The Pattern Playbook

A Senior Engineer's Field Guide to DSA Interviews

Recognize the pattern → Apply the technique → Implement with confidence

Master 7 core patterns: Greedy • Dynamic Programming • Divide & Conquer • Two Pointers / Sliding Window • Graphs • Backtracking • Binary Search & Heaps

How to read this guide — the "Onion" method. Every pattern is taught in three concentric layers. **Layer 1 — The Big Idea:** the one-sentence mental model and the trigger that tells you "this is the pattern." **Layer 2 — Going Deeper:** the mechanics, the recurring sub-types, and the reusable code skeleton. **Layer 3 — Expert Lens:** the subtle traps, complexity reasoning, and the senior-level insight that separates a passing answer from a great one. Read Layer 1 for all patterns first to build intuition, then circle back through Layers 2 and 3.

Prepared for Fernando • Target role: Senior Software Engineer • Python reference implementations • Practice sets + final exam included

Contents

1 Orientation: How Interviewers Think in Patterns **3**
- 1.1 The Recognition Cheat-Sheet . . . 3
- 1.2 A 90-Second Complexity Refresher . . . 3

2 Pattern 1 — Greedy **4**
- 2.1 Sub-types you will meet . . . 4
- 2.2 The reusable skeleton . . . 5
- 2.3 Worked intuition — Activity Selection . . . 5
- 2.4 Complexity profile . . . 5

3 Pattern 2 — Dynamic Programming **6**
- 3.1 The five-step recipe (say these out loud in the interview) . . . 6
- 3.2 Sub-types you will meet . . . 6
- 3.3 Two skeletons: top-down and bottom-up . . . 7
- 3.4 Worked intuition — 0/1 Knapsack table . . . 7
- 3.5 Complexity profile . . . 8

4 Pattern 3 — Divide & Conquer **9**
- 4.1 The Master Theorem in one box . . . 9
- 4.2 Sub-types you will meet . . . 9
- 4.3 The reusable skeleton . . . 10
- 4.4 Worked intuition — counting inversions while merging . . . 10
- 4.5 Complexity profile . . . 10

5 Pattern 4 — Two Pointers & Sliding Window **11**
- 5.1 Sub-types you will meet . . . 11
- 5.2 Two skeletons . . . 12
- 5.3 Worked intuition — the window breathing . . . 12
- 5.4 Complexity profile . . . 13

6 Pattern 5 — Graphs: BFS, DFS & Union-Find **14**
- 6.1 Choosing your traversal . . . 14
- 6.2 Sub-types you will meet . . . 14

6.3 Three skeletons . . . 15
6.4 Worked intuition — BFS spreads in rings . . . 15
6.5 Complexity profile . . . 16

7 Pattern 6 — Backtracking 17
7.1 The decision tree . . . 17
7.2 Sub-types you will meet . . . 17
7.3 The universal skeleton . . . 18
7.4 Complexity profile . . . 18

8 Pattern 7 — Binary Search, Heaps & Tree Structures 19
8.1 Binary search on the answer — the mental shift . . . 19
8.2 Sub-types you will meet . . . 19
8.3 Three skeletons . . . 20
8.4 Complexity profile . . . 21

9 Meta-Strategy: A Unified Decision Process 22
9.1 The master decision flowchart . . . 22
9.2 How the families relate . . . 23
9.3 The interview communication protocol . . . 23

10 Appendix: One-Page Pattern Cheat-Sheet 24

11 Final Examination — 30 Questions 25

Orientation: How Interviewers Think in Patterns

A data-structures-and-algorithms (DSA) interview is rarely a test of whether you can invent an algorithm on the spot. It is a test of **pattern recognition under time pressure**. Strong candidates do not see "a problem about painting houses" or "a problem about jumping across a river." They see *"this is an optimization over a sequence with an optimal-substructure smell — probably DP"* or *"this is asking for the minimum window satisfying a constraint — sliding window."* The surface story is a costume; underneath, only a dozen or so skeletons recur.

This guide trains that X-ray vision. For each of the seven families below you will learn the **trigger phrases** that betray the pattern, the **sub-types** the family splits into, a **reusable code skeleton** you can adapt in minutes, the **complexity profile** interviewers expect you to state, and the **pitfalls** that sink otherwise-correct solutions. At the end of each section is a curated practice set, ordered roughly easy → hard, so you can immediately rehearse the recognition reflex. A 30-question multiple-choice exam closes the guide.

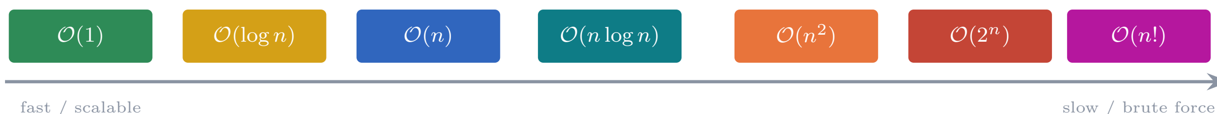
1.1 The Recognition Cheat-Sheet

Before the deep dives, internalize this triage table. In an interview, your first 60 seconds should be spent mapping the prompt onto one of these rows.

If the prompt says...	...the trigger you should feel	...reach for
"maximum/minimum," "how many ways," "can you reach" over choices	Overlapping sub-problems + optimal substructure (optimal answer is built from optimal sub-answers)	**Dynamic Programming**
"maximize/minimize" but a locally-best choice seems safe	Exchange argument (prove swapping any element for the greedy choice never worsens the solution); sortable by a key	**Greedy**
"sorted array," "find the boundary," "minimize the maximum"	Monotonic search space (a predicate over the answer is all-false then all-true, with a single crossover point)	**Binary Search**
"contiguous subarray/substring," "window," "at most K"	A sliding constraint over a sequence	**Sliding Window**
"pair that sums," "in-place," "sorted, two ends"	Converging indices	**Two Pointers**
"connected," "shortest path," "dependency," "grid"	Nodes + edges; reachability/order	**Graph BFS/DFS/Union-Find**
"all permutations/combinations/-subsets," "place N ..."	Build-and-undo a decision tree	**Backtracking**
"split the input and combine," "sort," "count inversions"	Recurse on halves, merge results	**Divide & Conquer**
"top K," "running median," "merge K streams"	Need the best element repeatedly	**Heap / Priority Queue**

1.2 A 90-Second Complexity Refresher

Interviewers expect you to state time and space complexity *unprompted*. Anchor every solution to this ladder, from fastest to slowest, and say the dominant term out loud.



Rules of thumb for input size n (a year-2020s laptop does $\sim 10^8$ simple ops/second): if $n \leq 12$ think $\mathcal{O}(n!)$ backtracking is fine; $n \leq 20$ suggests $\mathcal{O}(2^n)$ bitmask DP; $n \leq 2000$ tolerates $\mathcal{O}(n^2)$ DP; $n \leq 10^5$ wants $\mathcal{O}(n \log n)$ or better; $n \geq 10^6$ demands $\mathcal{O}(n)$ or $\mathcal{O}(\log n)$. Working backward from the constraint to the target complexity is itself a pattern-selection tool.

Pattern 1 — Greedy

greedy view

Layer 1 — The Big Idea: A greedy algorithm builds a solution one step at a time, and at each step it grabs the choice that looks best *right now* — never reconsidering. It is the optimist of algorithms: it assumes that a sequence of locally optimal moves lands on the globally optimal answer. The entire art of greedy is proving (or trusting) that this optimism is justified for the specific problem.
Layer 2 — Going Deeper: Greedy works only when the problem has the **greedy-choice property** (a globally optimal solution can be reached by a locally optimal choice) and **optimal substructure** (an optimal solution contains optimal solutions to sub-problems). In practice you almost always (1) sort the input by a cleverly chosen key, or (2) maintain a heap to always pull the current best, then (3) sweep once making the obvious choice. If a clean sort key exists and a swap argument holds, greedy beats DP because it is one pass instead of a table.
Layer 3 — Expert Lens: The senior-level skill is the **exchange argument** (a proof technique): to prove greedy is correct, assume some optimal solution differs from the greedy one at some position, then show you can swap that differing element for the greedy choice without making the objective worse — therefore greedy is at least as good as optimal, i.e., it *is* optimal. If you cannot construct such an argument, greedy is probably wrong and the problem is secretly DP. Classic traps: the coin-change problem is greedy for canonical coin systems (US coins) but *not* for arbitrary denominations (e.g. coins {1,3,4}, target 6: greedy picks 4+1+1=3 coins, optimal is 3+3=2 coins).

2.1 Sub-types you will meet

- **Interval scheduling / activity selection.** Maximize non-overlapping intervals. *Key:* sort by *end* time, greedily take the earliest finisher.
- **Interval merging & minimum resources.** Merge overlaps, or count the max simultaneous overlap (meeting rooms). *Key:* sort by start, sweep with a heap of end times.
- **Sorting by a ratio / deadline.** Job sequencing, fractional knapsack. *Key:* sort by value/weight or by deadline.
- **Greedy on a heap.** Repeatedly combine the two best elements (Huffman coding, "minimum cost to connect ropes," task schedulers).
- **Greedy + invariant sweep.** Jump Game, Gas Station, "Candy" — maintain a running "reach" or "tank" and decide on the fly.

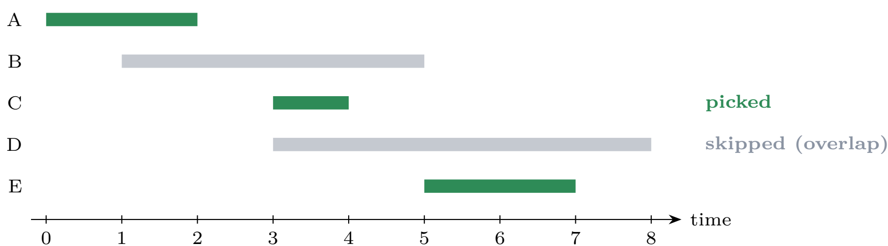
2.2 The reusable skeleton

Greedy almost always collapses to *sort, then sweep maintaining one running quantity.* Memorize this shape:

```
def greedy_template(items):
    # 1) Choose the sort key that makes the locally-best choice obvious.
    items.sort(key=lambda x: x.end)       # sort key: earliest finish time
                                          # (finishing earliest leaves maximum
                                          #  room for subsequent intervals)

    chosen, frontier = [], float('-inf') # 2) frontier = end-time of last chosen
                                          #    interval; float('-inf') is negative
                                          #    infinity -- a sentinel smaller than
                                          #    any real start time so the very
                                          #    first interval always passes the check
    for it in items:
        if it.start ≥ frontier:          # 3) greedy choice valid if it starts
                                          #    at or after the last chosen interval ends
            chosen.append(it)             #    take it (earliest-finishing = greedy pick)
            frontier = it.end             #    advance frontier: next item must start ≥ here
    return chosen                         # 4) one pass, O(n log n) total (sort dominates)
```

2.3 Worked intuition — Activity Selection

Why sort by *end* time and not start time or duration? A diagram makes the exchange argument visible: finishing earliest leaves the most room for everything after it.



Sorting by end time yields A (ends at 2) → C (starts 3, ends 4) → E (starts 5) for three activities. Any solution that picked the long interval B or D early would block more of the timeline — the exchange argument says swapping it for the earlier finisher never hurts.

> ! **Greedy traps to verbalize before you commit.** (1) *Does a counterexample exist?* Spend 20 seconds inventing a small adversarial input. (2) *Is the sort key right?* Sorting interval problems by start vs. end vs. length gives different — sometimes wrong — answers. (3) *Ties.* Decide tie-break order explicitly. (4) *If you can't justify it, switch to DP.* An interviewer who asks "are you sure greedy is optimal here?" is usually signaling that it is not.

2.4 Complexity profile

Dominated by the sort: $\mathcal{O}(n \log n)$ time, $\mathcal{O}(1)$–$\mathcal{O}(n)$ space. Heap-based greedy (Huffman, ropes) is $\mathcal{O}(n \log n)$ as well, with $\mathcal{O}(n)$ space for the heap.

Practice Problems

Warm-up: Assign Cookies; Lemonade Change; Best Time to Buy/Sell Stock II. **Core:** Jump Game; Jump Game II; Gas Station; Non-overlapping Intervals; Merge Intervals; Meeting Rooms II; Task Scheduler; Partition Labels. **Stretch:** Candy; Minimum Number of Arrows to Burst Balloons; Reorganize String; Hand of Straights; Minimum Cost to Connect Sticks (Huffman); IPO (heap + greedy).

Pattern 2 — Dynamic Programming

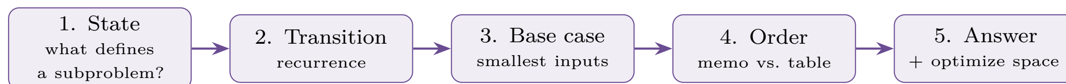
dpcol view

Layer 1 — The Big Idea: Dynamic programming is **recursion with memory**. You take a problem whose brute-force solution explores an exponential tree of choices, notice that the same sub-problems appear over and over, and *cache* each answer so it is computed once. DP = "brute-force recursion" + "stop recomputing what you already know." If you can phrase the answer as a recurrence on smaller versions of the same problem, you have a DP.

Layer 2 — Going Deeper: Two ingredients license DP: **overlapping sub-problems** (the recursion tree revisits the same sub-problem many times — if each sub-problem were unique, plain recursion or D&C would suffice) and **optimal substructure** (the globally optimal answer is composed of locally optimal answers to sub-problems — if the best solution to the whole problem doesn't use the best solution to a sub-problem, DP doesn't apply). The workflow is mechanical once you practice it: (1) define the **state** — the minimal set of variables that *uniquely* identifies a sub-problem (think: "what do I need to know to answer this sub-problem in isolation?"); (2) write the **recurrence/transition** — how a state's answer is built from strictly smaller states; (3) set **base cases** (the smallest inputs whose answers are known without recursion); (4) choose **top-down memoization** (recursion + cache, easiest to derive — start here) or **bottom-up tabulation** (iterative fill of a table, often faster and space-optimizable); (5) read the answer off the table. Most of the difficulty is step 1 — choosing the state.

Layer 3 — Expert Lens: Experts think in terms of the **DP "dimension"**: how many indices does a state need? 1-D (Fibonacci, House Robber, Climbing Stairs), 2-D (edit distance, LCS, knapsack, grid paths), interval DP (matrix-chain, burst balloons — state is a range $[i, j]$), DP-on-trees (rerooting), and bitmask DP (state encodes a visited-set, for $n \leq 20$). The real senior move is **state-space reduction**: a 2-D table that only depends on the previous row collapses to two rows or one (knapsack space optimization), turning $\mathcal{O}(nW)$ space into $\mathcal{O}(W)$. Recognizing that the transition only looks "one step back" is what lets you answer the inevitable "can you do it in less space?" follow-up.

3.1 The five-step recipe (say these out loud in the interview)



3.2 Sub-types you will meet

- **Linear / 1-D DP.** Fibonacci, Climbing Stairs, House Robber, Decode Ways, Maximum Subarray (Kadane), Word Break.
- **Knapsack family.** 0/1 knapsack, subset-sum, partition equal subset, coin change (unbounded), target sum. State: `dp[i][cap]`.
- **Two-sequence DP.** Longest Common Subsequence, Edit Distance, Distinct Subsequences, regex/wildcard matching. State: `dp[i][j]` over the two strings.
- **Grid DP.** Unique Paths, Minimum Path Sum, Dungeon Game. State: `dp[r][c]`.
- **Interval DP.** Matrix-Chain Multiplication, Burst Balloons, Palindrome Partitioning II. State: `dp[i][j]` over a sub-range; loop length outward.
- **Sequence/LIS DP.** Longest Increasing Subsequence ($\mathcal{O}(n^2)$ DP or $\mathcal{O}(n \log n)$ with patience sorting), Russian Doll Envelopes.
- **Bitmask DP.** Travelling Salesman, "Shortest Path Visiting All Nodes," assignment problems for $n \leq 20$.

3.3 Two skeletons: top-down and bottom-up

The same recurrence, two ergonomics. Derive top-down first (mirrors brute force), then convert to bottom-up if you need speed or space control. *Example mapping:* for Coin Change, `state` = remaining amount, `choices` = each coin denomination, `cost` = 1 (one more coin used), `next_state` = amount−coin, `WORST` = `float('inf')`, `better` = `min`. For House Robber, `state` = house index, `choices` = {rob this house, skip}, `cost(rob)` = `nums[i]`, `step(rob)` = 2, `step(skip)` = 1.

```
# PLACEHOLDER LEGEND (replace each name with your problem's concrete logic):
#   state           -- a tuple (or int) that uniquely identifies a sub-problem,
#                      e.g. (index, remaining_capacity) for knapsack
#   is_base(s)      -- True when s needs no further recursion (base case),
#                      e.g. index == 0 or capacity == 0
#   base_value(s)   -- the answer for that base case, e.g. 0 or float('inf')
#   WORST           -- the identity/sentinel for better(); use float('-inf')
#                      when maximizing, float('inf') when minimizing
#   choices(s)      -- the decisions available at state s, e.g. [skip, take]
#   better(a, b)    -- max(a, b) if maximizing, min(a, b) if minimizing
#   cost(c)         -- the immediate value/penalty of making choice c,
#                      e.g. value[i] when taking item i
#   next_state(s,c)-- the smaller sub-problem reached after making choice c
#                      from state s, e.g. (i-1, cap-wt[i]) after taking item i
#   step(c)         -- (bottom-up only) how far back in the table choice c looks,
#                      e.g. 1 for climbing-stairs 1-step, 2 for 2-step

from functools import lru_cache
# ---------- TOP-DOWN (memoized recursion) ----------
def solve_topdown(n, data):
    @lru_cache(maxsize=None)
    def dp(state):                  # state = minimal tuple identifying sub-problem
        if is_base(state):          # base case: no further choices needed
            return base_value(state)
        best = WORST                # WORST = -inf (max) or +inf (min)
        for choice in choices(state):           # loop over all decisions at this state
            val = cost(choice) + dp(next_state(state, choice))
            # cost(choice)      = immediate reward/penalty of this decision
            # next_state(s, c)  = the strictly smaller sub-problem after choosing c
            best = better(best, val)            # better = max or min
        return best
    return dp(initial_state)        # initial_state = the full original problem

# ---------- BOTTOM-UP (tabulation) ----------
def solve_bottomup(n, data):
    dp = [WORST] * (n + 1)          # WORST = sentinel; same value as top-down
    dp[0] = base_value(0)           # seed the smallest base case
    for i in range(1, n + 1):       # iterate states in dependency order (smallest first)
        for choice in choices(i):
            prev = i - step(choice)
            # step(choice) = how many positions back this choice looks in the table
            # (e.g. 1 for a 1-step move, 2 for a 2-step move)
            if prev ≥ 0:
                dp[i] = better(dp[i], cost(choice) + dp[prev])
    return dp[n]
```

3.4 Worked intuition — 0/1 Knapsack table

State `dp[i][w]` = best value using the first i items with capacity w. Transition: either skip item i (`dp[i-1][w]`) or take it if it fits (`value[i] + dp[i-1][w-wt[i]]`). The table is filled row by row; notice each row depends only on the row above — that is the hook for the $\mathcal{O}(W)$ space optimization.

$w \rightarrow$	0	1	2	3	4	5
$\emptyset$	0	0	0	0	0	0
item1	0	0	3	3	3	3
item2	0	0	3	4	4	7
item3	0	0	3	4	4	7

take vs. skip

! **DP failure modes.** (1) *Under-specified state* — forgetting a variable the answer truly depends on (e.g. "did we already use our one transaction?") silently corrupts the recurrence. (2) *Wrong iteration order* in bottom-up: a state must be computed *after* every state it depends on (for unbounded knapsack iterate capacity ascending; for 0/1 descending when using a 1-D array). (3) *Off-by-one* in base cases for two-sequence DP — the empty-prefix row/column matters. (4) *Reconstructing the choice*: if asked for the actual subsequence (not just its length), store parent pointers or backtrack through the table.

3.5 Complexity profile

Time = (number of states) × (work per transition). For 1-D it is typically $\mathcal{O}(n)$; knapsack $\mathcal{O}(nW)$ where n = items and W = capacity (*pseudo-polynomial* — polynomial in the numeric value of W, which can be exponential in the number of bits needed to represent it, so it is not truly polynomial); two-sequence $\mathcal{O}(nm)$ where n, m = lengths of the two strings; interval DP $\mathcal{O}(n^3)$ (n^2 sub-ranges × n split points); bitmask DP $\mathcal{O}(2^n \cdot n)$ (2^n visited-set states × n transitions each). Space equals the table size unless you collapse a dimension.

Practice Problems

Warm-up: Climbing Stairs; House Robber; Maximum Subarray; Min Cost Climbing Stairs. **Core:** Coin Change; Coin Change II; Word Break; Longest Increasing Subsequence; Longest Common Subsequence; Edit Distance; Unique Paths; Partition Equal Subset Sum; Decode Ways; House Robber II. **Stretch:** Burst Balloons; Best Time to Buy/Sell Stock III & IV; Regular Expression Matching; Palindrome Partitioning II; Stone Game; Shortest Path Visiting All Nodes (bitmask); Russian Doll Envelopes.

Pattern 3 — Divide & Conquer

dccol view

Layer 1 — The Big Idea: Divide and conquer (D&C) solves a problem by **splitting it into smaller independent sub-problems of the same type**, solving each (usually recursively), and **combining** their answers into the full answer. Three verbs: *divide*, *conquer*, *combine*. Merge sort is the canonical picture — halve the array, sort each half, merge.

Layer 2 — Going Deeper: The decisive question is *where the real work lives*: in the split, or in the combine? Merge sort does trivial splitting and heavy merging; quicksort does heavy partitioning and trivial combining. The recursion's cost is captured by a recurrence $T(n) = a\,T(n/b) + f(n)$, where you make a recursive calls on inputs of size n/b and spend $f(n)$ dividing/combining. The **Master Theorem** reads the closed-form complexity straight off a, b, and $f(n)$. D&C differs from DP in one crucial way: its sub-problems are **independent** (they do not overlap), so there is nothing to memoize.

Layer 3 — Expert Lens: Expert recognition: D&C shines when a **linear merge step can extract global information that neither half sees alone** — counting cross-pair inversions, the closest pair of points straddling the dividing line, or the maximum subarray crossing the midpoint. The senior insight is that "combine" is where you earn the speedup: a clever $\mathcal{O}(n)$ merge over two sorted halves turns an $\mathcal{O}(n^2)$ brute force into $\mathcal{O}(n \log n)$. Beware the recursion-depth and stack cost, and beware unbalanced splits (quicksort's $\mathcal{O}(n^2)$ worst case) — randomization or median-of-medians restores the balance that the analysis assumes.

4.1 The Master Theorem in one box

For $T(n) = a\,T(n/b) + f(n)$ — where $a \geq 1$ is the **number of recursive sub-calls**, $b > 1$ is the **factor by which the input shrinks** per call, and $f(n)$ is the **cost of dividing and combining** at each level — compare $f(n)$ against the *critical exponent* $n^{\log_b a}$. In Cases 1 and 3, $\epsilon > 0$ is a small positive constant meaning $f(n)$ must be *polynomially* smaller/larger (not just asymptotically equal to) $n^{\log_b a}$:

Case	Condition	Result
1 — leaves dominate	$f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = \Theta(n^{\log_b a})$
2 — balanced	$f(n) = \Theta(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
3 — root dominates	$f(n) = \Omega(n^{\log_b a + \epsilon})$	$T(n) = \Theta(f(n))$

Examples: merge sort $T(n) = 2T(n/2) + \Theta(n) \Rightarrow$ Case 2 $\Rightarrow \Theta(n \log n)$. Binary search $T(n) = T(n/2) + \Theta(1) \Rightarrow \Theta(\log n)$. Karatsuba multiplication $T(n) = 3T(n/2) + \Theta(n) \Rightarrow \Theta(n^{1.585})$.

4.2 Sub-types you will meet

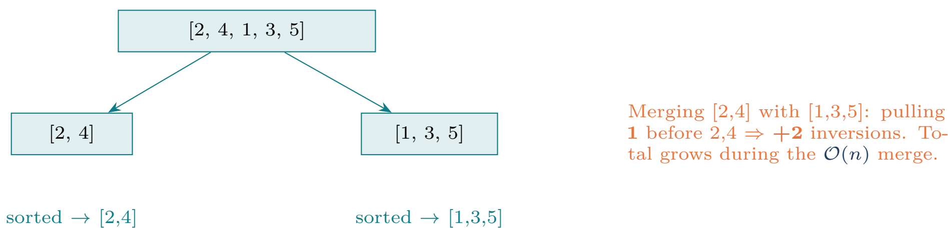
- **Sort-and-merge.** Merge sort; counting inversions during the merge; "Reverse Pairs."
- **Partition-based selection.** Quickselect for the k-th smallest in $\mathcal{O}(n)$ average; quicksort.
- **Crossing-the-midline.** Maximum Subarray (D&C variant); Closest Pair of Points in a plane.
- **Binary recursion on answer structure.** Construct BST/segment tree, "Different Ways to Add Parentheses," "Beautiful Array."
- **Fast arithmetic / matrix.** Karatsuba, Strassen, fast exponentiation ($\mathcal{O}(\log n)$ power).

4.3 The reusable skeleton

```
# Parameters use a HALF-OPEN range convention: the active slice is arr[lo:hi].
#   lo  -- start index, inclusive
#   hi  -- end index, exclusive
def divide_and_conquer(arr, lo, hi):
    # hi - lo == 0: empty slice; hi - lo == 1: single element.
    # Both are trivially solvable (no split needed).
    if hi - lo ≤ 1:
        return solve_trivial(arr, lo, hi)  # e.g., return 0 for counts, arr[lo] for value

    mid = (lo + hi) // 2                # midpoint: left sub-problem = arr[lo:mid]
                                        #           right sub-problem = arr[mid:hi]

    left  = divide_and_conquer(arr, lo, mid)   # solve left half INDEPENDENTLY
    right = divide_and_conquer(arr, mid, hi)   # solve right half INDEPENDENTLY

    # combine(left, right, arr, lo, mid, hi)
    #   left, right  -- results returned by the two recursive calls
    #   arr, lo, mid, hi -- raw bounds so combine can read ACROSS the midpoint
    #   (needed when the answer straddles both halves, e.g., inversion count,
    #    max-subarray crossing midpoint, closest pair of points)
    return combine(left, right, arr, lo, mid, hi)  # O(n) merge; this is where the speedup lives
```

4.4 Worked intuition — counting inversions while merging

An "inversion" is a pair of indices $i < j$ where $a_i > a_j$ — intuitively, elements that are out of order. The total inversion count measures how far an array is from sorted. Brute force checks all pairs in $\mathcal{O}(n^2)$; D&C counts them in $\mathcal{O}(n \log n)$ by piggy-backing on merge sort: when merging two sorted halves, every time you pull an element from the *right* half while elements still remain in the *left* half, that right element is smaller than *every one of those remaining left elements*, so it forms an inversion with each of them simultaneously — giving a bulk count in $\mathcal{O}(1)$ per right-half pick.



> **!** **D&C traps.** (1) *Forgetting the cross-boundary case* — the answer that straddles the midline (max-subarray, closest pair) is the whole point; a solution that only recurses into halves is wrong. (2) *Unbalanced splits* — quicksort/quickselect degrade to $\mathcal{O}(n^2)$ on sorted input without a randomized or median pivot. (3) *Recursion depth* — $\mathcal{O}(\log n)$ for balanced, but $\mathcal{O}(n)$ stack for degenerate splits can overflow. (4) *Re-solving overlaps* — if sub-problems *do* overlap, you want DP with memoization, not plain D&C.

4.5 Complexity profile

Balanced D&C with a linear combine is $\mathcal{O}(n \log n)$ time. Selection (quickselect) is $\mathcal{O}(n)$ average, $\mathcal{O}(n^2)$ worst. Fast exponentiation and binary search are $\mathcal{O}(\log n)$. Space is $\mathcal{O}(\log n)$ for the recursion stack plus any merge buffers.

Practice Problems

Warm-up: Merge Sort (implement it); Binary Search; Pow(x, n) by fast exponentiation; Majority Element (Boyer-Moore or D&C). **Core:** Sort an Array; Kth Largest Element (quickselect); Maximum Subarray (D&C version); Merge k Sorted Lists; Different Ways to Add Parentheses. **Stretch:** Count of Smaller Numbers After Self; Reverse Pairs; Count of Range Sum; The Skyline Problem; Closest Pair of Points.

Pattern 4 — Two Pointers & Sliding Window

tpcol view

Layer 1 — The Big Idea: Instead of nesting two loops ($\mathcal{O}(n^2)$) to examine pairs or ranges, you walk **two indices through the data in a single pass**. "Two pointers" usually means indices that start at opposite ends and *converge*; "sliding window" means a left and right index that bound a *contiguous range* which grows and shrinks. Both turn quadratic brute force into linear time by never re-examining what they have already ruled out.

Layer 2 — Going Deeper: The sliding-window mental model: `right` expands the window to include more, and whenever the window *violates* a constraint, `left` contracts it until it is valid again. You maintain a running summary of the window (a sum, a count, a hash map of character frequencies) so each expand/contract is $\mathcal{O}(1)$. The two-pointer convergence model: on a *sorted* array, if the current pair's sum is too small move `left` right (increase), if too big move `right` left (decrease) — monotonicity guarantees you never miss the answer. The unifying invariant: **each pointer only moves forward, so total work is $\mathcal{O}(n)$.**

Layer 3 — Expert Lens: Experts classify windows as **fixed-size** (window length k given — slide one in, one out) versus **variable-size** (find the smallest/largest window satisfying a *predicate* — a boolean function that says whether the window currently obeys the constraint — the "shrinkable" template). The hard variants hinge on *what makes the window valid* and *whether the validity predicate is monotonic* (i.e. once the window is valid, shrinking it can only keep it valid or make it invalid, never flip back and forth): "longest substring with at most K distinct" has this monotone shape, so the shrink-while-invalid template works. Problems where validity is not monotonic need a *monotonic deque* (a double-ended queue that maintains elements in sorted order by evicting candidates that can never be the window max/min) for sliding-window maximum, or a prefix-sum + hashmap instead. Knowing which substrate — counter, deque, or prefix sum — backs the window is the senior-level distinction.

5.1 Sub-types you will meet

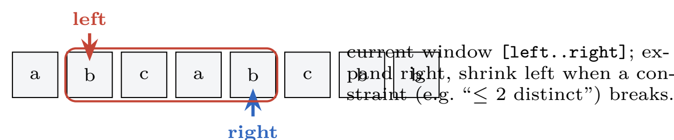
- **Converging two pointers (sorted).** Two Sum II, 3Sum, Container With Most Water, Trapping Rain Water, valid palindrome.
- **Fast/slow pointers.** Linked-list cycle detection (Floyd), find the middle, "Happy Number," duplicate number.
- **In-place array partition.** Remove Duplicates, Move Zeroes, Dutch National Flag (sort colors), partition step of quicksort.
- **Fixed-size window.** Maximum average subarray of length k; fixed-window anagram checks.
- **Variable-size window.** Longest Substring Without Repeating Characters; Minimum Size Subarray Sum; Longest Substring with At Most K Distinct; Minimum Window Substring.
- **Monotonic-deque window.** Used when you need the max or min of every window of size k in $\mathcal{O}(n)$. A *monotonic deque* (double-ended queue kept in sorted order by popping the back whenever a new element is larger/smaller) stores candidate indices so the front is always the current window's extreme. Classic problems: Sliding Window Maximum; "Shortest Subarray with Sum at Least K."

5.2 Two skeletons

```python
# ---------- CONVERGING TWO POINTERS (sorted input required) ----------
# PREMISE: arr must be sorted (or have a similar monotonic structure).
# Because arr is sorted, moving lo right strictly increases arr[lo],
# and moving hi left strictly decreases arr[hi]. This monotonicity
# means we can never miss the answer: if s < target we provably need
# a larger value from the left end; if s > target we need a smaller
# value from the right end. Each step eliminates at least one index
# for good, so the loop runs in O(n) total.
def two_pointers(arr, target):
    lo, hi = 0, len(arr) - 1          # start at opposite ends
    while lo < hi:                     # stop when pointers meet
        s = arr[lo] + arr[hi]
        if s == target: return (lo, hi)
        if s < target:  lo += 1        # sum too small → move left pointer right (larger value)
        else:           hi -= 1        # sum too large → move right pointer left (smaller value)
    return None                        # no pair found

# ---------- VARIABLE-SIZE SLIDING WINDOW ----------
def longest_valid_window(s):
    from collections import defaultdict
    # defaultdict(int) maps each character to its frequency in the window.
    # It auto-initialises missing keys to 0, so count[ch] += 1 is safe
    # even on the first occurrence.
    count = defaultdict(int)
    left = 0   # left boundary of the current window (inclusive)
    best = 0   # length of the longest valid window seen so far
    for right, ch in enumerate(s):
        count[ch] += 1                 # 1) EXPAND: pull s[right] into the window

        # 2) CONTRACT: while the window violates the constraint, evict s[left].
        # is_valid(count) is the window-validity predicate YOU define for the
        # specific problem -- it inspects `count` (the frequency map of the
        # current window) and returns True while the window obeys the constraint.
        # Example: for "at most K distinct chars", is_valid returns len(count) ≤ K
        # Example: for "no repeated character", is_valid returns all(v == 1 ...)
        while not is_valid(count):
            count[s[left]] -= 1
            # Delete the key when its count hits 0 so that len(count) accurately
            # reflects the number of DISTINCT characters in the window. Without
            # this, a 0-count char would still inflate len(count).
            if count[s[left]] == 0:
                del count[s[left]]
            left += 1                  # shrink from the left
        # 3) RECORD: right - left + 1 is the window length because both
        # indices are inclusive (s[left..right] has right-left+1 elements).
        best = max(best, right - left + 1)
    return best
```

5.3 Worked intuition — the window breathing

The window "breathes": **`right`** inhales new elements; when a rule breaks, **`left`** exhales until the rule holds again. Because neither pointer ever rewinds (each only moves forward — this is the *monotonicity* of the indices), the combined travel is at most $2n$ steps (**`right`** moves at most n times, **`left`** moves at most n times) — which is $\mathcal{O}(n)$, i.e. linear.



! **Two-pointer / window traps.** (1) *Two pointers need sorted (or otherwise monotonic) data* — applying convergence to unsorted input is wrong; sort first or switch to a hash map. (2) *Shrink condition* — decide whether you want the *longest* valid window (shrink only when invalid) or the *shortest* valid window (shrink while still valid, recording each time). (3) *Updating the window summary* on both expand and contract — forgetting to decrement a counter when `left` moves is the most common bug. (4) *Off-by-one window length:* it is `right - left + 1`. (5) Non-monotonic max/min queries need a **monotonic deque**, not a plain window.

5.4 Complexity profile

$\mathcal{O}(n)$ time for a single pass (each pointer advances at most n times); $\mathcal{O}(1)$ space for pure two-pointer, $\mathcal{O}(k)$ space for a window backed by a hash map of size k (distinct chars / window contents). Converging two pointers that require a sort are $\mathcal{O}(n \log n)$.

Practice Problems

Warm-up: Two Sum II (sorted); Valid Palindrome; Move Zeroes; Remove Duplicates from Sorted Array; Squares of a Sorted Array. **Core:** 3Sum; Container With Most Water; Linked List Cycle (fast/slow); Sort Colors; Longest Substring Without Repeating Characters; Minimum Size Subarray Sum; Permutation in String. **Stretch:** Trapping Rain Water; Minimum Window Substring; Longest Substring with At Most K Distinct; Sliding Window Maximum (deque); Subarrays with K Different Integers.

Pattern 5 — Graphs: BFS, DFS & Union-Find

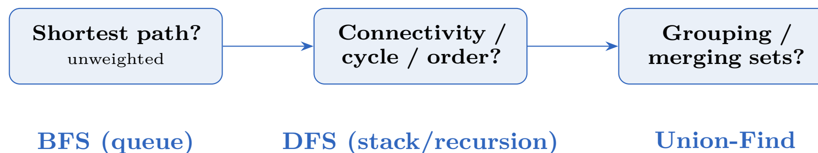
graphcol view

Layer 1 — The Big Idea: A graph is just **things (nodes) and relationships between them (edges)**. An astonishing share of interview problems are graphs in disguise: a grid is a graph where each cell connects to its neighbors; course prerequisites form a graph; a network of friends, a map of cities, a dependency tree. Once you see "entities + connections + a question about reachability, distance, ordering, or grouping," you are in graph territory, and three workhorses cover most of it: **BFS**, **DFS**, and **Union-Find**.

Layer 2 — Going Deeper: The choice among them follows the question. **BFS** explores level by level using a queue, so it finds the *shortest path in an unweighted graph* (fewest edges) and is natural for "minimum steps" problems. **DFS** dives deep using recursion or a stack, and is the tool for *connectivity, cycle detection, topological ordering, and exhaustively exploring components*. **Union-Find (Disjoint Set Union)** answers *"are these two nodes in the same group?"* and incrementally merges groups in near-constant time — ideal for connectivity that changes as edges arrive, and for Kruskal's MST. For *weighted* shortest paths you upgrade BFS to **Dijkstra** (a heap-ordered BFS); negative edges need Bellman-Ford.

Layer 3 — Expert Lens: Senior fluency is about the meta-decisions: representation (adjacency list for sparse graphs — the usual case; adjacency matrix only for dense or $O(1)$-edge-lookup needs), and *state design* when nodes are not given explicitly (in "word ladder" the nodes are words and edges are one-letter edits — you generate them on the fly). Topological sort comes in two flavors — Kahn's BFS on in-degrees, or DFS post-order — and a cycle is exactly what makes it fail, which is how you detect "is this schedule feasible?" Union-Find's two optimizations, *union by rank* and *path compression* (commonly implemented as *path halving*: each node skips to its grandparent during `find`), together give $\mathcal{O}(\alpha(n))$ amortized — effectively constant — and forgetting either one turns a clean solution into a quadratic one.

6.1 Choosing your traversal



6.2 Sub-types you will meet

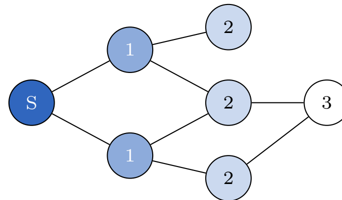
- **Grid / flood fill.** Number of Islands, Flood Fill, Rotting Oranges (multi-source BFS), Walls and Gates, Surrounded Regions.
- **Connectivity & components.** Number of Connected Components, Friend Circles, Accounts Merge, Redundant Connection (Union-Find).
- **Topological sort (DAGs).** A *topological ordering* is a linear sequence of nodes in a directed acyclic graph (DAG) such that every directed edge $u \to v$ places u before v — i.e., it respects all dependencies. Course Schedule I/II, Alien Dictionary, build-order/dependency resolution.
- **Shortest path.** Word Ladder (BFS), Network Delay Time / Cheapest Flights (Dijkstra / Bellman-Ford), Shortest Path in Binary Matrix.
- **Cycle detection.** Directed (3-color DFS) vs. undirected (Union-Find or parent-tracking DFS).
- **Bipartite / coloring.** Is Graph Bipartite, Possible Bipartition (BFS 2-coloring).
- **MST.** Kruskal (Union-Find) or Prim (heap): Min Cost to Connect All Points.

6.3 Three skeletons

```
from collections import deque
# ---------- BFS (shortest unweighted path / level order) ----------
# graph is an adjacency list: a dict (or list) mapping each node to a list of
# its neighbors, e.g. graph = {0: [1, 2], 1: [0, 3], ...}
def bfs(start, graph):
    q, seen = deque([(start, 0)]), {start}  # pre-add start so it's never re-queued
    while q:
        node, dist = q.popleft()
        if is_goal(node): return dist   # replace with your actual goal check,
                                        # e.g. node == target, or node == (R-1, C-1)
        for nxt in graph[node]:         # iterate neighbors from the adjacency list
            if nxt not in seen:
                seen.add(nxt)           # mark on ENQUEUE, not dequeue -- prevents
                q.append((nxt, dist + 1))  # the same node being queued multiple times
    return -1  # -1 signals goal was unreachable

# ---------- DFS (connectivity traversal; extend for topo sort via post-order) ----------
# graph is an adjacency list (same structure as BFS above).
def dfs(node, graph, seen):
    seen.add(node)                      # mark before recursing (pre-order)
    for nxt in graph[node]:             # visit each neighbor
        if nxt not in seen:
            dfs(nxt, graph, seen)
    # POST-ORDER HOOK: to get a topological ordering (on a DAG), uncomment:
    #   order.append(node)              # append AFTER all neighbors are processed
    # Then reverse order[] at the end. A cycle means topo sort is impossible.

# ---------- UNION-FIND (with path halving + union by rank) ----------
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))  # each node starts as its own root
        self.rank = [0] * n           # rank = upper bound on tree height;
                                      # keeps trees shallow so find() stays fast
    def find(self, x):
        # Walk up to the root, halving the chain at each step (path halving).
        # self.parent[x] = self.parent[self.parent[x]] points x to its
        # GRANDPARENT -- NOT all the way to the root (that would be full path
        # compression). Both achieve O(α(n)) amortized.
        while self.parent[x] ≠ x:
            self.parent[x] = self.parent[self.parent[x]]  # path halving
            x = self.parent[x]
        return x
    def union(self, a, b):
        ra, rb = self.find(a), self.find(b)  # find the roots of a and b
        if ra == rb: return False   # same root → already in one set; this edge
                                    # would create a cycle in an undirected graph
        # Attach the shorter tree under the taller one to keep height bounded.
        if self.rank[ra] < self.rank[rb]: ra, rb = rb, ra
        self.parent[rb] = ra                        # rb's tree goes under ra
        if self.rank[ra] == self.rank[rb]: self.rank[ra] += 1  # tie: height grows by 1
        return True  # merged two previously separate components
```

6.4 Worked intuition — BFS spreads in rings

BFS visits all nodes at distance 1, then all at distance 2, and so on — so the first time it reaches the goal, it has used the fewest edges. Multi-source BFS (Rotting Oranges) seeds the queue with *all* sources at once, and the rings expand simultaneously.



Distance rings: S(0) → {1} → {2} → {3}. First arrival = shortest path.

> ! **Graph traps.** (1) *Mark visited on enqueue*, not on dequeue, in BFS — between the time a node is enqueued and the time it is dequeued, other paths can reach it and enqueue it again; marking on enqueue prevents this, keeping complexity at $\mathcal{O}(V + E)$. (2) *Directed vs. undirected cycle detection* are different algorithms — 3-color DFS for directed, parent-tracking or DSU for undirected. (3) *Dijkstra forbids negative edges*; reaching for it on a negative-weight graph gives wrong answers — use Bellman-Ford. (4) *Recursion depth* — DFS on a 10^5-node graph can blow Python's stack; use an explicit stack. (5) *Union-Find without path compression* and *rank* is not near-constant — include both.

6.5 Complexity profile

BFS/DFS are $\mathcal{O}(V + E)$ time, $\mathcal{O}(V)$ space (adjacency list — a dict/list mapping each node to its neighbors, costing $\mathcal{O}(V + E)$ total storage). Dijkstra with a binary heap is $\mathcal{O}((V + E) \log V)$. Union-Find operations are $\mathcal{O}(\alpha(n))$ amortized — where $\alpha(n)$ is the inverse Ackermann function, a value that never exceeds 4 for any practical input, so treat it as effectively $\mathcal{O}(1)$. Topological sort (Kahn or DFS post-order) is $\mathcal{O}(V + E)$.

Practice Problems

Warm-up: Flood Fill; Number of Islands; Max Area of Island; Find if Path Exists in Graph (Union-Find). **Core:** Rotting Oranges; Course Schedule I & II; Number of Connected Components; Clone Graph; Word Ladder; Is Graph Bipartite; Pacific Atlantic Water Flow; Redundant Connection. **Stretch:** Alien Dictionary; Network Delay Time (Dijkstra); Cheapest Flights Within K Stops; Min Cost to Connect All Points (MST); Swim in Rising Water; Accounts Merge; Word Ladder II.

Pattern 6 — Backtracking

btcol view

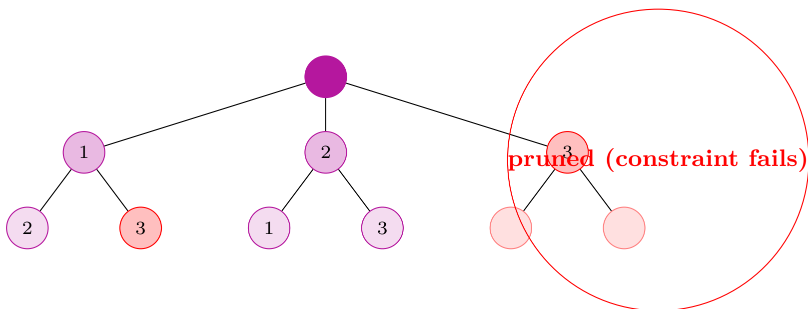
Layer 1 — The Big Idea: Backtracking is **systematic, exhaustive trial-and-error**. You build a candidate solution one decision at a time; whenever a partial choice cannot possibly lead to a valid complete answer, you *undo it (backtrack)* and try the next option. It is depth-first search over the tree of all possible decision sequences, with the power to prune whole branches. When a problem asks for *all* permutations, combinations, subsets, or arrangements satisfying constraints, backtracking is almost always the intended tool.

Layer 2 — Going Deeper: Every backtracking solution has the same anatomy: a **choose–explore–unchoose** loop. You *choose* an option (add it to the current path), *explore* by recursing deeper, then *unchoose* (remove it) to restore state before trying the next option. Two design decisions shape the search: the **branching** (what choices are available at each step — `choices` is simply the input list, e.g. `nums` or `candidates`) and the **pruning / constraint check** (*pruning* = aborting a branch the moment it cannot yield a valid answer, before fully expanding it). A `start` index prevents reusing earlier elements in combination problems; a `used[]` array prevents reusing elements in permutations. The base case appends a *copy* of the current path to the results.

Layer 3 — Expert Lens: The expert difference is **pruning power** and **duplicate handling**. Strong pruning — N-Queens checking column/diagonal conflicts before recursing, Sudoku checking row/col/box — converts a hopeless $\mathcal{O}(n!)$ search into something tractable. Handling duplicate inputs cleanly (sort first, then skip when `i > start` *and* `nums[i]==nums[i-1]` at the same tree depth — the `i > start` guard ensures the *first* occurrence at that depth is never skipped) is the single most common follow-up. And recognizing the **boundary with DP** matters: if the decision tree has overlapping sub-problems and you only need a *count* or an *optimum* (not the explicit arrangements), memoize and it becomes DP; backtracking is for when you must *enumerate* the actual solutions or when the state is too rich to memoize.

7.1 The decision tree

Backtracking explores a tree where each level fixes one more decision. Pruned branches (red) are abandoned the moment a constraint is violated, so they are never fully expanded.



7.2 Sub-types you will meet

- **Subsets / power set** (*power set* = the collection of *all* 2^n subsets of n elements)**.** Include-or-exclude each element. Subsets, Subsets II (with duplicates).
- **Combinations.** Choose k of n, or all combos summing to a target. Combinations, Combination Sum I/II/III.
- **Permutations.** All orderings; with/without duplicates. Permutations, Permutations II.
- **Partitioning.** Split a string/array under a predicate. Palindrome Partitioning, Restore IP Addresses.
- **Grid / constraint search.** Word Search, N-Queens, Sudoku Solver, generate parentheses.
- **Combinatorial generation.** Letter Combinations of a Phone Number, Generate Parentheses.

7.3 The universal skeleton

```
# choices: the INPUT list (e.g. nums, candidates) passed in or captured from scope.
# start:   the index in choices from which this call may pick; prevents reusing
#          elements that were already committed in earlier positions.
def backtrack(start, path, results, choices):
    # is_complete(path): True when path represents a finished answer.
    # Common implementations: len(path) == k, sum(path) == target, etc.
    if is_complete(path):
        results.append(path[:])          # path[:] = COPY; without this, every
        return                           # result points at the same live list.

    for i in range(start, len(choices)):
        # is_valid(choices[i], path): False when adding this element would break
        # a constraint (e.g. exceed the target sum, violate a board rule).
        # Checking HERE (before recursing) is what makes pruning effective.
        if not is_valid(choices[i], path):
            continue
        # Skip duplicates at the SAME tree depth (only valid after sorting inputs).
        # i > start ensures we do NOT skip the very first pick at this depth.
        if i > start and choices[i] == choices[i - 1]:
            continue
        path.append(choices[i])                  # 1) CHOOSE
        # Pass i+1 to disallow reuse of choices[i] (combinations, subsets).
        # Pass i   to allow reuse of choices[i] (e.g. Combination Sum I).
        # For permutations, replace start/i with a boolean used[] array instead.
        backtrack(i + 1, path, results, choices)  # 2) EXPLORE
        path.pop()                               # 3) UNCHOOSE -- restore state
    return results
```

! **Backtracking traps.** (1) *Appending the live list* instead of a copy — every result ends up pointing at the same (eventually empty) list; always `path[:]` or `list(path)`. (2) *Forgetting to undo* state (`path.pop()`, unmark `used[i]`, restore the board) corrupts sibling branches. (3) *Duplicate results* — sort inputs and skip same-value siblings at the same depth. (4) `start` *vs.* `i+1` *vs.* `i`: use `i+1` when each element is used at most once, `i` when reuse is allowed (Combination Sum), and a `used[]` array for permutations. (5) *Weak pruning* — push the constraint check as early as possible to avoid exploring doomed subtrees.

7.4 Complexity profile

Backtracking is inherently exponential because it enumerates: subsets $\mathcal{O}(2^n)$, permutations $\mathcal{O}(n \cdot n!)$, combinations $\mathcal{O}(C(n,k) \cdot k)$. Pruning lowers the constant dramatically but not the worst-case class. Space is $\mathcal{O}(n)$ for the recursion depth / current path (excluding the output list). This is why backtracking is acceptable only for *small* n — revisit the complexity ladder: $n \leq 12$ish.

Practice Problems

Warm-up: Subsets; Combinations; Letter Combinations of a Phone Number; Generate Parentheses. **Core:** Permutations; Permutations II; Subsets II; Combination Sum I/II/III; Word Search; Palindrome Partitioning. **Stretch:** N-Queens; Sudoku Solver; Restore IP Addresses; Word Search II (with a Trie); Expression Add Operators; Partition to K Equal Sum Subsets.

Pattern 7 — Binary Search, Heaps & Tree Structures

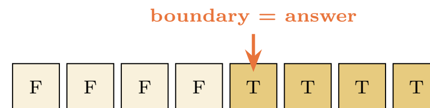
bscol view

Layer 1 — The Big Idea: This family is unified by one idea: **exploit structure to avoid looking at everything**. Binary search halves a *monotonic* (one-directional: candidate answers only ever go from "no" to "yes") search space each step ($\mathcal{O}(\log n)$). A heap keeps the single best element instantly available so you never sort the whole collection ($\mathcal{O}(\log n)$ push/pop). A balanced BST / tree keeps data ordered for fast range and successor queries. When the data is sorted, or the *answer* is monotonic, or you only ever need the current min/max, these structures turn linear or quadratic work logarithmic.

Layer 2 — Going Deeper: **Binary search** has two faces. The obvious one searches a sorted array for a value. The powerful one is **"binary search on the answer"**: when you can write a boolean *predicate* (a yes/no function) `feasible(x)` that is *monotonic* — it flips from `False` to `True` exactly once across the candidate range and never flips back — you binary-search the *answer space* for that **lower bound** (the first `True`) — "minimum capacity to ship in D days," "smallest divisor," "minimize the maximum." **Heaps** (priority queues) excel at "top K," "K-th largest," "merge K sorted lists," and the **two-heaps** trick for a running median (a max-heap for the lower half, a min-heap for the upper). **Trees** cover BST operations, in-order traversal (which yields sorted order), and tree-DP.

Layer 3 — Expert Lens: The senior craft in binary search is **getting the boundary conditions provably right**: pick `lo<hi` vs. `lo<=hi`, decide whether you return `lo` or `lo-1`, and use the "find first true" **lower-bound** template (*lower bound* = the leftmost index at which the predicate becomes `True`) to avoid off-by-one infinite loops. The mid computation `lo+(hi-lo)//2` avoids integer overflow in fixed-width languages like Java/C++ where `lo+hi` can exceed `INT_MAX`; Python integers are unbounded but the form is canonical. For heaps, knowing that a **bounded heap of size K** gives $\mathcal{O}(n \log k)$ (better than $\mathcal{O}(n \log n)$ full sort) for top-K is the standard optimization, and **quickselect** beats both at $\mathcal{O}(n)$ average when you need an unordered top-K. Python's `heapq` provides *only* a min-heap; to get a **max-heap via negation**, insert `-x` instead of `x` — then `heap[0]` is the most-negative stored value, and `-heap[0]` recovers the true maximum. For trees, the insight that **in-order traversal of a BST is sorted** unlocks a whole class of validation and k-th-element problems.

8.1 Binary search on the answer — the mental shift



`feasible(x)`: a boolean predicate (yes/no function) that is *monotonic* (all `F`s before all `T`s). Binary search finds the **lower bound**: the leftmost `T`.

8.2 Sub-types you will meet

- **Classic binary search.** Search Insert Position, Find First/Last Position, Search in Rotated Sorted Array, Find Minimum in Rotated Array.
- **Binary search on answer.** Koko Eating Bananas, Capacity to Ship Packages in D Days, Split Array Largest Sum, Minimize Max Distance, Median of Two Sorted Arrays.
- **Top-K / K-th with a heap.** Kth Largest Element, Top K Frequent Elements, K Closest Points to Origin, Kth Smallest in a Sorted Matrix.
- **Merge / streaming heaps.** Merge K Sorted Lists, Smallest Range Covering K Lists, Task Scheduler.
- **Two heaps.** Find Median from Data Stream, Sliding Window Median, IPO.
- **BST / tree.** Validate BST, Kth Smallest in a BST, Lowest Common Ancestor, Insert/Delete in a BST, range-sum.

8.3 Three skeletons

```
# ---- LOWER-BOUND BINARY SEARCH: find the first index where feasible(x) is True ----
# Half-open invariant [lo, hi): lo is inclusive (could be the answer),
#                               hi is exclusive (proven not the answer yet).
# The loop shrinks the interval until lo == hi, which is the answer.
def binary_search(lo, hi, feasible):
    # feasible(x): a monotonic boolean predicate -- False for too-small x, True once x
    # is large enough. Example: feasible(cap) = "can we ship all packages in D days
    # at this capacity?" -- False for small cap, True for large cap, flips once.
    while lo < hi:                          # stop when interval collapses to a single point
        mid = lo + (hi - lo) // 2           # safe midpoint: avoids (lo+hi) integer overflow
        if feasible(mid): hi = mid          # mid works → answer is mid or to the LEFT
        else:             lo = mid + 1      # mid fails  → answer must be strictly to the RIGHT
    return lo                               # lo == hi: the first index where feasible is True

# ---- TOP-K WITH A BOUNDED MIN-HEAP : O(n log k) ----
# Counter-intuitive key: to keep the K LARGEST, use a MIN-heap of size K.
# The heap root is always the WEAKEST candidate (the smallest of the K).
# When a stronger element arrives, it evicts the weakest via heappop.
import heapq
def top_k_largest(nums, k):
    heap = []                               # Python heapq is a min-heap by default
    for x in nums:
        heapq.heappush(heap, x)             # add the new element (O(log k))
        if len(heap) > k:                   # heap exceeds K → one too many candidates
            heapq.heappop(heap)             # evict the current minimum (weakest)
    return heap                             # the K largest values (unordered)

# ---- TWO HEAPS: running median ----
# Split the stream into a lower half and an upper half at all times:
#   lo = MAX-heap of the lower half  (Python has no max-heap, so store NEGATED values)
#   hi = MIN-heap of the upper half  (plain heapq)
# Cross-invariant: every value in lo ≤ every value in hi.
# Size rule: hi holds ≥ lo's count; hi gets the "extra" element on odd totals.
class MedianFinder:
    def __init__(self):
        self.lo, self.hi = [], []           # lo: max-heap via negation; hi: plain min-heap
    def add(self, num):
        # Step 1 -- route num through hi to maintain the cross-heap ordering invariant.
        # heappushpop(hi, num): pushes num into hi, then pops and returns hi's minimum.
        # That minimum belongs in the lower half, so push it (negated) into lo.
        heapq.heappush(self.lo, -heapq.heappushpop(self.hi, num))
        # Step 2 -- rebalance so hi stays ≥ lo in count.
        # heappop(self.lo) pops lo's most-negative value = the MAXIMUM of the lower
        # half; negating it back gives the true max, which we move up into hi.
        if len(self.lo) > len(self.hi):
            heapq.heappush(self.hi, -heapq.heappop(self.lo))
    def median(self):
        # Odd total: hi has one more element → hi[0] is the exact middle.
        if len(self.hi) > len(self.lo):
            return self.hi[0]
        # Even total: average the min of the upper half (hi[0]) and the max of the
        # lower half. lo[0] is NEGATIVE (negation trick), so max_lo = -self.lo[0],
        # and (hi[0] + (-lo[0]))/2 simplifies to (self.hi[0] - self.lo[0]) / 2.
        return (self.hi[0] - self.lo[0]) / 2
```

! **Binary-search / heap traps.** (1) *Loop termination* — the **half-open** template (`lo < hi`, `hi=mid`, `lo=mid+1`) and the **closed** template (`lo <= hi`, `hi=mid-1`, `lo=mid+1`) are each valid in isolation; mixing them (e.g. `lo < hi` with `hi=mid-1`) creates infinite loops. Pick one form and never mix. (2) *Predicate monotonicity* — binary-search-on-answer is only valid if `feasible`

! flips *at most once* across the search space; hand-verify it before committing. (3) *Rotated-array* mid logic must first determine *which half is sorted* before deciding which way to move the boundary. (4) *Python's heapq is min-only* — to simulate a max-heap, push `-x` and read the max as `-heap[0]`; forgetting the negation when reading back is a common bug. (5) *Top-K heap direction* is counter-intuitive: to keep the k *largest*, use a *min*-heap of size k (so the root is the weakest kept candidate and gets evicted when a better one arrives). (6) *Two-heaps median formula*: `(hi[0] - lo[0])/2` looks like a subtraction but is an addition in disguise — `lo[0]` is the negated maximum of the lower half, so subtracting it adds the true max. (7) *Validate BST* with propagated value bounds (`min_val < node.val < max_val`), not just parent comparison.

8.4 Complexity profile

Binary search $\mathcal{O}(\log n)$; binary-search-on-answer $\mathcal{O}(n \log(\text{range}))$. Heap push/pop $\mathcal{O}(\log n)$; building a heap $\mathcal{O}(n)$; top-K $\mathcal{O}(n \log k)$; quickselect $\mathcal{O}(n)$ average. Balanced-BST operations $\mathcal{O}(\log n)$; in-order traversal $\mathcal{O}(n)$.

Practice Problems

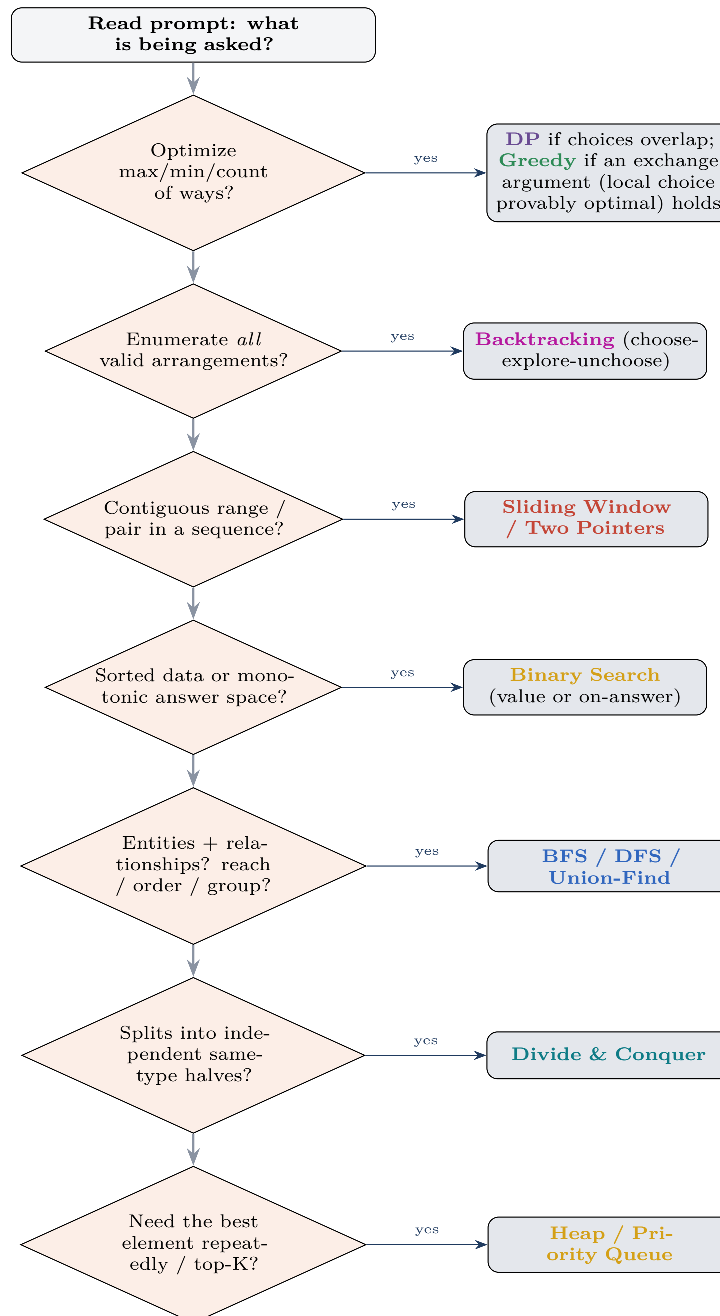
Warm-up: Binary Search; Search Insert Position; First Bad Version; Kth Largest Element in a Stream; Last Stone Weight (heap). **Core:** Find First and Last Position; Search in Rotated Sorted Array; Find Minimum in Rotated Sorted Array; Koko Eating Bananas; Top K Frequent Elements; K Closest Points to Origin; Kth Smallest Element in a BST; Validate BST. **Stretch:** Median of Two Sorted Arrays; Split Array Largest Sum; Find Median from Data Stream (two heaps); Merge k Sorted Lists; Smallest Range Covering Elements from K Lists; Kth Smallest in a Sorted Matrix.

Meta-Strategy: A Unified Decision Process

Knowing seven patterns is necessary but not sufficient — under interview pressure you need a **routing procedure** that maps an unseen prompt to the right pattern in the first two minutes, plus a **communication protocol** that demonstrates senior judgment regardless of which pattern you land on. This section ties the families together.

9.1 The master decision flowchart

Walk this top to bottom. At each diamond, a **yes** answer routes right to the pattern; **no** (or "not clearly") falls through to the next question. The first branch that fits is usually right; if two fit, the constraint size (from the complexity ladder) breaks the tie.



9.2 How the families relate

The patterns are not silos — they shade into one another, and naming the boundary is a senior signal.

Boundary	The distinction to articulate
Greedy vs. DP	Both optimize. Greedy commits to a local choice and never revisits (needs an exchange-argument proof); DP keeps all sub-results because choices interact. If greedy has a counterexample, it is DP.
DP vs. Backtracking	Both explore a decision tree. If sub-problems overlap and you need a count/optimum, memoize → DP. If you must list the actual arrangements, or the state is too rich to cache (e.g. it contains a mutable board or unbounded variables), enumerate → backtracking.
DP vs. Divide & Conquer	Both recurse on sub-problems. D&C sub-problems are *independent* (merge sort); DP sub-problems *overlap* (Fibonacci) and must be cached.
BFS vs. Dijkstra	Same skeleton; BFS for unweighted (FIFO queue), Dijkstra for weighted (min-heap). Dijkstra gives *incorrect results* on negative-weight edges (it does not detect the error) — use Bellman-Ford instead.
Two Pointers vs. Binary Search	Both exploit sortedness (when the input is sorted). Two pointers sweep linearly inward from both ends in one pass $\mathcal{O}(n)$; binary search jumps to the midpoint of a monotonic predicate space $\mathcal{O}(\log n)$.
Heap vs. Sorting	A full sort is $\mathcal{O}(n \log n)$; a size-K heap gives top-K in $\mathcal{O}(n \log k)$ and works on streams where you cannot sort everything.

9.3 The interview communication protocol

Pattern recognition wins the algorithm; communication wins the *offer*. Senior interviewers grade your process. Narrate these six beats:

1. **Clarify & restate.** Confirm input ranges, edge cases (empty, single element, duplicates, negatives), and the exact output. Input sizes drive complexity targets.
2. **Brute force first.** State the naive solution and its complexity out loud — it anchors the conversation and shows you understand the problem before optimizing.
3. **Identify the pattern.** "This is an optimization with overlapping sub-problems, so I'll use DP" — name it and justify the trigger.
4. **Design before coding.** Define the state / invariant / pointers, the transition, base cases, and the target complexity *before* typing. Get a nod.
5. **Code cleanly, narrate.** Write the skeleton, talk through each line, keep names meaningful.
6. **Test & analyze.** Dry-run a small example and the edge cases, then state final time/space complexity and any space optimization.

> ! **The most common senior-loop failure is not a wrong answer — it is silence.** Candidates who think correctly but quietly get dinged for "communication." Even when stuck, narrate your hypotheses and what you would try next. A working brute force you can explain beats a half-finished optimal solution you cannot.

Appendix: One-Page Pattern Cheat-Sheet

Photograph this page. It is the entire guide compressed into a single triage table — trigger, the one-line technique, the reusable skeleton's shape, and the complexity you should state aloud. In the interview, run your prompt down the *Trigger* column first.

Pattern	Trigger you feel	Technique / skeleton	Typical complexity
Greedy	"max/min" and a local choice seems safe; sortable by a key	Sort by the right key, sweep once keeping one running invariant; justify with an exchange argument	$\mathcal{O}(n \log n)$
Dynamic Programming	"how many ways," "max/min" over interacting choices; overlapping sub-problems	Define state → transition → base case → memoize or tabulate → optimize space	$\mathcal{O}(\text{states} \times \text{transition})$
Divide & Conquer	Splits into independent same-type halves; "count across," "sort"	Divide, conquer halves, combine in a clever $\mathcal{O}(n)$ merge; read off via Master Theorem	$\mathcal{O}(n \log n)$
Two Pointers / Window	Contiguous subarray/substring; "at most K"; sorted pair	Expand right, shrink left on violation; or converge two ends on sorted data	$\mathcal{O}(n)$
Graphs	Entities + relationships; reach / order / group / grid	BFS (shortest unweighted), DFS (connectivity/topo/cycle), Union-Find (grouping)	$\mathcal{O}(V + E)$
Backtracking	"all" permutations / combinations / subsets; place-N under constraints	choose → explore → unchoose; prune early; append a copy	$\mathcal{O}(2^n)$ subsets; $\mathcal{O}(n{\cdot}n!)$ perms
Binary Search / Heap	Sorted data or monotonic answer; "top-K," "K-th," running median	Lower-bound search on a monotonic predicate; size-K heap; two heaps for median	$\mathcal{O}(\log n)$, $\mathcal{O}(n \log k)$

Constraint-to-complexity map (work backward from n): $n \leq 12 \Rightarrow \mathcal{O}(n!)$ backtracking; $n \leq 20 \Rightarrow \mathcal{O}(2^n)$ bitmask DP; $n \leq 2000 \Rightarrow \mathcal{O}(n^2)$ DP; $n \leq 10^5 \Rightarrow \mathcal{O}(n \log n)$; $n \geq 10^6 \Rightarrow \mathcal{O}(n)$ or $\mathcal{O}(\log n)$.

Pattern-boundary one-liners: Greedy vs. DP — *can you prove an exchange argument?* DP vs. D&C — *do sub-problems overlap?* DP vs. Backtracking — *do you need a count/optimum or the actual list?* BFS vs. Dijkstra — *are edges weighted?* Two Pointers vs. Binary Search — *linear sweep or jump to the middle?*

Final Examination — 30 Questions

Instructions. Choose the single best answer. The goal is *pattern recognition*, so for each prompt ask "which trigger do I feel?" before reading the options. Answers and explanations follow on the last page — cover them until you are done. Aim for 24/30 (80%) before moving to live coding practice.

Q1. You must find the *minimum number of coins* to make an amount with coins $\{1, 5, 6, 9\}$. Which approach is provably correct?
(A) Greedy: always take the largest coin $\leq$ remaining.
(B) Dynamic programming over amounts.
(C) Divide and conquer on the amount.
(D) Backtracking with no memoization.

Q2. "Find the length of the *longest substring* with at most K distinct characters." Best pattern?
(A) Binary search.
(B) Variable-size sliding window with a hash map.
(C) Backtracking.
(D) Union-Find.

Q3. For a recurrence $T(n) = 2T(n/2) + \Theta(n)$, the Master Theorem gives:
(A) $\Theta(n)$.
(B) $\Theta(n^2)$.
(C) $\Theta(n \log n)$.
(D) $\Theta(\log n)$.

Q4. You need the *shortest path* in an *unweighted* graph. The right tool is:
(A) DFS.
(B) BFS.
(C) Dijkstra.
(D) Bellman-Ford.

Q5. "Return *all* subsets of a set of distinct integers." Pattern?
(A) Dynamic programming.
(B) Greedy.
(C) Backtracking.
(D) Sliding window.

Q6. Which property must hold for "binary search on the answer" to be valid?
(A) The array is already sorted.
(B) The feasibility predicate is monotonic in the candidate answer.
(C) The input contains no duplicates.
(D) The answer is unique.

Q7. The key distinction between Divide & Conquer and Dynamic Programming is:
(A) D&C uses recursion, DP never does.
(B) DP sub-problems overlap and are cached; D&C sub-problems are independent.
(C) D&C is always faster.
(D) DP cannot be written iteratively.

Q8. To keep the K *largest* elements of a stream efficiently, use:
(A) A max-heap of all elements.
(B) A min-heap of size K.
(C) A sorted list, re-sorted each insert.
(D) A stack.

Q9. "Detect a cycle in a *directed* graph." The cleanest method is:
(A) Union-Find.
(B) Three-color (white/gray/black) DFS.
(C) BFS level counting.
(D) Two pointers.

Q10. For 0/1 knapsack with a 1-D DP array, you must iterate the capacity dimension:
(A) Ascending.
(B) Descending.
(C) Order does not matter.
(D) In random order.

Q11. "Two Sum" on a *sorted* array, $O(1)$ extra space:
(A) Hash map.

(B) Converging two pointers.
(C) Nested loops.
(D) Heap.

Q12. Greedy is guaranteed optimal when you can construct:
(A) A memoization table.
(B) An exchange (swap) argument.
(C) A topological order.
(D) A monotonic predicate.

Q13. "Course Schedule: can all courses be finished given prerequisites?" is fundamentally:
(A) Topological sort / cycle detection on a DAG.
(B) Sliding window.
(C) Binary search.
(D) Knapsack DP.

Q14. The maximum-subarray-crossing-the-midpoint case is essential in which approach to Maximum Subarray?
(A) Kadane's DP.
(B) Divide and conquer.
(C) Sliding window.
(D) Greedy.

Q15. In backtracking, appending `path` instead of `path[:]` to results causes:
(A) A stack overflow.
(B) All results referencing the same mutated list.
(C) Duplicate pruning to fail.
(D) Slower asymptotic time.

Q16. A running median over a stream is best maintained with:
(A) One sorted array.
(B) Two heaps (a max-heap and a min-heap).
(C) A single min-heap.
(D) Union-Find.

Q17. "Minimum capacity to ship all packages within D days" is a classic:
(A) Greedy interval problem.
(B) Binary search on the answer.
(C) Graph shortest path.
(D) Interval DP.

Q18. In BFS you should mark a node visited:
(A) When you dequeue it.
(B) When you enqueue it.
(C) After processing all neighbors.
(D) Never.

Q19. Which constraint size most strongly suggests an $O(2^n \cdot n)$ *bitmask DP* is intended?
(A) $n \leq 10^6$.
(B) $n \leq 10^5$.
(C) $n \leq 2000$.
(D) $n \leq 20$.

Q20. Union-Find achieves near-constant amortized time only with:
(A) Path compression alone.
(B) Union by rank alone.
(C) Both path compression and union by rank.
(D) Neither; it is always $O(\log n)$.

Q21. "Longest Increasing Subsequence" in $O(n \log n)$ combines DP intuition with:
(A) A binary search into a tails array.
(B) Union-Find.
(C) A sliding window.
(D) Backtracking.

Q22. Quickselect finds the K-th smallest element in average time:
(A) $O(\log n)$.
(B) $O(n)$.
(C) $O(n \log n)$.
(D) $O(n^2)$.

Q23. "Generate all valid combinations of N pairs of parentheses" is:
(A) Greedy.
(B) Backtracking with a validity constraint.
(C) Binary search.
(D) BFS.

Q24. The danger of using Dijkstra on a graph with negative edge weights is:
(A) Infinite loop.
(B) It may finalize a node's distance before finding a shorter negative path.
(C) Stack overflow.
(D) Nothing; it is fine.

Q25. For "number of distinct ways to climb stairs taking 1 or 2 steps," the state dimension is:
(A) 1-D ($dp[i]$).
(B) 2-D ($dp[i][j]$).
(C) Interval ($dp[i][j]$ range).
(D) Bitmask.

Q26. "Minimum Window Substring" (smallest window containing all chars of T) uses which window variant?
(A) Fixed-size window.
(B) Variable window that shrinks *while still valid* to minimize.
(C) Two converging pointers on sorted input.
(D) Monotonic deque.

Q27. Sliding Window Maximum (max of every window of size k) in $O(n)$ requires:
(A) A monotonic decreasing deque.
(B) A max-heap with lazy deletion only.
(C) Re-scanning each window.
(D) Binary search.

Q28. In-order traversal of a Binary Search Tree yields nodes in:
(A) Sorted ascending order.
(B) Level order.
(C) Reverse insertion order.
(D) Random order.

Q29. "Counting inversions" in $O(n \log n)$ piggybacks on which algorithm's combine step?
(A) Quickselect.
(B) Merge sort.
(C) Heap sort.
(D) Counting sort.

Q30. You are told $n \leq 10^5$ and need an exact answer. Which target complexity is the safest design goal?
(A) $O(n^2)$.
(B) $O(2^n)$.
(C) $O(n \log n)$ or better.
(D) $O(n!)$.

Answer Key & Explanations

1. B. {1,5,6,9} is non-canonical (e.g. amount 11: greedy 9+1+1=3 coins, optimal 5+6=2). Greedy fails; DP over amounts is correct.

2. B. A contiguous substring under a "$\leq$ K" constraint is the textbook variable sliding window backed by a frequency map.

3. C. $n^{\log_b a} = n^{\log_2 2} = n$, and $f(n) = \Theta(n) \Rightarrow$ Case 2 $\Rightarrow \Theta(n \log n)$ (merge sort).

4. B. BFS explores in distance rings, so first arrival = fewest edges. Dijkstra is overkill (and equal to BFS) when all weights are 1.

5. C. Enumerating *all* subsets is choose/exclude backtracking (or bitmask), $\mathcal{O}(2^n)$.

6. B. Binary-search-on-answer needs the predicate to flip once (monotonic); the array itself need not be sorted.

7. B. Overlap + caching is the DP signature; D&C halves are independent.

8. B. A size-K *min*-heap keeps the K largest in $\mathcal{O}(n \log k)$; the root is the smallest of the K and is evicted.

9. B. Three-color DFS detects a back edge (a gray node reaches another gray ancestor) = directed cycle. Union-Find is for undirected graphs.

10. B. Descending capacity prevents reusing an item twice within one row (that would be unbounded knapsack).

11. B. Sorted + $O(1)$ space = converging two pointers; a hash map costs $O(n)$ space.

12. B. The exchange/swap argument is the proof technique that licenses greedy.

13. A. Feasibility = the prerequisite directed graph has no cycle; topological sort (Kahn's or DFS post-order) succeeds iff no cycle exists.

14. B. The crossing-the-midpoint case is unique to the D&C formulation; Kadane is linear DP.

15. B. `path` is mutated in place; appending the reference makes all results alias it. Append a copy.

16. B. Two heaps split the stream into a lower half (max-heap) and upper half (min-heap); the median sits at the tops.

17. B. Monotonic feasibility ("can we ship within D days at capacity x?") $\Rightarrow$ binary search on x.

18. B. Marking on enqueue prevents the same node being queued multiple times.

19. D. 2^n is only tractable for tiny n; $n \leq 20$ is the bitmask-DP tell.

20. C. Both optimizations together give $O(\alpha(n))$; either alone is worse.

21. A. Patience sorting binary-searches the position in the "tails" array $\Rightarrow O(n \log n)$.

22. B. Quickselect partitions and recurses into one side: $O(n)$ average (worst $O(n^2)$).

23. B. Build the string with constraints (open$\leq$N, close<open) and backtrack.

24. B. Dijkstra greedily finalizes the min-distance node; a later negative edge could have made a finalized node cheaper $\Rightarrow$ wrong. Use Bellman-Ford.

25. A. $dp[i] = dp[i-1] + dp[i-2]$ — a single index, 1-D (it is Fibonacci).

26. B. To *minimize* a valid window, shrink from the left while it stays valid, recording the size each time.

27. A. A monotonic decreasing deque keeps candidates so each element is pushed/popped once $\Rightarrow O(n)$.

28. A. In-order (left, node, right) of a BST is sorted ascending — the basis for Validate BST and Kth Smallest.

29. B. Inversions are counted during merge sort's merge: a right-half pull counts all remaining left elements.

30. C. $n \leq 10^5$ rules out $O(n^2)$; aim for $O(n \log n)$ or $O(n)$.

Score 27–30: interview-ready recognition. **22–26:** solid; drill the missed families. **<22:** re-read those sections' Layer 2 and redo the practice sets.

End of guide — now go solve the practice lists by hand. Recognition is a muscle; reps build it.