GROKKING THE CODING INTERVIEW: PATTERNS FOR CODING QUESTIONS

DSA Patterns Cheat Sheet: All 41 Coding Interview Patterns

Revision

DSA Patterns Cheat Sheet: All 41 Coding Interview Patterns

Every DSA pattern you need for coding interviews, from two pointers to topological sort, with signals and example problems.

What are DSA patterns?

DSA patterns are reusable templates for solving data structures and algorithms questions. Each pattern joins a recognizable problem shape to the technique that solves it. A sorted array and a target sum call for two pointers. A contiguous subarray calls for a sliding window. A list of prerequisites calls for topological sort.

Interviewers draw from a small set of these shapes, so learning the patterns is far more efficient than memorizing individual problems. Once you recognize the shape, the solution follows. This cheat sheet covers all 41 DSA patterns taught in *Grokking the Coding Interview*, and it is the document to review the night before an interview.

Every pattern below links straight to its lesson in Grokking the Coding Interview, where it is taught with guided problems and solutions in six languages.

All 41 DSA patterns at a glance

Find the clue that matches the question in front of you, then read that pattern's card below for the mechanism, the complexity, and the common mistakes.

#	Pattern	When to use it	Example problem
1	Two Pointers	Sorted input, and you need a pair, triplet, or subarray that meets a condition	Pair with Target Sum
2	Fast and Slow Pointers	A linked list question about a cycle, the middle node, or a palindrome	Linked List Cycle
3	Sliding Window	A contiguous subarray or substring that meets a size or content condition	Longest Substring Without Repeating Characters
4	Merge Intervals	A list of intervals to merge, intersect, insert into, or check for conflicts	Merge Intervals
5	Cyclic Sort	Integers in a contiguous range such as 1 to n, with a value missing or duplicated	Find the Missing Number
6	In-place Reversal of a Linked List	Reverse a list, a sublist, or groups of K nodes without extra memory	Reverse a Sub-list
7	Stacks	Matching, nesting, undo history, or anything phrased as the most recent	Balanced Parentheses
8	Monotonic Stack	The next or previous greater or smaller element for every position	Next Greater Element
9	Hash Maps	Unsorted input, asking whether a value was seen before, how often, or how to group it	First Non-repeating Character
10	Tree Level Order Traversal	The answer is per level, or the wording says shortest, minimum depth, or nearest	Binary Tree Level Order Traversal
11	Tree Depth First Search	Path sums, root-to-leaf paths, diameter, or ancestor relationships	All Paths for a Sum
12	Graphs	Entities and relationships, asking about reachability, connectivity, or fewest steps	Find if Path Exists in Graph
13	Island (Matrix Traversal)	A 2D grid of connected cells, where you count, measure, or paint regions	Number of Islands
14	Two Heaps	A stream of numbers that needs a running median, or its two middle halves	Find the Median of a Number Stream
15	Subsets	Generate all subsets, permutations, or combinations over a small input	Subsets With Duplicates
16	Modified Binary Search	Sorted, rotated, or monotonic input, and the constraints hint at log n	Minimum in Rotated Sorted Array

#	Pattern	When to use it	Example problem
17	Bitwise XOR	A single unpaired number, a complement, or a bit trick in constant space	Single Number
18	Top K Elements	The words top K, Kth largest, or K most frequent	Top K Frequent Elements
19	K-way Merge	Several sorted lists or streams, merged or queried for the Kth element across them	Merge K Sorted Lists
20	Greedy Algorithms	One clear ordering, by end time, size, or ratio, settles every choice	Maximum Length of Pair Chain
21	0/1 Knapsack (Dynamic Programming)	Items with weights and values under a capacity, or pick a subset summing to X	Equal Subset Sum Partition
22	Fibonacci Numbers (Dynamic Programming)	A recurrence of the form f(n) = f(n-1) + f(n-2), or counting ways to reach step n	Staircase
23	Palindromic Subsequence (Dynamic Programming)	Palindromes, or two strings compared, with the choice made at both ends of a range	Longest Palindromic Subsequence
24	Backtracking	All valid configurations, paths, or placements under strict constraints	N-Queens
25	Trie	A set of words queried by prefix: autocomplete, dictionary lookup, or starts with	Implement Trie (Prefix Tree)
26	Topological Sort (Graph)	Prerequisites, dependencies, or a valid build order over a directed graph	Course Schedule II
27	Union Find	Connected components, dynamic grouping, or are these two in the same group	Number of Provinces
28	Ordered Set	The closest value above or below a target, while the collection keeps changing	My Calendar I
29	Prefix Sum	Many range sums over an array that never changes, or counting subarrays by sum	Subarray Sum Equals K
30	Multi-threaded	The question names threads, ordering across threads, or producer and consumer	Print in Order
31	Counting	Frequency, occurrences, anagrams, or a majority element over a bounded value range	Group Anagrams

#	Pattern	When to use it	Example problem
32	Monotonic Queue	A sliding window that needs its maximum or minimum at every step	Sliding Window Maximum
33	Simulation	A process to play out step by step, with constraints small enough to allow it	Game of Life
34	Linear Sorting Algorithms	Integer keys in a small known range, such as 0 to 1000 or lowercase letters	Sort Colors (Dutch National Flag)
35	Meet in the Middle	A subset-sum search where n is near 30 to 40, too large for full brute force	Closest Subsequence Sum
36	Mo's Algorithm	Many offline range queries on a static array, answered after reordering them	XOR Queries of a Subarray
37	Serialize and Deserialize	Encode a tree or graph as a string, then rebuild it from that form	Serialize and Deserialize Binary Tree
38	Clone	A deep copy of a graph, a list with random pointers, or a tree with cross-links	Clone Graph
39	Articulation Point and Bridge	Connectivity under removal: critical routers or single points of failure	Critical Connections in a Network
40	Segment Tree	Range queries mixed with updates to single elements or to whole ranges	Range Sum Query - Mutable
41	Binary Indexed Tree (Fenwick Tree)	Prefix sums with point updates, or counting smaller elements to the right	Count of Smaller Numbers After Self

How to read each pattern card

Every one of the 41 cards below follows the same structure, so you can scan quickly and find the cue you need.

- **Core idea.** What the pattern is in one or two sentences.
- **Recognize it when.** The trigger words or problem shapes that signal this pattern.
- **How it works.** The mechanism in plain English.
- **Typical complexity.** Time and space cost when applied correctly.
- **Variations.** Common flavors of the same pattern.
- **Example problems.** Representative problems to anchor the pattern.
- **Watch out for.** The most common mistakes.

1. Pattern: Two Pointers

Study this pattern in Grokking the Coding Interview

Core idea. Use two indices that walk through the same array or string, either toward each other from the ends or together in the same direction. They replace a nested loop with a single pass, dropping the cost from $O(n^2)$ to $O(n)$.

Recognize it when. The input is sorted (or can be sorted), and you are looking for a pair, triplet, or subarray that satisfies a condition like a target sum, a comparison, or a duplicate-removal rule. The brute force would be two nested loops.

How it works. Place one pointer at the start and one at the end. Compare the values they point to. If the result is too small, move the left pointer right to increase it. If too large, move the right pointer left to decrease it. Stop when the pointers cross. The sortedness of the array is what makes this directional reasoning correct.

Typical complexity. Time: O(n). Space: O(1).

Variations. Opposite-end pointers for pair-sum problems. Same-direction pointers for remove-duplicates and partition problems. Three pointers for 3Sum where one index is fixed and the other two converge.

Example problems. Pair with Target Sum, Remove Duplicates, Squaring a Sorted Array, 3Sum, Container With Most Water, Trapping Rain Water.

Watch out for. The array must be sorted for opposite-end Two Pointers to work. For unsorted inputs you usually need a hash map instead. In 3Sum, remember to skip over duplicate values explicitly to avoid duplicate triplets in the answer.

2. Pattern: Fast and Slow Pointers

Study this pattern in Grokking the Coding Interview

Core idea. Two pointers walk the same data structure at different speeds. The faster one outpaces the slower one, and the relationship between their positions reveals structural facts like cycles or midpoints.

Recognize it when. The problem involves a linked list and asks about cycles, the middle node, the kth node from the end, or palindromes. It can also apply to sequences where you need to detect repetition without using extra memory.

How it works. The slow pointer moves one step at a time. The fast pointer moves two steps. If there is no cycle, fast reaches the end first. If there is a cycle, fast eventually laps slow inside the cycle and they meet. To find the cycle's start, reset one pointer to the head and move both at the same speed; they meet at the cycle entry. To find the middle, stop when fast reaches the end and slow is automatically at the middle.

Typical complexity. Time: O(n). Space: O(1).

Variations. Cycle detection (Floyd's algorithm). Finding the cycle start. Finding the middle node. Detecting palindromes by reversing the second half in place.

Example problems. Linked List Cycle, Linked List Cycle II, Middle of the Linked List, Palindrome Linked List, Happy Number.

Watch out for. Off-by-one errors when the list has even length and you need the "first middle" versus the "second middle." Always check whether fast and fast.next are non-null before moving fast two steps to avoid null-pointer errors.

3. Pattern: Sliding Window

Study this pattern in Grokking the Coding Interview

Core idea. Maintain a contiguous subarray or substring (the "window") and slide it across the input. As you extend the right edge and shrink the left edge, you keep an aggregate (sum, count, frequency map) updated in O(1) per step.

Recognize it when. The problem asks for a contiguous subarray or substring that meets a condition: maximum sum of size K, longest substring without repeats, smallest window containing all characters of a target, and so on. The brute force is recomputing over every subarray.

How it works. Two indices, left and right, both start at 0. Move right forward to expand the window, updating the running aggregate as each new element enters. When the window violates the condition or reaches a size limit, advance left to shrink it, updating the aggregate as each element leaves. Track the best window seen so far.

Typical complexity. Time: O(n). Space: O(1) or O(k) where k is the size of the character set or distinct values in the window.

Variations. Fixed-size window (size K is given). Dynamic window (size grows and shrinks based on a condition). Window with frequency map for character or element counts. Multi-pointer windows for problems with two coupled conditions.

Example problems. Maximum Sum Subarray of Size K, Smallest Subarray with a Given Sum, Longest Substring with K Distinct Characters, Longest Substring Without Repeating Characters, Permutation in String, Minimum Window Substring.

Watch out for. Forgetting to remove the leaving element from your aggregate when shrinking. Confusing "at most K distinct" with "exactly K distinct" (the second usually equals "at most K" minus "at most K minus 1"). Off-by-one errors in window length.

4. Pattern: Merge Intervals

Study this pattern in Grokking the Coding Interview

Core idea. When intervals overlap, you can combine them. Sorting by start time first turns a complicated geometric problem into a simple linear sweep.

Recognize it when. The input is a list of intervals or ranges, and you need to merge overlapping ones, find intersections, insert a new interval, or count conflicts. Calendars, meeting rooms, and time-range problems are all signals.

How it works. Sort intervals by start time. Walk through them in order, maintaining the current merged interval. For each new interval, check whether its start is less than or equal to the current end. If so, extend the current interval's end to the maximum of the two ends. If not, the new interval is disjoint, so push the current one to the result and start a new current interval.

Typical complexity. Time: O(n log n) due to sorting. Space: O(n) for the output, or O(1) if merging in place.

Variations. Merging overlapping intervals. Inserting one new interval into a sorted list. Finding intersections between two interval lists. Counting minimum meeting rooms (often solved with a min-heap of end times).

Example problems. Merge Intervals, Insert Interval, Intervals Intersection, Meeting Rooms II, Employee Free Time, Non-overlapping Intervals.

Watch out for. Edge-touching intervals like [1,3] and [3,5]: decide upfront whether they count as overlapping or disjoint, since the problem statement determines it. Sort by start time, not end time, unless the problem explicitly needs end-time order (like the activity selection greedy).

5. Pattern: Cyclic Sort

Study this pattern in Grokking the Coding Interview

Core idea. When an array contains numbers in a known range from 1 to n (or 0 to n minus 1), each number has a "correct" index. Place every number at its correct index in a single pass, and the array sorts itself in O(n).

Recognize it when. The array contains integers in a contiguous range like 1 to n, and the problem asks for a missing number, duplicate number, smallest positive missing, or to sort in O(n) without comparisons.

How it works. Walk through the array. For each index, check whether the value at that index belongs there (value equals index plus 1, in the standard 1-to-n case). If not, swap the value into its correct position. Repeat at the same index until the value is correct or out of range, then move on. After one pass, every misplaced number reveals a missing or duplicate slot.

Typical complexity. Time: O(n) amortized, since each swap places at least one number permanently. Space: O(1).

Variations. Find missing number. Find all missing numbers. Find duplicates. Find smallest missing positive integer. First k missing positives.

Example problems. Cyclic Sort, Find the Missing Number, Find All Duplicates, Find the Corrupt Pair, Find the Smallest Missing Positive Number.

Watch out for. This pattern only works when values are in a known small range. If you have arbitrary integers, use a hash set instead. Also, when swapping, do not advance the index until the current slot is correct, or you will leave misplaced values behind.

6. Pattern: In-place Reversal of a Linked List

Study this pattern in Grokking the Coding Interview

Core idea. Reverse a linked list (or a portion of one) by rewiring next-pointers as you walk through it, using only O(1) extra memory.

Recognize it when. The problem mentions reversing a linked list, reversing a sublist between positions, or reversing groups of K nodes. Any time the constraint says "do it in place" or "without extra memory," reversal in place is the move.

How it works. Maintain three pointers: previous, current, and next. At each step, save current.next into next, point current.next to previous, advance previous to current, and advance current to next. When current becomes null, previous is the new head. For sublist reversal, do the same thing between the two boundary nodes, then stitch the reversed portion back into the surrounding list.

Typical complexity. Time: O(n). Space: O(1).

Variations. Full reversal. Reverse a sub-list between two positions. Reverse every K-element sub-list. Reverse alternating K-element sub-lists. Rotate the list by K places.

Example problems. Reverse a Linked List, Reverse a Sub-list, Reverse Every K-element Sub-list, Reverse Alternating K-element Sub-list, Rotate a Linked List.

Watch out for. Losing the next pointer before saving it. Forgetting to set the original head's next to null after reversing (otherwise you create a cycle). For sublist reversals, drawing the diagram with dummy node first will save you many bugs.

7. Pattern: Stacks

Study this pattern in Grokking the Coding Interview

Core idea. A stack holds items in last-in, first-out order, so it always hands back the most recent unfinished item. That makes it the right structure whenever the answer depends on the nearest earlier thing.

Recognize it when. The problem involves matching or nesting (brackets, tags, folders), undo history, evaluating an expression, or anything phrased as "the most recent". Also when you need to reverse the order in which items were seen.

How it works. Push each item as you meet it. When the current item resolves whatever is on top, pop it and handle the pair. The top of the stack is always the most recent thing still waiting, which is exactly what these questions ask about. Anything left on the stack at the end was never resolved.

Typical complexity. Time: O(n), since each item is pushed and popped at most once. Space: O(n).

Variations. Bracket and tag matching. Expression evaluation, including postfix and infix. Undo or history stacks. Two stacks used to imitate a queue. A stack used to replace recursion with a loop.

Example problems. Balanced Parentheses, Reverse a String, Decimal to Binary Conversion, Next Greater Element, Sorting a Stack, Simplify Path.

Watch out for. Popping without checking that the stack is non-empty. Forgetting to check that the stack is empty at the end, which is what catches unmatched opening brackets. Confusing stack order with queue order when the problem wants the oldest item rather than the newest.

8. Pattern: Monotonic Stack

Study this pattern in Grokking the Coding Interview

Core idea. A stack where elements are kept in either non-increasing or non-decreasing order. When pushing a new element would violate the order, pop the offenders. Each pop usually reveals a "next greater" or "next smaller" relationship.

Recognize it when. The problem asks for the next greater element, next smaller element, previous greater, previous smaller, span of stock prices, largest rectangle in a histogram, or "for each element, find the closest X to the right." Anything with a "next/previous greater/smaller" theme fits.

How it works. Walk the array left to right. Maintain a stack of indices whose values are in monotonic order. Before pushing the current index, pop every element whose value violates the order (for example, every smaller value if the stack is non-increasing). Each popped element has just been "answered" by the current element, which is its next greater. Push the current index.

Typical complexity. Time: O(n). Space: O(n).

Variations. Next greater element. Next smaller element. Previous greater or smaller. Largest rectangle in histogram. Daily temperatures. Trapping rain water (alternative O(n) approach).

Example problems. Next Greater Element, Next Smaller Element, Largest Rectangle in Histogram, Daily Temperatures, Stock Span, Sum of Subarray Minimums.

Watch out for. Choosing the wrong monotonic direction (increasing vs decreasing) for the question being asked. Off-by-one in index versus value tracking. Forgetting to pop remaining elements at the end (some problems need a sentinel value at the end of the array to flush the stack).

9. Pattern: Hash Maps

Study this pattern in Grokking the Coding Interview

Core idea. A hash map turns a repeated linear search into a constant-time lookup by recording what you have already seen, keyed by value.

Recognize it when. The question asks whether something has been seen before, how often a value appears, or for a pair that meets a condition on unsorted input. Grouping items by a computed key and removing duplicates are the other common shapes.

How it works. Walk the input once. At each item, ask the map whether the thing you need is already there, which for a target-sum problem means the complement. If it is, you have the answer. If not, record the current item and move on. The single pass replaces the nested loop, and the map is what remembers the earlier half.

Typical complexity. Time: O(n) on average. Space: O(n).

Variations. Value to index for pair problems. Value to count for frequency. A sorted-letter key for grouping anagrams. A set when only membership matters. Running total to count, which is what makes subarray-sum counting linear.

Example problems. First Non-repeating Character, Largest Unique Number, Maximum Number of Balloons, Longest Palindrome, Ransom Note.

Watch out for. Claiming O(1) worst case, when the guarantee is only average. Reading a key that was never inserted instead of defaulting to zero. Forgetting to seed the map with a zero entry in the running-total counting problems. Using a map when the value range is small enough for a plain array, which is faster.

10. Pattern: Tree Level Order Traversal

Study this pattern in Grokking the Coding Interview

Core idea. Visit a tree level by level using a queue. Recording the queue's size before each round is what separates one level from the next, and because depth only ever increases, the first node that matches a condition is the closest one to the root.

Recognize it when. The answer is structured per level: level order, level averages, the largest value in each row, zigzag, or the width of the tree. Also when the wording says shortest, minimum depth, nearest, or first, or asks for a view of the tree from one side, or asks you to link nodes that sit side by side.

How it works. Push the root into a queue. While the queue is not empty, record its current size, which is the number of nodes at the current level, then loop that many times, popping each node, processing it, and pushing its children. Repeat until the queue empties. Capturing the size before the inner loop is the whole trick, because the loop adds children to the same queue as it runs. For a shortest-depth question, test each node as you pop it and return the current depth on the first match.

Typical complexity. Time: O(n) where n is the number of nodes. Space: O(w) where w is the widest level, which is ceil(n / 2) for a complete tree.

Variations. Standard level order. Reverse level order, by reversing the finished list. Zigzag, by alternating direction per level. One statistic per level, such as an average, sum or maximum. A rule tested per level. Early exit at the shallowest match. Connecting level order siblings with next pointers. Carrying a position index per node to measure width. Nodes with more than two children.

Example problems. Binary Tree Level Order Traversal, Reverse Level Order Traversal, Zigzag Traversal, Level Averages in a Binary Tree, Find Largest Value in Each Tree Row, Maximum Level Sum of a Binary

Tree, Even Odd Tree, Minimum Depth of a Binary Tree, Level Order Successor, Connect Level Order Siblings, Maximum Width of Binary Tree, N-ary Tree Level Order Traversal.

Watch out for. Forgetting to capture the level size before the inner loop, which mixes nodes from different levels. Treating a node with one child as a leaf, which gives a minimum depth that is too small. Pushing empty children, which corrupts every level count after it. Using an array as a queue, since removing from the front costs O(n) and makes the traversal quadratic. Not handling a null root up front. Measuring width by counting nodes rather than by the span between positions, which ignores the gaps.

11. Pattern: Tree Depth First Search

Study this pattern in Grokking the Coding Interview

Core idea. Walk a tree by going as deep as possible down one branch before backtracking and exploring the next. Naturally implemented with recursion (or an explicit stack).

Recognize it when. The problem asks for path-related quantities: root-to-leaf paths, path sums, longest path, diameter, lowest common ancestor, or any answer that depends on the relationship between an ancestor and its descendants.

How it works. Recurse on the left subtree, recurse on the right subtree, and combine their results at the current node. Pre-order, in-order, and post-order traversals differ only in when you process the current node relative to its children. For path-tracking problems, pass the current path as a parameter and either undo your changes when returning (true backtracking) or pass an immutable copy.

Typical complexity. Time: O(n). Space: O(h) where h is the height of the tree, due to the recursion stack. O(log n) for balanced trees, O(n) for skewed trees.

Variations. Pre-order, in-order, post-order. Path-sum problems. All-paths enumeration. Tree diameter (longest path between any two nodes). Lowest Common Ancestor. Serialize and deserialize.

Example problems. Binary Tree Path Sum, All Paths for a Sum, Count Paths for a Sum, Sum of Path Numbers, Tree Diameter, Path with Maximum Sum, Lowest Common Ancestor of a Binary Tree.

Watch out for. Forgetting the base case when the node is null. In path-tracking problems, not popping the last node from the path when returning (causes the path list to leak across branches). Confusing global state with returned values.

12. Pattern: Graphs

Study this pattern in Grokking the Coding Interview

Core idea. Model the input as nodes and edges, then traverse with breadth first or depth first search, marking nodes visited so nothing is explored twice.

Recognize it when. The input describes entities and relationships: networks, friendships, routes, flights, dependencies. The question asks about reachability, connectivity, or the fewest steps. A grid where each cell links to its neighbors is a graph in disguise.

How it works. Build an adjacency list from the edges. Start from a node, mark it visited, and repeatedly take a node and add its unvisited neighbors. A queue explores in rings of increasing distance, so it finds the fewest hops. A stack or recursion follows one route to its end before backing up. The visited set is what makes the cost linear rather than infinite.

Typical complexity. Time: O(V + E). Space: O(V + E) for the adjacency list, plus O(V) for the visited set.

Variations. Breadth first search for the shortest unweighted path. Depth first search for reachability and path building. Multi-source search that starts from several nodes at once. Two-coloring to test whether a graph is bipartite. Cycle detection.

Example problems. Find if Path Exists in Graph, Number of Provinces, Find Eventual Safe States, Minimum Number of Vertices to Reach All Nodes, Bus Routes.

Watch out for. Leaving out the visited set, which loops forever on a cycle. Marking a node visited when you pop it rather than when you push it, which lets the same node enter the queue many times. Assuming the graph is connected, when a disconnected one needs a fresh start from every unvisited node.

13. Pattern: Island (Matrix Traversal)

Study this pattern in Grokking the Coding Interview

Core idea. Treat a 2D grid like a graph and use BFS or DFS to find connected groups of cells. Each connected group is an "island."

Recognize it when. The input is a 2D matrix of land and water, 0s and 1s, or any cell type that can be connected to its neighbors. You need to count groups, find the largest group, paint a region, or check if cells are reachable from each other.

How it works. Walk through every cell. When you find an unvisited target cell, launch a DFS or BFS from it that explores all its connected neighbors (usually four directions, sometimes eight). Mark each visited cell so you do not revisit it. When the search finishes, you have processed one full island. Continue scanning for the next unvisited target cell.

Typical complexity. Time: O(rows × cols), since each cell is visited once. Space: O(rows × cols) for the visited set or recursion stack in the worst case.

Variations. Counting islands. Measuring island area. Flood fill (recoloring). Multi-source BFS for problems where multiple starting points spread simultaneously, like rotting oranges.

Example problems. Number of Islands, Max Area of Island, Flood Fill, Surrounded Regions, Rotting Oranges.

Watch out for. Forgetting to mark cells as visited, which causes infinite loops. Mixing up row and column indices. Missing the diagonal neighbors when the problem actually requires eight-directional connectivity.

14. Pattern: Two Heaps

Study this pattern in Grokking the Coding Interview

Core idea. Split a stream of numbers into a smaller half and a larger half, kept in a max-heap and a min-heap respectively. The tops of the two heaps give you instant access to the median or other order-statistic queries.

Recognize it when. You see a data stream of numbers and need the running median, or you need fast access to the largest of the small half and the smallest of the large half. Any "find the middle as numbers arrive" problem fits.

How it works. Maintain a max-heap for the smaller half and a min-heap for the larger half. When a new number arrives, push it onto one heap based on a comparison with the max-heap's top, then rebalance so the heap sizes differ by at most one. The median is either the top of the larger heap or the average of both tops.

Typical complexity. Insertion: O(log n). Median query: O(1). Total space: O(n).

Variations. Find the median from a number stream. Sliding window median (requires lazy deletion or an ordered multiset). Maximize capital with two heaps. IPO problem.

Example problems. Find the Median of a Number Stream, Sliding Window Median, Maximize Capital, IPO, Next Interval.

Watch out for. Forgetting to rebalance after each insertion, which lets one heap grow unbounded. In sliding window median, plain heaps cannot remove arbitrary elements efficiently, so use lazy deletion (mark elements as removed and lazily clean the top) or a balanced tree.

15. Pattern: Subsets

Study this pattern in Grokking the Coding Interview

Core idea. Enumerate all subsets, permutations, or combinations of an input by building them up incrementally. Each existing partial result spawns new partial results when the next element is considered.

Recognize it when. The problem asks you to generate all subsets, all permutations, all combinations, or to enumerate the power set. Any "find all possible X" question over a small input is a subset-pattern problem.

How it works. The classic iterative approach: start with a list containing the empty subset. For each input element, copy every existing subset and append the new element to the copies, then add the copies to the list. After processing all elements, the list contains all 2^n subsets. The recursive (backtracking) version makes a yes-or-no choice for each element.

Typical complexity. Time: O(n × 2^n) for subsets, O(n × n!) for permutations. Space: same as time, since you are storing the output.

Variations. Subsets without duplicates. Subsets with duplicates (sort first and skip equal siblings). Permutations. Permutations with duplicates. K-combinations. String permutations by changing case.

Example problems. Subsets, Subsets With Duplicates, Permutations, Letter Case Permutation, Balanced Parentheses, Unique Generalized Abbreviations.

Watch out for. Duplicate handling. When the input has duplicates, sort it first and skip elements equal to the previous one at the same recursion depth, otherwise you produce duplicate subsets. Also, distinguish between subsets (order does not matter) and permutations (order matters).

16. Pattern: Modified Binary Search

Study this pattern in Grokking the Coding Interview

Core idea. Binary search is not just for finding a value in a sorted array. The same halving logic applies whenever you can answer the question "is the answer to my left or my right" in O(1) at the midpoint.

Recognize it when. The input is sorted, partially sorted (rotated), or has a monotonic property. You need to find a target, a boundary, a peak, or the smallest value satisfying some condition. The constraint hints at log n or the input is huge.

How it works. Maintain low and high boundaries. At each step, compute mid, evaluate a condition, and decide whether to move low to mid plus 1 or high to mid minus 1. The trick in "modified" variants is the condition itself: it might compare neighbors, check rotation, or test a candidate answer in answer-space search.

Typical complexity. Time: O(log n). Space: O(1).

Variations. Order-agnostic binary search (you do not know if the array is ascending or descending). Search in a rotated sorted array. Find peak element. Find first or last position. Search in an infinite sorted array. Binary search on the answer (search the answer space, not the input).

Example problems. Order-agnostic Binary Search, Ceiling of a Number, Next Letter, Number Range, Search in a Sorted Infinite Array, Minimum in Rotated Sorted Array, Koko Eating Bananas, Capacity to Ship Packages.

Watch out for. The classic infinite loop: make sure low and high actually move on every iteration. Off-by-one errors in the boundary updates. For "binary search on the answer," correctly identifying the search space and the monotonic predicate is the whole problem.

17. Pattern: Bitwise XOR

Study this pattern in Grokking the Coding Interview

Core idea. XOR has two magical properties: a XOR a equals 0, and a XOR 0 equals a. Combined, these let you find unique elements among pairs or solve bit-manipulation problems with O(1) space.

Recognize it when. The problem involves finding a single number among pairs, finding two unique numbers, computing a complement, flipping bits, or using bit-level tricks. The input is integers and the constraints suggest constant-space.

How it works. XOR all the numbers together. Pairs cancel out, and the unique number remains. For "two unique numbers," XOR everything to get their XOR sum, then use any set bit in the result to partition the array into two groups, each containing one unique number, and XOR each group.

Typical complexity. Time: O(n). Space: O(1).

Variations. Single number among duplicates. Two single numbers among duplicates. Complement of a base-10 number. Flip and invert image. Missing number using XOR (XOR all values 0 to n with all array values).

Example problems. Single Number, Single Number II, Two Single Numbers, Complement of Base 10 Number, Flip an Image, Missing Number.

Watch out for. Bit-by-bit thinking is required: trace through small examples on paper before coding. Negative numbers and language-specific integer widths can bite you in problems like "complement of a base-10 number," where you only flip the bits used in the number's binary representation, not all 32.

18. Pattern: Top K Elements

Study this pattern in Grokking the Coding Interview

Core idea. When you need the top K largest, smallest, or most frequent elements out of n total, use a heap of size K rather than sorting all n elements. You trade O(n log n) for O(n log k).

Recognize it when. The problem asks for the K largest, K smallest, K most frequent, or Kth largest in a stream. Any "top K" or "Kth" phrasing is the strongest possible signal.

How it works. For the K largest, build a min-heap and push each element. If the heap grows beyond size K, pop the smallest. After processing every element, the heap contains the K largest. The current min on the heap is the Kth largest. For K smallest, mirror the logic with a max-heap.

Typical complexity. Time: O(n log k). Space: O(k).

Variations. Top K largest or smallest from an array. Top K frequent elements (use a frequency map plus heap). Kth largest in a stream. K closest points to origin. Reorganize string. Frequency sort.

Example problems. Top K Frequent Numbers, Kth Largest Number in a Stream, Top K Frequent Elements, K Closest Points to Origin, Connect Ropes with Minimum Cost, Reorganize String.

Watch out for. Choosing the wrong heap direction. For K largest, you need a min-heap of size K (counterintuitive at first). Forgetting to maintain the size bound, which collapses the optimization back to a full sort.

19. Pattern: K-way Merge

Study this pattern in Grokking the Coding Interview

Core idea. When you need to merge K sorted lists into one sorted output, a min-heap of "current head pointers" gives you the next smallest element in O(log K) per pop.

Recognize it when. The input is multiple sorted lists, sorted arrays, or sorted streams, and you need a single merged output, the Kth element across all of them, or a smallest range covering at least one element from each.

How it works. Push the first element of each list (along with its list identifier and index) onto a min-heap. Pop the smallest, append it to the output, and push the next element from that list onto the heap. Repeat until the heap is empty. The heap always contains exactly one candidate per list, so the next smallest is always at the top.

Typical complexity. Time: O(N log K) where N is the total number of elements across all lists. Space: O(K).

Variations. Merge K sorted linked lists. Merge K sorted arrays. Kth smallest in a sorted matrix. Smallest number range covering elements from K lists. Find K pairs with smallest sums.

Example problems. Merge K Sorted Lists, Kth Smallest Number in M Sorted Lists, Smallest Number Range, Find K Pairs with Smallest Sums, Kth Smallest Element in a Sorted Matrix.

Watch out for. Not handling empty input lists when building the initial heap (skip them). Forgetting to track which list each heap element came from, so you cannot push its successor. For "smallest range" problems, also track the current maximum across the heap.

20. Pattern: Greedy Algorithms

Study this pattern in Grokking the Coding Interview

Core idea. Take the best-looking choice at each step and never reconsider it. This is only correct when a local choice can be shown to be safe, and the argument almost always comes from sorting the input first.

Recognize it when. The question asks for the largest or smallest number of items, and one clear ordering makes each choice obvious: by end time, by size, by ratio. Interval scheduling, assignment, and resource problems are the usual shapes.

How it works. Sort by the quantity the argument depends on, then sweep once, taking each item that is still compatible with what you have already taken. Choosing the interval that ends earliest, for example, leaves the most room for everything after it. The sort is where the thinking happens. The sweep is trivial.

Typical complexity. Time: O(n log n), dominated by the sort. Space: O(1) beyond the sort.

Variations. Interval scheduling by earliest end time. Matching two sorted lists against each other. A heap when the best next choice changes as you go. Exchange arguments, where you show that swapping toward the greedy choice never makes the answer worse.

Example problems. Maximum Length of Pair Chain, Minimum Add to Make Parentheses Valid, Remove Duplicate Letters, Largest Palindromic Number, Removing Minimum and Maximum From Array.

Watch out for. Assuming greedy works without arguing why, which is the single most common mistake in this pattern. Sorting by the wrong key, which produces a confident wrong answer. Problems where an early choice costs you later, which need Dynamic Programming instead.

21. Pattern: 0/1 Knapsack (Dynamic Programming)

Study this pattern in Grokking the Coding Interview

Core idea. For each item, you make one yes-or-no decision: take it or skip it. The optimal answer is built up from optimal answers to smaller subproblems, indexed by the items considered so far and the remaining capacity.

Recognize it when. You have a set of items with weights or costs and values, and a capacity or budget. You need to maximize or minimize an aggregate. Or the problem can be reframed as "pick a subset that sums to X."

How it works. Define dp[i][c] as the best value achievable using the first i items with capacity c. The transition is a maximum over two cases: skip item i (dp[i minus 1][c]), or take item i if it fits (dp[i minus 1][c minus w_i] plus v_i). The answer is dp[n][C]. Space can usually be compressed from 2D to 1D by iterating capacity in reverse.

Typical complexity. Time: O(n × C). Space: O(n × C) for the 2D table, O(C) for the space-optimized 1D version.

Variations. Standard 0/1 Knapsack (maximize value). Equal Subset Sum Partition. Subset Sum (decision version). Minimum Subset Sum Difference. Count of Subsets with Given Sum. Target Sum (assign signs).

Example problems. 0/1 Knapsack, Equal Subset Sum Partition, Subset Sum, Minimum Subset Sum Difference, Count of Subsets with Given Sum, Target Sum.

Watch out for. Distinguishing 0/1 (each item used at most once) from Unbounded Knapsack (items can be reused). When space-optimizing to 1D, iterate capacity in reverse for 0/1 to avoid reusing the current item, and forward for Unbounded.

22. Pattern: Fibonacci Numbers (Dynamic Programming)

Study this pattern in Grokking the Coding Interview

Core idea. Each answer is built from a fixed number of earlier answers. Storing those instead of recomputing them turns an exponential recursion into a single linear pass.

Recognize it when. The recurrence has the shape f(n) = f(n-1) + f(n-2). Or the question counts the ways to reach step n, or asks for the best value at position n given the choices behind it. A plain recursion visibly recomputes the same subproblem many times.

How it works. Define the state as a position in the sequence. Write the recurrence that builds it from earlier states. Then fill an array forward from the base cases, so every value is ready when it is needed. Because only the last few entries are ever read, the array usually collapses to two or three variables.

Typical complexity. Time: O(n). Space: O(n) for the table, or O(1) once collapsed to variables.

Variations. Bottom-up table. Top-down recursion with memoization. Rolling variables for constant space. The same shape with a different combining rule, such as taking a maximum or minimum rather than a sum.

Example problems. Fibonacci Numbers, Staircase, Number Factors, Minimum Jumps to Reach the End, House Thief.

Watch out for. Wrong base cases, which shift every later answer. Collapsing to rolling variables before the recurrence is known to be correct. Overflow, since these values grow quickly.

23. Pattern: Palindromic Subsequence (Dynamic Programming)

Study this pattern in Grokking the Coding Interview

Core idea. The state is a range of the string rather than a single position. Whether a range works depends on its two ends plus the answer for the range inside them.

Recognize it when. The question is about palindromes, or about comparing two strings, or a string against its own reverse. The answer concerns a range, and the choice at each step is made at the two ends of that range.

How it works. Define the state as the range from i to j. If the characters at the two ends match, the answer builds on the range from i+1 to j-1. If they do not, take the better of dropping the left end or the right end. Fill the table by increasing range length, so every smaller range is already solved when a larger one needs it.

Typical complexity. Time: $O(n^2)$. Space: $O(n^2)$, often reducible to O(n).

Variations. Longest palindromic subsequence. Longest palindromic substring. Minimum deletions or insertions to make a palindrome. Counting palindromic substrings. Longest common subsequence and edit distance share the same two-index state.

Example problems. Longest Palindromic Subsequence, Longest Palindromic Substring, Count of Palindromic Substrings, Minimum Deletions in a String to make it a Palindrome, Palindromic Partitioning.

Watch out for. Filling the table in plain index order rather than by range length, which reads cells before they are written. Confusing a subsequence, which may skip characters, with a substring, which must be contiguous. Forgetting that a single character is itself a palindrome.

24. Pattern: Backtracking

Study this pattern in Grokking the Coding Interview

Core idea. Explore a tree of decisions by making a choice, recursing, and undoing the choice if it fails. Pruning bad branches early is what makes backtracking faster than pure brute force.

Recognize it when. The problem asks for all valid configurations, all paths, all placements, or for any one configuration meeting strict constraints. Combinatorial puzzles, board placements, and constraint satisfaction problems are textbook backtracking.

How it works. At each decision point, iterate over the available choices. For each choice, apply it, recurse to the next decision, and then undo it (the "backtrack" step) before trying the next choice. Cut off

branches early when partial state already violates a constraint. The decision tree has depth equal to the number of decisions and branching factor equal to the choices per decision.

Typical complexity. Highly variable. The worst case is exponential, often O(b^d) for branching factor b and depth d. Good pruning can dramatically reduce real-world runtime.

Variations. Generate all valid combinations. Constraint satisfaction (N-Queens, Sudoku). Path finding with constraints. Word search on a grid. Graph coloring. Generating parentheses.

Example problems. Sudoku Solver, N-Queens, Generate Parentheses, Word Search, Letter Combinations of a Phone Number, Combination Sum, Palindrome Partitioning.

Watch out for. Forgetting the undo step, which corrupts state across branches. Insufficient pruning, which makes the solution time out. Mutating shared state without undoing it on return.

25. Pattern: Trie

Study this pattern in Grokking the Coding Interview

Core idea. Store a set of strings in a tree where each edge carries one character, so a path from the root spells a prefix. Shared prefixes are stored once, which is what makes prefix questions cheap.

Recognize it when. The input is a set of words and the questions are about prefixes: autocomplete, dictionary lookup, spell check, or "starts with". Also when a search may contain a wildcard that matches any single character.

How it works. To insert a word, walk from the root following the child for each character, creating any child that does not exist. Mark the final node as the end of a word. To search a prefix, walk the same way and stop when the characters run out. To search a whole word, walk the same way and then check the end-of-word mark. That final check is the only difference between the two searches.

Typical complexity. Time: O(L) to insert or search a word of length L, whatever the size of the dictionary. Space: O(total characters), and far less when words share prefixes.

Variations. Prefix search. Whole-word search using the end-of-word mark. Wildcard search, which tries every child and backtracks. Collecting every word below a prefix for suggestions. Scanning a text by starting a walk at each position.

Example problems. Implement Trie (Prefix Tree), Design Add and Search Words Data Structure, Search Suggestions System, Index Pairs of a String, Extra Characters in a String.

Watch out for. Leaving out the end-of-word mark, which makes every prefix of a stored word look stored. Creating nodes during a search, so a failed lookup quietly grows the trie. Assuming lowercase input when using a 26-slot array. Claiming a trie beats a hash set for exact membership, which it does not.

26. Pattern: Topological Sort (Graph)

Study this pattern in Grokking the Coding Interview

Core idea. Order the vertices of a directed acyclic graph so every edge points from earlier to later in the order. If the graph has a cycle, no valid order exists.

Recognize it when. The problem describes prerequisites, dependencies, build order, course scheduling, or "do these things in some valid order." The data is a directed graph, and you need either a valid order or a count of valid orders, or to detect a cycle.

How it works. Kahn's algorithm: build an in-degree count for every vertex. Start a queue with all vertices of in-degree 0. Repeatedly pop a vertex, append it to the result, and decrement the in-degrees of its neighbors. When a neighbor reaches in-degree 0, push it. If the result contains all vertices, it is a valid topological order. If not, the graph has a cycle. The DFS-based variant pushes vertices onto a stack in post-order.

Typical complexity. Time: O(V plus E). Space: O(V plus E) for the graph and queue.

Variations. Single valid order (Kahn's BFS). All valid orders (backtracking over choices when multiple vertices have in-degree 0). Cycle detection. Lexicographically smallest order (use a min-heap instead of a queue). Alien dictionary (build the graph from string comparisons first).

Example problems. Task Scheduling Order, All Tasks Scheduling Orders, Alien Dictionary, Course Schedule, Course Schedule II, Reconstructing a Sequence.

Watch out for. Mistakenly running topological sort on an undirected graph: it does not apply. Forgetting that disconnected components still need to all show up in the result. When all-orders is required, the search space can blow up exponentially.

27. Pattern: Union Find

Study this pattern in Grokking the Coding Interview

Core idea. A data structure that tracks a partition of elements into disjoint sets and supports two operations efficiently: find (which set does this element belong to) and union (merge two sets). With path compression and union by rank, both are nearly O(1) on average.

Recognize it when. The problem involves connectivity in a graph, dynamic equivalence relations, or "are these two things in the same group?" Cycle detection in undirected graphs, connected components, Kruskal's MST algorithm, and the percolation problem all use Union Find.

How it works. Each element points to a parent, and the root of each tree represents the set. Find walks up the parent pointers until it reaches a root, optionally compressing the path by repointing every node on the way directly to the root. Union finds the roots of two elements and links one root to the other, optionally using rank or size to keep trees shallow.

Typical complexity. Time per operation: amortized O(α(n)), the inverse Ackermann function, which is effectively constant. Space: O(n).

Variations. Connectivity queries. Cycle detection in undirected graphs. Kruskal's minimum spanning tree. Number of connected components. Dynamic connectivity. Weighted Union Find for storing relationships beyond equivalence.

Example problems. Graph Redundant Connection, Number of Provinces, Is Graph Bipartite (alternative approach), Number of Islands II, Accounts Merge, Most Stones Removed with Same Row or Column.

Watch out for. Forgetting to apply path compression and union by rank, which degrades performance to O(n) per operation. Mutating the parent array while iterating, which is fine but easy to confuse with copying. For weighted Union Find, the relationship update during find requires care.

28. Pattern: Ordered Set

Study this pattern in Grokking the Coding Interview

Core idea. Keep a collection sorted while it is being changed, so the nearest element above or below a value stays cheap to find. A hash set answers only exact membership; an ordered set answers neighbor questions.

Recognize it when. You need the closest value above or below a target rather than an exact match. The collection is also inserted into or deleted from as you go. Bookings and clashes, and sliding windows that need both their largest and smallest value, are the common shapes.

How it works. The structure is a balanced binary search tree. Floor returns the largest element at or below a value, and ceiling the smallest at or above it. For an interval problem, only the two neighbors of a new interval can overlap it. Those two lookups settle the question, and everything further away is irrelevant.

Typical complexity. Time: O(log n) for insert, delete, floor, and ceiling. Space: O(n).

Variations. Sorted map from key to value. A set of non-overlapping intervals. A sliding window that needs both extremes at once. A multiset or a value-to-count map when duplicates matter.

Example problems. Merge Similar Items, 132 Pattern, My Calendar I, Longest Continuous Subarray.

Watch out for. Duplicates vanishing, since a set stores each value once. Floor or ceiling returning nothing at the ends of the range, which needs a check before use. Python having no built-in ordered set, so name the substitute you would use. Re-sorting a list after every insert, which is slower than the brute force it was meant to replace.

29. Pattern: Prefix Sum

Study this pattern in Grokking the Coding Interview

Core idea. Compute running totals once, so the sum of any range becomes the difference between two of them.

Recognize it when. The same fixed array is queried for range sums many times, or you are counting subarrays whose sum meets a condition. The wording mentions a running total, a balance point, or a pivot index. The brute force is a nested loop over every start and end.

How it works. Build an array where each entry holds the total of everything up to that point, which one pass does. The sum from i to j is then prefix[j] minus prefix[i-1]. Adding a zero entry at the front shifts

everything by one and removes the special case at index 0, which is where off-by-one errors live.

Typical complexity. Time: O(n) to build, then O(1) per query. Space: O(n).

Variations. Plain range sums. Counting subarrays that hit a target, using a map from running total to how often it has been seen. Grouping by remainder for divisible-by-k problems. Two-dimensional prefix sums for submatrix queries. Difference arrays, which apply the same idea to range updates.

Example problems. Find the Middle Index in Array, Subarray Sum Equals K, Subarray Sums Divisible by K, Maximum Size Subarray Sum Equals k, Binary Subarrays With Sum.

Watch out for. Off-by-one on the subtraction, so write down which version you are using before you start. Forgetting to seed the counting map with a zero entry. Using it on an array that changes, which needs a Binary Indexed Tree or a Segment Tree instead. Overflow, since running totals grow with the whole array.

30. Pattern: Multi-threaded

Study this pattern in Grokking the Coding Interview

Core idea. Use multiple threads with synchronization primitives (locks, semaphores, condition variables) to solve a problem where the order of execution matters or where work can be split safely.

Recognize it when. The problem statement explicitly mentions threads, concurrent execution, ordering across threads, or producer-consumer style scenarios. Common at companies that emphasize concurrency, especially for backend or systems roles.

How it works. Identify the synchronization need: ordering, mutual exclusion, or signaling. Use a counting semaphore to control how many threads can proceed past a point. Use a mutex for mutual exclusion. Use a condition variable (or wait-notify in Java) for "wait until X becomes true." Carefully design which thread releases the resource the next thread is waiting for, so no thread starves.

Typical complexity. Asymptotic complexity is usually the same as the single-threaded version; the goal is correctness under concurrent execution, not asymptotic speedup.

Variations. Strict ordering of prints across threads. Producer-consumer with bounded buffers. Reader-writer locks. Building a structure with multiple worker threads (web crawler). Dining philosophers (classic deadlock-avoidance).

Example problems. Print in Order, Print FooBar Alternately, Print Zero Even Odd, Building H2O, Fizz Buzz Multithreaded, Web Crawler Multithreaded, Dining Philosophers.

Watch out for. Deadlocks (always acquire locks in the same order). Race conditions (test shared state always inside a critical section). Spurious wake-ups in condition variables (always re-check the predicate inside a while loop, not an if).

31. Pattern: Counting

Study this pattern in Grokking the Coding Interview

Core idea. Many problems collapse to "count how many times each value appears" plus some downstream logic on those counts. Hash maps and frequency arrays are the workhorse data structures.

Recognize it when. The problem mentions frequency, occurrences, anagrams, majority element, finding duplicates, or "how many distinct/equal/different X are there." Inputs with bounded value ranges often use a fixed-size array instead of a hash map.

How it works. Walk the input once and increment a counter for each element. Then walk the counts and answer the actual question (find the max, find the threshold, find duplicates). For sliding-window variants, increment as elements enter and decrement as they leave.

Typical complexity. Time: O(n) for counting, plus whatever the downstream logic needs. Space: O(k) where k is the number of distinct values.

Variations. Frequency map for counting. Boyer-Moore majority vote (O(1) space for majority element). Counting sort as a stepping stone. Anagram detection by counting letters. Sliding window with frequency map (combine with the Sliding Window pattern).

Example problems. Count Elements With Maximum Frequency, Maximum Population Year, Least Number of Unique Integers after K Removals, Majority Element, Group Anagrams, Top K Frequent Elements.

Watch out for. Choosing a hash map when a fixed-size array would be faster (lowercase letters are 26 slots, ASCII is 128, etc.). Forgetting to handle case sensitivity or whitespace in string-counting problems.

32. Pattern: Monotonic Queue

Study this pattern in Grokking the Coding Interview

Core idea. A double-ended queue where elements are kept in monotonic order. Unlike a monotonic stack, you can pop from both ends, which makes it ideal for sliding window maximum or minimum queries.

Recognize it when. The problem combines a sliding window with a max or min query at each step, or asks for "the max in every window of size K." The brute force is O(n × k); the monotonic deque brings it to O(n).

How it works. Maintain a deque of indices. Before pushing the current index, pop from the back any index whose value violates monotonicity. Before reading the answer for the current window, pop from the front any index that has fallen outside the window. The front of the deque is always the index of the max (for a non-increasing deque) or min in the current window.

Typical complexity. Time: O(n), since each index is pushed and popped at most once. Space: O(k).

Variations. Sliding window maximum or minimum. Shortest subarray with sum at least K. Constrained subsequence sum. Jump game with deque optimization.

Example problems. Sliding Window Maximum, Shortest Subarray with Sum at Least K, Constrained Subsequence Sum, Jump Game VI, Sum of Subarray Minimums (combined with monotonic stack).

Watch out for. Storing values directly when you actually need indices (you usually need indices to detect when an entry has slid out of the window). Mixing up the front and back operations when implementing in your language of choice.

33. Pattern: Simulation

Study this pattern in Grokking the Coding Interview

Core idea. Some problems do not have a clever shortcut. The fastest path is to model the system step by step and execute the rules as written.

Recognize it when. The problem describes a process: a robot moving on a grid, water filling a container, a game playing out, a queue processing requests. The constraints are usually small enough that a direct simulation fits within the time limit.

How it works. Translate the rules of the system into code as faithfully as possible. Maintain the state (position, time, inventory, configuration) and update it according to the rules until a stopping condition is reached. For grid simulations, use direction arrays to handle movement compactly. For time-based simulations, often a priority queue keyed by event time is the right structure.

Typical complexity. Varies. Often O(steps × work-per-step). The constraints will hint at how large the simulation can be.

Variations. Robot or pointer moving on a grid. Queue or stream processing. Game state evolution. Spiral matrix traversal. Iterating until a fixed point or stable state.

Example problems. Array Transformation, Water Bottles, Spiral Matrix III, Robot Bounded in Circle, Game of Life, Pour Water.

Watch out for. Reaching for cleverness when simulation is the intended solution and would have been simpler. Underestimating the state size: if the state space repeats, you may need cycle detection. Off-by-one errors in movement and bounds checks.

34. Pattern: Linear Sorting Algorithms

Study this pattern in Grokking the Coding Interview

Core idea. Comparison-based sorting has a lower bound of O(n log n), but if the data is constrained (small range, fixed alphabet, distributable buckets), you can sort in O(n) using counting, radix, or bucket sort.

Recognize it when. The values are integers in a small known range, or strings of bounded length, or you have to sort by a key with a small distribution. Constraints like "0 to 1000" or "lowercase English letters" are strong signals.

How it works. Counting sort: count the frequency of each value, then write values back in order based on cumulative counts. Radix sort: counting-sort by each digit (or character) from least to most significant. Bucket sort: partition values into buckets by range, sort each bucket (often with insertion sort), and concatenate.

Typical complexity. Counting sort: O(n plus k) where k is the value range. Radix sort: O(d × (n plus k)) where d is the number of digits. Bucket sort: O(n) average, $O(n^2)$ worst case.

Variations. Counting sort. Radix sort (LSD or MSD). Bucket sort. Pigeonhole sort. Used as a subroutine for problems with extra constraints, like "sort by frequency."

Example problems. Relative Sort Array, Height Checker, Array Partition, Maximum Gap, Sort Colors (Dutch National Flag).

Watch out for. Memory usage: counting sort with a huge value range is wasteful and can blow up memory. Stability matters for radix sort: the inner counting sort must be stable. For bucket sort, choose bucket size carefully or you collapse to a single bucket.

35. Pattern: Meet in the Middle

Study this pattern in Grokking the Coding Interview

Core idea. When a brute-force search over n items is 2^n and too slow, split the input into two halves of size n/2, enumerate each half independently (2^(n/2) each), and combine the two halves smartly. The total cost drops from 2^n to roughly 2^(n/2) which can be the difference between feasible and impossible.

Recognize it when. The problem looks like a subset-sum or combination search, n is too large for full brute force (n around 30 to 40) but small enough that 2^(n/2) is fine. The brute force is exponential in n.

How it works. Split the input into halves A and B. Enumerate all subset sums (or whatever the partial state is) of A, and store them in a sorted list or hash map. Then enumerate all subsets of B; for each, compute what it needs from A and look it up. The lookup is O(log) or O(1), and the total cost is dominated by the two halves.

Typical complexity. Time: O(2^(n/2)) typically with a log factor for sorting or binary search. Space: O(2^(n/2)).

Variations. Subset sum equal to a target. Closest subset sum to target. Splitting into pairs of disjoint subsets. K-shortest path variants.

Example problems. Subset Sum Equal to Target, Subsets having Sum between A and B, Closest Subsequence Sum, Partition to K Equal Sum Subsets (with optimizations).

Watch out for. This pattern is only worth using when the input size makes it necessary. Below n equals 20 or so, plain DP or backtracking with pruning is usually simpler and just as fast. The combine step is the trickiest part to get right.

36. Pattern: Mo's Algorithm

Study this pattern in Grokking the Coding Interview

Core idea. When you have many range queries on a static array, group the queries cleverly so that successive queries share most of their range. Move the window's two boundaries incrementally between

queries and answer each in nearly constant amortized time per element move.

Recognize it when. The problem gives you an array and Q offline queries of the form "answer something about the range [L, R]." Q and N are both up to around 10^5 to 10^6, and a naive per-query loop times out. The queries are static and can be reordered.

How it works. Divide the array into blocks of size roughly √n. Sort the queries by (L's block, then by R), with R sorted in alternating directions per block. Maintain a current window [curL, curR] and an aggregate (count, frequency map). For each sorted query, slide curL and curR to match the query's L and R, updating the aggregate as elements enter and leave. Record the answer.

Typical complexity. Time: O((n plus q) × √n). Space: O(n) for the aggregate state.

Variations. Mo's algorithm on subarrays. Mo's on trees (with Euler tour). Mo's with updates (3D version, more complex).

Example problems. XOR Queries of a Subarray, Distinct Elements in a Subarray, Minimum Absolute Difference Queries, Powerful Array.

Watch out for. This pattern is rare in coding interviews and shows up mostly in competitive programming. Implementing the bucket sort and the alternating R-sort correctly is fiddly, so practice the template before relying on it. Online queries (where you must answer as they arrive) cannot use Mo's.

37. Pattern: Serialize and Deserialize

Study this pattern in Grokking the Coding Interview

Core idea. Convert a complex data structure (tree, graph) into a string or array (serialization), and reverse the process to reconstruct the original from that string (deserialization). The serialization format must capture enough structure for unambiguous reconstruction.

Recognize it when. The problem asks you to encode a tree or graph as a string, send it over a wire, persist it, or reconstruct it from a saved form. Codec design and BST-from-traversal problems are also serialization in disguise.

How it works. For trees, a common approach is pre-order traversal with explicit nulls (so you can rebuild the structure), or level-order with markers. For graphs, encode each node with a unique ID and serialize the adjacency list, then deserialize by first creating all nodes and then linking them. The format choice (delimiters, markers for null) determines how parsing works.

Typical complexity. Time: O(n) for both directions. Space: O(n) for the string and any reconstruction state.

Variations. Serialize and deserialize binary tree. Serialize and deserialize N-ary tree. Serialize and deserialize BST (can skip null markers since traversal order determines structure). Encode and decode strings.

Example problems. Serialize and Deserialize Binary Tree, Serialize and Deserialize N-ary Tree, Serialize and Deserialize BST, Encode and Decode Strings, Verify Preorder Serialization of a Binary Tree.

Watch out for. Forgetting to include null markers, which makes the structure ambiguous. Off-by-one errors when consuming the input string in deserialization. For graphs with cycles, using a visited map keyed by ID is critical to avoid infinite loops.

38. Pattern: Clone

Study this pattern in Grokking the Coding Interview

Core idea. Make a deep copy of a complex data structure (a graph or a linked list with extra pointers) such that no node in the copy shares memory with the original.

Recognize it when. The problem asks for a deep copy of a graph, a linked list with random pointers, or a tree with arbitrary cross-links. Anything saying "create an independent clone" or "modifications to the copy must not affect the original" is this pattern.

How it works. Walk the original structure with DFS or BFS. Maintain a hash map from each original node to its newly-cloned counterpart. When you encounter a node, create its clone if not already in the map, then recurse on its neighbors (or follow random pointers, child pointers, etc.) and link the clones together using the map. The map prevents infinite loops in cyclic structures and guarantees one clone per original.

Typical complexity. Time: O(n plus e) where n is nodes and e is edges or extra pointers. Space: O(n) for the map.

Variations. Clone a graph. Clone a linked list with random pointers (alternative O(1) space approach interleaves clones with originals). Clone an N-ary tree. Clone a binary tree with parent pointers.

Example problems. Clone Graph, Copy List with Random Pointer, Clone N-ary Tree, Clone Binary Tree with Random Pointer.

Watch out for. Forgetting to clone all neighbor links, which leaves the clone leaking back into the original. Not handling self-loops (a node whose neighbor includes itself). Missing the visited check, which causes infinite recursion in cyclic graphs.

39. Pattern: Articulation Point and Bridge

Study this pattern in Grokking the Coding Interview

Core idea. In an undirected graph, an articulation point is a vertex whose removal disconnects the graph; a bridge is an edge whose removal disconnects it. Both are found in linear time using DFS with discovery times and low-link values (Tarjan's algorithm).

Recognize it when. The problem is about graph connectivity under removal: critical routers in a network, single points of failure, edges whose loss splits the graph, or "what is the minimum to disconnect this." Network reliability and bridge-edge problems are the giveaway.

How it works. Run DFS, recording each node's discovery time and a "low" value defined as the minimum discovery time reachable from its subtree (including via at most one back-edge). A non-root node u is an articulation point if it has a child v with low[v] greater than or equal to disc[u]. The edge (u, v) is a bridge if low[v] is strictly greater than disc[u]. The root is an articulation point if it has more than one DFS child.

Typical complexity. Time: O(V plus E). Space: O(V) for the DFS state.

Variations. Find all articulation points. Find all bridges. Biconnected components. Two-edge-connected components. 2-vertex-connectivity.

Example problems. Critical Connections in a Network, Minimum Number of Days to Disconnect Island, Minimize Malware Spread, Minimize Malware Spread II.

Watch out for. The asymmetric condition for the root (uses child count, not low values). Confusing the strict greater-than for bridges with the greater-than-or-equal for articulation points. Make sure your graph is undirected; this technique does not apply to directed graphs as-is.

40. Pattern: Segment Tree

Study this pattern in Grokking the Coding Interview

Core idea. A balanced binary tree built over an array, where each internal node stores an aggregate (sum, min, max) of a range of the array. Point updates and range queries each cost O(log n).

Recognize it when. The problem mixes range queries (sum, min, max over [L, R]) with updates to individual elements or ranges. The number of queries plus updates is large enough that a per-query O(n) walk times out.

How it works. Build the tree bottom-up: leaves correspond to array elements, and each internal node stores the aggregate of its children. To query a range, recurse into nodes whose range overlaps with the query, combining their aggregates. To update, walk from the leaf up, recomputing aggregates along the path. Lazy propagation extends this to range updates by deferring changes to descendants until they are needed.

Typical complexity. Build: O(n). Query: O(log n). Update: O(log n). Space: O(4n) for safety in array-backed implementations.

Variations. Range sum or range min or range max with point updates. Range updates with lazy propagation. 2D segment trees. Persistent segment trees. Merge sort tree (segment tree where each node holds a sorted array).

Example problems. Range Sum Query - Mutable, Range Minimum Query, Queue Reconstruction by Height, Count of Smaller Numbers After Self, Falling Squares.

Watch out for. Off-by-one errors in node-to-range mapping. For lazy propagation, always push the lazy value down to children before doing further work. Use a 1-indexed array of size 4n to avoid out-of-bounds in some implementations.

41. Pattern: Binary Indexed Tree (Fenwick Tree)

Study this pattern in Grokking the Coding Interview

Core idea. A compact data structure for prefix sums (or other reversible aggregates) with O(log n) updates and prefix queries. Simpler to implement than a segment tree when you only need prefix-style

queries.

Recognize it when. The problem asks for prefix sums with point updates, count-of-smaller-elements (with coordinate compression), or any cumulative-frequency query. Range sum query plus update is the canonical use.

How it works. Use a 1-indexed array. Each index i stores the sum of a specific range determined by the lowest set bit of i. To update index i, walk upward by i plus (i AND minus i) until past n, adding the delta. To query prefix sum up to i, walk downward by i minus (i AND minus i) until 0, accumulating the sum. Range sums [L, R] are computed as prefix(R) minus prefix(L minus 1).

Typical complexity. Update: O(log n). Prefix query: O(log n). Space: O(n).

Variations. Prefix sum with point updates. Range update with point query (using a difference BIT). 2D BIT for matrix prefix sums. Order-statistics tree using a BIT over compressed coordinates.

Example problems. Number of Longest Increasing Subsequence, Maximum Profitable Triplets With Increasing Prices, Count of Range Sum, Reverse Pairs, Count of Smaller Numbers After Self.

Watch out for. Off-by-one indexing: BITs are usually 1-indexed, and forgetting this is the most common bug. The aggregate must be invertible (sum, XOR) for prefix subtraction to work; for max or min, you need a segment tree instead. Coordinate compression is often required when values are large but sparse.

Quick reference: pattern selection cheat sheet

When you see a problem, ask in this order:

1. **Is the input sorted?** Two Pointers, Modified Binary Search, or Merge Intervals are likely.
2. **Is it a contiguous subarray or substring problem?** Sliding Window or Monotonic Queue.
3. **Many range sums over an array that never changes?** Prefix Sum.
4. **Is it a linked list with cycle, middle, or in-place reversal?** Fast and Slow Pointers, or In-place Reversal of a Linked List.
5. **Is it a tree, answered per level, or by shortest depth?** Tree Level Order Traversal.
6. **Is it a tree, answered per path?** Tree Depth First Search.
7. **Is it a 2D grid with connected cells?** Island (Matrix Traversal).
8. **Nodes and edges, asking about routes or reachability?** Graphs.
9. **Matching, nesting, or the most recent unfinished item?** Stacks.
10. **Seen before, or grouping by a computed key?** Hash Maps.
11. **How often does each value appear?** Counting.
12. **Top K, Kth, or median?** Top K Elements or Two Heaps.
13. **The nearest value above or below, while the set keeps changing?** Ordered Set.
14. **All combinations or permutations?** Subsets or Backtracking.
15. **Prefixes, autocomplete, or a dictionary searched often?** Trie.
16. **Optimization with a capacity or budget?** 0/1 Knapsack.

17. **Counting the ways to reach step n?** Fibonacci Numbers.
18. **Palindromes, or comparing two strings?** Palindromic Subsequence.
19. **The best arrangement, where one sort settles every choice?** Greedy Algorithms.
20. **Dependencies or ordering?** Topological Sort.
21. **Connectivity or grouping?** Union Find.
22. **Multiple sorted streams?** K-way Merge.
23. **Next greater or smaller element?** Monotonic Stack.
24. **Many range queries, with the values changing too?** Segment Tree or Binary Indexed Tree.
25. **Many range queries for something that cannot be subtracted, all known up front?** Mo's Algorithm.
26. **None of the above, and the input is small?** Backtracking, Simulation, or Meet in the Middle.

The specialist patterns answer narrower questions: Cyclic Sort for a dense range of values, Bitwise XOR for a single unpaired number, Linear Sorting when the value range is small, Serialize and Deserialize and Clone for reproducing a structure, Articulation Point and Bridge for single points of failure, and Multi-threaded when the interviewer asks for concurrency.

If a problem still does not match, revisit the constraints. Constraints are the strongest hint about which pattern fits.

How to use this document the night before an interview

Pick the patterns most relevant to the company you are interviewing at and read those cards. For FAANG-style coding interviews, focus on Two Pointers, Sliding Window, Tree BFS, Tree DFS, Modified Binary Search, Top K Elements, Subsets, Backtracking, 0/1 Knapsack, and Monotonic Stack. Those ten patterns alone cover most questions you will see.

Read each card in the same order every time: the recognition cues first, then the mechanism, then the pitfalls. You do not need to memorize the example problem names. You need to be able to read a new problem and instantly say "this looks like a sliding window with a frequency map."

That is what pattern-based preparation gives you. Practice the recognition. The rest is filling in details.

Frequently asked questions

How many DSA patterns are there?

There is no official count. This cheat sheet lists 41, which is the set taught in *Grokking the Coding Interview* and one of the widest lists in use. Shorter lists, often around fifteen patterns, keep only the ones that come up most often and leave out the specialist techniques such as Mo's Algorithm, Meet in the Middle, and Binary Indexed Tree. The number matters less than the coverage. What you want is a list wide enough that no interview question falls outside it.

Which DSA patterns are most important?

Ten patterns account for most questions in a standard coding interview: Two Pointers, Sliding Window, Tree Level Order Traversal, Tree Depth First Search, Modified Binary Search, Top K Elements, Subsets, Backtracking, 0/1 Knapsack, and Monotonic Stack. Learn those first, and learn them well enough to recognize them from the question alone. Hash Maps, Graphs, and Island (Matrix Traversal) are the next three to add. The remaining patterns are worth a single read each, so that you recognize them if they appear.

Is there a DSA patterns PDF?

This page is the cheat sheet, and it is free to read without an account. There is no separate PDF file to download, but your browser's print dialog will save the whole page, table and all 41 cards included, as a PDF you can keep offline. Choose "Print" and then "Save as PDF" as the destination. Printing from a desktop browser gives the table enough width to stay readable.

Keep going

Each pattern in this cheat sheet links to its own lesson. To work through all 41 in order, with guided problems and solutions in six languages, see Grokking the Coding Interview.

For the 20 highest-return patterns written up with a code template and practice problems for each, read LeetCode Patterns: The Complete List.

More coding and system design interview courses are at DesignGurus.io.