Learn

Data structures and Algorithms

in JAVA

**Mannu Kumar Pandey**

Chapter 0: Introduction

Chapter 1: Arrays

Chapter 2: Strings

Chapter 3: Linked List

Chapter 4: Stacks

Chapter 5: Queue

Chapter 6: Hashing, HashSet, HashMap

Chapter 7: Searching Algorithms

Chapter 8: Sorting Algorithms

Chapter 9: Greedy Algorithms

Chapter 10: Recursion

Chapter 11: Backtracking

Chapter 12: Dynamic Programming

Chapter 13: Bit Manipulation

Chapter 14: Binary Tree

Chapter 15: Binary Search Tree

Chapter 16: Binary Heap

Chapter 17: Graph

Chapter 18: Trie Data Structures

Chapter 19: All Advanced Data Structures

**Chapter 0: Introduction**

**Introduction to Data Structures and Algorithms (DSA)**

* Data Structures and Algorithms (DSA) form the foundation of computer science and programming.
* They are essential tools for solving problems efficiently and writing optimized code. Understanding DSA helps developers build scalable, robust, and maintainable software solutions.

**1. What Are Data Structures?**

* A data structure is a way of organizing and storing data so that it can be accessed and modified efficiently.
* Different data structures are optimized for specific types of operations and use cases.

**Types of Data Structures:**

* Linear Data Structures: Data elements are arranged sequentially.
* Examples:
* Arrays
* Strings (Internally uses array)
* Hash Table (Internally uses array) but when we get hash collisions -> BST
* Linked Lists
* Singly Linked List
* Doubly Linked List
* Circular Linked List
* Stacks
* Queues
* Simple Queue
* Circular Queue
* Priority Queue
* Double Ended Queue
* Non-Linear Data Structures: Data elements are arranged hierarchically or graphically.
* Examples:
* Trees
* Binary Tree
* BST
* Balanced Tree
* AVL Tree
* Red Black Tree
* Heap
* Max Heap
* Min Heap
* Trie
* N-ary Tree
* Segment Tree
* Fenwik Tree(Binary Indexed Tree)
* Graphs
* Hash Tables

**2. What Are Algorithms?**

* An algorithm is a step-by-step procedure or formula for solving a problem.
* It is a sequence of instructions designed to achieve a specific task.

**Key Features of Algorithms:**

* Correctness: Produces the desired output for all valid inputs.
* Efficiency: Optimizes time and space usage.
* Finiteness: Terminates after a finite number of steps.
* Scalability: Works efficiently for both small and large inputs.

**3. Why To Learn DSA?**

* Efficient Problem Solving: Helps design solutions that are faster and more resource-efficient.
* Technical Interviews: Mastery of DSA is crucial for cracking coding interviews and securing jobs in top tech companies.
* Scalable Applications: DSA knowledge ensures that applications handle real-world data and traffic efficiently.
* Programming Fundamentals: It builds a solid understanding of the underlying principles of programming and computation.

**4. Importance of Time and Space Complexity**

* DSA emphasizes evaluating the performance of algorithms in terms of:
* Time Complexity: Measures how the runtime of an algorithm changes with input size.
* Space Complexity: Measures the amount of memory an algorithm uses during execution.
* This analysis helps in choosing the most optimal solution for a problem.

**5. Applications of DSA**

* Real-Time Systems: Scheduling tasks (priority queues, heaps).
* Databases: Indexing and searching (trees, hash tables).
* Networking: Routing algorithms (graphs, Dijkstra's algorithm).
* Artificial Intelligence: State-space search (graphs, backtracking).
* Web Development: Caching (hashing), DOM traversal (trees).

**6. Overview of Key Concepts**

* Sorting and Searching: Algorithms like Quick Sort, Merge Sort, and Binary Search.
* Recursion: Breaking problems into smaller, similar sub-problems.
* Dynamic Programming: Solving problems by reusing previously computed solutions.
* Greedy Algorithms: Making locally optimal choices to achieve a global solution.
* Graph Algorithms: BFS, DFS, Shortest Path (Dijkstra, Floyd-Warshall).

**Conclusion:**

Data Structures and Algorithms are the backbone of efficient programming. Mastering these concepts enables developers to handle complex problems and contribute to the design of high-performance software systems. The study of DSA is not just theoretical but also immensely practical, forming a bridge between academic knowledge and real-world application.

**Chapter 1: Arrays**

**Array Concepts**

1. **Sliding Window:**

* Definition: A technique where you maintain a "window" (a subarray or substring) and slide it across the input array to solve problems that involve continuous subarrays or substrings.
* Use cases: Subarray sum, maximum or minimum subarray size, longest substring with unique characters, etc.
* Time Complexity: O(n), as it processes each element at most twice (once when entering the window, once when leaving).
* Example: Maximum sum of a subarray of size k, Longest substring without repeating characters.

1. **Two Pointers**

* Definition: Two pointers approach involves using two indices to traverse the array in a controlled manner. It’s often used for problems involving sorted arrays or pairs.
* Use cases: Finding pairs or subarrays with specific properties (e.g., sum of elements), merging sorted arrays, etc.
* Time Complexity: O(n), as both pointers move from one end of the array to the other.
* Example: Finding two numbers that sum up to a given target (Two Sum), 3-sum problem.

**3. Prefix**

* Definition: A technique where you precompute the cumulative sum of elements up to each index in the array. This helps in quickly calculating the sum of any subarray.
* Use cases: Range sum queries, solving problems involving the sum of elements in a subarray.
* Time Complexity: O(n) for precomputing the prefix sum, and O(1) for querying a range sum.
* Example: Finding the sum of elements between two indices in constant time.

1. **Binary Search on Arrays:**

* Definition: A search algorithm that works on sorted arrays, dividing the search space in half with each step. Used to find an element or the position of an element in logarithmic time.
* Use cases: Searching in sorted arrays, finding the first/last occurrence of a number, searching for a range, finding the ceiling or floor of an element, etc.
* Time Complexity: O(log n).
* Example: Binary Search, finding the first occurrence of an element.

1. **Rotation & Circular Arrays:**

* Definition: A rotation in an array involves shifting elements in the array, either to the left or right, and dealing with circular or cyclic behaviour in arrays.
* Use cases: Rotating arrays, finding the smallest element in a rotated sorted array.
* Time Complexity: O(n) for simple rotations, O(log n) for rotated sorted array search.
* Example: Searching in a rotated sorted array, finding the smallest element in a rotated array.

**Problem Patterns in Arrays**

1. **Maximum Subarray Problem (Kadane’s Algorithm):**

* Time Complexity: O(n)
* Example: Find the maximum sum of a subarray in an array of integers. Kadane’s algorithm is efficient and works by maintaining a running sum and updating it as you traverse the array.
* Key Insight: Update the current subarray sum at each step and reset it if the sum becomes negative. Track the maximum sum encountered so far.

1. **Two Sum Problem:**

* Time Complexity: O(n)
* Example: Find two numbers in an array whose sum is equal to a target value. This can be done efficiently using a hash map.
* Key Insight: For each element, check if the complement (target - current element) exists in the hash map.

1. **Find Kth Largest Element in an Array:**

* Time Complexity: O(n log n) (sorting), O(n) (Quickselect)
* Example: Find the Kth largest element in an unsorted array.
* Key Insight: Use Quickselect (a variation of QuickSort) to find the Kth largest element in linear time, or use Heap to track the largest elements in the array.

1. **Maximum Product Subarray:**

* Time Complexity: O(n)
* Example: Find the contiguous subarray with the largest product.
* Key Insight: Keep track of the maximum and minimum products ending at each position in the array, as negative numbers can become positive when multiplied.

1. **Trapping Rain Water:**

* Time Complexity: O(n)
* Example: Find the amount of rain water that can be trapped between the bars of a histogram, represented by an array of heights.
* Key Insight: Use two pointers (one from the left and one from the right) to calculate the amount of trapped water by comparing the current height with the maximum heights seen from both ends.

1. **Container With Most Water:**

* Time Complexity: O(n)
* Example: Find two lines in an array that, when chosen as the sides of a container, form the largest container.
* Key Insight: Use the two-pointer technique to move inward, adjusting the pointers based on which line is shorter to maximize the area.

1. **Search in Rotated Sorted Array:**

* Time Complexity: O(log n) (Binary Search)
* Example: Search for an element in a rotated sorted array.
* Key Insight: Use modified binary search to identify which part of the array is sorted and narrow down the search space based on the rotation.

1. **Longest Consecutive Sequence:**

* Time Complexity: O(n)
* Example: Find the length of the longest consecutive sequence in an unsorted array.
* Key Insight: Use a HashSet to store the elements and check for consecutive numbers.

1. **Subarray Sum Equals K:**

* Time Complexity: O(n)
* Example: Find all subarrays whose sum equals a given target k.
* Key Insight: Use a hash map to store the prefix sum and check if the difference between the current prefix sum and k exists in the map.

1. **Product of Array Except Self:**

* Time Complexity: O(n)
* Example: Find the product of all elements in an array, except for the current element, for each position.
* Key Insight: Use two passes – one from left to right to store the left product and one from right to left to store the right product.

1. **Longest Substring Without Repeating Characters:**

* Time Complexity: O(n)
* Example: Find the length of the longest substring without repeating characters.
* Key Insight: Use the sliding window technique with a hash map to track the last occurrence of characters.

**Advanced Array Techniques**

1. **Binary Search Variations:**

* Search for First/Last Occurrence: Efficiently find the first or last occurrence of an element in a sorted array.
* Time Complexity: O(log n)
* Example: Find the first occurrence of a target element in a sorted array.

1. **Kadanes' Algorithm Optimization:**

* Kadane’s algorithm can be applied to problems with a minimum subarray sum as well, by treating the minimum sum as the maximum and applying the same logic.

1. **Partitioning an Array:**

* Partition an array into two parts such that the elements in one part are less than a given value and elements in the other part are greater. Quickselect or Dutch National Flag problems can be used.

1. **Median of Two Sorted Arrays:**

* Time Complexity: O(log(min(n, m)))
* Problem: Find the median of two sorted arrays.
* Approach: Use a binary search on the smaller array to partition both arrays around the median.

1. **Three Sum Problem:**

* Time Complexity: O(n^2)
* Problem: Find all unique triplets in an array that sum up to zero. Use two pointers after sorting the array to find pairs.

**Chapter 2: Strings**

**String Concepts**

**Core Concepts:**

1. **Sliding Window:**

* Definition: A technique where you maintain a "window" (subarray or substring) and slide it across the input string to solve problems related to continuous substrings with specific conditions (e.g., size or properties).
* Use cases: Longest substring without repeating characters, smallest substring with certain properties, etc.
* Time Complexity: O(n), as each character is processed at most twice (once when entering the window, once when leaving).
* Example: Longest substring with at most two distinct characters.

1. **Two Pointers:**

* Definition: This technique involves using two indices or pointers, typically one starting from the left and one from the right. You use these pointers to check specific conditions as they traverse the string.
* Use cases: Reversing a string, checking if a string is a palindrome, finding substrings with specific properties, etc.
* Time Complexity: O(n), since both pointers traverse the string once.
* Example: Checking if a string is a palindrome, rearranging a string.

1. **Hashing (Using HashMap or HashSet):**

* Definition: Hashing is used to store and check the existence of elements efficiently, often used for counting characters, checking for duplicates, or verifying the frequency of characters in substrings.
* Use cases: Finding anagrams, checking if a string contains all characters from another string, etc.
* Time Complexity: O(n) for traversing the string and O(1) for hash map operations.
* Example: Finding all anagrams of a string in another string.

1. **Prefix and Suffix Arrays:**

* Definition: Arrays that store the cumulative counts or prefixes/suffixes of characters in a string. These are useful for pattern matching and searching for substrings efficiently.
* Use cases: Pattern matching, solving string search problems, etc.
* Time Complexity: O(n) for computing prefix/suffix arrays.
* Example: KMP algorithm for substring search.

1. **Backtracking**:

* Definition: A technique where you build solutions incrementally and abandon (backtrack) as soon as you determine the solution is not feasible. Often used for generating permutations, combinations, and solving puzzles like N-Queens.
* Use cases: Generating all permutations of a string, solving combinatorial problems.
* Time Complexity: O(n!) for generating all permutations of a string (in the worst case).
* Example: Finding all permutations of a string.

1. **Dynamic Programming (DP):**

* Definition: A technique used to solve problems that have overlapping subproblems by storing the results of subproblems and reusing them. It’s useful for optimization problems on strings.
* Use cases: Longest common subsequence, edit distance, longest palindromic substring.
* Time Complexity: O(n^2) for many classical string DP problems.
* Example: Longest Palindromic Substring, Edit Distance.

**Problem Patterns in Strings**

1. **Longest Substring Without Repeating Characters:**

* Time Complexity: O(n)
* Example: Given a string, find the length of the longest substring that contains no repeating characters.
* Key Insight: Use the sliding window technique with a hash set/map to track characters and expand or contract the window.

1. **Longest Palindromic Substring:**

* Time Complexity: O(n^2) (expand around center)
* Example: Find the longest substring that is a palindrome in a given string.
* Key Insight: Use the expand around center method or dynamic programming. The optimal method involves considering each character (and pair of characters) as the center and expanding outward.

1. **Anagram Substring Search:**

* Time Complexity: O(n + m), where n is the length of the text and m is the length of the pattern.
* Example: Find all starting indices of an anagram of a given string (pattern) in another string (text).
* Key Insight: Use a hash map to count characters in the pattern and slide the window over the text while updating the counts.

1. **Valid Parentheses:**

* Time Complexity: O(n)
* Example: Given a string containing just the characters (, ), {, }, [ and ], determine if the input string is valid.
* Key Insight: Use a stack to track opening parentheses, and pop when encountering a closing parenthesis. Ensure all parentheses are balanced.

1. **String Compression:**

* Time Complexity: O(n)
* Example: Implement a basic string compression algorithm that compresses a string by counting consecutive occurrences of characters.
* Key Insight: Traverse the string and replace consecutive characters with the character followed by its frequency.

1. **Edit Distance (Levenshtein Distance):**

* Time Complexity: O(m \* n), where m is the length of string1 and n is the length of string2.
* Example: Given two strings, find the minimum number of operations required to convert one string to another (insertions, deletions, substitutions).
* Key Insight: Use dynamic programming to build a table of edit distances between substrings of both strings.

1. **Palindrome Partitioning:**

* Time Complexity: O(n^2)
* Example: Partition a string such that every substring is a palindrome.
* Key Insight: Use backtracking to explore all possible partitions while checking if each substring is a palindrome.

1. **KMP Pattern Matching Algorithm:**

* Time Complexity: O(n + m), where n is the length of the text and m is the length of the pattern.
* Example: Implement the Knuth-Morris-Pratt (KMP) algorithm to search for a pattern in a text.
* Key Insight: Use the LPS (Longest Prefix-Suffix) array to avoid unnecessary re-evaluations of matched portions.

1. **Rabin-Karp Algorithm:**

* Time Complexity: O(n + m) (average case), O(n \* m) (worst case)
* Example: A string searching algorithm that uses hashing to find a pattern in a text.
* Key Insight: Compute the hash of the pattern and sub-strings of the text to find matches efficiently.

1. **Word Break Problem:**

* Time Complexity: O(n^2)
* Example: Given a string and a dictionary of words, determine if the string can be segmented into a space-separated sequence of dictionary words.
* Key Insight: Use dynamic programming to check if the string can be split into valid words.

1. **Z Algorithm:**

* Time Complexity: O(n)
* Example: Compute the Z-array for a given string. The Z-array provides the length of the longest substring starting from i that is also a prefix of the string.
* Key Insight: This can be used for efficient string matching and pattern search.

1. **Smallest Window in String Containing All Characters of Another String:**

* Time Complexity: O(n)
* Example: Given two strings s and t, find the smallest substring in s which contains all the characters of t.
* Key Insight: Use a hash map to track characters in t and a sliding window to find the smallest matching substring.

**Advanced String Techniques**

1. **Suffix Arrays and Suffix Trees:**

* Definition: Data structures used to store all suffixes of a string in a sorted order to facilitate efficient substring search and other string queries.
* Use cases: Pattern matching, finding the longest repeated substring.
* Time Complexity: O(n log n) for suffix arrays, O(n) for suffix trees.
* Example: Use a suffix array for efficient substring matching.

1. **Trie (Prefix Tree):**

* Definition: A tree-like data structure that stores a dynamic set of strings, where the nodes represent prefixes of strings.
* Use cases: Efficient prefix-based search, autocomplete systems, and dictionary-based problems.
* Time Complexity: O(k) for search/insert/delete, where k is the length of the string.
* Example: Implementing a prefix tree for autocomplete.

1. **Manacher’s Algorithm:**

* Definition: A linear time algorithm to find the longest palindromic substring in O(n) time.
* Use cases: Finding the longest palindrome in a string.
* Time Complexity: O(n)
* Example: Finding the longest palindromic substring using Manacher's Algorithm.

1. **KMP Preprocessing:**

* Definition: The KMP (Knuth-Morris-Pratt) algorithm improves upon brute-force string matching by avoiding redundant rechecks through preprocessing.
* Use cases: Efficient substring searching.
* Time Complexity: O(n + m), where n is the length of the text and m is the length of the pattern.
* Example: Preprocess the pattern to find the Longest Prefix Suffix (LPS) array.

**Chapter 3: Linked List**

**Linked List Concepts**

**Core Concepts:**

1. **Singly Linked List:**

* Definition: A data structure where each element (node) contains a data part and a reference (or link) to the next node in the list.
* **Time Complexity:**
* Insertion at beginning: O(1)
* Insertion at end: O(n)
* Deletion: O(1) (if node is known, O(n) if node needs to be found)
* Searching: O(n)
* Example: A simple linked list with nodes having a data part and a link to the next node.

1. **Doubly Linked List:**

* Definition: A variation where each node has two references: one pointing to the next node and one pointing to the previous node. This allows for bidirectional traversal.
* Time Complexity:
* Insertion at beginning: O(1)
* Insertion at end: O(1) (with tail reference)
* Deletion: O(1) (if node is known)
* Searching: O(n)
* Example: A doubly linked list where each node points to both the next and previous node.

1. **Circular Linked List:**

* Definition: A linked list in which the last node points back to the first node, forming a cycle. It can be singly or doubly linked.
* Time Complexity:
* Insertion at beginning: O(1)
* Insertion at end: O(1)
* Searching: O(n)
* Example: A list where the last node's next pointer points to the head node.

1. **Reversal of Linked List:**

* Definition: Reversing a linked list means changing the direction of the pointers in each node such that the first node becomes the last, and vice versa.
* Time Complexity: O(n), where n is the number of nodes.
* Example: Reverse the order of elements in the linked list.

1. **Detecting a Cycle in a Linked List:**

* Definition: Identifying whether there is a cycle in a linked list (a loop where a node’s next pointer points back to an earlier node in the list).
* Key Insight: Use Floyd's Tortoise and Hare algorithm (two-pointer technique).
* Time Complexity: O(n), where n is the number of nodes in the list.
* Example: Detect if a linked list contains a cycle and return the starting point of the cycle.

1. **Middle Element of Linked List:**

* Definition: Finding the middle element of a linked list in one pass.
* Key Insight: Use the two-pointer technique, with one pointer advancing twice as fast as the other.
* Time Complexity: O(n)
* Example: Return the middle node of a singly linked list.

1. **Linked List Intersection:**

* Definition: Finding the intersection node of two linked lists.
* Key Insight: Use the difference in lengths between the two lists and align the starting points to check for the intersection.
* Time Complexity: O(n + m), where n and m are the lengths of the two lists.
* Example: Given two linked lists, find the node at which they intersect.

1. **Merging Two Sorted Linked Lists:**

* Definition: Merging two sorted linked lists into a single sorted linked list.
* Time Complexity: O(n + m), where n and m are the lengths of the two lists.
* Example: Merge two sorted linked lists into a single sorted list.

1. **Kth to Last Element:**

* Definition: Find the kth element from the end of a linked list.
* Key Insight: Use the two-pointer technique to traverse the list with one pointer at k nodes ahead of the other.
* Time Complexity: O(n), where n is the number of nodes.
* Example: Return the kth element from the last in a linked list.

1. **Flatten a Linked List:**

* Definition: Flatten a linked list where each node has an additional pointer to another list, which needs to be merged into the main list.
* Key Insight: Use recursion or iterative merging to flatten the structure.
* Time Complexity: O(n), where n is the number of nodes in the list.
* Example: Flatten a linked list where each node has a child pointer.

**Problem Patterns in Linked Lists**

1. **Reverse a Linked List:**

* Time Complexity: O(n)
* Example: Reverse a singly linked list in place.
* Key Insight: Use an iterative approach with three pointers (prev, current, next) to reverse the direction of the links.

1. **Detect Cycle in Linked List (Floyd’s Cycle-Finding Algorithm):**

* Time Complexity: O(n)
* Example: Detect if a linked list contains a cycle and find the node where the cycle starts.
* Key Insight: Use the Floyd’s Tortoise and Hare Algorithm where two pointers move at different speeds to detect a cycle.

1. **Find the Middle of a Linked List:**

* Time Complexity: O(n)
* Example: Find the middle element of a singly linked list using the slow and fast pointer technique.
* Key Insight: Use two pointers: one moves one step at a time, the other moves two steps at a time. The slow pointer will reach the middle when the fast pointer reaches the end.

1. **Merge Two Sorted Linked Lists:**

* Time Complexity: O(n + m), where n and m are the lengths of the two lists.
* Example: Merge two sorted linked lists into one sorted linked list.
* Key Insight: Use the two-pointer technique, comparing the nodes from each list and linking the smaller node to the result list.

1. **Find the Kth Element from the End:**

* Time Complexity: O(n)
* Example: Find the kth element from the end of a linked list.
* Key Insight: Use the two-pointer technique. Move one pointer k nodes ahead, then move both pointers one step at a time. When the first pointer reaches the end, the second pointer will be at the kth node from the end.

1. **Remove Nth Node from End of List:**

* Time Complexity: O(n)
* Example: Remove the nth node from the end of the linked list.
* Key Insight: Use the two-pointer technique to reach the node before the target and delete it in a single pass.

1. **Intersection of Two Linked Lists:**

* Time Complexity: O(n + m)
* Example: Find the intersection node of two singly linked lists.
* Key Insight: Calculate the difference in length of the two lists and align the starting pointers. Then, traverse both lists simultaneously to find the intersection.

1. **Flatten a Multilevel Doubly Linked List:**

* Time Complexity: O(n)
* Example: Flatten a multilevel doubly linked list where each node can have a child pointer to another doubly linked list.
* Key Insight: Use recursion or iterative flattening using a stack.

1. **Palindrome Linked List:**

* Time Complexity: O(n)
* Example: Check if a linked list is a palindrome.
* Key Insight: Use slow and fast pointers to find the middle, reverse the second half of the list, and compare both halves.

1. **Swap Nodes in Pairs:**

* Time Complexity: O(n)
* Example: Swap every two adjacent nodes in a linked list.
* Key Insight: Use pointer manipulation to swap nodes in pairs iteratively.

**Advanced Linked List Techniques**

1. **Floyd's Cycle-Finding Algorithm (Tortoise and Hare):**

* Definition: A famous algorithm used to detect cycles in a linked list.
* Use cases: Detecting if a linked list contains a cycle, and sometimes finding the start of the cycle.
* Time Complexity: O(n)
* Example: Use two pointers moving at different speeds to detect a cycle in the linked list.

1. **Rearrange a Linked List:**

* Definition: Reorganize the elements of a linked list (e.g., rearrange nodes such that all even indexed nodes appear before odd indexed nodes).
* Time Complexity: O(n)
* Example: Rearrange the nodes of a linked list based on certain conditions (like even and odd values).

1. **Linked List Sort:**

* Definition: Sorting a linked list (using Merge Sort or Quick Sort).
* Time Complexity: O(n log n)
* Example: Sort a singly linked list using merge sort. Merge sort is preferred due to its stable performance even for linked lists.

1. **K Reverse in a Linked List:**

* Definition: Reverse every k nodes in a linked list.
* Time Complexity: O(n)
* Example: Reverse the linked list in groups of k nodes at a time.

**Chapter 4: Stack**

**Expanded Concepts**

**Core Concepts:**

1. **Definition of Stack:**

* Concept: A stack is a linear data structure that follows the Last In, First Out (LIFO) order. The element that is inserted last is the one that gets removed first.
* Operations:
* Push: Insert an element onto the top of the stack.
* Pop: Remove the element from the top of the stack.
* Peek/Top: Retrieve the element from the top of the stack without removing it.
* isEmpty: Check if the stack is empty.
* Size: Get the current number of elements in the stack.
* Time Complexity: O(1) for all basic operations (Push, Pop, Peek, isEmpty).

1. **Implementation:**

* Array-based Stack: Uses a static or dynamic array to implement the stack operations.
* Time Complexity:
* Push/Pop/Peek: O(1) (amortized for dynamic resizing)
* Linked List-based Stack: Uses a linked list where each new node points to the previous node.
* Time Complexity:
* Push/Pop/Peek: O(1) (constant time for adding/removing nodes).

1. **Applications:**

* Expression Evaluation (Postfix, Prefix, Infix): Stack is used to evaluate expressions by converting them to postfix/prefix and then evaluating.
* Undo Mechanism: Storing previous actions in a stack to undo the last action.
* Depth First Search (DFS): DFS is often implemented using a stack for exploring nodes.
* Parentheses Matching: Checking if a given expression has balanced parentheses.

1. **Balanced Parentheses Problem:**

* Definition: Check if the parentheses in a given expression are balanced using a stack.
* Key Insight: For each opening parenthesis, push it onto the stack; for each closing parenthesis, pop the stack and check if it matches the expected type.
* Time Complexity: O(n), where n is the length of the input string.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Problem Patterns**

1. **Evaluate Postfix Expression:**

* Time Complexity: O(n)
* Example: Given a postfix expression, evaluate it and return the result.
* Key Insight: Use a stack to process operands and operators. For operands, push them onto the stack; for operators, pop operands from the stack, apply the operator, and push the result back onto the stack.

1. **Evaluate Prefix Expression:**

* Time Complexity: O(n)
* Example: Given a prefix expression, evaluate it and return the result.
* Key Insight: Similar to postfix, but process the expression from right to left. For operands, push them onto the stack; for operators, pop operands, apply the operator, and push the result back.

1. **Infix to Postfix Expression Conversion:**

* Time Complexity: O(n)
* Example: Convert an infix expression (e.g., a + b \* c) to its equivalent postfix expression (e.g., a b c \* +).
* Key Insight: Use a stack to handle operators and parentheses. Use operator precedence and associativity rules to determine the order of operators in the resulting postfix expression.

1. **Parentheses Matching:**

* Time Complexity: O(n)
* Example: Given a string, check if all parentheses are properly balanced.
* Key Insight: Use a stack to push opening parentheses and pop them when matching closing parentheses are found. If the stack is empty after processing, the expression is balanced.

1. **Next Greater Element (NGE) to the Right:**

* Time Complexity: O(n)
* Example: Given an array, for each element, find the next greater element to the right.
* Key Insight: Use a stack to store indices. Traverse the array from right to left and use the stack to find the next greater element for each item.

1. **Largest Rectangle in Histogram:**

* Time Complexity: O(n)
* Example: Given an array representing heights of a histogram, find the area of the largest rectangle that can be formed.
* Key Insight: Use a stack to maintain the indices of the bars. For each bar, pop from the stack until you find a smaller bar, then calculate the area formed by the popped bars.

1. **Trapping Rain Water:**

* Time Complexity: O(n)
* Example: Given an array of non-negative integers representing elevations, calculate the amount of water trapped between the bars after raining.
* Key Insight: Use a stack to store indices and calculate the trapped water whenever you encounter a higher bar that can trap water over the current bars in the stack.

1. **Stock Span Problem:**

* Time Complexity: O(n)
* Example: Given an array of stock prices, for each stock price, find the span of the stock's price (i.e., the number of consecutive days the price has been less than or equal to the current price).
* Key Insight: Use a stack to track indices of the previous prices and calculate the span based on those indices.

1. **Sort a Stack:**

* Time Complexity: O(n^2)
* Example: Given an unsorted stack, sort the stack using only stack operations (push, pop, peek, isEmpty).
* Key Insight: Use a temporary stack and iteratively pop elements from the original stack and place them in the correct position in the temporary stack.

1. **Daily Temperature Problem:**

* Time Complexity: O(n)
* Example: Given a list of daily temperatures, return a list that tells you how many days you would have to wait until a warmer temperature.
* Key Insight: Use a stack to keep track of indices where we haven't yet found a warmer temperature, and calculate the difference when we find one.

**Advanced Stack Techniques**

1. **Next Greater Element (NGE) to the Left:**

* Time Complexity: O(n)
* Example: Given an array, find the next greater element to the left for each element.
* Key Insight: Similar to NGE to the right, but traverse the array from left to right and use a stack to store indices and find the next greater element.

1. **Sliding Window Maximum:**

* Time Complexity: O(n)
* Example: Given an array, find the maximum element in every subarray of size k.
* Key Insight: Use a deque (double-ended queue) or stack to keep track of the indices of the elements that are currently in the window, ensuring that the maximum element is always accessible in constant time.

1. **Palindrome Stack Approach:**

* Time Complexity: O(n)
* Example: Given a string, check if the string is a palindrome using a stack.
* Key Insight: Push the first half of the string onto the stack, and then compare the second half of the string with the stack’s elements.

1. **Maximal Rectangle in Binary Matrix:**

* Time Complexity: O(m \* n)
* Example: Given a binary matrix, find the area of the largest rectangle of 1’s.
* Key Insight: Treat each row of the matrix as a histogram, then use the Largest Rectangle in Histogram technique with a stack for each row.

1. **Parentheses with Different Types:**

* Time Complexity: O(n)
* Example: Check if a string contains balanced parentheses of multiple types: (), {}, [].
* Key Insight: Use a stack to match opening and closing parentheses of different types using a hash map.

**Chapter 5: Queue**

**Expanded Concepts**

**Core Concepts:**

1. **Definition of Queue:**

* Concept: A queue is a linear data structure that follows the First In, First Out (FIFO) order. The element that is inserted first is the one that gets removed first.
* Operations:
* Enqueue: Insert an element at the rear of the queue.
* Dequeue: Remove an element from the front of the queue.
* Peek/Front: Retrieve the element at the front of the queue without removing it.
* isEmpty: Check if the queue is empty.
* Size: Get the current number of elements in the queue.
* Time Complexity:
* O(1) for all basic operations (Enqueue, Dequeue, Peek, isEmpty).

1. **Types of Queue:**

* Simple Queue: A basic queue that allows insertion at the rear and deletion at the front.
* Circular Queue: A queue where the last position is connected to the first position, forming a circle. It helps in utilizing memory efficiently by avoiding overflow when there are empty spaces in the front of the queue.
* Priority Queue: A queue where each element has a priority associated with it. Elements are dequeued based on priority, not on arrival time.
* Deque (Double-Ended Queue): A queue where elements can be inserted or deleted from both ends (front and rear).

1. **Implementation:**

* Array-based Queue: Uses an array to implement the queue. When using a circular queue, the head and tail pointers wrap around the array.
* Time Complexity:
* Enqueue/Dequeue/Peek: O(1)
* Linked List-based Queue: Uses a linked list with pointers to represent the front and rear of the queue.
* Time Complexity:
* Enqueue/Dequeue/Peek: O(1) (constant time for adding/removing nodes).

1. **Applications**:

* Job Scheduling: In operating systems, processes are queued and executed in the order they arrive.
* Breadth First Search (BFS): BFS traversal of graphs uses a queue to explore nodes layer by layer.
* Buffer Management: Queues are used to manage buffers in applications such as I/O operations.
* Print Queue: Print jobs are queued to be processed in the order they arrive.

**Problem Patterns**

1. **Implement a Queue using Stacks:**

* Time Complexity: O(1) for Enqueue, O(n) for Dequeue (amortized average case O(1))
* Example: Implement a queue using two stacks, one for enqueue and the other for dequeue operations.
* Key Insight: Use two stacks: one for enqueueing elements and the other for dequeueing elements. When dequeuing, transfer elements from the enqueue stack to the dequeue stack if the dequeue stack is empty.

1. **Queue using Array (Circular Queue):**

* Time Complexity: O(1) for Enqueue, Dequeue, Peek, and isEmpty.
* Example: Implement a circular queue using a fixed-size array.
* Key Insight: Use two pointers: one for the front of the queue and one for the rear. When the rear reaches the end of the array, it wraps around to the beginning to utilize the available space.

1. **First Non-Repeating Character in a Stream:**

* Time Complexity: O(n)
* Example: Given a stream of characters, find the first non-repeating character after each character is inserted.
* Key Insight: Use a queue to maintain the order of characters and a frequency map to track the count of each character. Remove characters from the front of the queue if they become repeating.

1. **Reverse the First K Elements of Queue:**

* Time Complexity: O(n)
* Example: Reverse the first k elements of a queue, keeping the remaining elements in the same order.
* Key Insight: Use a stack to reverse the first k elements. After reversing, enqueue the remaining elements back into the queue.

1. **Circular Queue - Implement the Circular Buffer:**

* Time Complexity: O(1)
* Example: Implement a circular queue with fixed size.
* Key Insight: Use modulo arithmetic to wrap around the queue. Both front and rear pointers should be adjusted to move in a circular manner.

1. **Generate Binary Numbers from 1 to N:**

* Time Complexity: O(n)
* Example: Using a queue, generate binary numbers from 1 to N.
* Key Insight: Start with 1, and then keep enqueueing the binary number and its extensions (current + '0' and current + '1').

1. **Binary Tree Level Order Traversal (BFS):**

* Time Complexity: O(n)
* Example: Perform a level-order traversal of a binary tree.
* Key Insight: Use a queue to store nodes level by level. For each node, enqueue its left and right child (if present) for further exploration.

1. **Rearrange a Queue:**

* Time Complexity: O(n)
* Example: Rearrange elements in the queue such that the front element is moved to the back.
* Key Insight: Use a temporary queue to process elements and rotate the queue.

1. **Sliding Window Maximum using Queue:**

* Time Complexity: O(n)
* Example: Given an array, find the maximum element in every subarray of size k.
* Key Insight: Use a deque (double-ended queue) to maintain the indices of the maximum elements in the sliding window. Only store elements that are within the window and are potentially the maximum.

1. **Queue to Stack using Two Queues:**

* Time Complexity: O(n)
* Example: Implement a stack using two queues.
* Key Insight: Use two queues: one for push operations and the other for pop operations. For pop, dequeue from one queue, transferring elements between the queues to maintain stack order.

**Advanced Queue Techniques**

1. **Implement a Priority Queue (Min/Max Heap):**

* Time Complexity: O(log n) for insertion and removal.
* Example: Implement a priority queue using a heap.
* Key Insight: Use a binary heap (min or max) to maintain the priority of elements. Insertion and removal operations can be done efficiently in O(log n) time.

1. **Rearrange a Queue in Alternating Order:**

* Time Complexity: O(n)
* Example: Rearrange elements in the queue such that odd-indexed elements come before even-indexed elements while maintaining their relative order.
* Key Insight: Use two queues to separate odd and even-indexed elements, then combine them in alternating order.

1. **K-th Smallest Element in a Stream:**

* Time Complexity: O(log k)
* Example: Given a stream of numbers, find the k-th smallest element at any point in the stream.
* Key Insight: Use a max heap of size k to track the k smallest elements. After each insertion, the top of the heap is the k-th smallest element.

1. **Median in a Stream:**

* Time Complexity: O(log n)
* Example: Find the median of a stream of numbers.
* Key Insight: Use two heaps — a max-heap for the lower half and a min-heap for the upper half of the numbers. This allows you to efficiently find the median by balancing the sizes of the heaps.

1. **Serialize and Deserialize Binary Tree:**

* Time Complexity: O(n)
* Example: Serialize and deserialize a binary tree to/from a queue.
* Key Insight: Use a queue for level-order traversal to serialize the tree and deserialize it back into the original structure.

**Chapter 6: Hashing, HashSet, HashMap**

**Hashing, Hash Set, and Hash Map - Expanded**

**Core Concepts:**

1. **Hashing:**

* Concept: Hashing is a technique used to convert a given key into a fixed-size value, called a hash code, that represents the original data. This is useful for quick lookups, insertions, and deletions.
* Hash Function: A function that takes an input (key) and produces a hash code. A good hash function distributes keys uniformly across the hash table to minimize collisions.
* Collisions: Occur when two keys have the same hash code. Common techniques to handle collisions include:
* Chaining: Each bucket of the hash table contains a linked list of keys that hash to the same bucket.
* Open Addressing: If a collision occurs, the algorithm searches for the next available bucket in the table.
* Time Complexity:
* O(1) for insert, delete, and search operations on average (with a good hash function and low collision rate).
* Worst-case time complexity can be O(n) in case of high collision, especially in open addressing.

1. **Hash Set:**

* Concept: A hash set is a collection of unique elements with no duplicates, implemented using hashing. It allows for efficient checking if an element exists in the set.
* Operations:
* add(): Inserts an element into the set.
* remove(): Deletes an element from the set.
* contains(): Checks if an element exists in the set.
* Time Complexity:
* O(1) for add, remove, and contains on average.
* Worst-case O(n) if the hash table suffers from collisions.

1. **Hash Map:**

* Concept: A hash map (or dictionary) is a key-value pair data structure, where each key maps to a value. It allows for fast lookups, insertions, and deletions.
* Operations:
* put(key, value): Inserts or updates the key-value pair in the map.
* get(key): Retrieves the value associated with the given key.
* remove(key): Deletes the key-value pair from the map.
* Time Complexity:
* O(1) for put, get, and remove on average.
* Worst-case O(n) if there are many collisions or poor hash distribution.

1. **Applications of Hashing:**

* Data Deduplication: Ensuring that a collection contains only unique elements (using HashSet).
* Efficient Lookups: Storing data in HashMap for fast lookups based on keys.
* Counting Frequencies: Counting the frequency of elements (using HashMap or HashSet).
* Hashing for Caching: Storing computed values in a cache to avoid recalculating the same values.
* Detecting Duplicates: Checking if an element has already appeared in the array or list.
* Anagram Detection: Identifying if two strings are anagrams using a HashMap.

**Problem Patterns in Hashing, Hash Set, and Hash Map**

1. **Two Sum Problem (using HashMap):**

* Time Complexity: O(n)
* Example: Given an array of integers, find two numbers that sum up to a specific target.
* Key Insight: Use a HashMap to store the complement of each element as you iterate through the array. Check if the current element exists in the map.

1. **Count Frequencies of Elements (using HashMap):**

* Time Complexity: O(n)
* Example: Count the frequency of each element in an array.
* Key Insight: Use a HashMap to store the frequency of each element. For each element, increment its count in the map.

1. **Find the First Non-Repeating Character (using HashMap):**

* Time Complexity: O(n)
* Example: Given a string, find the first non-repeating character.
* Key Insight: Traverse the string twice: first to populate a HashMap with character frequencies, and second to check which character has a count of 1.

1. **Anagram Detection (using HashMap):**

* Time Complexity: O(n)
* Example: Check if two strings are anagrams of each other.
* Key Insight: Sort both strings or use a HashMap to count character frequencies. If the counts match for both strings, they are anagrams.

1. **Intersection of Two Arrays (using HashSet):**

* Time Complexity: O(n)
* Example: Given two arrays, find the intersection of the two.
* Key Insight: Use a HashSet to store the elements of one array, and then check which elements of the second array exist in the set.

1. **Subarray Sum Equals K (using HashMap):**

* Time Complexity: O(n)
* Example: Given an array, find the number of subarrays whose sum equals k.
* Key Insight: Use a HashMap to store the cumulative sum of elements. For each element, check if the cumulative sum minus k exists in the map.

1. **Group Anagrams (using HashMap):**

* Time Complexity: O(nk log k), where n is the number of strings and k is the average length of the strings.
* Example: Group a list of strings into anagrams.
* Key Insight: Sort each string or use a character count HashMap as the key. Group strings with the same key together.

1. **Longest Substring Without Repeating Characters (using HashSet):**

* Time Complexity: O(n)
* Example: Given a string, find the length of the longest substring without repeating characters.
* Key Insight: Use a sliding window approach with a HashSet to keep track of unique characters.

1. **Find Duplicate in Array (using HashSet):**

* Time Complexity: O(n)
* Example: Given an array, find the first duplicate element.
* Key Insight: Use a HashSet to track seen elements. If an element is encountered again, it's a duplicate.

1. **HashMap for Caching:**

* Time Complexity: O(1) for get and put operations.
* Example: Implement an LRU (Least Recently Used) cache.
* Key Insight: Use a HashMap to store keys and values, and a doubly linked list to maintain the order of usage.

1. **Longest Consecutive Sequence (using HashSet):**

* Time Complexity: O(n)
* Example: Given an unsorted array of integers, find the length of the longest consecutive elements sequence.
* Key Insight: Use a HashSet to store elements, then for each element, check if it’s the start of a sequence by checking for element-1 in the set.

1. **Palindrome Permutation (using HashMap):**

* Time Complexity: O(n)
* Example: Check if any permutation of a string can form a palindrome.
* Key Insight: Use a HashMap to count the frequency of characters. A string can form a palindrome if at most one character has an odd frequency.

**Advanced Hashing Techniques**

1. **Implementing Custom HashMap:**

* Time Complexity: O(1) average, O(n) worst-case.
* Example: Implement a custom HashMap using arrays and linked lists (for collision handling via chaining).
* Key Insight: Handle hash collisions efficiently by using chaining (linked list) or open addressing (quadratic probing, linear probing). Choose the appropriate resizing and rehashing strategy.

1. **Bucket Sort using Hashing:**

* Time Complexity: O(n+k), where n is the number of elements and k is the number of buckets.
* Example: Sort an array of numbers using bucket sort.
* Key Insight: Distribute the elements into several buckets, sort each bucket, and then concatenate the results. This approach can work in linear time under certain conditions (when elements are uniformly distributed).

1. **Count Distinct Elements in a Stream (using HashSet):**

* Time Complexity: O(1)
* Example: Count the number of distinct elements in a stream of numbers.
* Key Insight: Use a HashSet to keep track of unique elements in the stream. Each insertion into the set will be O(1), so the overall complexity is linear in terms of the number of elements.

1. **Find the Majority Element (using HashMap):**

* Time Complexity: O(n)
* Example: Find the element that appears more than n/2 times in an array.
* Key Insight: Use a HashMap to count occurrences of each element. The element with the highest count that exceeds n/2 is the majority element.

**Chapter 7: Searching Algorithms**

**Searching Algorithms - Expanded Concepts**

1. **Linear Search**

* Concept: Linear search is the simplest search algorithm. It works by checking each element of the array sequentially until the target element is found or the entire array is traversed.
* Time Complexity:
* Best Case: O(1) (when the element is found at the first position).
* Worst Case: O(n) (when the element is not present or is at the last position).
* Use Case: Ideal for small datasets or unsorted data, but inefficient for larger datasets.

**2. Binary Search**

* Concept: Binary search is an efficient algorithm for finding an element in a sorted array. It works by repeatedly dividing the search interval in half:
* Start with the middle element.
* If the target value is smaller than the middle element, narrow the search to the left half.
* If the target value is larger, narrow the search to the right half.
* Repeat the process until the target element is found or the interval is empty.
* Time Complexity:
* Best Case: O(1) (when the middle element is the target).
* Average and Worst Case: O(log n) (because the array is halved in each step).
* Use Case: Works only for sorted arrays or data that can be sorted (e.g., binary search on the search tree).

**3. Exponential Search**

* Concept: Exponential search (or Doubling Search) is useful for searching in a sorted array when the size of the array is unknown. It works by first checking the first element, then the second element, the fourth element, the eighth element, and so on, to find a range where the target might lie, and then performs binary search within that range.
* Time Complexity:
* Best Case: O(1) (if the target element is found at the first check).
* Worst Case: O(log n) (since binary search is performed after finding the range).
* Use Case: When the data is sorted, and the size of the array is large or unknown.

**4. Ternary Search**

* Concept: Ternary search is similar to binary search, but instead of dividing the array into two parts, it divides the array into three parts. It works on sorted arrays and reduces the search space more aggressively than binary search.
* It checks two midpoints and eliminates one of the three subarrays at each step.
* Time Complexity:
* Best, Average, Worst Case: O(log₃ n), which is still O(log n) but with a smaller base constant.
* Use Case: Useful when dividing the problem space into three parts can provide more effective pruning (though binary search is more commonly used).

**5. Jump Search**

* Concept: Jump search is an algorithm for searching a sorted array. It works by jumping ahead by a fixed number of steps (called block size) and checking if the target is within that block.
* The algorithm divides the array into blocks and checks the block where the target might be, then performs a linear search within that block.
* Time Complexity:
* Best Case: O(√n) (when the block size is chosen optimally).
* Worst Case: O(√n).
* Use Case: Ideal for searching sorted arrays or linked lists, especially when binary search isn’t possible due to constraints like non-random access.

**Problem Patterns in Searching Algorithms**

1. **Find Element in Sorted Array**

* Problem: Given a sorted array, search for a specific element.
* Algorithm: Use Binary Search for O(log n) time complexity.
* Time Complexity: O(log n).

**2. Find Element in Rotated Sorted Array**

* Problem: Given a sorted array that is rotated at an unknown pivot, search for an element.
* Algorithm: Modified Binary Search. Compare the middle element with the start and end elements to determine which part of the array is sorted and recursively search in that part.
* Time Complexity: O(log n).

**3. Search in Infinite Sorted Array**

* Problem: Given an infinite array (or a large array where the length is unknown), find an element.
* Algorithm: Exponential Search to find the range where the element may be, then apply Binary Search.
* Time Complexity: O(log n).

**4. Find First or Last Occurrence in Sorted Array**

* Problem: Given a sorted array, find the first or last occurrence of a given element.
* Algorithm: Modify Binary Search to find the first or last occurrence by adjusting the search bounds after finding the target element.
* Time Complexity: O(log n).

**5. Find Minimum or Maximum in Rotated Sorted Array**

* Problem: Find the minimum or maximum element in a rotated sorted array.
* Algorithm: Use Binary Search to find the pivot where the rotation occurs.
* Time Complexity: O(log n).

**6. Search in 2D Matrix (Sorted in Rows and Columns)**

* Problem: Given a 2D matrix where each row and column is sorted, search for an element.
* Algorithm: Start from the top-right or bottom-left corner and move left or down depending on comparisons with the target.
* Time Complexity: O(m + n), where m is the number of rows and n is the number of columns.

**7. Search for Range (Find Elements in Given Range)**

* Problem: Given a sorted array and a range [low, high], find all elements within this range.
* Algorithm: Use Binary Search to find the lower and upper bounds of the range.
* Time Complexity: O(log n) to find the bounds, then O(k) for the number of elements within the range.

**8. Search in Bitonic Array**

* Problem: Given an array that first increases and then decreases, find an element.
* Algorithm: Find the peak element using Binary Search, then search in both increasing and decreasing subsections.
* Time Complexity: O(log n).

**9. Find Peak Element**

* Problem: Given an array where each element is strictly greater than its neighbors, find any peak element.
* Algorithm: Use Binary Search to find a peak element. At each step, check if the middle element is greater than or equal to its neighbors.
* Time Complexity: O(log n).

**Advanced Searching Algorithms**

1. **Interpolation Search**

* Concept: Interpolation search works on the principle of estimating the position of the key based on the values at the low and high ends of the array. It performs better than binary search for uniformly distributed data.
* Time Complexity:
* Best Case: O(1) (if the estimate is correct).
* Worst Case: O(n) (if the distribution is skewed).
* Use Case: Works efficiently when the array is sorted and uniformly distributed.

1. **K-th Smallest/Largest Element in an Unsorted Array (QuickSelect)**

* Concept: QuickSelect is an algorithm to find the k-th smallest or largest element in an unsorted array. It is a variation of QuickSort, but instead of sorting the entire array, it only focuses on finding the k-th element by partitioning the array.
* Time Complexity:
* Average Case: O(n).
* Worst Case: O(n²) (though the worst case is rare with random pivot selection).
* Use Case: Efficient for finding the k-th smallest or largest element without needing to sort the entire array.

1. **Binary Search on Answer (Search Space)**

* Concept: This technique involves performing binary search not on an array but on the possible values of the answer. This is useful in problems where you're trying to find the best possible answer given constraints.
* Time Complexity: O(log(max\_value)), where max\_value is the maximum possible value for the solution.
* Use Case: Often used in problems related to optimizing parameters or finding bounds on the solution.

**Chapter 8: Sorting Algorithms**

**Sorting Algorithms - Expanded Concepts**

**1. Bubble Sort**

* Concept: Bubble Sort repeatedly compares adjacent elements in an array and swaps them if they are in the wrong order. The process is repeated until no swaps are needed.
* Time Complexity:
* Best Case: O(n) (when the array is already sorted).
* Average and Worst Case: O(n²).
* Use Case: Not efficient for large datasets, mostly used for educational purposes or small datasets where simplicity is preferred.

1. **Selection Sort**

* Concept: Selection Sort works by selecting the minimum (or maximum) element from the unsorted portion of the array and swapping it with the first unsorted element. This is done repeatedly for the remaining unsorted portion.
* Time Complexity:
* Best, Average, Worst Case: O(n²).
* Use Case: Works for small datasets where simplicity is desired. However, it's inefficient for large datasets compared to other algorithms.

1. **Insertion Sort**

* Concept: Insertion Sort builds the final sorted array one element at a time by removing elements from the unsorted portion and inserting them into the correct position in the sorted portion.
* Time Complexity:
* Best Case: O(n) (when the array is already sorted).
* Average and Worst Case: O(n²).
* Use Case: Efficient for small datasets or nearly sorted arrays, often used in hybrid sorting algorithms like Timsort (used in Java and Python).

1. **Merge Sort**

* Concept: Merge Sort is a divide and conquer algorithm that splits the array into two halves, recursively sorts each half, and then merges the sorted halves back together.
* Time Complexity:
* Best, Average, Worst Case: O(n log n).
* Use Case: Merge Sort is stable and efficient for sorting large datasets. It is often used when stability (preserving the order of equal elements) is important.

**5. Quick Sort**

* Concept: Quick Sort is another divide and conquer algorithm that selects a pivot element from the array, partitions the array into two sub-arrays (elements less than the pivot and greater than the pivot), and recursively sorts the sub-arrays.
* Time Complexity:
* Best, Average Case: O(n log n).
* Worst Case: O(n²) (when the pivot is poorly chosen, e.g., picking the smallest or largest element as pivot in a sorted array).
* Use Case: Quick Sort is often faster in practice than Merge Sort and Heap Sort, due to smaller constant factors. It's widely used for general-purpose sorting.

**6. Heap Sort**

* Concept: Heap Sort uses a binary heap data structure to sort the array. It builds a max-heap (or min-heap), then repeatedly removes the largest (or smallest) element and places it in the sorted portion of the array.
* Time Complexity:
* Best, Average, Worst Case: O(n log n).
* Use Case: Heap Sort is not stable, but it's an in-place sorting algorithm with O(n log n) time complexity. It is often used when auxiliary space needs to be minimized.

**7. Radix Sort**

* Concept: Radix Sort is a non-comparative sorting algorithm that sorts numbers digit by digit, starting from the least significant digit (LSD) or most significant digit (MSD). It uses counting sort as a subroutine.
* Time Complexity:
* Best, Average, Worst Case: O(nk), where k is the number of digits in the largest number.
* Use Case: Radix Sort works best when the numbers are small or the data set has a limited range of values. It is often used in applications involving large integers or strings.

**8. Counting Sort**

* Concept: Counting Sort is a non-comparative sorting algorithm that counts the frequency of each element in the array and then uses this count to place elements in their correct position.
* Time Complexity:
* Best, Average, Worst Case: O(n + k), where n is the number of elements and k is the range of the input.
* Use Case: Ideal for sorting integers or categorical data, especially when the range of values (k) is small compared to the number of elements (n).

**9. Bucket Sort**

* Concept: Bucket Sort divides the elements into a finite number of "buckets", sorts each bucket individually (usually using a different sorting algorithm like insertion sort), and then combines the results.
* Time Complexity:
* Best, Average Case: O(n + k).
* Worst Case: O(n²) if the buckets are poorly distributed.
* Use Case: Best when the input is uniformly distributed over a range.

**Problem Patterns in Sorting Algorithms**

**1. Sort an Array of Integers**

* Problem: Given an unsorted array, sort it in ascending or descending order.
* Algorithm: Apply any of the sorting algorithms like Merge Sort, Quick Sort, or Heap Sort.
* Time Complexity: O(n log n) (for efficient sorting algorithms).

**2. Kth Smallest or Largest Element**

* Problem: Find the k-th smallest or largest element in an unsorted array.
* Algorithm: Use QuickSelect, which is a variation of Quick Sort, to find the k-th element in O(n) time on average.
* Time Complexity: O(n) (on average for QuickSelect).

**3. Merge Intervals**

* Problem: Given a collection of intervals, merge any overlapping intervals.
* Algorithm: Sort the intervals by the start point, and then iterate through the intervals to merge overlapping ones.
* Time Complexity: O(n log n) (for sorting), followed by O(n) (for merging).

**4. Sort an Array of Strings by Length**

* Problem: Given an array of strings, sort the strings based on their lengths.
* Algorithm: Use any sorting algorithm (e.g., Merge Sort) with a custom comparison function that compares string lengths.
* Time Complexity: O(n log n).

**5. Sort Colors (Dutch National Flag Problem)**

* Problem: Given an array of colors (represented as integers 0, 1, 2), sort the array such that all 0s come first, followed by 1s, and then 2s.
* Algorithm: Use three-way partitioning with Dutch National Flag algorithm (using three pointers).
* Time Complexity: O(n).

**6. Find Duplicate Elements in an Array**

* Problem: Given an unsorted array, find duplicate elements.
* Algorithm: Sorting the array and then checking adjacent elements for duplicates.
* Time Complexity: O(n log n) (due to sorting).

**7. Median of Two Sorted Arrays**

* Problem: Given two sorted arrays, find their median.
* Algorithm: Use Binary Search on the smaller array and adjust the partitioning of both arrays to find the median.
* Time Complexity: O(log(min(n, m))), where n and m are the sizes of the two arrays.

**8. Sort a Nearly Sorted Array (K-Sorted Array)**

* Problem: Given an array where each element is at most k positions away from its sorted position, sort the array efficiently.
* Algorithm: Use a min-heap to extract the minimum element from the top k elements efficiently.
* Time Complexity: O(n log k).

**Advanced Sorting Algorithms**

**1. Tim Sort**

* Concept: TimSort is a hybrid sorting algorithm based on Merge Sort and Insertion Sort. It is the default sorting algorithm used in Java's Arrays.sort() and Python’s sorted().
* It first divides the array into small blocks and sorts them using Insertion Sort.
* Then it merges the blocks using Merge Sort.
* Time Complexity:
* Best Case: O(n) (when the array is already nearly sorted).
* Worst Case: O(n log n).
* Use Case: Best for real-world data that may already be partially sorted.

**2. Block Sort**

* Concept: Block Sort works by dividing the array into blocks and sorting each block individually. The blocks are then merged together.
* Time Complexity:
* Best Case: O(n log n).
* Worst Case: O(n²).
* Use Case: Works well in some data streaming applications where sorting is required with a low memory footprint.

**3. Shell Sort**

* Concept: Shell Sort is an extension of Insertion Sort that allows the exchange of items that are far apart. It starts with large gaps between elements and reduces the gap progressively.
* Time Complexity:
* Best Case: O(n log n) (depends on the gap sequence used).
* Worst Case: O(n²).
* Use Case: Effective for medium-sized arrays where Insertion Sort may perform poorly.

**Chapter 9: Greedy Algorithms**

**Concept Overview of Greedy Algorithms:**

* Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. The key idea is to take the best immediate, or local, solution for each subproblem.

**Greedy Algorithms Key Properties:**

* Greedy Choice Property: A global optimum can be arrived at by selecting a local optimum.
* Optimal Substructure: A problem is said to have an optimal substructure if the optimal solution to the problem can be constructed efficiently from optimal solutions to its subproblems.

**Popular Greedy Algorithms**

**1. Activity Selection Problem**

* Concept: Given a set of activities with their start and finish times, the goal is to select the maximum number of activities that don’t overlap.
* Approach: Sort the activities by their finish times and iteratively select activities that start after the last selected one finishes.
* Time Complexity: O(n log n) (for sorting the activities).
* Use Case: Optimizing scheduling tasks or events in a timeline.

**2. Fractional Knapsack Problem**

* Concept: Given a set of items with weights and values, and a knapsack with a weight capacity, the task is to determine the maximum value that can be carried in the knapsack. Items can be broken into fractions.
* Approach: Calculate the value per unit weight for each item, sort the items by value per weight in descending order, and select items greedily until the knapsack is full.
* Time Complexity: O(n log n) (for sorting).
* Use Case: Optimizing storage or shipment problems where fractions of items can be taken.

**3. Job Sequencing Problem**

* Concept: Given a set of jobs, each with a profit and deadline, the goal is to schedule the jobs to maximize profit. Each job takes one unit of time, and no two jobs can be scheduled at the same time.
* Approach: Sort the jobs by profit, and schedule the job at the latest available time before its deadline.
* Time Complexity: O(n log n) (for sorting).
* Use Case: Task scheduling in environments like cloud computing or job scheduling systems.

**4. Huffman Coding**

* Concept: Given a set of characters with their frequencies, the goal is to construct the optimal prefix code for encoding. This problem is used in data compression algorithms.
* Approach: Use a min-heap to combine the least frequent characters into a binary tree to form an optimal encoding.
* Time Complexity: O(n log n) (for constructing the Huffman tree).
* Use Case: Compression algorithms (like ZIP files, JPEG, and MP3).

**5. Prim’s Algorithm (Minimum Spanning Tree)**

* Concept: Given a graph with weighted edges, the goal is to find a minimum spanning tree (MST) that connects all the vertices with the minimum possible total edge weight.
* Approach: Start with any vertex, and add the smallest edge that connects a vertex in the MST to a vertex outside the MST.
* Time Complexity: O(E log V) (with a priority queue implementation).
* Use Case: Networking (for minimizing cable length in connecting multiple devices) and designing communication networks.

**6. Kruskal’s Algorithm (Minimum Spanning Tree)**

* Concept: Similar to Prim's, but instead of starting with a vertex, we start with edges. The idea is to sort all edges by weight and keep adding edges to the MST, ensuring no cycles are formed (using Union-Find).
* Approach: Sort edges and use Union-Find to efficiently check for cycles.
* Time Complexity: O(E log E) (sorting edges).
* Use Case: Network design and resource optimization problems.

**7. Dijkstra’s Algorithm (Shortest Path)**

* Concept: Given a graph with weighted edges, the goal is to find the shortest path from a source vertex to all other vertices.
* Approach: Use a priority queue to always select the vertex with the smallest tentative distance.
* Time Complexity: O(E log V) (with a binary heap).
* Use Case: Navigation systems, routing in computer networks.

**8. Coin Change Problem (Minimum Coins)**

* Concept: Given a set of coin denominations, determine the minimum number of coins needed to make a given amount.
* Approach: Greedily select the largest denomination coin less than or equal to the remaining amount, and repeat until the amount is reduced to zero.
* Time Complexity: O(n) for a sorted list of coins, but can be O(n log n) if additional structures are needed for dynamic input.
* Use Case: Optimizing cash transactions or monetary systems.

**9. Minimum Number of Platforms (Train Station Problem)**

* Concept: Given train arrival and departure times, find the minimum number of platforms needed at a train station so that no trains have to wait.
* Approach: Sort the arrival and departure times, then use two pointers to track the number of platforms required at each time.
* Time Complexity: O(n log n) (due to sorting).
* Use Case: Optimizing public transportation systems or resource allocation problems.

**Problem Patterns in Greedy Algorithms**

**1. Activity Selection**

* Problem: Given a set of activities with start and finish times, select the maximum number of activities that do not overlap.
* Algorithm: Sort by finish time, and then select activities greedily.
* Time Complexity: O(n log n).

**2. Coin Change (Greedy Method)**

* Problem: Given a set of coins and a value, find the minimum number of coins needed to make up the value.
* Algorithm: Greedily select the largest coin that is less than or equal to the remaining value.
* Time Complexity: O(n) (for sorted coins).

**3. Scheduling Jobs**

* Problem: Given jobs with deadlines and profits, find the maximum profit possible if only one job can be scheduled at a time.
* Algorithm: Sort jobs by profit, and schedule them greedily.
* Time Complexity: O(n log n).

**4. Huffman Encoding**

* Problem: Given a set of characters and their frequencies, build an optimal prefix code.
* Algorithm: Use a min-heap to construct the tree by always combining the two least frequent characters.
* Time Complexity: O(n log n).

**5. Shortest Path (Dijkstra’s Algorithm)**

* Problem: Given a graph with weighted edges, find the shortest path from the source vertex to all other vertices.
* Algorithm: Use a priority queue to greedily choose the vertex with the minimum tentative distance.
* Time Complexity: O(E log V).

**6. Minimum Spanning Tree (Kruskal’s and Prim’s Algorithm)**

* Problem: Given a graph with weights, find the minimum spanning tree.
* Algorithm: Kruskal’s (sort edges and use Union-Find) or Prim’s (use a priority queue).
* Time Complexity: O(E log E) for Kruskal’s, O(E log V) for Prim’s.

**Advanced Greedy Algorithms**

**1. Job Scheduling with Deadlines**

* Problem: Given jobs with deadlines and profits, find the maximum profit we can earn by scheduling jobs within their deadlines.
* Algorithm: Sort jobs by profit and schedule them in the available slots starting from the latest deadline.
* Time Complexity: O(n log n).

**2. Minimum Spanning Tree (Prim's with Fibonacci Heap)**

* Problem: Efficiently find the minimum spanning tree for a graph with large edge weights.
* Algorithm: Use Fibonacci Heap to improve the performance of Prim's algorithm from O(E log V) to O(E + V log V).
* Time Complexity: O(E + V log V) (with Fibonacci Heap).

**3. Median of Two Sorted Arrays (Greedy Method)**

* Problem: Find the median of two sorted arrays.
* Algorithm: Partition the arrays and compare elements across the partition.
* Time Complexity: O(log(min(n, m))) where n and m are the sizes of the two arrays.

**Chapter 10: Recursion**

**Concepts Overview: Recursion**

Recursion is a technique in which a function calls itself in order to solve a problem. It breaks a problem into smaller subproblems and solves each subproblem recursively.

**Key Concepts:**

* Base Case: The condition under which the recursion stops. Without a base case, the recursion will continue indefinitely.
* Recursive Case: The part where the function calls itself with modified parameters to solve smaller subproblems.
* Stack Overflow: If the recursion depth is too large, it can cause a stack overflow due to too many function calls on the call stack.

**Recursion Techniques**

**a. Direct Recursion:**

* Definition: The function directly calls itself.
* Example: Factorial calculation.

**b. Indirect Recursion:**

* Definition: One function calls another function, which in turn calls the first function.
* Example: Mutual recursion in even-odd number checking.

**c. Tail Recursion:**

* Definition: A recursive function where the recursive call is the last operation in the function, allowing for optimization to iterative loops by the compiler.
* Example: Tail-recursive factorial.

**Common Recursion Patterns**

**a. Divide and Conquer:**

* A problem is divided into smaller subproblems that are easier to solve.
* Example: Merge Sort, Quick Sort, Binary Search.

**b. Backtracking:**

* Solving problems by trying out all possibilities and "backtracking" when a solution is invalid.
* Example: N-Queens, Sudoku Solver, Subset Sum Problem.

**Problem Patterns Involving Recursion**

**1. Factorial**

* Problem: Find the factorial of a given number n.
* Recursive Approach: n! = n \* (n-1)!
* Time Complexity: O(n).
* Base Case: 0! = 1.
* Example: factorial(5) returns 120.

**2. Fibonacci Sequence**

* Problem: Find the nth Fibonacci number.
* Recursive Approach: Fib(n) = Fib(n-1) + Fib(n-2)
* Time Complexity: O(2^n) (naive recursive), O(n) (with memoization).
* Base Case: Fib(0) = 0, Fib(1) = 1.
* Example: fib(6) returns 8.

**3. Binary Search**

* Problem: Given a sorted array, find the position of a target element.
* Recursive Approach: Compare target with the middle element and recursively search in the left or right half based on the comparison.
* Time Complexity: O(log n).
* Base Case: If the array has only one element or if the search range is invalid.
* Example: Search for target = 5 in a sorted array [1, 2, 5, 6, 8].

**4. Merge Sort**

* Problem: Sort an array using the divide and conquer technique.
* Recursive Approach: Divide the array into two halves, recursively sort each half, and merge the results.
* Time Complexity: O(n log n).
* Base Case: Arrays of size 1 or 0 are already sorted.
* Example: Sorting [5, 3, 8, 1] results in [1, 3, 5, 8].

**5. Quick Sort**

* Problem: Sort an array using partitioning around a pivot element.
* Recursive Approach: Choose a pivot, partition the array, then recursively sort the left and right subarrays.
* Time Complexity: O(n log n) (average), O(n^2) (worst case).
* Base Case: Arrays of size 1 or 0 are already sorted.
* Example: Sorting [3, 6, 8, 10] results in [3, 6, 8, 10].

**6. N-Queens Problem**

* Problem: Place n queens on an n x n chessboard such that no two queens attack each other.
* Recursive Approach: Place a queen in a row, then recursively try placing queens in subsequent rows.
* Time Complexity: O(n!) (due to permutations of queens).
* Base Case: When all n queens are placed successfully.
* Example: For n = 4, one solution is placing queens at (0,1), (1,3), (2,0), (3,2).

**7. Tower of Hanoi**

* Problem: Move n disks from source peg to destination peg using an auxiliary peg, while following the movement rules.
* Recursive Approach: Move n-1 disks to auxiliary peg, move the nth disk to destination, then move n-1 disks from auxiliary peg to destination.
* Time Complexity: O(2^n).
* Base Case: Move 1 disk directly.

**8. Subset Sum Problem**

* Problem: Given a set of integers, find if there is a subset whose sum is equal to a target value.
* Recursive Approach: For each element, either include it in the subset or exclude it, and recursively check both choices.
* Time Complexity: O(2^n).
* Base Case: If the target sum is zero or all elements are considered.

**9. Path Finding (Maze Problems)**

* Problem: Find a path from the start to the end in a maze.
* Recursive Approach: Move recursively in all possible directions and backtrack when a dead end is reached.
* Time Complexity: O(4^n) (depending on the maze size).
* Base Case: If the destination is reached.

**Advanced Recursion Concepts**

**a. Memoization and Dynamic Programming**

* Memoization: Storing the results of expensive function calls and reusing them when the same inputs occur again.
* Dynamic Programming: A technique where a problem is broken into overlapping subproblems, and solutions to these subproblems are stored for reuse.

**b. Tail Recursion**

* Definition: A form of recursion where the recursive call is the last operation performed. It can be optimized into an iterative loop.
* Example: Tail-recursive factorial.

**c. Recursion vs Iteration**

* Recursion: Easier to implement for problems with a natural recursive structure (e.g., tree traversal).
* Iteration: Can be more efficient for simple looping problems but less intuitive for problems that involve recursion.

**Problem Patterns in Recursion**

**1. Divide and Conquer**

* Problem Examples: Merge Sort, Quick Sort, Binary Search.
* Time Complexity: O(n log n) (for Divide and Conquer problems).

**2. Backtracking**

* Problem Examples: N-Queens, Subset Sum, Sudoku Solver.
* Time Complexity: O(2^n) or O(n!) depending on the problem.

**3. Tree Traversal**

* Problem Examples: In-order, Pre-order, Post-order tree traversal.
* Time Complexity: O(n) for all tree traversals.

**4. Combination and Permutation Generation**

* Problem Examples: Generate all subsets, Generate all permutations of a set.
* Time Complexity: O(2^n) or O(n!) depending on the problem.

**Time Complexity Summary for Recursion**

* Factorial: O(n)
* Fibonacci: O(2^n) (naive), O(n) (with memoization)
* Tower of Hanoi: O(2^n)
* Binary Search: O(log n)
* Merge Sort: O(n log n)
* Quick Sort: O(n log n) (average), O(n^2) (worst case)
* N-Queens Problem: O(n!)
* Subset Sum Problem: O(2^n)

**Best Practices for Recursion**

* Always define a base case to stop recursion.
* Memoize intermediate results to optimize overlapping subproblems.
* Use tail recursion where possible to prevent stack overflow issues.
* Choose recursion when the problem naturally fits a recursive structure, such as tree traversal or problems involving subproblems like divide and conquer.

**Chapter 11: Backtracking**

**Concepts and problem Patterns**

Backtracking is a general algorithmic technique that builds up solutions incrementally, abandoning a solution as soon as it determines that the solution cannot be completed.

**Key Concepts:**

* Exploring Choices: At each step, you explore all possible choices for a solution.
* Pruning: If you reach an invalid or incomplete solution, you backtrack by undoing the last decision and trying the next possibility.
* State Space Tree: The recursive structure used in backtracking, where each node represents a partial solution.
* Feasibility Check: At each step, a check is done to see whether the current state can lead to a valid solution.

**Backtracking Techniques**

**a. Constructing Solutions Step-by-Step:**

* The algorithm tries to build a solution incrementally, exploring each possible solution by adding one piece at a time.
* If at any point a partial solution becomes invalid or does not lead to a complete solution, the algorithm backtracks.

**b. Recursive Depth-First Search (DFS):**

* Backtracking is often implemented as a depth-first search, where you explore one solution as deeply as possible before backtracking to explore alternative solutions.
* Example: N-Queens problem, where you explore placing queens in each row and backtrack if a conflict arises.

**c. Pruning Invalid Solutions:**

* As you explore the solution space, pruning is done to avoid wasting time on invalid or impossible solutions.
* Example: In the N-Queens problem, if placing a queen in a position causes a conflict, that branch of the solution tree is pruned.

**Problem Patterns Involving Backtracking**

**1. N-Queens Problem**

* Problem: Place n queens on an n x n chessboard such that no two queens threaten each other.
* Recursive Approach: Try placing a queen in each row and column recursively. If it leads to a solution, proceed to the next row; otherwise, backtrack.
* Time Complexity: O(n!) (since we check every permutation of queens).
* Example: For n = 4, one solution is placing queens at (0,1), (1,3), (2,0), (3,2).

**2. Subset Sum Problem**

* Problem: Given a set of integers, find if there is a subset whose sum equals a target value.
* Recursive Approach: For each element, either include it in the subset or exclude it. Recursively check both choices.
* Time Complexity: O(2^n) (since we explore all subsets).
* Example: Given [3, 5, 7, 9] and target 12, one valid subset is [3, 9].

3. **Permutations**

* Problem: Generate all permutations of a given set of numbers or characters.
* Recursive Approach: For each element, swap it with the current element, and then recursively generate permutations for the remaining elements.
* Time Complexity: O(n!) (since there are n! permutations).
* Example: For input [1, 2, 3], the permutations are [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].

**4. Combination Sum**

* Problem: Given an array of distinct integers and a target, find all unique combinations in the array where the numbers sum up to the target.
* Recursive Approach: Try each number, include it in the current combination, and recursively try adding the next number. Backtrack if the sum exceeds the target.
* Time Complexity: O(2^n) (as we try all combinations).
* Example: For input [2, 3, 5] and target 8, the combinations are [3, 5] and [2, 3, 3].

**5. Sudoku Solver**

* Problem: Solve a given Sudoku puzzle by filling the empty cells with numbers 1-9 such that the numbers do not repeat in any row, column, or 3x3 subgrid.
* Recursive Approach: Try placing numbers in empty cells one by one. If a number violates Sudoku rules, backtrack to the previous state and try a different number.
* Time Complexity: O(9^(n^2)) (since there are n^2 cells and for each, we try 9 possibilities).
* Example: Solve a partially filled Sudoku puzzle.

**6. Word Search in a Grid**

* Problem: Given a 2D board and a word, find if the word exists in the board. You can start from any cell and move to adjacent cells.
* Recursive Approach: Explore each adjacent cell and recursively check if the word can be formed.
* Time Complexity: O(m \* n \* 4^L) (where m and n are the dimensions of the grid and L is the length of the word).
* Example: For a board of size 3x3, search for the word "ABCCED".

**Advanced Backtracking Concepts**

**a. Pruning the Search Tree**

* To make backtracking more efficient, you can prune the search tree by abandoning paths that are not promising early on.
* Example: In the N-Queens problem, if placing a queen in a row leads to an attack on another queen, that path is abandoned.

**b. Path Compression**

* In some problems, you can optimize by reducing the state space. For example, in the subset sum problem, avoid checking already visited states.

**c. Cutting Off Unnecessary Work (Branch and Bound)**

* This is a generalization of backtracking where we cut off branches of the search tree that cannot lead to a better solution than the best one found so far.
* Example: Solving optimization problems like the Traveling Salesman Problem (TSP) using branch and bound.

**Problem Patterns in Backtracking**

**1. Combination and Permutation Generation**

* Problem Examples: Generating subsets, permutations of a set.
* Time Complexity: O(2^n) or O(n!) depending on the problem.

**2. Path Finding Problems**

* Problem Examples: Maze traversal, Word search in a grid.
* Time Complexity: O(m \* n \* 4^L) for word search (where m \* n is the grid size, and L is the word length).

**3. Optimizing Search Space**

* Problem Examples: N-Queens, Sudoku Solver, Subset Sum Problem.
* Time Complexity: O(n!) for N-Queens, O(2^n) for Subset Sum problems.

**Time Complexity Summary for Backtracking**

* N-Queens: O(n!) (due to checking each permutation of queens).
* Subset Sum: O(2^n) (since we explore all subsets).
* Permutations: O(n!) (due to generating all permutations).
* Combination Sum: O(2^n) (as we try all combinations).
* Sudoku Solver: O(9^(n^2)) (with backtracking).
* Word Search in Grid: O(m \* n \* 4^L) (searching for word in a grid of size m x n with length L of the word).

**Best Practices for Backtracking**

* Start with a simple solution: Begin by solving a simpler version of the problem and build from there.
* Prune early: As soon as you detect an invalid or suboptimal state, backtrack to save time.
* Use DFS: Backtracking is often implemented using depth-first search.
* Avoid revisiting states: Use a set to store visited states or constraints to avoid redundant work.

**Chapter 12: Dynamic Programming**

**Concept and Problem Patterns:**

Dynamic Programming is an optimization technique used to solve problems by breaking them down into simpler subproblems and storing the results of overlapping subproblems to avoid redundant calculations.

**Key Concepts:**

* Optimal Substructure: The problem can be broken down into smaller subproblems, which can be solved independently.
* Overlapping Subproblems: The same subproblems are solved multiple times, so storing the results of previous subproblems helps in reducing the time complexity.
* Memoization (Top-Down Approach): Store the results of subproblems as they are computed to avoid redundant calculations.
* Tabulation (Bottom-Up Approach): Build up the solution iteratively from smaller subproblems to the final solution.
* State Representation: Define the problem in terms of state variables, which represent the solution to subproblems.

**Types of DP Approaches**

**a. Top-Down Approach (Memoization)**

* In the top-down approach, we recursively solve the problem by breaking it down into smaller subproblems and store the results in a memoization table (usually a dictionary or array).
* Time Complexity: Depends on the number of unique subproblems, typically O(n) or O(n²).

**b. Bottom-Up Approach (Tabulation)**

* The bottom-up approach starts by solving the simplest subproblems and gradually solving larger subproblems. The results of previous subproblems are stored in a table, and the final solution is built iteratively.
* Time Complexity: Usually O(n) or O(n²) depending on the problem.

**Problem Patterns Involving Dynamic Programming**

**1. Fibonacci Sequence**

* Problem: Compute the nth Fibonacci number.
* Approach: Use a bottom-up approach to compute each Fibonacci number from 0 to n, storing the results in an array.
* Time Complexity: O(n) (due to iterative calculation of Fibonacci numbers).
* Example: fib(5) results in 5 (the 5th Fibonacci number).

**2. 0/1 Knapsack Problem**

* Problem: Given a set of items, each with a weight and value, determine the maximum value you can carry in a knapsack with a given weight capacity.
* Approach: Use dynamic programming to build up solutions for smaller capacities and items, storing the results in a table.
* Time Complexity: O(nW) (where n is the number of items and W is the maximum capacity).
* Example: For items with values [60, 100, 120] and weights [10, 20, 30] and capacity 50, the maximum value is 220.

**3. Longest Common Subsequence (LCS)**

* Problem: Given two sequences, find the longest subsequence common to both.
* Approach: Use a table to store the lengths of common subsequences for all pairs of substrings.
* Time Complexity: O(m \* n) (where m and n are the lengths of the two sequences).
* Example: For sequences ABCDGH and AEDFHR, the LCS is ADH.

**4. Longest Increasing Subsequence (LIS)**

* Problem: Find the length of the longest increasing subsequence in a given array.
* Approach: Use dynamic programming to store the length of the LIS ending at each index and iteratively update it.
* Time Complexity: O(n²) (with an optimized approach of O(n log n)).
* Example: For input [10, 22, 9, 33, 21, 50, 41, 60, 80], the length of the LIS is 6.

**5. Coin Change Problem**

* Problem: Given a set of coin denominations and a target amount, find the minimum number of coins required to make the target amount.
* Approach: Use dynamic programming to find the minimum coins for each amount from 1 to the target value.
* Time Complexity: O(n \* m) (where n is the target value and m is the number of coin denominations).
* Example: For coins [1, 2, 5] and target 11, the minimum number of coins is 3 (2 + 5 + 5).

**6. Matrix Chain Multiplication**

* Problem: Given a sequence of matrices, determine the most efficient way to multiply them, minimizing the number of scalar multiplications.
* Approach: Use dynamic programming to compute the minimum number of multiplications needed for multiplying subchains of matrices.
* Time Complexity: O(n³) (where n is the number of matrices).
* Example: For matrices A1 = 10x20, A2 = 20x30, and A3 = 30x40, the minimum number of multiplications is 18000.

**7. Word Break Problem**

* Problem: Given a string and a dictionary of words, determine if the string can be segmented into words from the dictionary.
* Approach: Use dynamic programming to store whether the substring up to index i can be segmented.
* Time Complexity: O(n²) (where n is the length of the string).
* Example: For the string applepie and dictionary ["apple", "pie"], the string can be segmented as apple pie.

**Advanced Dynamic Programming Concepts**

**a. DP with Bitmasking**

* Problem: Use dynamic programming with bitmasking to solve problems where subsets or subsets of items need to be considered.
* Example: Travelling Salesman Problem (TSP), where bitmasking represents the set of visited cities.
* Time Complexity: O(n \* 2^n) for TSP (since we check each combination of cities).

**b. DP with Binary Search**

* Problem: Sometimes dynamic programming problems can be optimized with binary search.
* Example: Longest Increasing Subsequence (LIS) can be solved in O(n log n) using binary search in combination with dynamic programming.
* Time Complexity: O(n log n).

**c. Space Optimization in DP**

* Problem: In some problems, we can optimize space complexity by storing only necessary states.
* Example: In the 0/1 Knapsack problem, instead of using a 2D table, we can optimize space to O(W) by only keeping the current and previous rows.

**Time Complexity Summary for DP**

* Fibonacci Sequence: O(n) (using bottom-up).
* 0/1 Knapsack: O(nW) (where n is the number of items and W is the capacity).
* LCS (Longest Common Subsequence): O(m \* n) (where m and n are the lengths of the two sequences).
* Longest Increasing Subsequence: O(n²) (with an optimized approach of O(n log n)).
* Coin Change Problem: O(n \* m) (where n is the target value and m is the number of coin denominations).
* Matrix Chain Multiplication: O(n³) (where n is the number of matrices).
* Word Break Problem: O(n²) (where n is the length of the string).
* TSP with DP + Bitmasking: O(n \* 2^n).

**Best Practices for Dynamic Programming**

* Identify the Subproblems: Clearly define the subproblems and the relationships between them.
* Memoization vs. Tabulation: Choose memoization when solving top-down problems recursively and tabulation when iteratively building up solutions.
* State Representation: Use state variables to represent the subproblem and solve progressively.
* Optimize Space: Use techniques like space optimization and avoid unnecessary storage of states.

**Chapter 13: Bit Manipulation**

**Concept Overview: Bit Manipulation**

Bit Manipulation involves using bitwise operators (AND, OR, XOR, NOT, left shift, and right shift) to solve problems at the bit level. It allows you to perform operations on numbers at a more efficient, lower level, and often leads to more optimized solutions.

**Key Concepts:**

* Bitwise Operators: Perform operations directly on bits of integers.
* AND (&): Sets each bit to 1 if both corresponding bits are 1.
* OR (|): Sets each bit to 1 if at least one of the corresponding bits is 1.
* XOR (^): Sets each bit to 1 if the corresponding bits are different.
* NOT (~): Inverts all the bits.
* Left Shift (<<): Shifts bits to the left, effectively multiplying the number by 2.
* Right Shift (>>): Shifts bits to the right, effectively dividing the number by 2 (for positive numbers).
* Bitmasking: Technique of using bits to represent sets or subsets, and efficiently perform operations like checking membership, adding, or removing elements.
* Set Representation: Representing subsets or combinations of elements as binary numbers, where each bit represents the presence or absence of an element.
* Parity: Using bitwise operations to determine whether a number has an even or odd number of 1-bits.
* Hamming Distance: The number of positions at which the corresponding bits are different.

**Key Bit Manipulation Techniques**

**a. Checking if a Number is Odd or Even**

* Method: Use bitwise AND with 1. If num & 1 == 0, the number is even; otherwise, it is odd.
* Time Complexity: O(1).

**b. Counting Set Bits (Hamming Weight)**

* Method: Use the operation num & (num - 1), which removes the rightmost set bit. Repeat until the number becomes 0.
* Time Complexity: O(k), where k is the number of set bits in the number (also known as the Hamming weight).

**c. Toggle a Specific Bit**

* Method: Use XOR to toggle the bit at a particular position. For example, to toggle the i-th bit, use num ^= (1 << i).
* Time Complexity: O(1).

**d. Set a Specific Bit**

* Method: Use bitwise OR to set a particular bit. For example, to set the i-th bit, use num |= (1 << i).
* Time Complexity: O(1).

**e. Clear (Reset) a Specific Bit**

* Method: Use bitwise AND with the negation of the bit you want to reset. To clear the i-th bit, use num &= ~(1 << i).
* Time Complexity: O(1).

**f. Check if a Bit is Set**

* Method: Use bitwise AND to check if a specific bit is set. For the i-th bit, use num & (1 << i); if the result is non-zero, the bit is set.
* Time Complexity: O(1).

**g. Left and Right Shift**

* Method: Shifting bits left or right allows multiplication or division by powers of 2.
* Left Shift (<<): num << 1 (equivalent to multiplying by 2).
* Right Shift (>>): num >> 1 (equivalent to dividing by 2 for positive numbers).
* Time Complexity: O(1) for each shift operation.

**Problem Patterns Involving Bit Manipulation**

**1. Find the Only Non-Duplicate Element (XOR Approach)**

* Problem: Given an array where every element appears twice except one, find the element that appears once.
* Approach: Use XOR to cancel out all duplicate elements. The result will be the non-duplicate element.
* Time Complexity: O(n) (where n is the size of the array).
* Example: For array [4, 5, 4, 5, 7], the result is 7.

**2. Count Set Bits (Hamming Weight)**

* Problem: Given an integer, count the number of 1 bits it has.
* Approach: Use the operation num & (num - 1) to remove the rightmost set bit and repeat the process.
* Time Complexity: O(k), where k is the number of set bits.
* Example: For the number 15 (binary 1111), the result is 4.

**3. Power of Two Check**

* Problem: Given an integer, determine if it is a power of two.
* Approach: A number is a power of two if and only if it has exactly one bit set. You can check this with the expression num & (num - 1) == 0.
* Time Complexity: O(1).
* Example: For 8 (binary 1000), the result is True.

**4. Reverse Bits**

* Problem: Given a number, reverse its bits.
* Approach: Use bitwise shifts and OR to build the reversed number.
* Time Complexity: O(32) (since integers are 32 bits in most programming languages).
* Example: For input 43261596 (binary 00000010100101000001111010011100), the output is 964176192 (binary 00111001011110000010100101000000).

**5. Find the Two Non-Duplicate Elements**

* Problem: Given an array where all elements appear twice except for two, find the two non-duplicate elements.
* Approach: XOR all elements to cancel out the duplicates, then use the result to divide the array into two parts and find the two numbers.
* Time Complexity: O(n).
* Example: For array [1, 2, 3, 2, 1, 4], the result is 3 and 4.

**6. Subset Generation Using Bitmasking**

* Problem: Generate all subsets of a given set.
* Approach: Use integers as bitmasks, where each bit represents whether an element is included in the subset or not.
* Time Complexity: O(2^n), where n is the number of elements.
* Example: For input [1, 2, 3], the subsets are [], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3].

**7. Find the Hamming Distance**

* Problem: Given two integers, find the number of positions where their bits differ.
* Approach: XOR the two numbers and count the number of 1s in the result.
* Time Complexity: O(1) for XOR operation + O(k) for counting set bits, where k is the number of bits.
* Example: For integers 1 (0001) and 4 (0100), the Hamming distance is 2.

**Advanced Bit Manipulation Concepts**

**a. Bitmasking with Dynamic Programming**

* Problem: Use bitmasking to represent subsets in dynamic programming, especially in optimization problems.
* Example: Traveling Salesman Problem (TSP) where the state is represented by a bitmask to track visited cities.
* Time Complexity: O(n \* 2^n) for TSP (where n is the number of cities).

**b. Gray Code**

* Problem: Convert a number to its Gray code representation, where two successive values differ by only one bit.
* Approach: Use the formula GrayCode(num) = num ^ (num >> 1).
* Time Complexity: O(1).

**c. Efficient Modular Exponentiation**

* Problem: Compute (a^b) % mod efficiently.
* Approach: Use Exponentiation by Squaring (iterative method) combined with bitwise operations.
* Time Complexity: O(log b).

**Time Complexity Summary for Bit Manipulation**

* Find Non-Duplicate Element (XOR): O(n).
* Count Set Bits: O(k), where k is the number of set bits.
* Power of Two Check: O(1).
* Reverse Bits: O(32).
* Find Two Non-Duplicate Elements: O(n).
* Subset Generation (Bitmasking): O(2^n).
* Hamming Distance: O(1) for XOR + O(k) for counting set bits.

**Best Practices for Bit Manipulation**

* Understand the Bitwise Operators: Make sure you know how to use &, |, ^, ~, <<, and >> to solve problems.
* Use XOR for Pairwise Elimination: XOR is powerful for eliminating duplicates and finding non-repeating numbers.
* Optimize Space with Bitmasking: Use integers or arrays to represent sets or subsets, as it saves space compared to traditional methods.
* Leverage Shifting for Efficient Computations: Shifting bits can be used for fast multiplication or division by powers of 2.

**Chapter 14. Binary Tree**

**1. Concept Overview: Binary Trees**

A Binary Tree is a hierarchical data structure in which each node has at most two children, commonly referred to as the left and right child. Binary trees are widely used for tasks involving hierarchical data, efficient searching, sorting, and tree-based problems.

**Key Concepts:**

* Binary Tree: A tree in which each node has at most two children.
* Root: The topmost node in the tree.
* Leaf: A node with no children.
* Height of the Tree: The length of the longest path from the root to a leaf.
* Depth of a Node: The distance from the root node to the given node.
* Types of Binary Trees:
* Full Binary Tree: Every node has either 0 or 2 children.
* Complete Binary Tree: All levels are completely filled, except possibly the last level, which is filled from left to right.
* Perfect Binary Tree: All internal nodes have two children, and all leaf nodes are at the same level.
* Balanced Binary Tree: A tree where the height difference between the left and right subtrees is at most one for all nodes.
* Degenerate (or pathological) Tree: A tree in which each parent has only one child, resembling a linked list.
* Binary Search Tree (BST): A binary tree where for every node, the left child’s value is smaller, and the right child’s value is larger than the node’s value.

**2. Binary Tree Traversals**

Binary tree traversal is a way of visiting all the nodes in the tree, typically following a specific order:

**a. Preorder Traversal:**

* Concept: Visit the root, then recursively visit the left subtree, followed by the right subtree.
* Time Complexity: O(n), where n is the number of nodes in the tree.
* Use Case: Used when we need to copy the tree or process the root node first.

**b. Inorder Traversal:**

* Concept: Recursively visit the left subtree, then visit the root, and finally the right subtree.
* Time Complexity: O(n).
* Use Case: Used in Binary Search Trees (BST) to get nodes in sorted order.

**c. Postorder Traversal:**

* Concept: Recursively visit the left subtree, then the right subtree, and finally the root.
* Time Complexity: O(n).
* Use Case: Used for deletion of nodes or calculating the height of the tree.

**d. Level Order Traversal (BFS):**

* Concept: Visit nodes level by level from top to bottom, left to right.
* Time Complexity: O(n).
* Use Case: Used for problems like finding the height of the tree or level-wise processing.

**3. Binary Tree Operations**

**a. Insertion:**

* Concept: Inserting a new node into the tree. In a binary tree, insertion can be done by following a specific strategy depending on the tree type.
* Time Complexity:
* Binary Tree: O(n) in worst case (if the tree is unbalanced).
* BST: O(log n) on average (if balanced).

**b. Deletion:**

* Concept: Deleting a node in a binary tree involves three possible cases:
* The node is a leaf.
* The node has one child.
* The node has two children (in which case, we need to find the in-order successor or in-order predecessor).
* Time Complexity: O(n) (in worst case, unbalanced).

**c. Search:**

* Concept: Searching for a value in the binary tree.
* In a Binary Search Tree (BST): You can utilize the binary search property to search for a node efficiently.
* Time Complexity:
* Binary Tree: O(n).
* BST: O(log n) on average.

**d. Finding the Height of a Binary Tree:**

* Concept: The height is the number of edges in the longest path from the root to a leaf.
* Time Complexity: O(n).

**e. Finding the Depth of a Node:**

* Concept: The depth is the number of edges from the root to a given node.
* Time Complexity: O(n).

**4. Advanced Topics in Binary Trees**

**a. Balanced Binary Trees:**

* Concept: A balanced binary tree ensures that the difference in heights between the left and right subtrees of every node is at most 1.
* AVL Tree: A self-balancing binary search tree where the difference in heights of left and right subtrees is at most 1.
* Time Complexity:
* Insertion, Deletion, and Search: O(log n).

**b. AVL Tree Rotations:**

* Concept: To maintain balance in an AVL tree after insertion or deletion, we use tree rotations:
* Left Rotation: Used to balance the tree when the right subtree is taller.
* Right Rotation: Used to balance the tree when the left subtree is taller.
* Left-Right Rotation: A combination of left and right rotations.
* Right-Left Rotation: A combination of right and left rotations.

**c. Binarization:**

* Concept: Converting a non-binary tree to a binary tree. Used for binary search trees to maintain efficient searching.

**d. Threaded Binary Tree:**

* Concept: A binary tree where the null pointers are replaced by pointers to the next node in an inorder traversal. It’s useful for efficient traversal without using a stack or recursion.

**5. Problem Patterns in Binary Trees**

**1. Binary Tree Height**

* Problem: Find the height of a binary tree.
* Approach: Use DFS or BFS (Level Order Traversal).
* Time Complexity: O(n).

**2. Checking if a Tree is Balanced**

* Problem: Check if a binary tree is height-balanced.
* Approach: Use DFS while calculating the height of subtrees and checking the balance condition.
* Time Complexity: O(n).

**3. Lowest Common Ancestor (LCA)**

* Problem: Given two nodes in a binary tree, find their lowest common ancestor.
* Approach: Use DFS or Binary Lifting technique.
* Time Complexity: O(n) for DFS, O(log n) for Binary Lifting.

**4. Diameter of a Binary Tree**

* Problem: Find the diameter (longest path between two nodes) of a binary tree.
* Approach: Use DFS to calculate the diameter while calculating the height.
* Time Complexity: O(n).

**5. Convert Binary Tree to Linked List**

* Problem: Convert a binary tree into a linked list in preorder or inorder traversal order.
* Approach: Use recursion to perform the conversion.
* Time Complexity: O(n).

**6. Symmetric Tree Check**

* Problem: Check if a binary tree is symmetric around its center (mirror image).
* Approach: Use DFS or BFS to check the mirror symmetry of subtrees.
* Time Complexity: O(n).

**7. Path Sum**

* Problem: Find if there is a path from root to leaf whose sum of node values equals a given target.
* Approach: Use DFS.
* Time Complexity: O(n).

**8. Zigzag Level Order Traversal**

* Problem: Return the nodes of a binary tree in zigzag level order.
* Approach: Use Level Order Traversal with a flag to alternate between left-to-right and right-to-left traversal.
* Time Complexity: O(n).

**6. Time Complexity Summary for Binary Tree Operations**

* Insertion:
* Binary Tree: O(n) (in worst case).
* BST: O(log n) (in average case, balanced).
* Search:
* Binary Tree: O(n).
* BST: O(log n).
* Deletion:
* Binary Tree: O(n).
* BST: O(log n) (in average case).
* Height Calculation: O(n).
* Diameter Calculation: O(n).
* Level Order Traversal: O(n).

**Conclusion**

Binary trees form the foundation of several important algorithms and data structures, including binary search trees (BST), AVL trees, and heap structures. Mastering concepts such as tree traversal, insertion, deletion, and advanced techniques like balancing and LCA will give you a competitive edge for Google DSA interviews. Practice different problem patterns related to binary trees to gain proficiency and solve real-world problems efficiently.

**Chapter 15: Binary Search Tree**

**Concept Overview: Binary Search Tree (BST)**

A Binary Search Tree (BST) is a type of binary tree in which each node follows the property that:

* The left child’s value is less than the parent node’s value.
* The right child’s value is greater than the parent node’s value.

This property allows for efficient searching, insertion, and deletion operations, with average time complexities of O(log n) in a balanced tree.

**Key Concepts:**

* Root: The topmost node in the tree.
* Left Child: The node that is smaller than its parent node.
* Right Child: The node that is larger than its parent node.
* In-order Traversal: Produces the elements in sorted order for a BST.
* Balanced BST: The tree is balanced when the height of the left and right subtrees of any node differ by no more than one.

**Operations on BST:**

* Insertion: Insert a node by following the BST property (left child < parent < right child).
* Search: Efficiently search for a value by traversing the tree based on comparison.
* Deletion: Deleting a node requires handling three cases:
* Node has no children (leaf node).
* Node has one child.
* Node has two children (which requires finding the in-order successor or in-order predecessor).

**2. Binary Search Tree Operations**

**a. Insertion**

* Concept: Insert a new node by recursively comparing the value to be inserted with the current node and deciding whether to go left or right.
* Time Complexity:
* Best case: O(log n) (balanced tree).
* Worst case: O(n) (if the tree is degenerate, resembling a linked list).

**b. Search**

* Concept: Search for a node by comparing the target value to the current node and recursively moving left or right depending on the comparison.
* Time Complexity:
* Best case: O(log n) (balanced tree).
* Worst case: O(n) (if the tree is degenerate).

**c. Deletion**

* Concept: Deletion of a node depends on three cases:
* Case 1: Node is a leaf (no children) - simply remove the node.
* Case 2: Node has one child - replace the node with its child.
* Case 3: Node has two children - find the in-order successor (smallest node in the right subtree) or in-order predecessor (largest node in the left subtree), replace the node with it, and delete the successor/predecessor.
* Time Complexity:
* Best case: O(log n) (balanced tree).
* Worst case: O(n) (if the tree is degenerate).

**d. Finding the Minimum and Maximum**

**Concept**:

* The minimum value is the leftmost node.
* The maximum value is the rightmost node.
* Time Complexity: O(log n) (balanced tree), O(n) (degenerate tree).

**Advanced Topics in Binary Search Trees**

**a. AVL Tree (Self-Balancing BST)**

* Concept: An AVL Tree is a balanced BST where the height difference (balance factor) between the left and right subtrees of any node is at most 1. Rotations are used to maintain balance after insertions or deletions.
* Rotations:
* Left Rotation: Performed when the right subtree is heavier.
* Right Rotation: Performed when the left subtree is heavier.
* Left-Right Rotation and Right-Left Rotation: A combination of two rotations.
* Time Complexity:
* Insertion, Deletion, and Search: O(log n) (due to balancing).

**b. Red-Black Tree (Self-Balancing BST)**

* Concept: A Red-Black Tree is another type of self-balancing BST, where each node has a color (red or black) and follows several properties that guarantee balance.
* Properties:
* Every node is either red or black.
* The root is always black.
* Red nodes cannot have red children (no two red nodes can be adjacent).
* Every path from a node to its descendant NULL nodes has the same number of black nodes.
* Time Complexity:
* Insertion, Deletion, and Search: O(log n).

**c. Splay Tree**

* Concept: A Splay Tree is a self-adjusting BST where every operation (insertion, deletion, search) moves the accessed node to the root through a series of tree rotations (splaying). This ensures that frequently accessed nodes are quick to access.
* Time Complexity:
* Amortized O(log n) for insertion, deletion, and search.

**Problem Patterns in Binary Search Tree**

**1. Search in BST**

* Problem: Find if a value exists in a binary search tree.
* Approach: Traverse the tree starting from the root, comparing the value at each node and recursively traversing left or right.
* Time Complexity: O(log n) (balanced), O(n) (degenerate).

**2. Kth Smallest/Largest Element in BST**

* Problem: Find the kth smallest or kth largest element in a BST.
* Approach: Perform in-order traversal for kth smallest and reverse in-order traversal for kth largest.
* Time Complexity: O(k) for finding the kth smallest/largest (if traversal is done in-order).

**3. LCA (Lowest Common Ancestor) in BST**

* Problem: Find the lowest common ancestor (LCA) of two given nodes in a BST.
* Approach: Traverse the tree from the root, comparing the values. The LCA will be the first node where the values of both nodes split (i.e., one node is on the left and the other is on the right).
* Time Complexity: O(log n) (balanced), O(n) (degenerate).

**4. Check if a Tree is a Valid BST**

* Problem: Verify whether a given binary tree is a valid BST.
* Approach: Perform in-order traversal and check if the sequence of node values is strictly increasing.
* Time Complexity: O(n).

**5. Serialize and Deserialize BST**

* Problem: Convert a BST into a string representation (serialization) and rebuild the tree from the string (deserialization).
* Approach: Use preorder traversal for serialization and recursively reconstruct the tree for deserialization.
* Time Complexity: O(n).

**6. Inorder Successor and Predecessor**

* Problem: Find the in-order successor (next larger node) or in-order predecessor (previous smaller node) of a given node in a BST.
* Approach:
* Successor: If the node has a right child, find the minimum in the right subtree; otherwise, traverse upwards until you find a node that is a left child of its parent.
* Predecessor: If the node has a left child, find the maximum in the left subtree; otherwise, traverse upwards until you find a node that is a right child of its parent.
* Time Complexity: O(log n) (balanced), O(n) (degenerate).
* Time Complexity Summary for Binary Search Tree Operations

|  |  |  |
| --- | --- | --- |
| Operation | Best Case | Worst Case |
| Insertion | O(log n) | O(n) |
| Search | O(log n) | O(n) |
| Deletion | O(log n) | O(n) |
| Find Minimum | O(log n) | O(n) |
| Find Maximum | O(log n) | O(n) |
| In-order Traversal | O(n) | O(n) |
| Pre-order Traversal | O(n) | O(n) |
| Post-order Traversal | O(n) | O(n) |

**Conclusion**

Binary Search Trees (BST) are fundamental to many algorithms and provide efficient solutions for searching, insertion, and deletion operations. Mastering BST concepts such as balancing (AVL, Red-Black Trees), rotations, and problem-solving patterns is essential for Google DSA interviews. Focus on efficient traversal, LCA, kth smallest/largest element, and valid BST checks for a comprehensive understanding of BSTs.

**Chapter 16: Binary Heap**

**Concept Overview: Binary Heap**

* A Binary Heap is a complete binary tree that satisfies the heap property. It is used primarily in priority queues and supports efficient insertions and deletions of the maximum or minimum element.
* Min Heap: The value of each parent node is less than or equal to the values of its children. The minimum element is at the root.
* Max Heap: The value of each parent node is greater than or equal to the values of its children. The maximum element is at the root.
* Binary heaps are complete binary trees, meaning all levels are fully filled except possibly for the last level, which is filled from left to right.

**Operations on Binary Heap**

**a. Insertion**

* Concept: Insert an element into the heap by adding it at the last position (maintaining the complete binary tree property), then "bubble up" (or heapify up) to restore the heap property.
* Time Complexity: O(log n)
* Insertion involves inserting the element at the last level and then bubbling it up, which takes logarithmic time.

**b. Deletion (Pop the Root)**

* Concept: Remove the root element (the maximum in a max heap, or the minimum in a min heap), replace it with the last element in the heap, and then heapify down to restore the heap property.
* Time Complexity: O(log n)
* After replacing the root with the last element, the heap is adjusted by heapify down.

**c. Peek (Get the Root Element)**

* Concept: Return the root element of the heap without modifying the heap.
* Time Complexity: O(1)
* This operation involves simply accessing the root of the heap.

**d. Heapify (Heapify Down)**

* Concept: Heapify is the process of maintaining the heap property. If a node violates the heap property, it is adjusted by swapping it with one of its children (depending on the heap type) until the heap property is restored.
* Heapify Down: Starting from a node, compare it with its children and swap it with the smallest (in Min Heap) or largest (in Max Heap) child, then continue the process until the heap property is satisfied.

**e. Build Heap (Heap Construction)**

* Concept: Build a binary heap from an unordered array by applying heapify from the bottom-up, starting from the last non-leaf node to the root.
* Time Complexity: O(n)
* The build heap operation works in linear time due to heapifying each node.

**Advanced Concepts in Binary Heap**

**a. Priority Queue**

* Concept: A Priority Queue is an abstract data type that supports operations such as insert, remove, and peek. It is typically implemented using a binary heap.
* Min Priority Queue: Removes the element with the smallest priority.
* Max Priority Queue: Removes the element with the largest priority.
* Operations:
* Insert: Insert an element with a given priority.
* Delete Min/Max: Remove the element with the minimum or maximum priority.
* Peek: Retrieve the element with the minimum or maximum priority without removing it.

**b. Heap Sort**

* Concept: Heap sort is an efficient comparison-based sorting algorithm that uses a binary heap. It first builds a max-heap (for descending order) or min-heap (for ascending order) and then extracts elements one by one to build the sorted array.
* Time Complexity: O(n log n)
* The heapify operation is used repeatedly to sort the elements.

**4. Problem Patterns in Binary Heap**

**1. Kth Largest/Smallest Element in an Array**

* Problem: Find the kth largest or kth smallest element in an array using a binary heap.
* Approach: Use a min heap to find the kth largest element or a max heap to find the kth smallest element. You can also use a heap of size k to maintain the k largest or k smallest elements in the array.
* Time Complexity: O(n log k) (heap construction), O(k log k) (pop operations).

**2. Merge K Sorted Lists**

* Problem: Merge k sorted linked lists into a single sorted list.
* Approach: Use a min heap to track the smallest element among the current elements of the k lists. Each time the smallest element is extracted, the next element from the corresponding list is inserted into the heap.
* Time Complexity: O(n log k), where n is the total number of elements across the k lists.

**3. Top K Frequent Elements**

* Problem: Find the k most frequent elements in an array.
* Approach: Use a min heap to keep track of the top k frequent elements. The heap will store elements as (frequency, element) pairs, and you can replace the root if a higher frequency element is encountered.
* Time Complexity: O(n log k).

**4. Connect Ropes with Minimum Cost**

* Problem: Given an array of rope lengths, connect them into one rope with the minimum cost. The cost of connecting two ropes is the sum of their lengths.
* Approach: Use a min heap to repeatedly extract the two shortest ropes, connect them, and insert the new rope back into the heap until only one rope remains.
* Time Complexity: O(n log n).

**5. Time Complexity Summary for Binary Heap Operations**

|  |  |  |
| --- | --- | --- |
| Operation | Best Case | Worst Case |
| Insertion | O(log n) | O(log n) |
| Deletion (Pop Root) | O(log n) | O(log n) |
| Peek (Get Root) | O(1) | O(1) |
| Build Heap | O(n) | O(n) |
| Heapify | O(log n) | O(log n) |
| Heap Sort | O(n log n) | O(n log n) |
| Priority Queue Insert | O(log n) | O(log n) |
| Priority Queue Extract | O(log n) | O(log n) |

**6. Conclusion**

Binary Heaps are essential for implementing efficient priority queues and solving problems related to finding kth largest/smallest elements, heap sort, and merging sorted lists. Mastering operations like insertion, deletion, heapify, and heap construction is crucial for Google DSA interviews. Understanding min-heaps, max-heaps, and their applications in problems will help you solve a wide range of algorithmic challenges.

**Chapter 17: Graph Algorithms**

**Concept Overview: Graph Data Structures**

Graphs are non-linear data structures consisting of nodes (vertices) and edges that connect these nodes. Graphs are used to model relationships between entities in various fields like computer networks, social networks, recommendation systems, etc.

Key Concepts:

**Graph Representation:**

* Adjacency Matrix: A 2D array where each cell represents an edge between two vertices.
* Adjacency List: A list of lists (or a map), where each vertex has a list of vertices to which it is directly connected.

**Types of Graphs:**

* Directed Graph: Edges have a direction, i.e., one-way connections.
* Undirected Graph: Edges are bidirectional, i.e., the connection between two vertices is mutual.
* Weighted Graph: Edges have weights associated with them.
* Unweighted Graph: All edges are considered equal (weight is not important).
* Cyclic vs. Acyclic:
* Cyclic Graph: Contains at least one cycle.
* Acyclic Graph: Contains no cycles (e.g., Tree is an acyclic graph).
* Connected vs. Disconnected:
* Connected Graph: There is a path between every pair of vertices.
* Disconnected Graph: Some vertices are not reachable from others.

**Graph Traversal:**

* Breadth-First Search (BFS): Visits all nodes at the present depth level before moving on to nodes at the next depth level.
* Depth-First Search (DFS): Visits nodes by going deep into the graph before backtracking.

**Graph Properties:**

* Degree of a Vertex: The number of edges incident to the vertex. In a directed graph, it's divided into in-degree and out-degree.
* Connectivity: Determines if there is a path between all pairs of vertices.
* Shortest Path: The path that minimizes the sum of edge weights.
* Special Graphs:
* Trees: A special type of graph without cycles.
* DAG (Directed Acyclic Graph): A directed graph with no cycles, commonly used for modeling workflows or tasks.

**Graph Representation**

**Adjacency Matrix:**

* A 2D array where each element at position (i, j) represents an edge between vertex i and vertex j.
* Time Complexity:
* Space Complexity: O(V^2), where V is the number of vertices.
* Checking if an edge exists: O(1).
* Inserting an edge: O(1).
* Traversing all edges: O(V^2).

**Adjacency List:**

* An array (or hashmap) where each vertex has a list of adjacent vertices.
* Time Complexity:
* Space Complexity: O(V + E), where V is the number of vertices and E is the number of edges.
* Checking if an edge exists: O(V) in worst case.
* Inserting an edge: O(1).
* Traversing all edges: O(V + E).

**Graph Traversal Techniques**

**a. Depth-First Search (DFS)**

* Concept: DFS explores a graph by going as deep as possible along a branch before backtracking.
* Applications: Finding connected components, topological sorting, cycle detection in undirected graphs, pathfinding.
* Implementation: Can be implemented using Recursion or Stack.
* Time Complexity:
* O(V + E), where V is the number of vertices and E is the number of edges.

**b. Breadth-First Search (BFS)**

* Concept: BFS explores a graph level by level, starting from the source node and visiting all adjacent vertices before moving to the next level.
* Applications: Shortest path in unweighted graphs, finding the shortest number of edges between two nodes.
* Implementation: Typically implemented using a Queue.
* Time Complexity:
* O(V + E), where V is the number of vertices and E is the number of edges.

**Important Graph Algorithms**

**a. Dijkstra’s Algorithm (Shortest Path in Weighted Graphs)**

* Concept: Dijkstra’s algorithm finds the shortest path from a source node to all other nodes in a weighted graph with non-negative weights.
* Time Complexity: O(E + V log V) with a priority queue (min-heap).

**b. Bellman-Ford Algorithm**

* Concept: Finds the shortest path from a source node to all other nodes in a graph, even if the graph has negative edge weights. Can also detect negative weight cycles.
* Time Complexity: O(V \* E).

**c. Floyd-Warshall Algorithm (All-Pairs Shortest Path)**

* Concept: A dynamic programming algorithm used to find the shortest paths between all pairs of vertices.
* Time Complexity: O(V^3).

**d. Kruskal’s Algorithm (Minimum Spanning Tree)**

* Concept: Kruskal’s algorithm finds the minimum spanning tree (MST) of a graph by sorting the edges and adding them one by one while avoiding cycles.
* Time Complexity: O(E log E).

**e. Prim’s Algorithm (Minimum Spanning Tree)**

* Concept: Prim’s algorithm finds the minimum spanning tree (MST) by starting from any vertex and expanding the MST by adding the nearest vertex.
* Time Complexity: O(E + V log V).

**f. Topological Sorting (for DAGs)**

* Concept: Ordering the vertices of a Directed Acyclic Graph (DAG) in such a way that for every directed edge u -> v, vertex u comes before v.
* Applications: Task scheduling, build systems.
* Time Complexity: O(V + E).

**g. Cycle Detection in a Directed Graph (DFS)**

* Concept: Detects cycles using DFS by keeping track of the visited nodes and their recursion stack.
* Time Complexity: O(V + E).

**Problem Patterns Involving Graphs**

**1. Finding the Shortest Path**

* Problem: Given a graph, find the shortest path from a source node to a destination node.
* Approach: Use BFS for unweighted graphs, Dijkstra's Algorithm for weighted graphs.
* Time Complexity:
* BFS: O(V + E).
* Dijkstra: O(V log V + E) with a priority queue.

**2. Detecting a Cycle in a Graph**

* Problem: Check if a given directed graph contains a cycle.
* Approach: Use DFS with a recursion stack (for directed graphs), or Union-Find for undirected graphs.
* Time Complexity: O(V + E) for DFS, O(E α(V)) for Union-Find, where α is the inverse Ackermann function.

**3. Finding Strongly Connected Components (SCCs)**

* Problem: Given a directed graph, find all strongly connected components (SCCs) using Kosaraju's Algorithm or Tarjan’s Algorithm.
* Time Complexity: O(V + E).

**4. Minimum Spanning Tree (MST)**

* Problem: Find the minimum spanning tree of a weighted graph.
* Approach: Use Prim’s Algorithm or Kruskal’s Algorithm.
* Time Complexity: O(E log V) for Kruskal’s, O(V log V + E) for Prim’s.

**5. Shortest Path in a Weighted Graph**

* Problem: Given a weighted graph, find the shortest path from a source to all other nodes.
* Approach: Use Dijkstra’s Algorithm.
* Time Complexity: O(E + V log V).

**6. Bipartite Graph Check**

* Problem: Determine if a given graph is bipartite (can be colored with two colors such that no two adjacent vertices have the same color).
* Approach: Use BFS/DFS with two colors.
* Time Complexity: O(V + E).

**Advanced Graph Concepts and Algorithms**

**a. A Search Algorithm\***

* Concept: A\* is a pathfinding algorithm that combines Dijkstra’s Algorithm and heuristic estimation to find the shortest path more efficiently.
* Applications: Game development, robotics, navigation systems.
* Time Complexity: O(E) with a priority queue.

**b. Disjoint Set Union (Union-Find)**

* Concept: A data structure used for finding the connected components in a graph and performing efficient union operations.
* Applications: Cycle detection in undirected graphs, Kruskal’s MST algorithm.
* Time Complexity: O(α(V)) for both union and find operations, where α is the inverse Ackermann function.

**c. Edmonds-Karp Algorithm (Max Flow)**

* Concept: A specific implementation of the Ford-Fulkerson method for computing maximum flow in a flow network.
* Time Complexity: O(V \* E^2).

**Time Complexity Summary for Graph Algorithms**

* BFS: O(V + E).
* DFS: O(V + E).
* Dijkstra's Algorithm: O(V log V + E).
* Bellman-Ford: O(V \* E).
* Floyd-Warshall: O(V^3).
* Prim’s Algorithm: O(V log V + E).
* Kruskal’s Algorithm: O(E log E).
* Topological Sorting: O(V + E).
* Cycle Detection (DFS): O(V + E).

**Chapter 18: Trie Data Structure**

**Trie Data Structure - Concepts and Problem Patterns**

**Concept Overview: Trie**

* A Trie (also known as a Prefix Tree or Digital Tree) is a tree-like data structure used to store a dynamic set of strings, where the keys are usually strings. It is primarily used for efficient prefix matching, word search, and auto-completion tasks.
* Structure: Each node represents a character of a string, and each edge represents a link to the next character in the string.
* Root Node: The root node does not contain any character. It serves as the starting point for all strings.
* Leaf Node: A node where a complete string (or word) ends.
* Edge Labels: Represent the characters of the word.
* Prefix Property: Common prefixes of strings are stored only once in the trie, which helps in reducing space complexity.

Key Concepts:

* Insert: Insert a word by iterating through each character and adding nodes as necessary.
* Search: Search for a word or a prefix by following the nodes corresponding to each character in the string.
* Delete: Remove a word from the trie by deleting nodes as needed, while ensuring the trie structure remains valid.
* Prefix Matching: Efficiently find all words that start with a particular prefix.

**Trie Operations**

1. **Insertion**

* Concept: Insert a word into the trie by iterating through each character of the word, creating a new node for characters that do not exist in the trie.
* Time Complexity: O(m) where m is the length of the word being inserted.
* Insertion involves traversing each character of the word and adding corresponding nodes.

**b. Search**

* Concept: Search for a word or a prefix in the trie by following the edges corresponding to each character of the word.
* Word Search: Check if a complete word exists in the trie.
* Prefix Search: Check if there is any word that starts with the given prefix.
* Time Complexity: O(m) where m is the length of the word or prefix being searched.
* Searching involves traversing each character of the word or prefix in the trie.

**c. Deletion**

* Concept: Delete a word from the trie. Start from the root, follow the path of the word, and delete nodes when they are no longer required.
* Time Complexity: O(m) where m is the length of the word to be deleted.
* Deletion involves navigating through the nodes and ensuring that only unused nodes are deleted.

**d. Prefix Matching**

* Concept: Find all words that start with a given prefix. This can be done by finding the node corresponding to the prefix and then performing a DFS or BFS to retrieve all words that start with the prefix.
* Time Complexity: O(m + n) where m is the length of the prefix and n is the number of words with that prefix.
* The time complexity involves finding the prefix in the trie (O(m)) and then exploring all nodes under the prefix (O(n)).

**3. Advanced Topics in Trie**

**a. Trie with Wildcard Matching**

* Concept: Extend the basic trie to support wildcard matching (i.e., . can match any character). This is useful for pattern matching where some characters are unknown.
* Approach: During the search, recursively check all children nodes when encountering a . character, effectively exploring all possible matches.
* Time Complexity: O(m \* 4^m), where m is the length of the word or pattern (since there are 4 possible branches per node when using a wildcard).

**b. Trie with Frequency Count**

* Concept: A variation of the trie that stores the frequency of each word or prefix. This is useful for applications like auto-suggestions where we want to rank suggestions based on their frequency.
* Time Complexity: O(m) for inserting a word, O(m) for searching a word or prefix, and O(m) for deletion.

**c. Compressed Trie (Radix Tree)**

* Concept: A compressed trie (also known as a Radix Tree) is a variation where nodes with only one child are merged, making the structure more space-efficient.
* Time Complexity: Insertion, deletion, and search all operate in O(m) time where m is the length of the string.

**4. Problem Patterns in Trie**

**1. Insert a Word in Trie**

* Problem: Insert a word into a trie.
* Approach: Traverse through each character of the word, and for each character, create a new node if it doesn't exist.
* Time Complexity: O(m), where m is the length of the word.

**2. Search a Word in Trie**

* Problem: Check whether a word exists in the trie.
* Approach: Traverse the trie following the nodes corresponding to the characters of the word.
* Time Complexity: O(m), where m is the length of the word.

**3. Search Prefix in Trie**

* Problem: Check whether a given prefix exists in the trie.
* Approach: Traverse the trie following the nodes corresponding to the prefix.
* Time Complexity: O(m), where m is the length of the prefix.

**4. Count Words with Prefix**

* Problem: Count how many words in the trie start with a given prefix.
* Approach: Traverse to the node corresponding to the prefix and then perform a DFS or BFS to count all words.
* Time Complexity: O(m + n), where m is the length of the prefix and n is the number of words with that prefix.

**5. Implement Auto-Completion**

* Problem: Implement an auto-suggestion system using a trie where users type a prefix, and the system suggests words that start with the given prefix.
* Approach: Use the prefix matching operation and return all words that start with the prefix.
* Time Complexity: O(m + n), where m is the length of the prefix and n is the number of words with that prefix.

**6. Find Longest Prefix Match**

* Problem: Given a string, find the longest prefix in the trie.
* Approach: Traverse through the trie and keep track of the longest valid prefix.
* Time Complexity: O(m), where m is the length of the string.

**5. Time Complexity Summary for Trie Operations**

Operation Best Case Worst Case

Insertion O(m) O(m)

Search O(m) O(m)

Delete O(m) O(m)

Prefix Matching O(m + n) O(m + n)

Search with Wildcards O(m \* 4^m) O(m \* 4^m)

**6. Conclusion**

Tries are highly efficient data structures for applications that involve prefix matching, word search, and auto-completion. Mastering operations such as insertion, search, prefix matching, and wildcard matching in a trie is essential for Google DSA interviews. Additionally, advanced concepts like compressed tries and trie with frequency count can be leveraged to optimize space and improve performance for specific use cases like auto-suggestions and pattern matching. Understanding these concepts will help you solve a wide range of problems involving strings, dictionaries, and prefixes.

**Chapter 19: All Advanced Data structures**

**Advanced Data Structures**

**1. AVL Tree (Self-Balancing Binary Search Tree)**

Concept:

* An AVL tree (Adelson-Velsky and Landis tree) is a self-balancing binary search tree (BST).
* The tree ensures that the heights of the two child subtrees of any node differ by no more than one, which guarantees O(log n) time complexity for insertions, deletions, and lookups.
* Each node stores a balance factor (height(left subtree) - height(right subtree)), and if it becomes more than 1 or less than -1, rotations are performed to restore balance.
* Operations:
* Insertion: Insert the node as in a regular BST, then balance the tree using rotations.
* Deletion: Remove the node and rebalance the tree.
* Search: Standard binary search.

Time Complexity:

* Insertion, Deletion, Search: O(log n)

Problem Patterns:

* Balanced BST Operations: Ensuring efficient search, insert, and delete in scenarios requiring dynamic set operations (e.g., implementing priority queues).
* Interval Management: In problems like managing overlapping intervals or ranges, AVL trees provide efficient solutions.

**2. B-Tree**

Concept:

* A B-tree is a self-balancing search tree in which nodes can have more than two children.
* It is optimized for systems that read and write large blocks of data (e.g., databases).
* B-trees maintain sorted data and allow for efficient search, insert, delete, and range queries.
* B+ Tree is a variant where all data is stored at the leaf level, making range queries more efficient.

Operations:

* Insertion: Insert elements into the appropriate leaf node and split nodes if necessary.
* Deletion: Remove an element and merge or redistribute nodes to maintain balance.
* Search: Perform a binary search within the node and recurse into the appropriate child node.

Time Complexity:

* Insertion, Deletion, Search: O(log n)

Problem Patterns:

* Database Indexing: Used in databases to index large amounts of data and enable quick lookups.
* File System Indexing: Used in filesystems to manage data on disk.

**3. B+ Tree**

Concept:

* The B+ Tree is a self-balancing tree data structure with an ordered set of keys, used in databases and file systems.
* Unlike a B-tree, the B+ tree stores all data at the leaf level, and each leaf node points to the next leaf node, making it more efficient for range queries and sequential access.

Operations:

* Insertion: Insert elements like in a B-tree but ensure all data is stored at the leaf level.
* Deletion: Remove an element and re-balance the tree as necessary.
* Search: Perform binary search in internal nodes to find the appropriate leaf.

Time Complexity:

* Insertion, Deletion, Search: O(log n)

Problem Patterns:

* Efficient Range Queries: Range queries like finding the "next" or "previous" elements are faster due to linked leaf nodes.
* Database Querying: Optimal for disk-based databases for efficient retrieval of large datasets.

**4. Disjoint Set Union (Union-Find)**

Concept:

* The Union-Find or Disjoint Set Union (DSU) data structure tracks a collection of disjoint sets.
* It supports two primary operations:
* Find: Determine which set a particular element belongs to.
* Union: Merge two sets into one.
* This data structure uses path compression (to flatten the structure) and union by rank/size (to keep the tree balanced).

Operations:

* Union: Merge two sets by connecting the root nodes of the sets.
* Find: Use path compression to make future queries faster.

Time Complexity:

* Union, Find: O(α(n)) where α is the inverse Ackermann function, which grows extremely slowly (almost constant time).

Problem Patterns:

* Cycle Detection: Detect cycles in undirected graphs (e.g., finding cycles in a graph).
* Connected Components: Find connected components in a graph or determine if two nodes are connected.

**5. Fenwick Tree (Binary Indexed Tree)**

Concept:

* The Fenwick Tree is a data structure that efficiently supports range sum queries and point updates.
* It is used to store prefix sums of a sequence of numbers, allowing for efficient querying and updating of sums.
* The tree is represented as an array, and each node contains the sum of a range of elements.

Operations:

* Update: Increment the value at a specific index.
* Query: Compute the sum of the elements from the start up to a specific index.

Time Complexity:

* Update, Query: O(log n)

Problem Patterns:

* Range Queries: Efficiently compute prefix sums, or range sum queries.
* Dynamic Programming: Update and query for partial sums or counts efficiently.

**6. Segment Tree**

Concept:

* A Segment Tree is a binary tree used for storing intervals or segments.
* It allows querying over intervals efficiently (e.g., range sum, minimum, maximum).
* Segment trees are ideal for situations where you need to repeatedly query and update segments of an array.

Operations:

* Build: Construct the tree in O(n) time.
* Update: Modify a single element, and propagate the changes up the tree.
* Query: Query the segment tree for range sums or other aggregate values.

Time Complexity:

* Build: O(n)
* Update: O(log n)
* Query: O(log n)

Problem Patterns:

* Range Queries: Efficiently perform queries like sum, min, max, greatest common divisor, etc.
* Interval Updates: Update all values in a given range.

**7. Skip List**

Concept:

* A Skip List is a probabilistic data structure that allows for efficient search, insertion, and deletion in O(log n) time.
* It consists of multiple layers of linked lists, where each higher layer skips over more elements than the previous one, effectively reducing the search space.

Operations:

* Insert: Insert elements into the skip list by determining which levels to add the element to.
* Search: Traverse down the layers of the skip list to find the desired element.
* Delete: Remove an element by adjusting the pointers in each level.

Time Complexity:

* Insert, Search, Delete: O(log n) (on average)

Problem Patterns:

* Sorted Set Operations: Efficient handling of sorted collections with logarithmic time complexity for insertion, deletion, and search.

**8. Treap (Randomized Binary Search Tree)**

Concept:

* A Treap combines the properties of a binary search tree (BST) and a heap.
* The BST property ensures that for every node, the value of the left child is less than the parent, and the value of the right child is greater.
* The heap property ensures that the priority of each node (assigned randomly) is greater than that of its children, which maintains a balanced tree.

Operations:

* Insertion: Insert the element while maintaining both the heap and BST properties.
* Deletion: Remove the node and re-balance the tree using rotations.
* Search: Perform a standard binary search.

Time Complexity:

* Insertion, Deletion, Search: O(log n) (on average)

Problem Patterns:

* Efficient Search: Useful for dynamic search operations that need to maintain balance with minimal cost.

**9. Suffix Tree**

Concept:

* A Suffix Tree is a compressed trie of all the suffixes of a string. It is used for various string processing applications, such as substring search, pattern matching, and string compression.
* Suffix trees allow for efficient string operations such as substring search in O(m) time, where m is the length of the string.

Operations:

* Build: Construct the tree by inserting all suffixes of the string.
* Search: Find a substring or pattern in the tree.

Time Complexity:

* Build: O(n)
* Search: O(m), where m is the length of the pattern.

Problem Patterns:

* String Matching: Efficient substring matching and pattern search.
* Longest Repeated Substring: Find the longest repeated substring in a string.