PG-DAC Java Solutions - Complete Guide

TABLE OF CONTENTS

ARRAYS - BASIC LEVEL

1. Find Maximum Element

Difficulty: Easy | **Time:** O(n) | **Space:** O(1)

Approach 1: Linear Scan (Most Common)

```
public static int findMax(int[] arr) {
    if (arr == null || arr.length == 0) {
        return -1;
    }
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

Approach 2: Using Java Streams

```
public static int findMaxStream(int[] arr) {
    return Arrays.stream(arr)
                 .max()
                 .orElse(-1);
}
```

Approach 3: Using Collections

```
public static int findMaxCollections(int[] arr) {
    List<Integer> list = Arrays.stream(arr).boxed().collect(Collectors.toList());
    return Collections.max(list);
}
```

Key Points:

- Always handle edge cases (null, empty array)
- Initialize max with first element to avoid Integer.MIN_VALUE issues
- Streams are slower but more readable

---

2. Array Sum and Average

Difficulty: Easy | **Time:** O(n) | **Space:** O(1)

Approach 1: Manual Loop

```
public static double calculateAverage(int[] arr) {
    if (arr == null || arr.length == 0) {
        return 0;
    }
    long sum = 0;  // Use long to prevent overflow
    for (int num : arr) {
        sum += num;
    }
    return (double) sum / arr.length;
}
```

Approach 2: Using Streams

```
public static double calculateAverageStream(int[] arr) {
    return Arrays.stream(arr)
                 .average()
                 .orElse(0.0);
}
```

Approach 3: Two-Pass Sum

```
public static int[] sumAndAverage(int[] arr) {
    int sum = 0;
    int count = 0;
    for (int num : arr) {
        sum += num;
        count++;
    }
    int average = (count > 0) ? sum / count : 0;
    return new int[] { sum, average };
}
```

⚠ **Important:** Use `long` for sum to avoid integer overflow

---

3. Reverse an Array

Difficulty: Easy | **Time:** O(n) | **Space:** O(1) or O(n)

Approach 1: Two Pointer In-Place

```
public static void reverseArray(int[] arr) {
    int left = 0;
    int right = arr.length - 1;

    while (left < right) {
        // Swap elements
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left++;
        right--;
    }
}
```

Approach 2: Using New Array

```
public static int[] reverseArrayNew(int[] arr) {
    int[] reversed = new int[arr.length];
    for (int i = 0; i < arr.length; i++) {
        reversed[i] = arr[arr.length - 1 - i];
    }
    return reversed;
}
```

Approach 3: Recursion

```
public static void reverseRecursive(int[] arr, int left, int right) {
    if (left >= right) {
        return;
    }
    // Swap
    int temp = arr[left];
    arr[left] = arr[right];
    arr[right] = temp;

    reverseRecursive(arr, left + 1, right - 1);
}

// Call: reverseRecursive(arr, 0, arr.length - 1);
```

Approach 4: Using Collections

```
public static Integer[] reverseUsingCollections(Integer[] arr) {
    List<Integer> list = Arrays.asList(arr);
    Collections.reverse(list);
    return list.toArray(new Integer[0]);
}
```

---

4. Check if Array is Sorted

Difficulty: Easy | **Time:** O(n) | **Space:** O(1)

Approach 1: Single Pass

```
public static boolean isSorted(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            return false;
        }
    }
    return true;
}
```

Approach 2: Using Streams

```
public static boolean isSortedStream(int[] arr) {
    return IntStream.range(0, arr.length - 1)
                    .allMatch(i -> arr[i] <= arr[i + 1]);
}
```

Approach 3: Check for Both Ascending and Descending

```
public static int checkSorted(int[] arr) {
    boolean isAscending = true;
    boolean isDescending = true;

    for (int i = 0; i < arr.length - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            isAscending = false;
        }
        if (arr[i] < arr[i + 1]) {
            isDescending = false;
        }
    }

    if (isAscending) return 1;      // Ascending
    if (isDescending) return -1;    // Descending
    return 0;                        // Not sorted
}
```

---

5. Find Second Largest Element

Difficulty: Easy | **Time:** O(n) | **Space:** O(1)

Approach 1: Track Top 2 Elements

```
public static int secondLargest(int[] arr) {
    if (arr.length < 2) {
        return -1;
    }

    int largest = Integer.MIN_VALUE;
    int secondMax = Integer.MIN_VALUE;

    for (int num : arr) {
        if (num > largest) {
            secondMax = largest;
            largest = num;
        } else if (num > secondMax && num != largest) {
            secondMax = num;
        }
    }

    return secondMax == Integer.MIN_VALUE ? -1 : secondMax;
}
```

Approach 2: Sort and Access

```
public static int secondLargestSort(int[] arr) {
    if (arr.length < 2) {
        return -1;
    }
    Arrays.sort(arr);
    return arr[arr.length - 2];
}
```

Approach 3: Two-Pass Approach

```
public static int secondLargestTwoPass(int[] arr) {
    if (arr.length < 2) {
        return -1;
    }

    // Find largest
    int largest = arr[0];
    for (int num : arr) {
        if (num > largest) {
            largest = num;
        }
    }

    // Find second largest (not equal to largest)
    int secondMax = Integer.MIN_VALUE;
    for (int num : arr) {
        if (num < largest && num > secondMax) {
            secondMax = num;
        }
    }

    return secondMax == Integer.MIN_VALUE ? -1 : secondMax;
}
```

---

6. Count Occurrences of Element

Difficulty: Easy | **Time:** O(n) | **Space:** O(1)

Approach 1: Simple Loop

```
public static int countOccurrences(int[] arr, int target) {
    int count = 0;
    for (int num : arr) {
        if (num == target) {
            count++;
        }
    }
    return count;
}
```

Approach 2: Using Streams

```
public static long countOccurrencesStream(int[] arr, int target) {
    return Arrays.stream(arr)
                 .filter(n -> n == target)
                 .count();
}
```

Approach 3: Using HashMap

```
public static Map<Integer, Integer> countAllOccurrences(int[] arr) {
    Map<Integer, Integer> freqMap = new HashMap<>();
    for (int num : arr) {
        freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
    }
    return freqMap;
}
```

---

7. Remove Duplicates from Sorted Array

Difficulty: Easy | **Time:** O(n) | **Space:** O(1) or O(n)

Approach 1: Two Pointer In-Place

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

    int uniqueIndex = 0;
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] != arr[uniqueIndex]) {
            uniqueIndex++;
            arr[uniqueIndex] = arr[i];
        }
    }
    return uniqueIndex + 1;
}
```

Approach 2: Using LinkedHashSet (Preserves Order)

```
public static int[] removeDuplicatesSet(int[] arr) {
    Set<Integer> uniqueSet = new LinkedHashSet<>();
    for (int num : arr) {
        uniqueSet.add(num);
    }
    return uniqueSet.stream()
                    .mapToInt(i -> i)
                    .toArray();
}
```

Approach 3: Using Streams Distinct

```
public static int[] removeDuplicatesStream(int[] arr) {
    return Arrays.stream(arr)
                 .distinct()
                 .toArray();
}
```

---

8. Merge Two Sorted Arrays

Difficulty: Easy | **Time:** O(m + n) | **Space:** O(m + n)

Approach 1: Two Pointer Merge (Optimal)

```
public static int[] mergeSorted(int[] arr1, int[] arr2) {
    int[] merged = new int[arr1.length + arr2.length];
    int i = 0, j = 0, k = 0;

    while (i < arr1.length && j < arr2.length) {
        if (arr1[i] <= arr2[j]) {
            merged[k++] = arr1[i++];
        } else {
            merged[k++] = arr2[j++];
        }
    }

    // Copy remaining elements from arr1
    while (i < arr1.length) {
        merged[k++] = arr1[i++];
    }

    // Copy remaining elements from arr2
    while (j < arr2.length) {
        merged[k++] = arr2[j++];
    }

    return merged;
}
```

Approach 2: Concatenate and Sort

```
public static int[] mergeSortedSimple(int[] arr1, int[] arr2) {
    int[] merged = new int[arr1.length + arr2.length];
    int index = 0;

    for (int num : arr1) {
        merged[index++] = num;
    }
    for (int num : arr2) {
        merged[index++] = num;
    }

    Arrays.sort(merged);
    return merged;
}
```

Approach 3: Using Streams

```
public static int[] mergeSortedStreams(int[] arr1, int[] arr2) {
    return IntStream.concat(Arrays.stream(arr1), Arrays.stream(arr2))
                    .sorted()
                    .toArray();
}
```

---

9. Rotate Array by K Positions

Difficulty: Easy | **Time:** O(n) | **Space:** O(1)

Approach 1: Reverse Method (Optimal)

```
public static void rotateArray(int[] arr, int k) {
    k = k % arr.length;  // Handle k > array length

    reverse(arr, 0, arr.length - 1);
    reverse(arr, 0, k - 1);
    reverse(arr, k, arr.length - 1);
}

private static void reverse(int[] arr, int start, int end) {
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}
```

Approach 2: Extra Space

```
public static int[] rotateArrayNew(int[] arr, int k) {
    k = k % arr.length;
    int[] rotated = new int[arr.length];

    for (int i = 0; i < arr.length; i++) {
        rotated[(i + k) % arr.length] = arr[i];
    }
    return rotated;
}
```

Approach 3: Temporary Array

```
public static void rotateArrayTemp(int[] arr, int k) {
    k = k % arr.length;
    int[] temp = new int[k];

    // Copy last k elements
    for (int i = 0; i < k; i++) {
        temp[i] = arr[arr.length - k + i];
    }

    // Shift remaining elements
    for (int i = arr.length - 1; i >= k; i--) {
        arr[i] = arr[i - k];
    }

    // Copy from temp back
    for (int i = 0; i < k; i++) {
        arr[i] = temp[i];
    }
}
```

---

10. Linear vs Binary Search

Difficulty: Easy | **Time:** O(n) vs O(log n) | **Space:** O(1)

Approach 1: Linear Search

```
public static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Approach 2: Binary Search (Iterative)

```
public static int binarySearch(int[] arr, int target) {
    int left = 0, right = arr.length - 1;

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

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

Approach 3: Binary Search (Recursive)

```
public static int binarySearchRecursive(int[] arr, int target, int left, int right) {
    if (left > right) {
        return -1;
    }

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

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

Approach 4: Using Java Built-in

```
public static int binarySearchBuiltIn(int[] arr, int target) {
    int index = Arrays.binarySearch(arr, target);
    return index >= 0 ? index : -1;
}
```

When to Use:

- **Linear Search:** Unsorted array, small datasets, single search

- **Binary Search:** Sorted array, multiple searches, large datasets

---

ARRAYS - NEXT LEVEL

1. Two Sum Problem

Difficulty: Medium | **Time:** O(n) | **Space:** O(n)

Approach 1: HashMap (Best)

```
public static int[] twoSum(int[] arr, int target) {
    Map<Integer, Integer> map = new HashMap<>();

    for (int i = 0; i < arr.length; i++) {
        int complement = target - arr[i];

        if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
        }
        map.put(arr[i], i);
    }
    return new int[] {};
}
```

Approach 2: Two Pointer (Sorted Array)

```
public static int[] twoSumSorted(int[] arr, int target) {
    Arrays.sort(arr);
    int left = 0, right = arr.length - 1;

    while (left < right) {
        int sum = arr[left] + arr[right];

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

Approach 3: Brute Force

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

---

2. Maximum Subarray (Kadane's Algorithm)

Difficulty: Medium | **Time:** O(n) | **Space:** O(1)

Approach 1: Kadane's Algorithm (Optimal)

```
public static int maxSubarray(int[] arr) {
    int maxCurrent = arr[0];
    int maxGlobal = arr[0];

    for (int i = 1; i < arr.length; i++) {
        maxCurrent = Math.max(arr[i], maxCurrent + arr[i]);
        maxGlobal = Math.max(maxGlobal, maxCurrent);
    }
    return maxGlobal;
}
```

Approach 2: With Subarray Indices

```
public static void maxSubarrayWithIndices(int[] arr) {
    int maxSum = arr[0];
    int currentSum = arr[0];
    int start = 0, end = 0, tempStart = 0;

    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > currentSum + arr[i]) {
            currentSum = arr[i];
            tempStart = i;
        } else {
            currentSum += arr[i];
        }

        if (currentSum > maxSum) {
            maxSum = currentSum;
            start = tempStart;
            end = i;
        }
    }

    System.out.println("Max Sum: " + maxSum + " from index " + start + " to " + end);
}
```

Approach 3: Divide and Conquer

```
public static int maxSubarrayDivide(int[] arr, int left, int right) {
    if (left == right) {
        return arr[left];
    }

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

    int leftMax = maxSubarrayDivide(arr, left, mid);
    int rightMax = maxSubarrayDivide(arr, mid + 1, right);
    int crossMax = maxCrossingSum(arr, left, mid, right);

    return Math.max(Math.max(leftMax, rightMax), crossMax);
}

private static int maxCrossingSum(int[] arr, int left, int mid, int right) {
    int leftSum = Integer.MIN_VALUE, sum = 0;
    for (int i = mid; i >= left; i--) {
        sum += arr[i];
        leftSum = Math.max(leftSum, sum);
    }

    int rightSum = Integer.MIN_VALUE;
    sum = 0;
    for (int i = mid + 1; i <= right; i++) {
        sum += arr[i];
        rightSum = Math.max(rightSum, sum);
    }

    return leftSum + rightSum;
}
```

---

3. Product of Array Except Self

Difficulty: Medium | **Time:** O(n) | **Space:** O(n) or O(1)

Approach 1: Prefix and Suffix Products

```
public static int[] productExceptSelf(int[] arr) {
    int[] result = new int[arr.length];
    int[] prefix = new int[arr.length];
    int[] suffix = new int[arr.length];

    // Calculate prefix products
    prefix[0] = 1;
    for (int i = 1; i < arr.length; i++) {
        prefix[i] = prefix[i - 1] * arr[i - 1];
    }

    // Calculate suffix products
    suffix[arr.length - 1] = 1;
    for (int i = arr.length - 2; i >= 0; i--) {
        suffix[i] = suffix[i + 1] * arr[i + 1];
    }

    // Multiply prefix and suffix
    for (int i = 0; i < arr.length; i++) {
        result[i] = prefix[i] * suffix[i];
    }
    return result;
}
```

Approach 2: Space Optimized (Single Pass)

```
public static int[] productExceptSelfOptimized(int[] arr) {
    int[] result = new int[arr.length];

    // Forward pass: prefix products
    result[0] = 1;
    for (int i = 1; i < arr.length; i++) {
        result[i] = result[i - 1] * arr[i - 1];
    }

    // Backward pass: multiply with suffix
    int suffix = 1;
    for (int i = arr.length - 1; i >= 0; i--) {
        result[i] *= suffix;
        suffix *= arr[i];
    }

    return result;
}
```

Approach 3: Using Division (If No Zeros)

```
public static int[] productExceptSelfDivision(int[] arr) {
    int product = 1;
    int zeroCount = 0;
    int zeroIndex = -1;

    // Calculate total product and count zeros
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == 0) {
            zeroCount++;
            zeroIndex = i;
        } else {
            product *= arr[i];
        }
    }

    int[] result = new int[arr.length];

    if (zeroCount > 1) {
        return result;  // All zeros
    } else if (zeroCount == 1) {
        result[zeroIndex] = product;
    } else {
        for (int i = 0; i < arr.length; i++) {
            result[i] = product / arr[i];
        }
    }

    return result;
}
```

---

4. Container With Most Water

Difficulty: Medium | **Time:** O(n) | **Space:** O(1)

Approach 1: Two Pointer Greedy (Optimal)

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

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

        // Move the pointer with smaller height
        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
    return maxArea;
}
```

Approach 2: Brute Force

```
public static int maxAreaBrute(int[] height) {
    int maxArea = 0;

    for (int i = 0; i < height.length; i++) {
        for (int j = i + 1; j < height.length; j++) {
            int width = j - i;
            int h = Math.min(height[i], height[j]);
            maxArea = Math.max(maxArea, width * h);
        }
    }
    return maxArea;
}
```

---

5. First Missing Positive

Difficulty: Hard | **Time:** O(n) | **Space:** O(1)

Approach 1: Array as Hash Map

```
public static int firstMissingPositive(int[] arr) {
    // Place number n at index n-1
    for (int i = 0; i < arr.length; i++) {
        while (arr[i] > 0 && arr[i] <= arr.length && arr[arr[i] - 1] != arr[i]) {
            // Swap arr[i] with arr[arr[i] - 1]
            int correctIdx = arr[i] - 1;
            int temp = arr[i];
            arr[i] = arr[correctIdx];
            arr[correctIdx] = temp;
        }
    }

    // Find first missing positive
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] != i + 1) {
            return i + 1;
        }
    }
    return arr.length + 1;
}
```

Approach 2: Using HashSet

```
public static int firstMissingPositiveSet(int[] arr) {
    Set<Integer> nums = new HashSet<>();
    for (int num : arr) {
        if (num > 0) {
            nums.add(num);
        }
    }

    for (int i = 1; i <= arr.length + 1; i++) {
        if (!nums.contains(i)) {
            return i;
        }
    }
    return arr.length + 1;
}
```

---

6. Search in Rotated Sorted Array

Difficulty: Medium | **Time:** O(log n) | **Space:** O(1)

Approach 1: Modified Binary Search

```java
public static int searchRotated(int[] arr, int target) {
    int left = 0, right = arr.length - 1;

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

        if (arr[mid] == target) {
            return mid;
        }

        // Determine which side is sorted
        if (arr[left] <= arr[mid]) {
            // Left side is sorted
            if (target >= arr[left] && target < arr[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else {
            // Right side is sorted
            if (target > arr[mid] && target <= arr[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return -1;
}
```

STRINGS - BASIC LEVEL

1. Reverse a String

Difficulty: Easy | **Time:** O(n) | **Space:** O(n)

Approach 1: Character Array

```
public static String reverseString(String str) {
    char[] chars = str.toCharArray();
    int left = 0, right = chars.length - 1;

    while (left < right) {
        char temp = chars[left];
        chars[left] = chars[right];
        chars[right] = temp;
        left++;
        right--;
    }
    return new String(chars);
}
```

Approach 2: StringBuilder

```
public static String reverseStringBuilder(String str) {
    return new StringBuilder(str).reverse().toString();
}
```

Approach 3: Recursion

```
public static String reverseRecursive(String str) {
    if (str.isEmpty()) {
        return str;
    }
    return reverseRecursive(str.substring(1)) + str.charAt(0);
}
```

Approach 4: Using Loops

```
public static String reverseLoop(String str) {
    String reversed = "";
    for (int i = str.length() - 1; i >= 0; i--) {
        reversed += str.charAt(i);
    }
    return reversed;
}
```

---

2. Check Palindrome

Difficulty: Easy | **Time:** O(n) | **Space:** O(1) or O(n)

Approach 1: Two Pointer

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

    while (left < right) {
        if (str.charAt(left) != str.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

Approach 2: Reverse and Compare

```
public static boolean isPalindromeReverse(String str) {
    String reversed = new StringBuilder(str).reverse().toString();
    return str.equalsIgnoreCase(reversed);
}
```

Approach 3: Recursion

```
public static boolean isPalindromeRecursive(String str) {
    str = str.toLowerCase();
    return isPalindromeHelper(str, 0, str.length() - 1);
}

private static boolean isPalindromeHelper(String str, int left, int right) {
    if (left >= right) {
        return true;
    }
    if (str.charAt(left) != str.charAt(right)) {
        return false;
    }
    return isPalindromeHelper(str, left + 1, right - 1);
}
```

---

3. Count Vowels and Consonants

Difficulty: Easy | **Time:** O(n) | **Space:** O(1)

Approach 1: Simple Loop

```
public static void countVowelsConsonants(String str) {
    int vowels = 0, consonants = 0;
    str = str.toLowerCase();

    for (char c : str.toCharArray()) {
        if (c >= 'a' && c <= 'z') {
            if ("aeiou".indexOf(c) != -1) {
                vowels++;
            } else {
                consonants++;
            }
        }
    }
    System.out.println("Vowels: " + vowels + ", Consonants: " + consonants);
}
```

Approach 2: Using Streams

```
public static Map<String, Long> countVowelsConsonantsStream(String str) {
    str = str.toLowerCase();
    String vowels = "aeiou";

    long vowelCount = str.chars()
                         .filter(c -> vowels.indexOf(c) != -1)
                         .count();

    long consonantCount = str.chars()
                             .filter(c -> c >= 'a' && c <= 'z' && vowels.indexOf(c) == -1)
                             .count();

    Map<String, Long> result = new HashMap<>();
    result.put("vowels", vowelCount);
    result.put("consonants", consonantCount);
    return result;
}
```

---

4. Check Anagram

Difficulty: Easy | **Time:** O(n log n) or O(n) | **Space:** O(n)

Approach 1: Sort and Compare

```
public static boolean isAnagram(String s1, String s2) {
    char[] arr1 = s1.toLowerCase().toCharArray();
    char[] arr2 = s2.toLowerCase().toCharArray();

    Arrays.sort(arr1);
    Arrays.sort(arr2);

    return Arrays.equals(arr1, arr2);
}
```

Approach 2: Frequency Map

```
public static boolean isAnagramMap(String s1, String s2) {
    if (s1.length() != s2.length()) {
        return false;
    }

    Map<Character, Integer> map = new HashMap<>();

    for (char c : s1.toLowerCase().toCharArray()) {
        map.put(c, map.getOrDefault(c, 0) + 1);
    }

    for (char c : s2.toLowerCase().toCharArray()) {
        if (!map.containsKey(c) || map.get(c) == 0) {
            return false;
        }
        map.put(c, map.get(c) - 1);
    }
    return true;
}
```

Approach 3: Character Count Array

```
public static boolean isAnagramArray(String s1, String s2) {
    if (s1.length() != s2.length()) {
        return false;
    }

    int[] count = new int[26];

    for (int i = 0; i < s1.length(); i++) {
        count[s1.charAt(i) - 'a']++;
        count[s2.charAt(i) - 'a']--;
    }

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

---

5. Find Character Frequency

Difficulty: Easy | **Time:** O(n) | **Space:** O(n)

Approach 1: HashMap

```
public static void characterFrequency(String str) {
    Map<Character, Integer> freqMap = new LinkedHashMap<>();

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

    freqMap.forEach((key, value) ->
        System.out.println(key + ": " + value)
    );
}
```

Approach 2: Using Streams

```
public static Map<Character, Long> characterFrequencyStream(String str) {
    return str.chars()
              .mapToObj(c -> (char) c)
              .collect(Collectors.groupingBy(
                  c -> c,
                  Collectors.counting()
              ));
}
```

Approach 3: Character Array

```
public static void characterFrequencyArray(String str) {
    int[] freq = new int[256];

    for (char c : str.toCharArray()) {
        freq[c]++;
    }

    for (int i = 0; i < 256; i++) {
        if (freq[i] > 0) {
            System.out.println((char) i + ": " + freq[i]);
        }
    }
}
```

---

6. String Concatenation

Difficulty: Easy | **Time:** O(n) | **Space:** O(n)

Approach 1: Using + Operator

```
public static String concatenate(String s1, String s2) {
    return s1 + s2;
}
```

Approach 2: Using concat() Method

```
public static String concatenateConcat(String s1, String s2) {
    return s1.concat(s2);
}
```

Approach 3: Using StringBuilder

```
public static String concatenateBuilder(String... strings) {
    StringBuilder sb = new StringBuilder();
    for (String s : strings) {
        sb.append(s);
    }
    return sb.toString();
}
```

Approach 4: Using String.join()

```
public static String concatenateJoin(String... strings) {
    return String.join("", strings);
}
```

---

7. Remove Whitespaces

Difficulty: Easy | **Time:** O(n) | **Space:** O(n)

Approach 1: Using replaceAll()

```
public static String removeWhitespaces(String str) {
    return str.replaceAll("\\s+", "");
}
```

Approach 2: Using Streams

```
public static String removeWhitespacesStream(String str) {
    return str.chars()
              .filter(c -> !Character.isWhitespace(c))
              .collect(StringBuilder::new,
                       (sb, c) -> sb.append((char) c),
                       StringBuilder::append)
              .toString();
}
```

Approach 3: Using Loop

```
public static String removeWhitespacesLoop(String str) {
    StringBuilder sb = new StringBuilder();
    for (char c : str.toCharArray()) {
        if (!Character.isWhitespace(c)) {
            sb.append(c);
        }
    }
    return sb.toString();
}
```

---

8. Find Substring

Difficulty: Easy | **Time:** O(n*m) | **Space:** O(1)

Approach 1: Using indexOf()

```
public static int findSubstring(String str, String sub) {
    return str.indexOf(sub);
}
```

Approach 2: Manual Search

```
public static int findSubstringManual(String str, String sub) {
    for (int i = 0; i <= str.length() - sub.length(); i++) {
        if (str.substring(i, i + sub.length()).equals(sub)) {
            return i;
        }
    }
    return -1;
}
```

Approach 3: Character by Character

```
public static int findSubstringChar(String str, String sub) {
    for (int i = 0; i <= str.length() - sub.length(); i++) {
        boolean found = true;
        for (int j = 0; j < sub.length(); j++) {
            if (str.charAt(i + j) != sub.charAt(j)) {
                found = false;
                break;
            }
        }
        if (found) {
            return i;
        }
    }
    return -1;
}
```

---

9. String Comparison

Difficulty: Easy | **Time:** O(n) | **Space:** O(1)

Approach 1: equals()

```
public static boolean compareStrings(String s1, String s2) {
    return s1.equals(s2);
}
```

Approach 2: equalsIgnoreCase()

```
public static boolean compareStringsIgnoreCase(String s1, String s2) {
    return s1.equalsIgnoreCase(s2);
}
```

Approach 3: compareTo()

```
public static int compareStringsCompareTo(String s1, String s2) {
    return s1.compareTo(s2);  // Returns 0 if equal
}
```

---

10. String Conversion (Case)

Difficulty: Easy | **Time:** O(n) | **Space:** O(n)

Approach 1: toUpperCase() and toLowerCase()

```
public static void convertCase(String str) {
    System.out.println("Uppercase: " + str.toUpperCase());
    System.out.println("Lowercase: " + str.toLowerCase());
}
```

Approach 2: Using Streams

```
public static String convertToUpperCase(String str) {
    return str.chars()
              .map(Character::toUpperCase)
              .collect(StringBuilder::new,
                       (sb, c) -> sb.append((char) c),
                       StringBuilder::append)
              .toString();
}
```

NUMBERS - BASIC LEVEL

1. Check Prime Number

Difficulty: Easy | **Time:** O(√n) | **Space:** O(1)

Approach 1: Check up to √n (Optimal)

```
public static boolean isPrime(int n) {
    if (n <= 1) {
        return false;
    }
    if (n == 2) {
        return true;
    }
    if (n % 2 == 0) {
        return false;
    }

    for (int i = 3; i <= Math.sqrt(n); i += 2) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

Approach 2: Simple Check

```
public static boolean isPrimeSimple(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i < n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

---

2. Calculate Factorial

Difficulty: Easy | **Time:** O(n) | **Space:** O(1) or O(n)

Approach 1: Iterative

```
public static long factorial(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("Number must be positive");
    }
    long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

Approach 2: Recursive

```
public static long factorialRecursive(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    return n * factorialRecursive(n - 1);
}
```

Approach 3: Using Streams

```
public static long factorialStream(int n) {
    return IntStream.rangeClosed(1, n)
                    .asLongStream()
                    .reduce(1, (a, b) -> a * b);
}
```

---

3. Sum of Digits

Difficulty: Easy | **Time:** O(log n) | **Space:** O(1)

Approach 1: Using Modulo

```
public static int sumOfDigits(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}
```

Approach 2: Using String

```
public static int sumOfDigitsString(int n) {
    int sum = 0;
    for (char c : String.valueOf(n).toCharArray()) {
        sum += Character.getNumericValue(c);
    }
    return sum;
}
```

Approach 3: Using Streams

```
public static int sumOfDigitsStream(int n) {
    return String.valueOf(Math.abs(n))
                .chars()
                .map(Character::getNumericValue)
                .sum();
}
```

---

4. Reverse a Number

Difficulty: Easy | **Time:** O(log n) | **Space:** O(1)

Approach 1: Using Modulo with Overflow Check

```
public static int reverseNumber(int n) {
    int reversed = 0;
    int original = n;
    n = Math.abs(n);

    while (n != 0) {
        int digit = n % 10;

        // Check for overflow before multiplying
        if (reversed > Integer.MAX_VALUE / 10 ||
            (reversed == Integer.MAX_VALUE / 10 && digit > 7)) {
            return 0;  // Overflow
        }

        reversed = reversed * 10 + digit;
        n /= 10;
    }

    return original < 0 ? -reversed : reversed;
}
```

Approach 2: Using String

```
public static int reverseNumberString(int n) {
    String reversed = new StringBuilder(String.valueOf(Math.abs(n)))
                          .reverse()
                          .toString();
    return n < 0 ? -Integer.parseInt(reversed) : Integer.parseInt(reversed);
}
```

---

5. GCD (Greatest Common Divisor)

Difficulty: Easy | **Time:** O(log min(a,b)) | **Space:** O(1)

Approach 1: Euclidean Algorithm (Iterative)

```
public static int gcd(int a, int b) {
    a = Math.abs(a);
    b = Math.abs(b);

    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

Approach 2: Euclidean Algorithm (Recursive)

```
public static int gcdRecursive(int a, int b) {
    a = Math.abs(a);
    b = Math.abs(b);

    if (b == 0) {
        return a;
    }
    return gcdRecursive(b, a % b);
}
```

Approach 3: Using Java Built-in

```
public static int gcdBuiltIn(int a, int b) {
    return Math.abs(a) == 0 ? Math.abs(b) : gcdBuiltIn(Math.abs(b), Math.abs(a) % Math.abs(b));
}
```

---

6. LCM (Least Common Multiple)

Difficulty: Easy | **Time:** O(log min(a,b)) | **Space:** O(1)

Approach 1: Using GCD

```
public static int lcm(int a, int b) {
    return Math.abs(a * b) / gcd(a, b);
}

private static int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

---

7. Check Perfect Square

Difficulty: Easy | **Time:** O(1) | **Space:** O(1)

Approach 1: Using sqrt()

```
public static boolean isPerfectSquare(int n) {
    if (n < 0) {
        return false;
    }
    int sqrt = (int) Math.sqrt(n);
    return sqrt * sqrt == n;
}
```

Approach 2: Binary Search

```
public static boolean isPerfectSquareBinary(int n) {
    if (n < 0) {
        return false;
    }
    if (n == 0 || n == 1) {
        return true;
    }

    long left = 1, right = n;

    while (left <= right) {
        long mid = left + (right - left) / 2;
        long square = mid * mid;

        if (square == n) {
            return true;
        } else if (square < n) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return false;
}
```

---

8. Fibonacci Sequence

Difficulty: Easy | **Time:** O(n) | **Space:** O(1) or O(n)

Approach 1: Iterative (Best)

```
public static int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }

    int a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        int temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

Approach 2: Recursive with Memoization

```
public static int fibonacciMemo(int n, Map<Integer, Integer> memo) {
    if (n <= 1) {
        return n;
    }

    if (memo.containsKey(n)) {
        return memo.get(n);
    }

    int result = fibonacciMemo(n - 1, memo) + fibonacciMemo(n - 2, memo);
    memo.put(n, result);
    return result;
}

// Usage: fibonacciMemo(n, new HashMap<>())
```

Approach 3: Plain Recursion (Avoid for large n!)

```
public static int fibonacciNaive(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacciNaive(n - 1) + fibonacciNaive(n - 2);
}
```

---

9. Power Function

Difficulty: Easy | **Time:** O(log n) | **Space:** O(1)

Approach 1: Binary Exponentiation

```
public static double power(double x, int n) {
    long N = n;
    if (N < 0) {
        x = 1 / x;
        N = -N;
    }

    double result = 1.0;
    double current = x;

    while (N > 0) {
        if (N % 2 == 1) {
            result *= current;
        }
        current *= current;
        N /= 2;
    }
    return result;
}
```

Approach 2: Simple Multiplication

```
public static double powerSimple(double x, int n) {
    if (n == 0) {
        return 1.0;
    }
    if (n < 0) {
        return 1.0 / powerSimple(x, -n);
    }

    double result = 1.0;
    for (int i = 0; i < n; i++) {
        result *= x;
    }
    return result;
}
```

---

10. Palindrome Number (Without String)

Difficulty: Easy | **Time:** O(log n) | **Space:** O(1)

Approach 1: Reverse and Compare

```
public static boolean isPalindromeNumber(int x) {
    if (x < 0) {
        return false;
    }
    if (x % 10 == 0 && x != 0) {
        return false;
    }

    int reversed = 0;
    while (x > reversed) {
        reversed = reversed * 10 + x % 10;
        x /= 10;
    }

    return x == reversed || x == reversed / 10;
}
```

Approach 2: Two Pointer on String

```
public static boolean isPalindromeNumberString(int x) {
    String str = String.valueOf(x);
    int left = 0, right = str.length() - 1;

    while (left < right) {
        if (str.charAt(left) != str.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
```

NUMBERS - NEXT LEVEL

1. Power(x, n) - With Negative Exponents

Difficulty: Medium | **Time:** O(log n) | **Space:** O(1)

Approach 1: Binary Exponentiation (Optimal)

```
public static double powerOptimal(double x, int n) {
    long N = n;  // Use long to handle Integer.MIN_VALUE

    if (N < 0) {
        x = 1 / x;
        N = -N;
    }

    double result = 1.0;
    double currentPower = x;

    while (N > 0) {
        if ((N & 1) == 1) {  // If odd
            result *= currentPower;
        }
        currentPower *= currentPower;
        N >>= 1;  // N = N / 2
    }

    return result;
}
```

Approach 2: Recursive

```
public static double powerRecursive(double x, int n) {
    long N = n;

    if (N < 0) {
        x = 1 / x;
        N = -N;
    }

    return powerHelper(x, N);
}

private static double powerHelper(double x, long n) {
    if (n == 0) {
        return 1.0;
    }

    double half = powerHelper(x, n / 2);

    if (n % 2 == 0) {
        return half * half;
    } else {
        return half * half * x;
    }
}
```

---

2. Integer to Roman / Roman to Integer

Difficulty: Medium | **Time:** O(n) | **Space:** O(1)

Approach 1: Integer to Roman

```
public static String intToRoman(int num) {
    int[] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
    String[] symbols = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};

    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < values.length; i++) {
        while (num >= values[i]) {
            sb.append(symbols[i]);
            num -= values[i];
        }
    }

    return sb.toString();
}
```

Approach 2: Roman to Integer

```
public static int romanToInt(String s) {
    Map<Character, Integer> romanMap = new HashMap<>();
    romanMap.put('I', 1);
    romanMap.put('V', 5);
    romanMap.put('X', 10);
    romanMap.put('L', 50);
    romanMap.put('C', 100);
    romanMap.put('D', 500);
    romanMap.put('M', 1000);

    int num = 0;
    for (int i = 0; i < s.length(); i++) {
        int current = romanMap.get(s.charAt(i));
        int next = (i + 1 < s.length()) ? romanMap.get(s.charAt(i + 1)) : 0;

        if (current < next) {
            num -= current;
        } else {
            num += current;
        }
    }

    return num;
}
```

---

3. Happy Number

Difficulty: Medium | **Time:** O(log n) | **Space:** O(1)

Approach 1: Using HashSet (Cycle Detection)

```
public static boolean isHappyNumber(int n) {
    Set<Integer> seen = new HashSet<>();

    while (n != 1 && !seen.contains(n)) {
        seen.add(n);
        n = sumOfSquares(n);
    }

    return n == 1;
}

private static int sumOfSquares(int n) {
    int sum = 0;
    while (n > 0) {
        int digit = n % 10;
        sum += digit * digit;
        n /= 10;
    }
    return sum;
}
```

Approach 2: Floyd's Cycle Detection

```
public static boolean isHappyNumberFloyd(int n) {
    int slow = n;
    int fast = sumOfSquares(n);

    while (fast != 1 && slow != fast) {
        slow = sumOfSquares(slow);
        fast = sumOfSquares(sumOfSquares(fast));
    }

    return fast == 1;
}

private static int sumOfSquares(int n) {
    int sum = 0;
    while (n > 0) {
        int digit = n % 10;
        sum += digit * digit;
        n /= 10;
    }
    return sum;
}
```

---

4. Single Number (XOR Trick)

Difficulty: Medium | **Time:** O(n) | **Space:** O(1)

Approach 1: XOR Operation (Optimal)

```
public static int singleNumber(int[] arr) {
    int result = 0;
    for (int num : arr) {
        result ^= num;  // XOR: a ^ a = 0, a ^ 0 = a
    }
    return result;
}
```

Approach 2: HashSet

```
public static int singleNumberSet(int[] arr) {
    Set<Integer> set = new HashSet<>();
    for (int num : arr) {
        if (set.contains(num)) {
            set.remove(num);
        } else {
            set.add(num);
        }
    }
    return set.iterator().next();
}
```

Approach 3: HashMap

```
public static int singleNumberMap(int[] arr) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int num : arr) {
        map.put(num, map.getOrDefault(num, 0) + 1);
    }
    for (int num : arr) {
        if (map.get(num) == 1) {
            return num;
        }
    }
    return -1;
}
```

---

5. Sieve of Eratosthenes (Count Primes)

Difficulty: Medium | **Time:** O(n log log n) | **Space:** O(n)

Approach 1: Sieve Method

```
public static int countPrimes(int n) {
    boolean[] isPrime = new boolean[n];
    Arrays.fill(isPrime, true);

    if (n > 0) {
        isPrime[0] = false;
    }
    if (n > 1) {
        isPrime[1] = false;
    }

    for (int i = 2; i * i < n; i++) {
        if (isPrime[i]) {
            for (int j = i * i; j < n; j += i) {
                isPrime[j] = false;
            }
        }
    }

    int count = 0;
    for (int i = 0; i < n; i++) {
        if (isPrime[i]) {
            count++;
        }
    }
    return count;
}
```

Approach 2: Get All Primes

```
public static List<Integer> getAllPrimes(int n) {
    boolean[] isPrime = new boolean[n + 1];
    Arrays.fill(isPrime, true);
    isPrime[0] = isPrime[1] = false;

    for (int i = 2; i * i <= n; i++) {
        if (isPrime[i]) {
            for (int j = i * i; j <= n; j += i) {
                isPrime[j] = false;
            }
        }
    }

    List<Integer> primes = new ArrayList<>();
    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            primes.add(i);
        }
    }
    return primes;
}
```

---

6. Ugly Number

Difficulty: Medium | **Time:** O(n) | **Space:** O(n)

Approach 1: DP Approach

```
public static int nthUglyNumber(int n) {
    int[] dp = new int[n];
    dp[0] = 1;

    int i2 = 0, i3 = 0, i5 = 0;
    int nextMultiple2 = 2;
    int nextMultiple3 = 3;
    int nextMultiple5 = 5;

    for (int i = 1; i < n; i++) {
        int nextUgly = Math.min(nextMultiple2, Math.min(nextMultiple3, nextMultiple5));
        dp[i] = nextUgly;

        if (nextUgly == nextMultiple2) {
            nextMultiple2 = dp[++i2] * 2;
        }
        if (nextUgly == nextMultiple3) {
            nextMultiple3 = dp[++i3] * 3;
        }
        if (nextUgly == nextMultiple5) {
            nextMultiple5 = dp[++i5] * 5;
        }
    }

    return dp[n - 1];
}
```

---

7. Missing Number in Array

Difficulty: Medium | **Time:** O(n) | **Space:** O(1)

Approach 1: XOR Method

```
public static int missingNumber(int[] arr) {
    int xor1 = 0, xor2 = 0;

    for (int i = 0; i < arr.length; i++) {
        xor1 ^= arr[i];
        xor2 ^= (i + 1);
    }

    return xor1 ^ xor2;
}
```

Approach 2: Sum Method

```
public static int missingNumberSum(int[] arr) {
    int n = arr.length + 1;
    long expectedSum = (long) n * (n + 1) / 2;
    long actualSum = 0;

    for (int num : arr) {
        actualSum += num;
    }

    return (int) (expectedSum - actualSum);
}
```

---

8. Majority Element

Difficulty: Medium | **Time:** O(n) | **Space:** O(1)

Approach 1: Boyer-Moore Voting Algorithm

```
public static int majorityElement(int[] arr) {
    int candidate = arr[0];
    int count = 1;

    for (int i = 1; i < arr.length; i++) {
        if (arr[i] == candidate) {
            count++;
        } else {
            count--;
            if (count == 0) {
                candidate = arr[i];
                count = 1;
            }
        }
    }

    return candidate;
}
```

Approach 2: Using HashMap

```
public static int majorityElementMap(int[] arr) {
    Map<Integer, Integer> map = new HashMap<>();
    int majority = arr.length / 2;

    for (int num : arr) {
        map.put(num, map.getOrDefault(num, 0) + 1);
        if (map.get(num) > majority) {
            return num;
        }
    }

    return -1;
}
```

□ STUDY TIPS & PATTERNS

Time Complexity Cheat Sheet

```
O(1) - Constant: Array access, HashMap operations
O(log n) - Logarithmic: Binary search
O(n) - Linear: Single loop
O(n log n) - Linearithmic: Merge sort, Quick sort
O(n²) - Quadratic: Nested loops
O(2^n) - Exponential: Recursion without memoization
O(n!) - Factorial: Permutations
```

Common Patterns

Pattern	Problems	Time
Two Pointer	Reverse, Merge, Container	O(n)
HashMap/HashSet	Duplicates, Complement	O(n)
Binary Search	Sorted Array, Insertion	O(log n)
Dynamic Programming	Optimization, Counting	Varies
Bit Manipulation	XOR, Single Number	O(n)
Kadane's	Max Subarray	O(n)
Sliding Window	Substring Problems	O(n)
Recursion + Memoization	Fibonacci, DP	O(n)

Interview Tips

1. **Understand the problem first** - Ask clarifying questions
2. **Think of edge cases** - Empty, null, duplicates, overflow
3. **Discuss approach before coding** - Brute force → Optimized
4. **Write clean code** - Good variable names, comments

5. **Test with examples** - Small and large test cases
6. **Mention complexity** - Time and Space
7. **Optimize iteratively** - Don't jump to best solution

Code Quality Checklist

□ Handle null/empty inputs

□ Check for integer overflow

□ Use appropriate data structures

□ Avoid deep recursion (stack overflow)

□ Comments for complex logic

□ Test edge cases

□ Clean variable names

□ Mention complexity

Good Luck with your PG-DAC Placement! □