Complete Data Structures & Algorithms Guide - Beginner to Expert

Table of Contents

Java Basics for DSA

Essential Java Concepts

```java
```

```java
// Basic class structure
public class Main {
    public static void main(String[] args) {
        // Your code here
    }
}

// ArrayList (Dynamic Array)
import java.util.*;
List<Integer> list = new ArrayList<>();
list.add(1);  // Add element
list.get(0);  // Get element at index 0
list.size(); // Get size

// Arrays
int[] arr = new int[5];  // Fixed size array
int[] arr2 = {1, 2, 3, 4, 5};  // Initialize with values

// HashMap
Map<String, Integer> map = new HashMap<>();
map.put("key", 1);
map.get("key");
map.containsKey("key");

// HashSet
Set<Integer> set = new HashSet<>();
set.add(1);
set.contains(1);
```

Arrays

What is an Array?

An array is a collection of elements stored in contiguous memory locations. Each element can be accessed using an index.

Implementation and Operations

java

```java
public class ArrayOperations {
    private int[] arr;
    private int size;
    private int capacity;

    public ArrayOperations(int capacity) {
        this.capacity = capacity;
        this.arr = new int[capacity];
        this.size = 0;
    }

    // Insert at end - O(1)
    public void insert(int value) {
        if (size < capacity) {
            arr[size] = value;
            size++;
        }
    }

    // Insert at index - O(n)
    public void insertAt(int index, int value) {
        if (index >= 0 && index <= size && size < capacity) {
            for (int i = size; i > index; i--) {
                arr[i] = arr[i - 1];
            }
            arr[index] = value;
            size++;
        }
    }

    // Delete at index - O(n)
    public void deleteAt(int index) {
        if (index >= 0 && index < size) {
            for (int i = index; i < size - 1; i++) {
                arr[i] = arr[i + 1];
            }
            size--;
        }
    }

    // Search - O(n)
    public int search(int value) {
        for (int i = 0; i < size; i++) {
            if (arr[i] == value) {
                return i;
            }
```

```
        }
        return -1;
    }
}
```

Time & Space Complexity

- **Access**: O(1)
- **Search**: O(n)
- **Insert**: O(1) at end, O(n) at beginning/middle
- **Delete**: O(1) at end, O(n) at beginning/middle
- **Space**: O(n)

Blind 75 Problem: Two Sum

Problem: Given an array of integers and a target sum, return indices of two numbers that add up to target.

```java
```

```java
public class TwoSum {
    // Approach 1: Brute Force - O(n²) time, O(1) space
    public int[] twoSumBruteForce(int[] nums, int target) {
        for (int i = 0; i < nums.length; i++) {
            for (int j = i + 1; j < nums.length; j++) {
                if (nums[i] + nums[j] == target) {
                    return new int[]{i, j};
                }
            }
        }
        return new int[]{};
    }

    // Approach 2: Hash Map - O(n) time, O(n) space
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[]{map.get(complement), i};
            }
            map.put(nums[i], i);
        }
        return new int[]{};
    }
}
```

Explanation: We use a HashMap to store numbers we've seen and their indices. For each number, we check if its complement (target - current number) exists in the map.

Strings

String Operations in Java

```java
```

```java
public class StringOperations {

    // String comparison - O(n)
    public boolean areEqual(String s1, String s2) {
        return s1.equals(s2);
    }

    // Palindrome check - O(n)
    public boolean isPalindrome(String s) {
        int left = 0, right = s.length() - 1;
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }

    // Reverse string - O(n)
    public String reverse(String s) {
        StringBuilder sb = new StringBuilder();
        for (int i = s.length() - 1; i >= 0; i--) {
            sb.append(s.charAt(i));
        }
        return sb.toString();
    }
}
```

Blind 75 Problem: Valid Anagram

Problem: Given two strings, determine if they are anagrams of each other.

```java
```

```java
public class ValidAnagram {
    // Approach 1: Sorting - O(n log n) time, O(1) space
    public boolean isAnagramSort(String s, String t) {
        if (s.length() != t.length()) return false;

        char[] sArray = s.toCharArray();
        char[] tArray = t.toCharArray();
        Arrays.sort(sArray);
        Arrays.sort(tArray);

        return Arrays.equals(sArray, tArray);
    }

    // Approach 2: Character Count - O(n) time, O(1) space
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) return false;

        int[] count = new int[26]; // for lowercase letters

        for (int i = 0; i < s.length(); i++) {
            count[s.charAt(i) - 'a']++;
            count[t.charAt(i) - 'a']--;
        }

        for (int c : count) {
            if (c != 0) return false;
        }

        return true;
    }
}
```

Linked Lists

What is a Linked List?

A linked list is a linear data structure where elements are stored in nodes, and each node contains data and a reference to the next node.

Implementation

```java
```

```java
public class ListNode {
    int val;
    ListNode next;

    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

public class LinkedList {
    private ListNode head;

    // Insert at beginning - O(1)
    public void insertFirst(int val) {
        ListNode newNode = new ListNode(val);
        newNode.next = head;
        head = newNode;
    }

    // Insert at end - O(n)
    public void insertLast(int val) {
        ListNode newNode = new ListNode(val);
        if (head == null) {
            head = newNode;
            return;
        }

        ListNode current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }

    // Delete node - O(n)
    public void delete(int val) {
        if (head == null) return;

        if (head.val == val) {
            head = head.next;
            return;
        }

        ListNode current = head;
        while (current.next != null && current.next.val != val) {
            current = current.next;
```

```
        }

        if (current.next != null) {
            current.next = current.next.next;
        }
    }

    // Search - O(n)
    public boolean search(int val) {
        ListNode current = head;
        while (current != null) {
            if (current.val == val) return true;
            current = current.next;
        }
        return false;
    }
}
```

Time & Space Complexity

- **Access**: O(n)
- **Search**: O(n)
- **Insert**: O(1) at beginning, O(n) at end
- **Delete**: O(1) if node given, O(n) if value given
- **Space**: O(n)

Blind 75 Problem: Reverse Linked List

Problem: Reverse a singly linked list.

java

```java
public class ReverseLinkedList {
    // Iterative approach - O(n) time, O(1) space
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;

        while (current != null) {
            ListNode nextNode = current.next;
            current.next = prev;
            prev = current;
            current = nextNode;
        }

        return prev;
    }

    // Recursive approach - O(n) time, O(n) space
    public ListNode reverseListRecursive(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }

        ListNode newHead = reverseListRecursive(head.next);
        head.next.next = head;
        head.next = null;

        return newHead;
    }
}
```

Explanation: We maintain three pointers - prev, current, and next. We reverse the link between current and prev, then move all pointers forward.

Stacks

What is a Stack?

A stack is a LIFO (Last In, First Out) data structure where elements are added and removed from the same end called the top.

Implementation

java

```java
public class Stack {
    private int[] arr;
    private int top;
    private int capacity;

    public Stack(int capacity) {
        this.capacity = capacity;
        this.arr = new int[capacity];
        this.top = -1;
    }

    // Push - O(1)
    public void push(int val) {
        if (top < capacity - 1) {
            arr[++top] = val;
        }
    }

    // Pop - O(1)
    public int pop() {
        if (top >= 0) {
            return arr[top--];
        }
        throw new RuntimeException("Stack is empty");
    }

    // Peek - O(1)
    public int peek() {
        if (top >= 0) {
            return arr[top];
        }
        throw new RuntimeException("Stack is empty");
    }

    // isEmpty - O(1)
    public boolean isEmpty() {
        return top == -1;
    }
}
```

Time & Space Complexity

- **Push**: O(1)
- **Pop**: O(1)
- **Peek**: O(1)

- **Space**: O(n)

Blind 75 Problem: Valid Parentheses

Problem: Given a string containing just parentheses, determine if the input string is valid.

```java
public class ValidParentheses {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();

        for (char c : s.toCharArray()) {
            // Push opening brackets
            if (c == '(' || c == '[' || c == '{') {
                stack.push(c);
            }
            // Check closing brackets
            else {
                if (stack.isEmpty()) return false;

                char top = stack.pop();
                if ((c == ')' && top != '(') ||
                    (c == ']' && top != '[') ||
                    (c == '}' && top != '{')) {
                    return false;
                }
            }
        }

        return stack.isEmpty();
    }
}
```

Explanation: We use a stack to keep track of opening brackets. When we see a closing bracket, we check if it matches the most recent opening bracket.

Trees

What is a Tree?

A tree is a hierarchical data structure with nodes connected by edges. Each node has at most one parent and zero or more children.

Binary Tree Implementation

```java
```

```java
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

public class BinaryTree {
    private TreeNode root;

    // Inorder Traversal (Left, Root, Right) - O(n)
    public void inorder(TreeNode node) {
        if (node != null) {
            inorder(node.left);
            System.out.print(node.val + " ");
            inorder(node.right);
        }
    }

    // Preorder Traversal (Root, Left, Right) - O(n)
    public void preorder(TreeNode node) {
        if (node != null) {
            System.out.print(node.val + " ");
            preorder(node.left);
            preorder(node.right);
        }
    }

    // Postorder Traversal (Left, Right, Root) - O(n)
    public void postorder(TreeNode node) {
        if (node != null) {
            postorder(node.left);
            postorder(node.right);
            System.out.print(node.val + " ");
        }
    }

    // Level Order Traversal - O(n)
    public List<List<Integer>> levelOrder(TreeNode root) {
```

```
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) return result;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> level = new ArrayList<>();

            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                level.add(node.val);

                if (node.left != null) queue.offer(node.left);
                if (node.right != null) queue.offer(node.right);
            }
            result.add(level);
        }
        return result;
    }
}
```

Blind 75 Problem: Maximum Depth of Binary Tree

Problem: Find the maximum depth of a binary tree.

java

```java
public class MaxDepth {
    // Recursive approach - O(n) time, O(h) space where h is height
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;

        int leftDepth = maxDepth(root.left);
        int rightDepth = maxDepth(root.right);

        return Math.max(leftDepth, rightDepth) + 1;
    }

    // Iterative approach using level order traversal
    public int maxDepthIterative(TreeNode root) {
        if (root == null) return 0;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int depth = 0;

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            depth++;

            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                if (node.left != null) queue.offer(node.left);
                if (node.right != null) queue.offer(node.right);
            }
        }
        return depth;
    }
}
```

Dynamic Programming

What is Dynamic Programming?

Dynamic Programming is an optimization technique that solves complex problems by breaking them down into simpler subproblems and storing the results to avoid redundant calculations.

Types of DP:

1. **Memoization (Top-Down)**: Recursive approach with caching
2. **Tabulation (Bottom-Up)**: Iterative approach building solutions from smaller problems

Blind 75 Problem: Climbing Stairs

Problem: You're climbing stairs. It takes n steps to reach the top. Each time you can climb 1 or 2 steps. How many distinct ways can you climb to the top?

java

```java
public class ClimbingStairs {
    // Approach 1: Recursive (Exponential time) - Don't use this!
    public int climbStairsRecursive(int n) {
        if (n <= 1) return 1;
        return climbStairsRecursive(n - 1) + climbStairsRecursive(n - 2);
    }

    // Approach 2: Memoization - O(n) time, O(n) space
    public int climbStairs(int n) {
        return climbStairsHelper(n, new int[n + 1]);
    }

    private int climbStairsHelper(int n, int[] memo) {
        if (n <= 1) return 1;
        if (memo[n] != 0) return memo[n];

        memo[n] = climbStairsHelper(n - 1, memo) + climbStairsHelper(n - 2, memo);
        return memo[n];
    }

    // Approach 3: Bottom-up DP - O(n) time, O(n) space
    public int climbStairsDP(int n) {
        if (n <= 1) return 1;

        int[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = 1;

        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }

    // Approach 4: Space Optimized - O(n) time, O(1) space
    public int climbStairsOptimal(int n) {
        if (n <= 1) return 1;

        int prev2 = 1, prev1 = 1;

        for (int i = 2; i <= n; i++) {
            int current = prev1 + prev2;
            prev2 = prev1;
            prev1 = current;
        }
```

```
        return prev1;
    }
}
```

Explanation: This is essentially the Fibonacci sequence. Each step depends on the sum of ways to reach the previous two steps.

Blind 75 Problem: House Robber

Problem: You're a robber. Houses have money, but you can't rob adjacent houses. What's the maximum amount you can rob?

java

```java
public class HouseRobber {
    // Bottom-up DP - O(n) time, O(n) space
    public int rob(int[] nums) {
        if (nums.length == 0) return 0;
        if (nums.length == 1) return nums[0];

        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0], nums[1]);

        for (int i = 2; i < nums.length; i++) {
            dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
        }

        return dp[nums.length - 1];
    }

    // Space optimized - O(n) time, O(1) space
    public int robOptimal(int[] nums) {
        if (nums.length == 0) return 0;
        if (nums.length == 1) return nums[0];

        int prev2 = nums[0];
        int prev1 = Math.max(nums[0], nums[1]);

        for (int i = 2; i < nums.length; i++) {
            int current = Math.max(prev1, prev2 + nums[i]);
            prev2 = prev1;
            prev1 = current;
        }

        return prev1;
    }
}
```

Searching Algorithms

Binary Search

Works on sorted arrays by repeatedly dividing search space in half.

```java
```

```java
public class BinarySearch {
    // Iterative - O(log n) time, O(1) space
    public int binarySearch(int[] nums, int target) {
        int left = 0, right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return -1;
    }

    // Recursive - O(log n) time, O(log n) space
    public int binarySearchRecursive(int[] nums, int target) {
        return binarySearchHelper(nums, target, 0, nums.length - 1);
    }

    private int binarySearchHelper(int[] nums, int target, int left, int right) {
        if (left > right) return -1;

        int mid = left + (right - left) / 2;

        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            return binarySearchHelper(nums, target, mid + 1, right);
        } else {
            return binarySearchHelper(nums, target, left, mid - 1);
        }
    }
}
```

Blind 75 Problem: Search in Rotated Sorted Array

Problem: Search for a target value in a rotated sorted array.

```java
```

```java
public class SearchRotatedArray {
    public int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            }

            // Left half is sorted
            if (nums[left] <= nums[mid]) {
                if (target >= nums[left] && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
            // Right half is sorted
            else {
                if (target > nums[mid] && target <= nums[right]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

        return -1;
    }
}
```

Two Pointers Technique

Concept

Use two pointers moving towards each other or in the same direction to solve problems efficiently.

Blind 75 Problem: Container With Most Water

Problem: Given heights array, find two lines that form a container holding the most water.

```java
```

```java
public class ContainerWithMostWater {
    // Brute Force - O(n²) time
    public int maxAreaBruteForce(int[] height) {
        int maxArea = 0;
        for (int i = 0; i < height.length; i++) {
            for (int j = i + 1; j < height.length; j++) {
                int area = Math.min(height[i], height[j]) * (j - i);
                maxArea = Math.max(maxArea, area);
            }
        }
        return maxArea;
    }

    // Two Pointers - O(n) time, O(1) space
    public int maxArea(int[] height) {
        int left = 0, right = height.length - 1;
        int maxArea = 0;

        while (left < right) {
            int area = Math.min(height[left], height[right]) * (right - left);
            maxArea = Math.max(maxArea, area);

            // Move pointer with smaller height
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return maxArea;
    }
}
```

Explanation: We start with the widest container and move the pointer with the smaller height inward, as moving the taller one can't possibly give us a larger area.

Sliding Window Technique

Concept

Maintain a window of elements and slide it to find the optimal solution.

Blind 75 Problem: Best Time to Buy and Sell Stock

Problem: Find the maximum profit from buying and selling stock once.

java

```java
public class BestTimeToBuyStock {
    // Brute Force - O(n²) time
    public int maxProfitBruteForce(int[] prices) {
        int maxProfit = 0;
        for (int i = 0; i < prices.length; i++) {
            for (int j = i + 1; j < prices.length; j++) {
                int profit = prices[j] - prices[i];
                maxProfit = Math.max(maxProfit, profit);
            }
        }
        return maxProfit;
    }

    // Sliding Window - O(n) time, O(1) space
    public int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE;
        int maxProfit = 0;

        for (int price : prices) {
            if (price < minPrice) {
                minPrice = price;
            } else {
                maxProfit = Math.max(maxProfit, price - minPrice);
            }
        }

        return maxProfit;
    }
}
```

Hash Tables

Implementation

java

```
public class HashTable {
    private static class Entry {
        String key;
        Integer value;
        Entry next;

        Entry(String key, Integer value) {
            this.key = key;
            this.value = value;
        }
    }

    private Entry[] buckets;
    private int capacity;
    private int size;

    public HashTable(int capacity) {
        this.capacity = capacity;
        this.buckets = new Entry[capacity];
        this.size = 0;
    }

    private int hash(String key) {
        return Math.abs(key.hashCode()) % capacity;
    }

    // Put - O(1) average, O(n) worst case
    public void put(String key, Integer value) {
        int index = hash(key);
        Entry head = buckets[index];

        while (head != null) {
            if (head.key.equals(key)) {
                head.value = value;
                return;
            }
            head = head.next;
        }

        Entry newEntry = new Entry(key, value);
        newEntry.next = buckets[index];
        buckets[index] = newEntry;
        size++;
    }

    // Get - O(1) average, O(n) worst case
```

```
    public Integer get(String key) {
        int index = hash(key);
        Entry head = buckets[index];

        while (head != null) {
            if (head.key.equals(key)) {
                return head.value;
            }
            head = head.next;
        }
        return null;
    }
}
```

Complete Practice Problems

Easy Level Problems

1. Contains Duplicate

java

```
public boolean containsDuplicate(int[] nums) {
    Set<Integer> seen = new HashSet<>();
    for (int num : nums) {
        if (seen.contains(num)) {
            return true;
        }
        seen.add(num);
    }
    return false;
}
```

2. Maximum Subarray (Kadane's Algorithm)

java

```java
public int maxSubArray(int[] nums) {
    int maxSoFar = nums[0];
    int maxEndingHere = nums[0];

    for (int i = 1; i < nums.length; i++) {
        maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
}
```

Medium Level Problems

3. Product of Array Except Self

java

```java
public int[] productExceptSelf(int[] nums) {
    int n = nums.length;
    int[] result = new int[n];

    // Left products
    result[0] = 1;
    for (int i = 1; i < n; i++) {
        result[i] = result[i - 1] * nums[i - 1];
    }

    // Right products
    int rightProduct = 1;
    for (int i = n - 1; i >= 0; i--) {
        result[i] *= rightProduct;
        rightProduct *= nums[i];
    }

    return result;
}
```

4. 3Sum

java

```java
public List<List<Integer>> threeSum(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    Arrays.sort(nums);

    for (int i = 0; i < nums.length - 2; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;

        int left = i + 1, right = nums.length - 1;

        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];

            if (sum == 0) {
                result.add(Arrays.asList(nums[i], nums[left], nums[right]));

                while (left < right && nums[left] == nums[left + 1]) left++;
                while (left < right && nums[right] == nums[right - 1]) right--;

                left++;
                right--;
            } else if (sum < 0) {
                left++;
            } else {
                right--;
            }
        }
    }

    return result;
}
```

Hard Level Problems

5. Merge k Sorted Lists

```java
```

```java
public ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0) return null;

    PriorityQueue<ListNode> pq = new PriorityQueue<>((a, b) -> a.val - b.val);

    // Add all non-null list heads to priority queue
    for (ListNode list : lists) {
        if (list != null) {
            pq.offer(list);
        }
    }

    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

    while (!pq.isEmpty()) {
        ListNode node = pq.poll();
        current.next = node;
        current = current.next;

        if (node.next != null) {
            pq.offer(node.next);
        }
    }

    return dummy.next;
}
```

Graphs

What is a Graph?

A graph is a collection of nodes (vertices) connected by edges. Graphs can be directed or undirected, weighted or unweighted.

Graph Representations

1. Adjacency List

```java
```

```java
public class GraphAdjList {
    private Map<Integer, List<Integer>> adjList;

    public GraphAdjList() {
        adjList = new HashMap<>();
    }

    public void addVertex(int vertex) {
        adjList.putIfAbsent(vertex, new ArrayList<>());
    }

    public void addEdge(int source, int dest) {
        adjList.get(source).add(dest);
        adjList.get(dest).add(source); // For undirected graph
    }

    // DFS - O(V + E) time, O(V) space
    public void dfs(int start) {
        Set<Integer> visited = new HashSet<>();
        dfsHelper(start, visited);
    }

    private void dfsHelper(int vertex, Set<Integer> visited) {
        visited.add(vertex);
        System.out.print(vertex + " ");

        for (int neighbor : adjList.get(vertex)) {
            if (!visited.contains(neighbor)) {
                dfsHelper(neighbor, visited);
            }
        }
    }

    // BFS - O(V + E) time, O(V) space
    public void bfs(int start) {
        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();

        visited.add(start);
        queue.offer(start);

        while (!queue.isEmpty()) {
            int vertex = queue.poll();
            System.out.print(vertex + " ");

            for (int neighbor : adjList.get(vertex)) {
```

```
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.offer(neighbor);
            }
        }
    }
}
```

2. Adjacency Matrix

```java
public class GraphAdjMatrix {
    private int[][] matrix;
    private int numVertices;

    public GraphAdjMatrix(int numVertices) {
        this.numVertices = numVertices;
        matrix = new int[numVertices][numVertices];
    }

    public void addEdge(int source, int dest) {
        matrix[source][dest] = 1;
        matrix[dest][source] = 1; // For undirected graph
    }

    public boolean hasEdge(int source, int dest) {
        return matrix[source][dest] == 1;
    }
}
```

Blind 75 Problem: Number of Islands

Problem: Given a 2D grid of '1's (land) and '0's (water), count the number of islands.

```java
```

```java
public class NumberOfIslands {
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;

        int numIslands = 0;
        int rows = grid.length;
        int cols = grid[0].length;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] == '1') {
                    numIslands++;
                    dfs(grid, i, j);
                }
            }
        }

        return numIslands;
    }

    private void dfs(char[][] grid, int i, int j) {
        int rows = grid.length;
        int cols = grid[0].length;

        // Boundary check and water check
        if (i < 0 || i >= rows || j < 0 || j >= cols || grid[i][j] == '0') {
            return;
        }

        // Mark as visited
        grid[i][j] = '0';

        // Explore all 4 directions
        dfs(grid, i + 1, j);
        dfs(grid, i - 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i, j - 1);
    }

    // BFS approach
    public int numIslandsBFS(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;

        int numIslands = 0;
        int rows = grid.length;
        int cols = grid[0].length;
```

```
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] == '1') {
                    numIslands++;
                    bfs(grid, i, j);
                }
            }
        }

        return numIslands;
    }

    private void bfs(char[][] grid, int startI, int startJ) {
        Queue<int[]> queue = new LinkedList<>();
        queue.offer(new int[]{startI, startJ});
        grid[startI][startJ] = '0';

        int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

        while (!queue.isEmpty()) {
            int[] current = queue.poll();
            int i = current[0], j = current[1];

            for (int[] dir : directions) {
                int newI = i + dir[0];
                int newJ = j + dir[1];

                if (newI >= 0 && newI < grid.length &&
                    newJ >= 0 && newJ < grid[0].length &&
                    grid[newI][newJ] == '1') {

                    grid[newI][newJ] = '0';
                    queue.offer(new int[]{newI, newJ});
                }
            }
        }
    }
}
```

Explanation: We iterate through the grid and when we find a '1', we increment island count and use DFS/BFS to mark all connected land cells as visited.

Blind 75 Problem: Clone Graph

Problem: Deep clone an undirected graph.

java

```java
class Node {
    public int val;
    public List<Node> neighbors;

    public Node() {
        val = 0;
        neighbors = new ArrayList<Node>();
    }

    public Node(int _val) {
        val = _val;
        neighbors = new ArrayList<Node>();
    }

    public Node(int _val, ArrayList<Node> _neighbors) {
        val = _val;
        neighbors = _neighbors;
    }
}

public class CloneGraph {
    // DFS approach
    public Node cloneGraph(Node node) {
        if (node == null) return null;

        Map<Node, Node> visited = new HashMap<>();
        return dfs(node, visited);
    }

    private Node dfs(Node node, Map<Node, Node> visited) {
        if (visited.containsKey(node)) {
            return visited.get(node);
        }

        Node clone = new Node(node.val);
        visited.put(node, clone);

        for (Node neighbor : node.neighbors) {
            clone.neighbors.add(dfs(neighbor, visited));
        }

        return clone;
    }

    // BFS approach
    public Node cloneGraphBFS(Node node) {
```

```java
        if (node == null) return null;

        Map<Node, Node> visited = new HashMap<>();
        Queue<Node> queue = new LinkedList<>();

        visited.put(node, new Node(node.val));
        queue.offer(node);

        while (!queue.isEmpty()) {
            Node current = queue.poll();

            for (Node neighbor : current.neighbors) {
                if (!visited.containsKey(neighbor)) {
                    visited.put(neighbor, new Node(neighbor.val));
                    queue.offer(neighbor);
                }
                visited.get(current).neighbors.add(visited.get(neighbor));
            }
        }

        return visited.get(node);
    }
}
```

Sorting Algorithms

1. Bubble Sort

Time: O(n^2), **Space**: O(1)

```java
public void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

2. Selection Sort

Time: O(n^2), **Space**: O(1)

```java
public void selectionSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        // Swap
        int temp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = temp;
    }
}
```

3. Insertion Sort

Time: O(n^2), **Space**: O(1)

```java
public void insertionSort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

4. Merge Sort

Time: O(n log n), **Space**: O(n)

```java
```

```
public void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

private void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] L = new int[n1];
    int[] R = new int[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
```

```
        j++;
        k++;
    }
}
```

5. Quick Sort

Time: O(n log n) average, O(n^2) worst, **Space**: O(log n)

```java
public void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

private int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}
```

Advanced Topics

Backtracking

Concept: Try all possibilities by exploring paths and backtracking when we reach a dead end.

Blind 75 Problem: Combination Sum

Problem: Find all unique combinations where candidate numbers sum to target.

java

```java
public class CombinationSum {
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(candidates, target, 0, new ArrayList<>(), result);
        return result;
    }

    private void backtrack(int[] candidates, int target, int start,
                           List<Integer> current, List<List<Integer>> result) {
        if (target == 0) {
            result.add(new ArrayList<>(current));
            return;
        }

        if (target < 0) return;

        for (int i = start; i < candidates.length; i++) {
            current.add(candidates[i]);
            backtrack(candidates, target - candidates[i], i, current, result);
            current.remove(current.size() - 1); // Backtrack
        }
    }
}
```

Word Search

Problem: Check if a word exists in a 2D board.

java

```java
public class WordSearch {
    public boolean exist(char[][] board, String word) {
        int rows = board.length;
        int cols = board[0].length;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (dfs(board, word, i, j, 0)) {
                    return true;
                }
            }
        }
        return false;
    }

    private boolean dfs(char[][] board, String word, int i, int j, int index) {
        if (index == word.length()) return true;

        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length ||
            board[i][j] != word.charAt(index)) {
            return false;
        }

        char temp = board[i][j];
        board[i][j] = '#'; // Mark as visited

        boolean found = dfs(board, word, i + 1, j, index + 1) ||
                        dfs(board, word, i - 1, j, index + 1) ||
                        dfs(board, word, i, j + 1, index + 1) ||
                        dfs(board, word, i, j - 1, index + 1);

        board[i][j] = temp; // Backtrack
        return found;
    }
}
```

Heap / Priority Queue

Implementation

java

```java
public class MinHeap {
    private int[] heap;
    private int size;
    private int capacity;

    public MinHeap(int capacity) {
        this.capacity = capacity;
        this.heap = new int[capacity];
        this.size = 0;
    }

    private int parent(int i) { return (i - 1) / 2; }
    private int leftChild(int i) { return 2 * i + 1; }
    private int rightChild(int i) { return 2 * i + 2; }

    // Insert - O(log n)
    public void insert(int value) {
        if (size >= capacity) return;

        heap[size] = value;
        int current = size;
        size++;

        // Heapify up
        while (current != 0 && heap[current] < heap[parent(current)]) {
            swap(current, parent(current));
            current = parent(current);
        }
    }

    // Extract Min - O(log n)
    public int extractMin() {
        if (size <= 0) return Integer.MAX_VALUE;
        if (size == 1) {
            size--;
            return heap[0];
        }

        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;
        heapifyDown(0);

        return root;
    }
```

```
    private void heapifyDown(int i) {
        int left = leftChild(i);
        int right = rightChild(i);
        int smallest = i;

        if (left < size && heap[left] < heap[smallest]) {
            smallest = left;
        }

        if (right < size && heap[right] < heap[smallest]) {
            smallest = right;
        }

        if (smallest != i) {
            swap(i, smallest);
            heapifyDown(smallest);
        }
    }

    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }
}
```

Blind 75 Problem: Kth Largest Element

Problem: Find the kth largest element in an unsorted array.

java

```java
public class KthLargest {
    // Using Min Heap - O(n log k) time, O(k) space
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.offer(num);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }

        return minHeap.peek();
    }

    // Using QuickSelect - O(n) average, O(n²) worst
    public int findKthLargestQuickSelect(int[] nums, int k) {
        return quickSelect(nums, 0, nums.length - 1, nums.length - k);
    }

    private int quickSelect(int[] nums, int left, int right, int kSmallest) {
        if (left == right) return nums[left];

        Random rand = new Random();
        int pivotIndex = left + rand.nextInt(right - left + 1);
        pivotIndex = partition(nums, left, right, pivotIndex);

        if (kSmallest == pivotIndex) {
            return nums[kSmallest];
        } else if (kSmallest < pivotIndex) {
            return quickSelect(nums, left, pivotIndex - 1, kSmallest);
        } else {
            return quickSelect(nums, pivotIndex + 1, right, kSmallest);
        }
    }

    private int partition(int[] nums, int left, int right, int pivotIndex) {
        int pivotValue = nums[pivotIndex];
        swap(nums, pivotIndex, right);

        int storeIndex = left;
        for (int i = left; i < right; i++) {
            if (nums[i] < pivotValue) {
                swap(nums, storeIndex, i);
                storeIndex++;
            }
```

```
        }
        swap(nums, right, storeIndex);
        return storeIndex;
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

Trie (Prefix Tree)

Implementation

java

```java
public class Trie {
    private TrieNode root;

    private class TrieNode {
        TrieNode[] children;
        boolean isEndOfWord;

        public TrieNode() {
            children = new TrieNode[26]; // for lowercase letters
            isEndOfWord = false;
        }
    }

    public Trie() {
        root = new TrieNode();
    }

    // Insert - O(m) where m is length of word
    public void insert(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
        }
        current.isEndOfWord = true;
    }

    // Search - O(m)
    public boolean search(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return false;
            }
            current = current.children[index];
        }
        return current.isEndOfWord;
    }

    // Starts With - O(m)
    public boolean startsWith(String prefix) {
        TrieNode current = root;
```

```
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return false;
            }
            current = current.children[index];
        }
        return true;
    }
}
```

Key Tips for Problem Solving

1. Problem Analysis Framework

1. **Understand the problem** - Read carefully, ask questions
2. **Identify constraints** - Time limits, space limits, input size
3. **Think of examples** - Edge cases, normal cases
4. **Choose data structure** - Based on operations needed
5. **Design algorithm** - Start with brute force, then optimize
6. **Code implementation** - Clean, readable code
7. **Test thoroughly** - Edge cases, normal cases

2. Common Patterns Recognition

Pattern	When to Use	Example Problems
Two Pointers	Sorted arrays, pairs	Two Sum II, Container With Water
Sliding Window	Subarrays, substrings	Longest Substring, Max Subarray
Fast & Slow Pointers	Linked lists, cycles	Cycle Detection, Middle Node
Binary Search	Sorted arrays, search space	Search Insert Position
DFS/BFS	Trees, graphs, connected components	Number of Islands
Dynamic Programming	Optimization, overlapping subproblems	Climbing Stairs
Backtracking	All possible solutions	Permutations, N-Queens

3. Time & Space Complexity Cheat Sheet

Data Structure	Access	Search	Insert	Delete	Space
Array	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table	N/A	O(1)	O(1)	O(1)	O(n)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Binary Heap	O(log n)	O(n)	O(log n)	O(log n)	O(n)

Algorithm	Best	Average	Worst	Space
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)

Practice Schedule for Mastery

Week 1-2: Foundations

- Arrays and Strings (5 problems/day)
- Basic operations, two pointers
- Problems: Two Sum, Valid Anagram, Reverse String

Week 3-4: Linear Data Structures

- Linked Lists, Stacks, Queues (4 problems/day)
- Problems: Reverse Linked List, Valid Parentheses

Week 5-6: Trees and Recursion

- Binary Trees, BST, Tree Traversals (4 problems/day)
- Problems: Max Depth, Same Tree, Invert Binary Tree

Week 7-8: Advanced Structures

- Heaps, Hash Tables, Tries (3 problems/day)
- Problems: Top K Elements, Group Anagrams

Week 9-10: Algorithms

- Sorting, Searching, Two Pointers (3 problems/day)
- Problems: Merge Intervals, Binary Search

Week 11-12: Graph Algorithms

- DFS, BFS, Union Find (3 problems/day)
- Problems: Number of Islands, Clone Graph

Week 13-14: Dynamic Programming

- 1D DP, 2D DP (2-3 problems/day)
- Problems: Climbing Stairs, House Robber, Coin Change

Week 15-16: Advanced Techniques

- Backtracking, Greedy (2-3 problems/day)
- Problems: Permutations, Combination Sum

Remember: Consistency is key! Practice daily, understand concepts deeply, and don't just memorize solutions. Focus on problem-solving patterns and apply them to new problems.

Resources for Continued Learning

1. **LeetCode** - Primary practice platform
2. **GeeksforGeeks** - Detailed explanations
3. **Cracking the Coding Interview** - Book
4. **Introduction to Algorithms (CLRS)** - Comprehensive textbook
5. **YouTube Channels**: Abdul Bari, Tushar Roy, Back to Back SWE

Happy coding! 🚀