```java
// ═══════════════════════════════════════════════════════════════════
// WISE ROUND 2 — DS & ALGO IN JAVA (Complete Practice Set)
// All confirmed + likely questions with Java implementations
// ═══════════════════════════════════════════════════════════════════
//
// WHY JAVA?
// - Glassdoor May 2025: "brush up Java knowledge e.g. ConcurrentHashMaps"
// - Wise backend = Java microservices
// - HackerRank supports Java — you CAN choose Java if more comfortable
// - For Lead/Manager role, showing Java depth = full-stack credibility
//
// WISE'S OWN PAGE: "We don't care what programming language you use,
// so feel free to use the one you're most comfortable with."
//
// ═══════════════════════════════════════════════════════════════════

import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;

// ═══════════════════════════════════════════════════════════════════
// PATTERN 1: HASHMAP / HASHSET
// ═══════════════════════════════════════════════════════════════════

// ─── LC 1: Two Sum ─── [EASY] ★★★
// Time: O(n) | Space: O(n)
class TwoSum {
    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[]{};
    }
}

// ─── LC 49: Group Anagrams ─── [MEDIUM] ★★★
// Time: O(n × k log k) | Space: O(n)
class GroupAnagrams {
    public List<List<String>> groupAnagrams(String[] strs) {
        Map<String, List<String>> map = new HashMap<>();
        for (String s : strs) {
            char[] chars = s.toCharArray();
            Arrays.sort(chars);
            String key = new String(chars);
            map.computeIfAbsent(key, k -> new ArrayList<>()).add(s);
        }
        return new ArrayList<>(map.values());
    }
}
```

```java
// ─── LC 217: Contains Duplicate ─── [EASY] ★★
// Time: O(n) | Space: O(n)
class ContainsDuplicate {
    public boolean containsDuplicate(int[] nums) {
        Set<Integer> seen = new HashSet<>();
        for (int num : nums) {
            if (!seen.add(num)) return true;  // add returns false if
             already present
        }
        return false;
    }
}

// ─── LC 242: Valid Anagram ─── [EASY] ★★
// Time: O(n) | Space: O(1) — fixed 26 chars
class ValidAnagram {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) return false;
        int[] freq = new int[26];
        for (int i = 0; i < s.length(); i++) {
            freq[s.charAt(i) - 'a']++;
            freq[t.charAt(i) - 'a']--;
        }
        for (int f : freq) {
            if (f != 0) return false;
        }
        return true;
    }
}

// ─── LC 387: First Unique Character ─── [EASY] ★★
// Time: O(n) | Space: O(1)
class FirstUniqueChar {
    public int firstUniqChar(String s) {
        int[] freq = new int[26];
        for (char c : s.toCharArray()) freq[c - 'a']++;
        for (int i = 0; i < s.length(); i++) {
            if (freq[s.charAt(i) - 'a'] == 1) return i;
        }
        return -1;
    }
}

// ─── LC 560: Subarray Sum Equals K ─── [MEDIUM] ★★★
// "Find consecutive transactions summing to target"
// Time: O(n) | Space: O(n)
class SubarraySum {
    public int subarraySum(int[] nums, int k) {
        Map<Integer, Integer> prefixMap = new HashMap<>();
        prefixMap.put(0, 1);
        int count = 0, prefixSum = 0;
        for (int num : nums) {
            prefixSum += num;
            count += prefixMap.getOrDefault(prefixSum - k, 0);
            prefixMap.merge(prefixSum, 1, Integer::sum);
```

```
        }
        return count;
    }
}

// ─── LC 347: Top K Frequent Elements ─── [MEDIUM] ★★
// Time: O(n) | Space: O(n) — bucket sort approach
class TopKFrequent {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> freq = new HashMap<>();
        for (int n : nums) freq.merge(n, 1, Integer::sum);

        // Bucket sort: index = frequency
        @SuppressWarnings("unchecked")
        List<Integer>[] buckets = new List[nums.length + 1];
        for (var entry : freq.entrySet()) {
            int f = entry.getValue();
            if (buckets[f] == null) buckets[f] = new ArrayList<>();
            buckets[f].add(entry.getKey());
        }

        int[] result = new int[k];
        int idx = 0;
        for (int i = buckets.length - 1; i >= 0 && idx < k; i--) {
            if (buckets[i] != null) {
                for (int num : buckets[i]) {
                    result[idx++] = num;
                    if (idx == k) break;
                }
            }
        }
        return result;
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 2: ARRAYS & TWO POINTERS
// ═══════════════════════════════════════════════════════════════

// ─── LC 15: Three Sum ─── [MEDIUM] ★★★
// Time: O(n²) | Space: O(1)
class ThreeSum {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> result = new ArrayList<>();
        for (int i = 0; i < nums.length - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            int l = i + 1, r = nums.length - 1;
            while (l < r) {
                int sum = nums[i] + nums[l] + nums[r];
                if (sum == 0) {
                    result.add(Arrays.asList(nums[i], nums[l], nums[r]));
                    while (l < r && nums[l] == nums[l + 1]) l++;
                    while (l < r && nums[r] == nums[r - 1]) r--;
```

```
                    l++; r--;
                } else if (sum < 0) l++;
                else r--;
            }
        }
        return result;
    }
}

// ─── LC 121: Best Time to Buy/Sell Stock ─── [EASY] ★★★
// Time: O(n) | Space: O(1)
class MaxProfit {
    public int maxProfit(int[] prices) {
        int minPrice = Integer.MAX_VALUE, maxProfit = 0;
        for (int price : prices) {
            minPrice = Math.min(minPrice, price);
            maxProfit = Math.max(maxProfit, price - minPrice);
        }
        return maxProfit;
    }
}

// ─── LC 238: Product Except Self ─── [MEDIUM] ★★
// Time: O(n) | Space: O(1) excluding output
class ProductExceptSelf {
    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;
    }
}

// ─── LC 53: Maximum Subarray (Kadane's) ─── [MEDIUM] ★★★
// Time: O(n) | Space: O(1)
class MaxSubArray {
    public int maxSubArray(int[] nums) {
        int maxSum = nums[0], curSum = nums[0];
        for (int i = 1; i < nums.length; i++) {
            curSum = Math.max(nums[i], curSum + nums[i]);
            maxSum = Math.max(maxSum, curSum);
        }
        return maxSum;
    }
}

// ─── LC 11: Container With Most Water ─── [MEDIUM] ★★
// Time: O(n) | Space: O(1)
class MaxArea {
```

```
    public int maxArea(int[] height) {
        int l = 0, r = height.length - 1, maxA = 0;
        while (l < r) {
            maxA = Math.max(maxA, Math.min(height[l], height[r]) * (r - l));
            if (height[l] < height[r]) l++;
            else r--;
        }
        return maxA;
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 3: INTERVALS (CONFIRMED by Wise blog)
// ═══════════════════════════════════════════════════════════════

// ─── LC 56: Merge Intervals ─── [MEDIUM] ★★★ CONFIRMED
// Time: O(n log n) | Space: O(n)
class MergeIntervals {
    public int[][] merge(int[][] intervals) {
        Arrays.sort(intervals, (a, b) -> a[0] - b[0]);
        List<int[]> result = new ArrayList<>();
        result.add(intervals[0]);
        for (int i = 1; i < intervals.length; i++) {
            int[] last = result.get(result.size() - 1);
            if (intervals[i][0] <= last[1]) {
                last[1] = Math.max(last[1], intervals[i][1]);
            } else {
                result.add(intervals[i]);
            }
        }
        return result.toArray(new int[0][]);
    }
}

// ─── LC 986: Interval List Intersections ─── [MEDIUM] ★★★ CONFIRMED
// Time: O(m+n) | Space: O(m+n)
class IntervalIntersection {
    public int[][] intervalIntersection(int[][] A, int[][] B) {
        List<int[]> result = new ArrayList<>();
        int i = 0, j = 0;
        while (i < A.length && j < B.length) {
            int lo = Math.max(A[i][0], B[j][0]);
            int hi = Math.min(A[i][1], B[j][1]);
            if (lo <= hi) result.add(new int[]{lo, hi});
            if (A[i][1] < B[j][1]) i++;
            else j++;
        }
        return result.toArray(new int[0][]);
    }
}

// ─── LC 57: Insert Interval ─── [MEDIUM] ★★
// Time: O(n) | Space: O(n)
class InsertInterval {
```

```java
    public int[][] insert(int[][] intervals, int[] newInterval) {
        List<int[]> result = new ArrayList<>();
        int i = 0;
        while (i < intervals.length && intervals[i][1] < newInterval[0])
            result.add(intervals[i++]);
        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);
        while (i < intervals.length) result.add(intervals[i++]);
        return result.toArray(new int[0][]);
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 4: BACKTRACKING (Combination Sum CONFIRMED)
// ═══════════════════════════════════════════════════════════════

// ─── LC 39: Combination Sum ─── [MEDIUM] ★★★ CONFIRMED
// Time: O(2^t) | Space: O(target)
class CombinationSum {
    public List<List<Integer>> combinationSum(int[] candidates, int target)
     {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(candidates, target, 0, new ArrayList<>(), result);
        return result;
    }

    private void backtrack(int[] candidates, int remaining, int start,
                           List<Integer> current, List<List<Integer>>
                            result) {
        if (remaining == 0) { result.add(new ArrayList<>(current)); return;
         }
        if (remaining < 0) return;
        for (int i = start; i < candidates.length; i++) {
            current.add(candidates[i]);
            backtrack(candidates, remaining - candidates[i], i, current,
             result);
            current.remove(current.size() - 1);
        }
    }
}

// ─── LC 40: Combination Sum II ─── [MEDIUM] ★★★
// Time: O(2^n) | Space: O(n)
class CombinationSum2 {
    public List<List<Integer>> combinationSum2(int[] candidates, int
     target) {
        Arrays.sort(candidates);
        List<List<Integer>> result = new ArrayList<>();
        backtrack(candidates, target, 0, new ArrayList<>(), result);
        return result;
```

```
    }

    private void backtrack(int[] nums, int remaining, int start,
                           List<Integer> current, List<List<Integer>>
                            result) {
        if (remaining == 0) { result.add(new ArrayList<>(current)); return;
         }
        if (remaining < 0) return;
        for (int i = start; i < nums.length; i++) {
            if (i > start && nums[i] == nums[i - 1]) continue; // skip dupes
            current.add(nums[i]);
            backtrack(nums, remaining - nums[i], i + 1, current, result);
            current.remove(current.size() - 1);
        }
    }
}

// ─── LC 78: Subsets ─── [MEDIUM] ★★
class Subsets {
    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);
        }
    }
}

// ─── LC 46: Permutations ─── [MEDIUM] ★★
class Permutations {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        boolean[] used = new boolean[nums.length];
        backtrack(nums, used, new ArrayList<>(), result);
        return result;
    }

    private void backtrack(int[] nums, boolean[] used, List<Integer>
     current,
                           List<List<Integer>> result) {
        if (current.size() == nums.length) { result.add(new
         ArrayList<>(current)); return; }
        for (int i = 0; i < nums.length; i++) {
            if (used[i]) continue;
            used[i] = true;
            current.add(nums[i]);
            backtrack(nums, used, current, result);
```

```
            current.remove(current.size() - 1);
            used[i] = false;
        }
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 5: STACK / QUEUE
// ═══════════════════════════════════════════════════════════════

// ─── LC 20: Valid Parentheses ─── [EASY] ★★★
class ValidParentheses {
    public boolean isValid(String s) {
        Deque<Character> stack = new ArrayDeque<>();
        Map<Character, Character> pairs = Map.of(')', '(', ']', '[', '}',
         '{');
        for (char c : s.toCharArray()) {
            if (pairs.containsKey(c)) {
                if (stack.isEmpty() || stack.pop() != pairs.get(c)) return
                 false;
            } else {
                stack.push(c);
            }
        }
        return stack.isEmpty();
    }
}

// ─── LC 155: Min Stack ─── [MEDIUM] ★★★
class MinStack {
    private Deque<int[]> stack = new ArrayDeque<>(); // [value, currentMin]

    public void push(int val) {
        int curMin = stack.isEmpty() ? val : Math.min(val, stack.peek()[1]);
        stack.push(new int[]{val, curMin});
    }
    public void pop() { stack.pop(); }
    public int top() { return stack.peek()[0]; }
    public int getMin() { return stack.peek()[1]; }
}

// ─── LC 739: Daily Temperatures ─── [MEDIUM] ★★
// "How many days until a better exchange rate?"
class DailyTemperatures {
    public int[] dailyTemperatures(int[] temperatures) {
        int[] result = new int[temperatures.length];
        Deque<Integer> stack = new ArrayDeque<>(); // indices
        for (int i = 0; i < temperatures.length; i++) {
            while (!stack.isEmpty() && temperatures[i] >
             temperatures[stack.peek()]) {
                int j = stack.pop();
                result[j] = i - j;
            }
            stack.push(i);
```

```
        }
        return result;
    }
}


// ═══════════════════════════════════════════════════════════════════
// PATTERN 6: LINKED LIST
// ═══════════════════════════════════════════════════════════════════

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

// ─── LC 206: Reverse Linked List ─── [EASY] ★★★
class ReverseList {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null, cur = head;
        while (cur != null) {
            ListNode next = cur.next;
            cur.next = prev;
            prev = cur;
            cur = next;
        }
        return prev;
    }
}

// ─── LC 21: Merge Two Sorted Lists ─── [EASY] ★★★
class MergeTwoLists {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0), cur = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val <= l2.val) { cur.next = l1; l1 = l1.next; }
            else { cur.next = l2; l2 = l2.next; }
            cur = cur.next;
        }
        cur.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }
}

// ─── LC 141: Linked List Cycle ─── [EASY] ★★
class HasCycle {
    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;
    }
```

```java
}

// ─── LC 19: Remove Nth Node From End ─── [MEDIUM] ★★
class RemoveNthFromEnd {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(0);
        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;
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 7: BINARY SEARCH
// ═══════════════════════════════════════════════════════════════

// ─── LC 704: Binary Search ─── [EASY] ★★★
class BinarySearch {
    public int search(int[] nums, int target) {
        int l = 0, r = nums.length - 1;
        while (l <= r) {
            int mid = l + (r - l) / 2;
            if (nums[mid] == target) return mid;
            else if (nums[mid] < target) l = mid + 1;
            else r = mid - 1;
        }
        return -1;
    }
}

// ─── LC 33: Search in Rotated Sorted Array ─── [MEDIUM] ★★
class SearchRotated {
    public int search(int[] nums, int target) {
        int l = 0, r = nums.length - 1;
        while (l <= r) {
            int mid = l + (r - l) / 2;
            if (nums[mid] == target) return mid;
            if (nums[l] <= nums[mid]) {
                if (target >= nums[l] && target < nums[mid]) r = mid - 1;
                else l = mid + 1;
            } else {
                if (target > nums[mid] && target <= nums[r]) l = mid + 1;
                else r = mid - 1;
            }
        }
        return -1;
    }
}


// ═══════════════════════════════════════════════════════════════
```

```
// PATTERN 8: TREES
// ═══════════════════════════════════════════════════════════════

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

// ─── LC 104: Maximum Depth ─── [EASY] ★★★
class MaxDepth {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;
        return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
    }
}

// ─── LC 98: Validate BST ─── [MEDIUM] ★★★
class ValidBST {
    public boolean isValidBST(TreeNode root) {
        return validate(root, null, null);
    }

    private boolean validate(TreeNode node, Integer lo, Integer hi) {
        if (node == null) return true;
        if (lo != null && node.val <= lo) return false;
        if (hi != null && node.val >= hi) return false;
        return validate(node.left, lo, node.val) && validate(node.right,
         node.val, hi);
    }
}

// ─── LC 102: Level Order Traversal ─── [MEDIUM] ★★
class LevelOrder {
    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;
    }
}

// ─── LC 236: Lowest Common Ancestor ─── [MEDIUM] ★★
```

```
class LCA {
    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;
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 9: GRAPH / BFS / DFS
// ═══════════════════════════════════════════════════════════════

// ─── LC 200: Number of Islands ─── [MEDIUM] ★★★
class NumIslands {
    public int numIslands(char[][] grid) {
        int count = 0;
        for (int r = 0; r < grid.length; r++) {
            for (int c = 0; c < grid[0].length; c++) {
                if (grid[r][c] == '1') {
                    count++;
                    dfs(grid, r, c);
                }
            }
        }
        return count;
    }

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

// ─── WISE-SPECIFIC: Currency Graph BFS ─── ★★★★
// "Convert USD to GBP via intermediate currencies"
class CurrencyConverter {
    public double convert(Map<String, List<double[]>> graph, // graph
     stores [neighborIdx, rate]
                          Map<String, Integer> currencyIdx,
                          String from, String to, double amount) {
        if (from.equals(to)) return amount;
        if (!currencyIdx.containsKey(from) || !currencyIdx.containsKey(to))
         return -1;

        Set<String> visited = new HashSet<>();
        Queue<double[]> queue = new LinkedList<>(); // [currencyIdx,
         accumulatedRate]
        queue.offer(new double[]{currencyIdx.get(from), 1.0});
```

```
        visited.add(from);

        // Alternative: simpler adjacency list version
        Map<String, Map<String, Double>> adj = new HashMap<>();
        // populate adj...

        Queue<String> bfsQueue = new LinkedList<>();
        Map<String, Double> rates = new HashMap<>();
        bfsQueue.offer(from);
        rates.put(from, 1.0);

        while (!bfsQueue.isEmpty()) {
            String curr = bfsQueue.poll();
            if (curr.equals(to)) return amount * rates.get(to);
            for (var entry : adj.getOrDefault(curr, Map.of()).entrySet()) {
                if (!rates.containsKey(entry.getKey())) {
                    rates.put(entry.getKey(), rates.get(curr) *
                     entry.getValue());
                    bfsQueue.offer(entry.getKey());
                }
            }
        }
        return -1;
    }
}

// ─── LC 399: Evaluate Division (Graph BFS — identical pattern!) ─── ★★★
// This IS the currency conversion problem on LeetCode!
class EvaluateDivision {
    public double[] calcEquation(List<List<String>> equations, double[]
     values,
                                 List<List<String>> queries) {
        Map<String, Map<String, Double>> graph = new HashMap<>();
        for (int i = 0; i < equations.size(); i++) {
            String a = equations.get(i).get(0), b = equations.get(i).get(1);
            graph.computeIfAbsent(a, k -> new HashMap<>()).put(b,
             values[i]);
            graph.computeIfAbsent(b, k -> new HashMap<>()).put(a, 1.0 /
             values[i]);
        }

        double[] result = new double[queries.size()];
        for (int i = 0; i < queries.size(); i++) {
            result[i] = bfs(graph, queries.get(i).get(0),
             queries.get(i).get(1));
        }
        return result;
    }

    private double bfs(Map<String, Map<String, Double>> graph, String src,
     String dst) {
        if (!graph.containsKey(src) || !graph.containsKey(dst)) return -1.0;
        if (src.equals(dst)) return 1.0;

        Queue<String> queue = new LinkedList<>();
```

```
        Map<String, Double> dist = new HashMap<>();
        queue.offer(src);
        dist.put(src, 1.0);

        while (!queue.isEmpty()) {
            String curr = queue.poll();
            for (var entry : graph.get(curr).entrySet()) {
                if (!dist.containsKey(entry.getKey())) {
                    dist.put(entry.getKey(), dist.get(curr) *
                     entry.getValue());
                    if (entry.getKey().equals(dst)) return dist.get(dst);
                    queue.offer(entry.getKey());
                }
            }
        }
        return -1.0;
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 10: DYNAMIC PROGRAMMING
// ═══════════════════════════════════════════════════════════════

// ─── LC 322: Coin Change ─── [MEDIUM] ★★★ (Fintech!)
class CoinChange {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (coin <= i) dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
}

// ─── LC 70: Climbing Stairs ─── [EASY] ★★
class ClimbStairs {
    public int climbStairs(int n) {
        if (n <= 2) return n;
        int a = 1, b = 2;
        for (int i = 3; i <= n; i++) { int t = a + b; a = b; b = t; }
        return b;
    }
}

// ─── LC 198: House Robber ─── [MEDIUM] ★★
class HouseRobber {
    public int rob(int[] nums) {
        int prev2 = 0, prev1 = 0;
        for (int num : nums) {
            int temp = Math.max(prev1, prev2 + num);
```

```
            prev2 = prev1;
            prev1 = temp;
        }
        return prev1;
    }
}

// ─── LC 300: Longest Increasing Subsequence ─── [MEDIUM] ★★
class LIS {
    public int lengthOfLIS(int[] nums) {
        List<Integer> tails = new ArrayList<>();
        for (int num : nums) {
            int pos = Collections.binarySearch(tails, num);
            if (pos < 0) pos = -(pos + 1);
            if (pos == tails.size()) tails.add(num);
            else tails.set(pos, num);
        }
        return tails.size();
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 11: STRING MANIPULATION
// ═══════════════════════════════════════════════════════════════

// ─── LC 3: Longest Substring Without Repeating ─── [MEDIUM] ★★★
class LengthOfLongestSubstring {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> charIdx = new HashMap<>();
        int maxLen = 0, start = 0;
        for (int end = 0; end < s.length(); end++) {
            char c = s.charAt(end);
            if (charIdx.containsKey(c) && charIdx.get(c) >= start) {
                start = charIdx.get(c) + 1;
            }
            charIdx.put(c, end);
            maxLen = Math.max(maxLen, end - start + 1);
        }
        return maxLen;
    }
}

// ─── LC 5: Longest Palindromic Substring ─── [MEDIUM] ★★
class LongestPalindrome {
    private int start = 0, maxLen = 0;

    public String longestPalindrome(String s) {
        for (int i = 0; i < s.length(); i++) {
            expand(s, i, i);     // Odd
            expand(s, i, i + 1); // Even
        }
        return s.substring(start, start + maxLen);
    }
```

```
    private void expand(String s, int l, int r) {
        while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
            if (r - l + 1 > maxLen) { start = l; maxLen = r - l + 1; }
            l--; r++;
        }
    }
}


// ═══════════════════════════════════════════════════════════════
// PATTERN 12: DESIGN PROBLEMS (HIGH PRIORITY)
// ═══════════════════════════════════════════════════════════════

// ─── LC 146: LRU Cache ─── [MEDIUM] ★★★★ CONFIRMED
class LRUCache {
    private final int capacity;
    private final Map<Integer, Node> cache = new HashMap<>();
    private final Node head = new Node(0, 0); // dummy
    private final Node tail = new Node(0, 0); // dummy

    static class Node {
        int key, value;
        Node prev, next;
        Node(int key, int value) { this.key = key; this.value = value; }
    }

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

    public int get(int key) {
        if (!cache.containsKey(key)) return -1;
        Node node = cache.get(key);
        moveToHead(node);
        return node.value;
    }

    public void put(int key, int value) {
        if (cache.containsKey(key)) {
            Node node = cache.get(key);
            node.value = value;
            moveToHead(node);
        } else {
            Node newNode = new Node(key, value);
            cache.put(key, newNode);
            addAfterHead(newNode);
            if (cache.size() > capacity) {
                Node lru = tail.prev;
                removeNode(lru);
                cache.remove(lru.key);
            }
        }
    }
```

```
    private void addAfterHead(Node node) {
        node.prev = head; node.next = head.next;
        head.next.prev = node; head.next = node;
    }

    private void removeNode(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private void moveToHead(Node node) {
        removeNode(node);
        addAfterHead(node);
    }
}


// ─── Circuit Breaker (CONFIRMED Glassdoor) ─── ★★★★
class CircuitBreaker {
    enum State { CLOSED, OPEN, HALF_OPEN }

    private final int failureThreshold;
    private final long failureWindowMs;
    private final long recoveryTimeoutMs;

    private State state = State.CLOSED;
    private final List<Long> failureTimestamps = new ArrayList<>();
    private long lastOpenedAt = 0;

    public CircuitBreaker(int failureThreshold, long failureWindowMs, long
     recoveryTimeoutMs) {
        this.failureThreshold = failureThreshold;
        this.failureWindowMs = failureWindowMs;
        this.recoveryTimeoutMs = recoveryTimeoutMs;
    }

    public synchronized boolean canExecute() {
        switch (state) {
            case CLOSED: return true;
            case OPEN:
                if (System.currentTimeMillis() - lastOpenedAt >=
                 recoveryTimeoutMs) {
                    state = State.HALF_OPEN;
                    return true;
                }
                return false;
            case HALF_OPEN: return true;
            default: return false;
        }
    }

    public synchronized void recordSuccess() {
        if (state == State.HALF_OPEN) {
            state = State.CLOSED;
```

```
            failureTimestamps.clear();
        }
    }

    public synchronized void recordFailure() {
        long now = System.currentTimeMillis();
        if (state == State.CLOSED) {
            failureTimestamps.add(now);
            failureTimestamps.removeIf(t -> now - t > failureWindowMs);
            if (failureTimestamps.size() >= failureThreshold) {
                state = State.OPEN;
                lastOpenedAt = now;
            }
        } else if (state == State.HALF_OPEN) {
            state = State.OPEN;
            lastOpenedAt = now;
        }
    }

    public State getState() { return state; }
}


// ─── LC 208: Implement Trie ─── [MEDIUM] ★★
class Trie {
    private final TrieNode root = new TrieNode();

    static class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isEnd = false;
    }

    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 = find(word);
        return node != null && node.isEnd;
    }

    public boolean startsWith(String prefix) {
        return find(prefix) != null;
    }

    private TrieNode find(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
```

```
            int idx = c - 'a';
            if (node.children[idx] == null) return null;
            node = node.children[idx];
        }
        return node;
    }
}


// ═══════════════════════════════════════════════════════════════
// JAVA-SPECIFIC: COLLECTIONS & CONCURRENCY
// (Glassdoor: "brush up Java knowledge e.g. ConcurrentHashMaps")
// ═══════════════════════════════════════════════════════════════

/*
 ═══════════════════════════════════════════════════════════════
 JAVA COLLECTIONS CHEAT SHEET — KNOW THESE FOR WISE
 ═══════════════════════════════════════════════════════════════

 LIST:
 ├── ArrayList     — O(1) access, O(n) insert/remove at middle
 ├── LinkedList    — O(n) access, O(1) insert/remove at ends
 └── CopyOnWriteArrayList — thread-safe, snapshot reads

 SET:
 ├── HashSet       — O(1) add/remove/contains, unordered
 ├── LinkedHashSet — O(1) ops, insertion-ordered
 ├── TreeSet       — O(log n) ops, sorted
 └── ConcurrentSkipListSet — thread-safe sorted set

 MAP:
 ├── HashMap       — O(1) get/put, unordered, NOT thread-safe
 ├── LinkedHashMap — O(1) ops, insertion-ordered (good for LRU!)
 ├── TreeMap       — O(log n) ops, sorted by key
 ├── ConcurrentHashMap — thread-safe HashMap ← WISE ASKED THIS
 └── Hashtable     — legacy, synchronized (don't use)

 QUEUE:
 ├── LinkedList    — implements Queue interface
 ├── ArrayDeque    — faster than LinkedList for stack/queue
 ├── PriorityQueue — min-heap by default
 ├── ConcurrentLinkedQueue   — thread-safe non-blocking
 └── LinkedBlockingQueue     — thread-safe blocking

 KEY JAVA TIPS:
 ──────────────────────────────────────────────────────────────
 • HashMap vs ConcurrentHashMap:
   HashMap: NOT thread-safe, allows null key/value
   ConcurrentHashMap: thread-safe, NO null key/value
   ConcurrentHashMap uses segments for lock striping

 • LinkedHashMap for LRU Cache:
   new LinkedHashMap<>(capacity, 0.75f, true) {
       protected boolean removeEldestEntry(Map.Entry eldest) {
           return size() > capacity;
```

```
    }
  }
  accessOrder=true makes it LRU-ordered!
```

- ArrayDeque vs Stack:
  Stack is legacy (extends Vector, synchronized)
  Use ArrayDeque for both stack and queue operations

- PriorityQueue:
  Min-heap by default. For max-heap:
  new PriorityQueue<>(Collections.reverseOrder())
  new PriorityQueue<>((a, b) -> b - a)

- Collections.unmodifiableList/Map/Set:
  Returns immutable view (throws on modification)

- List.of(), Map.of(), Set.of() (Java 9+):
  Creates immutable collections directly

```
═══════════════════════════════════════════════════════════════
JAVA CONCURRENCY — WHAT WISE INTERVIEWERS PROBE
═══════════════════════════════════════════════════════════════

THREAD SAFETY:
├── synchronized keyword — method or block level lock
├── volatile — visibility guarantee, no atomicity
├── AtomicInteger/Long — lock-free atomic operations
├── ReentrantLock — explicit lock with tryLock, fairness
└── ReadWriteLock — concurrent reads, exclusive writes

CONCURRENT DATA STRUCTURES:
├── ConcurrentHashMap — segment-based locking
│   Key methods: putIfAbsent, compute, merge
│   ✅ Use for: shared caches, rate counters
│
├── CopyOnWriteArrayList — copy on mutation
│   ✅ Use for: frequently read, rarely modified lists
│   ❌ Don't use for: frequently modified lists (expensive copies)
│
└── BlockingQueue — producer-consumer pattern
    ├── LinkedBlockingQueue — unbounded
    ├── ArrayBlockingQueue — bounded
    └── PriorityBlockingQueue — sorted

EXECUTORS:
├── Executors.newFixedThreadPool(n)     — fixed thread count
├── Executors.newCachedThreadPool()      — dynamic thread count
├── Executors.newSingleThreadExecutor()  — sequential execution
└── Executors.newScheduledThreadPool(n)  — scheduled tasks

COMPLETABLE FUTURE (Java 8+):
CompletableFuture.supplyAsync(() -> fetchRate("USD", "EUR"))
    .thenApply(rate -> rate * amount)
```

```
    .thenAccept(converted -> updateUI(converted))
    .exceptionally(ex -> { handleError(ex); return null; });

VIRTUAL THREADS (Java 21+):
Thread.startVirtualThread(() -> { fetchRate("USD", "EUR"); });
// Lightweight, no thread pool needed, ideal for I/O-bound tasks

Q: "What's the difference between HashMap and ConcurrentHashMap?"
A: HashMap is not thread-safe and allows null keys/values.
   ConcurrentHashMap is thread-safe using segment-based locking
   (Java 7) or node-level locking (Java 8+), does NOT allow nulls.
   For read-heavy workloads, ConcurrentHashMap has almost no
   performance overhead vs HashMap. For write-heavy, there's some
   overhead from lock contention.

Q: "When would you use synchronized vs ConcurrentHashMap?"
A: ConcurrentHashMap for simple key-value thread safety.
   synchronized for complex atomic operations on multiple data
   structures that need to stay consistent together.

Q: "Explain volatile"
A: volatile ensures visibility — when one thread writes to a
   volatile variable, other threads immediately see the new value.
   But volatile does NOT guarantee atomicity. count++ on a volatile
   int is still a race condition (read-modify-write). Use
   AtomicInteger for atomic increment.

================================================================
JAVA STREAMS — MODERN JAVA IDIOMS
================================================================

// Filter + Map + Collect
List<String> names = transactions.stream()
    .filter(t -> t.getAmount() > 1000)
    .map(Transaction::getRecipientName)
    .distinct()
    .collect(Collectors.toList());

// Group by currency
Map<String, List<Transaction>> byCurrency = transactions.stream()
    .collect(Collectors.groupingBy(Transaction::getCurrency));

// Sum amounts
BigDecimal total = transactions.stream()
    .map(Transaction::getAmount)
    .reduce(BigDecimal.ZERO, BigDecimal::add);

// Find max
Optional<Transaction> largest = transactions.stream()
    .max(Comparator.comparing(Transaction::getAmount));

// Parallel stream (use for CPU-bound, NOT I/O-bound)
long count = transactions.parallelStream()
    .filter(t -> t.getStatus() == Status.COMPLETED)
    .count();
```

```
*/


// ═══════════════════════════════════════════════════════════════════
// SLIDING WINDOW
// ═══════════════════════════════════════════════════════════════════

// ─── LC 239: Sliding Window Maximum ─── [HARD] ★★
class MaxSlidingWindow {
    public int[] maxSlidingWindow(int[] nums, int k) {
        Deque<Integer> deque = new ArrayDeque<>();
        int[] result = new int[nums.length - k + 1];
        for (int i = 0; i < nums.length; i++) {
            while (!deque.isEmpty() && deque.peekFirst() <= i - k)
             deque.pollFirst();
            while (!deque.isEmpty() && nums[deque.peekLast()] <= nums[i])
             deque.pollLast();
            deque.offerLast(i);
            if (i >= k - 1) result[i - k + 1] = nums[deque.peekFirst()];
        }
        return result;
    }
}


// ═══════════════════════════════════════════════════════════════════
// HEAP / PRIORITY QUEUE
// ═══════════════════════════════════════════════════════════════════

// ─── LC 215: Kth Largest Element ─── [MEDIUM] ★★
class KthLargest {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(); // min-heap
         of size k
        for (int num : nums) {
            minHeap.offer(num);
            if (minHeap.size() > k) minHeap.poll();
        }
        return minHeap.peek();
    }
}

// ─── LC 23: Merge K Sorted Lists ─── [HARD] ★★
class MergeKLists {
    public ListNode mergeKLists(ListNode[] lists) {
        PriorityQueue<ListNode> pq = new PriorityQueue<>((a, b) -> a.val -
         b.val);
        for (ListNode l : lists) if (l != null) pq.offer(l);
        ListNode dummy = new ListNode(0), cur = dummy;
        while (!pq.isEmpty()) {
            ListNode node = pq.poll();
            cur.next = node;
            cur = cur.next;
            if (node.next != null) pq.offer(node.next);
```

```
        }
        return dummy.next;
    }
}


// ============================================================================
// BONUS: LC 399 = CURRENCY CONVERSION (PRACTICE THIS!)
// ============================================================================
// LC 399 "Evaluate Division" is LITERALLY the currency conversion
// problem. If you can solve this on HackerRank in Java, you're
// ready for any Wise graph question.
//
// Example:
// equations = [["USD","EUR"],["EUR","GBP"]]
// values = [0.92, 0.86]
// queries = [["USD","GBP"]]
// → USD → EUR (×0.92) → GBP (×0.86) = 0.7912
//
// This is BFS on a weighted directed graph.
// SEE EvaluateDivision class above for full implementation.
```