Top 100 LeetCode Interview Questions - Complete Guide

Table of Contents

Array Problems

1. Two Sum

Problem Summary: Given an array of integers and a target sum, return indices of two numbers that add up to the target.

Solution Summary: Use a HashMap to store complements while iterating through the array.

Java Code:

```
java
```

```java
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[]{};
}
```

Time Complexity: O(n) - single pass

Backtracking

56. Subsets

Problem Summary: Generate all possible subsets of a given set.

Solution Summary: Use backtracking to include/exclude each element.

Java Code:

```java
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    backtrack(result, new ArrayList<>(), nums, 0);
    return result;
}

private void backtrack(List<List<Integer>> result, List<Integer> path,
                      int[] nums, int start) {
    result.add(new ArrayList<>(path));

    for (int i = start; i < nums.length; i++) {
        path.add(nums[i]);
        backtrack(result, path, nums, i + 1);
        path.remove(path.size() - 1);
    }
}
```

Time Complexity: O(2^n) - each element can be included or excluded

---

57. Combination Sum

Problem Summary: Find all combinations that sum to target, allowing repeated use.

Solution Summary: Use backtracking with same element reusable.

Java Code:

```java
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new ArrayList<>();
    Arrays.sort(candidates);
    backtrack(result, new ArrayList<>(), candidates, target, 0);
    return result;
}

private void backtrack(List<List<Integer>> result, List<Integer> path,
                      int[] candidates, int remain, int start) {
    if (remain < 0) return;
    if (remain == 0) {
        result.add(new ArrayList<>(path));
        return;
    }

    for (int i = start; i < candidates.length; i++) {
        path.add(candidates[i]);
        backtrack(result, path, candidates, remain - candidates[i], i);
        path.remove(path.size() - 1);
    }
}
```

Time Complexity: O(N^(T/M)) where N is candidates, T is target, M is minimal value

---

58. Permutations

Problem Summary: Generate all possible permutations of given numbers.

Solution Summary: Use backtracking with visited array to track used elements.

Java Code:

```java
public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    boolean[] used = new boolean[nums.length];
    backtrack(result, new ArrayList<>(), nums, used);
    return result;
}

private void backtrack(List<List<Integer>> result, List<Integer> path,
                      int[] nums, boolean[] used) {
    if (path.size() == nums.length) {
        result.add(new ArrayList<>(path));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        if (used[i]) continue;

        path.add(nums[i]);
        used[i] = true;
        backtrack(result, path, nums, used);
        path.remove(path.size() - 1);
        used[i] = false;
    }
}
```

Time Complexity: O(n! * n) - n! permutations, each takes O(n) to build

---

59. Word Search

Problem Summary: Check if word exists in 2D character grid.

Solution Summary: Use DFS backtracking from each cell.

Java Code:

```java
```

```java
public boolean exist(char[][] board, String word) {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (dfs(board, word, i, j, 0)) {
                return true;
            }
        }
    }
    return false;
}

private boolean dfs(char[][] board, String word, int i, int j, int index) {
    if (index == word.length()) return true;
    if (i < 0 || i >= board.length || j < 0 || j >= board[0].length ||
        board[i][j] != word.charAt(index)) {
        return false;
    }

    char temp = board[i][j];
    board[i][j] = '#'; // mark visited

    boolean found = dfs(board, word, i + 1, j, index + 1) ||
                    dfs(board, word, i - 1, j, index + 1) ||
                    dfs(board, word, i, j + 1, index + 1) ||
                    dfs(board, word, i, j - 1, index + 1);

    board[i][j] = temp; // restore
    return found;
}
```

Time Complexity: O(M * N * 4^L) where M*N is board size, L is word length

---

60. Palindrome Partitioning

Problem Summary: Partition string such that every substring is a palindrome.

Solution Summary: Use backtracking with palindrome checking.

Java Code:

```
java
```

```
public List<List<String>> partition(String s) {
    List<List<String>> result = new ArrayList<>();
    backtrack(result, new ArrayList<>(), s, 0);
    return result;
}

private void backtrack(List<List<String>> result, List<String> path,
                      String s, int start) {
    if (start >= s.length()) {
        result.add(new ArrayList<>(path));
        return;
    }

    for (int end = start; end < s.length(); end++) {
        if (isPalindrome(s, start, end)) {
            path.add(s.substring(start, end + 1));
            backtrack(result, path, s, end + 1);
            path.remove(path.size() - 1);
        }
    }
}

private boolean isPalindrome(String s, int low, int high) {
    while (low < high) {
        if (s.charAt(low++) != s.charAt(high--)) return false;
    }
    return true;
}
```

Time Complexity: O(2^n * n) - 2^n possible partitions, O(n) to check palindrome

---

Two Pointers

61. Valid Palindrome

Problem Summary: Check if string is palindrome ignoring non-alphanumeric characters.

Solution Summary: Use two pointers from both ends.

Java Code:

```
java
```

```
public boolean isPalindrome(String s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {
            left++;
        }
        while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {
            right--;
        }

        if (Character.toLowerCase(s.charAt(left)) !=
            Character.toLowerCase(s.charAt(right))) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

Time Complexity: O(n) - single pass with two pointers

---

62. Two Sum II - Input Array Is Sorted

Problem Summary: Find two numbers in sorted array that sum to target.

Solution Summary: Use two pointers from both ends.

Java Code:

```
java
```

```
public int[] twoSum(int[] numbers, int target) {
    int left = 0, right = numbers.length - 1;

    while (left < right) {
        int sum = numbers[left] + numbers[right];
        if (sum == target) {
            return new int[]{left + 1, right + 1};
        } else if (sum < target) {
            left++;
        } else {
            right--;
        }
    }
    return new int[]{-1, -1};
}
```

Time Complexity: O(n) - single pass with two pointers

---

63. 3Sum Closest

Problem Summary: Find three numbers whose sum is closest to target.

Solution Summary: Sort array and use two pointers for each element.

Java Code:

```
java
```

```java
public int threeSumClosest(int[] nums, int target) {
    Arrays.sort(nums);
    int closest = nums[0] + nums[1] + nums[2];

    for (int i = 0; i < nums.length - 2; i++) {
        int left = i + 1, right = nums.length - 1;

        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];
            if (Math.abs(target - sum) < Math.abs(target - closest)) {
                closest = sum;
            }

            if (sum < target) {
                left++;
            } else {
                right--;
            }
        }
    }
    return closest;
}
```

Time Complexity: O(n^2) - O(n log n) sorting + O(n^2) two pointers

---

64. Remove Duplicates from Sorted Array

Problem Summary: Remove duplicates from sorted array in-place.

Solution Summary: Use two pointers to track unique elements.

Java Code:

```
java
```

```
public int removeDuplicates(int[] nums) {
    if (nums.length == 0) return 0;

    int slow = 0;
    for (int fast = 1; fast < nums.length; fast++) {
        if (nums[fast] != nums[slow]) {
            slow++;
            nums[slow] = nums[fast];
        }
    }
    return slow + 1;
}
```

Time Complexity: O(n) - single pass

---

65. Trapping Rain Water

Problem Summary: Calculate how much water can be trapped after raining.

Solution Summary: Use two pointers with left and right max heights.

Java Code:

```
java
```

```
public int trap(int[] height) {
    if (height == null || height.length == 0) return 0;

    int left = 0, right = height.length - 1;
    int leftMax = 0, rightMax = 0;
    int water = 0;

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

Time Complexity: O(n) - single pass with two pointers

Binary Search

66. Binary Search

Problem Summary: Search for target in sorted array.

Solution Summary: Classic binary search implementation.

Java Code:

```
java
```

```java
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;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Time Complexity: O(log n) - binary search

---

67. Find First and Last Position

Problem Summary: Find starting and ending position of target in sorted array.

Solution Summary: Use binary search to find leftmost and rightmost positions.

Java Code:

```
java
```

```
public int[] searchRange(int[] nums, int target) {
    int left = findFirst(nums, target);
    int right = findLast(nums, target);
    return new int[]{left, right};
}

private int findFirst(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            result = mid;
            right = mid - 1;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return result;
}

private int findLast(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    int result = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) {
            result = mid;
            left = mid + 1;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return result;
}
```

Time Complexity: O(log n) - two binary searches

68. Search Insert Position

Problem Summary: Find index where target should be inserted in sorted array.

Solution Summary: Binary search for insertion point.

Java Code:

```java
public int searchInsert(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;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return left;
}
```

Time Complexity: O(log n) - binary search

69. Search in Rotated Sorted Array II

Problem Summary: Search in rotated sorted array with duplicates.

Solution Summary: Binary search with handling of duplicates.

Java Code:

```java
```

```java
public boolean 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 true;

        if (nums[left] == nums[mid] && nums[mid] == nums[right]) {
            left++;
            right--;
        } else if (nums[left] <= nums[mid]) {
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else {
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return false;
}
```

Time Complexity: O(log n) average, O(n) worst case with many duplicates

---

70. Find Peak Element

Problem Summary: Find a peak element (greater than neighbors).

Solution Summary: Binary search towards higher neighbor.

Java Code:

```
java
```

```
public int findPeakElement(int[] nums) {
    int left = 0, right = nums.length - 1;

    while (left < right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] > nums[mid + 1]) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

Time Complexity: O(log n) - binary search

Stack and Queue

71. Valid Parentheses (Revisited)

Problem Summary: Check if brackets are properly matched.

Solution Summary: Use stack to match opening and closing brackets.

Java Code:

```
java
```

```java
public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();

    for (char c : s.toCharArray()) {
        if (c == '(' || c == '[' || c == '{') {
            stack.push(c);
        } else {
            if (stack.isEmpty()) return false;
            char top = stack.pop();
            if ((c == ')' && top != '(') ||
                (c == ']' && top != '[') ||
                (c == '}' && top != '{')) {
                return false;
            }
        }
    }
    return stack.isEmpty();
}
```

Time Complexity: O(n) - single pass through string

---

72. Min Stack

Problem Summary: Design stack with push, pop, top, and getMin in O(1).

Solution Summary: Use auxiliary stack to track minimum values.

Java Code:

```
java
```

```
class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
        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();
    }
}
```

Time Complexity: O(1) for all operations

---

73. Evaluate Reverse Polish Notation

Problem Summary: Evaluate arithmetic expression in postfix notation.

Solution Summary: Use stack to process operands and operators.

Java Code:

```
java
```

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

    for (String token : tokens) {
        if (token.equals("+")) {
            stack.push(stack.pop() + stack.pop());
        } else if (token.equals("-")) {
            int b = stack.pop();
            int a = stack.pop();
            stack.push(a - b);
        } else if (token.equals("*")) {
            stack.push(stack.pop() * stack.pop());
        } else if (token.equals("/")) {
            int b = stack.pop();
            int a = stack.pop();
            stack.push(a / b);
        } else {
            stack.push(Integer.parseInt(token));
        }
    }
    return stack.pop();
}
```

Time Complexity: O(n) - process each token once

---

74. Generate Parentheses

Problem Summary: Generate all valid combinations of n pairs of parentheses.

Solution Summary: Use backtracking with stack-like counting.

Java Code:

```
java
```

```
public List<String> generateParenthesis(int n) {
    List<String> result = new ArrayList<>();
    backtrack(result, "", 0, 0, n);
    return result;
}

private void backtrack(List<String> result, String current,
                      int open, int close, int max) {
    if (current.length() == max * 2) {
        result.add(current);
        return;
    }

    if (open < max) {
        backtrack(result, current + "(", open + 1, close, max);
    }
    if (close < open) {
        backtrack(result, current + ")", open, close + 1, max);
    }
}
```

Time Complexity: O(4^n / √n) - Catalan number

---

75. Daily Temperatures

Problem Summary: Find how many days until warmer temperature.

Solution Summary: Use stack to track indices of previous temperatures.

Java Code:

```
java
```

```java
public int[] dailyTemperatures(int[] temperatures) {
    int[] result = new int[temperatures.length];
    Stack<Integer> stack = new Stack<>();

    for (int i = 0; i < temperatures.length; i++) {
        while (!stack.isEmpty() &&
               temperatures[i] > temperatures[stack.peek()]) {
            int index = stack.pop();
            result[index] = i - index;
        }
        stack.push(i);
    }
    return result;
}
```

Time Complexity: O(n) - each element pushed and popped once

---

76. Car Fleet

Problem Summary: Calculate how many car fleets reach target.

Solution Summary: Sort by position and use stack to track fleets.

Java Code:

```
java
```

```
public int carFleet(int target, int[] position, int[] speed) {
    int n = position.length;
    double[][] cars = new double[n][2];

    for (int i = 0; i < n; i++) {
        cars[i][0] = position[i];
        cars[i][1] = (double)(target - position[i]) / speed[i];
    }

    Arrays.sort(cars, (a, b) -> Double.compare(b[0], a[0]));

    int fleets = 0;
    double slowest = 0;

    for (double[] car : cars) {
        if (car[1] > slowest) {
            slowest = car[1];
            fleets++;
        }
    }
    return fleets;
}
```

Time Complexity: O(n log n) - sorting dominates

---

77. Largest Rectangle in Histogram

Problem Summary: Find largest rectangle area in histogram.

Solution Summary: Use stack to track indices of increasing heights.

Java Code:

```
java
```

```
public int largestRectangleArea(int[] heights) {
    Stack<Integer> stack = new Stack<>();
    int maxArea = 0;

    for (int i = 0; i <= heights.length; i++) {
        int currentHeight = i == heights.length ? 0 : heights[i];

        while (!stack.isEmpty() && heights[stack.peek()] > currentHeight) {
            int height = heights[stack.pop()];
            int width = stack.isEmpty() ? i : i - stack.peek() - 1;
            maxArea = Math.max(maxArea, height * width);
        }
        stack.push(i);
    }
    return maxArea;
}
```

Time Complexity: O(n) - each element pushed and popped once

Additional Important Problems

78. Median of Two Sorted Arrays

Problem Summary: Find median of two sorted arrays in O(log(m+n)) time.

Solution Summary: Use binary search to partition arrays equally.

Java Code:

```
java
```

```
public double findMedianSortedArrays(int[] nums1, int[] nums2) {
    if (nums1.length > nums2.length) {
        return findMedianSortedArrays(nums2, nums1);
    }

    int x = nums1.length;
    int y = nums2.length;
    int low = 0;
    int high = x;

    while (low <= high) {
        int cutX = (low + high) / 2;
        int cutY = (x + y + 1) / 2 - cutX;

        int maxLeftX = cutX == 0 ? Integer.MIN_VALUE : nums1[cutX - 1];
        int minRightX = cutX == x ? Integer.MAX_VALUE : nums1[cutX];

        int maxLeftY = cutY == 0 ? Integer.MIN_VALUE : nums2[cutY - 1];
        int minRightY = cutY == y ? Integer.MAX_VALUE : nums2[cutY];

        if (maxLeftX <= minRightY && maxLeftY <= minRightX) {
            if ((x + y) % 2 == 0) {
                return (Math.max(maxLeftX, maxLeftY) + Math.min(minRightX, minRightY)) / 2.0;
            } else {
                return Math.max(maxLeftX, maxLeftY);
            }
        } else if (maxLeftX > minRightY) {
            high = cutX - 1;
        } else {
            low = cutX + 1;
        }
    }
    return 1.0;
}
```

Time Complexity: O(log(min(m,n))) - binary search on smaller array

---

79. Merge k Sorted Lists

Problem Summary: Merge k sorted linked lists into one sorted list.

Solution Summary: Use divide and conquer with merge two lists.

Java Code:

```java
public ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0) return null;
    return mergeKListsHelper(lists, 0, lists.length - 1);
}

private ListNode mergeKListsHelper(ListNode[] lists, int start, int end) {
    if (start == end) return lists[start];
    if (start < end) {
        int mid = start + (end - start) / 2;
        ListNode l1 = mergeKListsHelper(lists, start, mid);
        ListNode l2 = mergeKListsHelper(lists, mid + 1, end);
        return mergeTwoLists(l1, l2);
    }
    return null;
}

private ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode curr = dummy;

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

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

Time Complexity: O(n log k) where n is total nodes, k is number of lists

80. Regular Expression Matching

Problem Summary: Implement regular expression matching with '.' and '*'.

Solution Summary: Use dynamic programming or recursion with memoization.

Java Code:

```java
public boolean isMatch(String s, String p) {
    Boolean[][] memo = new Boolean[s.length() + 1][p.length() + 1];
    return helper(s, p, 0, 0, memo);
}

private boolean helper(String s, String p, int i, int j, Boolean[][] memo) {
    if (memo[i][j] != null) return memo[i][j];

    boolean result;
    if (j == p.length()) {
        result = i == s.length();
    } else {
        boolean firstMatch = i < s.length() &&
                    (p.charAt(j) == s.charAt(i) || p.charAt(j) == '.');

        if (j + 1 < p.length() && p.charAt(j + 1) == '*') {
            result = helper(s, p, i, j + 2, memo) ||
                    (firstMatch && helper(s, p, i + 1, j, memo));
        } else {
            result = firstMatch && helper(s, p, i + 1, j + 1, memo);
        }
    }

    memo[i][j] = result;
    return result;
}
```

Time Complexity: O(m * n) where m, n are lengths of string and pattern

---

81. Wildcard Matching

Problem Summary: Implement wildcard pattern matching with '?' and '*'.

Solution Summary: Use dynamic programming to handle wildcards.

Java Code:

```java
public boolean isMatch(String s, String p) {
    int m = s.length(), n = p.length();
    boolean[][] dp = new boolean[m + 1][n + 1];
    dp[0][0] = true;

    for (int j = 1; j <= n; j++) {
        if (p.charAt(j - 1) == '*') {
            dp[0][j] = dp[0][j - 1];
        }
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (p.charAt(j - 1) == '*') {
                dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
            } else if (p.charAt(j - 1) == '?' || s.charAt(i - 1) == p.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1];
            }
        }
    }
    return dp[m][n];
}
```

Time Complexity: O(m * n) where m, n are string lengths

---

82. Edit Distance

Problem Summary: Find minimum operations to transform one string to another.

Solution Summary: Use dynamic programming with three operations: insert, delete, replace.

Java Code:

```java
```

```
public int minDistance(String word1, String word2) {
    int m = word1.length(), n = word2.length();
    int[][] dp = new int[m + 1][n + 1];

    for (int i = 0; i <= m; i++) dp[i][0] = i;
    for (int j = 0; j <= n; j++) dp[0][j] = j;

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = 1 + Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]),
                                      dp[i - 1][j - 1]);
            }
        }
    }
    return dp[m][n];
}
```

Time Complexity: O(m * n) where m, n are string lengths

---

83. Sliding Window Maximum

Problem Summary: Find maximum element in each sliding window of size k.

Solution Summary: Use deque to maintain indices in decreasing order of values.

Java Code:

```
java
```

```java
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;
}
```

Time Complexity: O(n) - each element added and removed at most once

---

84. Meeting Rooms II

Problem Summary: Find minimum conference rooms needed for given meeting intervals.

Solution Summary: Sort start and end times separately, use two pointers.

Java Code:

```
java
```

```java
public int minMeetingRooms(int[][] intervals) {
    int[] starts = new int[intervals.length];
    int[] ends = new int[intervals.length];

    for (int i = 0; i < intervals.length; i++) {
        starts[i] = intervals[i][0];
        ends[i] = intervals[i][1];
    }

    Arrays.sort(starts);
    Arrays.sort(ends);

    int rooms = 0, endPtr = 0;

    for (int i = 0; i < starts.length; i++) {
        if (starts[i] >= ends[endPtr]) {
            endPtr++;
        } else {
            rooms++;
        }
    }
    return rooms;
}
```

Time Complexity: O(n log n) - sorting dominates

---

85. Insert Interval

Problem Summary: Insert new interval into sorted non-overlapping intervals.

Solution Summary: Find position and merge overlapping intervals.

Java Code:

```
java
```

```java
public int[][] insert(int[][] intervals, int[] newInterval) {
    List<int[]> result = new ArrayList<>();
    int i = 0;

    // Add intervals before newInterval
    while (i < intervals.length && intervals[i][1] < newInterval[0]) {
        result.add(intervals[i]);
        i++;
    }

    // Merge overlapping intervals
    while (i < intervals.length && intervals[i][0] <= newInterval[1]) {
        newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
        newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
        i++;
    }
    result.add(newInterval);

    // Add remaining intervals
    while (i < intervals.length) {
        result.add(intervals[i]);
        i++;
    }

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

Time Complexity: O(n) - single pass through intervals

---

86. Merge Intervals

Problem Summary: Merge overlapping intervals in array of intervals.

Solution Summary: Sort by start time and merge overlapping ones.

Java Code:

```
java
```

```
public int[][] merge(int[][] intervals) {
    if (intervals.length <= 1) return intervals;

    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

    List<int[]> result = new ArrayList<>();
    int[] current = intervals[0];
    result.add(current);

    for (int[] interval : intervals) {
        if (current[1] >= interval[0]) {
            current[1] = Math.max(current[1], interval[1]);
        } else {
            current = interval;
            result.add(current);
        }
    }

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

Time Complexity: O(n log n) - sorting dominates

---

87. Non-overlapping Intervals

Problem Summary: Find minimum intervals to remove to make rest non-overlapping.

Solution Summary: Sort by end time and greedily select non-overlapping intervals.

Java Code:

```java
```

```
public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals.length <= 1) return 0;

    Arrays.sort(intervals, (a, b) -> Integer.compare(a[1], b[1]));

    int count = 0;
    int end = intervals[0][1];

    for (int i = 1; i < intervals.length; i++) {
        if (intervals[i][0] < end) {
            count++;
        } else {
            end = intervals[i][1];
        }
    }
    return count;
}
```

Time Complexity: O(n log n) - sorting dominates

---

88. Top K Frequent Elements

Problem Summary: Find k most frequent elements in array.

Solution Summary: Use HashMap for frequency and min-heap for top k.

Java Code:

```
java
```

```java
public int[] topKFrequent(int[] nums, int k) {
    Map<Integer, Integer> count = new HashMap<>();
    for (int num : nums) {
        count.put(num, count.getOrDefault(num, 0) + 1);
    }

    PriorityQueue<int[]> heap = new PriorityQueue<>((a, b) -> a[1] - b[1]);

    for (Map.Entry<Integer, Integer> entry : count.entrySet()) {
        heap.offer(new int[]{entry.getKey(), entry.getValue()});
        if (heap.size() > k) {
            heap.poll();
        }
    }

    int[] result = new int[k];
    for (int i = 0; i < k; i++) {
        result[i] = heap.poll()[0];
    }
    return result;
}
```

Time Complexity: O(n log k) where n is array length

---

89. Kth Largest Element in Array

Problem Summary: Find kth largest element in unsorted array.

Solution Summary: Use quickselect algorithm or min-heap.

Java Code:

```
java
```

```java
public int findKthLargest(int[] nums, int k) {
    PriorityQueue<Integer> heap = new PriorityQueue<>();

    for (int num : nums) {
        heap.offer(num);
        if (heap.size() > k) {
            heap.poll();
        }
    }
    return heap.peek();
}
```

Time Complexity: O(n log k) - maintain heap of size k

---

90. Find Median from Data Stream

Problem Summary: Design data structure to find median from stream of integers.

Solution Summary: Use two heaps - max heap for smaller half, min heap for larger half.

Java Code:

```
java
```

```
class MedianFinder {
    private PriorityQueue<Integer> small; // max heap
    private PriorityQueue<Integer> large; // min heap

    public MedianFinder() {
        small = new PriorityQueue<>(Collections.reverseOrder());
        large = new PriorityQueue<>();
    }

    public void addNum(int num) {
        small.offer(num);
        large.offer(small.poll());

        if (small.size() < large.size()) {
            small.offer(large.poll());
        }
    }

    public double findMedian() {
        if (small.size() > large.size()) {
            return small.peek();
        } else {
            return (small.peek() + large.peek()) / 2.0;
        }
    }
}
```

Time Complexity: O(log n) for addNum, O(1) for findMedian

---

91. Alien Dictionary

Problem Summary: Derive the order of characters in alien language from sorted words.

Solution Summary: Build graph from character order and use topological sort.

Java Code:

```
java
```

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

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

    // Build graph
    for (int i = 0; i < words.length - 1; i++) {
        String first = words[i];
        String second = words[i + 1];

        if (first.startsWith(second) && first.length() > second.length()) {
            return "";
        }

        for (int j = 0; j < Math.min(first.length(), second.length()); j++) {
            char through array with O(1) HashMap operations

---

### 2. Best Time to Buy and Sell Stock
**Problem Summary:** Find the maximum profit from buying and selling a stock once.

**Solution Summary:** Track minimum price seen so far and calculate maximum profit at each step.

**Java Code:**
```java
public int maxProfit(int[] prices) {
    int minPrice = Integer.MAX_VALUE;
    int maxProfit = 0;
    for (int price : prices) {
        if (price < minPrice) {
            minPrice = price;
        } else if (price - minPrice > maxProfit) {
            maxProfit = price - minPrice;
        }
    }
~~~

```
    return maxProfit;
}
```

Time Complexity: O(n) - single pass through the array

---

3. Contains Duplicate

Problem Summary: Determine if an array contains any duplicate values.

Solution Summary: Use a HashSet to track seen elements.

Java Code:

```java
public boolean containsDuplicate(int[] nums) {
    Set<Integer> seen = new HashSet<>();
    for (int num : nums) {
        if (!seen.add(num)) {
            return true;
        }
    }
    return false;
}
```

Time Complexity: O(n) - single pass with O(1) HashSet operations

---

4. Product of Array Except Self

Problem Summary: Return an array where each element is the product of all elements except itself.

Solution Summary: Use two passes - left products then right products.

Java Code:

```java
```

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

Time Complexity: O(n) - two passes through the array

---

5. Maximum Subarray

Problem Summary: Find the contiguous subarray with the largest sum.

Solution Summary: Use Kadane's algorithm to track maximum sum ending at current position.

Java Code:

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

Time Complexity: O(n) - single pass through array

---

6. Maximum Product Subarray

Problem Summary: Find the contiguous subarray with the largest product.

Solution Summary: Track both maximum and minimum products due to negative numbers.

Java Code:

```java
public int maxProduct(int[] nums) {
    int maxProduct = nums[0];
    int maxHere = nums[0];
    int minHere = nums[0];

    for (int i = 1; i < nums.length; i++) {
        if (nums[i] < 0) {
            int temp = maxHere;
            maxHere = minHere;
            minHere = temp;
        }
        maxHere = Math.max(nums[i], maxHere * nums[i]);
        minHere = Math.min(nums[i], minHere * nums[i]);
        maxProduct = Math.max(maxProduct, maxHere);
    }
    return maxProduct;
}
```

Time Complexity: O(n) - single pass through array

---

7. Find Minimum in Rotated Sorted Array

Problem Summary: Find the minimum element in a rotated sorted array.

Solution Summary: Use binary search to find the rotation point.

Java Code:

```java
```

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

Time Complexity: O(log n) - binary search

---

8. Search in Rotated Sorted Array

Problem Summary: Search for a target in a rotated sorted array.

Solution Summary: Use binary search with additional logic to handle rotation.

Java Code:

```
java
```

```
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 (nums[left] <= target && target < nums[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else {
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return -1;
}
```

Time Complexity: O(log n) - binary search

---

9. 3Sum

Problem Summary: Find all unique triplets that sum to zero.

Solution Summary: Sort array and use two pointers for each element.

Java Code:

```
java
```

```
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 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;
}
```

Time Complexity: O(n^2) - O(n log n) sorting + O(n^2) two pointers

---

10. Container With Most Water

Problem Summary: Find two lines that together with x-axis forms a container holding the most water.

Solution Summary: Use two pointers from both ends, move the shorter line.

Java Code:

```
java
```

```java
public int maxArea(int[] height) {
    int left = 0, right = height.length - 1;
    int maxArea = 0;

    while (left < right) {
        int area = Math.min(height[left], height[right]) * (right - left);
        maxArea = Math.max(maxArea, area);

        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
    return maxArea;
}
```

Time Complexity: O(n) - single pass with two pointers

String Problems

11. Valid Anagram

Problem Summary: Determine if two strings are anagrams of each other.

Solution Summary: Count character frequencies and compare.

Java Code:

```
java
```

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

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

    for (int c : count) {
        if (c != 0) return false;
    }
    return true;
}
```

Time Complexity: O(n) - single pass through both strings

---

12. Valid Parentheses

Problem Summary: Determine if parentheses, brackets, and braces are properly matched.

Solution Summary: Use a stack to match opening and closing brackets.

Java Code:

```
java
```

```java
public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();
    Map<Character, Character> map = new HashMap<>();
    map.put(')', '(');
    map.put('}', '{');
    map.put(']', '[');

    for (char c : s.toCharArray()) {
        if (map.containsKey(c)) {
            if (stack.isEmpty() || stack.pop() != map.get(c)) {
                return false;
            }
        } else {
            stack.push(c);
        }
    }
    return stack.isEmpty();
}
```

Time Complexity: O(n) - single pass through string

---

13. Longest Substring Without Repeating Characters

Problem Summary: Find the length of the longest substring without repeating characters.

Solution Summary: Use sliding window with HashSet to track characters.

Java Code:

```
java
```

```java
public int lengthOfLongestSubstring(String s) {
    Set<Character> set = new HashSet<>();
    int left = 0, maxLen = 0;

    for (int right = 0; right < s.length(); right++) {
        while (set.contains(s.charAt(right))) {
            set.remove(s.charAt(left));
            left++;
        }
        set.add(s.charAt(right));
        maxLen = Math.max(maxLen, right - left + 1);
    }
    return maxLen;
}
```

Time Complexity: O(n) - each character visited at most twice

---

14. Longest Repeating Character Replacement

Problem Summary: Find longest substring with same character after replacing at most k characters.

Solution Summary: Use sliding window with character frequency map.

Java Code:

java

```java
public int characterReplacement(String s, int k) {
    int[] count = new int[26];
    int left = 0, maxCount = 0, maxLength = 0;

    for (int right = 0; right < s.length(); right++) {
        maxCount = Math.max(maxCount, ++count[s.charAt(right) - 'A']);

        if (right - left + 1 - maxCount > k) {
            count[s.charAt(left) - 'A']--;
            left++;
        }
        maxLength = Math.max(maxLength, right - left + 1);
    }
    return maxLength;
}
```

Time Complexity: O(n) - single pass through string

---

15. Minimum Window Substring

Problem Summary: Find minimum window substring containing all characters of target string.

Solution Summary: Use sliding window with character frequency maps.

Java Code:

```
java
```

```
public String minWindow(String s, String t) {
    Map<Character, Integer> need = new HashMap<>();
    Map<Character, Integer> window = new HashMap<>();

    for (char c : t.toCharArray()) {
        need.put(c, need.getOrDefault(c, 0) + 1);
    }

    int left = 0, right = 0, valid = 0;
    int start = 0, len = Integer.MAX_VALUE;

    while (right < s.length()) {
        char c = s.charAt(right);
        right++;

        if (need.containsKey(c)) {
            window.put(c, window.getOrDefault(c, 0) + 1);
            if (window.get(c).equals(need.get(c))) {
                valid++;
            }
        }

        while (valid == need.size()) {
            if (right - left < len) {
                start = left;
                len = right - left;
            }

            char d = s.charAt(left);
            left++;

            if (need.containsKey(d)) {
                if (window.get(d).equals(need.get(d))) {
                    valid--;
                }
                window.put(d, window.get(d) - 1);
            }
        }
    }
    return len == Integer.MAX_VALUE ? "" : s.substring(start, start + len);
}
```

Time Complexity: O(n) - each character visited at most twice

16. Group Anagrams

Problem Summary: Group strings that are anagrams of each other.

Solution Summary: Use sorted string as key in HashMap.

Java Code:

```java
public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> map = new HashMap<>();

    for (String str : strs) {
        char[] chars = str.toCharArray();
        Arrays.sort(chars);
        String key = String.valueOf(chars);

        map.computeIfAbsent(key, k -> new ArrayList<>()).add(str);
    }

    return new ArrayList<>(map.values());
}
```

Time Complexity: O(n * m log m) where n is number of strings, m is average length

17. Valid Palindrome

Problem Summary: Check if string is a palindrome considering only alphanumeric characters.

Solution Summary: Use two pointers from both ends, skip non-alphanumeric characters.

Java Code:

```java
```

```java
public boolean isPalindrome(String s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {
            left++;
        }
        while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {
            right--;
        }

        if (Character.toLowerCase(s.charAt(left)) !=
            Character.toLowerCase(s.charAt(right))) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

Time Complexity: O(n) - single pass with two pointers

---

18. Longest Palindromic Substring

Problem Summary: Find the longest palindromic substring.

Solution Summary: Expand around centers for odd and even length palindromes.

Java Code:

```
java
```

```
public String longestPalindrome(String s) {
    if (s == null || s.length() < 1) return "";
    int start = 0, end = 0;

    for (int i = 0; i < s.length(); i++) {
        int len1 = expandAroundCenter(s, i, i);
        int len2 = expandAroundCenter(s, i, i + 1);
        int len = Math.max(len1, len2);

        if (len > end - start) {
            start = i - (len - 1) / 2;
            end = i + len / 2;
        }
    }
    return s.substring(start, end + 1);
}

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

Time Complexity: O(n^2) - expand around each possible center

---

19. Palindromic Substrings

Problem Summary: Count the number of palindromic substrings.

Solution Summary: Expand around each possible center.

Java Code:

```
java
```

```java
public int countSubstrings(String s) {
    int count = 0;
    for (int i = 0; i < s.length(); i++) {
        count += expandAroundCenter(s, i, i);
        count += expandAroundCenter(s, i, i + 1);
    }
    return count;
}

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

Time Complexity: O(n^2) - expand around each possible center

---

20. Encode and Decode Strings

Problem Summary: Design an algorithm to encode and decode a list of strings.

Solution Summary: Use length prefix with delimiter.

Java Code:

```
java
```

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

public List<String> decode(String s) {
    List<String> result = new ArrayList<>();
    int i = 0;

    while (i < s.length()) {
        int delim = s.indexOf("#", i);
        int len = Integer.parseInt(s.substring(i, delim));
        result.add(s.substring(delim + 1, delim + 1 + len));
        i = delim + 1 + len;
    }
    return result;
}
```

Time Complexity: O(n) where n is total length of all strings

Linked List Problems

21. Reverse Linked List

Problem Summary: Reverse a singly linked list.

Solution Summary: Use three pointers to reverse links iteratively.

Java Code:

```
java
```

```
public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode curr = head;

    while (curr != null) {
        ListNode next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

Time Complexity: O(n) - single pass through list

---

22. Detect Cycle in Linked List

Problem Summary: Determine if a linked list has a cycle.

Solution Summary: Use Floyd's cycle detection (tortoise and hare).

Java Code:

```java
public boolean hasCycle(ListNode head) {
    if (head == null || head.next == null) return false;

    ListNode slow = head;
    ListNode fast = head.next;

    while (slow != fast) {
        if (fast == null || fast.next == null) return false;
        slow = slow.next;
        fast = fast.next.next;
    }
    return true;
}
```

Time Complexity: O(n) - at most n iterations

---

23. Merge Two Sorted Lists

Problem Summary: Merge two sorted linked lists into one sorted list.

Solution Summary: Use two pointers to compare and merge nodes.

Java Code:

```java
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode curr = dummy;

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

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

Time Complexity: O(m + n) where m, n are lengths of the lists

---

24. Remove Nth Node From End

Problem Summary: Remove the nth node from the end of a linked list.

Solution Summary: Use two pointers with n gap between them.

Java Code:

```java
```

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

    for (int i = 0; i <= n; i++) {
        first = first.next;
    }

    while (first != null) {
        first = first.next;
        second = second.next;
    }

    second.next = second.next.next;
    return dummy.next;
}
```

Time Complexity: O(L) where L is length of list

---

25. Reorder List

Problem Summary: Reorder list to L0→Ln→L1→Ln-1→L2→Ln-2→...

Solution Summary: Find middle, reverse second half, merge alternately.

Java Code:

```
java
```

```
public void reorderList(ListNode head) {
    if (head == null || head.next == null) return;

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

    // Reverse second half
    ListNode head2 = reverseList(slow.next);
    slow.next = null;

    // Merge
    ListNode head1 = head;
    while (head2 != null) {
        ListNode next1 = head1.next;
        ListNode next2 = head2.next;
        head1.next = head2;
        head2.next = next1;
        head1 = next1;
        head2 = next2;
    }
}

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

Time Complexity: O(n) - three passes through list

Tree Problems

26. Maximum Depth of Binary Tree

Problem Summary: Find the maximum depth of a binary tree.

Solution Summary: Use recursion to find max depth of left and right subtrees.

Java Code:

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

Time Complexity: O(n) - visit each node

---

35. Kth Smallest Element in BST

Problem Summary: Find the kth smallest element in a binary search tree.

Solution Summary: Use inorder traversal to visit nodes in sorted order.

Java Code:

```java
```

```
private int count = 0;
private int result = 0;

public int kthSmallest(TreeNode root, int k) {
    inorder(root, k);
    return result;
}

private void inorder(TreeNode node, int k) {
    if (node == null) return;

    inorder(node.left, k);
    count++;
    if (count == k) {
        result = node.val;
        return;
    }
    inorder(node.right, k);
}
```

Time Complexity: O(H + k) where H is height of tree

---

36. Lowest Common Ancestor of BST

Problem Summary: Find lowest common ancestor of two nodes in BST.

Solution Summary: Use BST property to navigate towards LCA.

Java Code:

```
java
```

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

Time Complexity: O(H) where H is height of tree

---

37. Implement Trie (Prefix Tree)

Problem Summary: Implement a trie with insert, search, and startsWith methods.

Solution Summary: Use tree structure where each node represents a character.

Java Code:

```
java
```

```
class Trie {
    class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isEndOfWord = false;
    }

    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    public void insert(String word) {
        TrieNode curr = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (curr.children[index] == null) {
                curr.children[index] = new TrieNode();
            }
            curr = curr.children[index];
        }
        curr.isEndOfWord = true;
    }

    public boolean search(String word) {
        TrieNode node = searchPrefix(word);
        return node != null && node.isEndOfWord;
    }

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

    private TrieNode searchPrefix(String word) {
        TrieNode curr = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (curr.children[index] == null) {
                return null;
            }
            curr = curr.children[index];
        }
        return curr;
    }
}
```

Time Complexity: O(m) for all operations where m is key length

---

38. Word Search II

Problem Summary: Find all words from dictionary that can be constructed from board.

Solution Summary: Build trie from dictionary, then DFS on board with trie traversal.

Java Code:

```
java
```

```
public List<String> findWords(char[][] board, String[] words) {
    TrieNode root = buildTrie(words);
    List<String> result = new ArrayList<>();

    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            dfs(board, i, j, root, result);
        }
    }
    return result;
}

private void dfs(char[][] board, int i, int j, TrieNode p, List<String> result) {
    char c = board[i][j];
    if (c == '#' || p.next[c - 'a'] == null) return;

    p = p.next[c - 'a'];
    if (p.word != null) {
        result.add(p.word);
        p.word = null; // avoid duplicates
    }

    board[i][j] = '#';
    if (i > 0) dfs(board, i - 1, j, p, result);
    if (j > 0) dfs(board, i, j - 1, p, result);
    if (i < board.length - 1) dfs(board, i + 1, j, p, result);
    if (j < board[0].length - 1) dfs(board, i, j + 1, p, result);
    board[i][j] = c;
}

class TrieNode {
    TrieNode[] next = new TrieNode[26];
    String word;
}

private TrieNode buildTrie(String[] words) {
    TrieNode root = new TrieNode();
    for (String w : words) {
        TrieNode p = root;
        for (char c : w.toCharArray()) {
            int i = c - 'a';
            if (p.next[i] == null) p.next[i] = new TrieNode();
            p = p.next[i];
        }
        p.word = w;
    }
    return root;
}
```

Time Complexity: O(M * N * 4^L) where M*N is board size, L is max word length

---

Graph Problems

39. Number of Islands

Problem Summary: Count the number of islands in a 2D binary grid.

Solution Summary: Use DFS or BFS to mark connected land cells.

Java Code:

```
java
```

```
public int numIslands(char[][] grid) {
    if (grid == null || grid.length == 0) return 0;

    int count = 0;
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[0].length; j++) {
            if (grid[i][j] == '1') {
                count++;
                dfs(grid, i, j);
            }
        }
    }
    return count;
}

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

    grid[i][j] = '0';
    dfs(grid, i + 1, j);
    dfs(grid, i - 1, j);
    dfs(grid, i, j + 1);
    dfs(grid, i, j - 1);
}
```

Time Complexity: O(M * N) where M*N is grid size

---

40. Clone Graph

Problem Summary: Deep clone an undirected graph.

Solution Summary: Use DFS/BFS with HashMap to track cloned nodes.

Java Code:

```
java
```

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

    Map<Node, Node> cloned = new HashMap<>();
    return dfs(node, cloned);
}

private Node dfs(Node node, Map<Node, Node> cloned) {
    if (cloned.containsKey(node)) {
        return cloned.get(node);
    }

    Node clone = new Node(node.val);
    cloned.put(node, clone);

    for (Node neighbor : node.neighbors) {
        clone.neighbors.add(dfs(neighbor, cloned));
    }

    return clone;
}
```

Time Complexity: O(V + E) where V is vertices, E is edges

---

41. Course Schedule

Problem Summary: Determine if you can finish all courses given prerequisites.

Solution Summary: Detect cycle in directed graph using DFS or topological sort.

Java Code:

```
java
```

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

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

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

    for (int i = 0; i < numCourses; i++) {
        if (state[i] == 0 && hasCycle(graph, state, i)) {
            return false;
        }
    }
    return true;
}

private boolean hasCycle(List<List<Integer>> graph, int[] state, int node) {
    if (state[node] == 1) return true; // cycle detected
    if (state[node] == 2) return false; // already processed

    state[node] = 1;
    for (int neighbor : graph.get(node)) {
        if (hasCycle(graph, state, neighbor)) {
            return true;
        }
    }
    state[node] = 2;
    return false;
}
```

Time Complexity: O(V + E) where V is courses, E is prerequisites

---

42. Pacific Atlantic Water Flow

Problem Summary: Find cells where water can flow to both Pacific and Atlantic oceans.

Solution Summary: DFS from ocean borders to find reachable cells.

Java Code:

```java
public List<List<Integer>> pacificAtlantic(int[][] heights) {
    int m = heights.length, n = heights[0].length;
    boolean[][] pacific = new boolean[m][n];
    boolean[][] atlantic = new boolean[m][n];

    // DFS from Pacific borders
    for (int i = 0; i < m; i++) {
        dfs(heights, pacific, i, 0, heights[i][0]);
        dfs(heights, atlantic, i, n - 1, heights[i][n - 1]);
    }
    for (int j = 0; j < n; j++) {
        dfs(heights, pacific, 0, j, heights[0][j]);
        dfs(heights, atlantic, m - 1, j, heights[m - 1][j]);
    }

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

private void dfs(int[][] heights, boolean[][] visited, int i, int j, int prevHeight) {
    if (i < 0 || i >= heights.length || j < 0 || j >= heights[0].length ||
        visited[i][j] || heights[i][j] < prevHeight) {
        return;
    }

    visited[i][j] = true;
    dfs(heights, visited, i + 1, j, heights[i][j]);
    dfs(heights, visited, i - 1, j, heights[i][j]);
    dfs(heights, visited, i, j + 1, heights[i][j]);
    dfs(heights, visited, i, j - 1, heights[i][j]);
}
```

Time Complexity: O(M * N) where M*N is matrix size

---

43. Graph Valid Tree

Problem Summary: Check if edges form a valid tree with n nodes.

Solution Summary: Check if graph is connected and has exactly n-1 edges.

Java Code:

java

```java
public boolean validTree(int n, int[][] edges) {
    if (edges.length != n - 1) return false;

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

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

    boolean[] visited = new boolean[n];
    dfs(graph, visited, 0);

    for (boolean v : visited) {
        if (!v) return false;
    }
    return true;
}

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

Time Complexity: O(V + E) where V is nodes, E is edges

---

44. Number of Connected Components

Problem Summary: Find number of connected components in undirected graph.

Solution Summary: Use DFS/BFS to count separate components.

Java Code:

```java
public int countComponents(int n, int[][] edges) {
    List<List<Integer>> graph = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        graph.add(new ArrayList<>());
    }

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

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

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

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

Time Complexity: O(V + E) where V is nodes, E is edges

Dynamic Programming

45. Climbing Stairs

Problem Summary: Count ways to climb n stairs taking 1 or 2 steps at a time.

Solution Summary: Use Fibonacci sequence - f(n) = f(n-1) + f(n-2).

Java Code:

```java
public int climbStairs(int n) {
    if (n <= 2) return n;

    int prev2 = 1, prev1 = 2;
    for (int i = 3; i <= n; i++) {
        int curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

Time Complexity: O(n) - single pass

---

46. Coin Change

Problem Summary: Find minimum coins needed to make amount.

Solution Summary: Use DP where dp[i] is min coins for amount i.

Java Code:

```java
```

```java
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];
}
```

Time Complexity: O(amount * coins.length)

---

47. Longest Increasing Subsequence

Problem Summary: Find length of longest increasing subsequence.

Solution Summary: Use DP with binary search for O(n log n) solution.

Java Code:

```
java

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();
}
```

Time Complexity: O(n log n) - binary search for each element

---

48. Longest Common Subsequence

Problem Summary: Find length of longest common subsequence between two strings.

Solution Summary: Use 2D DP where dp[i][j] is LCS length for first i chars of s1 and first j chars of s2.

Java Code:

java

```java
public int longestCommonSubsequence(String text1, String text2) {
    int m = text1.length(), n = text2.length();
    int[][] dp = new int[m + 1][n + 1];

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

Time Complexity: O(m * n) where m, n are string lengths

---

49. Word Break

Problem Summary: Check if string can be segmented using dictionary words.

Solution Summary: Use DP where dp[i] indicates if first i characters can be segmented.

Java Code:

java

```java
public boolean wordBreak(String s, List<String> wordDict) {
    Set<String> wordSet = new HashSet<>(wordDict);
    boolean[] dp = new boolean[s.length() + 1];
    dp[0] = true;

    for (int i = 1; i <= s.length(); i++) {
        for (int j = 0; j < i; j++) {
            if (dp[j] && wordSet.contains(s.substring(j, i))) {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[s.length()];
}
```

Time Complexity: O(n^3) in worst case due to substring operations

---

50. Combination Sum IV

Problem Summary: Find number of combinations that sum to target.

Solution Summary: Use DP where dp[i] is number of ways to make sum i.

Java Code:

```java
public int combinationSum4(int[] nums, int target) {
    int[] dp = new int[target + 1];
    dp[0] = 1;

    for (int i = 1; i <= target; i++) {
        for (int num : nums) {
            if (num <= i) {
                dp[i] += dp[i - num];
            }
        }
    }
    return dp[target];
}
```

Time Complexity: O(target * nums.length)

---

51. House Robber

Problem Summary: Find maximum money that can be robbed without robbing adjacent houses.

Solution Summary: Use DP where dp[i] is max money from first i houses.

Java Code:

```java
public int rob(int[] nums) {
    if (nums.length == 0) return 0;
    if (nums.length == 1) return nums[0];

    int prev2 = nums[0];
    int prev1 = Math.max(nums[0], nums[1]);

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

Time Complexity: O(n) - single pass

---

52. House Robber II

Problem Summary: Houses are arranged in circle - first and last are adjacent.

Solution Summary: Consider two cases: rob first house or rob last house.

Java Code:

```java
```

```
public int rob(int[] nums) {
    if (nums.length == 1) return nums[0];
    return Math.max(robLinear(nums, 0, nums.length - 2),
                    robLinear(nums, 1, nums.length - 1));
}

private int robLinear(int[] nums, int start, int end) {
    int prev2 = 0, prev1 = 0;
    for (int i = start; i <= end; i++) {
        int curr = Math.max(prev1, prev2 + nums[i]);
        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

Time Complexity: O(n) - two linear passes

---

53. Decode Ways

Problem Summary: Count ways to decode a string of digits to letters.

Solution Summary: Use DP considering single digit and two digit decodings.

Java Code:

```
java
```

```
public int numDecodings(String s) {
    if (s == null || s.length() == 0 || s.charAt(0) == '0') return 0;

    int prev2 = 1, prev1 = 1;

    for (int i = 1; i < s.length(); i++) {
        int curr = 0;

        if (s.charAt(i) != '0') {
            curr += prev1;
        }

        int twoDigit = Integer.parseInt(s.substring(i - 1, i + 1));
        if (twoDigit >= 10 && twoDigit <= 26) {
            curr += prev2;
        }

        prev2 = prev1;
        prev1 = curr;
    }
    return prev1;
}
```

Time Complexity: O(n) - single pass

---

54. Unique Paths

Problem Summary: Count unique paths from top-left to bottom-right in grid.

Solution Summary: Use DP where dp[i][j] is paths to cell (i,j).

Java Code:

```
java
```

```java
public int uniquePaths(int m, int n) {
    int[] dp = new int[n];
    Arrays.fill(dp, 1);

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

Time Complexity: O(m * n) - fill DP table

---

55. Jump Game

Problem Summary: Check if you can reach the last index from first index.

Solution Summary: Track the farthest reachable position.

Java Code:

```java
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;
}
```

Time Complexity: O(n) - single pass once

---

27. Same Tree

Problem Summary: Check if two binary trees are structurally identical.

Solution Summary: Recursively compare nodes and their children.

Java Code:

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

Time Complexity: O(min(m,n)) where m,n are sizes of trees

---

28. Invert Binary Tree

Problem Summary: Invert a binary tree (swap left and right children).

Solution Summary: Recursively swap left and right children of each node.

Java Code:

```java
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;
}
```

Time Complexity: O(n) - visit each node once

---

29. Binary Tree Maximum Path Sum

Problem Summary: Find the maximum sum path between any two nodes.

Solution Summary: For each node, calculate max gain from left/right subtrees.

Java Code:

```java
```

```
private int maxSum = Integer.MIN_VALUE;

public int maxPathSum(TreeNode root) {
    maxGain(root);
    return maxSum;
}

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

    int leftGain = Math.max(maxGain(node.left), 0);
    int rightGain = Math.max(maxGain(node.right), 0);

    int priceNewPath = node.val + leftGain + rightGain;
    maxSum = Math.max(maxSum, priceNewPath);

    return node.val + Math.max(leftGain, rightGain);
}
```

Time Complexity: O(n) - visit each node once

---

30. Binary Tree Level Order Traversal

Problem Summary: Return level order traversal of binary tree nodes.

Solution Summary: Use BFS with queue to traverse level by level.

Java Code:

```
java
```

```java
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 levelSize = queue.size();
        List<Integer> level = new ArrayList<>();

        for (int i = 0; i < levelSize; 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;
}
```

Time Complexity: O(n) - visit each node once

---

31. Serialize and Deserialize Binary Tree

Problem Summary: Design algorithm to serialize and deserialize binary tree.

Solution Summary: Use preorder traversal with null markers.

Java Code:

```
java
```

```
public String serialize(TreeNode root) {
    StringBuilder sb = new StringBuilder();
    serializeHelper(root, sb);
    return sb.toString();
}

private void serializeHelper(TreeNode node, StringBuilder sb) {
    if (node == null) {
        sb.append("null,");
        return;
    }
    sb.append(node.val).append(",");
    serializeHelper(node.left, sb);
    serializeHelper(node.right, sb);
}

public TreeNode deserialize(String data) {
    Queue<String> nodes = new LinkedList<>();
    nodes.addAll(Arrays.asList(data.split(",")));
    return deserializeHelper(nodes);
}

private TreeNode deserializeHelper(Queue<String> nodes) {
    String val = nodes.poll();
    if ("null".equals(val)) return null;

    TreeNode node = new TreeNode(Integer.parseInt(val));
    node.left = deserializeHelper(nodes);
    node.right = deserializeHelper(nodes);
    return node;
}
```

Time Complexity: O(n) for both serialize and deserialize

---

32. Subtree of Another Tree

Problem Summary: Check if tree t is subtree of tree s.

Solution Summary: For each node in s, check if subtree matches t.

Java Code:

```java
public boolean isSubtree(TreeNode s, TreeNode t) {
    if (s == null) return false;
    return isSameTree(s, t) || isSubtree(s.left, t) || isSubtree(s.right, t);
}

private boolean isSameTree(TreeNode s, TreeNode t) {
    if (s == null && t == null) return true;
    if (s == null || t == null) return false;
    if (s.val != t.val) return false;
    return isSameTree(s.left, t.left) && isSameTree(s.right, t.right);
}
```

Time Complexity: O(m*n) where m,n are sizes of trees

---

33. Construct Binary Tree from Preorder and Inorder

Problem Summary: Construct binary tree from preorder and inorder traversal arrays.

Solution Summary: Use preorder for root, inorder to split left/right subtrees.

Java Code:

```java
```

```
private int preorderIndex = 0;

public TreeNode buildTree(int[] preorder, int[] inorder) {
    Map<Integer, Integer> inorderMap = new HashMap<>();
    for (int i = 0; i < inorder.length; i++) {
        inorderMap.put(inorder[i], i);
    }
    return buildTreeHelper(preorder, inorderMap, 0, inorder.length - 1);
}

private TreeNode buildTreeHelper(int[] preorder, Map<Integer, Integer> inorderMap,
                                int left, int right) {
    if (left > right) return null;

    int rootVal = preorder[preorderIndex++];
    TreeNode root = new TreeNode(rootVal);

    int inorderIndex = inorderMap.get(rootVal);
    root.left = buildTreeHelper(preorder, inorderMap, left, inorderIndex - 1);
    root.right = buildTreeHelper(preorder, inorderMap, inorderIndex + 1, right);

    return root;
}
```

Time Complexity: O(n) - each node processed once

---

34. Validate Binary Search Tree

Problem Summary: Determine if binary tree is valid BST.

Solution Summary: Use inorder traversal or check bounds for each node.

Java Code:

```
java
```

```
public boolean isValidBST(TreeNode root) {
    return validate(root, null, null);
}

private boolean validate(TreeNode node, Integer low, Integer high) {
    if (node == null) return true;

    if ((low != null && node.val <= low) ||
        (high != null && node.val >= high)) {
        return false;
    }

    return validate(node.left, low, node.val) &&
           validate(node.right, node.val, high);
}
```

Time Complexity: O(n) - visit each node