LinkedIn

Coding Interview Questions

Complete Problem Statements & Java Solutions

53 Problems | Arrays • Trees • DP • Graphs • Design

📂 Arrays & Strings

1. Two Sum [Easy]

📋 Problem Statement

Given an array of integers nums and an integer target, return 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.

Example:

Input: nums = [2,7,11,15], target = 9

Output: [0,1]

Explanation: nums[0] + nums[1] == 9

💡 **Approach:** **Use a HashMap to store each number and its index. For each element, check if (target - element) exists in the map.**

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

☕ **Java Implementation**

```
public class Solution {

    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[]{};

    }

}
```

15. 3Sum [Medium]

📋 Problem Statement

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, j != k, and nums[i] + nums[j] + nums[k] == 0. The solution set must not contain duplicate triplets.

Example:

Input: nums = [-1,0,1,2,-1,-4]

Output: [[-1,-1,2],[-1,0,1]]

💡 **Approach: Sort the array. For each element, use two pointers (left and right) to find pairs that sum to the negation of the current element. Skip duplicates.**

⏱️ **Complexity: Time: O(n²) | Space: O(n)**

☕ Java Implementation

```
public class Solution {

    public List<List<Integer>> threeSum(int[] nums) {

        List<List<Integer>> result = new ArrayList<>();

        Arrays.sort(nums);

        for (int i = 0; i < nums.length - 2; i++) {

            if (i > 0 && nums[i] == nums[i - 1]) continue;

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

            while (left < right) {

                int sum = nums[i] + nums[left] + nums[right];

                if (sum == 0) {

                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));

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

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

                    left++; right--;

                } else if (sum < 0) {

                    left++;

                } else {

                    right--;

                }

            }

        }

        return result;

    }

}
```

26. Remove Duplicates from Sorted Array [Easy]

📋 Problem Statement

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Return k after placing the final result in the first k slots of nums.

Example:

Input: nums = [1,1,2]

Output: 2, nums = [1,2,_]

💡 **Approach:** **Use a slow pointer k to track where to write the next unique element. Iterate with a fast pointer; whenever we see a new value, write it at nums[k] and increment k.**

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

☕ **Java Implementation**

```
public class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;
        int k = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] != nums[i - 1]) {
                nums[k++] = nums[i];
            }
        }
        return k;
    }
}
```

53. Maximum Subarray [Medium]

📋 **Problem Statement**

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

Example:

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

Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

💡 **Approach: Kadane's Algorithm: maintain a running sum. If the running sum becomes negative, reset it to 0. Track the maximum seen so far.**

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

☕ Java Implementation

```
public class Solution {
    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;
    }
}
```

56. Merge Intervals [Medium]

📋 Problem Statement

Given an array of intervals where intervals[i] = [start_i, end_i], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example:

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

Output: [[1,6],[8,10],[15,18]]

💡 **Approach:** **Sort intervals by start time. Iterate and merge if the current interval overlaps with the last merged interval (current start <= last end).**

⏱️ **Complexity:** **Time: O(n log n) | Space: O(n)**

☕ Java Implementation

```
public class Solution {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
        List<int[]> merged = new ArrayList<>();
        for (int[] interval : intervals) {
            if (merged.isEmpty() || merged.get(merged.size() - 1)[1] < interval[0])
{
                merged.add(interval);
            } else {
                merged.get(merged.size() - 1)[1] =
                    Math.max(merged.get(merged.size() - 1)[1], interval[1]);
            }
        }
        return merged.toArray(new int[merged.size()][]);
    }
}
```

152. Maximum Product Subarray [Medium]

📋 Problem Statement

Given an integer array nums, find a subarray that has the largest product, and return the product.

Example:

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

Output: 6

Explanation: [2,3] has the largest product 6.

💡 **Approach:** **Track both max and min products at each position (min is needed because multiplying two negatives gives a positive). Update global max at each step.**

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

☕ **Java Implementation**

```
public class Solution {

    public int maxProduct(int[] nums) {

        int maxProd = nums[0], minProd = nums[0], result = nums[0];

        for (int i = 1; i < nums.length; i++) {

            int temp = maxProd;

            maxProd = Math.max(nums[i], Math.max(maxProd * nums[i], minProd *
nums[i]));

            minProd = Math.min(nums[i], Math.min(temp * nums[i], minProd *
nums[i]));

            result = Math.max(result, maxProd);

        }

        return result;

    }

}
```

163. Missing Ranges [Easy]

📋 **Problem Statement**

You are given an inclusive range [lower, upper] and a sorted unique integer array nums, where all elements are within [lower, upper]. A number x is considered missing if x is in [lower, upper] and x is not in nums. Return the shortest sorted list of ranges that exactly covers all the missing numbers.

Example:

Input: nums = [0,1,3,50,75], lower = 0, upper = 99

Output: ["2","4->49","51->74","76->99"]

💡 **Approach:** **Iterate through nums along with boundary sentinels. For each gap between consecutive numbers, add the missing range.**

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

☕ **Java Implementation**

```
public class Solution {
    public List<String> findMissingRanges(int[] nums, int lower, int upper) {
        List<String> result = new ArrayList<>();
        long prev = (long) lower - 1;
        for (int i = 0; i <= nums.length; i++) {
            long curr = (i == nums.length) ? (long) upper + 1 : nums[i];
            if (curr - prev >= 2) {
                long start = prev + 1, end = curr - 1;
                result.add(start == end ? String.valueOf(start) : start + "->" +
end);
            }
            prev = curr;
        }
        return result;
    }
}
```

238. Product of Array Except Self [Medium]

📋 **Problem Statement**

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all elements of nums except nums[i]. You must write an algorithm that runs in O(n) time and without using the division operation.

Example:

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

Output: [24,12,8,6]

💡 **Approach:** **Compute prefix products left to right, then multiply by suffix products right to left — all in a single output array without division.**

⏱ **Complexity:** **Time: O(n) | Space: O(1) extra**

☕ **Java Implementation**

```
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;
        int[] result = new int[n];
        result[0] = 1;
        for (int i = 1; i < n; i++) {
            result[i] = result[i - 1] * nums[i - 1];
        }
        int right = 1;
        for (int i = n - 1; i >= 0; i--) {
            result[i] *= right;
            right *= nums[i];
        }
        return result;
    }
}
```

242. Valid Anagram [Easy]

📋 **Problem Statement**

Given two strings s and t, return true if t is an anagram of s, and false otherwise. An anagram is a word formed by rearranging the letters of another.

Example:

Input: s = "anagram", t = "nagaram"

Output: true

💡 **Approach:** **Use a frequency array of size 26. Increment for characters in s, decrement for characters in t. If all counts are 0, the strings are anagrams.**

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

☕ Java Implementation

```
public class 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']--;
        for (int n : count) {
            if (n != 0) return false;
        }
        return true;
    }
}
```

271. Encode and Decode Strings [Medium]

📋 Problem Statement

Design an algorithm to encode a list of strings to a single string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Example:

encode(["Hello","World"]) -> some encoded string

decode(encoded) -> ["Hello","World"]

💡 **Approach:** **Use a length-prefix encoding: for each string, write its length followed by "#" and then the string content. Decoding reads the number before "#" to know how many chars to consume.**

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

☕ **Java Implementation**

```
public class Codec {
    public String encode(List<String> strs) {
        StringBuilder sb = new StringBuilder();
        for (String s : strs) {
            sb.append(s.length()).append('#').append(s);
        }
        return sb.toString();
    }

    public List<String> decode(String s) {
        List<String> result = new ArrayList<>();
        int i = 0;
        while (i < s.length()) {
            int j = s.indexOf('#', i);
            int len = Integer.parseInt(s.substring(i, j));
            result.add(s.substring(j + 1, j + 1 + len));
            i = j + 1 + len;
        }
        return result;
    }
}
```

📂 Linked Lists

21. Merge Two Sorted Lists [Easy]

📋 Problem Statement

You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. Return the head of the merged linked list.

Example:

Input: list1 = [1,2,4], list2 = [1,3,4]

Output: [1,1,2,3,4,4]

💡 **Approach: Use a dummy node. Compare heads of both lists, append the smaller node to result, and advance that list's pointer. Append any remaining nodes.**

⏱️ **Complexity: Time: O(n + m) | Space: O(1)**

☕ Java Implementation

```
public class Solution {

    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {

        ListNode dummy = new ListNode(0);

        ListNode curr = dummy;

        while (list1 != null && list2 != null) {

            if (list1.val <= list2.val) {

                curr.next = list1;

                list1 = list1.next;

            } else {

                curr.next = list2;

                list2 = list2.next;

            }

            curr = curr.next;

        }

        curr.next = (list1 != null) ? list1 : list2;

        return dummy.next;

    }

}
```

23. Merge K Sorted Lists [Hard]

📋 Problem Statement

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order. Merge all the linked-lists into one sorted linked-list and return it.

Example:

Input: lists = [[1,4,5],[1,3,4],[2,6]]

Output: [1,1,2,3,4,4,5,6]

💡 **Approach: Use a min-heap (PriorityQueue) of size k. Add the head of each list. Repeatedly extract the minimum node, append it to result, and push its next node into the heap.**

⏱️ **Complexity:** **Time: O(N log k) | Space: O(k)**

☕ Java Implementation

```
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        PriorityQueue<ListNode> pq = new PriorityQueue<>((a, b) -> a.val - b.val);
        for (ListNode node : lists) {
            if (node != null) pq.offer(node);
        }
        ListNode dummy = new ListNode(0), curr = dummy;
        while (!pq.isEmpty()) {
            ListNode node = pq.poll();
            curr.next = node;
            curr = curr.next;
            if (node.next != null) pq.offer(node.next);
        }
        return dummy.next;
    }
}
```

206. Reverse Linked List [Easy]

📋 Problem Statement

Given the head of a singly linked list, reverse the list, and return the reversed list.

Example:

Input: head = [1,2,3,4,5]

Output: [5,4,3,2,1]

💡 **Approach:** **Iteratively reverse the links. Keep track of prev, curr, and next. At each step, point**

curr.next to prev, then advance both pointers.

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

☕ Java Implementation

```
public class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null, curr = head;
        while (curr != null) {
            ListNode next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
}
```

📂 Trees & Graphs

98. Validate Binary Search Tree [Medium]

📋 Problem Statement

Given the root of a binary tree, determine if it is a valid binary search tree (BST). A valid BST is defined as: left subtree contains only nodes less than the node's key, right subtree only nodes greater, and both subtrees must also be BSTs.

Example:

Input: root = [2,1,3]

Output: true

💡 **Approach:** **Pass valid min/max bounds when recursing. For each node, check if its value is within the allowed range, then recurse with updated bounds.**

⏱️ **Complexity:** **Time: O(n) | Space: O(h)**

☕ Java Implementation

```
public class Solution {
    public boolean isValidBST(TreeNode root) {
        return validate(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }
    private boolean validate(TreeNode node, long min, long max) {
        if (node == null) return true;
        if (node.val <= min || node.val >= max) return false;
        return validate(node.left, min, node.val) &&
               validate(node.right, node.val, max);
    }
}
```

102. Binary Tree Level Order Traversal [Medium]

📋 Problem Statement

Given the root of a binary tree, return the level order traversal of its nodes' values (i.e., from left to right, level by level).

Example:

Input: root = [3,9,20,null,null,15,7]

Output: [[3],[9,20],[15,7]]

💡 **Approach:** **BFS using a queue. At each level, process all current nodes, collect their values, and**

enqueue their children.

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

☕ Java Implementation

```
public class Solution {
    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 node = queue.poll();
                level.add(node.val);
                if (node.left != null) queue.offer(node.left);
                if (node.right != null) queue.offer(node.right);
            }
            result.add(level);
        }
        return result;
    }
}
```

103. Binary Tree Zigzag Level Order Traversal [Medium]

📋 Problem Statement

Given the root of a binary tree, return the zigzag level order traversal of its nodes' values (i.e., from left to right, then right to left for the next level and alternate between).

Example:

Input: root = [3,9,20,null,null,15,7]

Output: [[3],[20,9],[15,7]]

💡 **Approach:** **BFS level order traversal. Use a boolean flag to alternate the direction of inserting nodes. Use a LinkedList to add from front or back based on the flag.**

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

☕ **Java Implementation**

```
public class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) return result;
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        boolean leftToRight = true;
        while (!queue.isEmpty()) {
            int size = queue.size();
            LinkedList<Integer> level = new LinkedList<>();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                if (leftToRight) level.addLast(node.val);
                else level.addFirst(node.val);
                if (node.left != null) queue.offer(node.left);
                if (node.right != null) queue.offer(node.right);
            }
            result.add(level);
            leftToRight = !leftToRight;
        }
        return result;
    }
}
```

104. Maximum Depth of Binary Tree [Easy]

📋 Problem Statement

Given the root of a binary tree, return its maximum depth. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example:

Input: root = [3,9,20,null,null,15,7]

Output: 3

💡 **Approach: Recursive DFS: the depth is 1 + max(depth(left), depth(right)). Base case: null node returns 0.**

⏱️ **Complexity: Time: O(n) | Space: O(h)**

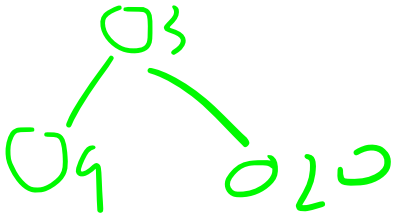
☕ Java Implementation

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

110. Balanced Binary Tree [Easy]

📋 Problem Statement

Given a binary tree, determine if it is height-balanced (a tree where every node's left and right subtrees differ in height by no more than 1).



Example:

Input: root = [3,9,20,null,null,15,7]

Output: true

💡 **Approach:** **DFS returning height of subtree. If any subtree is unbalanced, return -1 as a signal. The tree is balanced only if no -1 is propagated to the root.**

⏱️ **Complexity:** **Time: O(n) | Space: O(h)**

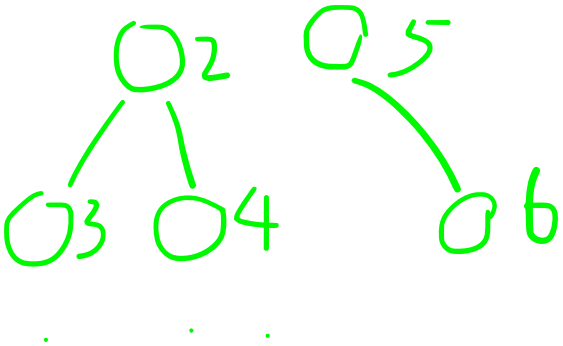
☕ **Java Implementation**

```
public class Solution {
    public boolean isBalanced(TreeNode root) {
        return checkHeight(root) != -1;
    }
    private int checkHeight(TreeNode node) {
        if (node == null) return 0;
        int left = checkHeight(node.left);
        if (left == -1) return -1;
        int right = checkHeight(node.right);
        if (right == -1) return -1;
        if (Math.abs(left - right) > 1) return -1;
        return 1 + Math.max(left, right);
    }
}
```

114. Flatten Binary Tree to Linked List [Medium]

📋 **Problem Statement**

Given the root of a binary tree, flatten the tree into a "linked list" in-place using the same TreeNode class. The linked list should follow preorder traversal (root, left, right). left child of every node must be null.



Example:

Input: root = [1,2,5,3,4,null,6]

Output: [1,null,2,null,3,null,4,null,5,null,6]

💡 **Approach:** **For each node with a left child, find the rightmost node of the left subtree, connect it to the right subtree, move left subtree to right, set left to null.**

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

☕ **Java Implementation**

```
public class Solution {
    public void flatten(TreeNode root) {
        TreeNode curr = root;
        while (curr != null) {
            if (curr.left != null) {
                TreeNode rightmost = curr.left;
                while (rightmost.right != null) rightmost = rightmost.right;
                rightmost.right = curr.right;
                curr.right = curr.left;
                curr.left = null;
            }
            curr = curr.right;
        }
    }
}
```

199. Binary Tree Right Side View [Medium]

📋 **Problem Statement**

Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of

the nodes you can see ordered from top to bottom.

Example:

Input: root = [1,2,3,null,5,null,4]

Output: [1,3,4]

💡 **Approach:** **BFS level order traversal. For each level, the last node encountered is the rightmost (visible) node. Add only that node's value to the result.**

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

☕ **Java Implementation**

```
public class Solution {
    public List<Integer> rightSideView(TreeNode root) {
        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();
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                if (i == size - 1) result.add(node.val);
                if (node.left != null) queue.offer(node.left);
                if (node.right != null) queue.offer(node.right);
            }
        }
        return result;
    }
}
```

226. Invert Binary Tree [Easy]

📋 Problem Statement

Given the root of a binary tree, invert the tree, and return its root. Inverting means swapping every left child with the corresponding right child recursively.

Example:

Input: root = [4,2,7,1,3,6,9]

Output: [4,7,2,9,6,3,1]

💡 **Approach:** **Recursively swap left and right children for every node using DFS.**

⏱️ **Complexity:** **Time: O(n) | Space: O(h)**

☕ Java Implementation

```
public class Solution {
    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;
    }
}
```

236. Lowest Common Ancestor of a Binary Tree [Medium]

📋 Problem Statement

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes p and q. The LCA is the lowest node in the tree that has both p and q as descendants (a node can be a descendant of itself).

Example:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

💡 **Approach:** **Post-order DFS. If the current node is p or q, return it. If both left and right return non-null, current node is the LCA. Otherwise return the non-null side.**

⏱️ **Complexity:** **Time: O(n) | Space: O(h)**

☕ Java Implementation

```
public class Solution {
    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;
    }
}
```

173. Binary Search Tree Iterator [Medium]

📋 Problem Statement

Implement the BSTIterator class that represents an iterator over the in-order traversal of a BST. next() returns the next smallest number. hasNext() returns true if there is still a next number in the traversal.

Example:

BSTIterator(root=[7,3,15,null,null,9,20])

next() -> 3

next() -> 7

hasNext() -> true

💡 **Approach:** **Use an explicit stack to simulate in-order traversal. Push all left nodes initially. For next(), pop a node, then push all left nodes of its right child.**

⏱️ **Complexity:** **Time: O(1) amortized | Space: O(h)**

☕ **Java Implementation**

```
public class BSTIterator {

    private Stack<TreeNode> stack = new Stack<>();


    public BSTIterator(TreeNode root) {

        pushLeft(root);

    }


    private void pushLeft(TreeNode node) {

        while (node != null) {

            stack.push(node);

            node = node.left;

        }

    }


    public int next() {

        TreeNode node = stack.pop();

        pushLeft(node.right);

        return node.val;

    }


    public boolean hasNext() {

        return !stack.isEmpty();

    }
}
```

297. Serialize and Deserialize Binary Tree [Hard]

📋 Problem Statement

Design an algorithm to serialize and deserialize a binary tree. Serialization is the process of converting a tree to a string, and deserialization is converting the string back to the original tree structure.

Example:

serialize([1,2,3,null,null,4,5]) -> "1,2,3,null,null,4,5"

deserialize("1,2,3,null,null,4,5") -> [1,2,3,null,null,4,5]

💡 **Approach:** **BFS serialization: encode each node's value, use 'null' for missing children. Deserialization processes values from a queue, building the tree level by level.**

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

☕ **Java Implementation**

```
public class Codec {

    public String serialize(TreeNode root) {

        if (root == null) return "";

        StringBuilder sb = new StringBuilder();

        Queue<TreeNode> q = new LinkedList<>();

        q.offer(root);

        while (!q.isEmpty()) {

            TreeNode node = q.poll();

            if (node == null) { sb.append("null,"); continue; }

            sb.append(node.val).append(",");

            q.offer(node.left);

            q.offer(node.right);

        }

        return sb.toString();

    }


    public TreeNode deserialize(String data) {

        if (data.isEmpty()) return null;

        String[] vals = data.split(",");

        TreeNode root = new TreeNode(Integer.parseInt(vals[0]));

        Queue<TreeNode> q = new LinkedList<>();

        q.offer(root);

        int i = 1;

        while (!q.isEmpty() && i < vals.length) {

            TreeNode node = q.poll();

            if (!vals[i].equals("null")) {

                node.left = new TreeNode(Integer.parseInt(vals[i]));

                q.offer(node.left);

            }

            i++;

            if (i < vals.length && !vals[i].equals("null")) {

                node.right = new TreeNode(Integer.parseInt(vals[i]));

                q.offer(node.right);

            }
```

200. Number of Islands [Medium]

📋 Problem Statement

Given an m x n 2D binary grid where '1' represents land and '0' represents water, return the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically.

Example:

Input: grid = [["1","1","0"],["1","1","0"],["0","0","1"]]

Output: 2

💡 **Approach:** **DFS flood-fill: iterate through grid. When a '1' is found, increment count and DFS to mark all connected '1's as '0' (visited). Each DFS call covers one island.**

⏱️ **Complexity:** **Time: O(m*n) | Space: O(m*n)**

☕ **Java Implementation**

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

323. Number of Connected Components in Undirected Graph [Medium]

📋 Problem Statement

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

Example:

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

Output: 2

💡 **Approach:** **Union-Find (Disjoint Set Union): initialize each node as its own parent. For each edge, union the two endpoints. Count the number of remaining distinct roots.**

⏱️ **Complexity:** **Time: O(n + E * α(n)) | Space: O(n)**

☕ **Java Implementation**

```
public class Solution {
    int[] parent;
    public int countComponents(int n, int[][] edges) {
        parent = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;
        int components = n;
        for (int[] edge : edges) {
            int p1 = find(edge[0]), p2 = find(edge[1]);
            if (p1 != p2) {
                parent[p1] = p2;
                components--;
            }
        }
        return components;
    }
    private int find(int x) {
        while (parent[x] != x) {
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }
}
```

📂 Dynamic Programming

55. Jump Game [Medium]

📋 Problem Statement

You are given an integer array nums. Each element represents your maximum jump length at that position. Return true if you can reach the last index, or false otherwise.

Example:

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

Output: true

💡 **Approach: Greedy: track the furthest reachable index. If the current index exceeds it, return false. Update max reach at each step.**

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

☕ Java Implementation

```
public class Solution {
    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;
    }
}
```

45. Jump Game II [Medium]

📋 Problem Statement

Given an integer array nums where each element is the max jump length, return the minimum number of jumps to reach the last index.

Example:

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

Output: 2

Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

💡 **Approach: Greedy BFS: treat each 'jump' as a level. Track the farthest reachable index within the current jump range. When the range is exhausted, increment jumps.**

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

☕ Java Implementation

```
public class Solution {
    public int jump(int[] nums) {
        int jumps = 0, currEnd = 0, farthest = 0;
        for (int i = 0; i < nums.length - 1; i++) {
            farthest = Math.max(farthest, i + nums[i]);
            if (i == currEnd) {
                jumps++;
                currEnd = farthest;
            }
        }
        return jumps;
    }
}
```

62. Unique Paths [Medium]

📋 Problem Statement

A robot is located at the top-left corner of an m x n grid. It can only move either down or right. It is trying to reach the bottom-right corner. How many possible unique paths are there?

Example:

Input: m = 3, n = 7

Output: 28

💡 **Approach: DP: dp[i][j] = dp[i-1][j] + dp[i][j-1]. The number of ways to reach (i,j) is the sum of ways from the cell above and the cell to the left.**

⏱️ **Complexity: Time: O(m*n) | Space: O(m*n)**

☕ Java Implementation

```
public class Solution {
    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];
    }
}
```

198. House Robber [Easy]

📋 Problem Statement

You are a professional robber. Given an integer array nums representing the amount of money at each house, return the maximum money you can rob without robbing two adjacent houses.

Example:

Input: nums = [1,2,3,1]

Output: 4

Explanation: Rob house 1 (1) then house 3 (3) = 4.

💡 **Approach: DP: at each house, choose max of (skip current house) vs (rob current + best 2 houses ago). Maintain two rolling variables instead of an array.**

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

☕ Java Implementation

```
public class Solution {
    public int rob(int[] nums) {
        int prev2 = 0, prev1 = 0;
        for (int num : nums) {
            int curr = Math.max(prev1, prev2 + num);
            prev2 = prev1;
            prev1 = curr;
        }
        return prev1;
    }
}
```

337. House Robber III [Medium]

📋 Problem Statement

The thief has found a new place which has only one entrance (root). All houses form a binary tree. Adjacent means directly linked parent-child. Determine the maximum amount of money the thief can rob without alerting the police.

Example:

Input: root = [3,2,3,null,3,null,1]

Output: 7

Explanation: Rob root (3) + grandchildren (3+1) = 7.

💡 **Approach: DFS returning a pair [rob, skip]. rob = node.val + skip(left) + skip(right). skip = max(rob/skip left) + max(rob/skip right). No extra memo needed.**

⏱️ **Complexity: Time: O(n) | Space: O(h)**

☕ **Java Implementation**

```
public class Solution {
    public int rob(TreeNode root) {
        int[] result = dfs(root);
        return Math.max(result[0], result[1]);
    }
    // returns [rob_this_node, skip_this_node]
    private int[] dfs(TreeNode node) {
        if (node == null) return new int[]{0, 0};
        int[] left = dfs(node.left);
        int[] right = dfs(node.right);
        int rob = node.val + left[1] + right[1];
        int skip = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
        return new int[]{rob, skip};
    }
}
```

📂 Sliding Window & Two Pointers

3. Longest Substring Without Repeating Characters [Medium]

📋 Problem Statement

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 without repeating.

💡 **Approach: Sliding window with a HashMap storing the last seen index of each character. When a duplicate is found, move the left pointer past the previous occurrence.**

⏱️ **Complexity: Time: O(n) | Space: O(min(m,n))**

☕ Java Implementation

```
public class Solution {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> map = new HashMap<>();
        int maxLen = 0, left = 0;
        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);
            if (map.containsKey(c) && map.get(c) >= left) {
                left = map.get(c) + 1;
            }
            map.put(c, right);
            maxLen = Math.max(maxLen, right - left + 1);
        }
        return maxLen;
    }
}
```

76. Minimum Window Substring [Hard]

📋 Problem Statement

Given two strings s and t, return the minimum window substring of s such that every character in t (including duplicates) is included in the window. Return the empty string if no such window exists.

Example:

Input: s = "ADOBECODEBANC", t = "ABC"

Output: "BANC"

💡 **Approach: Sliding window: expand right pointer to include all chars of t. Once valid, contract left pointer to minimize. Track counts via frequency maps.**

⏱️ **Complexity: Time: O(s + t) | Space: O(s + t)**

☕ Java Implementation

```
public class Solution {
    public String minWindow(String s, String t) {
        Map<Character, Integer> need = new HashMap<>();
        for (char c : t.toCharArray()) need.merge(c, 1, Integer::sum);
        int left = 0, have = 0, required = need.size();
        int[] best = {-1, 0, 0};
        Map<Character, Integer> window = new HashMap<>();
        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);
            window.merge(c, 1, Integer::sum);
            if (need.containsKey(c) && window.get(c).equals(need.get(c))) have++;
            while (have == required) {
                if (best[0] == -1 || right - left + 1 < best[0]) {
                    best[0] = right - left + 1; best[1] = left; best[2] = right;
                }
                char lc = s.charAt(left);
                window.merge(lc, -1, Integer::sum);
                if (need.containsKey(lc) && window.get(lc) < need.get(lc)) have--;
                left++;
            }
        }
        return best[0] == -1 ? "" : s.substring(best[1], best[2] + 1);
    }
}
```

📂 Binary Search

33. Search in Rotated Sorted Array [Medium]

📋 Problem Statement

There is a sorted array that has been rotated at some pivot. Given the array nums and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.

Example:

Input: nums = [4,5,6,7,0,1,2], target = 0

Output: 4

💡 **Approach: Modified binary search. At each step, determine which half is sorted, then check if target is in that half to decide which side to continue searching.**

⏱️ **Complexity: Time: O(log n) | Space: O(1)**

☕ Java Implementation

```
public class Solution {
    public int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) return mid;
            if (nums[left] <= nums[mid]) {
                if (target >= nums[left] && target < nums[mid]) right = mid - 1;
                else left = mid + 1;
            } else {
                if (target > nums[mid] && target <= nums[right]) left = mid + 1;
                else right = mid - 1;
            }
        }
        return -1;
    }
}
```

34. Find First and Last Position of Element in Sorted Array [Medium]

📋 Problem Statement

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value. Return [-1, -1] if not found.

Example:

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

💡 **Approach:** **Two binary searches: one to find the leftmost (first) occurrence, one for the rightmost (last) occurrence. Both are O(log n).**

⏱️ **Complexity:** **Time: O(log n) | Space: O(1)**

☕ Java Implementation

```
public class Solution {
    public int[] searchRange(int[] nums, int target) {
        return new int[]{findFirst(nums, target), findLast(nums, target)};
    }
    private int findFirst(int[] nums, int target) {
        int lo = 0, hi = nums.length - 1, idx = -1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            if (nums[mid] == target) { idx = mid; hi = mid - 1; }
            else if (nums[mid] < target) lo = mid + 1;
            else hi = mid - 1;
        }
        return idx;
    }
    private int findLast(int[] nums, int target) {
        int lo = 0, hi = nums.length - 1, idx = -1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            if (nums[mid] == target) { idx = mid; lo = mid + 1; }
            else if (nums[mid] < target) lo = mid + 1;
            else hi = mid - 1;
        }
        return idx;
    }
}
```

📂 Backtracking & Recursion

46. Permutations [Medium]

📋 Problem Statement

Given an array nums of distinct integers, return all the possible permutations in any order.

Example:

Input: nums = [1,2,3]

Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

💡 **Approach: Backtracking: build permutations by swapping elements. At each position, swap the current element with each remaining element, recurse, then swap back.**

⏱️ **Complexity: Time: O(n * n!) | Space: O(n)**

☕ Java Implementation

```java
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(nums, 0, result);
        return result;
    }
    private void backtrack(int[] nums, int start, List<List<Integer>> result) {
        if (start == nums.length) {
            List<Integer> perm = new ArrayList<>();
            for (int n : nums) perm.add(n);
            result.add(perm);
            return;
        }
        for (int i = start; i < nums.length; i++) {
            swap(nums, start, i);
            backtrack(nums, start + 1, result);
            swap(nums, start, i);
        }
    }
    private void swap(int[] nums, int i, int j) {
        int tmp = nums[i]; nums[i] = nums[j]; nums[j] = tmp;
    }
}
```

78. Subsets [Medium]

📋 Problem Statement

Given an integer array nums of unique elements, return all possible subsets (the power set). The solution set must not contain duplicate subsets. Return the solution in any order.

Example:

Input: nums = [1,2,3]

Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

💡 **Approach:** **Backtracking: at each step, include or exclude the current element. Add each constructed list to results before recursing deeper.**

⏱️ **Complexity:** **Time: O(n * 2^n) | Space: O(n)**

☕ **Java Implementation**

```
public class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(nums, 0, new ArrayList<>(), result);
        return result;
    }
    private void backtrack(int[] nums, int start, List<Integer> current,
                           List<List<Integer>> result) {
        result.add(new ArrayList<>(current));
        for (int i = start; i < nums.length; i++) {
            current.add(nums[i]);
            backtrack(nums, i + 1, current, result);
            current.remove(current.size() - 1);
        }
    }
}
```

📂 Design & Data Structures

146. LRU Cache [Medium]

📋 Problem Statement

Design a data structure that follows the Least Recently Used (LRU) cache constraints. Implement LRUCache(capacity), get(key), and put(key, value). get and put must run in O(1).

Example:

LRUCache(2); put(1,1); put(2,2); get(1)->1; put(3,3); get(2)->-1

💡 **Approach:** **Combine a HashMap (for O(1) lookup) with a Doubly Linked List (for O(1) insertion/deletion). Keep the most recently used at the head, evict from tail.**

⏱️ **Complexity:** **Time: O(1) for get/put | Space: O(capacity)**

☕ Java Implementation

```
public class LRUCache {
    class Node {
        int key, val;
        Node prev, next;
        Node(int k, int v) { key = k; val = v; }
    }
    private Map<Integer, Node> map = new HashMap<>();
    private int capacity;
    private Node head = new Node(0, 0), tail = new Node(0, 0);

    public LRUCache(int capacity) {
        this.capacity = capacity;
        head.next = tail; tail.prev = head;
    }

    public int get(int key) {
        if (!map.containsKey(key)) return -1;
        Node node = map.get(key);
        remove(node); insertFront(node);
        return node.val;
    }

    public void put(int key, int value) {
        if (map.containsKey(key)) remove(map.get(key));
        Node node = new Node(key, value);
        map.put(key, node);
        insertFront(node);
        if (map.size() > capacity) {
            Node lru = tail.prev;
            remove(lru); map.remove(lru.key);
        }
    }

    private void remove(Node node) {
```

155. Min Stack [Medium]

📋 Problem Statement

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

Example:

```
MinStack ms = new MinStack();

ms.push(-2); ms.push(0); ms.push(-3);

ms.getMin(); -> -3

ms.pop(); ms.top(); -> 0; ms.getMin(); -> -2
```

💡 **Approach:** **Maintain two stacks: one for values and one for minimums. Push to min stack when the new value is <= current min. Pop from min stack in sync.**

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

☕ Java Implementation

```
public class MinStack {

    private Stack<Integer> stack = new Stack<>();

    private 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();

    }

}
```

295. Find Median from Data Stream [Hard]

📋 Problem Statement

The median is the middle value in an ordered list. Implement MedianFinder: addNum(int) adds a number to the data stream, findMedian() returns the median.

Example:

addNum(1); addNum(2); findMedian() -> 1.5; addNum(3); findMedian() -> 2.0

💡 **Approach:** **Two heaps: a max-heap for the lower half and a min-heap for the upper half. Balance them so they differ by at most 1 element. Median is from the top(s).**

⏱️ **Complexity:** **Time: O(log n) add, O(1) find | Space: O(n)**

☕ **Java Implementation**

```
public class MedianFinder {

    private PriorityQueue<Integer> lo = new
PriorityQueue<>(Collections.reverseOrder());

    private PriorityQueue<Integer> hi = new PriorityQueue<>();


    public void addNum(int num) {

        lo.offer(num);

        hi.offer(lo.poll());

        if (lo.size() < hi.size()) {

            lo.offer(hi.poll());

        }

    }


    public double findMedian() {

        if (lo.size() > hi.size()) return lo.peek();

        return (lo.peek() + hi.peek()) / 2.0;

    }

}
```

341. Flatten Nested List Iterator [Medium]

📋 **Problem Statement**

You are given a nested list of integers nestedList. Each element is either an integer or a list whose elements may also be integers or other lists. Implement an iterator to flatten it. next() returns the

next integer, hasNext() returns true if there are still some integers in the list.

Example:

Input: nestedList = [[1,1],2,[1,1]]

Output: [1,1,2,1,1]

💡 **Approach: Pre-flatten on construction using a recursive DFS to build a flat list. Alternatively, use a stack and lazily flatten when hasNext() is called.**

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

☕ **Java Implementation**

```
public class NestedIterator implements Iterator<Integer> {
    private List<Integer> flat = new ArrayList<>();
    private int index = 0;

    public NestedIterator(List<NestedInteger> nestedList) {
        flatten(nestedList);
    }

    private void flatten(List<NestedInteger> list) {
        for (NestedInteger ni : list) {
            if (ni.isInteger()) {
                flat.add(ni.getInteger());
            } else {
                flatten(ni.getList());
            }
        }
    }

    public Integer next() {
        return flat.get(index++);
    }

    public boolean hasNext() {
        return index < flat.size();
    }
}
```

📂 Strings & Math

20. Valid Parentheses [Easy]

📋 Problem Statement

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. A string is valid if every open bracket is closed by the correct closing bracket in the correct order.

Example:

Input: s = "()[]{}"

Output: true

💡 **Approach: Stack: push open brackets. For each closing bracket, check if the top of the stack is the matching open bracket. Valid if stack is empty at end.**

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

☕ Java Implementation

```
public class Solution {
    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 != '(') return false;
                if (c == ']' && top != '[') return false;
                if (c == '}' && top != '{') return false;
            }
        }
        return stack.isEmpty();
    }
}
```

125. Valid Palindrome [Easy]

📋 Problem Statement

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Given a string s, return true if it is a palindrome, or false otherwise.

Example:

Input: s = "A man, a plan, a canal: Panama"

Output: true

💡 **Approach: Two pointers from both ends. Skip non-alphanumeric characters. Compare characters case-insensitively. Return false on mismatch.**

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

☕ Java Implementation

```
public class 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;
    }
}
```

50. Pow(x, n) [Medium]

📋 Problem Statement

Implement pow(x, n), which calculates x raised to the power n (x^n).

Example:

Input: x = 2.00000, n = 10

Output: 1024.00000

💡 **Approach:** **Fast exponentiation (binary exponentiation): if n is even, pow(x,n) = pow(x*x, n/2). If odd, multiply one extra x. Handle negative n by using 1/x.**

⏱️ **Complexity:** **Time: O(log n) | Space: O(log n)**

☕ Java Implementation

```
public class Solution {
    public double myPow(double x, int n) {
        long N = n;
        if (N < 0) { x = 1 / x; N = -N; }
        double result = 1;
        while (N > 0) {
            if ((N & 1) == 1) result *= x;
            x *= x;
            N >>= 1;
        }
        return result;
    }
}
```

227. Basic Calculator II [Medium]

📋 Problem Statement

Given a string s which represents an expression, evaluate this expression and return its value. The expression contains non-negative integers, '+', '-', '*', '/' operators, and empty spaces. Integer division truncates toward zero.

Example:

Input: s = "3+2*2"

Output: 7

💡 **Approach:** **Use a stack. For + and -, push the number (negated for -). For * and /, pop the top, compute with current number, and push the result. Sum the stack at end.**

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

☕ Java Implementation

```
public class Solution {
    public int calculate(String s) {
        Stack<Integer> stack = new Stack<>();
        int num = 0;
        char sign = '+';
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isDigit(c)) num = num * 10 + (c - '0');
            if ((!Character.isDigit(c) && c != ' ') || i == s.length() - 1) {
                if (sign == '+') stack.push(num);
                else if (sign == '-') stack.push(-num);
                else if (sign == '*') stack.push(stack.pop() * num);
                else if (sign == '/') stack.push(stack.pop() / num);
                sign = c;
                num = 0;
            }
        }
        int result = 0;
        for (int n : stack) result += n;
        return result;
    }
}
```

📂 Shortest Word Distance (LinkedIn Special)

243. Shortest Word Distance [Easy]

📋 Problem Statement

Given an array of strings wordsDict and two different strings word1 and word2, return the shortest distance between occurrences of word1 and word2 in the list.

Example:

Input: wordsDict = ["practice","makes","perfect","coding","makes"], word1 = "coding", word2 = "practice"

Output: 3

💡 **Approach:** **Single pass: track the last seen indices of word1 and word2. Whenever either is found, update its index and compute the distance between the two indices.**

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

☕ **Java Implementation**

```
public class Solution {
    public int shortestDistance(String[] wordsDict, String word1, String word2) {
        int i1 = -1, i2 = -1, minDist = Integer.MAX_VALUE;
        for (int i = 0; i < wordsDict.length; i++) {
            if (wordsDict[i].equals(word1)) i1 = i;
            else if (wordsDict[i].equals(word2)) i2 = i;
            if (i1 != -1 && i2 != -1) {
                minDist = Math.min(minDist, Math.abs(i1 - i2));
            }
        }
        return minDist;
    }
}
```

244. Shortest Word Distance II [Medium]

📋 **Problem Statement**

Design a data structure that is initialized with an array of strings wordsDict and allows repeated shortest() queries for two words.

Example:

WordDistance(["practice","makes","perfect","coding","makes"])

shortest("coding","practice") -> 3

shortest("makes","coding") -> 1

💡 **Approach:** **Precompute a map from word to its list of indices. For each query, use two pointers on the sorted index lists to find the minimum distance in O(m+n) time.**

⏱️ **Complexity:** **Time: O(m+n) per query | Space: O(n)**

☕ **Java Implementation**

```
public class WordDistance {
    private Map<String, List<Integer>> map = new HashMap<>();

    public WordDistance(String[] wordsDict) {
        for (int i = 0; i < wordsDict.length; i++) {
            map.computeIfAbsent(wordsDict[i], k -> new ArrayList<>()).add(i);
        }
    }

    public int shortest(String word1, String word2) {
        List<Integer> l1 = map.get(word1), l2 = map.get(word2);
        int i = 0, j = 0, minDist = Integer.MAX_VALUE;
        while (i < l1.size() && j < l2.size()) {
            minDist = Math.min(minDist, Math.abs(l1.get(i) - l2.get(j)));
            if (l1.get(i) < l2.get(j)) i++;
            else j++;
        }
        return minDist;
    }
}
```

245. Shortest Word Distance III [Medium]

📋 Problem Statement

Given an array of strings wordsDict and two strings word1 and word2, return the shortest distance between these two words. Note that word1 and word2 may be the same word.

Example:

Input: wordsDict = ["practice","makes","perfect","coding","makes"], word1 = "makes", word2 = "coding"

Output: 1

💡 **Approach:** **Similar to #243 but handle word1==word2 separately: in that case, track previous and current occurrence of the same word and find min gap between consecutive occurrences.**

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

☕ Java Implementation

```
public class Solution {

    public int shortestWordDistance(String[] wordsDict, String word1, String word2)
{

        int i1 = -1, i2 = -1, minDist = Integer.MAX_VALUE;

        boolean same = word1.equals(word2);

        for (int i = 0; i < wordsDict.length; i++) {

            if (wordsDict[i].equals(word1)) {

                if (same) {

                    if (i1 != -1) minDist = Math.min(minDist, i - i1);

                    i1 = i;

                } else {

                    i1 = i;

                    if (i2 != -1) minDist = Math.min(minDist, Math.abs(i1 - i2));

                }

            } else if (!same && wordsDict[i].equals(word2)) {

                i2 = i;

                if (i1 != -1) minDist = Math.min(minDist, Math.abs(i1 - i2));

            }

        }

        return minDist;

    }

}
```

📂 Nested List Weight Sum (LinkedIn Special)

339. Nested List Weight Sum [Easy]

📋 Problem Statement

You are given a nested list of integers nestedList. Each element is either an integer, or a list whose elements may also be integers or other lists. The depth of an integer is the number of lists that it is inside of. Return the sum of each integer in nestedList multiplied by its depth.

Example:

Input: [[1,1],2,[1,1]]

Output: 10

Explanation: Four 1's at depth 2 (4*2=8) and one 2 at depth 1 (2*1=2).

💡 **Approach: DFS with depth parameter. For integers, add value*depth. For lists, recurse with depth+1.**

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

☕ **Java Implementation**

```
public class Solution {
    public int depthSum(List<NestedInteger> nestedList) {
        return dfs(nestedList, 1);
    }
    private int dfs(List<NestedInteger> list, int depth) {
        int sum = 0;
        for (NestedInteger ni : list) {
            if (ni.isInteger()) {
                sum += ni.getInteger() * depth;
            } else {
                sum += dfs(ni.getList(), depth + 1);
            }
        }
        return sum;
    }
}
```

364. Nested List Weight Sum II [Medium]

📋 Problem Statement

Same as #339 but deeper integers receive lower weight. The weight of an integer is maxDepth - depth + 1.

Example:

Input: [[1,1],2,[1,1]]

Output: 8

Explanation: Four 1s at depth 2 get weight 1 (4*1=4), one 2 at depth 1 gets weight 2 (2*2=4). Total=8.

💡 **Approach:** **Two-pass DFS: first find maxDepth, then compute weighted sum using (maxDepth - depth + 1) as the weight.**

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

☕ Java Implementation

```
public class Solution {
    int maxDepth = 0;

    public int depthSumInverse(List<NestedInteger> nestedList) {
        findMaxDepth(nestedList, 1);
        return dfs(nestedList, 1);
    }

    private void findMaxDepth(List<NestedInteger> list, int depth) {
        maxDepth = Math.max(maxDepth, depth);
        for (NestedInteger ni : list) {
            if (!ni.isInteger()) findMaxDepth(ni.getList(), depth + 1);
        }
    }

    private int dfs(List<NestedInteger> list, int depth) {
        int sum = 0;
        for (NestedInteger ni : list) {
            if (ni.isInteger()) {
                sum += ni.getInteger() * (maxDepth - depth + 1);
            } else {
                sum += dfs(ni.getList(), depth + 1);
            }
        }
        return sum;
    }
}
```

📂 Hard Problems

301. Remove Invalid Parentheses [Hard]

📋 Problem Statement

Given a string s that contains parentheses and letters, remove the minimum number of invalid parentheses to make the input string valid. Return all possible results in any order.

Example:

Input: s = "()())()"

Output: ["(())()", "()()()"]

💡 **Approach: BFS: explore all strings by removing one parenthesis at a time. The first level where a valid string is found gives the minimum removals. Use a visited set to avoid duplicates.**

⏱️ **Complexity: Time: O(n * 2^n) | Space: O(n * 2^n)**

☕ Java Implementation

```
public class Solution {
    public List<String> removeInvalidParentheses(String s) {
        List<String> result = new ArrayList<>();
        Set<String> visited = new HashSet<>();
        Queue<String> queue = new LinkedList<>();
        queue.offer(s);
        visited.add(s);
        boolean found = false;
        while (!queue.isEmpty()) {
            String curr = queue.poll();
            if (isValid(curr)) { result.add(curr); found = true; }
            if (found) continue;
            for (int i = 0; i < curr.length(); i++) {
                if (curr.charAt(i) != '(' && curr.charAt(i) != ')') continue;
                String next = curr.substring(0, i) + curr.substring(i + 1);
                if (!visited.contains(next)) { queue.offer(next);
visited.add(next); }
            }
        }
        return result;
    }
    private boolean isValid(String s) {
        int count = 0;
        for (char c : s.toCharArray()) {
            if (c == '(') count++;
            else if (c == ')' && --count < 0) return false;
        }
        return count == 0;
    }
}
```

269. Alien Dictionary [Hard]

📋 Problem Statement

There is a new alien language that uses the English alphabet. Given a list of words sorted lexicographically by the rules of this new language, return the characters in the alien language in any valid order. If no valid order exists, return "".

Example:

Input: words = ["wrt","wrf","er","ett","rftt"]

Output: "wertf"

💡 **Approach: Topological sort (Kahn's BFS). Build a directed graph by comparing adjacent words to extract character ordering. Run BFS using in-degree counts.**

⏱️ **Complexity: Time: O(C) where C = total chars | Space: O(1) (26 chars)**

☕ Java Implementation

```
public class Solution {
    public String alienOrder(String[] words) {
        Map<Character, Set<Character>> adj = new HashMap<>();
        Map<Character, Integer> inDegree = new HashMap<>();
        for (String w : words) for (char c : w.toCharArray()) {
            adj.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];
            int len = Math.min(w1.length(), w2.length());
            if (w1.length() > w2.length() && w1.startsWith(w2)) return "";
            for (int j = 0; j < len; j++) {
                if (w1.charAt(j) != w2.charAt(j)) {
                    char from = w1.charAt(j), to = w2.charAt(j);
                    if (!adj.get(from).contains(to)) {
                        adj.get(from).add(to);
                        inDegree.put(to, inDegree.get(to) + 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 next : adj.get(c)) {
                inDegree.put(next, inDegree.get(next) - 1);
                if (inDegree.get(next) == 0) q.offer(next);
            }
        }
```

149. Max Points on a Line [Hard]

📋 Problem Statement

Given an array of points where points[i] = [xi, yi], return the maximum number of points that lie on the same straight line.

Example:

Input: points = [[1,1],[2,2],[3,3]]

Output: 3

💡 **Approach:** **For each point, compute the slope (as a reduced fraction gcd) to every other point and store in a HashMap. The max frequency + 1 (the anchor point) gives the answer.**

⏱️ **Complexity:** **Time: O(n²) | Space: O(n)**

☕ Java Implementation

```
public class Solution {
    public int maxPoints(int[][] points) {
        int n = points.length;
        if (n < 3) return n;
        int max = 2;
        for (int i = 0; i < n; i++) {
            Map<String, Integer> slopeMap = new HashMap<>();
            for (int j = i + 1; j < n; j++) {
                int dx = points[j][0] - points[i][0];
                int dy = points[j][1] - points[i][1];
                int g = gcd(Math.abs(dx), Math.abs(dy));
                if (g != 0) { dx /= g; dy /= g; }
                if (dx < 0) { dx = -dx; dy = -dy; }
                String key = dx + "," + dy;
                slopeMap.merge(key, 1, Integer::sum);
                max = Math.max(max, slopeMap.get(key) + 1);
            }
        }
        return max;
    }
    private int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
}
```