Complete Algorithmic Problems Documentation

Compiled from Markdown Files

August 3, 2025

Contents

Chapter 1

Easy Problems

1.1 Array and String Problems

1.1.1 Two Number Sum

Problem: Find two numbers in array that sum to target value.
Solution: Use hash set to store seen numbers, check for complement.
Time: O(n), **Space**: O(n)
Key Insight: Use hash table for O(n) time, or sort and use two pointers for O(n log n) time.

1.1.2 Two Number Sum (Sorted)

Problem: Alternative solution using sorted array.
Solution: Use two pointers from ends, move based on sum comparison.
Time: O(n log n), **Space**: O(1)
Key Insight: Two-pointer technique works well with sorted arrays.

1.1.3 Valid Subsequence

Problem: Check if sequence is subsequence of array.
Solution: Use pointer to track sequence position, traverse array once.
Time: O(n), **Space**: O(1)
Key Insight: Single pass with pointer tracking is optimal.

1.1.4 Sorted Squared Array

Problem: Return sorted array of squares of input array.
Solution: Use two pointers from ends, fill result from largest to smallest.
Time: O(n), **Space**: O(n)
Key Insight: Largest squares come from largest absolute values at ends.

1.1.5 Tournament Winner

Problem: Determine tournament winner based on competitions and results.
Solution: Use hash map to track team scores, find maximum.
Time: O(n), **Space**: O(k)
Key Insight: Hash map for O(1) score updates and lookups.

1.1.6 Non-Constructible Change

Problem: Find minimum amount of change that cannot be created.
Solution: Sort coins, track current change created, check gaps.
Time: O(n log n), **Space**: O(1)
Key Insight: If coin ¿ current_change + 1, we found the gap.

1.1.7 Transpose Matrix

Problem: Transpose the given matrix.
Solution: Create new matrix with swapped dimensions, fill by swapping indices.
Time: O(n*m), **Space**: O(n*m)
Key Insight: Swap row and column indices: result[j][i] = matrix[i][j].

1.2 Binary Search Tree Problems

1.2.1 Find Closest Value in BST

Problem: Find closest value to target in BST.
Solution: Recursive traversal, update closest value, navigate based on target.
Time: O(h), **Space**: O(h)
Key Insight: Use BST property to eliminate half the tree each step.

1.2.2 Branch Sums

Problem: Calculate sum of all branches in binary tree.
Solution: Recursive DFS, track running sum, add to result at leaves.
Time: O(n), **Space**: O(h)
Key Insight: Use recursion to explore all paths to leaves.

1.2.3 Node Depths

Problem: Calculate sum of depths of all nodes in binary tree.
Solution: Recursive traversal, pass current depth, sum all depths.
Time: O(n), **Space**: O(h)
Key Insight: Each node's depth = parent's depth + 1.

1.3 Expression Tree and Graph Problems

1.3.1 Evaluate Expression Tree

Problem: Evaluate a binary expression tree.
Solution: Recursive evaluation, apply operators at internal nodes.
Time: O(n), **Space**: O(h)
Key Insight: Post-order traversal evaluates expressions correctly.

1.3.2 Depth-First Search

Problem: Perform DFS on graph and return array of node names.
Solution: Recursive traversal, add current node, process all children.
Time: O(v + e), **Space**: O(v)
Key Insight: Use recursion to explore all connected nodes.

1.4 Greedy Algorithms

1.4.1 Minimum Waiting Time

Problem: Calculate minimum total waiting time for queries.
Solution: Sort queries, process shortest first, multiply by remaining queries.
Time: O(n log n), **Space**: O(1)
Key Insight: Shortest queries should be processed first to minimize waiting.

1.4.2 Class Photos

Problem: Determine if students can be arranged in two rows for photo.
Solution: Sort both arrays, determine front row, check height constraints.
Time: O(n log n), **Space**: O(1)
Key Insight: Front row must be entirely shorter than back row.

1.4.3 Tandem Bicycle

Problem: Calculate total speed of tandem bicycles.
Solution: Sort both arrays, pair based on fastest/slowest preference.
Time: O(n log n), **Space**: O(1)
Key Insight: For maximum speed, pair fastest with fastest; for minimum, pair fastest with slowest.

1.4.4 Optimal Freelancing

Problem: Find optimal schedule for freelancing jobs.
Solution: Sort by payment, schedule on latest possible day within deadline.
Time: O(n log n), **Space**: O(1)
Key Insight: Greedy approach: take highest paying job on latest available day.

1.5 Linked List Problems

1.5.1 Remove Duplicates from Linked List

Problem: Remove duplicates from linked list.
Solution: Use two pointers, skip duplicates by linking to next distinct node.
Time: O(n), **Space**: O(1)
Key Insight: In-place removal by updating next pointers.

1.5.2 Middle Node

Problem: Find middle node of linked list.
Solution: Use two pointers (slow and fast), slow moves at half speed.
Time: O(n), **Space**: O(1)
Key Insight: Fast pointer reaches end when slow pointer is at middle.

1.6 Dynamic Programming

1.6.1 Get Nth Fibonacci

Problem: Calculate nth Fibonacci number.
Solution: Iterative approach, track previous two numbers.
Time: O(n), **Space**: O(1)
Key Insight: Only need to track last two numbers, not entire sequence.

1.6.2 Product Sum

Problem: Calculate product sum of special array with nested arrays.
Solution: Recursive traversal, multiply by depth level.
Time: O(n), **Space**: O(d)
Key Insight: Use recursion to handle nested structures, track depth.

1.7 Searching and Sorting

1.7.1 Binary Search

Problem: Perform binary search to find target in sorted array.
Solution: Divide and conquer, eliminate half the search space each step.
Time: O(log n), **Space**: O(1)
Key Insight: Halve search space each iteration for logarithmic time.

1.7.2 Find Three Largest Numbers

Problem: Find three largest numbers in array.
Solution: Track three largest values, update and shift as needed.
Time: O(n), **Space**: O(1)
Key Insight: Only need to track three values, not entire array.

1.7.3 Bubble Sort

Problem: Sort array using bubble sort algorithm.
Solution: Compare adjacent elements, swap if needed, repeat until sorted.
Time: O(n^2), **Space**: O(1)
Key Insight: Largest elements "bubble up" to their correct positions.

1.7.4 Insertion Sort

Problem: Sort array using insertion sort algorithm.
Solution: Build sorted array incrementally, insert each element in correct position.
Time: O(n^2), **Space**: O(1)
Key Insight: Each element is inserted into already sorted portion.

1.7.5 Selection Sort

Problem: Sort array using selection sort algorithm.
Solution: Find minimum element, swap to front, repeat for unsorted portion.
Time: O(n^2), **Space**: O(1)
Key Insight: Find minimum of unsorted portion and place at beginning.

1.8 String Manipulation

1.8.1 Is Palindrome

Problem: Check if string is palindrome.
Solution: Use two pointers from ends, compare characters moving inward.
Time: O(n), **Space**: O(1)
Key Insight: Compare characters from both ends simultaneously.

1.8.2 Caesar Cipher Encryptor

Problem: Encrypt string using Caesar cipher.
Solution: Shift each character by key, handle wrapping around alphabet.
Time: O(n), **Space**: O(n)
Key Insight: Use modulo to handle key wrapping: key = key % 26.

1.8.3 Run-Length Encoding

Problem: Encode string using run-length encoding.
Solution: Count consecutive characters, encode as count + character.
Time: O(n), **Space**: O(n)
Key Insight: Track current character and count, reset when character changes.

1.8.4 Common Characters

Problem: Find characters common to all strings.
Solution: Track minimum count of each character across all strings.
Time: O(n*m), **Space**: O(c)
Key Insight: Character must appear in all strings with minimum frequency.

1.8.5 Generate Document

Problem: Check if document can be generated from characters.
Solution: Count available characters, check if sufficient for document.
Time: O(n + m), **Space**: O(c)
Key Insight: Use hash map to track character frequencies.

1.8.6 First Non-Repeating Character

Problem: Find first non-repeating character in string.
Solution: Count character frequencies, find first with count 1.
Time: O(n), **Space**: O(1)
Key Insight: Two-pass approach: count frequencies, then find first unique.

1.8.7 Semordnilap

Problem: Find all semordnilap pairs (words that are reverse of each other).
Solution: Use hash set, check if reverse of each word exists.
Time: O(n*m), **Space**: O(n)
Key Insight: Check if reverse of each word exists in the set.

Chapter 2

Medium Problems

2.1 Array Algorithms

2.1.1 Three Number Sum

Problem: Find all triplets in array that sum to target value.
Solution: Sort array, use three pointers (one fixed, two moving), eliminate duplicates.
Time: O(n^2), **Space**: O(n)
Key Insight: Use sorting and two-pointer technique to avoid O(n^3) complexity.

2.1.2 Smallest Difference

Problem: Find pair of numbers (one from each array) with smallest absolute difference.
Solution: Sort both arrays, use two pointers to find closest pair.
Time: O(n log n + m log m), **Space**: O(1)
Key Insight: Moving larger pointer can only increase difference.

2.1.3 Move Element to End

Problem: Move all instances of specified integer to end of array.
Solution: Use two pointers from ends, swap when left pointer finds target element.
Time: O(n), **Space**: O(1)
Key Insight: Use two pointers to avoid extra space.

2.1.4 Monotonic Array

Problem: Check if array is entirely non-increasing or non-decreasing.
Solution: Single pass, track direction, check for direction breaks.
Time: O(n), **Space**: O(1)
Key Insight: Track direction and check for breaks.

2.1.5 Spiral Traverse

Problem: Traverse 2D array in spiral order starting from top-left.
Solution: Use four boundary variables, traverse in four directions, update boundaries.
Time: O(n), **Space**: O(n)
Key Insight: Use four boundary variables to track spiral pattern.

2.1.6 Longest Peak

Problem: Find length of longest peak (strictly increasing then strictly decreasing).
Solution: Find peak points, expand left and right to find full peak length.
Time: O(n), **Space**: O(1)
Key Insight: Expand from peak points to find full peak boundaries.

2.1.7 Array of Products

Problem: Return array where each element is product of every other number.
Solution: Two-pass algorithm - left products then right products.
Time: O(n), **Space**: O(n)
Key Insight: Use two passes to avoid division and handle zeros.

2.1.8 First Duplicate Value

Problem: Find first duplicate value in array where values are between 1 and n.
Solution: Use array as hash set, mark visited indices as negative.
Time: O(n), **Space**: O(1)
Key Insight: Use array indices as hash table, mark visited as negative.

2.1.9 Merge Overlapping Intervals

Problem: Merge overlapping intervals in list of intervals.
Solution: Sort intervals, merge overlapping ones sequentially.
Time: O(n log n), **Space**: O(n)
Key Insight: Sort first, then merge overlapping intervals sequentially.

2.1.10 Best Seat

Problem: Find best seat in row where 0=empty, 1=occupied. Best seat maximizes distance to closest person.
Solution: Find largest consecutive empty section, sit in middle.
Time: O(n), **Space**: O(1)
Key Insight: Find largest consecutive empty section and sit in middle.

2.1.11 Zero Sum Subarray

Problem: Check if there exists a subarray that sums to zero.
Solution: Use hash table to store prefix sums.

Time: O(n), **Space**: O(n)
Key Insight: If same prefix sum appears twice, subarray between them sums to zero.

2.1.12 Missing Numbers

Problem: Find all missing numbers in array containing numbers from 1 to n.
Solution: Use array as hash set, mark existing numbers as negative.
Time: O(n), **Space**: O(1)
Key Insight: Use array indices to track which numbers exist.

2.1.13 Majority Element

Problem: Find the majority element in array (appears more than n/2 times).
Solution: Boyer-Moore voting algorithm.
Time: O(n), **Space**: O(1)
Key Insight: Cancel out pairs of different elements, remaining element is majority.

2.1.14 Sweet and Savory

Problem: Find best sweet and savory dish combination that comes closest to target.
Solution: Sort dishes by type, use two pointers to find optimal pair.
Time: O(n log n), **Space**: O(n)
Key Insight: Separate sweet and savory dishes, use two-pointer technique.

2.2 Binary Search Tree Algorithms

2.2.1 Validate BST

Problem: Check if binary tree is valid Binary Search Tree.
Solution: Recursive validation with min/max bounds.
Time: O(n), **Space**: O(h)
Key Insight: Pass valid range to each subtree recursively.

2.2.2 BST Traversal

Problem: Perform different types of tree traversals.
Solution: Inorder (Left-Root-Right), Preorder (Root-Left-Right), Postorder (Left-Right-Root).
Time: O(n), **Space**: O(n)
Key Insight: Different traversal orders visit nodes in different sequences.

2.2.3 Min Height BST

Problem: Construct BST with minimum height from sorted array.
Solution: Use middle element as root, recursively build subtrees.
Time: O(n), **Space**: O(n)
Key Insight: Use middle element as root to ensure balanced tree.

2.2.4 Find Kth Largest Value in BST

Problem: Find kth largest value in Binary Search Tree.
Solution: Reverse inorder traversal, stop after k elements.
Time: O(h + k), **Space**: O(h)
Key Insight: Reverse inorder traversal gives values in descending order.

2.2.5 Reconstruct BST

Problem: Reconstruct BST from preorder traversal.
Solution: First element is root, find split point for left/right subtrees.
Time: O(n), **Space**: O(n)
Key Insight: Preorder gives root first, then left subtree, then right subtree.

2.2.6 Invert Binary Tree

Problem: Invert a binary tree (mirror image).
Solution: Recursively swap left and right children.
Time: O(n), **Space**: O(h)
Key Insight: Swap children at each node recursively.

2.2.7 Binary Tree Diameter

Problem: Find the diameter of a binary tree (longest path between any two nodes).
Solution: Calculate height and update diameter at each node.
Time: O(n), **Space**: O(h)
Key Insight: Diameter = max(left_height + right_height) for any node.

2.2.8 Find Successor

Problem: Find the in-order successor of a given node in a BST.
Solution: If right child exists, find leftmost node in right subtree. Otherwise, find closest ancestor where node is in left subtree.
Time: O(h), **Space**: O(1)
Key Insight: Successor is either leftmost node in right subtree or closest ancestor.

2.2.9 Height Balanced Binary Tree

Problem: Check if a binary tree is height-balanced.
Solution: Calculate height and check balance at each node.
Time: O(n), **Space**: O(h)
Key Insight: Tree is balanced if left and right subtrees are balanced and height difference ≤ 1.

2.2.10 Merge Binary Trees

Problem: Merge two binary trees by adding corresponding nodes.
Solution: Recursively merge corresponding nodes.

Time: O(min(n1, n2)), **Space**: O(min(h1, h2))
Key Insight: Add values of corresponding nodes, handle null nodes.

2.2.11 Symmetrical Tree

Problem: Check if a binary tree is symmetrical (mirror image of itself).
Solution: Compare left and right subtrees recursively.
Time: O(n), **Space**: O(h)
Key Insight: Tree is symmetrical if left and right subtrees are mirrored.

2.2.12 Split Binary Tree

Problem: Check if a binary tree can be split into two trees with equal sums.
Solution: Calculate total sum, check if any subtree has half the total sum.
Time: O(n), **Space**: O(h)
Key Insight: If total sum is odd, cannot split. Otherwise, look for subtree with half sum.

2.3 Dynamic Programming

2.3.1 Max Subset Sum No Adjacent

Problem: Find maximum sum of non-adjacent elements in array.
Solution: DP with formula: current = max(include, exclude).
Time: O(n), **Space**: O(1)
Key Insight: At each step, choose between including or excluding current element.

2.3.2 Number of Ways to Make Change

Problem: Find number of ways to make change for amount n using given denominations.
Solution: DP with formula: ways[amount] += ways[amount - denom].
Time: O(nd), **Space**: O(n)
Key Insight: Use DP to build up solutions from smaller amounts.

2.3.3 Min Number of Coins for Change

Problem: Find minimum number of coins needed to make change.
Solution: DP with formula: min(current, previous + 1).
Time: O(nd), **Space**: O(n)
Key Insight: Use DP to find minimum coins for each amount.

2.3.4 Levenshtein Distance

Problem: Calculate edit distance between two strings.
Solution: DP matrix with min(delete, insert, replace) operations.
Time: O(mn), **Space**: O(mn)
Key Insight: Use DP to find minimum operations to transform one string to another.

2.3.5 Number of Ways to Traverse Graph

Problem: Find number of ways to traverse graph from top-left to bottom-right.
Solution: DP with formula: dp[i][j] = dp[i-1][j] + dp[i][j-1].
Time: O(wh), **Space**: O(wh)
Key Insight: Each cell can be reached from above or left.

2.3.6 Kadane's Algorithm

Problem: Find the maximum subarray sum.
Solution: Keep track of current and global maximum.
Time: O(n), **Space**: O(1)
Key Insight: Reset current sum to current element if it becomes negative.

2.4 Graph Algorithms

2.4.1 Single Cycle Check

Problem: Check if single cycle exists in array where each element represents a jump.
Solution: Follow jumps, track visited nodes, check if back to start.
Time: O(n), **Space**: O(n)
Key Insight: Single cycle visits each node exactly once and returns to start.

2.4.2 River Sizes

Problem: Find sizes of all rivers in matrix where 1=water, 0=land.
Solution: DFS to find connected components of water cells.
Time: O(wh), **Space**: O(wh)
Key Insight: Use DFS to find connected components of water cells.

2.4.3 Remove Islands

Problem: Remove islands (1s not connected to border) from matrix.
Solution: Mark all 1s connected to border, remove unmarked 1s.
Time: O(wh), **Space**: O(wh)
Key Insight: Mark all 1s connected to border, then remove unmarked 1s.

2.4.4 Cycle in Graph

Problem: Check if cycle exists in directed graph.
Solution: DFS with recursion stack to detect back edges.
Time: O(V + E), **Space**: O(V)
Key Insight: Use recursion stack to detect back edges in DFS.

2.4.5 Minimum Passes of Matrix

Problem: Find minimum passes needed to convert all negative numbers to positive.
Solution: Simulate conversion process pass by pass.
Time: O(wh * passes), **Space**: O(wh)
Key Insight: Simulate the conversion process pass by pass.

2.4.6 Two Colorable

Problem: Check if a graph is two-colorable (bipartite).
Solution: DFS with color assignment, check for conflicts.
Time: O(V + E), **Space**: O(V)
Key Insight: Use DFS to assign colors and check for conflicts.

2.4.7 Task Assignment

Problem: Assign tasks to workers optimally.
Solution: Sort tasks, pair fastest with slowest.
Time: O(n log n), **Space**: O(n)
Key Insight: Pair fastest worker with slowest task to minimize total time.

2.4.8 Valid Starting City

Problem: Find a valid starting city for a circular route.
Solution: Track fuel remaining, reset when negative.
Time: O(n), **Space**: O(1)
Key Insight: Start from city where fuel remaining becomes negative.

2.4.9 Stable Internships

Problem: Find stable internship assignments using Gale-Shapley algorithm.
Solution: Use Gale-Shapley algorithm for stable matching.
Time: O(n^2), **Space**: O(n^2)
Key Insight: Use Gale-Shapley algorithm for stable matching.

2.5 Heap Algorithms

2.5.1 Min Heap Construction

Problem: Build a min heap from an array.
Solution: Sift down from last parent to root.
Time: O(n), **Space**: O(1)
Key Insight: Start from last parent and sift down each node.

2.5.2 Min Heap Operations

Problem: Implement min heap operations (insert, remove, peek).
Solution: Sift up for insert, sift down for remove.
Time: O(log n) per operation, **Space**: O(1)
Key Insight: Maintain heap property with sift up/down operations.

2.6 Linked List Algorithms

2.6.1 Remove Kth Node From End

Problem: Remove the kth node from the end of a linked list.
Solution: Use two pointers, fast pointer moves k steps ahead.
Time: O(n), **Space**: O(1)
Key Insight: Use two pointers to find kth node from end in one pass.

2.6.2 Sum of Linked Lists

Problem: Add two numbers represented as linked lists.
Solution: Add corresponding digits, handle carry.
Time: O(max(n, m)), **Space**: O(max(n, m))
Key Insight: Add digits from right to left, handle carry.

2.6.3 Merging Linked Lists

Problem: Find the intersection point of two linked lists.
Solution: Find lengths, align pointers, find intersection.
Time: O(n + m), **Space**: O(1)
Key Insight: Align pointers by adjusting for length difference.

2.7 Recursion and Backtracking

2.7.1 Permutations

Problem: Generate all permutations of an array.
Solution: Recursive approach, choose each element as first.
Time: O(n!), **Space**: O(n!)
Key Insight: Use recursion to build permutations by choosing each element as first.

2.7.2 Powerset

Problem: Generate all subsets (powerset) of an array.
Solution: Backtracking approach, include/exclude each element.
Time: O(2^n), **Space**: O(2^n)
Key Insight: Use backtracking to explore all possible combinations.

2.7.3 Phone Number Mnemonics

Problem: Generate all possible mnemonics for a phone number.
Solution: Backtracking with digit-to-letter mapping.
Time: O(4^n * n), **Space**: O(4^n * n)
Key Insight: Use backtracking to explore all letter combinations for each digit.

2.7.4 Staircase Traversal

Problem: Find number of ways to climb staircase with given height and max steps.
Solution: Recursive approach, try all possible step sizes.
Time: O(k^n), **Space**: O(n)
Key Insight: Use recursion to explore all possible step combinations.

2.7.5 Blackjack Probability

Problem: Calculate probability of reaching target in blackjack.
Solution: Recursive probability calculation with memoization.
Time: O(target), **Space**: O(target)
Key Insight: Use recursion with memoization to calculate probabilities.

2.7.6 Reveal Minesweeper

Problem: Reveal cells in minesweeper board starting from given position.
Solution: BFS to reveal connected safe cells.
Time: O(wh), **Space**: O(wh)
Key Insight: Use BFS to reveal all connected safe cells.

2.7.7 Search in Sorted Matrix

Problem: Search for target value in sorted matrix.
Solution: Start from top-right, eliminate half the search space each step.
Time: O(n + m), **Space**: O(1)
Key Insight: Use matrix properties to eliminate half the search space each step.

2.7.8 Three Number Sort

Problem: Sort array containing only three distinct values according to given order.
Solution: Three pointers to partition array in one pass.
Time: O(n), **Space**: O(1)
Key Insight: Use three pointers to partition array in one pass.

2.8 Stack Algorithms

2.8.1 Min Max Stack

Problem: Implement stack that tracks minimum and maximum values.
Solution: Use auxiliary stack to track min/max at each push.
Time: O(1) all operations, **Space**: O(n)
Key Insight: Maintain auxiliary stack with min/max information.

2.8.2 Balanced Brackets

Problem: Check if a string has balanced brackets.
Solution: Use stack to match opening and closing brackets.
Time: O(n), **Space**: O(n)
Key Insight: Use stack to match opening and closing brackets.

2.8.3 Sunset Views

Problem: Find buildings that have a sunset view.
Solution: Use stack to track buildings with sunset view.
Time: O(n), **Space**: O(n)
Key Insight: Use stack to track buildings with sunset view.

2.8.4 Best Digits

Problem: Find the best digits by removing k digits to get the largest possible number.
Solution: Use stack to maintain largest digits.
Time: O(n), **Space**: O(n)
Key Insight: Use stack to maintain largest digits.

2.8.5 Sort Stack

Problem: Sort a stack using only stack operations.
Solution: Use temporary stack to sort elements.
Time: $O(n^2)$, **Space**: O(n)
Key Insight: Use temporary stack to sort elements.

2.8.6 Next Greater Element

Problem: Find the next greater element for each element in the array.
Solution: Use stack to find next greater element.
Time: O(n), **Space**: O(n)
Key Insight: Use stack to find next greater element.

2.8.7 Reverse Polish Notation

Problem: Evaluate a reverse polish notation expression.
Solution: Use stack to evaluate postfix expression.
Time: O(n), **Space**: O(n)
Key Insight: Use stack to evaluate postfix expression.

2.8.8 Colliding Asteroids

Problem: Simulate asteroid collisions.
Solution: Use stack to simulate collisions.
Time: O(n), **Space**: O(n)
Key Insight: Use stack to simulate collisions.

2.9 String Algorithms

2.9.1 Longest Palindromic Substring

Problem: Find the longest palindromic substring in a string.
Solution: Expand around center for each character.
Time: $O(n^2)$, **Space**: O(1)
Key Insight: Expand around center for each character.

2.9.2 Group Anagrams

Problem: Group words that are anagrams of each other.
Solution: Use sorted word as key in hash map.
Time: O(n * k * log k), **Space**: O(n * k)
Key Insight: Use sorted word as key in hash map.

2.9.3 Valid IP Addresses

Problem: Generate all valid IP addresses from a string of digits.
Solution: Backtracking with IP validation.
Time: O(1), **Space**: O(1)
Key Insight: Use backtracking with IP validation.

2.9.4 Reverse Words in String

Problem: Reverse the order of words in a string.
Solution: Split by spaces and reverse.
Time: O(n), **Space**: O(n)
Key Insight: Split by spaces and reverse.

2.9.5 Minimum Characters for Words

Problem: Find minimum characters needed to write all words.
Solution: Find maximum count of each character across all words.
Time: O(n * k), **Space**: O(c)
Key Insight: Find maximum count of each character across all words.

2.9.6 One Edit

Problem: Check if two strings are one edit away from each other.
Solution: Compare strings character by character.
Time: O(n), **Space**: O(1)
Key Insight: Compare strings character by character.

2.10 Trie Algorithms

2.10.1 Suffix Trie Construction

Problem: Build a suffix trie from a string.
Solution: Insert all suffixes into trie.
Time: $O(n^2)$, **Space**: $O(n^2)$
Key Insight: Insert all suffixes into trie.

2.10.2 Suffix Trie Search

Problem: Check if string is contained in the trie.
Solution: Traverse trie following string characters.
Time: O(m), **Space**: O(1)
Key Insight: Traverse trie following string characters.

Chapter 3

Advanced Problems

3.1 Array and Matrix Problems

3.1.1 Four Number Sum

Problem: Find all quadruplets in array that sum to target value.
Solution: Use hash table to store pairs and their sums, find complements.
Time: O(n^2) average, O(n^3) worst, **Space**: O(n^2)
Key Insight: Store pair sums in hash table, look for complements to form quadruplets.

3.1.2 Subarray Sort

Problem: Find smallest subarray that needs sorting to make entire array sorted.
Solution: Find min/max values out of order, determine boundaries.
Time: O(n), **Space**: O(1)
Key Insight: Elements out of order determine the boundaries of subarray to sort.

3.1.3 Largest Range

Problem: Find largest range of consecutive integers in array.
Solution: Use hash table to track visited numbers, expand from each number.
Time: O(n), **Space**: O(n)
Key Insight: Use hash table for O(1) lookups, expand left and right from each number.

3.1.4 Min Rewards

Problem: Distribute minimum rewards based on scores (higher scores get more rewards).
Solution: Two-pass algorithm - left to right, then right to left.
Time: O(n), **Space**: O(n)
Key Insight: Two passes ensure all constraints are satisfied.

3.1.5 Zigzag Traverse

Problem: Traverse 2D array in zigzag pattern.
Solution: Track direction and handle edge cases at boundaries.
Time: O(n), **Space**: O(n)
Key Insight: Change direction when hitting array boundaries.

3.1.6 Longest Subarray with Sum

Problem: Find longest subarray that sums to target.
Solution: Use hash table to store cumulative sums.
Time: O(n), **Space**: O(n)
Key Insight: If sum[i] - sum[j] = target, subarray j+1 to i sums to target.

3.1.7 Count Squares

Problem: Count number of squares of 1s in binary matrix.
Solution: DP - each cell represents size of largest square ending there.
Time: O(m*n), **Space**: O(m*n)
Key Insight: DP[i][j] = min(DP[i-1][j], DP[i][j-1], DP[i-1][j-1]) + 1.

3.1.8 Maximum Sum Submatrix

Problem: Find maximum sum of submatrix of given size.
Solution: 2D sliding window with prefix sums.
Time: O(m*n), **Space**: O(m*n)
Key Insight: Use prefix sums to calculate submatrix sums in O(1).

3.1.9 Largest Rectangle Under Skyline

Problem: Find area of largest rectangle under skyline.
Solution: Stack-based approach to track increasing heights.
Time: O(n), **Space**: O(n)
Key Insight: Use stack to maintain increasing heights, calculate area when popping.

3.2 Tree and BST Problems

3.2.1 Same BSTs

Problem: Check if two arrays represent same BST.
Solution: Recursive comparison with BST property.
Time: O(n^2) worst, **Space**: O(d)
Key Insight: Compare roots, then recursively compare left and right subtrees.

3.2.2 Validate Three Nodes

Problem: Check if node_two is descendant of node_one and node_three is descendant of node_two.
Solution: Check both directions of the relationship.
Time: O(h), **Space**: O(1)
Key Insight: Search from both nodes to find the relationship.

3.2.3 Repair BST

Problem: Repair BST where exactly two nodes have been swapped.
Solution: Inorder traversal to find swapped nodes.
Time: O(n), **Space**: O(h)
Key Insight: Inorder traversal reveals swapped nodes in sorted sequence.

3.2.4 Sum BSTs

Problem: Calculate sum of all values in BST.
Solution: Recursive traversal.
Time: O(n), **Space**: O(h)
Key Insight: Simple recursive sum of current node and subtrees.

3.2.5 Max Path Sum in Binary Tree

Problem: Find maximum path sum in binary tree.
Solution: Recursive DFS with path tracking.
Time: O(n), **Space**: O(h)
Key Insight: At each node, consider path through node or max of left/right subtree.

3.2.6 Find Nodes Distance K

Problem: Find all nodes at distance k from target node.
Solution: BFS from target with distance tracking.
Time: O(n), **Space**: O(n)
Key Insight: Build parent map, use BFS to explore all directions.

3.3 Dynamic Programming

3.3.1 Max Sum Increasing Subsequence

Problem: Find maximum sum of increasing subsequence.
Solution: DP with binary search optimization.
Time: O(n log n), **Space**: O(n)
Key Insight: Use binary search to find optimal position for each element.

3.3.2 Longest Common Subsequence

Problem: Find length of longest common subsequence between two strings.
Solution: Dynamic programming matrix.
Time: O(m*n), **Space**: O(m*n)
Key Insight: DP[i][j] = max(DP[i-1][j], DP[i][j-1]) or DP[i-1][j-1] + 1 if match.

3.3.3 Min Number of Jumps

Problem: Find minimum jumps needed to reach end of array.
Solution: Greedy algorithm with reach tracking.
Time: O(n), **Space**: O(1)
Key Insight: Track current reach and max reach, jump when current reach is exhausted.

3.3.4 Knapsack Problem

Problem: Solve 0/1 knapsack problem.
Solution: Dynamic programming.
Time: O(n*capacity), **Space**: O(n*capacity)
Key Insight: DP[i][w] = max(DP[i-1][w], DP[i-1][w-weight[i]] + value[i]).

3.3.5 Disk Stacking

Problem: Find maximum height by stacking disks.
Solution: DP with sorting.
Time: $O(n^2)$, **Space**: O(n)
Key Insight: Sort by dimensions, use DP to find optimal stacking.

3.3.6 Numbers in Pi

Problem: Find minimum spaces needed to separate pi into valid numbers.
Solution: DP with memoization.
Time: $O(n^3 + m)$, **Space**: O(n + m)
Key Insight: Try all possible prefixes, memoize results.

3.3.7 Maximize Expression

Problem: Maximize A - B + C - D where A,B,C,D are array elements.
Solution: DP with state tracking.
Time: O(n), **Space**: O(1)
Key Insight: Track state of expression building with DP.

3.3.8 Dice Throws

Problem: Count ways to roll n dice to get total sum.
Solution: Dynamic programming.
Time: O(n*total*faces), **Space**: O(n*total)
Key Insight: DP[i][j] = sum of DP[i-1][j-k] for all valid k.

3.3.9 Juice Bottling

Problem: Find optimal way to bottle juice to maximize profit.
Solution: Dynamic programming.
Time: O(n^2), **Space**: O(n)
Key Insight: Try all possible cuts, use DP to find optimal solution.

3.4 Graph Algorithms

3.4.1 Knight Connection

Problem: Find minimum moves for knight to reach from position a to b.
Solution: BFS with chess board constraints.
Time: O(1), **Space**: O(1)
Key Insight: Fixed board size makes this O(1), use BFS for shortest path.

3.4.2 Dijkstra Algorithm

Problem: Find shortest paths from start vertex to all other vertices.
Solution: Dijkstra's algorithm with priority queue.
Time: O((V + E) log V), **Space**: O(V)
Key Insight: Use priority queue to always process closest unvisited vertex.

3.4.3 Topological Sort

Problem: Perform topological sort on directed acyclic graph.
Solution: DFS with cycle detection.
Time: O(V + E), **Space**: O(V)
Key Insight: Use DFS with visiting/visited states to detect cycles.

3.4.4 Kruskal Algorithm

Problem: Find minimum spanning tree using Kruskal's algorithm.
Solution: Union-Find with sorting.
Time: O(E log E), **Space**: O(V)
Key Insight: Sort edges, use Union-Find to avoid cycles.

3.4.5 Prim Algorithm

Problem: Find minimum spanning tree using Prim's algorithm.
Solution: Priority queue with adjacency list.
Time: O(E log V), **Space**: O(V)
Key Insight: Use priority queue to always add minimum weight edge.

3.4.6 Boggle Board

Problem: Find all words from dictionary that can be formed on boggle board.
Solution: Trie + DFS.
Time: O(nm * 8^s), **Space**: O(w*s)
Key Insight: Build trie from words, use DFS to explore all paths.

3.4.7 Largest Island

Problem: Find size of largest island in binary matrix.
Solution: DFS with area calculation.
Time: O(m*n), **Space**: O(m*n)
Key Insight: Use DFS to explore connected components, track maximum area.

3.5 Linked List Problems

3.5.1 Continuous Median

Problem: Maintain running median of stream of numbers.
Solution: Two heaps - max heap for lower half, min heap for upper half.
Time: O(log n) per insertion, **Space**: O(n)
Key Insight: Keep heaps balanced, median is max of lower heap or average.

3.5.2 Find Loop

Problem: Detect cycle in linked list and return cycle start node.
Solution: Floyd's Cycle-Finding Algorithm (Tortoise and Hare).
Time: O(n), **Space**: O(1)
Key Insight: Use two pointers, find meeting point, then find cycle start.

3.5.3 Reverse Linked List

Problem: Reverse linked list iteratively.
Solution: Three pointers (prev, curr, next).
Time: O(n), **Space**: O(1)
Key Insight: Update pointers in reverse direction.

3.5.4 Merge Linked Lists

Problem: Merge two sorted linked lists.
Solution: Compare and link nodes.
Time: O(n + m), **Space**: O(1)
Key Insight: Compare heads, link smaller one, advance pointer.

3.5.5 Shift Linked List

Problem: Shift linked list by k positions.
Solution: Find length, adjust k, find new head and tail.
Time: O(n), **Space**: O(1)
Key Insight: Normalize k, find new head by traversing length-k positions.

3.5.6 Lowest Common Manager

Problem: Find lowest common manager of two reports in organizational hierarchy.
Solution: DFS with return values.
Time: O(n), **Space**: O(h)
Key Insight: Return number of reports found, return node when count reaches 2.

3.6 String Problems

3.6.1 Interweaving Strings

Problem: Check if string three can be formed by interweaving strings one and two.
Solution: DP with memoization.
Time: O(n*m), **Space**: O(n*m)
Key Insight: Try matching from both strings, memoize results.

3.6.2 Solve Sudoku

Problem: Solve 9x9 Sudoku puzzle.
Solution: Backtracking with constraint checking.
Time: O(9^(n²)), **Space**: O(n²)
Key Insight: Try all numbers 1-9, check row/column/box constraints.

3.6.3 Generate Div Tags

Problem: Generate all valid combinations of opening and closing div tags.
Solution: Backtracking with balance tracking.
Time: O(4^n / sqrt(n)), **Space**: O(n)
Key Insight: Track open/close count, ensure balance.

3.6.4 Ambiguous Measurements

Problem: Check if target range can be measured using given measuring cups.
Solution: DP with memoization.
Time: O(n * (high - low)), **Space**: O(high - low)
Key Insight: Try all cup combinations, memoize results.

3.6.5 Longest Substring Without Duplication

Problem: Find longest substring without duplicate characters.
Solution: Sliding window with hash set.
Time: O(n), **Space**: O(min(m, n))
Key Insight: Expand window, shrink when duplicate found.

3.6.6 Underscorify Substring

Problem: Add underscores around all occurrences of substring in string.
Solution: Find all occurrences and merge overlapping ones.
Time: O(n*m), **Space**: O(n)
Key Insight: Find locations, collapse overlapping, insert underscores.

3.6.7 Pattern Matcher

Problem: Check if string matches pattern where x and y can be any strings.
Solution: Try different combinations of x and y.
Time: $O(n^2)$, **Space**: O(n)
Key Insight: Try different lengths for x and y, verify pattern.

3.6.8 Multi String Search

Problem: Find which small strings are contained in big string.
Solution: Build suffix trie and search.
Time: $O(b^2 + ns)$, **Space**: $O(b^2 + ns)$
Key Insight: Build trie of all suffixes, search each small string.

3.6.9 Longest Most Frequent Prefix

Problem: Find longest prefix that appears in most strings.
Solution: Build prefix trie and count occurrences.
Time: O(n*s), **Space**: O(n*s)
Key Insight: Count prefix occurrences, find longest with majority.

3.6.10 Shortest Unique Prefixes

Problem: Find shortest unique prefix for each string.
Solution: Build prefix trie and find unique prefixes.
Time: O(n*s), **Space**: O(n*s)
Key Insight: Stop when count becomes 1 for unique prefix.

3.7 Sorting and Searching

3.7.1 Sort K Sorted Array

Problem: Sort array where each element is at most k positions away from sorted position.
Solution: Use min heap to maintain k+1 elements.
Time: O(n log k), **Space**: O(k)
Key Insight: Use heap to maintain sorted window of k+1 elements.

3.7.2 Shifted Binary Search

Problem: Search for target in shifted sorted array.
Solution: Modified binary search.
Time: O(log n), **Space**: O(1)
Key Insight: Check if left or right half is sorted, adjust search accordingly.

3.7.3 Search for Range

Problem: Find first and last occurrence of target in sorted array.
Solution: Binary search for first and last occurrence.
Time: O(log n), **Space**: O(1)
Key Insight: Use binary search twice - once for first, once for last.

3.7.4 Quickselect

Problem: Find kth smallest element in unsorted array.
Solution: Quickselect algorithm.
Time: O(n) average, O(n^2) worst, **Space**: O(1)
Key Insight: Use partition from quicksort, recurse on relevant half.

3.7.5 Index Equals Value

Problem: Find first index where array[index] == index.
Solution: Binary search.
Time: O(log n), **Space**: O(1)
Key Insight: Use binary search, adjust based on comparison.

3.7.6 Quick Sort

Problem: Sort array using quicksort algorithm.
Solution: Divide and conquer with pivot selection.
Time: O(n log n) average, O(n^2) worst, **Space**: O(log n)
Key Insight: Choose pivot, partition around it, recurse on halves.

3.7.7 Heap Sort

Problem: Sort array using heapsort algorithm.
Solution: Build max heap and extract elements.
Time: O(n log n), **Space**: O(1)
Key Insight: Build max heap, repeatedly extract maximum.

3.7.8 Radix Sort

Problem: Sort array using radix sort algorithm.
Solution: Sort by each digit from least to most significant.
Time: O(d * (n + k)), **Space**: O(n + k)
Key Insight: Use counting sort for each digit position.

Chapter 4

Very Advanced Problems

4.1 Optimization and Resource Allocation

4.1.1 Apartment Hunting

Problem: Find optimal block to live in that minimizes maximum distance to all required amenities.
Solution: Calculate minimum distances for each requirement, find block with minimum maximum distance.
Time: O(B * R), **Space**: O(B * R)
Key Insight: Precompute distances for each requirement, then find minimum of maximums.

4.1.2 Calendar Matching

Problem: Find available time slots for meeting between two people given their calendars.
Solution: Merge calendars, find gaps between meetings that fit meeting duration.
Time: O(C1 + C2), **Space**: O(C1 + C2)
Key Insight: Convert times to minutes, merge overlapping meetings, find available gaps.

4.1.3 Waterfall Streams

Problem: Calculate percentage of water that reaches each position at bottom of 2D waterfall.
Solution: DP approach, track water flow and splitting at blocks.
Time: O(W * H), **Space**: O(W * H)
Key Insight: Use DP to track water percentages, split water when blocked.

4.1.4 Minimum Area Rectangle

Problem: Find minimum area of rectangle that can be formed by any four points.
Solution: Try all pairs as diagonal corners, check if other corners exist.
Time: O(N^2), **Space**: O(N)
Key Insight: Use hash set for O(1) corner existence check.

4.1.5 Line Through Points

Problem: Find maximum number of points that can be placed on single straight line.
Solution: Calculate slopes between all point pairs, find most common slope.
Time: O(N^2), **Space**: O(N)
Key Insight: Use GCD to normalize slopes, count most frequent slope.

4.1.6 Right Smaller Than

Problem: For each element, count how many elements to its right are smaller.
Solution: Merge sort with inversion counting.
Time: O(N log N), **Space**: O(N)
Key Insight: Use merge sort to count inversions efficiently.

4.2 Tree and Graph Algorithms

4.2.1 Iterative Inorder Traversal

Problem: Perform in-order traversal of binary tree using iteration.
Solution: Use stack to simulate recursion.
Time: O(N), **Space**: O(H)
Key Insight: Use stack to maintain path to current node.

4.2.2 Flatten Binary Tree

Problem: Flatten binary tree into linked list in pre-order traversal order.
Solution: Recursive approach, connect left subtree between current and right.
Time: O(N), **Space**: O(H)
Key Insight: Recursively flatten subtrees, insert left between current and right.

4.2.3 Right Sibling Tree

Problem: Connect each node to its right sibling at same level.
Solution: Level-by-level traversal, connect nodes at each level.
Time: O(N), **Space**: O(1)
Key Insight: Process each level, connect children to siblings.

4.2.4 All Kinds of Node Depths

Problem: Calculate sum of depths of all nodes in binary tree.
Solution: Recursive traversal, accumulate depths.
Time: O(N), **Space**: O(H)
Key Insight: Each node's depth = parent's depth + 1.

4.2.5 Compare Leaf Traversal

Problem: Compare leaf traversal of two binary trees.
Solution: Get leaf sequences using in-order traversal, compare.
Time: O(N1 + N2), **Space**: O(H1 + H2)
Key Insight: In-order traversal gives leaf sequence in order.

4.2.6 Airport Connections

Problem: Find minimum new routes needed to make all airports reachable from starting airport.
Solution: Find strongly connected components, count unreachable ones.
Time: O(A + R), **Space**: O(A + R)
Key Insight: Use Kosaraju's algorithm to find SCCs.

4.2.7 Two-Edge-Connected Graph

Problem: Check if graph is 2-edge-connected (removing any edge doesn't disconnect).
Solution: Find bridges using Tarjan's algorithm.
Time: O(V + E), **Space**: O(V + E)
Key Insight: Graph is 2-edge-connected if it has no bridges.

4.3 Dynamic Programming and Optimization

4.3.1 Max Profit With K Transactions

Problem: Find maximum profit with at most k transactions.
Solution: DP with state tracking for transactions and days.
Time: O(N * K), **Space**: O(N * K)
Key Insight: Track max profit before buying for each transaction.

4.3.2 Palindrome Partitioning Min Cuts

Problem: Find minimum cuts needed to partition string into palindromes.
Solution: Build palindrome table, use DP to find minimum cuts.
Time: O(N^2), **Space**: O(N^2)
Key Insight: Precompute all palindromes, then find minimum cuts.

4.3.3 Longest Increasing Subsequence

Problem: Find length of longest strictly increasing subsequence.
Solution: Patience sorting algorithm with binary search.
Time: O(N log N), **Space**: O(N)
Key Insight: Use binary search to find optimal position for each element.

4.3.4 Longest String Chain

Problem: Find length of longest word chain where each word is formed by adding one letter.
Solution: Sort by length, use DP to find longest chain.
Time: O(N * L^2), **Space**: O(N)
Key Insight: Try removing each character to find predecessor.

4.3.5 Square of Zeroes

Problem: Find largest square of zeroes in binary matrix.
Solution: DP to track consecutive zeros right and below.
Time: O(N^3), **Space**: O(N^2)
Key Insight: Precompute consecutive zeros, then check square formation.

4.3.6 Number of Binary Tree Topologies

Problem: Calculate number of different binary tree topologies with n nodes.
Solution: DP with Catalan number pattern.
Time: O(N^2), **Space**: O(N)
Key Insight: Use Catalan number formula: C(n) = sum(C(i) * C(n-1-i)).

4.3.7 Non-Attacking Queens

Problem: Find number of ways to place n queens on n×n board without attacks.
Solution: Backtracking with pruning.
Time: O(N!), **Space**: O(N)
Key Insight: Use backtracking with row, column, and diagonal checks.

4.4 Advanced String Algorithms

4.4.1 Knuth-Morris-Pratt Algorithm

Problem: Find all occurrences of substring in string using KMP algorithm.
Solution: Build failure function, use it to skip unnecessary comparisons.
Time: O(N + M), **Space**: O(M)
Key Insight: Use failure function to avoid recomparing known matches.

4.4.2 Smallest Substring Containing

Problem: Find smallest substring containing all characters from small string.
Solution: Sliding window with character counting.
Time: O(N + M), **Space**: O(M)
Key Insight: Use sliding window to find minimum valid substring.

4.4.3 Longest Balanced Substring

Problem: Find length of longest balanced substring (equal brackets).
Solution: Stack-based approach with index tracking.
Time: O(N), **Space**: O(N)
Key Insight: Use stack to track unmatched opening brackets.

4.4.4 Strings Made Up Of Strings

Problem: Check if big string can be constructed by concatenating small strings.
Solution: DP with string matching.
Time: O(N * M), **Space**: O(N)
Key Insight: Use DP to track if each position can be reached.

4.5 Graph and Pathfinding Algorithms

4.5.1 A* Algorithm

Problem: Find shortest path from start to goal in grid using A* search.
Solution: Priority queue with heuristic function.
Time: O(N log N), **Space**: O(N)
Key Insight: Use Manhattan distance heuristic with priority queue.

4.5.2 Rectangle Mania

Problem: Count number of rectangles that can be formed by given coordinates.
Solution: Try all pairs as diagonal corners, check if other corners exist.
Time: $O(N^2)$, **Space**: O(N)
Key Insight: Use hash set for O(1) corner existence check.

4.5.3 Detect Arbitrage

Problem: Detect arbitrage opportunity in currency exchange rate matrix.
Solution: Convert to logarithms, use Floyd-Warshall to find negative cycles.
Time: $O(N^3)$, **Space**: $O(N^2)$
Key Insight: Convert multiplication to addition using logarithms.

4.6 Advanced Data Structures

4.6.1 LRU Cache

Problem: Implement Least Recently Used cache with O(1) operations.
Solution: Hash map with doubly linked list.
Time: O(1), **Space**: O(C)
Key Insight: Use hash map for O(1) access, doubly linked list for ordering.

4.6.2 Continuous Median

Problem: Maintain running median of stream of numbers.
Solution: Two heaps - max heap for lower half, min heap for upper half.
Time: O(log N) per insertion, **Space**: O(N)
Key Insight: Keep heaps balanced, median is max of lower heap or average.

4.6.3 Merge Sorted Arrays

Problem: Merge multiple sorted arrays into single sorted array.
Solution: Min heap to merge arrays efficiently.
Time: O(N log K), **Space**: O(N)
Key Insight: Use heap to always process smallest element.

4.7 Linked List Algorithms

4.7.1 Rearrange Linked List

Problem: Rearrange list so all nodes ¡ k come before = k, which come before ¿ k.
Solution: Create three separate lists, then connect them.
Time: O(N), **Space**: O(1)
Key Insight: Use three pointers to maintain separate lists.

4.7.2 Linked List Palindrome

Problem: Check if linked list is palindrome.
Solution: Find middle, reverse second half, compare, restore.
Time: O(N), **Space**: O(1)
Key Insight: Use fast/slow pointers to find middle, reverse second half.

4.7.3 Zip Linked List

Problem: Rearrange list in zip pattern: first, last, second, second-to-last, etc.
Solution: Find middle, reverse second half, merge in zip pattern.
Time: O(N), **Space**: O(1)
Key Insight: Reverse second half, then merge alternating nodes.

4.7.4 Node Swap

Problem: Swap every two adjacent nodes in linked list.
Solution: Iterative approach with three pointers.
Time: O(N), **Space**: O(1)
Key Insight: Use dummy node to handle head case, swap pairs.

4.8 Advanced Sorting and Searching

4.8.1 Median of Two Sorted Arrays

Problem: Find median of two sorted arrays.
Solution: Binary search on partition points.
Time: O(log(min(M, N))), **Space**: O(1)
Key Insight: Use binary search to find correct partition.

4.8.2 Optimal Assembly Line

Problem: Assign tasks to workers to minimize maximum time any worker spends.
Solution: Binary search on maximum time with feasibility check.
Time: O(N * log(SUM)), **Space**: O(1)
Key Insight: Use binary search to find minimum feasible maximum time.

4.8.3 Merge Sort

Problem: Sort array using merge sort algorithm.
Solution: Divide and conquer with merging.
Time: O(N log N), **Space**: O(N)
Key Insight: Recursively sort halves, then merge sorted halves.

4.8.4 Count Inversions

Problem: Count number of inversions in array.
Solution: Merge sort with inversion counting.
Time: O(N log N), **Space**: O(N)
Key Insight: Count inversions during merge step of merge sort.

4.9 Geometric and Matrix Algorithms

4.9.1 Largest Park

Problem: Find largest rectangular area for park surrounded by buildings.
Solution: Largest rectangle in histogram for each row.
Time: O(R * C), **Space**: O(C)
Key Insight: Use histogram approach for each row.

4.9.2 Water Area

Problem: Calculate total water area trapped between bars.
Solution: Two-pointer approach with left/right max tracking.
Time: O(N), **Space**: O(1)
Key Insight: Water trapped = min(left_max, right_max) - height.

4.9.3 Laptop Rentals

Problem: Find minimum number of laptops needed for rental times.
Solution: Sort start/end times, use two pointers.
Time: O(N log N), **Space**: O(N)
Key Insight: Track overlapping intervals, count maximum simultaneous rentals.

4.9.4 Shorten Path

Problem: Shorten Unix-style file path by resolving . and .. components.
Solution: Stack-based approach.
Time: O(N), **Space**: O(N)
Key Insight: Use stack to handle directory navigation.

4.10 Advanced Optimization Problems

4.10.1 Max Sum Increasing Subsequence

Problem: Find maximum sum of increasing subsequence.
Solution: DP with binary search optimization.
Time: O(N log N), **Space**: O(N)
Key Insight: Use binary search to find optimal position for each element.

4.10.2 Longest Common Subsequence

Problem: Find length of longest common subsequence between two strings.
Solution: Dynamic programming matrix.
Time: O(M * N), **Space**: O(M * N)
Key Insight: DP[i][j] = max(DP[i-1][j], DP[i][j-1]) or DP[i-1][j-1] + 1 if match.

4.10.3 Min Number of Jumps

Problem: Find minimum jumps needed to reach end of array.
Solution: Greedy algorithm with reach tracking.
Time: O(N), **Space**: O(1)
Key Insight: Track current reach and max reach, jump when current reach is exhausted.

4.10.4 Knapsack Problem

Problem: Solve 0/1 knapsack problem.
Solution: Dynamic programming.
Time: O(N * capacity), **Space**: O(N * capacity)
Key Insight: DP[i][w] = max(DP[i-1][w], DP[i-1][w-weight[i]] + value[i]).

4.10.5 Disk Stacking

Problem: Find maximum height by stacking disks.
Solution: DP with sorting.
Time: $O(N^2)$, **Space**: O(N)
Key Insight: Sort by dimensions, use DP to find optimal stacking.

4.10.6 Numbers in Pi

Problem: Find minimum spaces needed to separate pi into valid numbers.
Solution: DP with memoization.
Time: $O(N^3 + M)$, **Space**: O(N + M)
Key Insight: Try all possible prefixes, memoize results.

4.10.7 Maximize Expression

Problem: Maximize A - B + C - D where A,B,C,D are array elements.
Solution: DP with state tracking.
Time: O(N), **Space**: O(1)
Key Insight: Track state of expression building with DP.

4.10.8 Dice Throws

Problem: Count ways to roll n dice to get total sum.
Solution: Dynamic programming.
Time: O(N * total * faces), **Space**: O(N * total)
Key Insight: DP[i][j] = sum of DP[i-1][j-k] for all valid k.

4.10.9 Juice Bottling

Problem: Find optimal way to bottle juice to maximize profit.
Solution: Dynamic programming.
Time: $O(N^2)$, **Space**: O(N)
Key Insight: Try all possible cuts, use DP to find optimal solution.

4.11 Advanced Tree Problems

4.11.1 Same BSTs

Problem: Check if two arrays represent same BST.
Solution: Recursive comparison with BST property.
Time: $O(N^2)$ worst, **Space**: O(D)
Key Insight: Compare roots, then recursively compare left and right subtrees.

4.11.2 Validate Three Nodes

Problem: Check if node_two is descendant of node_one and node_three is descendant of node_two.
Solution: Check both directions of the relationship.
Time: O(H), **Space**: O(1)
Key Insight: Search from both nodes to find the relationship.

4.11.3 Repair BST

Problem: Repair BST where exactly two nodes have been swapped.
Solution: Inorder traversal to find swapped nodes.
Time: O(N), **Space**: O(H)
Key Insight: Inorder traversal reveals swapped nodes in sorted sequence.

4.11.4 Sum BSTs

Problem: Calculate sum of all values in BST.
Solution: Recursive traversal.
Time: O(N), **Space**: O(H)
Key Insight: Simple recursive sum of current node and subtrees.

4.11.5 Max Path Sum in Binary Tree

Problem: Find maximum path sum in binary tree.
Solution: Recursive DFS with path tracking.
Time: O(N), **Space**: O(H)
Key Insight: At each node, consider path through node or max of left/right subtree.

4.11.6 Find Nodes Distance K

Problem: Find all nodes at distance k from target node.
Solution: BFS from target with distance tracking.
Time: O(N), **Space**: O(N)
Key Insight: Build parent map, use BFS to explore all directions.

4.12 Advanced Matrix and Array Problems

4.12.1 Count Squares

Problem: Count number of squares of 1s in binary matrix.
Solution: DP - each cell represents size of largest square ending there.
Time: O(M * N), **Space**: O(M * N)
Key Insight: DP[i][j] = min(DP[i-1][j], DP[i][j-1], DP[i-1][j-1]) + 1.

4.12.2 Maximum Sum Submatrix

Problem: Find maximum sum of submatrix of given size.
Solution: 2D sliding window with prefix sums.
Time: O(M * N), **Space**: O(M * N)
Key Insight: Use prefix sums to calculate submatrix sums in O(1).

4.12.3 Largest Rectangle Under Skyline

Problem: Find area of largest rectangle under skyline.
Solution: Stack-based approach to track increasing heights.
Time: O(N), **Space**: O(N)
Key Insight: Use stack to maintain increasing heights, calculate area when popping.

4.12.4 Longest Subarray with Sum

Problem: Find longest subarray that sums to target.
Solution: Use hash table to store cumulative sums.
Time: O(N), **Space**: O(N)
Key Insight: If sum[i] - sum[j] = target, subarray j+1 to i sums to target.

4.12.5 Knight Connection

Problem: Find minimum moves for knight to reach from position a to b.
Solution: BFS with chess board constraints.
Time: O(1), **Space**: O(1)
Key Insight: Fixed board size makes this O(1), use BFS for shortest path.

4.13 Advanced String and Pattern Matching

4.13.1 Interweaving Strings

Problem: Check if string three can be formed by interweaving strings one and two.
Solution: DP with memoization.
Time: O(N * M), **Space**: O(N * M)
Key Insight: Try matching from both strings, memoize results.

4.13.2 Solve Sudoku

Problem: Solve 9×9 Sudoku puzzle.
Solution: Backtracking with constraint checking.
Time: O(9^(N²)), **Space**: O(N²)
Key Insight: Try all numbers 1-9, check row/column/box constraints.

4.13.3 Generate Div Tags

Problem: Generate all valid combinations of opening and closing div tags.
Solution: Backtracking with balance tracking.
Time: O(4^N / sqrt(N)), **Space**: O(N)
Key Insight: Track open/close count, ensure balance.

4.13.4 Ambiguous Measurements

Problem: Check if target range can be measured using given measuring cups.
Solution: DP with memoization.
Time: O(N * (high - low)), **Space**: O(high - low)
Key Insight: Try all cup combinations, memoize results.