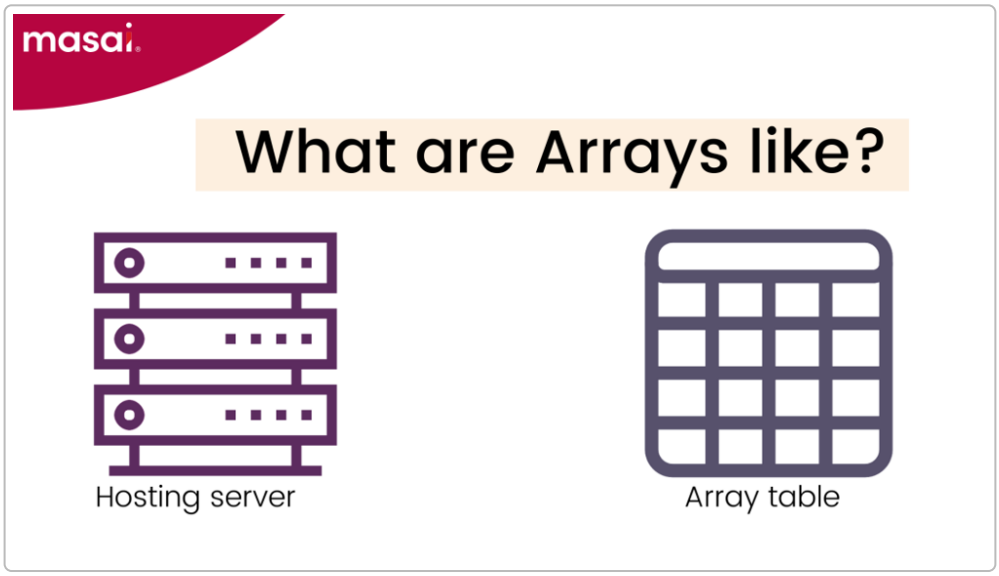
ChatGPT

Data Structures & Algorithms Interview Prep (6+ yrs Experience)

Introduction

This guide is a comprehensive, beginner-friendly learning path for core data structures and algorithms, tailored for mid-level developers preparing for product-based interviews (e.g. Google, Amazon). It covers fundamental topics starting from the basics and progressing to advanced concepts, each explained with clear examples and simple illustrations. Related ideas are linked together (e.g., showing how DFS uses recursion), and each section includes a set of progressively challenging coding problems from LeetCode and similar platforms.

Arrays



An *array* is a linear data structure where elements are stored in contiguous memory locations [1]. Each element has an index, allowing **O(1)** time access by index. Arrays are fundamental (used to implement other structures like stacks, queues, graphs) [2]. However, array size is fixed at allocation, making insertions or deletions (especially in the middle) costly (O(n)). For example, storing daily temperatures in an array lets you quickly retrieve any day's reading.

- Practice problems (easy → hard):
- Two Sum (Easy) – find two numbers that add up to a target.
- Contains Duplicate (Easy) – check if any number appears twice.
- Move Zeroes (Easy) – shift all 0s to the end of the array.
- Sort Colors (Medium) – sort array of 0s, 1s, and 2s with one-pass.

- Product of Array Except Self (Medium) – compute product of all other elements.
- 3Sum (Medium) – find triplets that sum to zero.
- Subarray Sum Equals K (Medium) – count subarrays summing to k.
- Maximum Subarray (Easy/Medium) – find contiguous subarray with max sum.
- Merge Intervals (Medium) – merge overlapping intervals (array of pairs).
- Longest Consecutive Sequence (Hard) – longest sequence of consecutive integers.

Each problem above builds on array concepts (sorting, two pointers, hashing), reinforcing array manipulation skills crucial for interviews.

Linked Lists

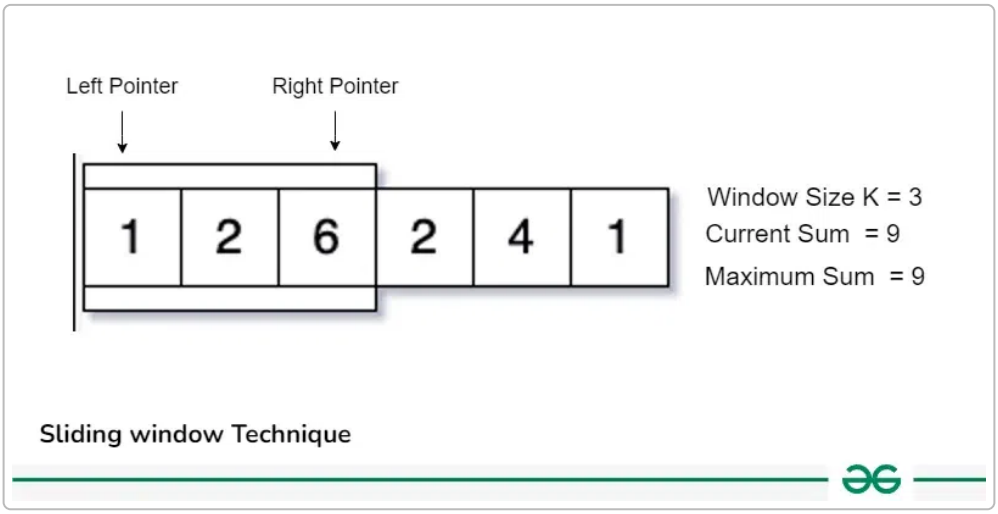
A *linked list* is a linear collection of nodes, where each node contains a value and a pointer to the next node [3]. Unlike arrays, linked list elements are **not stored at contiguous memory locations** [3]. This structure allows O(1) insertion or removal of nodes if you have a pointer to the correct spot, but random access is **O(n)** since you must traverse from the head. For example, a music playlist can be modeled as a linked list, where each song points to the next.

Linked lists implement stacks, queues, and other structures. Practice both *singly* and *doubly* linked lists (with `prev` pointers) for all operations: insertion, deletion, reversal, etc. Linked list problems often involve pointer manipulation, and they relate to other topics (e.g., merging sorted lists uses sorting logic, cycle detection can use hashing or two-pointers).

- Practice problems:
- Reverse Linked List (Easy) – reverse a singly linked list.
- Linked List Cycle (Easy) – detect if a cycle exists in the list.
- Merge Two Sorted Lists (Easy) – merge two sorted linked lists.
- Remove Nth Node From End (Medium) – remove the nth node from a list's end.
- Add Two Numbers (Medium) – add numbers represented by linked lists.
- Palindrome Linked List (Easy) – check if a linked list is a palindrome.
- Intersection of Two Linked Lists (Easy) – find the node where two lists intersect.
- Copy List with Random Pointer (Medium) – clone a list where nodes have random pointers.
- Merge k Sorted Lists (Hard) – merge multiple sorted linked lists.
- Linked List Cycle II (Medium) – find the start of a cycle in a linked list.
- Flatten a Multilevel Doubly Linked List (Medium) – flatten a nested linked list.
- Reorder List (Medium) – reorder list as L0→Ln→L1→Ln-1→…

Linked list problems reinforce pointer usage. They also have ties to other topics: for instance, combining lists (like Merge Lists) is akin to merging sorted arrays; cycle detection uses two-pointer or hashing techniques.

Sliding Window



The *sliding window* technique is used for problems involving subarrays or substrings. It involves defining a window (a contiguous range of indices) and moving it through the data to maintain a running state (sum, count, etc.) efficiently [4]. Instead of recomputing from scratch, you adjust the window by adding or removing elements. For example, to find the maximum sum of any subarray of size *k*, compute the sum of the first *k* elements. Then "slide" the window right one position each time: subtract the leftmost element and add the new rightmost element, updating the running sum. This makes each step O(1), for an overall O(n) solution in a single pass.

- Practice problems (arrays/strings):
- Minimum Size Subarray Sum (Medium) – smallest-length subarray with sum $\geq S$.
- Longest Substring Without Repeating Characters (Medium) – longest substring of unique chars.
- Permutation in String (Medium) – check if a string contains a permutation of another (sliding window with counts).
- Longest Repeating Character Replacement (Medium) – replace up to *k* chars to maximize repeating char substring.
- Maximum Average Subarray I (Easy) – fixed-size window average.
- Sliding Window Maximum (Hard) – find max in each sliding window (use deque or max-heap).
- Subarrays with K Different Integers (Hard) – count subarrays with exactly *K* distinct characters.
- Fruit Into Baskets (Medium) – longest subarray with ≤ 2 distinct fruit types.
- Max Consecutive Ones III (Medium) – longest subarray of 1s with $\leq$K zeros.
- Find All Anagrams in a String (Medium) – find substrings that are permutations of a pattern.

Tip: Sliding window is often implemented with a two-pointer approach. Adjust one pointer to expand or contract the window based on conditions, keeping track of the current count or sum.

Two-Pointer Technique

The two-pointer technique uses two indices moving through data to solve problems efficiently. A common pattern is placing one pointer at the start and one at the end of a sorted array, then moving them inward based on some condition. This technique can reduce an $O(n^2)$ solution to O(n). For example, to check if any

two numbers sum to a target in a sorted array, move left/right pointers inward until the sum matches (moving the smaller-side pointer up if sum is too small, etc.). This approach is a simple form of a greedy local adjustment on sorted data.

- Practice problems:
- Container With Most Water (Medium) – maximize water area between lines with two pointers.
- 3Sum (Medium) – fix one number, use two-pointer for the other two in a sorted array.
- Two Sum II - Input array is sorted (Easy) – find two numbers that sum to target (sorted input).
- Palindrome Number (Easy) – check if number is palindrome by comparing digits from both ends.
- Valid Palindrome III (Hard) – check if string can be palindrome by removing $\leq$k chars (two-pointer).
- Squares of a Sorted Array (Easy) – merge technique for sorted squares.
- Sort Array By Parity II (Easy) – rearrange array by parity using pointer swaps.
- 3Sum Smaller (Medium) – count triplets with sum < target using two-pointers.
- Leftmost Column With at Least a One (Medium) – use two-pointer search on a row-sorted matrix.
- Sum of Two Integers (Medium) – add without '+' using bitwise operations (conceptually treating bits from ends).

Two-pointer methods often appear in array/string problems (and some on linked lists, e.g., finding middle or merging two sorted lists). Note that the sliding window method is a form of two-pointer approach where the window boundaries move.

Recursion

Recursion is when a function calls itself to solve smaller subproblems. It often follows a divide-and-conquer approach: break a problem into simpler cases, solve those (with recursive calls), and combine results. Recursion is essential for solving tree/graph problems and underpins many dynamic programming solutions. For example, computing the Fibonacci sequence or factorial can be done recursively (though naive Fibonacci is slow without memoization). Each recursive call uses the call stack, so deep recursion can risk stack overflow.

Always define base cases in recursion. A common example is depth-first search (DFS) on a tree or graph: process the current node and recursively visit its neighbors. Notice that DFS on a graph is inherently recursive (or uses an explicit stack) [5]. Recursion also underpins backtracking: you try choices, recurse deeper, and backtrack on dead ends.

- Practice problems:
- Path Sum (Binary Tree) (Easy) – recursive traversal to check if a root-to-leaf sum exists.
- Climbing Stairs (Easy) – similar to Fibonacci (use recursion or DP).
- Pow(x, n) (Medium) – fast exponentiation using recursion.
- Decode Ways (Medium) – recursion + memo for counting decodings of a string.
- House Robber (Easy) – can be solved recursively with memo (linear DP).
- Merge Sort (Medium) – implement merge sort recursively.
- Quick Sort Partition (same as above).

These problems reinforce how recursion breaks problems into simpler ones. Many of these can be optimized with memoization (leading into DP).

Sorting & Binary Search

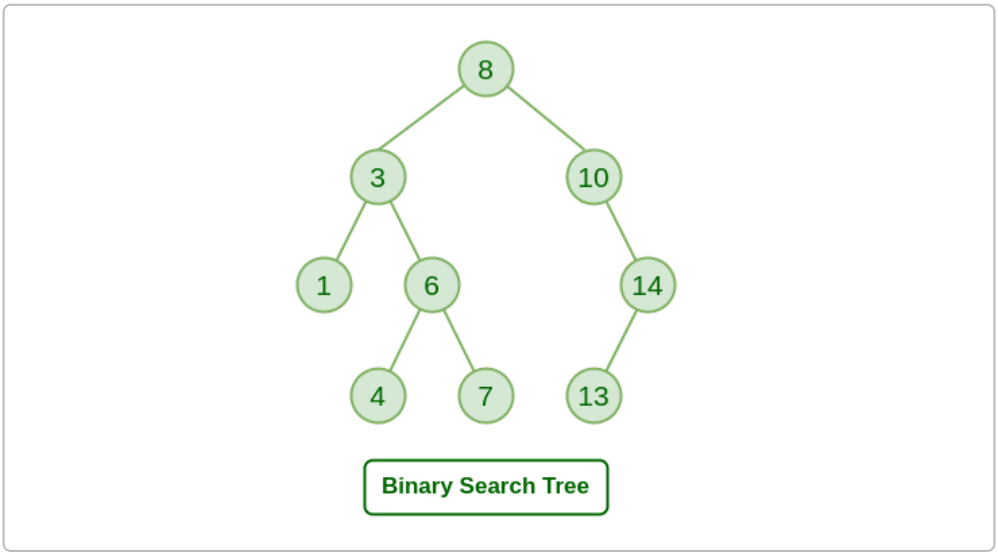
Sorting reorders elements, typically into ascending order. Classic algorithms (QuickSort, MergeSort, HeapSort) run in **O(n log n)** time [6]. Sorting is a foundational step: once data is sorted, many other algorithms become easier (like binary search or two-pointer solutions). For example, sorting an interval list by start times simplifies merging overlaps.

After sorting, you can use *binary search* to find elements quickly. Binary search finds a target in a sorted array by comparing to the middle element and halving the search interval [7]. It runs in O(log n) time [7], which is much faster than linear search for large data. Note: you must sort the array first if it isn't sorted. Binary search variants include finding insertion points or range boundaries.

- Practice problems:
- Sort Colors (Easy/Medium) – Dutch national flag problem, one-pass sort for three values.
- Merge Intervals (Medium) – sort intervals then merge them.
- Sort an Array (Medium) – implement merge sort or quicksort.
- Kth Largest Element in an Array (Medium) – use sort or Quickselect (average O(n)).
- Search in Rotated Sorted Array (Medium) – modified binary search on a rotated list.
- Find Minimum in Rotated Sorted Array (Medium) – binary search variant.
- Median of Two Sorted Arrays (Hard) – binary search solution.
- Binary Search (Easy) – implement binary search directly.
- Search Insert Position (Easy) – binary search for insertion index.
- Peak Element (Medium) – find a peak with binary search (divide-and-conquer).
- Kth Smallest Element in a Sorted Matrix (Medium) – use binary search on value or merge with heap.

Sorting connects to many topics: quicksort is recursive, and sorted arrays allow two-pointer or binary search solutions. Efficient sorting is key to optimizing numerous problems.

Trees (Binary Trees & BST)



In computer science, a *binary tree* is a hierarchical (non-linear) data structure in which each node has at most two children [8]. A special type is the *Binary Search Tree (BST)*: each node's left child has a value less than the node, and the right child has a value greater [9]. This BST property allows efficient search, insertion, and deletion (O(h) time, where h is height; O(log n) if balanced [9]).

Trees model hierarchical relationships (e.g., file directories). Common tasks include traversals: inorder, preorder, postorder (usually done recursively). Many problems on binary trees use DFS or BFS: e.g., tree height, path sums, or checking balance. For example, you might find the lowest common ancestor of two nodes, serialize/deserialize a tree, or traverse level-by-level. Trees and recursion go hand-in-hand: traversing a tree is often done with DFS. A *heap* is a special binary tree for priority queues (see Heap section).

- Practice problems:
- Maximum Depth of Binary Tree (Easy) – find tree height via DFS.
- Invert Binary Tree (Easy) – flip each node's children.
- Symmetric Tree (Easy) – check if tree is a mirror of itself.
- Binary Tree Level Order Traversal (Medium) – BFS by levels.
- Lowest Common Ancestor of a Binary Tree (Medium) – find LCA via recursion.
- Binary Tree Zigzag Level Order Traversal (Medium).
- Serialize and Deserialize Binary Tree (Hard) – convert tree to string and back.
- Validate Binary Search Tree (Medium) – check BST property.
- Kth Smallest Element in a BST (Medium) – using inorder traversal.
- Convert Sorted List to BST (Medium) – build balanced BST from sorted list.
- Binary Tree Maximum Path Sum (Hard) – DFS with DP on tree.
- Populate Next Right Pointers (Medium) – link tree levels.

Trees and recursion are deeply connected. A *BST* lets you do binary-search-like operations on tree nodes. A *heap* (priority queue) is a special tree for making quick max/min selection (see Heap section).

Backtracking

Backtracking is a recursive search strategy where you build a solution incrementally and abandon (backtrack) when a partial solution cannot lead to a valid full solution [10]. It's often used in constraint-satisfaction problems (puzzles, permutations, combinations). For example, solving the N-Queens problem involves placing queens row by row and backtracking when a conflict occurs. Backtracking algorithms use DFS at their core [5].

- Practice problems:
- Permutations (Medium) – generate all permutations of a list.
- Generate Parentheses (Medium) – build all valid parentheses combinations.
- Subsets (Medium) – list all subsets of a set.
- Combination Sum (Medium) – find combinations that add up to a target.
- N-Queens (Hard) – place queens on a chessboard with backtracking.
- Word Search (Medium) – search a word in a 2D board with DFS.
- Sudoku Solver (Hard) – fill a Sudoku board.
- Palindrome Partitioning (Medium) – partition a string into palindromes.
- Letter Combinations of a Phone Number (Medium) – generate letter combinations via backtracking.
- Combination Sum II (Medium) – variation of combination sum with no repeats.

Backtracking systematically explores choices and backtracks on dead ends [10]. It relies on recursion and often requires pruning branches to stay efficient.

Heap / Priority Queue

A *heap* is a special binary tree used to implement a priority queue. In a max-heap (for example), each parent's value is $\geq$ its children; the top (root) is the maximum. In a min-heap, each parent's value is $\leq$ its children. Heaps support fast priority access: operations like insert and extract-min/max take O(log n) time.

[11] A *priority queue* is an abstract data type where each element has a priority and the highest-priority element is processed first. Standard libraries implement priority queues using heaps. For example, to continually retrieve the smallest element, use a min-heap. Heaps are used in algorithms such as HeapSort (O(n log n) sorting) and Dijkstra's shortest path.

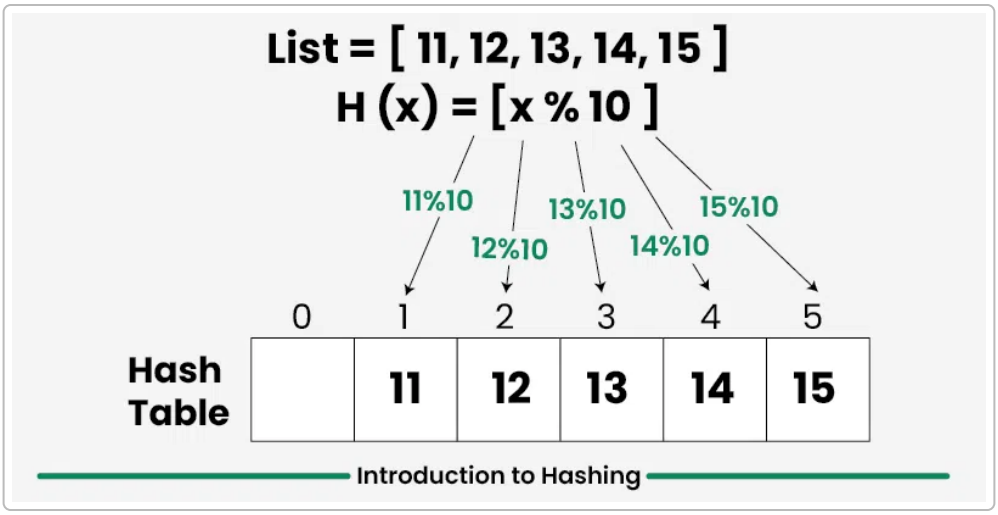
- Practice problems:
- Merge k Sorted Lists (Hard) – use a min-heap to merge multiple sorted lists.
- Top K Frequent Elements (Medium) – use a heap of element frequencies.
- Find Median from Data Stream (Hard) – maintain two heaps (max-heap/min-heap).
- Sliding Window Maximum (Hard) – use a max-heap to find window max each step.
- Task Scheduler (Medium) – schedule tasks by frequency (heap-based).
- Ugly Number II (Medium) – generate sequence using a min-heap (or DP).
- Kth Smallest Element in a Sorted Matrix (Medium) – use a heap over matrix elements.
- Sort Characters By Frequency (Medium) – count + heap to order by frequency.
- Kth Largest Element in an Array (Medium) – min-heap of size k.
- Meeting Rooms II (Medium) – schedule interval end-times with a min-heap.

Heaps often combine with other topics: e.g., greedy (always pick next best item), graphs (Dijkstra's priority queue), and arrays/strings (Top K problems). Learning heap operations helps in many interview scenarios.

Hashing

Hashing is a technique to map keys to array indices via a hash function, typically into a hash table [12]. A well-designed hash function distributes keys evenly. Common implementations use chaining (linked lists in buckets) or open addressing to resolve collisions. Hash tables support average-case O(1) insert, delete, and lookup [12], making them extremely useful for fast retrieval. Hashing powers data structures like sets and dictionaries.

For example, you can hash each number in an array to check for complements quickly (as in Two Sum), or hash substrings of text for fast pattern matching (rolling hash). Good hash usage is key in many problems.



For instance, hashing a list of integers by a modulo function is shown below: each number is placed in a table index based on *x % 10*, allowing direct lookup.

- Practice problems:
- Two Sum (Easy) – classic hash map for complements.
- Group Anagrams (Medium) – hash by sorted string or character frequency.
- Happy Number (Easy) – detect cycles in sum-of-squares via a hash set of seen sums.
- Isomorphic Strings (Easy) – map characters between two strings.
- Valid Anagram (Easy) – count chars with hash maps.
- Top K Frequent Elements (Medium) – hash count + heap (see above).
- Minimum Window Substring (Hard) – sliding window with hash counts.
- LRU Cache (Hard) – implement cache with hash map + linked list.
- Copy List with Random Pointer (Medium) – use hash to map old nodes to new.
- Word Pattern (Easy) – bijection check with maps.
- Longest Consecutive Sequence (Hard) – hash set to check consecutive neighbors.

Hashing intersects with many topics: arrays and strings (counting/frequency), sliding windows (character counts), greedy (frequency sort), or graph problems (visited sets). It's a fundamental tool for achieving efficient lookups.

Greedy Algorithms

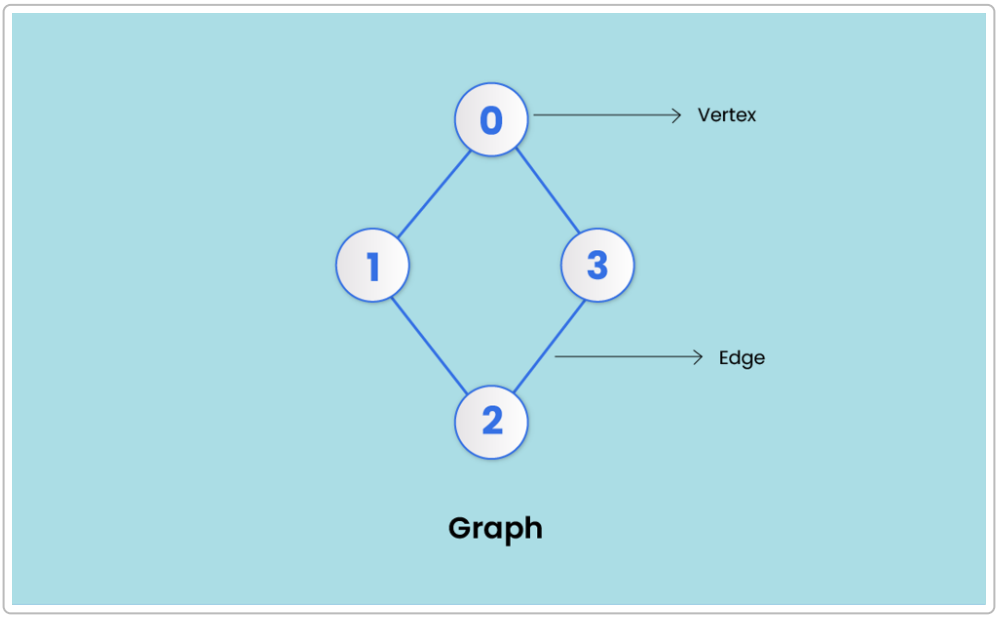
A *greedy algorithm* builds a solution by always taking the best immediate choice, without revisiting previous decisions [13]. It is often used for optimization problems where this local choice leads to a global optimum. However, it must satisfy the *greedy-choice property* and *optimal substructure* to be correct [14]. Classic examples: selecting the earliest finishing intervals to maximize a schedule, or picking the largest denominations first in making change (for canonical coin systems).

- Practice problems:
- Jump Game (Medium) – jump greedily to the furthest reachable index.
- Jump Game II (Hard) – reach end in minimum jumps.
- Gas Station (Medium) – find starting point in a circular route.

- Task Scheduler (Medium) – schedule tasks using frequency counts.
- Non-overlapping Intervals (Medium) – pick intervals by earliest finish time.
- Meeting Rooms II (Medium) – use a heap to count concurrent intervals.
- Candy (Hard) – distribute candies by local comparisons of ratings.
- Minimum Number of Arrows to Burst Balloons (Medium) – greedy on interval endpoints.
- Minimum Number of Refueling Stops (Hard) – greedy with max-heap of available fuel.

Greedy algorithms often pair with sorting: sort the items by a key and then apply a simple rule. It contrasts with DP because greediness never revisits past choices. Always consider whether a greedy approach is provably optimal for a given problem.

Graphs



A *graph* is a set of vertices (nodes) connected by edges [15]. It can be directed or undirected, weighted or unweighted. Graphs model networks like social connections, web links, or road maps. Key algorithms include: - **DFS (Depth-First Search)**: explores as deep as possible (often recursively) [5]. - **BFS (Breadth-First Search)**: explores layer by layer (using a queue). - **Shortest paths**: Dijkstra's, Bellman-Ford (for weighted graphs), or BFS for unweighted graphs. - **Minimum Spanning Tree**: Prim's, Kruskal's (the latter uses DSU). - **Topological Sort**: for DAGs (using DFS or Kahn's algorithm). - **Cycle Detection**: via DFS (coloring) or Union-Find for undirected graphs.

Graphs often use adjacency lists (or matrices). Remember to mark visited nodes to prevent infinite loops. Many interview problems translate real scenarios into graph problems (e.g., "friend groups" as connected components).

- Practice problems:
- Number of Islands (Medium) – count connected components in a grid (DFS/BFS).
- Clone Graph (Medium) – deep-copy a graph (BFS/DFS with a map).
- Course Schedule (Medium) – detect cycle in directed graph (DFS).
- Course Schedule II (Medium) – find a topological order of courses.

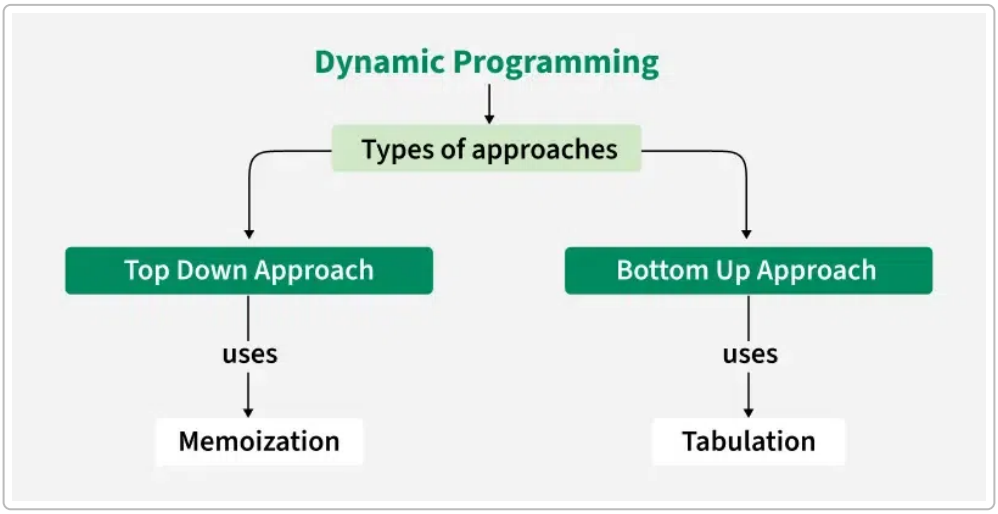
- Minimum Height Trees (Medium) – find tree centers by pruning leaves.
- Graph Valid Tree (Medium) – check if graph is acyclic and connected.
- Number of Connected Components (Medium) – DSU/DFS to count components.
- Rotting Oranges (Medium) – multi-source BFS in a grid.
- Walls and Gates (Medium) – multi-source BFS from gates.
- Longest Increasing Path in a Matrix (Hard) – DFS + memo on a graph-like grid.
- Word Ladder (Hard) – BFS on word graph.
- Alien Dictionary (Hard) – build graph of character order, then topo sort.
- Critical Connections in a Network (Hard) – Tarjan's algorithm to find bridges.

Graphs tie back to other topics: DFS is recursive (see Recursion), BFS uses queues (arrays/lists). Many graph problems involve DFS/BFS plus additional logic, so having a solid foundation in recursion and basic data structures helps.

Dynamic Programming

Dynamic Programming (DP) is an optimization technique that solves problems by breaking them into overlapping subproblems [16] and storing their solutions. Use DP when the problem has optimal substructure (overall optimum built from subproblem optima) and overlapping subproblems. There are two approaches: *top-down* with memoization (cache results of recursion) or *bottom-up* tabulation (fill a table iteratively) [17].

- **Benefits:** DP converts exponential-time recurrences into polynomial time by avoiding repeated work.
- **Key ideas:** Identify subproblem state and recurrence. Draw a DP table or memoize recursive calls.



DP comes in two approaches as shown: top-down (memo) and bottom-up (tabulation). Visualize the states (e.g. a 2D table for string problems) to understand transitions.

- Practice problems:
- Climbing Stairs (Easy) – Fibonacci-like DP.
- House Robber (Easy) – linear DP: choose or skip each house.
- Longest Increasing Subsequence (Medium) – DP or patience algorithm.

- Longest Common Subsequence (Medium) – 2D DP table for two strings.
- Palindrome Partitioning II (Hard) – DP on string splits.
- Coin Change (Medium) – DP on coin sums.
- Unique Paths (Medium) – grid DP count paths.
- Edit Distance (Hard) – string transform DP.
- Word Break (Medium) – DP on string segmentation.
- Decode Ways (Medium) – DP for count of decodings.
- Binary Tree Maximum Path Sum (Hard) – DFS with DP storing two states.
- Partition Equal Subset Sum (Medium) – subset-sum DP.

DP often builds on topics above: it frequently uses recursion underneath, and sometimes bit manipulation for state compression. Master DP after being comfortable with arrays, recursion, and simpler greedy/recursion problems.

Bit Manipulation

Bit manipulation involves operations on the binary representation of numbers [18]. Using bitwise operators like AND (`&`), OR (`|`), XOR (`^`), NOT (`~`), left shift (`<<`), and right shift (`>>`) can optimize tasks. For example, XOR of a number with itself is 0 (useful in "Single Number"), and shifting left by 1 multiplies by 2.

- Practice problems:
- Single Number (Easy) – XOR to find the unique element.
- Number of 1 Bits (Easy) – count set bits in an integer.
- Counting Bits (Medium) – use DP and the trick `dp[i] = dp[i & (i-1)] + 1`.
- Power of Two (Easy) – check if exactly one bit is set.
- Reverse Bits (Easy) – reverse bits in a 32-bit number.
- Sum of Two Integers (Medium) – add without '+' using bitwise XOR/AND.
- Bitwise AND of Numbers Range (Medium) – compute common leading bits of a range.
- Missing Number (Easy) – XOR all indices and numbers to find missing.
- Hamming Distance (Easy) – count differing bits between two numbers.
- Single Number II (Medium) – bit counting trick for triple occurrences.
- [Bitmask DP problems] (Advanced) – use bits to represent subsets for efficient DP.

Bit tricks complement other topics: e.g., DP on subsets often uses bitmasks, hashing sometimes uses bit operations, and greedy might use bits. They're essential for low-level optimizations and classic interview puzzles.

Tries (Prefix Trees)

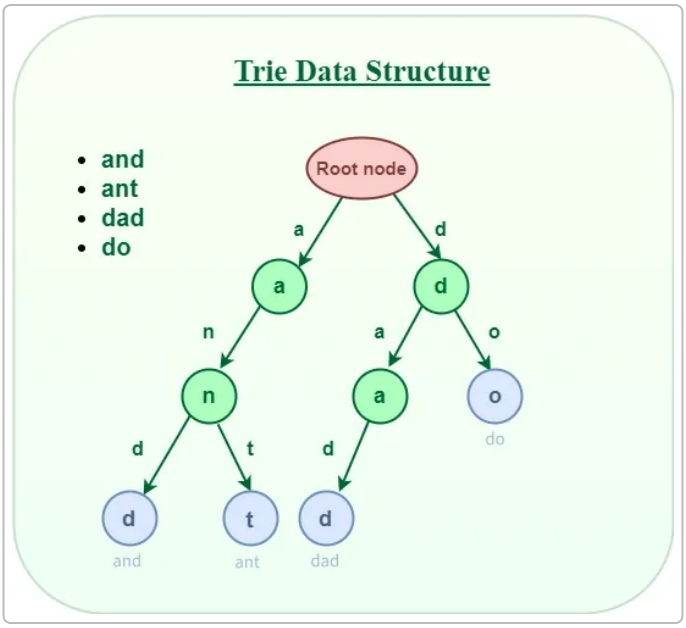
Tries (Prefix Trees)

A *Trie* (prefix tree) is a tree-like data structure used to efficiently store a dynamic set of strings [19]. Each node typically represents a character, and paths down the tree form keys. Tries allow fast prefix queries: for example, finding all words with a given prefix in O(*m*) time (m = prefix length), independent of the number of keys. They are used in autocomplete, spell-checkers, IP routing, and dictionary implementations.

- Practice problems:

- Implement Trie (Prefix Tree) (Medium) – build insert/search/startsWith.
- Word Search II (Hard) – search board for a list of words (use a trie to speed up searches).
- Longest Word in Dictionary (Medium) – find the longest word you can build one character at a time.
- Replace Words (Medium) – replace words with shortest root form (using a trie of roots).
- Stream of Characters (Hard) – query suffixes of a stream with a trie.
- [Contacts Search (custom)] – implement a contact list that returns number of names matching a prefix.
- Design Add and Search Words Data Structure (Medium) – implement a dictionary with wildcard search via trie.

Tries offer prefix-specific performance. They are essentially specialized trees for strings. They often work alongside hashing: tries use more space but enable prefix operations.



The pictured trie stores the words ["and","ant","dad","do"]. Each path from the root represents a word. For example, one path spells **"ant"** by following edges *a* → *n* → *t*.

Disjoint Set Union (Union-Find)

Disjoint Set Union (DSU) is a data structure that keeps track of a partition of elements into disjoint sets. It supports two main operations: **find** (determine which set an element belongs to) and **union** (merge two sets) [20] . Internally, each set has a representative (root), and with union-by-rank and path compression heuristics, these operations are nearly **O(1)** (amortized).

- Use Cases: Kruskal's Minimum Spanning Tree, tracking connected components, network connectivity queries.
- After each union operation, the structure can answer queries like "are elements x and y in the same set?" by comparing their set representatives.

- Practice problems:

- Number of Provinces (Medium) – count friend circles (connected components using DSU or DFS).
- Accounts Merge (Medium) – group email accounts using DSU on email addresses.
- Graph Valid Tree (Medium) – check for cycles/connectivity with DSU.
- Redundant Connection (Medium) – find an edge that creates a cycle.
- Sentence Similarity II (Medium) – check equivalence relation with DSU.
- Friend Requests II (Medium) – dynamic connectivity with DSU.
- Min Cost to Connect All Points (Kruskal's) (Medium) – MST via DSU.
- [Network / Percolation (custom)] – design connectivity queries on a dynamic network.

DSU is closely linked to graph algorithms. For example, in Kruskal's MST, edges are processed by increasing weight (a greedy choice) and unioned in DSU if they connect different trees. DSU elegantly handles connectivity and cycle detection in graphs.

Suggested Study Order

The topics above are organized from foundational to advanced. A recommended learning path is:

1. **Arrays & Linked Lists** – Master linear data structures first.
2. **Two-Pointers & Sliding Window** – Techniques built on arrays/strings.
3. **Hashing** – Key-value storage for O(1) access.
4. **Sorting & Binary Search** – Ordering data and fast search in sorted data.
5. **Recursion** – Fundamental approach used in trees and DP.
6. **Greedy Algorithms** – Simple local-optimization strategies (ensure correctness).
7. **Trees & Heaps** – Hierarchical structures (binary trees, BSTs, binary heaps).
8. **Graphs** – Complex networks, BFS/DFS, shortest paths.
9. **Backtracking** – Recursive search with pruning (requires recursion).
10. **Tries** – Prefix trees for strings.
11. **Disjoint Set (Union-Find)** – Connectivity and merging sets.
12. **Dynamic Programming** – Problem-solving by memo/table (builds on recursion).
13. **Bit Manipulation** – Low-level optimizations and tricks.

Follow this path for gradual complexity build-up. For example, recursion and pointers are used in many later topics (DFS on graphs, backtracking), so solidify them early. However, you can adjust the order based on your strengths.

Weekly Study Plan (22.5 hours/week)

Commit 2.5 hours on each weekday, 5 hours on weekends in a structured manner:

- **Week 1 (Arrays & Two-Pointers)**
  Mon–Fri: Cover array basics and practice simple array problems (Two Sum, Remove Duplicates, Move Zeroes). Learn two-pointer patterns on sorted arrays (Two Sum II, 3Sum, Container With Most Water).
  Sat–Sun: Solve 5–6 array and two-pointer problems (e.g., Sort Colors, Squares of Sorted Array, Max Consecutive Ones).

- **Week 2 (Linked Lists & Sliding Window)**
  Mon–Fri: Review linked list implementation and operations (Reverse, Cycle detect, Merge). Study the sliding window technique and implement it on simple examples.
  Sat–Sun: Solve 6–8 linked-list problems (Reverse, Merge, Remove Nth) and sliding-window problems (Min Subarray Sum, Longest Unique Substring, Sliding Window Maximum).

- **Week 3 (Hashing & Sorting/Binary Search)**
  Mon–Fri: Learn hash tables/maps (collisions, load factor) and basic sorting algorithms (merge/quick sort) plus binary search [7]. Practice easy hashing tasks (Valid Anagram, HashSet uses).
  Sat–Sun: Focus on sorting and binary search problems (Sort an Array, Merge Intervals, Kth Largest, Rotated Array search, Find Min in Rotated Array).

- **Week 4 (Recursion & Greedy)**
  Mon–Fri: Practice recursion on simple problems (factorial, Fibonacci with memo, DFS on tree) and study greedy strategies (interval scheduling, activity selection).
  Sat–Sun: Solve recursion/backtracking problems (Generate Parentheses, DFS puzzles) and greedy problems (Jump Game, Interval Scheduling, Gas Station).

- **Week 5 (Trees & Heaps)**
  Mon–Fri: Cover binary tree terminology and traversals (inorder, preorder, postorder using recursion and BFS). Learn heap/priority queue operations and library usage.
  Sat–Sun: Tackle 5–7 tree problems (Max Depth, Invert, LCA) and heap problems (Merge k Lists, Top K Frequent, Sliding Window Max, Median Data Stream).

- **Week 6 (Graphs)**
  Mon–Fri: Study graph representations and implement BFS & DFS (iterative and recursive) [5].
  Sat–Sun: Solve graph problems (Number of Islands, Clone Graph, Course Schedule, Graph Valid Tree). Include cycle/connectivity problems (with DFS and DSU).

- **Week 7 (Backtracking & Tries)**
  Mon–Fri: Learn backtracking (try possibilities, backtrack on dead ends) [10]. Practice puzzles like N-Queens, Sudoku. Understand Trie operations (build/search).
  Sat–Sun: Solve backtracking problems (Permutations, Combination Sum, Sudoku) and Trie problems (Implement Trie, Word Search II, Replace Words).

- **Week 8 (Union-Find, DP & Bit Manipulation)**
  Mon–Fri: Learn DSU (union-find) [20] and practice on connectivity problems. Begin dynamic programming basics (Climbing Stairs, Fibonacci).
  Sat–Sun: Solve DP problems (House Robber, Coin Change, LCS, Edit Distance) and bit manipulation puzzles (Single Number, Count Bits, Sum of Two Integers). Review any weak topics.

Adjust the schedule if needed (spend extra days on tough topics). The key is consistent daily practice and progressively harder problems.

Note: Use both reading (tutorials, visual aids) and coding practice (LeetCode, interviews). Drawing structures (linked lists, trees, graphs) can clarify concepts. Good luck with your preparation!

References: Definitions and insights above are supported by standard resources [1] [7] [9] [5] [10] [11] [12] [13] [20]. Embedded images illustrate key concepts (shown inline).

---

[1] [2] Array Data Structure Guide | GeeksforGeeks
https://www.geeksforgeeks.org/array-data-structure-guide/

[3] Linked List Data Structure | GeeksforGeeks
https://www.geeksforgeeks.org/linked-list-data-structure/

[4] Sliding Window Technique | GeeksforGeeks
https://www.geeksforgeeks.org/window-sliding-technique/

[5] Depth First Search Tutorials & Notes | Algorithms | HackerEarth
https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

[6] Sorting algorithm | Definition, Time Complexity, & Facts | Britannica
https://www.britannica.com/technology/sorting-algorithm

[7] Binary search - Wikipedia
https://en.wikipedia.org/wiki/Binary_search

[8] Binary tree - Wikipedia
https://en.wikipedia.org/wiki/Binary_tree

[9] Binary Search Tree | GeeksforGeeks
https://www.geeksforgeeks.org/binary-search-tree-data-structure/

[10] Introduction to Backtracking | GeeksforGeeks
https://www.geeksforgeeks.org/introduction-to-backtracking-2/

[11] Priority queue - Wikipedia
https://en.wikipedia.org/wiki/Priority_queue

[12] Hashing in Data Structure | GeeksforGeeks
https://www.geeksforgeeks.org/hashing-data-structure/

[13] [14] Greedy Algorithm
https://www.programiz.com/dsa/greedy-algorithm

[15] Graph Data Structure - Explained With Examples
https://www.masaischool.com/blog/graph-data-structure-explained-with-examples/

[16] [17] Dynamic Programming (DP) Introduction | GeeksforGeeks
https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/

[18] What is Bit Manipulation | GeeksforGeeks
https://www.geeksforgeeks.org/what-is-bit-manipulation/

[19] Trie Data Structure | GeeksforGeeks
https://www.geeksforgeeks.org/trie-insert-and-search/

[20] Introduction to Disjoint Set (Union-Find Algorithm) | GeeksforGeeks
https://www.geeksforgeeks.org/introduction-to-disjoint-set-data-structure-or-union-find-algorithm/