# Amazon

15 August 2025 04:02

Alright — I'll prepare **Day 1 (Arrays)**, **Day 2 (Binary)**, and **Day 3 (Dynamic Programming)** in full **LeetCode-style** with:

- **Problem Statement**
- **Example(s)**
- **Constraints**
- **Approach** (clear, interview-ready explanation)
- **Java Solution** (clean, with comments, graduate style)

Here we go — you can paste this directly into **Notepad/OneNote**.

# Day 1: Arrays

## 1. Two Sum

**Problem:**
Given an array of integers nums and an integer target, return the indices of the two numbers such that they add up to target.
You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order.
**Example:**
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: nums[0] + nums[1] = 2 + 7 = 9
**Constraints:**

- 2 <= nums.length <= 10^4
- -10^9 <= nums[i] <= 10^9
- -10^9 <= target <= 10^9
- Only one valid answer exists.

**Approach:**
Use a HashMap to store each number and its index. For each element, check if target - nums[i] exists in the map. If yes, return indices.
**Java Code:**

```
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[0];
}
```

## 2. Best Time to Buy and Sell Stock

**Problem:**
You are given an array prices where prices[i] is the price of a stock on the i-th day.
Maximize your profit by choosing a single day to buy one stock and a different day to sell. Return the maximum profit. If no profit is possible, return 0.
**Example:**
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 1 (price=1) and sell on day 4 (price=6)
**Constraints:**

- 1 <= prices.length <= 10^5

- 0 <= prices[i] <= 10^4

**Approach:**
Track the minimum price so far and calculate the profit for each day, updating the maximum profit.
**Java Code:**
```
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;
}
```

## 3. Contains Duplicate

**Problem:**
Given an integer array nums, return true if any value appears at least twice in the array, and false if every element is distinct.
**Example:**
Input: nums = [1,2,3,1]
Output: true
**Constraints:**
- 1 <= nums.length <= 10^5
- -10^9 <= nums[i] <= 10^9

**Approach:**
Use a HashSet to check for duplicates while iterating.
**Java Code:**
```
public boolean containsDuplicate(int[] nums) {
   Set<Integer> set = new HashSet<>();
   for (int num : nums) {
      if (!set.add(num)) return true;
   }
   return false;
}
```

## 4. Product of Array Except Self

**Problem:**
Given an integer array nums, return an array answer where answer[i] is the product of all the elements except nums[i]. Solve without division in O(n).
**Example:**
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
**Constraints:**
- 2 <= nums.length <= 10^5
- -30 <= nums[i] <= 30
- The product fits in a 32-bit integer.

**Approach:**
Compute prefix and suffix products in two passes.
**Java Code:**
```
public int[] productExceptSelf(int[] nums) {
   int n = nums.length;
   int[] result = new int[n];
   int prefix = 1;
   for (int i = 0; i < n; i++) {
      result[i] = prefix;
```

```
        prefix *= nums[i];
    }
    int suffix = 1;
    for (int i = n - 1; i >= 0; i--) {
        result[i] *= suffix;
        suffix *= nums[i];
    }
    return result;
}
```

## 5. Kadane's Algorithm (Maximum Subarray Sum)

**Problem:**

Given an integer array nums, find the contiguous subarray with the largest sum and return its sum.

**Example:**

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6

**Constraints:**

- 1 <= nums.length <= 10^5
- -10^4 <= nums[i] <= 10^4

**Approach:**

Iterate and keep track of current subarray sum; reset if it goes negative.

**Java Code:**

```
public int maxSubArray(int[] nums) {
    int maxSum = nums[0];
    int currentSum = nums[0];
    for (int i = 1; i < nums.length; i++) {
        currentSum = Math.max(nums[i], currentSum + nums[i]);
        maxSum = Math.max(maxSum, currentSum);
    }
    return maxSum;
}
```

# Day 2: Binary

## 1. Sum of Two Integers

**Problem:**
Given two integers a and b, return the sum of the two integers without using the operators + and -.
**Example:**

Input: a = 1, b = 2
Output: 3
**Constraints:**

- -1000 <= a, b <= 1000

**Approach:**
Use bitwise operations:

- XOR (^) gives sum without carry.
- AND (&) followed by left shift gives carry.
- Repeat until carry becomes zero.

**Java Code:**

```
public int getSum(int a, int b) {
   while (b != 0) {
      int carry = (a & b) << 1;
      a = a ^ b;
      b = carry;
   }
   return a;
}
```

## 2. Number of 1 Bits

**Problem:**
Write a function that takes an unsigned integer and returns the number of '1' bits in its binary representation.
**Example:**

Input: n = 00000000000000000000000000001011
Output: 3
**Constraints:**

- The input must be treated as an unsigned integer.

**Approach:**
Iterate through bits and count set bits using bitwise AND.
**Java Code:**

```
public int hammingWeight(int n) {
   int count = 0;
   while (n != 0) {
      count += (n & 1);
      n >>>= 1; // unsigned right shift
   }
   return count;
}
```

## 3. Counting Bits

**Problem:**
Given an integer n, return an array ans where ans[i] is the number of 1 bits in the binary representation

of i.
**Example:**

Input: n = 5
Output: [0,1,1,2,1,2]
**Constraints:**
- 0 <= n <= 10^5

**Approach:**
Use DP: bits[i] = bits[i >> 1] + (i & 1).
**Java Code:**

```
public int[] countBits(int n) {
   int[] ans = new int[n + 1];
   for (int i = 1; i <= n; i++) {
      ans[i] = ans[i >> 1] + (i & 1);
   }
   return ans;
}
```

## 4. Find Missing Number in an Array

**Problem:**
Given an array nums containing n distinct numbers in the range [0, n], return the missing number.
**Example:**

Input: nums = [3,0,1]
Output: 2
**Constraints:**
- n == nums.length
- 0 <= nums[i] <= n
- All numbers are unique.

**Approach:**
Use XOR of all indices and elements.
**Java Code:**

```
public int missingNumber(int[] nums) {
   int xor = nums.length;
   for (int i = 0; i < nums.length; i++) {
      xor ^= i ^ nums[i];
   }
   return xor;
}
```

## 5. Reverse Bits

**Problem:**
Reverse bits of a given 32 bits unsigned integer.
**Example:**

Input: n = 00000010100101000001111010011100
Output:  964176192
**Constraints:**
- The input must be treated as unsigned.

**Approach:**
Shift result left and append the last bit of n in each iteration.

**Java Code:**

```
public int reverseBits(int n) {
   int result = 0;
   for (int i = 0; i < 32; i++) {
      result <<= 1;
      result |= (n & 1);
      n >>>= 1;
   }
   return result;
}
```

# Day 3: Dynamic Programming

## 1. Climbing Stairs

**Problem:**
You are climbing a staircase. It takes n steps to reach the top.
Each time you can climb 1 or 2 steps. Return the number of distinct ways to climb to the top.
**Example:**

```
Input: n = 3
Output: 3
Explanation: 1+1+1, 1+2, 2+1
```

**Constraints:**

- 1 <= n <= 45

**Approach:**
This is Fibonacci: ways[i] = ways[i-1] + ways[i-2].
**Java Code:**

```
public int climbStairs(int n) {
   if (n <= 2) return n;
   int a = 1, b = 2;
   for (int i = 3; i <= n; i++) {
      int temp = a + b;
      a = b;
      b = temp;
   }
   return b;
}
```

## 2. Coin Change

**Problem:**
Given coins of different denominations and an amount, return the minimum coins needed to make up that amount. If impossible, return -1.
**Example:**

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 5+5+1
```

**Constraints:**

- 1 <= coins.length <= 12
- 1 <= coins[i] <= 2^31 - 1
- 0 <= amount <= 10^4

**Approach:**
DP: dp[i] = min(dp[i], dp[i-coin] + 1).
**Java Code:**

```
public int coinChange(int[] coins, int amount) {
   int max = amount + 1;
   int[] dp = new int[amount + 1];
   Arrays.fill(dp, max);
   dp[0] = 0;
   for (int coin : coins) {
      for (int i = coin; i <= amount; i++) {
         dp[i] = Math.min(dp[i], dp[i - coin] + 1);
      }
   }
   return dp[amount] > amount ? -1 : dp[amount];
}
```

## 3. Longest Increasing Subsequence

**Problem:**
Given an integer array nums, return the length of the longest strictly increasing subsequence.
**Example:**

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: LIS is [2,3,7,101]
```

**Constraints:**

- 1 <= nums.length <= 2500
- -10^4 <= nums[i] <= 10^4

**Approach:**
DP: dp[i] = max(dp[j] + 1) for all j < i with nums[j] < nums[i].
**Java Code:**

```
public int lengthOfLIS(int[] nums) {
   int[] dp = new int[nums.length];
   Arrays.fill(dp, 1);
   int maxLen = 1;
   for (int i = 1; i < nums.length; i++) {
      for (int j = 0; j < i; j++) {
         if (nums[i] > nums[j]) {
            dp[i] = Math.max(dp[i], dp[j] + 1);
         }
      }
      maxLen = Math.max(maxLen, dp[i]);
   }
   return maxLen;
}
```

## 4. Longest Common Subsequence

**Problem:**
Given two strings text1 and text2, return the length of their longest common subsequence.
**Example:**

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
```

Explanation: LCS is "ace"

**Constraints:**

- 1 <= text1.length, text2.length <= 1000
- Only lowercase letters.

**Approach:**

2D DP: dp[i][j] = LCS length of first i chars of text1 and first j chars of text2.

**Java Code:**

```
public int longestCommonSubsequence(String text1, String text2) {
   int m = text1.length(), n = text2.length();
   int[][] dp = new int[m + 1][n + 1];
   for (int i = 1; i <= m; i++) {
      for (int j = 1; j <= n; j++) {
         if (text1.charAt(i - 1) == text2.charAt(j - 1))
            dp[i][j] = dp[i - 1][j - 1] + 1;
         else
            dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
   }
   return dp[m][n];
}
```

## 5. Word Break

**Problem:**

Given a string s and a dictionary of words, return true if s can be segmented into one or more dictionary words.

**Example:**

```
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
```

**Constraints:**

- 1 <= s.length <= 300
- 1 <= wordDict.length <= 1000

**Approach:**

DP: dp[i] = true if dp[j] is true and s[j:i] in dict.

**Java Code:**

```
public boolean wordBreak(String s, List<String> wordDict) {
   Set<String> set = new HashSet<>(wordDict);
   boolean[] dp = new boolean[s.length() + 1];
   dp[0] = true;
   for (int i = 1; i <= s.length(); i++) {
      for (int j = 0; j < i; j++) {
         if (dp[j] && set.contains(s.substring(j, i))) {
            dp[i] = true;
            break;
         }
      }
   }
   return dp[s.length()];
}
```

## 6. Combination Sum

**Problem:**
Given an array of distinct integers candidates and a target integer target, return all unique combinations of candidates where the chosen numbers sum to target.
You may use the same number unlimited times.
**Example:**

Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
**Constraints:**

- 1 <= candidates.length <= 30
- 1 <= candidates[i] <= 200
- 1 <= target <= 500

**Approach:**
Use DFS backtracking. At each step, include current number or move to next.
**Java Code:**

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
   List<List<Integer>> result = new ArrayList<>();
   backtrack(result, new ArrayList<>(), candidates, target, 0);
   return result;
}
private void backtrack(List<List<Integer>> res, List<Integer> temp, int[] nums, int remain, int start) {
   if (remain < 0) return;
   if (remain == 0) {
      res.add(new ArrayList<>(temp));
      return;
   }
   for (int i = start; i < nums.length; i++) {
      temp.add(nums[i]);
      backtrack(res, temp, nums, remain - nums[i], i); // reuse allowed
      temp.remove(temp.size() - 1);
   }
}
```

## 7. Maximum Sum of Non-Adjacent Elements (DP 5)

**Problem:**
Given an array nums, find the maximum sum of elements such that no two chosen elements are adjacent.
**Example:**

Input: nums = [3,2,5,10,7]
Output: 15
Explanation: 3 + 10 + 2 or 3 + 7 + 5, etc.
**Constraints:**

- 1 <= nums.length <= 10^5
- 0 <= nums[i] <= 10^4

**Approach:**
DP relation: dp[i] = max(dp[i-1], dp[i-2] + nums[i]).
**Java Code:**

```
public int maxNonAdjacentSum(int[] nums) {
```

```
    if (nums.length == 1) return nums[0];
    int prev2 = nums[0], prev1 = Math.max(nums[0], nums[1]);
    for (int i = 2; i < nums.length; i++) {
        int curr = Math.max(prev1, prev2 + nums[i]);
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

## 8. House Robber (DP 6)

**Problem:**
You are a robber planning to rob houses along a street. Each house has some money, but you can't rob two adjacent houses. Find the maximum amount you can rob.
**Example:**

Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (1) + house 3 (3)
**Constraints:**

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 400

**Approach:**
Same as "max sum non-adjacent elements" problem.
**Java Code:**

```
public int rob(int[] nums) {
    if (nums.length == 1) return nums[0];
    int prev2 = nums[0], prev1 = Math.max(nums[0], nums[1]);
    for (int i = 2; i < nums.length; i++) {
        int curr = Math.max(prev1, prev2 + nums[i]);
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

## 9. Decode Ways

**Problem:**
A message containing letters 'A'-'Z' is encoded where 'A' = "1", ..., 'Z' = "26".
Given a string of digits, return the total number of ways to decode it.
**Example:**

Input: s = "226"
Output: 3
Explanation: "2 2 6", "22 6", "2 26"
**Constraints:**

- 1 <= s.length <= 100
- s contains only digits and no leading zero unless "0" itself.

**Approach:**
DP:

- If s[i] != '0', dp[i] += dp[i-1]
- If s[i-2..i] in 10–26, dp[i] += dp[i-2]

**Java Code:**

```
public int numDecodings(String s) {
   if (s.charAt(0) == '0') return 0;
   int n = s.length();
   int[] dp = new int[n + 1];
   dp[0] = dp[1] = 1;
   for (int i = 2; i <= n; i++) {
      if (s.charAt(i - 1) != '0') dp[i] += dp[i - 1];
      int twoDigit = Integer.parseInt(s.substring(i - 2, i));
      if (twoDigit >= 10 && twoDigit <= 26) dp[i] += dp[i - 2];
   }
   return dp[n];
}
```

## 10. Grid Unique Paths

**Problem:**
A robot is at the top-left corner of an m x n grid. The robot can only move down or right. Count the number of possible unique paths to reach the bottom-right corner.
**Example:**

Input: m = 3, n = 7
Output: 28
**Constraints:**

- 1 <= m, n <= 100

**Approach:**
DP: dp[i][j] = dp[i-1][j] + dp[i][j-1]
**Java Code:**

```
public int uniquePaths(int m, int n) {
   int[][] dp = new int[m][n];
   for (int i = 0; i < m; i++) dp[i][0] = 1;
   for (int j = 0; j < n; j++) dp[0][j] = 1;
   for (int i = 1; i < m; i++) {
      for (int j = 1; j < n; j++) {
         dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
      }
   }
   return dp[m - 1][n - 1];
}
```

## 11. Jump Game

**Problem:**
You are given an integer array nums. You are initially positioned at the first index, and each element represents the maximum jump length from that position.
Return true if you can reach the last index, otherwise false.
**Example:**

Input: nums = [2,3,1,1,4]
Output: true
**Constraints:**

- 1 <= nums.length <= 10^4
- 0 <= nums[i] <= 10^5

**Approach:**
Greedy: Track the furthest reachable index. If you ever reach beyond the last index, return true.
**Java Code:**

```
public boolean canJump(int[] nums) {
   int maxReach = 0;
   for (int i = 0; i < nums.length; i++) {
      if (i > maxReach) return false;
      maxReach = Math.max(maxReach, i + nums[i]);
   }
   return true;
}
```

# 1. Clone Graph

You are given a reference to a node in a connected undirected graph. Each node has a unique value and a list of its neighbors.
Return a deep copy (clone) of the graph.

**Example:**

Input: Node with value 1 connected to nodes 2 and 4, etc.
Output: Deeply cloned graph with the same connections.

**Constraints:**

- Number of nodes is between 1 and 100.
- No duplicate edges.

**Approach:**

- Use DFS or BFS.
- Maintain a map from original node → cloned node to avoid revisiting.

**Java Code:**

```
class Node {
   public int val;
   public List<Node> neighbors;
   public Node() { neighbors = new ArrayList<>(); }
   public Node(int val) { this.val = val; neighbors = new ArrayList<>(); }
}
class Solution {
   private Map<Node, Node> visited = new HashMap<>();

   public Node cloneGraph(Node node) {
      if (node == null) return null;

      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(cloneGraph(neighbor));
      }

      return clone;
    }
}
```

# 2. Course Schedule I

You are given numCourses and a list of prerequisites.
Determine if you can finish all courses.

**Example:**

Input: numCourses = 2, prerequisites = [[1,0]]
Output: true

**Constraints:**

- 1 ≤ numCourses ≤ 2000
- prerequisites[i].length == 2

**Approach:**

- Model as a directed graph.
- Use DFS to detect cycles (cycle means impossible).

**Java Code:**

```
class Solution {
   public boolean canFinish(int numCourses, int[][] prerequisites) {
      List<List<Integer>> graph = new ArrayList<>();
      for (int i = 0; i < numCourses; i++) graph.add(new ArrayList<>());

      for (int[] pre : prerequisites) {
        graph.get(pre[1]).add(pre[0]);
      }

      int[] visited = new int[numCourses]; // 0=unvisited, 1=visiting, 2=done

      for (int i = 0; i < numCourses; i++) {
        if (!dfs(graph, visited, i)) return false;
      }
      return true;
   }

   private boolean dfs(List<List<Integer>> graph, int[] visited, int course) {
      if (visited[course] == 1) return false; // cycle
      if (visited[course] == 2) return true;  // already done

      visited[course] = 1;
      for (int next : graph.get(course)) {
        if (!dfs(graph, visited, next)) return false;
      }
      visited[course] = 2;
      return true;
   }
}
```

# 3. Pacific Atlantic Water Flow

Given an m x n matrix of heights, determine which cells can flow to both the Pacific and Atlantic oceans.

**Approach:**

- Perform DFS/BFS from ocean borders and mark reachable cells.
- Intersect sets to get cells that can reach both.

**Java Code:**

```
class Solution {
   int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};
   public List<List<Integer>> pacificAtlantic(int[][] heights) {
      int m = heights.length, n = heights[0].length;
      boolean[][] pacific = new boolean[m][n];
      boolean[][] atlantic = new boolean[m][n];

      for (int i = 0; i < m; i++) {
         dfs(heights, pacific, i, 0);
         dfs(heights, atlantic, i, n - 1);
      }
      for (int j = 0; j < n; j++) {
         dfs(heights, pacific, 0, j);
         dfs(heights, atlantic, m - 1, j);
      }

      List<List<Integer>> res = new ArrayList<>();
      for (int i = 0; i < m; i++) {
         for (int j = 0; j < n; j++) {
            if (pacific[i][j] && atlantic[i][j]) {
               res.add(Arrays.asList(i, j));
            }
         }
      }
      return res;
   }

   private void dfs(int[][] h, boolean[][] visited, int i, int j) {
      visited[i][j] = true;
      for (int[] d : dirs) {
         int x = i + d[0], y = j + d[1];
         if (x < 0 || y < 0 || x >= h.length || y >= h[0].length || visited[x][y]) continue;
         if (h[x][y] >= h[i][j]) dfs(h, visited, x, y);
      }
   }
}
```

# 4. Number of Islands

Count the number of islands in a binary grid.

**Java Code:**

```
class Solution {
   public int numIslands(char[][] grid) {
      int count = 0;
      for (int i = 0; i < grid.length; i++) {
         for (int j = 0; j < grid[0].length; j++) {
            if (grid[i][j] == '1') {
               count++;
```

```
                dfs(grid, i, j);
            }
        }
    }
    return count;
  }

  private void dfs(char[][] grid, int i, int j) {
    if (i < 0 || j < 0 || i >= grid.length || j >= grid[0].length || grid[i][j] == '0') return;
    grid[i][j] = '0';
    dfs(grid, i+1, j);
    dfs(grid, i-1, j);
    dfs(grid, i, j+1);
    dfs(grid, i, j-1);
  }
}
```

# 5. Longest Consecutive Sequence

Given an unsorted array, find the length of the longest consecutive elements sequence.

**Java Code:**

```
class Solution {
  public int longestConsecutive(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int n : nums) set.add(n);

    int max = 0;
    for (int num : set) {
      if (!set.contains(num - 1)) {
        int current = num;
        int streak = 1;
        while (set.contains(current + 1)) {
          current++;
          streak++;
        }
        max = Math.max(max, streak);
      }
    }
    return max;
  }
}
```

# 6. Alien Dictionary *(Topological Sort)*

Given a list of words sorted lexicographically according to an alien language, find the order of letters.

**Java Code:**

```
class Solution {
  public String alienOrder(String[] words) {
    Map<Character, Set<Character>> graph = new HashMap<>();
    Map<Character, Integer> indegree = new HashMap<>();

    for (String w : words) {
      for (char c : w.toCharArray()) {
        graph.putIfAbsent(c, new HashSet<>());
        indegree.putIfAbsent(c, 0);
      }
```

```
        }

        for (int i = 0; i < words.length - 1; i++) {
            String w1 = words[i], w2 = words[i+1];
            if (w1.length() > w2.length() && w1.startsWith(w2)) return "";
            for (int j = 0; j < Math.min(w1.length(), w2.length()); j++) {
                if (w1.charAt(j) != w2.charAt(j)) {
                    if (graph.get(w1.charAt(j)).add(w2.charAt(j))) {
                        indegree.put(w2.charAt(j), indegree.get(w2.charAt(j)) + 1);
                    }
                    break;
                }
            }
        }

        Queue<Character> q = new LinkedList<>();
        for (char c : indegree.keySet()) {
            if (indegree.get(c) == 0) q.offer(c);
        }

        StringBuilder sb = new StringBuilder();
        while (!q.isEmpty()) {
            char c = q.poll();
            sb.append(c);
            for (char nei : graph.get(c)) {
                indegree.put(nei, indegree.get(nei) - 1);
                if (indegree.get(nei) == 0) q.offer(nei);
            }
        }

        return sb.length() == indegree.size() ? sb.toString() : "";
    }
}
```

# Graph Valid Tree

**Problem Statement:**

Given n nodes labeled from 0 to n - 1 and a list of edges where edges[i] = [ai, bi] indicates that there is an undirected edge between nodes ai and bi, return true if the edges form a valid tree, and false otherwise.

A valid tree has two properties:

It is fully connected — every node can be reached from any other node.

It contains no cycles.

**Example 1:**

Input:
n = 5
edges = [[0,1],[0,2],[0,3],[1,4]]
Output:
true

**Example 2:**

Input:

n = 5
edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
Output:
false

**Constraints:**

- 1 <= n <= 2000
- 0 <= edges.length <= 2000
- edges[i].length == 2
- 0 <= ai, bi < n
- No duplicate edges

**Approach (Logic Explanation):**

For a graph to be a tree:

- It must have **exactly n - 1 edges** (tree property).
- It must be connected (every node reachable from one node).

If edges.length != n - 1, return false immediately.
Build an adjacency list from the given edges.
Use BFS or DFS to traverse from node 0 and count visited nodes.
If all n nodes are visited and no cycles found, return true.

**Java Code:**

```
import java.util.*;
public class GraphValidTree {
   public boolean validTree(int n, int[][] edges) {
      if (edges.length != n - 1) return false; // Tree must have n-1 edges

      List<List<Integer>> adj = new ArrayList<>();
      for (int i = 0; i < n; i++) adj.add(new ArrayList<>());

      for (int[] e : edges) {
         adj.get(e[0]).add(e[1]);
         adj.get(e[1]).add(e[0]);
      }

      boolean[] visited = new boolean[n];
      dfs(0, -1, adj, visited);

      // Check if all nodes are visited (connected)
      for (boolean v : visited) {
         if (!v) return false;
      }
      return true;
   }

   private void dfs(int node, int parent, List<List<Integer>> adj, boolean[] visited) {
      visited[node] = true;
      for (int nei : adj.get(node)) {
         if (!visited[nei]) {
            dfs(nei, node, adj, visited);
         }
      }
   }
}
```

# Number of Connected Components in an Undirected Graph

**Problem Statement:**
You have n nodes labeled from 0 to n - 1 and a list of edges.
Return the **number of connected components** in the graph.

**Example 1:**

Input:
n = 5
edges = [[0,1],[1,2],[3,4]]
Output:
2
**Example 2:**

Input:
n = 5
edges = [[0,1],[1,2],[2,3],[3,4]]
Output:
1

**Constraints:**

- 1 <= n <= 2000
- 0 <= edges.length <= 2000
- edges[i].length == 2
- 0 <= ai, bi < n

**Approach (Logic Explanation):**

Build an adjacency list from the edges.
Create a visited[] array to track visited nodes.
Iterate over all nodes:
- If a node is unvisited, perform DFS/BFS from that node and increment the component count.

Return the total component count.

**Java Code:**

```
import java.util.*;
public class ConnectedComponents {
   public int countComponents(int n, int[][] edges) {
      List<List<Integer>> adj = new ArrayList<>();
      for (int i = 0; i < n; i++) adj.add(new ArrayList<>());

      for (int[] e : edges) {
         adj.get(e[0]).add(e[1]);
         adj.get(e[1]).add(e[0]);
      }

      boolean[] visited = new boolean[n];
      int count = 0;

      for (int i = 0; i < n; i++) {
         if (!visited[i]) {
            dfs(i, adj, visited);
            count++;
         }
      }
      return count;
```

```
    }

    private void dfs(int node, List<List<Integer>> adj, boolean[] visited) {
        visited[node] = true;
        for (int nei : adj.get(node)) {
            if (!visited[nei]) {
                dfs(nei, adj, visited);
            }
        }
    }
}
```

# 1. Insert Interval

**Problem:**

You are given an array of non-overlapping intervals intervals where intervals[i] = [starti, endi] is sorted in ascending order by starti.

You are also given an interval newInterval = [start, end].

Insert newInterval into intervals so that the list remains sorted and non-overlapping.

Return the resulting array of intervals.

**Example 1:**

Input: intervals = [[1,3],[6,9]], newInterval = [2,5]

Output: [[1,5],[6,9]]

**Example 2:**

Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]

Output: [[1,2],[3,10],[12,16]]

**Constraints:**

- $0 \le$ intervals.length $\le 10^4$
- intervals[i].length == 2
- $0 \le$ starti $\le$ endi $\le 10^5$
- newInterval.length == 2
- $0 \le$ start $\le$ end $\le 10^5$

**Approach:**

Add all intervals that end before newInterval starts.

Merge overlapping intervals with newInterval.
Add the rest of the intervals.

**Code (Java):**

```
public int[][] insert(int[][] intervals, int[] newInterval) {
   List<int[]> result = new ArrayList<>();
   int i = 0;
// Step 1: Add all intervals before newInterval
   while (i < intervals.length && intervals[i][1] < newInterval[0]) {
      result.add(intervals[i]);
      i++;
   }
// Step 2: Merge overlapping intervals
   while (i < intervals.length && intervals[i][0] <= newInterval[1]) {
      newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
      newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
      i++;
   }
   result.add(newInterval);
// Step 3: Add remaining intervals
   while (i < intervals.length) {
      result.add(intervals[i]);
      i++;
   }
return result.toArray(new int[result.size()][]);
}
```

# 2. Merge Intervals

**Problem:**

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals.

**Example:**

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
```

**Constraints:**

- $1 \le intervals.length \le 10^4$
- intervals[i].length == 2
- $0 \le starti \le endi \le 10^4$

**Approach:**

Sort intervals by start time.
Compare current interval's start with last merged interval's end to detect overlap.

**Code (Java):**

```
public int[][] merge(int[][] intervals) {
   if (intervals.length <= 1) return intervals;
Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
   List<int[]> merged = new ArrayList<>();
int[] current = intervals[0];
   merged.add(current);
for (int[] interval : intervals) {
      if (interval[0] <= current[1]) { // Overlap
         current[1] = Math.max(current[1], interval[1]);
      } else {
         current = interval;
         merged.add(current);
```

```
    }
  }
return merged.toArray(new int[merged.size()][]);
}
```

# 3. Non-overlapping Intervals

**Problem:**
Given an array of intervals intervals where intervals[i] = [starti, endi],
return the minimum number of intervals you need to remove to make the rest non-overlapping.
**Example:**

Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
**Approach:**
Sort by end time.
Greedily pick non-overlapping intervals and count removals.
**Code (Java):**

```
public int eraseOverlapIntervals(int[][] intervals) {
  Arrays.sort(intervals, (a, b) -> Integer.compare(a[1], b[1]));
  int count = 0;
  int end = Integer.MIN_VALUE;
for (int[] interval : intervals) {
    if (interval[0] < end) {
      count++; // Remove this one
    } else {
      end = interval[1];
    }
  }
  return count;
}
```

# 4. Repeat and Missing Number

**Problem:**
You are given an unsorted array of n integers containing numbers from 1 to n.
One number is missing and one number is repeated.
Find these two numbers.
**Example:**

Input: nums = [3,1,2,5,3]
Output: [3,4]  // 3 repeated, 4 missing
**Approach:**

- Use mathematical formulas for sum and sum of squares to find missing and repeated numbers.

**Code (Java):**

```
public int[] findErrorNums(int[] nums) {
  long n = nums.length;
  long sum = (n * (n + 1)) / 2;
  long sqSum = (n * (n + 1) * (2 * n + 1)) / 6;
long actualSum = 0, actualSqSum = 0;
  for (int num : nums) {
    actualSum += num;
    actualSqSum += (long) num * num;
  }
```

```
long diff = sum - actualSum; // missing - repeated
   long sqDiff = sqSum - actualSqSum; // (missing^2 - repeated^2)
long sumMR = sqDiff / diff; // missing + repeated
int missing = (int) ((diff + sumMR) / 2);
   int repeated = (int) (sumMR - missing);
return new int[]{repeated, missing};
}
```

# 5. Meeting Rooms (Premium but implemented here)

**Problem:**
Given an array of meeting time intervals intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.
**Example:**

Input: intervals = [[0,30],[5,10],[15,20]]
Output: false
**Approach:**

- Sort by start time and check for overlapping intervals.

**Code (Java):**

```
public boolean canAttendMeetings(int[][] intervals) {
   Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
   for (int i = 1; i < intervals.length; i++) {
      if (intervals[i][0] < intervals[i - 1][1]) return false;
   }
   return true;
}
```

# 6. Meeting Rooms II (Premium but implemented here)

**Problem:**
Given an array of meeting time intervals, return the **minimum number of conference rooms** required.
**Example:**

Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2
**Approach:**

- Use a **min-heap** to keep track of meeting end times.

**Code (Java):**

```
public int minMeetingRooms(int[][] intervals) {
   if (intervals.length == 0) return 0;
Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
   PriorityQueue<Integer> heap = new PriorityQueue<>();
heap.add(intervals[0][1]);
for (int i = 1; i < intervals.length; i++) {
      if (intervals[i][0] >= heap.peek()) {
         heap.poll();
      }
      heap.add(intervals[i][1]);
   }
return heap.size();
}
```

# 1. Reverse a LinkedList

**Logic:**
We iterate through the list, reversing next pointers one by one.
**Java Code:**

```
class ReverseLinkedList {
   static class ListNode {
      int val;
      ListNode next;
      ListNode(int val) { this.val = val; }
   }
public ListNode reverseList(ListNode head) {
      ListNode prev = null, curr = head;
      while (curr != null) {
         ListNode nextNode = curr.next;
         curr.next = prev;
         prev = curr;
         curr = nextNode;
      }
      return prev;
   }
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)

# 2. Detect a Cycle in Linked List

**Logic (Floyd's Cycle Detection):**
Use two pointers (slow, fast). If they meet → cycle exists.
**Java Code:**

```
class DetectCycle {
   static class ListNode {
      int val;
      ListNode next;
      ListNode(int val) { this.val = val; }
   }
public boolean hasCycle(ListNode head) {
      ListNode slow = head, fast = head;
      while (fast != null && fast.next != null) {
         slow = slow.next;
         fast = fast.next.next;
         if (slow == fast) return true;
      }
      return false;
   }
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)

## 3. Merge Two Sorted Linked Lists (Merge Sort method)

**Logic:**

Use two pointers, pick smaller node each time.

**Java Code:**

```
class MergeSortedLists {
   static class ListNode {
      int val;
      ListNode next;
      ListNode(int val) { this.val = val; }
   }
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
      ListNode dummy = new ListNode(-1);
      ListNode tail = dummy;
while (l1 != null && l2 != null) {
         if (l1.val < l2.val) {
            tail.next = l1;
            l1 = l1.next;
         } else {
            tail.next = l2;
            l2 = l2.next;
         }
         tail = tail.next;
      }
tail.next = (l1 != null) ? l1 : l2;
      return dummy.next;
   }
}
```

**Time Complexity:** O(m + n)

**Space Complexity:** O(1)

## 4. Merge K Sorted Linked Lists

**Logic:**

Use a **min-heap** to always take the smallest node among the heads.

**Java Code:**

```
import java.util.*;
class MergeKSortedLists {
   static class ListNode {
      int val;
      ListNode next;
      ListNode(int val) { this.val = val; }
   }
public ListNode mergeKLists(ListNode[] lists) {
      PriorityQueue<ListNode> pq = new PriorityQueue<>((a, b) -> a.val - b.val);
      for (ListNode node : lists) {
         if (node != null) pq.add(node);
      }
ListNode dummy = new ListNode(-1), tail = dummy;
while (!pq.isEmpty()) {
         ListNode node = pq.poll();
         tail.next = node;
         tail = node;
         if (node.next != null) pq.add(node.next);
      }
```

```
return dummy.next;
   }
}
```
**Time Complexity:** O(N log k) (N = total nodes, k = lists)
**Space Complexity:** O(k)

## 5. Remove N-th Node from End of Linked List

**Logic:**
Two-pointer method:
- Move fast N steps ahead.
- Move both until fast reaches end.
- Delete the node.

**Java Code:**

```
class RemoveNthNode {
   static class ListNode {
      int val;
      ListNode next;
      ListNode(int val) { this.val = val; }
   }
public ListNode removeNthFromEnd(ListNode head, int n) {
      ListNode dummy = new ListNode(-1);
      dummy.next = head;
      ListNode fast = dummy, slow = dummy;
for (int i = 0; i <= n; i++) {
         fast = fast.next;
      }
while (fast != null) {
         fast = fast.next;
         slow = slow.next;
      }
slow.next = slow.next.next;
      return dummy.next;
   }
}
```
**Time Complexity:** O(n)
**Space Complexity:** O(1)

## 6. Reorder List

**Logic:**
- Find middle.
- Reverse 2nd half.
- Merge two halves alternately.

**Java Code:**

```
class ReorderList {
   static class ListNode {
      int val;
      ListNode next;
      ListNode(int val) { this.val = val; }
   }
public void reorderList(ListNode head) {
      if (head == null || head.next == null) return;
// Step 1: Find middle
      ListNode slow = head, fast = head;
      while (fast.next != null && fast.next.next != null) {
```

```
            slow = slow.next;
            fast = fast.next.next;
        }
    // Step 2: Reverse second half
        ListNode prev = null, curr = slow.next;
        slow.next = null;
        while (curr != null) {
            ListNode nextNode = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextNode;
        }
    // Step 3: Merge
        ListNode first = head, second = prev;
        while (second != null) {
            ListNode temp1 = first.next;
            ListNode temp2 = second.next;
            first.next = second;
            second.next = temp1;
            first = temp1;
            second = temp2;
        }
    }
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)

Alright — here's your **Day 7: Matrix** prep breakdown with explanations, logic, and Java implementations for all 4 problems.

# Day 7 – Matrix Problems

## 1. Set Matrix Zeros

**Problem:**
Given an m x n matrix, if an element is 0, set its entire row and column to 0.

**Logic Explanation:**

- Naive approach: Create extra arrays to mark rows & cols to be zeroed → O(m+n) space.
- Optimized approach: Use the first row & first column as markers.
    - Step 1: Mark first row/col separately if they contain 0.
    - Step 2: For each matrix[i][j] == 0, mark matrix[i][0] and matrix[0][j].
    - Step 3: Iterate again and set rows & cols to 0 based on markers.

**Java Code:**

```
class Solution {
    public void setZeroes(int[][] matrix) {
        int m = matrix.length, n = matrix[0].length;
        boolean firstRow = false, firstCol = false;
    for (int i = 0; i < m; i++)
            if (matrix[i][0] == 0) firstCol = true;
    for (int j = 0; j < n; j++)
            if (matrix[0][j] == 0) firstRow = true;
    for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
```

```
            if (matrix[i][j] == 0) {
                matrix[i][0] = 0;
                matrix[0][j] = 0;
            }
        }
    }
for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                matrix[i][j] = 0;
            }
        }
    }
if (firstCol) for (int i = 0; i < m; i++) matrix[i][0] = 0;
    if (firstRow) for (int j = 0; j < n; j++) matrix[0][j] = 0;
  }
}
```

**Time Complexity:** O(m×n)
**Space Complexity:** O(1)

## 2. Print the Matrix in Spiral Manner

**Problem:**
Given a matrix, return all elements in spiral order.
**Logic Explanation:**

- Keep four pointers: top, bottom, left, right.
- Traverse in four directions: Left→Right, Top→Bottom, Right→Left, Bottom→Top.
- After each pass, shrink the boundary.

**Java Code:**

```
class Solution {
  public List<Integer> spiralOrder(int[][] matrix) {
    List<Integer> result = new ArrayList<>();
    int top = 0, bottom = matrix.length - 1;
    int left = 0, right = matrix[0].length - 1;
while (top <= bottom && left <= right) {
        for (int j = left; j <= right; j++)
            result.add(matrix[top][j]);
        top++;
for (int i = top; i <= bottom; i++)
            result.add(matrix[i][right]);
        right--;
if (top <= bottom) {
            for (int j = right; j >= left; j--)
                result.add(matrix[bottom][j]);
            bottom--;
        }
if (left <= right) {
            for (int i = bottom; i >= top; i--)
                result.add(matrix[i][left]);
            left++;
        }
    }
    return result;
  }
}
```

**Time Complexity:** O(m×n)
**Space Complexity:** O(1) (excluding result list)

## 3. Rotate Matrix by 90 Degrees

**Problem:**
Rotate an n x n matrix by 90° clockwise.
**Logic Explanation:**
- Step 1: Transpose the matrix (swap matrix[i][j] with matrix[j][i]).
- Step 2: Reverse each row.

**Java Code:**
```
class Solution {
   public void rotate(int[][] matrix) {
      int n = matrix.length;

      // Transpose
      for (int i = 0; i < n; i++) {
         for (int j = i; j < n; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
         }
      }
// Reverse each row
      for (int i = 0; i < n; i++) {
         int left = 0, right = n - 1;
         while (left < right) {
            int temp = matrix[i][left];
            matrix[i][left] = matrix[i][right];
            matrix[i][right] = temp;
            left++;
            right--;
         }
      }
   }
}
```
**Time Complexity:** $O(n^2)$
**Space Complexity:** O(1)

# 4. Word Search
**Problem:**
Given a board of characters and a word, check if the word exists in the board.
You can move up, down, left, or right — but cannot reuse the same cell.
**Logic Explanation:**
- DFS backtracking approach:
  - Start DFS from every cell matching first letter.
  - Recursively explore neighbors.
  - Mark cell as visited temporarily.
  - Backtrack after exploring.

**Java Code:**
```
class Solution {
   public boolean exist(char[][] board, String word) {
      int m = board.length, n = board[0].length;
      for (int i = 0; i < m; i++) {
         for (int j = 0; j < n; 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 || j < 0 || i >= board.length || j >= board[0].length ||
            board[i][j] != word.charAt(index)) return false;
    char temp = board[i][j];
        board[i][j] = '#'; // mark 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;
      }
    }
```

**Time Complexity:** O(m × n × 4^L) (L = word length)
**Space Complexity:** O(L) (recursion stack)

Got it — you want **Day 8 String** with **full question statement + optimal answer + Java code + explanation** for *all problems* in that list.
I'll prepare it in the exact **LeetCode-style** format so you can directly practice.

# Day 8 – String Problems with Solutions

## 1. Longest Substring Without Repeating Characters – *Medium*

**Question:**
Given a string s, find the length of the longest substring without repeating characters.
**Example:**
Input: s = "abcabcbb"
Output: 3
Explanation: "abc" is the longest substring.
**Approach:**

- Use sliding window + HashSet.
- Expand right pointer, if char is duplicate → shrink from left until removed.
- Track max length.

**Java Solution:**

```
public int lengthOfLongestSubstring(String s) {
  Set<Character> set = new HashSet<>();
  int left = 0, maxLen = 0;
  for (int right = 0; right < s.length(); right++) {
    while (set.contains(s.charAt(right))) {
      set.remove(s.charAt(left));
      left++;
    }
    set.add(s.charAt(right));
```

```
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}
```
**Time Complexity:** O(n)
**Space Complexity:** O(min(n, charset size))

## 2. Longest Repeating Character Replacement – *Medium*

**Question:**
Given a string s and integer k, return the length of the longest substring that can be obtained by replacing at most k characters so all characters are the same.
**Example:**
Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace one 'B' → "AAAA".

**Approach:**
- Keep count of each char in current window.
- Track max frequency char count.
- If (window length - maxFreq) > k, shrink from left.

**Java Solution:**
```
public int characterReplacement(String s, int k) {
    int[] count = new int[26];
    int left = 0, maxCount = 0, result = 0;
    for (int right = 0; right < s.length(); right++) {
        maxCount = Math.max(maxCount, ++count[s.charAt(right) - 'A']);
        while ((right - left + 1) - maxCount > k) {
            count[s.charAt(left) - 'A']--;
            left++;
        }
        result = Math.max(result, right - left + 1);
    }
    return result;
}
```
**Time Complexity:** O(n)
**Space Complexity:** O(1)

## 3. Minimum Window Substring – *Hard*

**Question:**
Given strings s and t, return the minimum window in s that contains all characters of t. If no such window, return "".
**Example:**
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
**Approach:**
- Maintain two frequency maps: target & window.
- Expand right until all chars covered, then shrink from left to minimize.

**Java Solution:**
```
public String minWindow(String s, String t) {
    if (t.length() > s.length()) return "";
    Map<Character, Integer> target = new HashMap<>();
    for (char c : t.toCharArray()) target.put(c, target.getOrDefault(c, 0) + 1);
int left = 0, matchCount = 0, minLen = Integer.MAX_VALUE, startIdx = 0;
    Map<Character, Integer> window = new HashMap<>();
for (int right = 0; right < s.length(); right++) {
        char ch = s.charAt(right);
        window.put(ch, window.getOrDefault(ch, 0) + 1);
        if (target.containsKey(ch) && window.get(ch).intValue() == target.get(ch).intValue())
```

```
            matchCount++;
    while (matchCount == target.size()) {
            if (right - left + 1 < minLen) {
               minLen = right - left + 1;
               startIdx = left;
            }
            char leftChar = s.charAt(left);
            window.put(leftChar, window.get(leftChar) - 1);
            if (target.containsKey(leftChar) && window.get(leftChar) < target.get(leftChar))
               matchCount--;
            left++;
         }
      }
      return minLen == Integer.MAX_VALUE ? "" : s.substring(startIdx, startIdx + minLen);
    }
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)

## 4. Check for Anagrams – *Easy*

**Question:**
Given two strings, return true if they are anagrams, false otherwise.
**Example:**
Input: s = "anagram", t = "nagaram"
Output: true
**Java Solution:**

```
public boolean isAnagram(String s, String t) {
   if (s.length() != t.length()) return false;
   int[] count = new int[26];
   for (char c : s.toCharArray()) count[c - 'a']++;
   for (char c : t.toCharArray()) {
      count[c - 'a']--;
      if (count[c - 'a'] < 0) return false;
   }
   return true;
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)

## 5. Group Anagrams – *Medium*

**Question:**
Group strings that are anagrams of each other.
**Example:**
Input: ["eat","tea","tan","ate","nat","bat"]
Output: [["eat","tea","ate"],["tan","nat"],["bat"]]
**Java Solution:**

```
public List<List<String>> groupAnagrams(String[] strs) {
   Map<String, List<String>> map = new HashMap<>();
   for (String s : strs) {
      char[] arr = s.toCharArray();
      Arrays.sort(arr);
      String key = new String(arr);
      map.computeIfAbsent(key, k -> new ArrayList<>()).add(s);
   }
   return new ArrayList<>(map.values());
}
```

**Time Complexity:** O(n * k log k)
**Space Complexity:** O(n * k)

## 6. Check Balanced Parentheses – *Medium*

**Question:**
Given a string of brackets, return true if balanced, else false.
**Example:**
Input: "({[]})"
Output: true
**Java Solution:**

```
public boolean isValid(String s) {
   Stack<Character> stack = new Stack<>();
   Map<Character, Character> map = Map.of(')', '(', ']', '[', '}', '{');
   for (char c : s.toCharArray()) {
      if (map.containsValue(c)) stack.push(c);
      else if (map.containsKey(c)) {
         if (stack.isEmpty() || stack.pop() != map.get(c)) return false;
      }
   }
   return stack.isEmpty();
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(n)

## 7. Check Palindrome – *Easy*

**Question:**
Given a string, check if it is palindrome ignoring non-alphanumeric.
**Example:**
Input: "A man, a plan, a canal: Panama"
Output: true
**Java Solution:**

```
public boolean isPalindrome(String s) {
   int left = 0, right = s.length() - 1;
   while (left < right) {
      while (left < right && !Character.isLetterOrDigit(s.charAt(left))) left++;
      while (left < right && !Character.isLetterOrDigit(s.charAt(right))) right--;
      if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right)))
         return false;
      left++;
      right--;
   }
   return true;
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)

## 8. Longest Palindromic Substring – *Hard*

**Java Solution:**

```
public String longestPalindrome(String s) {
   int start = 0, end = 0;
   for (int i = 0; i < s.length(); i++) {
      int len1 = expand(s, i, i);
      int len2 = expand(s, i, i + 1);
      int len = Math.max(len1, len2);
      if (len > end - start) {
         start = i - (len - 1) / 2;
         end = i + len / 2;
      }
   }
   return s.substring(start, end + 1);
}
```

```
private int expand(String s, int left, int right) {
   while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
      left--;
      right++;
   }
   return right - left - 1;
}
```

## 9. Palindromic Substrings – *Medium*

**Java Solution:**

```
public int countSubstrings(String s) {
   int count = 0;
   for (int i = 0; i < s.length(); i++) {
      count += expandCount(s, i, i);
      count += expandCount(s, i, i + 1);
   }
   return count;
}
private int expandCount(String s, int left, int right) {
   int cnt = 0;
   while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
      cnt++;
      left--;
      right++;
   }
   return cnt;
}
```

## 10. Encode and Decode Strings – *Medium* (LeetCode Premium)

**Java Solution:**

```
public class Codec {
   public String encode(List<String> strs) {
      StringBuilder sb = new StringBuilder();
      for (String str : strs) {
         sb.append(str.length()).append('#').append(str);
      }
      return sb.toString();
   }
public List<String> decode(String s) {
      List<String> result = new ArrayList<>();
      int i = 0;
      while (i < s.length()) {
         int j = i;
         while (s.charAt(j) != '#') j++;
         int length = Integer.parseInt(s.substring(i, j));
         i = j + 1;
         result.add(s.substring(i, i + length));
         i += length;
      }
      return result;
   }
}
```

# Day 9 – Tree Problems

## 1. Height of a Binary Tree

**Question:**
Given a binary tree, return its height (maximum depth).
**Logic:**

- Height = 1 + max(height(left), height(right))
- Base case: if root is null, return 0.

**Java Code:**

```
class TreeNode {
   int val;
   TreeNode left, right;
   TreeNode(int val) { this.val = val; }
}
public int maxDepth(TreeNode root) {
   if (root == null) return 0;
   return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(h) (h = height of tree due to recursion stack)

## 2. Check if Two Trees are Identical

**Question:**
Given two binary trees, check if they are identical.
**Logic:**

- Both null → true
- One null → false
- Both non-null → compare values, then recurse left & right.

**Java Code:**

```
public boolean isSameTree(TreeNode p, TreeNode q) {
   if (p == null && q == null) return true;
   if (p == null || q == null) return false;
   return (p.val == q.val) &&
        isSameTree(p.left, q.left) &&
        isSameTree(p.right, q.right);
}
```

## 3. Invert/Flip Binary Tree

**Question:**
Invert a binary tree (mirror image).
**Logic:**

- Swap left and right recursively.

**Java Code:**

```
public TreeNode invertTree(TreeNode root) {
   if (root == null) return null;
   TreeNode temp = root.left;
   root.left = invertTree(root.right);
   root.right = invertTree(temp);
   return root;
}
```

## 4. Maximum Path Sum in Binary Tree

**Question:**
Return the maximum sum of any path (node → node).
**Logic:**

- Use DFS, keep a global max, at each node calculate: node.val + max(0, left) + max(0, right)
- Return node.val + max(left, right) for recursion.

**Java Code:**

```
int maxSum = Integer.MIN_VALUE;
public int maxPathSum(TreeNode root) {
   maxGain(root);
   return maxSum;
}
private int maxGain(TreeNode node) {
   if (node == null) return 0;
   int left = Math.max(maxGain(node.left), 0);
   int right = Math.max(maxGain(node.right), 0);
   maxSum = Math.max(maxSum, node.val + left + right);
   return node.val + Math.max(left, right);
}
```

## 5. Level Order Traversal

**Question:**
Return level-wise traversal of a binary tree.
**Logic:**

- Use BFS with a queue.

**Java Code:**

```
public List<List<Integer>> levelOrder(TreeNode root) {
   List<List<Integer>> res = new ArrayList<>();
   if (root == null) return res;
   Queue<TreeNode> q = new LinkedList<>();
   q.add(root);
   while (!q.isEmpty()) {
      int size = q.size();
      List<Integer> level = new ArrayList<>();
      for (int i = 0; i < size; i++) {
         TreeNode node = q.poll();
         level.add(node.val);
         if (node.left != null) q.add(node.left);
         if (node.right != null) q.add(node.right);
      }
      res.add(level);
   }
   return res;
}
```

## 6. Serialize and Deserialize Binary Tree

**Question:**
Serialize a tree to a string, then deserialize back.
**Logic:**

- Preorder + "null" markers.

**Java Code:**

```
public class Codec {
   public String serialize(TreeNode root) {
      if (root == null) return "null,";
      return root.val + "," + serialize(root.left) + serialize(root.right);
   }
int index = 0;
   public TreeNode deserialize(String data) {
      String[] nodes = data.split(",");
      return build(nodes);
   }
private TreeNode build(String[] nodes) {
      if (nodes[index].equals("null")) {
         index++;
         return null;
      }
      TreeNode root = new TreeNode(Integer.parseInt(nodes[index++]));
      root.left = build(nodes);
      root.right = build(nodes);
      return root;
   }
}
```

## 7. Subtree of Another Tree

**Question:**

Check if tree B is a subtree of tree A.

**Logic:**

- If roots are same → check identical.
- Else → check left and right.

**Java Code:**

```
public boolean isSubtree(TreeNode root, TreeNode subRoot) {
   if (root == null) return false;
   if (isSameTree(root, subRoot)) return true;
   return isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);
}
```

## 8. Construct Tree from Inorder & Postorder

**Logic:**

- Last element of postorder → root.
- Find root in inorder to split left/right.
- Recurse.

**Java Code:**

```
Map<Integer, Integer> inorderMap;
int postIndex;
public TreeNode buildTree(int[] inorder, int[] postorder) {
   inorderMap = new HashMap<>();
   for (int i = 0; i < inorder.length; i++) inorderMap.put(inorder[i], i);
   postIndex = postorder.length - 1;
   return build(postorder, 0, inorder.length - 1);
}
private TreeNode build(int[] postorder, int inStart, int inEnd) {
   if (inStart > inEnd) return null;
   TreeNode root = new TreeNode(postorder[postIndex--]);
   int inIndex = inorderMap.get(root.val);
   root.right = build(postorder, inIndex + 1, inEnd);
```

```
    root.left = build(postorder, inStart, inIndex - 1);
    return root;
}
```

## 9. Check if a Tree is BST

**Logic:**

- Inorder should be sorted, or
- Recursively check min/max constraints.

**Java Code:**

```
public boolean isValidBST(TreeNode root) {
    return isValid(root, null, null);
}
private boolean isValid(TreeNode node, Integer min, Integer max) {
    if (node == null) return true;
    if ((min != null && node.val <= min) || (max != null && node.val >= max)) return false;
    return isValid(node.left, min, node.val) && isValid(node.right, node.val, max);
}
```

## 10. K-th Smallest in BST

**Logic:**

- Inorder traversal → sorted order → get k-th.

**Java Code:**

```
int count = 0, ans = 0;
public int kthSmallest(TreeNode root, int k) {
    inorder(root, k);
    return ans;
}
private void inorder(TreeNode node, int k) {
    if (node == null) return;
    inorder(node.left, k);
    count++;
    if (count == k) {
        ans = node.val;
        return;
    }
    inorder(node.right, k);
}
```

## 11. LCA of Two Nodes in BST

**Logic:**

- If both < root → go left.
- If both > root → go right.
- Else root is LCA.

**Java Code:**

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (p.val < root.val && q.val < root.val)
        return lowestCommonAncestor(root.left, p, q);
    if (p.val > root.val && q.val > root.val)
        return lowestCommonAncestor(root.right, p, q);
    return root;
}
```

## 12. Implement Trie

**Logic:**

- Each node has array[26] children + isEnd flag.
- Insert, search, startsWith.

**Java Code:**

```
class TrieNode {
   TrieNode[] children = new TrieNode[26];
   boolean isEnd;
}
class Trie {
   TrieNode root;
   public Trie() { root = new TrieNode(); }
public void insert(String word) {
      TrieNode node = root;
      for (char c : word.toCharArray()) {
         int idx = c - 'a';
         if (node.children[idx] == null)
            node.children[idx] = new TrieNode();
         node = node.children[idx];
      }
      node.isEnd = true;
   }
public boolean search(String word) {
      TrieNode node = root;
      for (char c : word.toCharArray()) {
         int idx = c - 'a';
         if (node.children[idx] == null) return false;
         node = node.children[idx];
      }
      return node.isEnd;
   }
public boolean startsWith(String prefix) {
      TrieNode node = root;
      for (char c : prefix.toCharArray()) {
         int idx = c - 'a';
         if (node.children[idx] == null) return false;
         node = node.children[idx];
      }
      return true;
   }
}
```

## 13. Implement Trie – 2 (Word Count)

**Logic:**

- Extend Trie to store countPrefix and countWord.
- Useful for counting occurrences.

**Java Code:**

```
class TrieNode2 {
   TrieNode2[] children = new TrieNode2[26];
   int countPrefix = 0;
   int countWord = 0;
}
class Trie2 {
   TrieNode2 root;
```

```
    public Trie2() { root = new TrieNode2(); }
  public void insert(String word) {
        TrieNode2 node = root;
        for (char c : word.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null)
                node.children[idx] = new TrieNode2();
            node = node.children[idx];
            node.countPrefix++;
        }
        node.countWord++;
    }
  public int countWordsEqualTo(String word) {
        TrieNode2 node = root;
        for (char c : word.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null) return 0;
            node = node.children[idx];
        }
        return node.countWord;
    }
  public int countWordsStartingWith(String prefix) {
        TrieNode2 node = root;
        for (char c : prefix.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null) return 0;
            node = node.children[idx];
        }
        return node.countPrefix;
    }
  public void erase(String word) {
        TrieNode2 node = root;
        for (char c : word.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null) return;
            node = node.children[idx];
            node.countPrefix--;
        }
        node.countWord--;
    }
  }
```

Alright — here's **Day 10 (Heap)** with **full questions and Java solutions**.

# 1. K Most Frequent Elements – *Medium*

**Problem:**
Given an integer array nums and an integer k, return the k most frequent elements.
You may return the answer in any order.
**Example:**
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
**Solution (Java using PriorityQueue + HashMap):**

```
import java.util.*;
public class KMostFrequentElements {
   public static int[] topKFrequent(int[] nums, int k) {
      Map<Integer, Integer> freqMap = new HashMap<>();
      for (int num : nums) {
         freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
      }
PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[1] - b[1]); // Min heap
      for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {
         pq.offer(new int[]{entry.getKey(), entry.getValue()});
         if (pq.size() > k) pq.poll();
      }
int[] result = new int[k];
      for (int i = k - 1; i >= 0; i--) {
         result[i] = pq.poll()[0];
      }
      return result;
   }
public static void main(String[] args) {
      int[] nums = {1,1,1,2,2,3};
      int k = 2;
      System.out.println(Arrays.toString(topKFrequent(nums, k)));
```

```
    }
}
```
**Logic:**

- Count frequencies using HashMap.
- Use a **min-heap** to keep the top k frequent elements.
- Pop from heap when size exceeds k.
- Extract elements from heap.

**Time Complexity:** O(n log k)
**Space Complexity:** O(n)

# 2. Find Median from Data Stream – *Hard*

**Problem:**
Design a data structure that supports adding numbers and finding the median of all elements so far.
**Example:**
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
**Solution (Java using Two Heaps):**

```
import java.util.*;
class MedianFinder {
   PriorityQueue<Integer> maxHeap; // lower half
   PriorityQueue<Integer> minHeap; // upper half
public MedianFinder() {
      maxHeap = new PriorityQueue<>(Collections.reverseOrder());
      minHeap = new PriorityQueue<>();
   }
public void addNum(int num) {
      maxHeap.offer(num);
      minHeap.offer(maxHeap.poll());
if (minHeap.size() > maxHeap.size()) {
         maxHeap.offer(minHeap.poll());
      }
   }
public double findMedian() {
      if (maxHeap.size() == minHeap.size()) {
         return (maxHeap.peek() + minHeap.peek()) / 2.0;
      } else {
         return maxHeap.peek();
      }
   }
public static void main(String[] args) {
      MedianFinder mf = new MedianFinder();
      mf.addNum(1);
      mf.addNum(2);
      System.out.println(mf.findMedian()); // 1.5
      mf.addNum(3);
      System.out.println(mf.findMedian()); // 2
   }
}
```

**Logic:**

- Maintain **two heaps**:
  - Max-heap for the smaller half of numbers.
  - Min-heap for the larger half.
- Balance the heaps so size difference is at most 1.
- Median is either the root of one heap or average of both.

**Time Complexity:** O(log n) for add, O(1) for findMedian.

**Space Complexity:** O(n)

# Advanced Medium & Hard Heap & Graph Problems

## 1. Convert Max Heap to Min Heap *(Medium)*

**Problem:** Convert a given max-heap into a min-heap **in-place** (no extra space).

**Java Solution:**

```
class ConvertHeap {
   public void convertToMinHeap(int[] heap) {
      int n = heap.length;
      for (int i = n / 2 - 1; i >= 0; i--) {
         heapify(heap, n, i);
      }
   }
private void heapify(int[] heap, int n, int i) {
      int smallest = i;
      int left = 2 * i + 1, right = 2 * i + 2;
      if (left < n && heap[left] < heap[smallest]) smallest = left;
      if (right < n && heap[right] < heap[smallest]) smallest = right;
      if (smallest != i) {
         int temp = heap[i];
         heap[i] = heap[smallest];
         heap[smallest] = temp;
         heapify(heap, n, smallest);
      }
   }
}
```

## 2. Merge K Sorted Arrays *(Hard)*

**Problem:** Given k sorted arrays, merge them into one large sorted array.

**Java Solution:**

```
import java.util.*;
class MergeKArrays {
   static class Node {
      int val, row, col;
      Node(int v, int r, int c) { val = v; row = r; col = c; }
   }
public List<Integer> mergeKArrays(int[][] arrays) {
      PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a.val));
      List<Integer> result = new ArrayList<>();
for (int i = 0; i < arrays.length; i++) {
         if (arrays[i].length > 0)
            pq.offer(new Node(arrays[i][0], i, 0));
      }
while (!pq.isEmpty()) {
         Node n = pq.poll();
         result.add(n.val);
         if (n.col + 1 < arrays[n.row].length) {
            pq.offer(new Node(
               arrays[n.row][n.col + 1],
               n.row,
               n.col + 1
            ));
         }
      }
      return result;
   }
```

```
}
```

## 3. Sliding Window Maximum *(Hard)*

**Problem:** Given an array nums and window size k, return the maximum value for each sliding window.

**Java Solution (Deque Optimized):**

```
import java.util.*;
class SlidingWindowMax {
   public int[] maxSlidingWindow(int[] nums, int k) {
      int n = nums.length;
      if (n * k == 0) return new int[0];
      int[] res = new int[n - k + 1];
      Deque<Integer> deque = new ArrayDeque<>();
for (int i = 0, r = 0; i < n; i++) {
         while (!deque.isEmpty() && nums[i] > nums[deque.peekLast()])
            deque.pollLast();
         deque.offerLast(i);
         if (deque.peekFirst() <= i - k)
            deque.pollFirst();
         if (i >= k - 1)
            res[r++] = nums[deque.peekFirst()];
      }
      return res;
   }
}
```

## 4. Dijkstra's Shortest Path Algorithm *(Hard)*

**Problem:** Compute shortest paths from a source node to **all other nodes** in a weighted graph with non-negative weights.

**Java Solution (Priority Queue):**

```
import java.util.*;
class Dijkstra {
   static class Edge {
      int node, weight;
      Edge(int n, int w) { node = n; weight = w; }
   }
public int[] dijkstra(List<List<Edge>> graph, int src) {
      int n = graph.size();
      int[] dist = new int[n];
      Arrays.fill(dist, Integer.MAX_VALUE);
      dist[src] = 0;
PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(e -> e.weight));
      pq.offer(new Edge(src, 0));
while (!pq.isEmpty()) {
         Edge curr = pq.poll();
         for (Edge nei : graph.get(curr.node)) {
            int ndist = curr.weight + nei.weight;
            if (ndist < dist[nei.node]) {
               dist[nei.node] = ndist;
               pq.offer(new Edge(nei.node, ndist));
            }
         }
      }
      return dist;
   }
}
```

# Advanced Heap & Graph Problems

## 1. K Maximum Sum Combinations from Two Arrays *(Medium–Hard)*

**Problem:**

Given two integer arrays A and B of the same length n, and an integer k, find the top k maximum sums each formed by adding one element from A and one from B.

**Example:**

A = [3,2]; B = [1,4]; k = 2 → Output: [7,6]

Explanation: Max sums are 3+4=7 and 2+4=6.

**Approach:**

- Sort both arrays descending.
- Use a max-heap to track sums with index pairs.
- Use a set to avoid revisiting pairs.

**Java Code:**

```
import java.util.*;
class PairSum {
   public List<Integer> maxKSum(int[] A, int[] B, int k) {
      int n = A.length;
      Arrays.sort(A);
      Arrays.sort(B);
      PriorityQueue<int[]> pq = new PriorityQueue<>((x, y) -> y[0] - x[0]);
      Set<String> visited = new HashSet<>();
      List<Integer> res = new ArrayList<>();
pq.offer(new int[]{A[n - 1] + B[n - 1], n - 1, n - 1});
      visited.add((n - 1) + "-" + (n - 1));
while (k-- > 0 && !pq.isEmpty()) {
         int[] top = pq.poll();
         res.add(top[0]);
         int i = top[1], j = top[2];
if (i - 1 >= 0 && visited.add((i - 1) + "-" + j)) {
            pq.offer(new int[]{A[i - 1] + B[j], i - 1, j});
         }
         if (j - 1 >= 0 && visited.add(i + "-" + (j - 1))) {
            pq.offer(new int[]{A[i] + B[j - 1], i, j - 1});
         }
      }
      return res;
   }
}
```

Useful for top-k sum-combination optimizations.
(GeeksforGeeks)

## 2. Minimum Cost Path in Grid Using Dijkstra *(Hard)*

**Problem:**
Given an n×n grid with non-negative costs, find the minimum path cost from the top-left to bottom-right, moving in four directions (up/down/left/right).

**Approach:**

- Treat the grid as a graph.
- Use Dijkstra's algorithm with a min-heap storing [cost, x, y].

**Java Code:**

```
import java.util.*;
class MinCostPath {
   static class Cell { int cost, x, y; Cell(int c, int x, int y) { this.cost = c; this.x = x; this.y = y; } }
public int minCostPath(int[][] grid) {
      int n = grid.length;
      int[][] dist = new int[n][n];
      for (int[] row : dist) Arrays.fill(row, Integer.MAX_VALUE);
dist[0][0] = grid[0][0];
      PriorityQueue<Cell> pq = new PriorityQueue<>(Comparator.comparingInt(c -> c.cost));
      pq.offer(new Cell(dist[0][0], 0, 0));
int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};
      while (!pq.isEmpty()) {
         Cell cur = pq.poll();
         if (cur.cost > dist[cur.x][cur.y]) continue;
         for (int[] d : dirs) {
            int nx = cur.x + d[0], ny = cur.y + d[1];
            if (nx >= 0 && ny >= 0 && nx < n && ny < n) {
               int newCost = cur.cost + grid[nx][ny];
               if (newCost < dist[nx][ny]) {
                  dist[nx][ny] = newCost;
                  pq.offer(new Cell(newCost, nx, ny));
               }
            }
         }
      }
      return dist[n - 1][n - 1];
   }
}
```

An application of shortest-path on grids.
(GeeksforGeeks)

## 3. Connect N Ropes with Minimum Cost *(Medium)*

**Problem:**
Given n ropes with various lengths, connect them into one rope with minimal cost. The cost to combine two ropes is the sum of their lengths.

**Approach:**
Repeatedly combine the two shortest ropes using a min-heap.

**Java Code:**

```
import java.util.*;
class ConnectRopes {
   public int minCost(int[] ropes) {
      PriorityQueue<Integer> pq = new PriorityQueue<>();
      for (int r : ropes) pq.offer(r);
      int total = 0;
      while (pq.size() > 1) {
         int a = pq.poll(), b = pq.poll();
         total += a + b;
```

```
            pq.offer(a + b);
        }
        return total;
    }
}
```

## 4. Smallest Range in K Lists *(Hard)*

**Problem:**

Given k sorted lists, find the smallest range that includes at least one number from each list.

**Approach:**

- Use a min-heap to hold the current window of elements (one from each list).
- Track current maximum and minimize range.

**Java Code:**

```
import java.util.*;
class SmallestRange {
    static class Node { int val, listIdx, idx; Node(int v, int l, int i) { val = v; listIdx = l; idx = i; } }
public int[] smallestRange(List<List<Integer>> nums) {
        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.val));
        int max = Integer.MIN_VALUE, start = 0, end = Integer.MAX_VALUE;
for (int i = 0; i < nums.size(); i++) {
            pq.offer(new Node(nums.get(i).get(0), i, 0));
            max = Math.max(max, nums.get(i).get(0));
        }
while (pq.size() == nums.size()) {
            Node min = pq.poll();
            if (max - min.val < end - start) {
                start = min.val;
                end = max;
            }
            if (min.idx + 1 < nums.get(min.listIdx).size()) {
                int nextVal = nums.get(min.listIdx).get(min.idx + 1);
                pq.offer(new Node(nextVal, min.listIdx, min.idx + 1));
                max = Math.max(max, nextVal);
            } else {
                break;
            }
        }
        return new int[]{start, end};
    }
}
```