Top 50 DSA Interview Questions with Java Solutions

◇ Array (5 Questions)

🔹 1. Two Sum

📝 Problem:

Find indices of the two numbers in an array that add up to a target.

💡 Brute Force – O(n²)

```
java
CopyEdit
public int[] twoSumBrute(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] == target) {
                return new int[]{i, j};
            }
        }
    }
    return new int[]{-1, -1};
}
```

⚡ Optimized – O(n) using HashMap

```
java
CopyEdit
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[]{-1, -1};
}
```

🔹 2. Maximum Subarray (Kadane's Algorithm)

📝 Problem:

Find the contiguous subarray with the largest sum.

💡 Brute Force – O(n²)

```
java
CopyEdit
public int maxSubArrayBrute(int[] nums) {
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < nums.length; i++) {
        int sum = 0;
        for (int j = i; j < nums.length; j++) {
            sum += nums[j];
            max = Math.max(max, sum);
        }
    }
    return max;
}
```

⚡ Optimized – O(n)

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

🔹 3. Move Zeroes

📝 Problem:

Move all 0's to the end while maintaining the relative order of non-zero elements.

💡 Brute Force (Extra Array) – O(n)

```
java
CopyEdit
public void moveZeroesBrute(int[] nums) {
    int[] temp = new int[nums.length];
    int index = 0;
    for (int num : nums) {
        if (num != 0) {
            temp[index++] = num;
        }
```

```
    }
    System.arraycopy(temp, 0, nums, 0, nums.length);
}
```

⚡ Optimized – O(n), in-place

```
java
CopyEdit
public void moveZeroes(int[] nums) {
    int insertPos = 0;
    for (int num : nums) {
        if (num != 0) {
            nums[insertPos++] = num;
        }
    }
    while (insertPos < nums.length) {
        nums[insertPos++] = 0;
    }
}
```

🔹 4. Rotate Array

📝 Problem:

Rotate the array to the right by `k` steps.

💡 Brute Force – O(n × k)

```
java
CopyEdit
public void rotateBrute(int[] nums, int k) {
    k %= nums.length;
    for (int i = 0; i < k; i++) {
        int last = nums[nums.length - 1];
        for (int j = nums.length - 1; j > 0; j--) {
            nums[j] = nums[j - 1];
        }
        nums[0] = last;
    }
}
```

⚡ Optimized – O(n) with reversal

```
java
CopyEdit
public void rotate(int[] nums, int k) {
    k %= nums.length;
    reverse(nums, 0, nums.length - 1);
    reverse(nums, 0, k - 1);
    reverse(nums, k, nums.length - 1);
}

private void reverse(int[] nums, int start, int end) {
    while (start < end) {
```

```
        int temp = nums[start];
        nums[start++] = nums[end];
        nums[end--] = temp;
    }
}
```

🔹 5. Trapping Rain Water

📝 Problem:

Calculate the amount of water that can be trapped after raining.

💡 Brute Force – O(n²)

```
java
CopyEdit
public int trapBrute(int[] height) {
    int n = height.length, water = 0;
    for (int i = 0; i < n; i++) {
        int leftMax = 0, rightMax = 0;
        for (int j = i; j >= 0; j--) leftMax = Math.max(leftMax,
height[j]);
        for (int j = i; j < n; j++) rightMax = Math.max(rightMax,
height[j]);
        water += Math.min(leftMax, rightMax) - height[i];
    }
    return water;
}
```

⚡ Optimized – O(n) with Two Pointers

```
java
CopyEdit
public int trap(int[] height) {
    int left = 0, right = height.length - 1;
    int leftMax = 0, rightMax = 0, water = 0;

    while (left < right) {
        if (height[left] < height[right]) {
            if (height[left] >= leftMax) leftMax = height[left];
            else water += leftMax - height[left];
            left++;
        } else {
            if (height[right] >= rightMax) rightMax = height[right];
            else water += rightMax - height[right];
            right--;
        }
    }
    return water;
}
```

◇ String (5 Questions)

🔹 1. Valid Anagram

📝 Problem:

Given two strings `s` and `t`, return `true` if `t` is an anagram of `s`.

💡 Brute Force – Sort & Compare – O(n log n)

```
java
CopyEdit
public boolean isAnagramBrute(String s, String t) {
    if (s.length() != t.length()) return false;
    char[] a = s.toCharArray();
    char[] b = t.toCharArray();
    Arrays.sort(a);
    Arrays.sort(b);
    return Arrays.equals(a, b);
}
```

⚡ Optimized – Count Characters – O(n)

```
java
CopyEdit
public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) return false;

    int[] count = new int[26];
    for (int i = 0; i < s.length(); i++) {
        count[s.charAt(i) - 'a']++;
        count[t.charAt(i) - 'a']--;
    }

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

    return true;
}
```

🔹 2. Longest Palindromic Substring

📝 Problem:

Return the **longest substring** of `s` that is a palindrome.

💡 Brute Force – Check all substrings – $O(n^3)$

```
java
CopyEdit
public String longestPalindromeBrute(String s) {
    int maxLen = 0;
    String result = "";

    for (int i = 0; i < s.length(); i++) {
        for (int j = i; j < s.length(); j++) {
            String sub = s.substring(i, j + 1);
            if (isPalindrome(sub) && sub.length() > maxLen) {
                result = sub;
                maxLen = sub.length();
            }
        }
    }
    return result;
}

private boolean isPalindrome(String str) {
    int l = 0, r = str.length() - 1;
    while (l < r) {
        if (str.charAt(l++) != str.charAt(r--)) return false;
    }
    return true;
}
```

⚡ Optimized – Expand Around Center – $O(n^2)$

```
java
CopyEdit
public String longestPalindrome(String s) {
    if (s == null || s.length() < 1) return "";

    int start = 0, end = 0;

    for (int i = 0; i < s.length(); i++) {
        int len1 = expand(s, i, i);       // Odd length
        int len2 = expand(s, i, i + 1);   // Even length
        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;
}
```

◇ Linked List (5 Questions)

🔹 1. Reverse a Linked List

📝 Problem:

Reverse a singly linked list.

⚡ Optimized – Iterative O(n)

```
java
CopyEdit
public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode nextNode = head.next;
        head.next = prev;
        prev = head;
        head = nextNode;
    }
    return prev;
}
```

💡 **Explanation:** Keep re-pointing `next` to the previous node while moving forward.

🔹 2. Detect Cycle in Linked List

📝 Problem:

Return `true` if a cycle exists in the linked list.

⚡ Optimized – Floyd's Cycle Detection – O(n)

```
java
CopyEdit
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;
}
```

💡 Explanation:

If there's a loop, fast and slow pointers will eventually meet.

🔹 3. Merge Two Sorted Lists

📝 Problem:

Merge two sorted linked lists into one sorted list.

⚡ Optimized – Iterative Merge – O(n + m)

```
java
CopyEdit
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(-1);
    ListNode curr = dummy;

    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            curr.next = l1;
            l1 = l1.next;
        } else {
            curr.next = l2;
            l2 = l2.next;
        }
        curr = curr.next;
    }

    curr.next = (l1 != null) ? l1 : l2;
    return dummy.next;
}
```

🔹 4. Palindrome Linked List

📝 Problem:

Return `true` if the list is a palindrome.

⚡ Optimized – Reverse 2nd Half + Compare – O(n)

```
java
CopyEdit
public boolean isPalindrome(ListNode head) {
    if (head == null || head.next == null) return true;

    // Find middle
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
```

```
        slow = slow.next;
        fast = fast.next.next;
    }

    // Reverse second half
    ListNode secondHalf = reverseList(slow);

    // Compare both halves
    ListNode firstHalf = head;
    while (secondHalf != null) {
        if (firstHalf.val != secondHalf.val) return false;
        firstHalf = firstHalf.next;
        secondHalf = secondHalf.next;
    }

    return true;
}

private ListNode reverseList(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode next = head.next;
        head.next = prev;
        prev = head;
        head = next;
    }
    return prev;
}
```

🔹 5. Remove Nth Node From End

📝 Problem:

Remove the Nth node from the end of the list.

⚡ Optimized – Two Pointer Approach – O(n)

```
java
CopyEdit
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0);
    dummy.next = head;

    ListNode first = dummy, second = dummy;

    // Move first n+1 steps ahead
    for (int i = 0; i <= n; i++) {
        first = first.next;
    }

    // Move both pointers
    while (first != null) {
        first = first.next;
        second = second.next;
    }
```

◇ Tree (5 Questions)

🔹 1. Invert Binary Tree

📝 Problem:

Flip the binary tree (mirror it).

⚡ Optimized – Recursive – O(n)

```
java
CopyEdit
public TreeNode invertTree(TreeNode root) {
    if (root == null) return null;

    TreeNode left = invertTree(root.left);
    TreeNode right = invertTree(root.right);

    root.left = right;
    root.right = left;

    return root;
}
```

🔹 2. Level Order Traversal

📝 Problem:

Return nodes of a binary tree level-by-level (BFS).

⚡ BFS Using Queue – O(n)

```
java
CopyEdit
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;

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

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

        for (int i = 0; i < size; i++) {
            TreeNode curr = queue.poll();
            level.add(curr.val);
```

```
            if (curr.left != null) queue.offer(curr.left);
            if (curr.right != null) queue.offer(curr.right);
        }

        result.add(level);
    }

    return result;
}
```

🔹 3. Lowest Common Ancestor (LCA) of Binary Tree

📝 Problem:

Given `root`, and two nodes `p` and `q`, return their LCA.

⚡ Recursive DFS – O(n)

```
java
CopyEdit
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q)
{
    if (root == null || root == p || root == q) return root;

    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    if (left != null && right != null) return root;
    return (left != null) ? left : right;
}
```

🔹 4. Diameter of Binary Tree

📝 Problem:

Find the length of the longest path between any two nodes.

⚡ DFS + Height Tracking – O(n)

```
java
CopyEdit
int max = 0;

public int diameterOfBinaryTree(TreeNode root) {
    maxDepth(root);
    return max;
}

private int maxDepth(TreeNode node) {
    if (node == null) return 0;
```

```
    int left = maxDepth(node.left);
    int right = maxDepth(node.right);
    max = Math.max(max, left + right); // Update global max

    return 1 + Math.max(left, right);
}
```

🔹 5. Symmetric Tree

📝 Problem:

Check if the tree is a mirror of itself.

⚡ Recursive Mirror Check – O(n)

```
java
CopyEdit
public boolean isSymmetric(TreeNode root) {
    if (root == null) return true;
    return isMirror(root.left, root.right);
}

private boolean isMirror(TreeNode t1, TreeNode t2) {
    if (t1 == null && t2 == null) return true;
    if (t1 == null || t2 == null) return false;

    return (t1.val == t2.val) &&
           isMirror(t1.left, t2.right) &&
           isMirror(t1.right, t2.left);
}
```

◇ Stack & Queue (5 Questions)

🔹 1. Valid Parentheses

📝 Problem:

Check if the string has valid open-close brackets.

⚡ Stack – O(n)

```
java
CopyEdit
public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();
    for (char c : s.toCharArray()) {
        if (c == '(' || c == '{' || c == '[') {
            stack.push(c);
        } else {
            if (stack.isEmpty()) return false;
            char top = stack.pop();
            if ((c == ')' && top != '(') ||
                (c == '}' && top != '{') ||
                (c == ']' && top != '[')) return false;
        }
    }
    return stack.isEmpty();
}
```

🔹 2. Min Stack

📝 Problem:

Design a stack that supports `push`, `pop`, `top`, and retrieving the minimum in constant time.

⚡ Two Stacks – O(1) min()

```
java
CopyEdit
class MinStack {
    Stack<Integer> stack = new Stack<>();
    Stack<Integer> minStack = new Stack<>();

    public void push(int val) {
        stack.push(val);
        if (minStack.isEmpty() || val <= minStack.peek())
            minStack.push(val);
    }
```

```
    public void pop() {
        if (stack.pop().equals(minStack.peek())) {
            minStack.pop();
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}
```

🔹 3. Evaluate Reverse Polish Notation

📝 Problem:

Evaluate the RPN expression like `["2","1","+","3","*"]`.

⚡ Stack – O(n)

```
java
CopyEdit
public int evalRPN(String[] tokens) {
    Stack<Integer> stack = new Stack<>();

    for (String token : tokens) {
        if ("+-*/".contains(token)) {
            int b = stack.pop(), a = stack.pop();
            switch (token) {
                case "+": stack.push(a + b); break;
                case "-": stack.push(a - b); break;
                case "*": stack.push(a * b); break;
                case "/": stack.push(a / b); break;
            }
        } else {
            stack.push(Integer.parseInt(token));
        }
    }

    return stack.pop();
}
```

🔹 4. Implement Queue using Stacks

📝 Problem:

Use two stacks to implement a queue.

⚡ Two Stacks – Amortized O(1)

```
java
CopyEdit
class MyQueue {
    Stack<Integer> in = new Stack<>();
    Stack<Integer> out = new Stack<>();

    public void push(int x) {
        in.push(x);
    }

    public int pop() {
        peek();
        return out.pop();
    }

    public int peek() {
        if (out.isEmpty()) {
            while (!in.isEmpty())
                out.push(in.pop());
        }
        return out.peek();
    }

    public boolean empty() {
        return in.isEmpty() && out.isEmpty();
    }
}
```

🔹 5. Sliding Window Maximum

📝 Problem:

Return max in each window of size `k`.

⚡ Deque (Monotonic Queue) – O(n)

```
java
CopyEdit
public int[] maxSlidingWindow(int[] nums, int k) {
    Deque<Integer> dq = new LinkedList<>();
    int n = nums.length;
    int[] res = new int[n - k + 1];

    for (int i = 0; i < n; i++) {
        while (!dq.isEmpty() && dq.peekFirst() <= i - k)
            dq.pollFirst(); // remove out of window
        while (!dq.isEmpty() && nums[dq.peekLast()] < nums[i])
            dq.pollLast(); // maintain decreasing order
        dq.offerLast(i);

        if (i >= k - 1)
            res[i - k + 1] = nums[dq.peekFirst()];
    }

    return res;}
```

◇ Dynamic Programming (DP) (5 Questions)

🔹 1. Climbing Stairs

📝 Problem:

You can take 1 or 2 steps. How many distinct ways to reach the top of `n` stairs?

⚡ DP (Fibonacci style) – O(n), O(1) space

```
java
CopyEdit
public int climbStairs(int n) {
    if (n <= 2) return n;
    int first = 1, second = 2;
    for (int i = 3; i <= n; i++) {
        int third = first + second;
        first = second;
        second = third;
    }
    return second;
}
```

🔹 2. House Robber

📝 Problem:

Can't rob adjacent houses. Max money you can rob?

⚡ DP – O(n), O(1) space

```
java
CopyEdit
public int rob(int[] nums) {
    if (nums.length == 0) return 0;
    int prev1 = 0, prev2 = 0;
    for (int num : nums) {
        int temp = prev1;
        prev1 = Math.max(prev2 + num, prev1);
        prev2 = temp;
    }
    return prev1;
}
```

🔹 3. Coin Change

📝 Problem:

Minimum number of coins to make amount. Return -1 if not possible.

⚡ Bottom-Up DP – O(n * amount)

```
java
CopyEdit
public int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, amount + 1); // use amount+1 as "infinity"
    dp[0] = 0;

    for (int a = 1; a <= amount; a++) {
        for (int c : coins) {
            if (a - c >= 0) {
                dp[a] = Math.min(dp[a], 1 + dp[a - c]);
            }
        }
    }

    return dp[amount] > amount ? -1 : dp[amount];
}
```

🔹 4. Longest Increasing Subsequence

📝 Problem:

Find the length of the longest increasing subsequence in array.

⚡ DP with Binary Search – O(n log n)

```
java
CopyEdit
public int lengthOfLIS(int[] nums) {
    List<Integer> sub = new ArrayList<>();
    for (int num : nums) {
        int i = Collections.binarySearch(sub, num);
        if (i < 0) i = -(i + 1);
        if (i == sub.size()) sub.add(num);
        else sub.set(i, num);
    }
    return sub.size();
}
```

🔹 5. Edit Distance

📝 Problem:

Min operations to convert word1 → word2 (insert, delete, replace).