7-Day DSA Mock Interview Preparation Plan

Day 1: Arrays

Question 1. Equilibrium Index of an array

```
private static int solve(int[] a, int size) {
    int cnt = 0;
    int[] prefixSum = new int[size];
    prefixSum[0] = a[0];
    for (int i = 1; i < size; i++) {
        prefixSum[i] = prefixSum[i - 1] + a[i];
    }
    for (int i = 0; i < size; i++) {
        // check if i = equilibrium index
        int leftSum = 0;
        if (i == 0) {
            leftSum = 0;
        } else {
            leftSum = prefixSum[i - 1] ;
        }
        int rightSum = prefixSum[size - 1] - prefixSum[i];
        if (leftSum == rightSum) {
            cnt++;
        }
    }
    return cnt;
}
```

Question 2. Max Sum Contiguous Subarray

```
private static int maxSubArray(final int[] A) {
    int sum = Integer.MIN_VALUE;
    int last = 0;

    for (int num : A) {
        last += num;
        sum = Math.max(sum, last);
        if (last < 0)
            last = 0;
    }
    return sum;
}
```

Question 3. Sum of all subarrays

```
private static int sumOfAllSubArrays(int[] A, Integer N) {
    int totalSum = 0;
```

```
    for (int i = 0; i < N; i++) {
        int sum = 0;
        for (int j = i; j < N; j++) {
            sum += A[j];
            totalSum += sum;
        }
    }
    return totalSum;
}
/**
 * Using Contribution Technique
 * **/
private static int sumOfAllSubArrays(int[] A, Integer N) {
    int totalSum;
    totalSum = 0;
    for (int i = 0; i < N; i++) {
        int totalNumberOfSubarrayAtIndexI = (i + 1) * (N - i);
        int contribution = A[i] * totalNumberOfSubarrayAtIndexI;
        totalSum += contribution;
    }
    return totalSum;
}
```

Question 4. Subarray with given sum and length

```
public int solve(int[] A, int B, int C) {
    int sum = 0;
    for (int i = 0; i < A.length; i++) {
        sum += A[i];
        if (i >= B) {
            sum -= A[i - B];
        }
        if (sum == C && i >= B - 1) {
            return 1;
        }
    }
    return 0;
}
```

Question 5. First Missing Integer

```
public int firstMissingPositive(ArrayList<Integer> A) {
    int n = A.size();
    for (int i = 0; i < n; i++) {
        if (A.get(i) > 0 && A.get(i) <= n) {
            int pos = A.get(i) - 1;
```

```
            if (A.get(pos) != A.get(i)) {
                Collections.swap(A, pos, i);
                i--;
                // We are doing i-- as we have swapped i'th element with pos'th element and
            }
        }
    }
    for (int i = 0; i < n; i++) {
        if (A.get(i) != i + 1)
            return (i + 1);
    }
    return n + 1;
}
```

Question 6. Rain water trapped

```
private static int trapWaterTrapped1(int[] A) {
    int N = A.length, ans = 0;
    int[] prefixMax = calculatePrefixMax(A);
    int[] suffixMax = calculateSuffixMax(A);
    for (int i = 1; i < N - 1; i++) {
        // Calculate the amount of water trapped
        int leftMax = prefixMax[i - 1]; // Max of all 0...(i - 1)
        int rightMax = suffixMax[i + 1]; // <ax of all (i + 1)...(N - 1)
        int height = Math.min(leftMax, rightMax);
        int waterTrapped = height - A[i];
        if (waterTrapped > 0) {
            ans += waterTrapped;
        }
    }
    return ans;
}
private static int[] calculatePrefixMax(int[] A) {
    int n = A.length;
    int[] prefixMax = new int[n];
    int maxSoFar = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        maxSoFar = Math.max(maxSoFar, A[i]);
        prefixMax[i] = maxSoFar;
    }
    return prefixMax;
}

private static int[] calculateSuffixMax(int[] A) {
    int n = A.length;
    int[] suffixMax = new int[n];
```

```
    int maxSoFar = Integer.MIN_VALUE;
    for (int i = n - 1; i >= 0; i--) {
        maxSoFar = Math.max(maxSoFar, A[i]);
        suffixMax[i] = maxSoFar;
    }
    return suffixMax;
}
```

Day 2: Searching and Sorting

Question 1. Rotated Sorted Array Search

```
public class Q4_Search_In_Rotated_Array {
    private static int searchInRotatedArray(int[] A, int K) {
        int left = 0, right = A.length - 1;
        while (left <= right) {
            int mid = (right + left) / 2;
            if (A[mid] == K) {
                return mid;
            }
            if (A[left] <= A[mid]) {
                if (A[left] <= K && A[mid] > K) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            } else {
                if (A[mid] < K && A[mid] >= K) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }
        return -1;
    }
    public static void main(String[] args) {
        int[] A = {10, 20, 30, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        int K = 20;
        int result = searchInRotatedArray(A, K);
        PrintUtils.print(result);
    }
}
```

Question 2. Painters Partition Problem

```java
public class Q1_Painters_Partition_Problem {
    /**
     * We have to paint N boards of length A1, A2, A3.....AN.
     * There are K painters available and each takes one unit of time to paint one unit of b
     * Find minimum time to get the job done.
     * Note: One painter will paint only a continuous section of the board.
     * **/
    private static int minTimeToPaint(int[] boards, int K) {
        int totalLength = 0, maxBoardLength = 0;
        int minimumTime = Integer.MAX_VALUE;
        for (int board: boards) {
            totalLength += board;
            maxBoardLength = Math.max(maxBoardLength, board);
        }
        int left = maxBoardLength, right = totalLength;
        while (left <= right) {
            int mid = (right + left) / 2;
            if (isPossible(boards, mid,  K)) {
                minimumTime = mid;
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return minimumTime;
    }
    private static boolean isPossible(int[] boards, int mid, int K) {
        int paintersRequired = 1;
        int currentBoardLength = 0;
        for (int i = 0; i < boards.length; i++) {
            currentBoardLength += boards[i];
            if (currentBoardLength > mid) {
                paintersRequired++;
                currentBoardLength = boards[i];
                if (paintersRequired > K) {
                    return false;
                }
            }
        }
        return true;
    }
    public static void main(String[] args) {
        int[] A = {3, 5, 1, 7, 8, 2, 5, 3, 10, 1, 4, 7, 5, 4, 6};
        int K = 4;
        int result = minTimeToPaint(A, K);
        PrintUtils.print(result);
```

```
    }
}
```

Question 3. Merge Sort

```
public class Q4_MergeSort {
    private static void mergeSort(int[] A, int start, int end) {
        if (start == end) return;
        int mid = (start + end) / 2;
        mergeSort(A, start, mid);
        mergeSort(A, mid + 1, end);
        merge(A, start, mid, end);
    }
    private static void merge(int[] A, int start, int mid, int end) {
        int P1 = start, P2 = mid + 1, P3 = 0;
        int[] tempArr = new int[end - start + 1];
        while (P1 <= mid && P2 <= end) {
            if (A[P1] < A[P2]) tempArr[P3++] = A[P1++];
            else tempArr[P3++] = A[P2++];
        }
        while (P1 <= mid)
            tempArr[P3++] = A[P1++];
        while (P2 <= end)
            tempArr[P3++] = A[P2++];
        for (int i = start; i <= end; i++)
            A[i] = tempArr[i - start];
    }
    public static void main(String[] args) {
        int[] A = {18, 17, 15, 2, 6, 10, 9, 1, 8};
        int start = 0, end = A.length - 1;
        mergeSort(A, start, end);
        PrintUtils.print1DArray(A);
    }
}
```

Question 4. B closest Point to Origin [Custom comparator]

```
public class Solution {
    public ArrayList<ArrayList<Integer>> solve(ArrayList<ArrayList<Integer>> A, int B) {
        ArrayList <ArrayList <Integer>> ans = new ArrayList <ArrayList <Integer>>();
        // sorts the list based on euclidean distance from origin
        Collections.sort(A, new Comparator<ArrayList <Integer>>() {
            public int compare(ArrayList <Integer> a, ArrayList <Integer> b) {
                long d1 = (long)a.get(0) * a.get(0) + (long)a.get(1) * a.get(1);
                long d2 = (long)b.get(0) * b.get(0) + (long)b.get(1) * b.get(1);
                if(d1 < d2)return -1;
```

```
                else if(d2 < d1)return 1;
                else return 0;
            }
        });
        for(int i = 0; i < B; i++){
            ans.add(A.get(i));
        }
        return ans;
    }
}
```

Question 5. Quick Sort

```
public class QuickSort {
    // Function to perform the quicksort algorithm
    public static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            // Partition the array and get the index of the pivot
            int pivotIndex = partition(array, low, high);
            // Recursively sort the sub-arrays on both sides of the pivot
            quickSort(array, low, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, high);
        }
    }
    // Function to partition the array and return the index of the pivot
    private static int partition(int[] array, int low, int high) {
        // Choose the last element as the pivot
        int pivot = array[high];
        int i = low - 1;
        // Iterate through the array
        for (int j = low; j < high; j++) {
            // If the current element is smaller than or equal to the pivot, swap it with th
            if (array[j] <= pivot) {
                i++;
                swap(array, i, j);
            }
        }
        // Swap the pivot element with the element at index i+1
        swap(array, i + 1, high);
        // Return the index of the pivot after partitioning
        return i + 1;
    }
    // Function to swap two elements in an array
    private static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
```

```
        array[j] = temp;
    }
    public static void main(String[] args) {
        int[] array = {7, 2, 1, 6, 8, 5, 3, 4};
        int n = array.length;
        System.out.println("Original Array: " + Arrays.toString(array));
        // Call the quickSort function to sort the array
        quickSort(array, 0, n - 1);
        System.out.println("Sorted Array: " + Arrays.toString(array));
    }
}

public class Q4_QuickSort {
    private static void quickSort(int[] A, int start, int end) {
        if (start >= end) return;
        int pivot = partition(A, start, end);
        quickSort(A, start, pivot - 1);
        quickSort(A, pivot + 1, end);
    }
    private static int partition(int[] A, int start, int end) {
        int P1 = start + 1, P2 = end;
        while (P1 <= P2) {
            if (A[start] >= A[P1]) P1++;
            else if (A[start] < A[P2]) P2--;
            else {
                swap(A, P1, P2);
                P1++; P2--;
            }
        }
        swap(A, start, P2);
        return P2;
    }
    private static void swap(int[] A, int start, int end) {
        int temp = A[start];
        A[start] = A[end];
        A[end] = temp;
    }
    public static void main(String[] args) {
        int[] A = {18, 8, 6, 3, 11, 14, 23, 20, 31, 27};
        int start = 0, end = A.length - 1;
        PrintUtils.print1DArray(A);
        quickSort(A, start, end);
        PrintUtils.printNewLine();
        PrintUtils.print1DArray(A);
    }
}
```

Study Binary Search

```java
public class BinarySearch {
    private static int binarySearch(int[] arr, int target) {
        int left = 0;
        int right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            // int mid = (right + left) / 2;
            if (arr[mid] == target) {
                return mid;
            } else if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {2, 6, 9, 13, 15, 18, 19};
        int target = 15;
        int result = binarySearch(arr, target);
        PrintUtils.print(result);
    }
}
```

Day 3: Stacks, Queues, and Hashing

Question 1. Longest Subarray Zero Sum

```java
public class Q3_Longest_Subarray_Zero_Sum {
    private static int solve(int[] A) {
        int n = A.length;
        int ans = 0;
        long sum = 0;
        HashMap<Long, Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
            sum = sum + A[i];
            if (sum == 0) {
                ans = Math.max(ans, i + 1);
            } else if (map.containsKey(sum)) {
                ans = Math.max(ans, i - map.get(sum));
            } else {
                map.put(sum, i);
            }
```

```
        }
        return ans;
    }

    public static void main(String[] args) {
        int[] A = {1, -2, 1, 2};
        int result = solve(A);
        PrintUtils.printInt(result);
    }
}
```

Question 2. Balanced Parenthesis

```
import java.util.Stack;

public class BalancedParenthesis {
    // Function to check if the expression has balanced parenthesis
    public static boolean isBalanced(String expression) {
        Stack<Character> stack = new Stack<>();

        for (char ch : expression.toCharArray()) {
            if (ch == '{' || ch == '(' || ch == '[') {
                // Opening bracket, push onto the stack
                stack.push(ch);
            } else if (ch == '}' || ch == ')' || ch == ']') {
                // Closing bracket, check if stack is not empty
                if (stack.isEmpty()) {
                    return false; // Unmatched closing bracket
                }

                // Pop the top element from the stack
                char top = stack.pop();

                // Check if the popped opening bracket matches the corresponding closing bra
                if ((ch == '}' && top != '{') || (ch == ')' && top != '(') || (ch == ']' &&
                    return false; // Mismatched brackets
                }
            }
        }
        // Check if there are unmatched opening brackets
        return stack.isEmpty();
    }

    public static void main(String[] args) {
        String expression1 = "{[()]}";
        String expression2 = "{[(])}";
```

```
        String expression3 = "{[}";

        System.out.println("Expression 1: " + isBalanced(expression1)); // true
        System.out.println("Expression 2: " + isBalanced(expression2)); // false
        System.out.println("Expression 3: " + isBalanced(expression3)); // false
    }
}

public class Solution {
    public int solve(String A) {
        HashMap < Character, Character > mp = new HashMap < Character, Character > ();
        Stack < Character > st = new Stack < Character > ();
        mp.put(')', '(');
        mp.put('}', '{');
        mp.put(']', '[');
        for (int i = 0; i < A.length(); i++) {
            char c = A.charAt(i);
            if (c == '(' || c == '[' || c == '{') {
                // push any opening bracket into the stack
                st.push(c);
            } else if (st.empty() || st.peek() != mp.get(c)) {
                // check if the last unpaired opening bracket is of the same type
                // as the current closing bracket
                return 0;
            } else {
                st.pop();
            }
        }
        // checks if all the opening brackets are paired
        if (st.empty())
            return 1;
        return 0;
    }
}
```

Question 3. Largest rectangle in histogram

```
public class Q2_Largest_Rectangle_in_Histogram {
    private static int largestRectangleArea(int[] A) {
        int n = A.length;
        int ans = 0;
        int[] left = smallestElementIndexLeft(A);
        int[] right = smallestElementIndexRight(A);

        for (int i = 0; i < n; i++) {
            int p1 = left[i];
```

```
            int p2 = right[i];
            int width = p2 - p1 - 1;
            ans = Math.max(ans, width * A[i]);
        }
        return ans;
    }

    private static int[] smallestElementIndexLeft(int[] A) {
        int n = A.length;
        int[] ans = new int[n];
        Arrays.fill(ans, -1);
        Stack<Integer> st = new Stack<>();

        for (int i = 0; i < n; i++) {
            while (!st.isEmpty() && A[st.peek()] >= A[i]) {
                st.pop();
            }
            if (!st.isEmpty()) {
                ans[i] = st.peek();
            }
            st.push(i);
        }
        return ans;
    }

    private static int[] smallestElementIndexRight(int[] A) {
        int n = A.length;
        int[] ans = new int[n];
        Arrays.fill(ans, n);
        Stack<Integer> st = new Stack<>();
        for (int i = n - 1; i >= 0; i--) {
            while (!st.isEmpty() && A[st.peek()] >= A[i]) {
                st.pop();
            }
            if (!st.isEmpty()) {
                ans[i] = st.peek();
            }
            st.push(i);
        }
        return ans;
    }

    public static void main(String[] args) {
        int[] A = {2, 1, 5, 6, 2, 3};
        int result = largestRectangleArea(A);
        PrintUtils.print(result);
```

```
    }
}
```

Question 4. Queue using stack

```
public class QueueUsingStacks {
    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public QueueUsingStacks() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    public void enqueue(int value) {
        stack1.push(value);
    }

    public int dequeue() throws IllegalAccessException {
        if (isEmpty()) {
            throw new IllegalAccessException("Queue is empty");
        }
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }

    public int peek() throws IllegalAccessException {
        if (isEmpty()) {
            throw new IllegalAccessException("Queue is empty");
        }
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }

    public boolean isEmpty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}
```

```
public class Q2_Implement_Queue_using_stacks {
    public static void main(String[] args) throws IllegalAccessException {
        QueueUsingStacks queue = new QueueUsingStacks();
        queue.enqueue(5);
        queue.enqueue(4);
        queue.enqueue(7);
        queue.enqueue(9);
        System.out.println("Dequeued element: " + queue.dequeue());
        queue.enqueue(8);
        queue.enqueue(10);
        System.out.println("Dequeued element: " + queue.dequeue());
        System.out.println("Dequeued element: " + queue.dequeue());
        queue.enqueue(14);
        System.out.println("Dequeued element: " + queue.dequeue());
        System.out.println("Dequeued element: " + queue.dequeue());
        queue.enqueue(14);
        System.out.println("Front element: " + queue.peek());
    }
}
```

Day 4 Linked List and Trees

Question 1. Detect a cycle in a linked list and find the starting node of the cycle

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}

public class CycleDetection {
    // Function to detect a cycle and find the starting node of the cycle
    public static ListNode detectCycle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;
        // Step 1: Detect cycle using Floyd's algorithm
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;

            // Check for cycle
```

```
            if (slow == fast) {
                // Step 2: Find starting node of the cycle
                slow = head;
                while (slow != fast) {
                    slow = slow.next;
                    fast = fast.next;
                }
                return slow; // Starting node of the cycle
            }
        }
        return null; // No cycle detected
    }
    public static void main(String[] args) {
        // Example usage
        ListNode head = new ListNode(3);
        head.next = new ListNode(2);
        head.next.next = new ListNode(0);
        head.next.next.next = new ListNode(-4);
        head.next.next.next.next = head.next; // Create a cycle

        ListNode cycleStart = detectCycle(head);
        if (cycleStart != null) {
            System.out.println("Starting node of the cycle: " + cycleStart.val);
        } else {
            System.out.println("No cycle detected");
        }
    }
}
```

Question 2. Find the lowest common ancestor (LCA) of two nodes in a binary tree.

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

    public TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
public class LowestCommonAncestor {
    // Function to find the Lowest Common Ancestor (LCA) of two nodes in a binary tree
    public static TreeNode findLCA(TreeNode root, TreeNode p, TreeNode q) {
```

```
        // Base case: If the current node is null, return null
        if (root == null) {
            return null;
        }
        // Base case: If either node is found, return the current node as LCA
        if (root == p || root == q) {
            return root;
        }
        // Recursively search for LCA in the left and right subtrees
        TreeNode leftLCA = findLCA(root.left, p, q);
        TreeNode rightLCA = findLCA(root.right, p, q);
        // If both sides return non-null values, the current node is LCA
        if (leftLCA != null && rightLCA != null) {
            return root;
        }
        // If only one side returns a non-null value, that side's result is the LCA
        return (leftLCA != null) ? leftLCA : rightLCA;
    }
    public static void main(String[] args) {
        // Example usage
        TreeNode root = new TreeNode(3);
        root.left = new TreeNode(5);
        root.right = new TreeNode(1);
        root.left.left = new TreeNode(6);
        root.left.right = new TreeNode(2);
        root.right.left = new TreeNode(0);
        root.right.right = new TreeNode(8);
        root.left.right.left = new TreeNode(7);
        root.left.right.right = new TreeNode(4);

        TreeNode nodeP = root.left;  // Node with value 5
        TreeNode nodeQ = root.right; // Node with value 1

        TreeNode lca = findLCA(root, nodeP, nodeQ);
        System.out.println("Lowest Common Ancestor: " + lca.val); // Output: 3
    }
}
```

Question 3. ZigZag Level order

```
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Stack;
class TreeNode {
    int val;
```

```
    TreeNode left;
    TreeNode right;

    public TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
public class ZigZagLevelOrderTraversal {
    // Function to perform ZigZag Level Order traversal
    public static List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> result = new LinkedList<>();
        if (root == null) {
            return result;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        boolean leftToRight = true;
        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> levelNodes = new LinkedList<>();
            Stack<Integer> tempStack = new Stack<>();
            for (int i = 0; i < levelSize; i++) {
                TreeNode current = queue.poll();
                if (leftToRight) {
                    levelNodes.add(current.val);
                } else {
                    tempStack.push(current.val);
                }
                if (current.left != null) {
                    queue.offer(current.left);
                }
                if (current.right != null) {
                    queue.offer(current.right);
                }
            }
            while (!tempStack.isEmpty()) {
                levelNodes.add(tempStack.pop());
            }
            result.add(levelNodes);
            leftToRight = !leftToRight; // Reverse the order for the next level
        }
        return result;
    }
    public static void main(String[] args) {
```

```
        // Example usage
        TreeNode root = new TreeNode(3);
        root.left = new TreeNode(9);
        root.right = new TreeNode(20);
        root.right.left = new TreeNode(15);
        root.right.right = new TreeNode(7);

        List<List<Integer>> result = zigzagLevelOrder(root);
        System.out.println("ZigZag Level Order Traversal: " + result);
        // Output: [[3], [20, 9], [15, 7]]
    }
}
```

Question 4. Remove Loop from Linked List

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

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

public class RemoveLoop {

    // Function to detect and remove a loop in a linked list
    public static void removeLoop(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;
        // Detect the loop using Floyd's cycle-finding algorithm
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) { // Loop detected
                break;
            }
        }
        if (slow != fast) { // If no loop is detected, return
            return;
        }
        // Move one pointer to the head and find the starting point of the loop
        ListNode ptr1 = head;
        ListNode ptr2 = slow;
        while (ptr1 != ptr2) {
```

```
            ptr1 = ptr1.next;
            ptr2 = ptr2.next;
        }
        // Move another pointer to find the node that contributes to the loop
        ListNode prev = head;

        while (prev.next != ptr1) {
            prev = prev.next;
            ptr2 = ptr2.next;
        }
        // Break the loop by setting the next of the contributing node to null
        ptr2.next = null;
    }
    public static void main(String[] args) {
        // Example usage
        ListNode head = new ListNode(1);
        head.next = new ListNode(2);
        head.next.next = new ListNode(3);
        head.next.next.next = new ListNode(4);
        head.next.next.next.next = new ListNode(5);
        // Create a loop for testing
        head.next.next.next.next.next = head.next.next;
        // Remove the loop
        removeLoop(head);
        // Print the modified linked list
        ListNode current = head;
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        // Output: 1 2 3 4 5
    }
}
```

Question 5. Binary Tree from In and Preorder

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    public TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
```

```java
public class BinaryTreeFromInPre {
    // Function to construct a binary tree from inorder and preorder traversals
    public static TreeNode buildTree(int[] preorder, int[] inorder) {
        return buildTreeHelper(preorder, 0, preorder.length - 1, inorder, 0, inorder.length
    }
    // Recursive helper function
    private static TreeNode buildTreeHelper(int[] preorder, int preStart, int preEnd,
                                            int[] inorder, int inStart, int inEnd) {
        // Base case: If the inorder or preorder array is empty, return null
        if (preStart > preEnd || inStart > inEnd) {
            return null;
        }
        // The root of the current subtree is the first element of the preorder array
        int rootValue = preorder[preStart];
        TreeNode root = new TreeNode(rootValue);
        // Find the index of the root element in the inorder array
        int rootIndexInorder = findRootIndex(inorder, inStart, inEnd, rootValue);
        // Recursively call for the left subtree
        root.left = buildTreeHelper(preorder, preStart + 1, preStart + rootIndexInorder - in
                                    inorder, inStart, rootIndexInorder - 1);
        // Recursively call for the right subtree
        root.right = buildTreeHelper(preorder, preStart + rootIndexInorder - inStart + 1, pr
                                     inorder, rootIndexInorder + 1, inEnd);
        return root;
    }
    // Function to find the index of a value in the inorder array
    private static int findRootIndex(int[] inorder, int start, int end, int value) {
        for (int i = start; i <= end; i++) {
            if (inorder[i] == value) {
                return i;
            }
        }
        return -1; // Not found (should not happen in a valid tree)
    }
    // Function to perform inorder traversal and print the elements of the tree
    public static void inorderTraversal(TreeNode root) {
        if (root != null) {
            inorderTraversal(root.left);
            System.out.print(root.val + " ");
            inorderTraversal(root.right);
        }
    }
    public static void main(String[] args) {
        // Example usage
        int[] preorder = {3, 9, 20, 15, 7};
        int[] inorder = {9, 3, 15, 20, 7};
```

```
        TreeNode root = buildTree(preorder, inorder);
        System.out.println("Inorder Traversal of Constructed Tree:");
        inorderTraversal(root);
        // Output: 9 3 15 20 7
    }
}
```

Question 6. Diameter of Binary Tree

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

    public TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
public class DiameterOfBinaryTree {
    // Global variable to store the diameter
    static int diameter = 0;
    // Function to calculate the diameter of a binary tree
    public static int diameterOfBinaryTree(TreeNode root) {
        calculateHeight(root);
        return diameter;
    }
    // Recursive helper function to calculate the height and update the diameter
    private static int calculateHeight(TreeNode node) {
        // Base case: If the current node is null, return 0
        if (node == null) {
            return 0;
        }
        // Recursively calculate the height of the left subtree
        int leftHeight = calculateHeight(node.left);
        // Recursively calculate the height of the right subtree
        int rightHeight = calculateHeight(node.right);
        // Update the diameter
        diameter = Math.max(diameter, leftHeight + rightHeight);
        // Return the height of the current subtree
        return 1 + Math.max(leftHeight, rightHeight);
    }
    public static void main(String[] args) {
        // Example usage
        TreeNode root = new TreeNode(1);
```

```
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);
        int result = diameterOfBinaryTree(root);
        System.out.println("Diameter of Binary Tree: " + result); // Output: 3
    }
}
```

Question 7. K Places Apart

```
import java.util.*;
public class SortQueueWithPriority {
    // Function to sort the queue with priorities using a window of size B
    public static List<Integer> sortQueue(List<Integer> queue, int B) {
        List<Integer> result = new ArrayList<>();
        // Create a min heap to represent the window of size B
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        // Initialize the heap with the first B elements from the queue
        for (int i = 0; i < B; i++) {
            minHeap.offer(queue.get(i));
        }
        // Traverse the queue from the (B+1)-th element to the end
        for (int i = B; i < queue.size(); i++) {
            // Pop the smallest element from the heap and add it to the result
            result.add(minHeap.poll());
            // Add the current element to the heap
            minHeap.offer(queue.get(i));
        }
        // Add the remaining elements from the heap to the result
        while (!minHeap.isEmpty()) {
            result.add(minHeap.poll());
        }
        return result;
    }
    public static void main(String[] args) {
        // Example usage
        List<Integer> queue = Arrays.asList(6, 5, 3, 2, 8, 10, 9);
        int B = 3;
        List<Integer> result = sortQueue(queue, B);
        System.out.println("Sorted Queue: " + result);
        // Output: [2, 3, 5, 6, 8, 9, 10]
    }
}
```

Day 5: Dynamic Programming

Question 1. 0-1 Knapsack

```java
public class Knapsack {
    // Function to solve the 0-1 Knapsack problem
    public static int knapsack(int[] weights, int[] values, int capacity) {
        int n = weights.length;
        int[][] dp = new int[n + 1][capacity + 1];
        // Initialize base cases
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= capacity; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                }
            }
        }
        // Build the dp array using the recurrence relation
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= capacity; j++) {
                if (weights[i - 1] <= j) {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weights[i - 1]] + values
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        return dp[n][capacity];
    }
    public static void main(String[] args) {
        // Example usage
        int[] weights = {2, 3, 4, 5};
        int[] values = {3, 4, 5, 6};
        int capacity = 5;
        int maxValue = knapsack(weights, values, capacity);
        System.out.println("Maximum Value in Knapsack: " + maxValue); // Output: 8
    }
}
```

Question 2. Fibonacci Sequence

```java
public class FibonacciMemoization {
    static int[] memo;
    // Function to calculate Fibonacci number using memoization
    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
```

```
        }
        if (memo[n] != -1) {
            return memo[n];
        }
        memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
        return memo[n];
    }
    private static int fibonacci1(int N) {
        int[] DP = new int[N + 1];
        DP[0] = 0;
        DP[1] = 1;
        for (int i = 2; i <= N; i++) {
            DP[i] = DP[i - 1] + DP[i - 2];
        }
        return DP[N];
    }

    public static void main(String[] args) {
        // Example usage
        int n = 6;
        memo = new int[n + 1];
        Arrays.fill(memo, -1);

        int result = fibonacci(n);

        System.out.println("Fibonacci(" + n + ") = " + result); // Output: 8
    }
}
```

Question 3. N Stairs

```
public class Q2_N_Stairs {
    /**
     * Problem: N Stairs
     * Given N steps. In how many ways can we go from 0th step to Nth step?
     * Note: From an ith step, we can go to (i + 1)th or (i + 2)th step.
     * **/
    // Function to calculate the number of ways to reach the Nth step
    private static int countWaysToNthStep(int N) {
        if (N <= 1) {
            return 1; // There is only one way to reach the 0th and 1st steps
        }
        // Create a table to store the number of ways for each step
        int[] DP = new int[N + 1];
        // Base cases: There is one way to reach the 0th and 1st steps
        DP[0] = 1;
```

```
        DP[1] = 1;
        // Calculate the number of ways for each step starting from the 2nd step
        for (int i = 2; i <= N; i++) {
            // The number of ways to reach the ith step is the sum of ways to reach (i-1)th
            DP[i] = DP[i - 1] + DP[i - 2];
        }
        // The result is stored in the last entry of the ways array (Nth step)
        return DP[N];
    }
    public static void main(String[] args) {
        int N = 5;
        int result = countWaysToNthStep(N);
        System.out.println(result);
    }
}
```

Question 4. Minimum no of squares

```
public class MinSquares {
    // Function to calculate the minimum number of squares using dynamic programming
    public static int minSquares(int n) {
        int[] dp = new int[n + 1];
        dp[0] = 0;
        for (int i = 1; i <= n; i++) {
            dp[i] = Integer.MAX_VALUE; // Initialize with a large value
            // Try every square less than or equal to i
            for (int j = 1; j * j <= i; j++) {
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
            }
        }
        return dp[n];
    }
    public static void main(String[] args) {
        // Example usage
        int n = 12;
        int result = minSquares(n);
        System.out.println("Minimum number of squares to represent " + n + ": " + result); /
    }
}
```

Question 5. Max Sum Without Adjacent Elements

```
public class MaxSumWithoutAdjacent {
    static int[] dpArr;

    private static int maxSumWithoutAdjacent(int[][] A) {
```

```
        dpArr = new int[A[0].length];
        Arrays.fill(dpArr, -1);
        return recursion(A, 0);
    }

    private static int recursion(int[][] A, int col) {
        if (col >= A[0].length) return 0;

        if (dpArr[col] != -1) return dpArr[col];

        int a = recursion(A, col + 2) + Math.max(A[0][col], A[1][col]); //if curr col selec
        int b = recursion(A, col + 1); //if curr col not selected then adjacent one has to b
        dpArr[col] = Math.max(a, b); //taking max of the two choices and updating DP array
        return dpArr[col];
    }

    public static void main(String[] args) {
        // Example usage
        int[][] grid = {
            {1, 2, 3, 4},
            {2, 3, 4, 5}
        };
        int result = maxSumWithoutAdjacent(grid);

        System.out.println("Maximum sum without adjacent elements: " + result); // Output: 8
    }
}
```

Question 6. Longest common subsequence

```
public class LongestCommonSubsequence {
    // Function to calculate the length of the Longest Common Subsequence (LCS)
    public static int longestCommonSubsequence(String A, String B) {
        int m = A.length();
        int n = B.length();
        // Create a 2D array to store the length of the LCS at each position
        int[][] dp = new int[m + 1][n + 1];
        // Initialize the first row (base case for empty string A)
        for (int j = 0; j <= n; j++) {
            dp[0][j] = 0;
        }
        // Initialize the first column (base case for empty string B)
        for (int i = 0; i <= m; i++) {
            dp[i][0] = 0;
        }
        // Iterate through the characters of both strings
```

```
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (A.charAt(i - 1) == B.charAt(j - 1)) {
                    // Characters match
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    // Characters do not match
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        // The final result is stored in dp[m][n]
        return dp[m][n];
    }
    public static void main(String[] args) {
        // Example usage
        String A = "ABCBDAB";
        String B = "BDCAB";
        int result = longestCommonSubsequence(A, B);
        System.out.println("Length of Longest Common Subsequence: " + result); // Output: 4
    }
}
```

Day 6: Graphs

Question 1. Rotten oranges

```
public class Solution {
    public int solve(int[][] grid) {
        Queue <int[]> queue = new LinkedList <> ();
        int fresh = 0;
        int time = 0;
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[0].length; j++) {
                if (grid[i][j] == 2) {
                    queue.add(new int[] { i, j });
                } else if (grid[i][j] == 1) {
                    fresh++;
                }
            }
        }

        int[][] direction = {
            { 1, 0 },
            { 0, 1 },
            {-1, 0 },
```

```
            { 0, -1 } };
        while (!queue.isEmpty() && fresh > 0) {
            time++;
            int size = queue.size();
            while (size > 0) {
                int[] bad = queue.poll();
                for (int[] dir: direction) {
                    int nrow = bad[0] + dir[0];
                    int ncol = bad[1] + dir[1];

                    if (nrow < 0 || nrow >= grid.length || ncol < 0 || ncol >= grid[0].lengt
                        continue;
                    }
                    grid[nrow][ncol] = 2;
                    fresh--;
                    queue.add(new int[] {
                        nrow,
                        ncol
                    });
                }
                size--;
            }
        }

        if (fresh == 0) {
            return time;
        } else
            return -1;
    }
    public static void main(String[] args) {
        int[][] grid = {
                {2, 1, 1},
                {1, 1, 0},
                {0, 1, 1}
        };

        int minTime = solve(grid);

        System.out.println("Minimum Time to Rot All Oranges: " + minTime);
    }
}
```

Question 2. Number of islands

```
public class Q2_Number_of_Islands_2 {
    private static int numIslands(char[][] grid) {
```

```
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return 0;
        }
        int rows = grid.length;
        int cols = grid[0].length;
        boolean[][] visited = new boolean[rows][cols];
        int islandCount = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] == '1' && !visited[i][j]) {
                    islandCount++;
                    dfs(grid, i, j, visited);
                }
            }
        }
        return islandCount;
    }
    private static void dfs(char[][] grid, int row, int col, boolean[][] visited) {
        int rows = grid.length;
        int cols = grid[0].length;
        // Base cases
        if (row < 0 || row >= rows || col < 0 || col >= cols || grid[row][col] == '0' || vis
            return;
        }
        visited[row][col] = true;
        // Explore neighbors (up, down, left, right)
        dfs(grid, row - 1, col, visited);
        dfs(grid, row + 1, col, visited);
        dfs(grid, row, col - 1, visited);
        dfs(grid, row, col + 1, visited);
    }
    public static void main(String[] args) {
        char[][] grid = {
                {'1', '1', '0', '0', '0'},
                {'0', '1', '0', '1', '0'},
                {'1', '0', '0', '1', '1'},
                {'0', '0', '0', '0', '0'},
                {'1', '0', '1', '1', '1'}
        };
        int islandCount = numIslands(grid);
        System.out.println("Number of Islands: " + islandCount);
    }
}
```

Question 3. Cycle in directed graph

```java
public class Q1_Cycle_in_a_directed_graph_2 {
    private int numVertices;
    private List<List<Integer>> adjacencyList;

    public Q1_Cycle_in_a_directed_graph_2(int numVertices) {
        this.numVertices = numVertices;
        this.adjacencyList = new ArrayList<>();
        for (int i = 0; i < this.numVertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }
    private void addEdge(int source, int destination) {
        adjacencyList.get(source).add(destination);
    }

    private boolean hasCycle() {
        boolean[] vis = new boolean[numVertices];
        boolean[] path = new boolean[numVertices];

        for (int i = 0; i < numVertices; i++) {
            if (!vis[i] && dfs(i, vis, path)) {
                return true;
            }
        }
        return false;
    }

    private boolean dfs(int s, boolean[] vis, boolean[] path) {
        vis[s] = true;
        path[s] = true;

        for (int v : adjacencyList.get(s)) {
            if (!vis[v]) {
                if (dfs(v, vis, path)) {
                    return true;
                }
            } else if (path[v]) {
                return true; // Cycle detected
            }
        }

        path[s] = false; // Backtrack
        return false;
    }
    public static void main(String[] args) {
        int numNodes = 4;
```

```
        Q1_Cycle_in_a_directed_graph_2 graph = new Q1_Cycle_in_a_directed_graph_2(numNodes);

        graph.addEdge(0, 1);
        graph.addEdge(1, 2);
        graph.addEdge(2, 0);
        graph.addEdge(2, 3);

        if (graph.hasCycle()) {
            System.out.println("The graph has a cycle.");
        } else {
            System.out.println("The graph does not have a cycle.");
        }
    }
}
```

Question 4. Dijkstra

```
public class Dijkstra_s_Algorithm {
    static class Edge implements Comparable<Edge> {
        int destination, weight;

        public Edge(int destination, int weight) {
            this.destination = destination;
            this.weight = weight;
        }

        @Override
        public int compareTo(Edge other) {
            return Integer.compare(this.weight, other.weight);
        }
    }
    /**
     * Dijkstra's algorithm is a popular algorithm for finding the shortest
     * paths between nodes in a graph. Here's a simple Java implementation of Dijkstra's alg
     * In this example, the graph is represented as an adjacency list. The Edge class is use
     * represent edges with their destinations and weights. The dijkstra function calculates
     * the shortest distances from the start vertex to all other vertices in the graph.
     * The result is then printed to the console.
     * **/
    private static int[] dijkstra(ArrayList<ArrayList<Edge>> graph, int start) {
        int vertices = graph.size();
        int[] distance = new int[vertices];
        Arrays.fill(distance, Integer.MAX_VALUE);

        PriorityQueue<Edge> minHeap = new PriorityQueue<>();
        distance[start] = 0;
```

```
        minHeap.add(new Edge(start, 0));

        while (!minHeap.isEmpty()) {
            Edge current = minHeap.poll();
            int currentVertex = current.destination;

            for (Edge neighbor : graph.get(currentVertex)) {
                int newDistance = distance[currentVertex] + neighbor.weight;
                if (newDistance < distance[neighbor.destination]) {
                    distance[neighbor.destination] = newDistance;
                    minHeap.add(new Edge(neighbor.destination, newDistance));
                }
            }
        }

        return distance;
    }

    public static void main(String[] args) {
        int vertices = 6;
        ArrayList<ArrayList<Edge>> graph = new ArrayList<>(vertices);
        for (int i = 0; i < vertices; i++) {
            graph.add(new ArrayList<>());
        }
        // Add edges to the graph
        graph.get(0).add(new Edge(1, 2));
        graph.get(0).add(new Edge(2, 4));
        graph.get(1).add(new Edge(2, 1));
        graph.get(1).add(new Edge(3, 7));
        graph.get(2).add(new Edge(4, 3));
        graph.get(3).add(new Edge(4, 1));
        graph.get(3).add(new Edge(5, 5));
        graph.get(4).add(new Edge(5, 2));

        int startVertex = 0;
        int[] shortestDistances = dijkstra(graph, startVertex);

        // Print the shortest distances
        System.out.println("Shortest Distances from vertex " + startVertex + ":");
        for (int i = 0; i < vertices; i++) {
            System.out.println("To vertex " + i + ": " + shortestDistances[i]);
        }
    }
}
```

Question 5. depth-first search

```java
public class Q2_Graph_Traversal_Depth_First_Search_DFS_1 {
    /**
     * Graph Traversal DFS
     * In simple words, traversal means the process of visiting every node in the graph.
     * There are 2 standard methods of graph traversal Breadth-First Search and Depth First
     * Depth First Search
     * Depth First Search (DFS) is a graph traversal algorithm, where
     * we start from a selected(source) node and go into the depth of
     * this node by recursively calling the DFS function until no children are encountered.
     * When the dead-end is reached, this algorithm backtracks and starts visiting
     * the other children of the current node.
     * **/
    private int numVertices;
    private List<List<Integer>> adjacencyList;

    public Q2_Graph_Traversal_Depth_First_Search_DFS_1(int numVertices) {
        this.numVertices = numVertices;
        this.adjacencyList = new ArrayList<>();
        for (int i = 0; i < this.numVertices; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

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

    private boolean[] visited;
    private  void traversalDFS(int N /* number of nodes */) {
        visited = new boolean[N + 1];
        Arrays.fill(visited, false);
        for (int i = 1; i <= N; i++) {
            if (!visited[i]) {
                this.dfs(i);
            }
        }
    }
    private void dfs(int s) {
        System.out.print(s + ", ");
        visited[s] = true;
        for (List<Integer> list: this.adjacencyList) {
            for (int v : list) {
                if (!visited[v]) {
```

```
                    dfs(v);
                }
            }
        }
    }
    public static void main(String[] args) {
        int numNodes = 5;
        Q2_Graph_Traversal_Depth_First_Search_DFS_1 graph =
                new Q2_Graph_Traversal_Depth_First_Search_DFS_1(numNodes);

        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 3);
        graph.addEdge(2, 5);
        graph.addEdge(3, 4);

        System.out.println("Depth First Search starting from node 0:");
        graph.traversalDFS(numNodes);
    }
}
```

Question 6. breadth-first search

```
public class Q3_Graph_Traversal_Breadth_First_Search_Search_BFS_2 {
    private int numVertices;
    private Map<Integer, List<Integer>> adjacencyList;

    public Q3_Graph_Traversal_Breadth_First_Search_Search_BFS_2(int numVertices) {
        this.numVertices = numVertices;
        this.adjacencyList = new HashMap<>();
        for (int i = 0; i < numVertices; i++) {
            adjacencyList.put(i, new ArrayList<>());
        }
    }

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

    public void bfsTraversal(int startVertex) {
        boolean[] visited = new boolean[numVertices];
        Queue<Integer> queue = new LinkedList<>();

        visited[startVertex] = true;
        queue.offer(startVertex);
```

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

            for (int neighbor : adjacencyList.get(currentVertex)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.offer(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {
        int numNodes = 5;
        Q3_Graph_Traversal_Breadth_First_Search_Search_BFS_2 graph = new Q3_Graph_Traversal_

        graph.addEdge(0, 1);
        graph.addEdge(0, 4);
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 3);
        graph.addEdge(3, 4);

        System.out.println("Breadth First Search Traversal starting from node 0:");
        graph.bfsTraversal(0);
    }
}
```