)$ per step. Store the full word at its terminal node (as `TrieNode.word` does), or keep a list and join only when reporting.
3. **Reporting duplicates in Word Search II.** A word reachable by two board paths is found twice unless you remove it from the trie (or use a set).
4. **Forgetting memory cost.** A trie with a node object per character can use far more memory than a hash set of the same words. Use it when you need prefix operations.

Union-Find

1. **Comparing elements instead of roots.** `parent[x] == parent[y]` is not the same-group test; `find(x) == find(y)` is.
2. **Skipping union by size (or rank) and path compression.** Without them, a sequence of unions can build a linked list, and finds degrade to $O(n)$.
3. **Unioning non-roots.** `parent[x] = y` links x's node, not x's whole tree. Always link `find(x)` to `find(y)`.
4. **Passing raw labels as elements.** `UnionFind` works on integers `0..n-1`. Map letters, strings or grid cells to indices first (`letter_index`, `cell_id`, or a `dict`), or key the parent map itself by the element.

5. **Expecting deletions to work.** Union-find merges only. For splits, reverse time or use a different structure.

Practice Problems

Problem	Structure / idea
Implement Trie (LC 208)	Basic trie
Design Add and Search Words (LC 211)	Trie + DFS on wildcards
Word Search II (LC 212)	Trie driving a grid DFS, with pruning
Replace Words (LC 648)	Shortest root prefix
Search Suggestions System (LC 1268)	Trie + DFS, or sort + binary search
Maximum XOR of Two Numbers (LC 421)	Binary trie, greedy by bit
Number of Connected Components (LC 323)	Union-find
Number of Provinces (LC 547)	Union-find on an adjacency matrix
Redundant Connection (LC 684)	First failed union
Graph Valid Tree (LC 261)	$n - 1$ edges and no cycle
Accounts Merge (LC 721)	Union by shared email
Satisfiability of Equality Equations (LC 990)	Union `==`, check `!=`
Longest Consecutive Sequence (LC 128)	Union neighbors (or a hash set)
Number of Islands II (LC 305)	Online union-find on a grid
Min Cost to Connect All Points (LC 1584)	Kruskal with union-find

Combinatorial Search

Some problems have no clever shortcut: to find all valid arrangements, or the best one, you must in principle consider every candidate. **Combinatorial search** is the art of doing that efficiently. Its workhorse is **backtracking**, which enumerates every configuration of a search space exactly once while abandoning partial configurations the moment they can't possibly succeed.

Exhaustive search has a bad reputation, but surprisingly large problems yield to it, and it has one great virtue: it is *obviously correct*. If you've tried every possibility, you haven't missed the answer. The craft lies in four places:

- **Generating** each candidate exactly once, with no repeats and no omissions.
- **Pruning** the search so that you only look at candidates that matter.
- **Ordering** the search so that the best candidates are examined first.
- **Splitting** the search space when it is too big to enumerate as a whole.

How Big Is Exhaustive?

It's important to have a feel for how big — and how small — the search spaces are (the counting behind these sizes is in Combinatorics). Python can explore on the order of a million simple search states per second:

Search space	Size	Feasible up to about
All subsets of n items	2^n	$n \approx 20$ (10^6)
All permutations of n items	$n!$	$n \approx 10$ (3.6×10^6)
All k-subsets	$\binom{n}{k}$	depends; $\binom{30}{5} \approx 1.4 \times 10^5$
All strings of length n over Σ	$\lvert\Sigma\rvert^n$	$4^{10} \approx 10^6$

These limits are for *unpruned* search. Good pruning pushes them much further — sometimes by factors of thousands — because it cuts off whole subtrees at once. When a LeetCode problem has $n \leq 15$ or so, it is usually telling you that exponential search is expected.

The Backtracking Framework

Model a solution as a vector $a = (a_1, a_2, \ldots, a_n)$, built one position at a time. For subsets, a_i might be "is item i included?"; for permutations, a_i is the item in position i; for a path in a graph, a_i is the i-th vertex.

At each step, extend the partial solution $(a_1, \ldots, a_k)$ by one element. If it's complete, process it; otherwise, consider every legal candidate for position $k+1$ in turn. The partial solutions form a **tree**, and backtracking is simply a **depth-first search** of that implicit tree. DFS (rather than BFS) is the right choice because it only needs to remember the current root-to-node path — space proportional to the *depth* of the tree, not its enormous width.

Every backtracking algorithm is the same skeleton with five problem-specific pieces:

```python
def backtrack(partial: list, state) -> None:
    if is_solution(partial, state):              # 1. is the partial solution complete?
        process_solution(partial, state)         # 2. record / count / print it
        return
    for candidate in candidates(partial, state): # 3. legal choices for the next position
        make_move(partial, candidate, state)     # 4. apply the choice
        backtrack(partial, state)
        unmake_move(partial, candidate, state)   # 5. undo it exactly
```

The `make_move` / `unmake_move` pair is what makes backtracking efficient. Rather than copying the whole state at each node, you mutate one shared state and **undo** each change on the way back up. The undo must restore the state *exactly*; most backtracking bugs are an incomplete undo.

> **Take-Home Lesson**
>
> Backtracking guarantees correctness by enumerating all possibilities, and efficiency by never visiting a state twice. For each new problem, you only need to decide what a partial solution looks like, what the candidates for the next position are, and how to apply and undo a move.

Generating the Basic Objects

Almost every backtracking problem is a variation on generating subsets, permutations, or combinations. Learn these three cold.

Subsets (LC 78)

Every node of the search tree is itself a subset; its children add one element *later* than any already chosen. Starting each loop at `start` is what prevents generating `{2, 1}` after `{1, 2}`.

```python
def subsets(nums: list[int]) -> list[list[int]]:
    result: list[list[int]] = []
    chosen: list[int] = []

    def extend(start: int) -> None:
        result.append(chosen[:])                 # every node is a valid subset; copy, chosen
↪   keeps changing
        for i in range(start, len(nums)):
            chosen.append(nums[i])
            extend(i + 1)
            chosen.pop()

    extend(0)
    return result
```

$O(n \cdot 2^n)$: 2^n subsets, each copied in $O(n)$. (Subsets can also be generated without recursion by binary counting; see Bit Manipulation.) The combination, duplicate-handling, and Combination Sum generators below are all this same tree with an extra rule.

Permutations (LC 46)

Position k can hold any element not already used. A `used` array makes "not already used" an $O(1)$ check:

```python
def permutations(nums: list[int]) -> list[list[int]]:
    result: list[list[int]] = []
    perm: list[int] = []
    used = [False] * len(nums)

    def place() -> None:
        if len(perm) == len(nums):
            result.append(perm[:])
            return
        for i, x in enumerate(nums):
            if not used[i]:
                used[i] = True                  # make move
                perm.append(x)
                place()
                perm.pop()                      # unmake move
                used[i] = False

    place()
    return result
```

$O(n \cdot n!)$. There are $n!$ leaves, and the tree has fewer than $e \cdot n!$ nodes in total, so the internal nodes don't change the bound.

Combinations: k-subsets (LC 77)

Choose k of the numbers $1 \dots n$. It's the subsets tree, recording only nodes at depth k — plus a cheap but valuable **prune**: if there aren't enough numbers left to reach size k, stop.

```python
def combine(n: int, k: int) -> list[list[int]]:
    result: list[list[int]] = []
    chosen: list[int] = []

    def extend(start: int) -> None:
        if len(chosen) == k:
            result.append(chosen[:])
            return
        need = k - len(chosen)
        for i in range(start, n - need + 1):
            chosen.append(i + 1)        # index i holds the number i + 1
            extend(i + 1)
            chosen.pop()

    extend(0)
    return result
```

Paths in a graph (LC 797)

A path is a vector of vertices where each a_{k+1} is a neighbor of a_k not already on the path. In a DAG, "not already on the path" is automatic:

```python
def all_paths_source_target(graph: list[list[int]]) -> list[list[int]]:
    target = len(graph) - 1
    result: list[list[int]] = []
    path = [0]
```

```
    def walk(u: int) -> None:
        if u == target:
            result.append(path[:])
            return
        for v in graph[u]:
            path.append(v)
            walk(v)
            path.pop()

    walk(0)
    return result
```

The number of paths can be exponential in the size of the graph, so the running time is too — output size alone forces it.

Each Configuration Exactly Once: Handling Duplicates

When the input contains duplicates, the generators above produce duplicate outputs: `[1, 2a]` and `[1, 2b]` look identical. Generating everything and deduplicating with a set works but wastes time. It's better to never generate the duplicates at all.

The rule: sort the input so equal values are adjacent. At any single node of the search tree, try each **distinct value** as the next choice only once.

Subsets with duplicates (LC 90) — in the subsets tree, siblings are the choices at the same node; skip a value equal to the previous sibling:

```
def subsets_with_dup(nums: list[int]) -> list[list[int]]:
    nums = sorted(nums)
    result: list[list[int]] = []
    chosen: list[int] = []

    def extend(start: int) -> None:
        result.append(chosen[:])
        for i in range(start, len(nums)):
            if i > start and nums[i] == nums[i - 1]:
                continue                        # same value already tried at this node
            chosen.append(nums[i])
            extend(i + 1)
            chosen.pop()

    extend(0)
    return result
```

Note the condition `i > start`, not `i > 0`. Taking a second copy of a value **deeper** in the tree (as a child of the first copy) is fine — that's how `[2, 2]` gets generated. Only sibling repeats are redundant.

Permutations with duplicates (LC 47) — among equal values, insist that they are used in their original left-to-right order. A copy may be placed only if the previous equal copy has already been placed:

```
def permute_unique(nums: list[int]) -> list[list[int]]:
    nums = sorted(nums)
    result: list[list[int]] = []
```

```
    perm: list[int] = []
    used = [False] * len(nums)

    def place() -> None:
        if len(perm) == len(nums):
            result.append(perm[:])
            return
        for i in range(len(nums)):
            if used[i]:
                continue
            if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
                continue                        # use equal copies left to right only
            used[i] = True
            perm.append(nums[i])
            place()
            perm.pop()
            used[i] = False

    place()
    return result
```

Combination Sum II (LC 40) combines both ideas: sibling-skip for duplicates, and each element used at most once.

Pruning: Cutting the Tree Early

Generating all $n!$ tours of a traveling salesman instance and then evaluating each one is wasteful. If the tour so far is already more expensive than the best complete tour found, **no extension can help** — so stop exploring this branch. Checking constraints on *partial* solutions, rather than waiting for complete ones, is the single most important technique in exhaustive search.

> **Take-Home Lesson**
>
> Clever pruning can make short work of surprisingly hard search problems. Proper pruning has a far greater effect on search time than any other factor — the data structures, the language, or the speed of the machine.

There are three main kinds of pruning.

1. Feasibility pruning: abandon a partial solution as soon as it violates a constraint or can no longer be completed. In **Combination Sum** (LC 39), sort the candidates; once a candidate overshoots the remaining target, every later (larger) candidate will too, so `break` rather than `continue`:

```
def combination_sum(candidates: list[int], target: int) -> list[list[int]]:
    candidates = sorted(candidates)
    result: list[list[int]] = []
    chosen: list[int] = []

    def extend(start: int, remaining: int) -> None:
        if remaining == 0:
            result.append(chosen[:])
            return
        for i in range(start, len(candidates)):
```

```
            c = candidates[i]
            if c > remaining:
                break                           # sorted: everything after is too big as well
            chosen.append(c)
            extend(i, remaining - c)            # i, not i + 1: reuse allowed
            chosen.pop()

    extend(0, target)
    return result
```

Generate Parentheses (LC 22) is pure feasibility pruning: never place more than n opening brackets, and never close more than you've opened. Every leaf reached is valid, so no work is wasted on invalid strings at all.

2. Bounding (branch and bound): in optimization problems, abandon a partial solution when even an optimistic estimate of its best completion can't beat the best solution found so far. See the worked example below.

3. Symmetry: if two branches are guaranteed to lead to equivalent results, explore only one. For a TSP tour, rotations of the same cycle are the same tour, so fix the starting city: a factor of n saved for free.

A worked example: Partition to K Equal Sum Subsets (LC 698)

Can the numbers be split into k groups with equal sums? Each group must sum to `target = total / k`. The naive search assigns each number to one of k buckets — k^n possibilities. Three prunings make it fast:

```
def can_partition_k_subsets(nums: list[int], k: int) -> bool:
    total = sum(nums)
    if total % k:
        return False
    target = total // k
    nums = sorted(nums, reverse=True) # (a) big items first: fail fast
    if nums[0] > target:
        return False
    buckets = [0] * k

    def place(i: int) -> bool:
        if i == len(nums):
            return True              # all placed: no bucket exceeds target and they sum to k *
             ↪  target, so all equal it
        seen: set[int] = set()
        for b in range(k):
            if buckets[b] + nums[i] > target or buckets[b] in seen:
                continue              # (b) infeasible, or (c) symmetric to a tried bucket
            seen.add(buckets[b])
            buckets[b] += nums[i]
            if place(i + 1):
                return True
            buckets[b] -= nums[i]
        return False

    return place(0)
```

- **(a) Order the choices**: placing large numbers first makes buckets overflow early, near the root of the tree, where pruning saves the most.
- **(b) Feasibility**: never overfill a bucket.
- **(c) Symmetry**: two buckets with the same current sum are interchangeable — putting `nums[i]` into

either leads to equivalent searches. Try only one. In particular, all the empty buckets are the same bucket.

Branch and bound: Find Minimum Time to Finish All Jobs (LC 1723)

LC 698 asks a yes/no question. Turn it into an optimization: assign every job to one of k workers so that the **largest** total is as small as possible. Backtracking still tries every assignment, but now it also remembers `best`, the cost of the best complete assignment found so far. Branch and bound needs two things:

- **A lower bound on every completion.** The largest load so far, `current_max`, can only grow as more jobs are assigned. So no completion of the current branch can cost less than it.
- **A good solution to compare against.** As soon as `current_max >= best`, the branch can't produce anything better than what's already known, so abandon it.

```
def minimum_time_required(jobs: list[int], k: int) -> int:
    jobs = sorted(jobs, reverse=True)            # big jobs first: high loads show up early
    loads = [0] * k
    best = sum(jobs)                            # a valid answer to start from: one worker does
↪   it all

    def assign(i: int, current_max: int) -> None:
        nonlocal best
        if current_max >= best:
            return                               # bound: no completion of this branch can win
        if i == len(jobs):
            best = current_max                   # a strictly better complete solution
            return
        tried: set[int] = set()
        for w in range(k):
            if loads[w] in tried:
                continue                        # symmetry: equal loads give equivalent subtrees
            tried.add(loads[w])
            loads[w] += jobs[i]
            assign(i + 1, max(current_max, loads[w]))
            loads[w] -= jobs[i]

    assign(0, 0)
    return best
```

The bound and the search help each other. Every improvement to `best` makes the bound prune more, and sorting the jobs largest first finds good solutions early, while the tree is still small. Two refinements tighten it further: start `best` from a greedy solution instead of the trivial one, and use a stronger lower bound, such as $\max(\text{current_max}, \lceil \text{remaining work}/k \rceil)$. The stronger the bound, the earlier branches die.

Constraint Satisfaction

In constraint satisfaction problems, every position has a domain of values and the constraints forbid certain combinations. Backtracking assigns positions one at a time and checks constraints incrementally — the `make_move` step also updates whatever data structure answers "is this value still allowed?" in $O(1)$.

N-Queens (LC 51)

Place n queens so that none attack each other. One queen per row, so the solution vector gives each row's column. A queen at (r, c) occupies column c, "diagonal" $r - c$, and "anti-diagonal" $r + c$; three sets make each safety check $O(1)$.

The solution vector has a fixed length n, with `-1` marking an empty row. Besides reading naturally (`queens[r] = c`), this supports a variant where some queens are placed in advance: set their rows and constraint sets before the search, and skip those rows during it. The Sudoku solver below works exactly this way with its given cells.

```
def solve_n_queens(n: int) -> list[list[str]]:
    result: list[list[str]] = []
    cols: set[int] = set()
    diag: set[int] = set()
    anti: set[int] = set()
    queens = [-1] * n       # queens[r] = column of the queen in row r, -1 if empty

    def is_free(r: int, c: int) -> bool:
        return c not in cols and r - c not in diag and r + c not in anti

    def set_queen(r: int, c: int) -> None:
        queens[r] = c
        cols.add(c)
        diag.add(r - c)
        anti.add(r + c)

    def unset_queen(r: int, c: int) -> None:
        queens[r] = -1
        cols.remove(c)
        diag.remove(r - c)
        anti.remove(r + c)

    def place(r: int) -> None:
        if r == n:
            result.append(["." * c + "Q" + "." * (n - c - 1) for c in queens])
            return
        for c in range(n):
            if is_free(r, c):
                set_queen(r, c)
                place(r + 1)
                unset_queen(r, c)

    place(0)
    return result
```

Sudoku (LC 37): choose the most constrained cell

A natural Sudoku solver fills empty cells in reading order, trying each digit not already in the cell's row, column, or box. It works — but experiments show that for hard puzzles it can take an extremely long time. Two changes make it orders of magnitude faster:

- **Most constrained first.** Instead of the next cell in reading order, fill the empty cell with the **fewest** remaining candidates. A cell with one candidate is a forced move and costs nothing; a cell with two candidates is a coin flip rather than a 1-in-9 guess. Since the branching factors multiply down the tree, reducing the average branching from 3 to 2 over 20 cells cuts the search from $3^{20} \approx 3.5 \times 10^9$ to $2^{20} \approx 10^6$ — a factor of over 3,000.

- **Look ahead.** If any empty cell has **zero** candidates left, the current partial solution is dead — backtrack now, instead of discovering it many moves later. Most-constrained selection gets this for free: the cell with zero candidates is the one it picks.

```
def solve_sudoku(board: list[list[str]]) -> None:
    rows: list[set[str]] = [set() for _ in range(9)]
    cols: list[set[str]] = [set() for _ in range(9)]
    boxes: list[set[str]] = [set() for _ in range(9)]
    empty: set[tuple[int, int]] = set()

    def box(r: int, c: int) -> int:
        return r // 3 * 3 + c // 3

    def candidates(r: int, c: int) -> set[str]:
        return set("123456789") - rows[r] - cols[c] - boxes[box(r, c)]

    def num_candidates(cell: tuple[int, int]) -> int:
        r, c = cell
        return len(candidates(r, c))

    def make_move(r: int, c: int, d: str) -> None:
        board[r][c] = d
        rows[r].add(d)
        cols[c].add(d)
        boxes[box(r, c)].add(d)

    def unmake_move(r: int, c: int, d: str) -> None:
        board[r][c] = "."
        rows[r].remove(d)
        cols[c].remove(d)
        boxes[box(r, c)].remove(d)

    for r in range(9):
        for c in range(9):
            if board[r][c] == ".":
                empty.add((r, c))
            else:
                make_move(r, c, board[r][c])     # register the given digit

    def solve() -> bool:
        if not empty:
            return True
        r, c = min(empty, key=num_candidates)    # most constrained; 0 candidates = dead end
        empty.remove((r, c))
        for d in candidates(r, c):
            make_move(r, c, d)
            if solve():
                return True
            unmake_move(r, c, d)
        empty.add((r, c))
        return False

    solve()
```

Choosing the most constrained cell looks like it only *reorders* the work, but it changes the shape of the search tree: fixing constrained positions first adds constraints that shrink the branching everywhere below. The same heuristic — **fail first** — applies to any constraint problem: graph coloring, scheduling, crossword filling.

> **Take-Home Lesson**
>
> The order in which you make decisions matters enormously. Make the most constrained decision first, and check for dead ends as early as possible. Both reduce the branching factor, and the branching factor is exponentiated by the depth of the search.

Searching Grids and Strings

Word Search (LC 79)

Trace a word through adjacent cells without reusing a cell. The partial solution is the path so far; the candidates are unvisited neighbors holding the next letter. Marking the cell in place and restoring it afterward is the make/unmake pair:

```python
def grid_neighbors(r: int, c: int, rows: int, cols: int):
    for nr, nc in ((r + 1, c), (r - 1, c), (r, c + 1), (r, c - 1)):
        if 0 <= nr < rows and 0 <= nc < cols:
            yield nr, nc


def exist(board: list[list[str]], word: str) -> bool:
    rows, cols = len(board), len(board[0])

    def trace(r: int, c: int, i: int) -> bool:
        """Can word[i:] be traced from (r, c)? The caller guarantees board[r][c] == word[i]."""
        if i == len(word) - 1:
            return True
        board[r][c] = "#"                       # make move: mark the cell as used
        found = False
        for nr, nc in grid_neighbors(r, c, rows, cols):
            if board[nr][nc] == word[i + 1] and trace(nr, nc, i + 1):
                found = True
                break
        board[r][c] = word[i]                   # unmake move
        return found

    for r in range(rows):
        for c in range(cols):
            if board[r][c] == word[0] and trace(r, c, 0):
                return True
    return False
```

Every call is made only on a cell holding the next letter: `word[0]` for the starting cells, `word[i + 1]` for the neighbors. A used cell holds `"#"`, which never equals a letter of `word`, so the same check also rejects revisits.

A cheap global prune before searching: if the board doesn't contain enough copies of each letter in the word, return `False` immediately. For many words at once, drive the search with a trie (Word Search II; see Trie & Union Find).

Partitioning a string (LC 131)

Split a string into palindromes, returning all ways. The solution vector is the list of pieces; the candidates for the next piece are the palindromic prefixes of what remains.

```python
def partition(s: str) -> list[list[str]]:
    n = len(s)
    # is_pal[i][j]: s[i..j] is a palindrome — precomputed so each check is O(1)
    is_pal = [[False] * n for _ in range(n)]
    for i in range(n - 1, -1, -1):
        for j in range(i, n):
            is_pal[i][j] = s[i] == s[j] and (j - i < 2 or is_pal[i + 1][j - 1])

    result: list[list[str]] = []
    pieces: list[str] = []

    def cut(start: int) -> None:
        if start == n:
            result.append(pieces[:])
            return
        for end in range(start, n):
            if is_pal[start][end]:
                pieces.append(s[start:end + 1])
                cut(end + 1)
                pieces.pop()

    cut(0)
    return result
```

Precomputing the palindrome table is itself a small dynamic program; it turns an $O(n)$ check at each node into $O(1)$. Restore IP Addresses (LC 93) and Word Break II (LC 140) have the same shape.

Best-First Search and A*

Backtracking explores the tree depth-first, in whatever order the candidates happen to come. For optimization problems, it can pay to always expand the **most promising** partial solution next. **Best-first search** keeps all the partial solutions generated so far in a priority queue, ordered by a cost, and repeatedly expands the cheapest one.

When can it stop? Not necessarily at the first complete solution it pops. That solution was the cheapest *partial* solution when it was chosen, but a more expensive partial solution might still finish more cheaply. The search can stop once the cheapest entry left in the queue costs at least as much as the best complete solution found. That rule is only correct if a partial solution's cost is a **lower bound** on the cost of every completion of it. Otherwise something deeper in the queue could still turn into a better answer.

A* sharpens the cost. Ordering by *cost so far* alone favors short partial solutions: in a traveling-salesman search, half a tour is almost always cheaper than any full tour, so every half-tour gets expanded before the search finishes anything. A* orders by *cost so far* + *an estimate of the remaining cost*. If the estimate never overestimates, the total is still a lower bound, the stopping rule stays correct, and partial solutions near completion are no longer penalized. With a consistent estimate, like the one below, the first complete solution popped is optimal. Dijkstra's algorithm is A* with an estimate of zero.

Sliding Puzzle (LC 773) asks for the fewest moves that turn a 2×3 board into `[[1, 2, 3], [4, 5, 0]]`, where each move slides a tile into the blank. Each move shifts one tile by one cell, so every tile needs at least as many moves as its Manhattan distance from its goal cell. The sum of those distances is therefore a lower bound on the moves remaining:

```
import heapq


def sliding_puzzle(board: list[list[int]]) -> int:
    goal = (1, 2, 3, 4, 5, 0)
    moves_from = {0: (1, 3), 1: (0, 2, 4), 2: (1, 5), 3: (0, 4), 4: (1, 3, 5), 5: (2, 4)}

    def estimate(state: tuple[int, ...]) -> int:
        """Sum of each tile's Manhattan distance to its goal cell: never overestimates."""
        return sum(
            abs(i // 3 - (tile - 1) // 3) + abs(i % 3 - (tile - 1) % 3)
            for i, tile in enumerate(state)
            if tile
        )

    start = tuple(tile for row in board for tile in row)
    fewest = {start: 0}                          # fewest moves found so far to each state
    heap = [(estimate(start), 0, start)]         # (moves + estimate, moves, state)
    while heap:
        _, moves, state = heapq.heappop(heap)
        if state == goal:
            return moves
        if moves > fewest[state]:
            continue                             # stale entry
        blank = state.index(0)
        for j in moves_from[blank]:
            nxt = list(state)
            nxt[blank], nxt[j] = nxt[j], nxt[blank]
            nxt = tuple(nxt)
            if moves + 1 < fewest.get(nxt, moves + 2):
                fewest[nxt] = moves + 1
                heapq.heappush(heap, (moves + 1 + estimate(nxt), moves + 1, nxt))
    return -1                                    # the goal is unreachable from this board
```

This board has only 360 reachable states, so plain BFS also works. A* matters when the state space is large: the tighter the estimate, the more of the space it never touches.

The price is memory. Depth-first backtracking stores only the current path, while best-first search stores the whole frontier. In Skiena's traveling-salesman experiments with 11 cities, the priority queue grew to 202,063 entries; the backtracking stack needed 11. A slow program eventually gives an answer, but one that runs out of memory never does.

> **Take-Home Lesson**
>
> The promise of a partial solution is not just its cost so far, but also the cost of the rest of the solution. An estimate of the remaining cost that is tight, yet still a lower bound, makes best-first search far more efficient.

Meet in the Middle

At $n = 40$, enumerating all $2^{40} \approx 10^{12}$ subsets is hopeless, but $2^{20} \approx 10^{6}$ is easy. **Meet in the middle** gets from one to the other by splitting the items into two halves, enumerating all subsets of each half separately, and combining the two lists cleverly instead of trying every pair.

Closest Subsequence Sum (LC 1755): find the subset sum closest to `goal`, with $n \le 40$. Every subset is a subset of the left half plus a subset of the right half, so its sum is $s_L + s_R$. For each left sum s_L, the best partner is the right sum closest to goal $- s_L$. After sorting the right sums, binary search finds it:

```
import bisect


def subset_sums(nums: list[int]) -> list[int]:
    sums = [0]
    for x in nums:
        sums += [s + x for s in sums]            # every old subset, with and without x
    return sums


def min_abs_difference(nums: list[int], goal: int) -> int:
    half = len(nums) // 2
    left = subset_sums(nums[:half])
    right = sorted(subset_sums(nums[half:]))
    best = abs(goal)                             # the empty subset
    for s in left:
        i = bisect.bisect_left(right, goal - s)  # the best partner is right[i - 1] or right[i]
        for j in (i - 1, i):
            if 0 <= j < len(right):
                best = min(best, abs(s + right[j] - goal))
    return best
```

Time: $O(2^{n/2} \cdot n)$, dominated by sorting the right half's $2^{n/2}$ sums: well under a second in Python at $n = 40$, compared with hours for brute force. The idea works whenever a solution splits into two independent halves whose combination can be checked quickly: by sorting and binary search, a hash map, or two pointers. **Partition Array Into Two Arrays to Minimize Sum Difference** (LC 2035) is the same trick, with the left and right sums grouped by how many elements they use.

In LC 2035, `nums` has $2n$ elements and each part must take exactly n. If the first part takes k elements from the left half, it must take $n-k$ from the right, so sums are matched only within those groups. With $t =$ `total - 2 * s`, the difference is $|t - 2r|$, so the best partner r is the one closest to $t/2$:

```
def minimum_difference(nums: list[int]) -> int:
    n, total = len(nums) // 2, sum(nums)

    def sums_by_count(half: list[int]) -> list[list[int]]:
        groups: list[list[int]] = [[] for _ in range(n + 1)]
        for mask in range(1 << n):
            groups[mask.bit_count()].append(sum(x for i, x in enumerate(half) if mask >> i & 1))
        return groups

    left = sums_by_count(nums[:n])
    right = [sorted(g) for g in sums_by_count(nums[n:])]
    best = float("inf")
    for k in range(n + 1):
        partners = right[n - k]
        for s in left[k]:
            t = total - 2 * s
            i = bisect.bisect_left(partners, t // 2)
            for j in (i - 1, i):
                if 0 <= j < len(partners):
                    best = min(best, abs(t - 2 * partners[j]))
```

```
    return best
```

Heuristic Search (Briefly)

When the search space is too large even for pruning, and an exact answer isn't required, **heuristic search** trades the guarantee of optimality for speed. **Random sampling** tries random solutions and keeps the best. **Local search** starts from some solution and repeatedly makes small changes, such as swapping two cities in a tour, as long as they improve it. It stops at a *local* optimum, which may be far from the best. **Simulated annealing** also accepts some changes that make the solution worse, with a probability that shrinks over time. That lets it escape local optima early, then settle down. These methods power real-world scheduling and layout tools, but they don't appear in interview problems, which always have exact answers.

When Backtracking Should Become DP

If the same sub-search is reached by different paths — for example, "can the rest of the string starting at position i be segmented into words?" in Word Break — backtracking recomputes it every time. When the answer to a sub-search depends only on a small **state** (here, just i), cache it. That's memoization, and it turns exponential backtracking into polynomial dynamic programming. The signal: the subproblem's answer doesn't depend on the choices that led to it. See Dynamic Programming.

Complexity Summary

Problem	Search space	Time
Subsets	2^n leaves	$O(n \cdot 2^n)$
Permutations	$n!$ leaves	$O(n \cdot n!)$
k-combinations	$\binom{n}{k}$ leaves	$O(k\binom{n}{k})$
N-Queens	$\leq n!$	$O(n!)$, much less in practice
Word Search	$mn \cdot 3^L$ paths	$O(mn \cdot 3^L)$ for word length L
Palindrome partitioning	2^{n-1} ways to cut	$O(n \cdot 2^n)$
Meet in the middle on subsets	$2 \cdot 2^{n/2}$ half-subsets	$O(n \cdot 2^{n/2})$

Common Mistakes

1. **Appending the shared list instead of a copy.** `result.append(chosen)` stores a reference to the list you keep mutating; at the end every entry is empty. Use `chosen[:]`.
2. **Incomplete undo.** Every change made before the recursive call must be reversed after it — the path, the `used` flags, the constraint sets, the board cell.

3. **Wrong duplicate-skipping condition.** `i > start` (skip repeated *siblings*), not `i > 0` (which also forbids legitimate repeats deeper in the tree).
4. **Checking constraints only at the leaves.** Test partial solutions as early as possible; that is where pruning gets its power.
5. **Passing `i + 1` vs. `i`.** Reuse allowed (LC 39): recurse with `i`. No reuse (LC 40, LC 78): recurse with `i + 1`.
6. **Building strings by concatenation in deep recursions.** Use a list and `"".join` when recording a solution.
7. **Using backtracking where the subproblems overlap.** If the future depends only on a small state, memoize it.
8. **Stopping best-first search too early.** The first complete solution popped is optimal only when the queue's costs are lower bounds on every completion (and, for A*, the estimate never overestimates). Otherwise keep searching until the cheapest queue entry can't beat the best solution found.

Practice Problems

Problem	Idea
Subsets (LC 78)	Choose the next element
Subsets II (LC 90)	Sort + skip repeated siblings
Permutations (LC 46)	`used` flags
Permutations II (LC 47)	Equal values in original order
Combinations (LC 77)	Choose-next with room-left pruning
Combination Sum (LC 39)	Reuse allowed; sorted `break` pruning
Combination Sum II (LC 40)	No reuse; sibling skip
Generate Parentheses (LC 22)	Feasibility pruning on counts
Letter Combinations of a Phone Number (LC 17)	Product of candidate sets
All Paths From Source to Target (LC 797)	Paths in a DAG
Word Search (LC 79)	Grid DFS with mark / unmark
Palindrome Partitioning (LC 131)	Cut at palindromic prefixes
N-Queens (LC 51)	Column / diagonal sets
Sudoku Solver (LC 37)	Most constrained cell first
Partition to K Equal Sum Subsets (LC 698)	Ordering + symmetry pruning
Matchsticks to Square (LC 473)	Same as LC 698 with $k = 4$
Find Minimum Time to Finish All Jobs (LC 1723)	Branch and bound
Sliding Puzzle (LC 773)	A* with Manhattan distance (or BFS)
Closest Subsequence Sum (LC 1755)	Meet in the middle
Partition Array Into Two Arrays to Minimize Sum Difference (LC 2035)	Meet in the middle, grouped by size

Greedy Algorithms

A greedy algorithm builds a solution one decision at a time, always taking the choice that looks best **right now**, and never reconsidering. It is the most natural way to attack an optimization problem, and when it works it is hard to beat: usually a sort followed by a single pass.

The difficulty is that it often doesn't work. Greedy algorithms are easy to invent, easy to code, and — very frequently — wrong in ways that simple test cases won't reveal. So this chapter is only partly about greedy *techniques*. It is mostly about the discipline that makes them safe: **proving a greedy choice correct, or finding the counterexample that shows it isn't**.

> **Take-Home Lesson**
>
> There is a fundamental difference between an **algorithm**, which always produces a correct result, and a **heuristic**, which usually does a good job but offers no guarantee. A greedy procedure is a heuristic until you have proved it is an algorithm.

Where Greedy Sits

Every optimization problem can be attacked in three ways:

Approach	Correct?	Fast?
Exhaustive search (backtracking)	Always — it tries everything	Usually exponential
Greedy	Only with proof — it commits to one choice	Usually $O(n \log n)$
Dynamic programming	Always — it tries everything *systematically*	Polynomial when there are few distinct subproblems

Greedy and DP both build a solution from decisions. The difference is that DP keeps the consequences of **every** option for each decision, while greedy keeps only **one**. So greedy is correct precisely when you can prove that the locally best choice is always *part of some* optimal solution — that you never need the options you threw away.

When Greedy Fails

Before learning to prove greedy algorithms right, it helps to see how easily they go wrong.

Making change. Pay n units with as few coins as possible. Greedy: repeatedly take the largest coin that fits. With US coins $\{1, 5, 10, 25\}$ this is optimal. But with coins $\{1, 6, 10\}$ and $n = 12$, greedy takes $10 + 1 + 1$ (three coins), while $6 + 6$ uses two. Nothing in the greedy rule changed; the *problem instance* did. Coin Change (LC 322) therefore needs DP.

0/1 knapsack. Pack items into a bag of capacity W to maximize total value. Greedy by value-per-weight fails with capacity 10 and items (weight 6, value 7), (weight 5, value 5), (weight 5, value 5): greedy takes

the densest item first and has room for nothing else (value 7), while the two lighter items give 10. The exact solution is DP. But if items can be **cut** (the *fractional* knapsack), greedy by density is optimal — the troublesome leftover space can always be filled with a fraction of the next item.

Movie scheduling. Choosing the largest set of non-overlapping intervals by earliest start or by shortest length both fail, on instances with just two or three intervals (see Intervals).

The pattern in all three: greedy fails when an early choice **blocks** better combinations later. A proof of correctness must show that this can't happen.

How to Prove a Greedy Algorithm Correct

1. The exchange argument

Take any optimal solution that *disagrees* with the greedy choice, and show you can **exchange** part of it for the greedy choice without making it worse. Then some optimal solution agrees with greedy's first choice; repeat the argument for the rest.

Example: minimizing total completion time. One machine, n jobs with processing times t_i. In what order should they run to minimize the sum of their completion times?

Claim: **shortest job first**. Suppose an optimal order has two adjacent jobs a then b with $t_a > t_b$. Swap them. Jobs before and after the pair are unaffected. Before the swap, the pair contributes $(T + t_a) + (T + t_a + t_b)$ to the sum; after, $(T + t_b) + (T + t_b + t_a)$. The swap saves $t_a - t_b > 0$. So an optimal order has no such "inversions" — it's sorted by processing time.

```python
def min_total_completion_time(times: list[int]) -> int:
    total = elapsed = 0
    for t in sorted(times):                 # shortest processing time first
        elapsed += t
        total += elapsed
    return total
```

The same adjacent-swap argument proves many "sort by the right key" algorithms: sort by the key under which swapping any out-of-order adjacent pair never hurts.

2. Greedy stays ahead

Show by induction that after each step, greedy's partial solution is **at least as good** as any other algorithm's partial solution after the same number of steps — so at the end, greedy can't be behind. The proof that earliest-finish-first maximizes the number of non-overlapping intervals has this form: greedy's k-th interval always ends no later than any other solution's k-th interval.

3. Match a lower bound

Find a simple reason why **no** solution can do better than some value L, then show greedy achieves L. For meeting rooms, L is the maximum number of meetings overlapping at one instant; greedy room assignment never opens more rooms than that (see Intervals). This is often the quickest proof when it applies.

... and always hunt for counterexamples

Before investing in a proof, spend two minutes trying to break the greedy rule: **think small** (two or three items), **seek extremes** (one huge item among tiny ones), and **go for ties** (make everything look equally

attractive so the rule has nothing to go on). If a small counterexample exists, you'll usually find it quickly; if you can't find one, the attempt often shows you why the proof works.

Classic Greedy Algorithms

Earliest deadline first

n jobs each have a processing time and a deadline. Is there an order in which every job finishes by its deadline? **Sort by deadline.** If any feasible order exists, this one is feasible: by the exchange argument, if a job with a later deadline runs immediately before one with an earlier deadline, swapping them doesn't make either one late — the earlier-deadline job finishes sooner, and the later-deadline job now finishes when the first one used to, which met an even earlier deadline.

```
def all_deadlines_met(jobs: list[tuple[int, int]]) -> bool:
    """jobs[i] = (processing_time, deadline)."""
    elapsed = 0
    for duration, deadline in sorted(jobs, key=lambda j: j[1]):
        elapsed += duration
        if elapsed > deadline:
            return False
    return True
```

If jobs can be *dropped* to fit as many as possible (Course Schedule III, LC 630), process by deadline and keep a max-heap of the durations taken; whenever the total exceeds the current deadline, drop the longest job so far. Dropping the longest job frees the most time while losing only one job.

Huffman coding

To compress text, give frequent symbols short codes and rare symbols long ones, with no code a prefix of another. **Huffman's algorithm**: repeatedly merge the two **least frequent** symbols into a combined symbol whose frequency is their sum, until one remains. The merges form a binary tree; each symbol's code is its root-to-leaf path. The greedy choice is right because the two rarest symbols can always be placed as siblings at the deepest level of some optimal tree (an exchange argument).

With a heap, this is $O(n \log n)$. The same algorithm solves **Minimum Cost to Connect Sticks** (LC 1167), where merging two sticks costs their total length:

```
import heapq


def connect_sticks(sticks: list[int]) -> int:
    heap = sticks[:]
    heapq.heapify(heap)
    cost = 0
    while len(heap) > 1:
        merged = heapq.heappop(heap) + heapq.heappop(heap)   # the two cheapest
        cost += merged
        heapq.heappush(heap, merged)
    return cost
```

Graph algorithms

Several of the most important graph algorithms are greedy, each with an exchange-argument proof: **Prim's** and **Kruskal's** minimum spanning tree algorithms (always take the lightest edge that crosses a cut) and **Dijkstra's** shortest paths (always finalize the closest unfinished vertex). See Graphs.

Interview Patterns

Each of these comes with the argument that makes it correct. The argument is the part to remember.

Reachability: Jump Game (LC 55)

`nums[i]` is the maximum jump from index i. Can you reach the end?

Track the **farthest index reachable** so far. If the scan ever reaches an index beyond it, that index — and everything after it — is unreachable.

```python
def can_jump(nums: list[int]) -> bool:
    reach = 0
    for i, jump in enumerate(nums):
        if i > reach:
            return False
        reach = max(reach, i + jump)
    return True
```

Why: the reachable indices always form a prefix $[0, \text{reach}]$ — if you can reach index j, you can reach every index before it (by stopping short on the jump that passed it).

Jump Game II (LC 45): BFS in disguise

Minimum number of jumps to reach the end. Think of it as **BFS by levels**: level k is the range of indices reachable in exactly k jumps, and the next level extends to the farthest point reachable from anywhere in the current one.

```python
def jump(nums: list[int]) -> int:
    jumps = 0
    level_end = farthest = 0
    for i in range(len(nums) - 1):
        farthest = max(farthest, i + nums[i])
        if i == level_end:              # finished scanning this level
            jumps += 1
            level_end = farthest
    return jumps
```

It's greedy in the sense of never tracking individual paths, but its correctness is BFS's: levels are explored in order, so the first level containing the last index gives the fewest jumps.

Gas Station (LC 134)

A circular route of stations; `gas[i]` is available at station i and `cost[i]` is needed to reach the next. Find a start from which the full loop is possible.

```python
def can_complete_circuit(gas: list[int], cost: list[int]) -> int:
    if sum(gas) < sum(cost):
        return -1
    start = tank = 0
    for i in range(len(gas)):
        tank += gas[i] - cost[i]
        if tank < 0:                    # can't reach i + 1 from start
            start, tank = i + 1, 0
    return start
```

Two facts make this correct. **(1)** If starting at s you first run dry before reaching $j+1$, then no start between s and j works either: you arrived at each of them with a non-negative tank, so starting there fresh (with an empty tank) can only be worse. That's why the start jumps straight to $j+1$. **(2)** If the total gas covers the total cost, the start that survives the scan works for the whole loop: the deficit accumulated before it is exactly covered by the surplus gathered after it.

Partition Labels (LC 763)

Split a string into as many parts as possible so that each letter appears in only one part. Each part must extend at least to the **last occurrence** of every letter inside it; greedily close a part the moment the scan reaches the farthest last-occurrence seen so far.

```python
def partition_labels(s: str) -> list[int]:
    last = {ch: i for i, ch in enumerate(s)}
    sizes = []
    start = end = 0
    for i, ch in enumerate(s):
        end = max(end, last[ch])
        if i == end:
            sizes.append(end - start + 1)
            start = i + 1
    return sizes
```

Closing as early as possible is safe because any valid partition must include the whole range [start, end] in one part; closing there leaves the rest free to be split as finely as possible.

Constraints from both sides: Candy (LC 135)

Children in a row with ratings; each gets at least one candy, and a child with a higher rating than a neighbor gets more than that neighbor. Minimize total candy.

Each child faces two independent constraints — from the left neighbor and from the right. Satisfy each with a greedy pass, then take the maximum:

```python
def candy(ratings: list[int]) -> int:
    n = len(ratings)
    left = [1] * n
    for i in range(1, n):
        if ratings[i] > ratings[i - 1]:
            left[i] = left[i - 1] + 1
    right = [1] * n
    for i in range(n - 2, -1, -1):
        if ratings[i] > ratings[i + 1]:
            right[i] = right[i + 1] + 1
    return sum(max(l, r) for l, r in zip(left, right))
```

Why minimal: `left[i]` is the smallest value satisfying all left-side constraints, and `right[i]` all right-side ones — each is forced by the length of the increasing run leading to i. So every valid assignment gives child i at least `max(left[i], right[i])`, and that assignment is itself valid.

Pairing after sorting

Boats to Save People (LC 881): each boat holds two people, with a weight limit. Sort; pair the heaviest remaining person with the lightest if they fit, otherwise send the heaviest alone.

```
def num_rescue_boats(people: list[int], limit: int) -> int:
    people = sorted(people)
    lo, hi = 0, len(people) - 1
    boats = 0
    while lo <= hi:
        if people[lo] + people[hi] <= limit:
            lo += 1                     # the lightest rides along
        hi -= 1                         # the heaviest always leaves
        boats += 1
    return boats
```

Exchange argument: if the heaviest person can share with anyone, they can share with the lightest. In any optimal solution, swapping the lightest person into the heaviest's boat (and moving whoever was there into the lightest's old spot) keeps every boat within the limit. If the heaviest can't share even with the lightest, they must go alone.

Assign Cookies (LC 455) and **Two City Scheduling** (LC 1029, sort by the *difference* in the two costs) follow the same "sort, then match" shape, each justified by an exchange.

Structure forced by the smallest element: Hand of Straights (LC 846)

Can the cards be split into groups of W consecutive values? The **smallest** remaining card can't be the middle or end of a group — nothing smaller is left — so it must start one. That forced choice is the greedy step:

```
from collections import Counter


def is_n_straight_hand(hand: list[int], w: int) -> bool:
    count = Counter(hand)
    for x in sorted(count):
        need = count[x]
        if need:
            for y in range(x, x + w):
                if count[y] < need:
                    return False
                count[y] -= need
    return True
```

When a greedy choice is **forced** — every valid solution must make it — no exchange argument is needed at all.

Counting the slack: Task Scheduler (LC 621)

Tasks with a cooldown of n between identical tasks; find the minimum total time. Let the most frequent task appear f times, and m tasks tie for that frequency. The most frequent task alone forces $f - 1$ full cycles of length $n + 1$, plus a final partial cycle containing the m tied tasks — a **lower bound** of $(f - 1)(n + 1) + m$. Other tasks fill the idle slots; if there are more tasks than slots, there's no idle time at all and the answer is

simply the number of tasks.

```
def least_interval(tasks: list[str], n: int) -> int:
    freq = Counter(tasks).values()
    f = max(freq)
    m = sum(1 for v in freq if v == f)
    return max(len(tasks), (f - 1) * (n + 1) + m)
```

Local gains: Best Time to Buy and Sell Stock II (LC 122)

With unlimited transactions, every price increase between consecutive days can be captured. The total profit is the sum of all positive day-to-day differences — any longer transaction's profit decomposes into exactly those steps.

```
def max_profit(prices: list[int]) -> int:
    return sum(max(0, b - a) for a, b in zip(prices, prices[1:]))
```

Greedy as Approximation

For some problems, no efficient exact algorithm is known at all (they are NP-complete), and greedy is used as a heuristic with a **provable quality guarantee**.

Set cover: choose as few of the given subsets as possible to cover every element. Greedy repeatedly picks the subset covering the **most still-uncovered** elements. It isn't always optimal — but it is never worse than about $\ln n$ times optimal. The argument: when u elements are still uncovered, the OPT subsets of an optimal solution cover all of them, so one of those subsets covers at least u/OPT. Greedy picks a subset at least that good, so each pick shrinks the uncovered count by a factor of at least $1 - 1/\text{OPT}$. After $\text{OPT} \cdot \ln n$ picks, fewer than $n \cdot (1 - 1/\text{OPT})^{\text{OPT} \ln n} < n \cdot e^{-\ln n} = 1$ elements remain — none.

> **Take-Home Lesson**
>
> When a problem is too hard to solve exactly, a greedy heuristic with a proven approximation ratio is a principled compromise: fast, simple, and guaranteed to be close. "Guaranteed close" is a far stronger statement than "usually works."

Common Mistakes

1. **Trusting a greedy rule without proof.** Passing the examples proves nothing. Try small counterexamples; if none turns up, sketch the exchange argument.
2. **Sorting by the wrong key.** Earliest start vs. earliest end, value vs. value per weight, cost vs. cost *difference*. The right key is the one under which an adjacent swap never helps.
3. **Using greedy on problems that need DP.** Coin change with arbitrary denominations, 0/1 knapsack, longest increasing subsequence: any time an early choice can block a better combination later.
4. **Forgetting the global feasibility check.** Gas Station first checks that total gas covers total cost; without it, the scan's answer is meaningless.
5. **Confusing "optimal for these examples" with "optimal."** US coin denominations make greedy change-making correct; most denomination systems don't.

Practice Problems

Problem	Greedy idea	Justification
Assign Cookies (LC 455)	Sort both, match smallest adequate	Exchange
Best Time to Buy and Sell Stock II (LC 122)	Sum positive differences	Decomposition
Jump Game (LC 55)	Track max reach	Reachable set is a prefix
Jump Game II (LC 45)	Level-by-level reach	BFS
Gas Station (LC 134)	Reset start on deficit	Skipped starts are no better
Partition Labels (LC 763)	Close at farthest last occurrence	Forced range
Candy (LC 135)	Two passes, take max	Each pass is the minimum for its side
Boats to Save People (LC 881)	Heaviest with lightest	Exchange
Two City Scheduling (LC 1029)	Sort by cost difference	Exchange
Hand of Straights (LC 846)	Smallest card starts a group	Forced choice
Task Scheduler (LC 621)	Frequency formula	Lower bound achieved
Non-overlapping Intervals (LC 435)	Earliest finish first	Stays ahead
Minimum Cost to Connect Sticks (LC 1167)	Merge two smallest	Huffman exchange
Course Schedule III (LC 630)	By deadline, drop longest	Exchange
Queue Reconstruction by Height (LC 406)	Tallest first, insert at index k	Shorter people don't affect taller ones' counts

Dynamic Programming

The hardest algorithmic problems ask for the *best* of something: the cheapest, the longest, the fewest. For these optimization problems we've seen two approaches, each with a fatal flaw. **Exhaustive search** tries every possibility, so it's always correct — but it's usually exponential. **Greedy** algorithms commit to one locally best choice, so they're fast — but they're usually wrong.

Dynamic programming combines the best of both. Like exhaustive search, it considers every possibility, so it's correct. But it notices that a naive search solves the **same subproblems over and over**, and it solves each one just once, storing the answer in a table. When the number of distinct subproblems is small, an exponential search collapses into a polynomial algorithm.

Until you understand it, dynamic programming seems like magic. Once you do, it's perhaps the easiest design technique to apply — often easier to re-derive from scratch than to look up.

Caching vs. Computation

The simplest illustration is the Fibonacci numbers, $F_n = F_{n-1} + F_{n-2}$ with $F_0 = 0$, $F_1 = 1$.

Recursion. Transcribe the definition directly:

```python
def fib_recursive(n: int) -> int:
    if n < 2:
        return n
    return fib_recursive(n - 1) + fib_recursive(n - 2)
```

It's correct, and it's catastrophically slow. The call tree for F_n has F_{n+1} leaves, and F_n grows like 1.618^n: computing F_{45} this way makes over a billion calls. The waste is plain in the call tree: F_{n-2} is computed twice, F_{n-3} three times, F_{n-4} five times…

Caching. There are only $n + 1$ distinct arguments. Store each result the first time it's computed, and look it up after that:

```python
from functools import cache


@cache
def fib_memo(n: int) -> int:
    if n < 2:
        return n
    return fib_memo(n - 1) + fib_memo(n - 2)
```

Now each value is computed once: $O(n)$ time. This is **memoization**, or top-down DP.

> **Take-Home Lesson**
>
> Explicitly caching the results of recursive calls gives most of the benefit of dynamic programming — usually the same running time. If you can write a correct recursive solution whose arguments take few distinct values, adding a cache is often all it takes.

Computation in order. Better still, notice that F_i depends only on smaller values, so fill a table from smallest to largest. No recursion at all:

```python
def fib_table(n: int) -> int:
    F = [0] * (n + 1)                    # F[i] = the i-th Fibonacci number
    if n > 0:
        F[1] = 1
    for i in range(2, n + 1):
        F[i] = F[i - 1] + F[i - 2]
    return F[n]
```

This is **tabulation**, or bottom-up DP. Now look at what the loop *reads*: only `F[i-1]` and `F[i-2]`. Everything older is dead, so two variables replace the table:

```python
def fib_dp(n: int) -> int:
    back2, back1 = 0, 1                  # F[i-2], F[i-1]
    for _ in range(n):
        back2, back1 = back1, back1 + back2
    return back2
```

$O(n)$ time, $O(1)$ space. Every space-optimized DP in this chapter is a full table compressed this way.

The Recipe

Dynamic programming is a way of efficiently implementing a **recursive** algorithm. So the recurrence comes first, and the table second. Every DP solution goes through three steps:

1. **Formulate the answer as a recurrence** relating the problem to smaller instances of itself. This is where correctness lives.
2. **Show that the recurrence takes only polynomially many distinct parameter values.** Those values are the **states** — the cells of the table. Write down in one sentence what a cell means ("`T[i][j]` = can some subset of the first i items sum to exactly j?"). If you can't, the recurrence isn't clear yet. This is where efficiency lives.
3. **Choose an evaluation order** so that every subproblem is solved before it's needed.

Then, optionally:

- **Compress the table.** Only after the full table is correct, check which cells the recurrence reads. If row i needs only row $i-1$, keep one or two rows.
- **Reconstruct the solution** itself (not just its cost) by recording which choice won at each state, then following those choices backward.

The running time is always

$$\text{(number of states)} \times \text{(work to evaluate one state)}.$$

Top-down or bottom-up?

	Memoization (top-down)	Tabulation (bottom-up)
How	Recursive function + `@cache`	Loops filling a table in dependency order

	Memoization (top-down)	Tabulation (bottom-up)
Evaluation order	Automatic	You must choose it
States computed	Only those reachable from the start	All of them
Space optimization	Hard	Easy: keep only the rows you still need
Pitfalls	Recursion depth; cache overhead	Getting the loop order right

Write the memoized version first when the recurrence is new to you — it's the recurrence, transcribed. Convert to tabulation when you need to save space, avoid deep recursion, or squeeze out constant factors.

One Sequence: "The Best Answer Ending Here"

The most common DP state for a sequence is an index i, meaning "the best answer for the first i elements" or "the best answer that **ends at** element i."

House Robber (LC 198)

Maximize the sum of chosen houses with no two adjacent. For the first i houses, either the i-th house is robbed (so house $i-1$ isn't) or it isn't:

$$\text{best}(i) = \max\big(\text{best}(i-1),\ \text{best}(i-2) + a_i\big)$$

```python
def rob(nums: list[int]) -> int:
    n = len(nums)
    best = [0] * (n + 1)                # best[i] = max loot from the first i houses
    for i in range(1, n + 1):
        leave = best[i - 1]                                    # house i not robbed
       take = (best[i - 2] if i >= 2 else 0) + nums[i - 1]    # robbed: house i-1 must be skipped
        best[i] = max(leave, take)
    return best[n]
```

`best[i]` reads only the previous two entries, so, as with Fibonacci, two variables suffice:

```python
def rob(nums: list[int]) -> int:
    back2 = back1 = 0                   # best[i-2], best[i-1]
    for x in nums:
        back2, back1 = back1, max(back1, back2 + x)
    return back1
```

That "take it or leave it" split — the item is either in the solution or not — is the most common recurrence pattern in all of DP. **Climbing Stairs** (LC 70) is Fibonacci; **Maximum Subarray** (LC 53) uses "best ending here" (see Kadane's algorithm).

Word Break (LC 139)

Can the string be split into dictionary words? `ok[i]` is true if the prefix `s[:i]` can be split. It can be exactly when some split point $j < i$ has `ok[j]` true and `s[j:i]` is a word:

```python
def word_break(s: str, words: list[str]) -> bool:
    dictionary = set(words)
    longest = max(map(len, dictionary), default=0)
    ok = [True] + [False] * len(s)      # ok[i] = can s[:i] be split into words? (empty prefix:
↪   yes)
    for i in range(1, len(s) + 1):
        for j in range(max(0, i - longest), i):
            if ok[j] and s[j:i] in dictionary:
                ok[i] = True
                break
    return ok[-1]
```

This is exactly what the backtracking solution computes, except that "can `s[j:]` be segmented?" is answered once per j instead of once per path to j. **Decode Ways** (LC 91) has the same shape.

Longest Increasing Subsequence (LC 300)

Given a sequence, find the longest subsequence (not necessarily contiguous) whose elements strictly increase. For $S = (2, 4, 3, 5, 1, 7, 6, 9, 8)$, one answer is $(2, 3, 5, 6, 8)$, of length 5.

Deriving the recurrence is instructive. Ask: *what would I need to know about the first $n-1$ elements to finish the job?*

- The **length** L of the longest increasing subsequence of the first $n-1$ elements? Not enough. If $L = 5$ and $s_n = 8$, whether 8 extends it depends on whether *some* length-5 subsequence ends below 8.
- So we need, for each possible ending element, the longest subsequence ending there. Define L_i = length of the longest increasing subsequence **ending at** s_i:

$$L_i = 1 + \max_{\substack{j<i \\ s_j<s_i}} L_j \qquad \text{(or 1 if no such } j\text{)}$$

The answer is $\max_i L_i$ — the winning subsequence must end somewhere.

i	1	2	3	4	5	6	7	8	9
s_i	2	4	3	5	1	7	6	9	8
L_i	1	2	2	3	1	4	4	5	5

Store each element's best predecessor, and the subsequence itself can be read backward from the element with the largest L_i:

```python
def longest_increasing_subsequence(nums: list[int]) -> list[int]:
    n = len(nums)
    length = [1] * n                    # length[i] = longest increasing subsequence ending at
↪   nums[i]
    prev = [-1] * n                     # prev[i] = index of the element before nums[i] in it
    for i in range(n):
        for j in range(i):
            if nums[j] < nums[i] and length[j] + 1 > length[i]:
                length[i] = length[j] + 1
                prev[i] = j
```

```
    if not nums:
        return []
    i = max(range(n), key=length.__getitem__)
    seq = []
    while i != -1:
        seq.append(nums[i])
        i = prev[i]
    return seq[::-1]
```

$O(n^2)$ time. Notice the move we made — the obvious quantity wasn't enough, so we asked for more information about every prefix. It's the same "strengthen the hypothesis" move as in tree recursion, and it's how most DP states are discovered.

$O(n \log n)$ with patience sorting

Keep an array `tails`, where `tails[k]` is the **smallest possible last element** of an increasing subsequence of length $k+1$ seen so far. `tails` is always strictly increasing, so for each new element, binary search finds the first tail $\geq x$ and replaces it (or appends x if x beats them all):

```
import bisect


def length_of_lis(nums: list[int]) -> int:
    tails: list[int] = []
    for x in nums:
        k = bisect.bisect_left(tails, x)
        if k == len(tails):
            tails.append(x)             # x extends the longest subsequence so far
        else:
            tails[k] = x                # a length-(k+1) subsequence can now end lower
    return len(tails)
```

Replacing a tail with a smaller value never hurts: a subsequence ending lower is at least as easy to extend later. (`tails` is *not* itself an increasing subsequence of the input — only its length is meaningful.) **Russian Doll Envelopes** (LC 354) is LIS after sorting by width ascending and height *descending*, so that envelopes of equal width can't nest.

Two Sequences: Edit Distance (LC 72)

How many single-character **insertions, deletions, and substitutions** turn string P into string T? This edit distance underlies spell checkers, `diff`, and DNA alignment.

Finding the best sequence of edits looks daunting: where should the insertions go? The trick is to think about the problem **in reverse**, and ask only what happens to the **last characters**. In an optimal alignment of $P[1..i]$ and $T[1..j]$, the last operation is one of:

- **Match or substitute** P_i with T_j: cost 0 if they're equal, else 1, plus the cost of aligning $P[1..i-1]$ with $T[1..j-1]$.
- **Insert** T_j: cost 1, plus aligning $P[1..i]$ with $T[1..j-1]$.
- **Delete** P_i: cost 1, plus aligning $P[1..i-1]$ with $T[1..j]$.

There's no other possibility. So, with $D[i][j]$ the edit distance between the first i characters of P and the first j of T:

$$D[i][j] = \min \begin{cases} D[i-1][j-1] + [P_i \neq T_j] \\ D[i][j-1] + 1 \\ D[i-1][j] + 1 \end{cases} \qquad D[i][0] = i,\ D[0][j] = j$$

The naive recursion branches three ways at every step — exponential. But there are only $(|P|+1)(|T|+1)$ distinct (i, j) pairs, so the table has that many cells, each filled in $O(1)$:

```python
def min_distance(p: str, t: str) -> int:
    m, n = len(p), len(t)
    # D[i][j] = edit distance between p[:i] and t[:j]
    D = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1):
        D[i][0] = i                                  # delete all of p[:i]
    for j in range(n + 1):
        D[0][j] = j                                  # insert all of t[:j]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            D[i][j] = min(
                D[i - 1][j - 1] + (p[i - 1] != t[j - 1]),   # match / substitute
                D[i][j - 1] + 1,                            # insert t[j-1]
                D[i - 1][j] + 1,                            # delete p[i-1]
            )
    return D[m][n]
```

$O(mn)$ time and space. To reconstruct the actual edits, start at `D[m][n]` and step back to whichever of the three neighbors produced its value.

Compressing. Row `i` reads only row `i - 1` (`D[i-1][j-1]`, `D[i-1][j]`) and the cell to its left in row `i` itself. So keep two rows, `prev` for row `i - 1` and `curr` for row `i`, giving $O(n)$ space (but no reconstruction):

```python
def min_distance(p: str, t: str) -> int:
    m, n = len(p), len(t)
    prev = list(range(n + 1))                    # row 0: D[0][j] = j
    for i in range(1, m + 1):
        curr = [i] + [0] * n                     # D[i][0] = i
        for j in range(1, n + 1):
            curr[j] = min(
                prev[j - 1] + (p[i - 1] != t[j - 1]),
                curr[j - 1] + 1,
                prev[j] + 1,
            )
        prev = curr
    return prev[n]
```

One routine, many problems

Small changes to the costs, the base cases, or the goal cell turn edit distance into other classic problems:

- **Longest Common Subsequence** (LC 1143): forbid substitution. Then the only way to handle mismatched characters is to delete them, so the cheapest alignment keeps the most matches. Directly:

```python
def longest_common_subsequence(a: str, b: str) -> int:
    m, n = len(a), len(b)
    # L[i][j] = length of the LCS of a[:i] and b[:j]; row/column 0 = empty prefix, LCS 0
```

```
    L = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                L[i][j] = L[i - 1][j - 1] + 1          # both last characters join the LCS
            else:
                L[i][j] = max(L[i - 1][j], L[i][j - 1]) # drop a's last or b's last
    return L[m][n]
```

The same two-row compression as edit distance brings the space down to $O(n)$.

- **Delete Operation for Two Strings** (LC 583): $|a| + |b| - 2 \cdot \text{LCS}$.
- **Longest increasing subsequence** is the LCS of the sequence with its own sorted, de-duplicated version.
- **Approximate substring matching**: to find where pattern P best matches *anywhere* in text T, make skipping a prefix of T free ($D[0][j] = 0$) and take the best cell in the last row. Without this change, the "best" match of a short pattern in a long text is dominated by the cost of deleting everything else.
- **Distinct Subsequences** (LC 115), **Interleaving String** (LC 97), **Regular Expression Matching** (LC 10), and **Wildcard Matching** (LC 44) are all two-index DPs with the same "what happens to the last characters?" derivation.

Regular Expression Matching (LC 10) is the trickiest of these, because `*` belongs to the character before it. Let $M[i][j]$ mean that `s[:i]` matches `p[:j]`. If the pattern ends in `x*`, either the pair matches nothing (drop it: $M[i][j-2]$), or it matches the last character of `s` and may match more ($M[i-1][j]$, keeping `x*`):

```
def is_match(s: str, p: str) -> bool:
    m, n = len(s), len(p)
    M = [[False] * (n + 1) for _ in range(m + 1)]
    M[0][0] = True
    for i in range(m + 1):                      # i = 0 too: "a*b*" matches ""
        for j in range(1, n + 1):
            if p[j - 1] == "*":
                M[i][j] = M[i][j - 2] or (i > 0 and p[j - 2] in (s[i - 1], ".") and M[i - 1][j])
            else:
                M[i][j] = i > 0 and p[j - 1] in (s[i - 1], ".") and M[i - 1][j - 1]
    return M[m][n]
```

Wildcard Matching (LC 44) is simpler: a `*` stands alone and matches any run, so $M[i][j] = M[i][j-1] \vee M[i-1][j]$.

> **Take-Home Lesson**
>
> For optimization problems on objects with a natural **left-to-right order** — characters of a string, elements of a sequence, points along a line, leaves of a tree — dynamic programming very likely gives an efficient algorithm. The states are simply positions in the order.

Subset Sum and Knapsack

All of these problems have the same structure: go through the items one at a time and decide, for each one, **take it or leave it**. So the state needs two indices:

- **how many items have been decided**: a prefix, as in every sequence DP;
- **how much of the budget is left**: the sum still to make, or the capacity still free.

The second index is a *value*, not a position. That's what's new here.

Subset sum: the full table

Problem. Given positive integers $s_1, \ldots, s_n$ and a target k, is there a subset summing to exactly k?

State. $T[i][j]$ = *can some subset of the first i items sum to exactly j?* We need it for every $0 \le i \le n$ and every $0 \le j \le k$, and the answer is $T[n][k]$.

Why every j, and not just k? Taking an item changes the target. If we take s_i, the other items must make $j - s_i$, so we need answers for smaller targets too. As with LIS, the obvious question isn't enough, so we ask a more general one.

Recurrence. Look at the last item, s_i. A subset of the first i items that sums to j either:

- **leaves** s_i: then it's a subset of the first $i-1$ items summing to j, so $T[i-1][j]$;
- **takes** s_i: then the rest of it is a subset of the first $i-1$ items summing to $j - s_i$, so $T[i-1][j-s_i]$ (possible only if $s_i \le j$).

$$T[i][j] = T[i-1][j] \ \vee \ T[i-1][j-s_i]$$

Base case. With no items, the only subset is the empty one, which sums to 0: $T[0][0] =$ true, and $T[0][j] =$ false for $j > 0$.

Example. Items $(2, 3, 5)$, $k = 8$ (T = reachable):

	j=0	1	2	3	4	5	6	7	8
$i = 0$: no items	T								
$i = 1$: + 2	T		T						
$i = 2$: + 3	T		T	T		T			
$i = 3$: + 5	T		T	T		T		T	T

Each row is the row above, plus the row above shifted right by s_i. $T[3][8]$ is true because $T[2][3]$ is true: take the 5, and $\{2, 3\}$ can make the remaining 3.

```python
def subset_sum(nums: list[int], k: int) -> bool:
    n = len(nums)
    # T[i][j] = can some subset of nums[:i] sum to exactly j?
    T = [[False] * (k + 1) for _ in range(n + 1)]
    T[0][0] = True                                  # empty subset makes 0
    for i in range(1, n + 1):
        x = nums[i - 1]                             # the i-th item
        for j in range(k + 1):
            T[i][j] = T[i - 1][j]                   # leave x
            if j >= x and T[i - 1][j - x]:
                T[i][j] = True                      # take x
    return T[n][k]
```

That's $O(nk)$ time for n items and target k. This is polynomial in the *value* k, not in the number of bits needed to write k down. It's fast when k is modest, but with $k \approx 10^{18}$ the table is useless. (Subset sum is NP-complete; DP doesn't contradict that.)

Partition Equal Subset Sum (LC 416)

Problem. Can `nums` be split into two groups with equal sums?

This is subset sum with $k = \text{total}/2$. Why is finding *one* group of that sum enough? Because every element goes into exactly one of the two groups. If group A sums to a, the other group is everything else, and it sums to total $- a$. So:

- If the two groups are equal, then $a = \text{total} - a$, so $a = \text{total}/2$. A valid split always contains a subset summing to total/2.
- If some subset A sums to total/2, put every other element in B. Then B sums to total $-$ total/2 $=$ total/2 with no extra work.

Finding one half is enough, because its complement is the other half. If the total is odd, no integer half exists.

```python
def can_partition(nums: list[int]) -> bool:
    total = sum(nums)
    if total % 2:
        return False                    # odd total: two equal integer halves are impossible
    return subset_sum(nums, total // 2) # the complement of that subset is the other half
```

Compressing to one array: why downward

Row i reads only row $i - 1$, so we can keep a single array, `reachable[j]` = *can some subset of the items seen so far sum to j?*, and overwrite it in place, one item at a time.

There's a catch. The take branch reads `T[i-1][j - x]`, a cell **to the left** in the **previous** row. If `j` goes upward, then by the time we reach `j`, the cell `j - x` has already been overwritten with row i. It may already include x, so x gets used twice.

Take `nums = [3]`, `k = 6`. Going upward, `reachable[3]` becomes true (0 + 3). Then `reachable[6]` reads `reachable[3]` and becomes true, meaning "6 = 3 + 3", which uses the single 3 twice. Going **downward**, `j = 6` is computed while `reachable[3]` still holds the previous row (false), and only then does `j = 3` become true. That's correct.

```python
def can_partition(nums: list[int]) -> bool:
    total = sum(nums)
    if total % 2:
        return False
    target = total // 2
    reachable = [True] + [False] * target       # reachable[j] = some subset of the items so
↪   far sums to j
    for x in nums:
        for j in range(target, x - 1, -1):      # downward: reachable[j - x] is still the
         ↪  previous row
            reachable[j] = reachable[j] or reachable[j - x]
    return reachable[target]
```

The loop stops at `x`: for `j < x` only "leave" is possible, so `T[i][j] = T[i-1][j]`, and the cell is left unchanged.

0/1 knapsack

Items have weights w_i and values v_i. Each can be used **at most once**. Maximize the total value with total weight at most C.

State. best$[i][c]$ = the largest value from a subset of the first i items with total weight $\le c$.

Recurrence. It's the same take-or-leave split, but now we compare values (max) instead of asking "is it possible?" (or):

$$\text{best}[i][c] = \max\big(\text{best}[i-1][c],\ \text{best}[i-1][c-w_i] + v_i\big) \qquad \text{(take only if } w_i \le c\text{)}$$

Base case. best$[0][c] = 0$ for every c: no items, no value. (Because the state says weight $\le c$, not $= c$, every capacity is achievable with the empty set.)

```python
def knapsack(weights: list[int], values: list[int], C: int) -> int:
    n = len(weights)
    # best[i][c] = max value from a subset of the first i items with total weight <= c
    best = [[0] * (C + 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(C + 1):
            best[i][c] = best[i - 1][c]                          # leave item i
            if c >= w:
                best[i][c] = max(best[i][c], best[i - 1][c - w] + v)    # take item i
    return best[n][C]
```

The take branch reads row $i-1$, exactly as in subset sum, so the one-array version also goes **downward**: `for c in range(C, w - 1, -1): best[c] = max(best[c], best[c - w] + v)`. Subset sum is the special case where each item's value equals its weight, and we ask whether best$[n][k] = k$.

Unbounded items: coin change (LC 322)

Now each coin can be used **any number of times**. We want the fewest coins summing to exactly `amount`.

State. fewest$[i][s]$ = the fewest coins summing to exactly s, using only the first i coin types (∞ if impossible).

Recurrence. Take or leave coin type i, with value c_i:

- **leave**: we never use c_i again, so fewest$[i-1][s]$;
- **take one copy**: and c_i is *still available* for the rest, so fewest$[\mathbf{i}][s-c_i]+1$.

$$\text{fewest}[i][s] = \min\big(\text{fewest}[i-1][s],\ \text{fewest}[i][s-c_i]+1\big)$$

The only difference from 0/1 is the row index in the take branch: i **instead of** $i-1$. Taking a coin doesn't remove it from the menu.

Base case. fewest$[0][0] = 0$ and fewest$[0][s] = \infty$ for $s > 0$. Here the sum must be *exact*, so unreachable states are ∞, not 0.

```python
def coin_change(coins: list[int], amount: int) -> int:
    INF = float("inf")
    n = len(coins)
    # fewest[i][s] = fewest coins summing to exactly s, using only the first i coin types
    fewest = [[INF] * (amount + 1) for _ in range(n + 1)]
    fewest[0][0] = 0
    for i in range(1, n + 1):
        c = coins[i - 1]
        for s in range(amount + 1):
            fewest[i][s] = fewest[i - 1][s]                          # never use c
            if s >= c:
                fewest[i][s] = min(fewest[i][s], fewest[i][s - c] + 1)  # one more c; row i: c
↪   still allowed
    return fewest[n][amount] if fewest[n][amount] != INF else -1
```

Compressing. The take branch now reads the **current** row at `s - c`. In a single array, iterating **upward** gives exactly that, because `fewest[s - c]` has already been updated for coin `c`. So the loop direction isn't a trick to memorize. It follows from which row the recurrence reads:

```python
def coin_change(coins: list[int], amount: int) -> int:
    INF = amount + 1                          # more coins than any real answer needs
    fewest = [0] + [INF] * amount             # fewest[s] = fewest coins (types so far) making s
    for c in coins:
        for s in range(c, amount + 1):        # upward: fewest[s - c] is already row i
            fewest[s] = min(fewest[s], fewest[s - c] + 1)
    return fewest[amount] if fewest[amount] != INF else -1
```

Here greedy fails (coins $\{1, 6, 10\}$, amount 12; see Greedy), but DP is always right, because it considers every possibility for each coin.

Counting: combinations vs. permutations

When **counting** the ways to reach a total, the right state depends on whether order matters.

Combinations (LC 518), where $\{1, 2\}$ and $\{2, 1\}$ are the same way. Use the coin-change state: ways$[i][s]$ = the number of ways to make s using only the first i coin types. Every way either uses no c_i at all, or uses at least one c_i (remove one copy, and the rest still uses types $1..i$):

$$\text{ways}[i][s] = \text{ways}[i-1][s] + \text{ways}[i][s - c_i]$$

Coin types are decided in a fixed order (c_1 first, then c_2, …), so each multiset of coins is counted exactly once. In one array this becomes coins outer, sums upward, as in coin change.

Permutations (LC 377), where $(1, 2)$ and $(2, 1)$ are different. Now there is no "first i items" index, because any item can come at any position. The state is just ways$[s]$ = the number of ordered sequences summing to s. Split on the **last element** x: the sequence is some sequence summing to $s - x$, followed by x.

$$\text{ways}[s] = \sum_{x \le s} \text{ways}[s - x], \qquad \text{ways}[0] = 1 \text{ (the empty sequence)}$$

To compute ways$[s]$ we need every smaller total, with every item allowed, so totals go in the outer loop. This is Climbing Stairs with arbitrary step sizes. It's a genuinely 1-D state, not a compressed table.

```python
def count_combinations(coins: list[int], amount: int) -> int:
    """LC 518. ways[s] = ways to make s with the coin types seen so far."""
    ways = [1] + [0] * amount
    for c in coins:                           # coin types outer: the table's row index
        for s in range(c, amount + 1):        # upward: c may be reused
            ways[s] += ways[s - c]
    return ways[amount]


def count_sequences(nums: list[int], target: int) -> int:
    """LC 377. ways[s] = ordered sequences summing to s."""
    ways = [1] + [0] * target
    for s in range(1, target + 1):            # totals outer
        for x in nums:                        # every choice of last element
            if x <= s:
                ways[s] += ways[s - x]
    return ways[target]
```

Variant	State	Take branch reads	One-array loops
0/1: subset sum, knapsack	first i items, sum/capacity	row $i-1$	items outer, capacity **downward**
Unbounded: coin change	first i item types, sum	row i	items outer, capacity **upward**
Count combinations	first i item types, sum	row i	items outer, upward
Count sequences	sum only; split on the last element	—	totals outer, items inner

Ordered Partition

Three workers must scan a shelf of books, each taking a **contiguous** section. How should the shelf be divided to minimize the largest section? For books with page counts $100, 200, \ldots, 900$, splitting into equal numbers of books gives sections of 600, 1500, and 2400 pages. The best split, $(100 \ldots 500 \mid 600, 700 \mid 800, 900)$, has a largest section of 1700.

Heuristics like "aim for the average" fail on some inputs. Instead, think recursively about the **last divider**. If the last section is books $i+1 \ldots n$, the cost is the larger of that section's total and the best way to split the first i books into $k-1$ sections:

$$M[n][k] = \min_{i<n} \max\left(M[i][k-1], \sum_{j=i+1}^{n} s_j\right)$$

With prefix sums, each range sum is $O(1)$, and the table has nk cells each taking $O(n)$: $O(kn^2)$ total.

```python
def linear_partition(s: list[int], k: int) -> int:
    n = len(s)
    prefix = [0]
    for x in s:
        prefix.append(prefix[-1] + x)
    INF = float("inf")
    # M[j][p]: best max-section for the first j items in p sections
    M = [[INF] * (k + 1) for _ in range(n + 1)]
    M[0][0] = 0
    for p in range(1, k + 1):
        for j in range(1, n + 1):
            for i in range(p - 1, j):                    # last section is s[i:j]
                M[j][p] = min(M[j][p], max(M[i][p - 1], prefix[j] - prefix[i]))
    return min(M[n][p] for p in range(1, k + 1))       # "k or fewer" sections
```

This is **Split Array Largest Sum** (LC 410). The binary search on the answer solution is faster, $O(n \log \sum s)$ — but the DP generalizes to costs where no greedy feasibility check exists.

Interval DP

Some problems are naturally about **substrings or subarrays** $[i, j]$, and the answer for an interval combines answers for smaller intervals inside it. Evaluate in order of increasing **length**, so that every smaller interval is ready.

Longest Palindromic Subsequence (LC 516): if the end characters match, they wrap a palindrome inside; otherwise drop one end.

```python
def longest_palindrome_subseq(s: str) -> int:
    n = len(s)
    dp = [[0] * n for _ in range(n)]                    # dp[i][j] = longest palindromic
    ↪ subsequence of s[i..j]
    for i in range(n - 1, -1, -1):                      # i descending: dp[i+1] is ready
        dp[i][i] = 1
        for j in range(i + 1, n):
            if s[i] == s[j]:
                dp[i][j] = dp[i + 1][j - 1] + 2
            else:
                dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])
    return dp[0][n - 1] if n else 0
```

Burst Balloons (LC 312) is the famous hard case, and it shows a key trick for interval DP: think about the **last** operation inside an interval, not the first. If balloon k is the last one burst in the open interval (i, j), then its neighbors at that moment are exactly i and j, and the two sides are independent subproblems:

$$\text{dp}[i][j] = \max_{i<k<j} \left(\text{dp}[i][k] + a_i \cdot a_k \cdot a_j + \text{dp}[k][j]\right)$$

Pad the array with a virtual balloon of value 1 at each end, and fill in order of increasing interval length, so both subintervals are ready:

```python
def max_coins(nums: list[int]) -> int:
    a = [1] + nums + [1]
    n = len(a)
    dp = [[0] * n for _ in range(n)]            # dp[i][j]: best for the open interval (i, j)
    for length in range(2, n):
        for i in range(n - length):
            j = i + length
            dp[i][j] = max(dp[i][k] + a[i] * a[k] * a[j] + dp[k][j] for k in range(i + 1, j))
    return dp[0][n - 1]
```

Choosing the *first* balloon instead would leave neighbors that depend on everything burst later — no clean subproblems. The same "last step" idea gives matrix-chain multiplication and optimal binary search trees, all $O(n^3)$.

State Machine DP

When the decision at each step depends on a small amount of **mode** — holding a stock or not, in cooldown or not — make the mode part of the state. **Best Time to Buy and Sell Stock with Cooldown** (LC 309) has three modes, so the state is (day d, mode), and each cell is the best profit at the end of day d in that mode:

- hold[d]: holding a stock. Either we already held it, or we bought today, which needs the "free" mode

yesterday: $\text{hold}[d] = \max(\text{hold}[d-1],\ \text{rest}[d-1] - p_d)$

- $\text{sold}[d]$: sold today, so tomorrow is cooldown. We must have held yesterday: $\text{sold}[d] = \text{hold}[d-1] + p_d$
- $\text{rest}[d]$: not holding and free to buy tomorrow. Either we were already free, or yesterday's sale finished its cooldown: $\text{rest}[d] = \max(\text{rest}[d-1],\ \text{sold}[d-1])$

Before day 1, holding is impossible ($-\infty$) and the other two modes have profit 0. Day d reads only day $d-1$, so three variables replace the three columns:

```python
def max_profit_cooldown(prices: list[int]) -> int:
    hold, sold, rest = float("-inf"), 0, 0      # hold[d-1], sold[d-1], rest[d-1]
    for p in prices:
        hold, sold, rest = max(hold, rest - p), hold + p, max(rest, sold)
    return max(sold, rest)
```

The tuple assignment matters: every right-hand side must read *yesterday's* values. The other stock problems (LC 123, 188, 714) add a transaction counter or a fee to the same machine. With at most two transactions (LC 123), the modes are "after the first buy," "after the first sell," and so on:

```python
def max_profit_two_transactions(prices: list[int]) -> int:
    buy1 = buy2 = float("-inf")
    sell1 = sell2 = 0
    for p in prices:
        buy1, sell1, buy2, sell2 = (
            max(buy1, -p), max(sell1, buy1 + p), max(buy2, sell1 - p), max(sell2, buy2 + p)
        )
    return sell2
```

For at most k transactions (LC 188), use arrays `buy[1..k]` and `sell[1..k]` with `buy[t]` reading `sell[t - 1]`.

Beyond Sequences: Trees and Subsets

On trees, the natural order is bottom-up: a node's answer combines its children's. **House Robber III** (LC 337) returns a pair per node — the best total with the node robbed, and without it — which is DP with the state stored in the recursion's return value (see Trees).

On subsets, when there's no natural order, the state must remember *which* items have been used. Consider finding the shortest route visiting every city exactly once (the traveling salesman problem). A recurrence over "the path so far" is correct but has $n!$ states — it's just backtracking. But the future doesn't depend on the **order** of the cities visited, only on **which** cities were visited and **where you are now**. That gives $2^n \cdot n$ states:

$$\text{best}[S][v] = \min_{u \in S \setminus \{v\}} \big(\text{best}[S \setminus \{v\}][u] + d(u, v)\big)$$

```python
def shortest_tour(dist: list[list[int]]) -> int:
    """Held-Karp: shortest cycle through all cities, starting and ending at city 0."""
    n = len(dist)
    INF = float("inf")
    best = [[INF] * n for _ in range(1 << n)]           # best[S][v] = shortest path from 0
↪   visiting exactly S, ending at v
    best[1][0] = 0                                      # visited {0}, standing at 0
```

```
    for S in range(1 << n):
        if not S & 1:
            continue                                    # every path starts at city 0
        for v in range(n):
            if best[S][v] == INF:
                continue
            for w in range(n):
                if not S >> w & 1:
                    T = S | 1 << w
                    best[T][w] = min(best[T][w], best[S][v] + dist[v][w])
    full = (1 << n) - 1
    return min(best[full][v] + dist[v][0] for v in range(n)) if n > 1 else 0
```

$O(n^2 2^n)$ — feasible for $n \approx 20$, where $n!$ would be hopeless. Subsets as integers are covered in Bit Manipulation.

When Does DP Work?

When is it correct? DP requires the **principle of optimality**: an optimal solution can be extended using only the *state* reached so far, not the specific decisions that led there. Edit distance satisfies it — the best way to finish aligning two strings doesn't depend on which edits were used to align their prefixes. A "longest simple path" recurrence that tracks only the current vertex violates it: whether the next step is legal depends on *which* vertices the path has already used. Fix it by enlarging the state (the subset of visited vertices), at exponential cost.

When is it efficient? When the number of states is small. Objects with an inherent **left-to-right order** — strings, sequences, rooted trees — have only $O(n)$ or $O(n^2)$ possible "stopping points," so they give polynomial DPs. Without such an order, the state must record an arbitrary subset, and the table is exponential.

> **Take-Home Lesson**
>
> Without an inherent left-to-right ordering on the objects, dynamic programming is usually doomed to require exponential space and time. When you're designing a DP, look first for the order that lets you describe a partial solution by a position (or a pair of positions).

Recognizing a DP Problem

- The problem asks for an **optimum** (min, max, longest, fewest) or a **count** (number of ways) — rarely for the actual list of all solutions.
- A brute-force search would make a **sequence of decisions**, and the future depends only on a **small summary** of the decisions made so far.
- The recursive brute force **revisits** the same arguments.
- Greedy fails on small counterexamples.
- Constraints are modest in the dimensions that define the state: $n \leq 5000$ for $O(n^2)$ states, $n \leq 20$ for $O(2^n)$ subset states, sums up to about 10^4–10^5 for knapsack.

Common Mistakes

1. **Writing the table before the recurrence.** Get a correct recursive definition first; the table is just a way to evaluate it.
2. **Missing information in the state.** If the recurrence needs to know something the state doesn't record (like the last element in LIS, or the mode in the stock problems), it will be wrong. Add it to the state.
3. **Wrong evaluation order.** Every state must be computed after the states it depends on. Interval DP goes by increasing length; 1-D knapsack goes downward for 0/1, upward for unbounded.
4. **Off-by-one in table dimensions.** Tables indexed by prefix *length* have $n + 1$ rows, with row 0 for the empty prefix. Decide whether `dp[i]` means "first i items" or "ending at item i", and write it down.
5. **Wrong base cases.** `dp[0] = 1` for counting (one way to make zero), `dp[0] = 0` for costs; unreachable states should be $\pm\infty$, not 0.
6. **Recursion depth in memoized solutions.** `@cache` recursion 10^4 levels deep exceeds Python's limit. Tabulate, or raise the limit carefully.
7. **Confusing combinations and permutations** in counting problems — the loop order decides.

Practice Problems

Problem	State / idea
Climbing Stairs (LC 70)	Fibonacci
House Robber (LC 198)	Take or skip
House Robber II (LC 213)	Two linear runs (circle broken)
Decode Ways (LC 91)	Prefix length
Word Break (LC 139)	Prefix segmentable?
Unique Paths (LC 62)	Grid counting (Pascal's triangle)
Minimum Path Sum (LC 64)	Grid cost
Longest Increasing Subsequence (LC 300)	Best ending here; patience sorting
Russian Doll Envelopes (LC 354)	LIS after a clever sort
Longest Common Subsequence (LC 1143)	Two prefixes
Edit Distance (LC 72)	What happens to the last characters
Partition Equal Subset Sum (LC 416)	Subset sum, downward 1-D
Coin Change (LC 322)	Unbounded, upward
Coin Change II (LC 518)	Count combinations: items outer
Combination Sum IV (LC 377)	Count sequences: totals outer
Longest Palindromic Subsequence (LC 516)	Interval DP
Burst Balloons (LC 312)	Interval DP on the last operation
Best Time to Buy and Sell Stock with Cooldown (LC 309)	State machine
House Robber III (LC 337)	Tree DP returning a pair
Split Array Largest Sum (LC 410)	Ordered partition
Maximal Square (LC 221)	Largest square ending at each cell
Regular Expression Matching (LC 10)	Two indices, pattern cases

References

The Cheat Sheet and Common Patterns notes are study notes based on:

> *Steven S. Skiena.* The Algorithm Design Manual*, 3rd edition. Texts in Computer Science. Springer, Cham, 2020. ISBN 978-3-030-54255-9. doi:10.1007/978-3-030-54256-6*