ChatGPT

LeetCode Mastery Roadmap – 300 Problems by Topic

Preparing for coding interviews is most effective when you **practice a curated set of problems** that cover all fundamental topics [1] [2]. Below is a structured list of ~300 recommended LeetCode problems **grouped by topic**. Each topic starts with a brief explanation of **why it matters**, followed by problems ordered from *Easy* to *Medium* to *Hard*. Solving these in order will reinforce patterns and gradually increase difficulty, helping you build mastery.

Arrays & Hashing

Why it matters: Arrays are a **fundamental data structure** and appear in many algorithmic problems. They are essential for numerous algorithms and are frequently tested in interviews [3]. Hash maps (dictionaries) are often used alongside arrays for efficient lookups, counting, and to solve problems like two-sum or anagrams. Mastering arrays and hash maps builds a foundation for many other topics.

- Two Sum – Easy [4] (Hash map for complement lookup)
- Contains Duplicate – Easy [5] (Use a hash set to detect duplicates)
- Valid Anagram – Easy [6] (Frequency counting using hash map)
- Majority Element – Easy (Boyer-Moore voting or counting)
- Product of Array Except Self – Medium [7] (Prefix/suffix products, no division)
- Group Anagrams – Medium [8] (Hash map with sorted string or count key)
- Top K Frequent Elements – Medium [9] (Hash map + heap or bucket sort)
- Encode and Decode Strings – Medium [10] (Design problem – encode list of strings)
- Valid Sudoku – Medium [11] (Use hash sets to validate rows, cols, blocks)
- Longest Consecutive Sequence – Medium [12] (Hash set for O(n) solution)
- Find the Duplicate Number – Medium (Detect cycle or use binary search on range)

Two Pointers & Sliding Window

Why it matters: Two-pointer techniques solve problems by **using two indices** to scan from ends or move at different speeds, enabling linear-time solutions for sorting, searching, or pairing problems [13]. The sliding window pattern is a variant that **maintains a window** over the data to handle subarray or substring problems efficiently (common for string challenges like finding substrings). These patterns eliminate the need for nested loops in many scenarios, making them favorites in interviews for array/string problems.

- Valid Palindrome – Easy [14] (Check from both ends with two pointers)
- Best Time to Buy and Sell Stock – Easy [15] (One-pass two-pointer for max profit)
- Remove Duplicates from Sorted Array – Easy (Two pointers to overwrite duplicates in-place)
- Two Sum II (Input Array Is Sorted) – Medium [16] (Two-pointer approach in sorted array)
- 3Sum – Medium [17] (Two pointers after sorting to find triplets)
- Sort Colors – Medium (Dutch national flag, three pointers or two-pass counting)

- Container With Most Water – Medium [18] (Two-pointer approach from ends for max area)
- Longest Substring Without Repeating Characters – Medium [19] (Sliding window with hash set/map)
- Longest Repeating Character Replacement – Medium [20] (Sliding window with frequency count)
- Permutation in String – Medium [21] (Sliding window + count comparison)
- Remove Nth Node From End of List – Medium [22] (Two-pointer technique in linked list)
- Minimum Window Substring – Hard [23] (Sliding window + counting, tricky)
- Trapping Rain Water – Hard [24] (Two pointers or stack to compute water trapping)

Stack & Queue

Why it matters: Stacks and queues are **foundational data structures** that come up often in interviews [25]. A **stack** (LIFO) is useful for parsing, backtracking, or tracking nested structures (e.g. parsing parentheses, evaluating expressions) [26]. A **queue** (FIFO) is essential for level-order traversals (BFS) and scheduling problems. Mastery of these simple structures is key to solving many problems efficiently (e.g. using a stack instead of recursion or using a queue for multi-step simulations).

- Valid Parentheses – Easy [27] (Stack to check matching brackets)
- Implement Queue using Stacks – Easy (Use two stacks to simulate queue operations)
- Min Stack – Medium [28] (Stack that can get min in O(1))
- Evaluate Reverse Polish Notation – Medium [29] (Stack to evaluate postfix expression)
- Generate Parentheses – Medium [30] (Backtracking, but stack helps validate generation)
- Daily Temperatures – Medium [31] (Monotonic stack for next greater element)
- Car Fleet – Medium [32] (Sort and use stack to count fleets)
- Design Browser History – Medium (Use two stacks to simulate back/forward history)

Binary Search

Why it matters: Binary search is a **fundamental algorithm** for searching sorted data in O(log n) time. It underpins many interview problems that ask you to find an element or optimal value in a sorted array or search space [2]. Mastering binary search (including variations like binary search on answer) allows you to handle problems that demand better-than-linear time.

- Binary Search – Easy [33] (Classic binary search in array)
- Search Insert Position – Easy (Binary search for insertion point)
- Search a 2D Matrix – Medium [34] (Binary search in matrix treated as sorted list)
- Find Peak Element – Medium (Binary search for local peak in unsorted array)
- Koko Eating Bananas – Medium [35] (Binary search on answer – find minimum eating speed)
- Find Minimum in Rotated Sorted Array – Medium [36] (Binary search in rotated array)
- Search in Rotated Sorted Array – Medium [37] (Binary search variation in rotated array)
- Time Based Key-Value Store – Medium [38] (Use binary search to find latest timestamp <= query)
- Median of Two Sorted Arrays – Hard [39] (Hard binary search partition technique)

Linked List

Why it matters: Linked lists are **fundamental data structures** commonly encountered in interviews [40]. They test your understanding of pointers and memory (since nodes are not contiguous like arrays). Mastery of linked lists demonstrates you can manipulate node pointers and solve problems like reversals, cycle

detection, and merging lists – skills that also help in more complex data structures. Practicing linked list problems also builds proficiency in recursion and two-pointer techniques.

- Reverse Linked List – Easy [41] (Iterative reversal of singly linked list)
- Merge Two Sorted Lists – Easy [42] (Merge linked lists like merging arrays)
- Linked List Cycle – Easy [43] (Detect cycle using two-pointer fast/slow)
- Intersection of Two Linked Lists – Easy (Find where two lists merge using two-pointer trick)
- Remove Nth Node from End of List – Medium [22] (Two-pointer technique to remove Kth from end)
- Reorder List – Medium [44] (Rearrange list in-place using middle finding + reverse second half)
- Add Two Numbers – Medium [45] (Simulate addition using linked list nodes)
- Copy List with Random Pointer – Medium [46] (Hash map or two-pass solution to clone list)
- LRU Cache – Medium (Design a cache with O(1) get/put using linked list + hash map)

Trees & Binary Search Trees

Why it matters: Trees are **hierarchical data structures** that appear frequently in interviews [47]. Many real-world structures (XML/HTML DOM, file systems, database indices) are trees. Interview questions often involve binary trees or BSTs to test recursion, DFS/BFS traversal, and understanding of tree properties (BST invariant allows efficient searching) [47] [48]. Mastering tree problems is crucial for tackling complex hierarchical problems and shows you can handle recursion and multiple pointer (parent-child) relationships.

- Invert Binary Tree – Easy [49] (Classic tree inversion, test simple DFS/BFS)
- Maximum Depth of Binary Tree – Easy [50] (DFS recursion for height)
- Diameter of Binary Tree – Easy [51] (DFS computing longest path through root)
- Balanced Binary Tree – Easy [52] (Check height balance via DFS)
- Same Tree – Easy [53] (DFS compare structure and values)
- Subtree of Another Tree – Easy [54] (Check if one tree is contained in another)
- Lowest Common Ancestor of a BST – Medium [55] (BST property for LCA)
- Lowest Common Ancestor of a Binary Tree – Medium [56] (General LCA via recursion)
- Binary Tree Level Order Traversal – Medium [57] (BFS using queue)
- Binary Tree Right Side View – Medium [58] (BFS or DFS to get rightmost node at each level)
- Populating Next Right Pointers in Each Node – Medium (Connect nodes at same level using queue or recursion)
- Validate Binary Search Tree – Medium [59] (DFS with bounds to validate BST property)
- Kth Smallest Element in a BST – Medium [60] (In-order traversal to find k-th smallest)
- Construct Binary Tree from Preorder and Inorder Traversal – Medium [61] (Reconstruct tree using recursion and inorder index map)
- Serialize and Deserialize Binary Tree – Hard (Design an algorithm to encode and decode a tree to/from a string)
- Binary Tree Maximum Path Sum – Hard (DFS to compute max path through each node, track global max)

Tries (Prefix Trees)

Why it matters: Tries (prefix trees) are special tree structures that store strings by their prefixes, enabling extremely fast prefix lookups. They are efficient for tasks like autocomplete, spell-check, or IP routing, which makes them **important in certain interview problems** focusing on string processing [62]. While tries are

not as commonly asked as arrays or trees, understanding them shows you can handle advanced data structures and string algorithms.

- Implement Trie (Prefix Tree) – Medium [63] (Design a trie with insert/search operations)
- Design Add and Search Words Data Structure – Medium [64] (Trie with `.` wildcard support in search)
- Word Search II – Hard [65] (Use a trie plus DFS backtracking to find all words in a grid)
- Replace Words – Medium (Use trie to replace prefixes in multiple words efficiently)

Backtracking (Recursion & DFS)

Why it matters: Backtracking is a **powerful algorithmic technique** essential for solving a wide range of combinatorial problems, especially in interviews [66]. It systematically explores all possibilities (depth-first), and **prunes** paths that fail constraints. Many problems – generating permutations/combinations, solving puzzles (sudoku), or DFS on implicit search spaces – rely on backtracking. Proficiency with backtracking demonstrates strong recursion skills and the ability to handle complex search scenarios.

- Subsets – Medium [67] [68] (Backtrack to generate all subsets of a set)
- Subsets II (with duplicates) – Medium [69] (Backtracking with skipping duplicates)
- Permutations – Medium [70] (Backtracking to generate all permutations)
- Combination Sum – Medium [67] [71] (Backtracking to find combinations that sum to target)
- Combination Sum II – Medium [72] (Combination sum with candidates only usable once)
- Word Search – Medium [73] (DFS backtracking in a grid to find a word)
- Palindrome Partitioning – Medium [74] (Backtrack to partition string into palindromic substrings)
- Sudoku Solver – Hard (Backtracking to fill a 9x9 Sudoku board)
- N-Queens – Hard (Place N queens on an N×N board via backtracking)

Heap / Priority Queue

Why it matters: Heaps (priority queues) provide efficient retrieval of the largest or smallest element. **Understanding heaps is essential** for many optimal solutions, as they are fundamental in solving various algorithmic problems efficiently [75] (e.g. scheduling, merging sorted data, running Dijkstra's algorithm). Interview questions often use heaps for k-th element retrieval or real-time ordering of streaming data.

- Kth Largest Element in a Stream – Easy [76] (Use a min-heap of size k)
- Last Stone Weight – Easy [77] (Max-heap to simulate stone collisions)
- K Closest Points to Origin – Medium [78] (Max-heap of size k or quickselect to find k closest points)
- Kth Largest Element in an Array – Medium [79] (Use heap or quickselect partition)
- Task Scheduler – Medium [80] (Max-heap for scheduling tasks with cooling)
- Design Twitter – Medium [81] (Use max-heap to fetch most recent tweets)
- Merge k Sorted Lists – Hard [82] (Min-heap to merge lists in O(n log k))
- Find Median from Data Stream – Hard [83] (Two heaps to maintain lower and upper halves for median)

Graphs (BFS/DFS & Union-Find)

Why it matters: Graphs model networks of relationships (e.g. social networks, web links, maps) and are a **popular topic in technical interviews** [48]. Many problems reduce to traversing a graph (using BFS/DFS) or

finding connectivity. Mastering graph algorithms demonstrates that you can handle complex problems like navigating mazes, scheduling tasks (topological sort), or grouping related items (union-find). Key patterns include **graph traversal (BFS/DFS)** and using data structures like adjacency lists or union-find to manage connectivity.

- Number of Islands – Medium [84] (DFS/BFS to count connected components in a grid)
- Max Area of Island – Medium [85] (DFS to find size of connected region)
- Clone Graph – Medium [86] (DFS/BFS + hash map to clone graph nodes)
- Walls and Gates – Medium [87] (Multi-source BFS from all gates to fill distances)
- Rotting Oranges – Medium [88] (BFS for multi-source spread process)
- Pacific Atlantic Water Flow – Medium [89] [90] (DFS/BFS from oceans to find reachable cells)
- Surrounded Regions – Medium [91] (DFS/BFS or union-find to capture regions)
- Course Schedule – Medium [92] (Detect cycle in directed graph via DFS or Kahn's BFS)
- Course Schedule II – Medium [93] (Topological sort of courses graph)
- Graph Valid Tree – Medium [94] (Use DFS or union-find to check connectivity & acyclicity)
- Number of Connected Components in an Undirected Graph – Medium [95] (Union-find or DFS to count components)
- Redundant Connection – Medium [96] (Union-find to find an extra edge creating a cycle)
- Network Delay Time – Medium (Dijkstra's algorithm on weighted graph to find delay)
- Word Ladder – Hard [90] (BFS in word graph to find shortest transformation sequence)
- Alien Dictionary – Hard (Topological sort to infer alphabet order from words)

Intervals (Interval Scheduling)

Why it matters: Interval problems involve ranges (start/end), such as meetings or events. They test your ability to **sort and then greedily process intervals**, a pattern that appears in scheduling and merging problems. Many interview questions use interval merging or scanning techniques, requiring careful handling of overlapping intervals. Mastering this topic helps in solving calendar/meeting problems and other scenarios where sorting by start or end times is key.

- Meeting Rooms – Easy [97] (Check if any intervals overlap after sorting by start time)
- Insert Interval – Medium [98] (Insert an interval into sorted list and merge overlaps)
- Merge Intervals – Medium [99] (Merge all overlapping intervals in a list)
- Non-overlapping Intervals – Medium [100] (Greedy elimination of intervals to avoid overlaps)
- Meeting Rooms II – Medium [101] (Min-number of conference rooms via min-heap or chronograph)
- Minimum Interval to Include Each Query – Hard [102] (Use a min-heap to cover queries with smallest intervals)

Greedy Algorithms

Why it matters: Greedy algorithms build a solution by **always choosing the best-looking option at each step** [103]. This technique is efficient for problems where the *greedy-choice property* holds (local optimum leads to global optimum). Common examples include interval scheduling (choose meeting that ends earliest) [104], coin change, or choosing moves in games. Interviews often include greedy problems to test if you can identify when a myopic strategy yields an optimal solution and prove its correctness.

- Maximum Subarray – Medium [105] (Kadane's algorithm – running max subarray sum)

- Jump Game – Medium [106] (Greedy to track the furthest reachable index)
- Jump Game II – Medium [107] (Greedy BFS-like approach to minimize jumps)
- Gas Station – Medium [108] (Greedy: if you can't reach a station, start from next candidate)
- Hand of Straights – Medium [109] (Greedy with sorting and min-heap to form sequences)
- Merge Triplets to Form Target Triplet – Medium [110] (Greedy check of conditions for merging triplets)
- Candy – Hard (Greedy two-pass algorithm to distribute candies fairly)
- Scheduling Courses – Hard (Greedy with max-heap to select courses within total duration)

Dynamic Programming

Why it matters: Dynamic programming (DP) is a technique to optimize recursive solutions by storing subproblem results. It is crucial for solving complex problems with overlapping subproblems and optimal substructure, turning exponential brute-force into efficient polynomial solutions [2]. DP is common in interviews for problems like pathfinding, sequence alignment, or optimal game strategies, and it showcases your ability to reason about states and transitions.

- Climbing Stairs – Easy [111] (Basic DP: Fibonacci-like steps count)
- Min Cost Climbing Stairs – Easy [112] (DP to accumulate min cost to reach top)
- House Robber – Medium [113] (DP on linear array – rob alternate houses)
- House Robber II – Medium [114] (DP on circular array – consider two cases)
- Unique Paths – Medium [115] (Combinatorial DP count in grid)
- Minimum Path Sum – Medium (DP for min sum path in matrix)
- Coin Change – Medium [116] (DP to find fewest coins for amount)
- Coin Change 2 – Medium (DP to count combinations of coins for amount)
- Longest Increasing Subsequence – Medium [117] (DP or patience sorting method for LIS length)
- Longest Common Subsequence – Medium [118] (Classic 2D DP for two-string subsequence)
- Palindromic Substrings – Medium [119] (DP or expand-around-center to count palindromes)
- Longest Palindromic Substring – Medium [119] (DP or center expansion to find longest palindrome)
- Decode Ways – Medium [120] (DP count of ways to decode digit strings)
- Word Break – Medium [121] (DP to check if string can be segmented into dictionary words)
- Partition Equal Subset Sum – Medium [122] (DP subset-sum to check if array can split evenly)
- Maximum Product Subarray – Medium [123] (DP track max/min products due to negatives)
- Edit Distance – Hard [124] (Classic DP for converting one string to another)
- Regular Expression Matching – Hard [125] (DP for string pattern matching with `.` and `*`)
- Burst Balloons – Hard (DP on intervals for max coins)

Bit Manipulation

Why it matters: Bit manipulation techniques use low-level binary operations to achieve extremely efficient solutions for certain problems (often O(1) or O(n) with very low constants). Interviewers include bit problems to test your understanding of binary representations and operations like XOR, AND, and shifts. Mastering bit tricks (like using XOR to find unique elements or bit masks for subsets) can make complex problems simpler and demonstrates a strong grasp of computer arithmetic.

- Single Number – Easy [126] (XOR all elements to find unique number)
- Number of 1 Bits – Easy [127] ("Hamming weight" – count bits set to 1)
- Counting Bits – Easy [128] (DP/bit trick: count bits for 0..n)

- Reverse Bits – Easy [129] (Bitwise operations to reverse 32-bit integer)
- Missing Number – Easy [130] (Use XOR or Gauss sum to find missing number from 0..n)
- Sum of Two Integers – Medium [131] (Bitwise addition without `+` using XOR and carry)
- Reverse Integer – Medium [132] (Careful math/overflow when reversing digits of int)
- Bitwise AND of Numbers Range – Medium (Bit shift common prefix of range [m,n])
- Maximum XOR of Two Numbers in an Array – Medium (Use bit tries to find max XOR pair)

Math & Miscellaneous (Geometry, Matrix)

Why it matters: A few interview problems don't fall neatly into the above categories but test your **mathematical reasoning or understanding of geometry/matrix operations**. These include problems on number theory (e.g., happy number, power of n), as well as 2D matrix manipulation or simulation problems. They are included in top lists to round out your knowledge and ensure you can handle anything from bit math to spiral matrix traversal.

- Happy Number – Easy [133] (Detect cycle in sum-of-squares process using fast/slow pointers)
- Plus One – Easy [134] (Simple simulation of adding one to number represented by array)
- Rotate Image – Medium [135] (Rotate matrix 90° in-place by transposing and reversing)
- Spiral Matrix – Medium [136] (Traverse matrix in spiral order using boundaries)
- Set Matrix Zeroes – Medium [137] (Use first row/col as markers to zero matrix in-place)
- Pow(x, n) – Medium [138] (Fast exponentiation by squaring – recursion/bitmask)
- Multiply Strings – Medium [139] (Simulate multiplication digit by digit)
- Detect Squares – Medium [140] (Use hash map to count points and detect squares)
- Permutation Sequence – Hard (Math to find k-th permutation using factorial number system)

Each of these topics and problems comes from **trusted sources** like Blind 75, **NeetCode 150**, LeetCode Explore cards, and other curated lists [2] [1]. By following this progression from easier to harder problems in each category, you'll cover a broad range of data structures and algorithms. Consistent practice of these ~300 problems will expose you to the most important patterns and prepare you for anything a real interview can throw at you. Good luck and happy coding!

Sources: Top curated problem lists and guides [2] [3] [40] [47] [141], including Blind 75 by Yangshun Tay and the NeetCode roadmap. Each problem listed is widely known for interview prep and appears in one or more of these recommended lists.

---

[1] I solved 1583 LeetCode problems. But you only need these 300. | by Ashish Pratap Singh | Medium
https://medium.com/@ashishps/i-solved-1583-leetcode-problems-but-you-only-need-these-300-db17e9297e00

[2] LeetCode Blind 75 patterns: Crack the coding interviews - DEV Community
https://dev.to/educative/leetcode-blind-75-1e00

[3] Top 10 Array Interview Questions: A Comprehensive Guide | by Kirti Arora | Medium
https://medium.com/@kirti07arora/top-10-array-interview-questions-a-comprehensive-guide-40557456a262

4 5 6 7 8 9 10 11 12 14 15 16 17 18 19 20 21 22 23 24 27 28 29 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46 49 50 51 52 53 54 55 56 57 58 59 60 61 63 64 65 67 68 69 70 71 72 73 74 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 GitHub - envico801/Neetcode-150-and-Blind-75: Neetcode 150 practice problems + Blind 75 techniques. Includes quizzes/questions/tests in flashcards format (Anki) to learn patterns and solutions. Collection of 225 leetcode problems.
https://github.com/envico801/Neetcode-150-and-Blind-75

13 Mastering the Two Pointer Technique for Technical Interviews | by Vidyasagar Ranganaboina | Medium
https://medium.com/@vidyasagarr7/mastering-the-two-pointer-technique-for-technical-interviews-fcce04128c0a

25 26 Understanding Stacks and Queues in JAVA: A Guide for Coding Interviews (EASY) | by Nathan Chiche | Medium
https://medium.com/@CHICHEEE/understanding-stacks-and-queues-in-java-a-guide-for-coding-interviews-easy-f5cfb5799e03

40 Mastering Linked Lists for Interview Preparations | by Rajat Sharma | The Pythoneers | Medium
https://medium.com/pythoneers/mastering-linked-lists-for-interview-preparations-885e9d09fc88

47 48 Trees and Their Applications in Computer Science – AlgoCademy Blog
https://algocademy.com/blog/trees-and-their-applications-in-computer-science/

62 How Often Does the Trie Data Structure Appear in Coding Interviews? – AlgoCademy Blog
https://algocademy.com/blog/how-often-does-the-trie-data-structure-appear-in-coding-interviews/

66 Mastering Backtracking: A Comprehensive Guide for Coding Interviews – AlgoCademy Blog
https://algocademy.com/blog/mastering-backtracking-a-comprehensive-guide-for-coding-interviews/

75 How to understand heap data structures for interviews?
https://www.designgurus.io/answers/detail/how-to-understand-heap-data-structures-for-interviews

103 104 Greedy Algorithms | Interview Cake
https://www.interviewcake.com/concept/java/greedy

141 Graph Algorithms Cheat Sheet For Coding Interviews
https://memgraph.com/blog/graph-algorithms-cheat-sheet-for-coding-interviews