Hashing Question

Find All Anagrams in a String: Given a string s and a pattern p, find all start indices of p's anagrams in s.

First Non-Repeating Character: Find the first non-repeating character in a string.

Group Anagrams: Given an array of strings, group the anagrams together.

Count Distinct Elements in Every Window of Size K: Given an array and an integer k, count the number of distinct elements in every window of size k.

Longest Consecutive Sequence: Find the length of the longest consecutive elements sequence in an unsorted array.

Subarray with Given Sum: Given an array of positive integers and a sum, find the subarray with the given sum.

Find Duplicate Number: Given an array of integers where each integer is between 1 and n (inclusive), find the duplicate number.

Check for Subarray Sum Equal to K: Given an array of integers and a sum k, check if there is a subarray with a sum equal to k.

Find Common Elements in Three Sorted Arrays: Given three sorted arrays, find the common elements in all three arrays.

Longest Substring Without Repeating Characters: Given a string, find the length of the longest substring without repeating characters.

explain hashing problems with example that contains input and output in simple way and give optimized and simple java code with its time complexity

Memory updated

Here are explanations of the hashing problems you mentioned, with examples and optimized Java code:

1. Find All Anagrams in a String

Problem: Given a string `s` and a pattern `p`, find all start indices of `p`'s anagrams in `s`.
Example:

- **Input**: `s = "cbaebabacd"`, `p = "abc"`

- **Output**: `[0, 6]`
  Anagrams of `"abc"` are `"cba"` (at index 0) and `"bac"` (at index 6).

Approach: Use a sliding window and a hash map to compare the frequency of characters in the current window with the frequency of characters in the pattern.

```java
import java.util.*;

public class FindAllAnagrams {
    public List<Integer> findAnagrams(String s, String p) {
        List<Integer> result = new ArrayList<>();
        if (s.length() < p.length()) return result;

        int[] pCount = new int[26];
        int[] sCount = new int[26];

        for (int i = 0; i < p.length(); i++) {
            pCount[p.charAt(i) - 'a']++;
            sCount[s.charAt(i) - 'a']++;
        }

        if (Arrays.equals(pCount, sCount)) result.add(0);

        for (int i = p.length(); i < s.length(); i++) {
            sCount[s.charAt(i) - 'a']++;
            sCount[s.charAt(i - p.length()) - 'a']--;

            if (Arrays.equals(pCount, sCount)) result.add(i - p.length() + 1);
        }

        return result;
    }

    public static void main(String[] args) {
        FindAllAnagrams obj = new FindAllAnagrams();
        System.out.println(obj.findAnagrams("cbaebabacd", "abc")); // Output: [0, 6]
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the string `s`.

---

2. First Non-Repeating Character

Problem: Find the first non-repeating character in a string.
Example:

- **Input**: `"leetcode"`
- **Output**: `l`

Approach: Use a hash map to store the frequency of each character, then find the first character with a frequency of 1.

```java
import java.util.*;

public class FirstNonRepeatingCharacter {
    public char firstUniqChar(String s) {
        Map<Character, Integer> countMap = new HashMap<>();
```

```
        for (char c : s.toCharArray()) {
            countMap.put(c, countMap.getOrDefault(c, 0) + 1);
        }

        for (char c : s.toCharArray()) {
            if (countMap.get(c) == 1) return c;
        }

        return '_'; // return '_' if no unique character is found
    }

    public static void main(String[] args) {
        FirstNonRepeatingCharacter obj = new FirstNonRepeatingCharacter();
        System.out.println(obj.firstUniqChar("leetcode")); // Output: l
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the string.

---

3. Group Anagrams

Problem: Given an array of strings, group the anagrams together.
Example:

- **Input**: `["eat", "tea", "tan", "ate", "nat", "bat"]`
- **Output**: `[["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]`

Approach: Use a hash map to group strings that, when sorted, are the same.

java

```
import java.util.*;

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

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

            map.putIfAbsent(key, new ArrayList<>());
            map.get(key).add(s);
        }

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

    public static void main(String[] args) {
        GroupAnagrams obj = new GroupAnagrams();
        String[] strs = {"eat", "tea", "tan", "ate", "nat", "bat"};
        System.out.println(obj.groupAnagrams(strs));
        // Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]
    }
}
```

Time Complexity: `O(n * k log k)` where `n` is the number of strings and `k` is the maximum length of a string.

---

4. Count Distinct Elements in Every Window of Size K

Problem: Given an array and an integer `k`, count the number of distinct elements in every window of size `k`.
Example:

- **Input**: `arr = [1, 2, 1, 3, 4, 2, 3]`, `k = 4`
- **Output**: `[3, 4, 4, 3]`

Approach: Use a hash map to track the frequency of elements in the current window and slide the window one element at a time.

java

```java
import java.util.*;

public class DistinctElementsInWindow {
    public List<Integer> countDistinct(int[] arr, int k) {
        List<Integer> result = new ArrayList<>();
        Map<Integer, Integer> countMap = new HashMap<>();

        for (int i = 0; i < k; i++) {
            countMap.put(arr[i], countMap.getOrDefault(arr[i], 0) + 1);
        }
        result.add(countMap.size());

        for (int i = k; i < arr.length; i++) {
            int removeElement = arr[i - k];
            if (countMap.get(removeElement) == 1) countMap.remove(removeElement);
            else countMap.put(removeElement, countMap.get(removeElement) - 1);

            countMap.put(arr[i], countMap.getOrDefault(arr[i], 0) + 1);
            result.add(countMap.size());
        }

        return result;
    }

    public static void main(String[] args) {
        DistinctElementsInWindow obj = new DistinctElementsInWindow();
        int[] arr = {1, 2, 1, 3, 4, 2, 3};
        System.out.println(obj.countDistinct(arr, 4)); // Output: [3, 4, 4, 3]
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

Let me know if you'd like similar explanations for the remaining problems!

give remaining

Here are explanations and Java code for the remaining problems:

5. Longest Consecutive Sequence

Problem: Find the length of the longest consecutive elements sequence in an unsorted array.
Example:

- **Input**: `[100, 4, 200, 1, 3, 2]`
- **Output**: `4` (The longest consecutive sequence is `[1, 2, 3, 4]`)

Approach: Use a hash set to check for the start of a sequence and then count the consecutive elements.

```java
import java.util.*;

public class LongestConsecutiveSequence {
    public int longestConsecutive(int[] nums) {
        Set<Integer> numSet = new HashSet<>();
        for (int num : nums) {
            numSet.add(num);
        }

        int longestStreak = 0;

        for (int num : numSet) {
            if (!numSet.contains(num - 1)) { // Check if it's the start of a sequence
                int currentNum = num;
                int currentStreak = 1;

                while (numSet.contains(currentNum + 1)) {
                    currentNum++;
                    currentStreak++;
                }

                longestStreak = Math.max(longestStreak, currentStreak);
            }
        }

        return longestStreak;
    }

    public static void main(String[] args) {
        LongestConsecutiveSequence obj = new LongestConsecutiveSequence();
        int[] nums = {100, 4, 200, 1, 3, 2};
        System.out.println(obj.longestConsecutive(nums)); // Output: 4
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

6. Subarray with Given Sum

Problem: Given an array of positive integers and a sum, find the subarray with the given sum.
Example:

- **Input**: `arr = [1, 2, 3, 7, 5]`, `sum = 12`
- **Output**: `[2, 4]` (Subarray is `[3, 7, 2]`)

Approach: Use a sliding window approach to find the subarray.

java

```java
public class SubarrayWithGivenSum {
    public int[] findSubarray(int[] arr, int sum) {
        int currSum = 0;
        int start = 0;

        for (int end = 0; end < arr.length; end++) {
            currSum += arr[end];

            while (currSum > sum) {
                currSum -= arr[start];
                start++;
            }

            if (currSum == sum) {
                return new int[]{start + 1, end + 1}; // 1-based index
            }
        }

        return new int[]{-1}; // No subarray found
    }

    public static void main(String[] args) {
        SubarrayWithGivenSum obj = new SubarrayWithGivenSum();
        int[] arr = {1, 2, 3, 7, 5};
        int[] result = obj.findSubarray(arr, 12);
        for (int i : result) {
            System.out.print(i + " ");
        }
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

7. Find Duplicate Number

Problem: Given an array of integers where each integer is between 1 and `n` (inclusive), find the duplicate number.
Example:

- **Input**: `[1, 3, 4, 2, 2]`
- **Output**: `2`

Approach: Use Floyd's Tortoise and Hare (Cycle Detection) algorithm.

java

```java
public class FindDuplicateNumber {
    public int findDuplicate(int[] nums) {
        int slow = nums[0];
        int fast = nums[0];

        // Detect cycle
        do {
            slow = nums[slow];
            fast = nums[nums[fast]];
        } while (slow != fast);

        // Find entry point of cycle
        slow = nums[0];
        while (slow != fast) {
            slow = nums[slow];
```

```
            fast = nums[fast];
        }

        return slow;
    }

    public static void main(String[] args) {
        FindDuplicateNumber obj = new FindDuplicateNumber();
        int[] nums = {1, 3, 4, 2, 2};
        System.out.println(obj.findDuplicate(nums)); // Output: 2
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

8. Check for Subarray Sum Equal to K

Problem: Given an array of integers and a sum `k`, check if there is a subarray with a sum equal to `k`.
Example:

- **Input**: `arr = [1, 2, 3]`, `k = 3`
- **Output**: `True` (Subarray `[1, 2]` has sum 3)

Approach: Use a hash map to track the cumulative sum and check for the difference with `k`.

java

```
import java.util.*;

public class SubarraySumEqualsK {
    public boolean checkSubarraySum(int[] nums, int k) {
        Map<Integer, Integer> sumMap = new HashMap<>();
        int sum = 0;
        sumMap.put(0, -1);

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

            if (sumMap.containsKey(sum - k)) {
                return true;
            }

            sumMap.put(sum, i);
        }

        return false;
    }

    public static void main(String[] args) {
        SubarraySumEqualsK obj = new SubarraySumEqualsK();
        int[] nums = {1, 2, 3};
        System.out.println(obj.checkSubarraySum(nums, 3)); // Output: true
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

9. Find Common Elements in Three Sorted Arrays

Problem: Given three sorted arrays, find the common elements in all three arrays.
Example:

- **Input**: `arr1 = [1, 5, 10, 20, 40, 80]`, `arr2 = [6, 7, 20, 80, 100]`, `arr3 = [3, 4, 15, 20, 30, 70, 80, 120]`
- **Output**: `[20, 80]`

Approach: Use three pointers to compare elements in all arrays.

```java
import java.util.*;

public class CommonElementsThreeArrays {
    public List<Integer> findCommon(int[] arr1, int[] arr2, int[] arr3) {
        List<Integer> result = new ArrayList<>();
        int i = 0, j = 0, k = 0;

        while (i < arr1.length && j < arr2.length && k < arr3.length) {
            if (arr1[i] == arr2[j] && arr2[j] == arr3[k]) {
                result.add(arr1[i]);
                i++; j++; k++;
            } else if (arr1[i] < arr2[j]) i++;
            else if (arr2[j] < arr3[k]) j++;
            else k++;
        }

        return result;
    }

    public static void main(String[] args) {
        CommonElementsThreeArrays obj = new CommonElementsThreeArrays();
        int[] arr1 = {1, 5, 10, 20, 40, 80};
        int[] arr2 = {6, 7, 20, 80, 100};
        int[] arr3 = {3, 4, 15, 20, 30, 70, 80, 120};
        System.out.println(obj.findCommon(arr1, arr2, arr3)); // Output: [20, 80]
    }
}
```

Time Complexity: `O(n1 + n2 + n3)` where `n1`, `n2`, and `n3` are the lengths of the arrays.

---

10. Longest Substring Without Repeating Characters

Problem: Given a string, find the length of the longest substring without repeating characters.
Example:

- **Input**: `"abcabcbb"`
- **Output**: `3` (The longest substring is `"abc"`)

Approach: Use a sliding window and a hash map to track the last seen index of each character.

```java
import java.util.*;

public class LongestSubstringWithoutRepeating {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> map = new HashMap<>();
        int maxLen = 0, start = 0;

        for (int end = 0; end < s.length(); end++) {
```

```
                char c = s.charAt(end);
                if (map.containsKey(c)) {
                    start = Math.max(start, map.get(c) + 1);
                }
                map.put(c, end);
                maxLen = Math.max(maxLen, end - start + 1);
            }

            return maxLen;
        }

        public static void main(String[] args) {
            LongestSubstringWithoutRepeating obj = new LongestSubstringWithoutRepeating();
            System.out.println(obj.lengthOfLongestSubstring("abcabcbb")); // Output: 3
        }
    }
```

Time Complexity: `O(n)` where `n` is the length of the string.

These are optimized and simple Java implementations for the remaining problems with explanations and time complexities. Let me know if you need further clarification!

Count Occurrences of Each Character: Given a string, count the occurrences of each character.

Find the Kth Largest Element in an Array: Find the Kth largest element in an unsorted array.

Find the Pair with the Given Sum: Given an array and a sum, find if there exists a pair in the array that adds up to the given sum.

Find the Majority Element: Given an array, find the element that appears more than n/2 times (Majority Element).

Subarray Sum Equals K: Find the number of subarrays that sum up to k.

Find Missing Number: Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array.

Check if Two Strings are Isomorphic: Determine if two strings are isomorphic, i.e., they can be transformed into each other by character replacement.

Longest Palindromic Substring: Find the longest substring that is a palindrome.

Count Subarrays with Zero Sum: Count the number of subarrays whose sum is zero.

Find the Intersection of Two Arrays: Given two arrays, find their intersection.

Here are the Java solutions with explanations for the requested problems:

1. Count Occurrences of Each Character

Problem: Given a string, count the occurrences of each character.
Example:

- **Input**: `"apple"`
- **Output**: `a: 1, p: 2, l: 1, e: 1`

Approach: Use a hash map to store the frequency of each character.

```java
import java.util.*;

public class CountOccurrences {
    public Map<Character, Integer> countOccurrences(String s) {
        Map<Character, Integer> frequencyMap = new HashMap<>();

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

        return frequencyMap;
    }

    public static void main(String[] args) {
        CountOccurrences obj = new CountOccurrences();
        String str = "apple";
        System.out.println(obj.countOccurrences(str)); // Output: {a=1, p=2, l=1, e=1}
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the string.

---

2. Find the Kth Largest Element in an Array

Problem: Find the Kth largest element in an unsorted array.
Example:

- **Input**: `arr = [3, 2, 1, 5, 6, 4], k = 2`
- **Output**: `5`

Approach: Use a min-heap of size `k` to keep track of the Kth largest element.

```java
import java.util.*;

public class KthLargestElement {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.add(num);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }

        return minHeap.peek();
    }

    public static void main(String[] args) {
```

```
        KthLargestElement obj = new KthLargestElement();
        int[] nums = {3, 2, 1, 5, 6, 4};
        System.out.println(obj.findKthLargest(nums, 2)); // Output: 5
    }
}
```

Time Complexity: `O(n log k)` where `n` is the length of the array.

---

3. Find the Pair with the Given Sum

Problem: Given an array and a sum, find if there exists a pair in the array that adds up to the given sum.
Example:

- **Input**: `arr = [2, 7, 11, 15], target = 9`
- **Output**: `[0, 1]` (Pair is 2 and 7)

Approach: Use a hash map to check if the complement exists.

java

```
import java.util.*;

public class PairWithSum {
    public int[] findPair(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];
            if (map.containsKey(complement)) {
                return new int[]{map.get(complement), i};
            }
            map.put(nums[i], i);
        }

        return new int[]{-1, -1}; // No pair found
    }

    public static void main(String[] args) {
        PairWithSum obj = new PairWithSum();
        int[] nums = {2, 7, 11, 15};
        int[] result = obj.findPair(nums, 9);
        System.out.println(Arrays.toString(result)); // Output: [0, 1]
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

4. Find the Majority Element

Problem: Given an array, find the element that appears more than `n/2` times.
Example:

- **Input**: `arr = [2, 2, 1, 1, 1, 2, 2]`
- **Output**: `2`

Approach: Use the Boyer-Moore Voting Algorithm.

java

```java
public class MajorityElement {
    public int findMajorityElement(int[] nums) {
        int count = 0;
        Integer candidate = null;

        for (int num : nums) {
            if (count == 0) {
                candidate = num;
            }
            count += (num == candidate) ? 1 : -1;
        }

        return candidate;
    }

    public static void main(String[] args) {
        MajorityElement obj = new MajorityElement();
        int[] nums = {2, 2, 1, 1, 1, 2, 2};
        System.out.println(obj.findMajorityElement(nums)); // Output: 2
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

5. Subarray Sum Equals K

Problem: Find the number of subarrays that sum up to `k`.
Example:

- **Input**: `arr = [1, 1, 1], k = 2`
- **Output**: `2`

Approach: Use a hash map to store cumulative sums and their frequencies.

java

```java
import java.util.*;

public class SubarraySumK {
    public int subarraySum(int[] nums, int k) {
        Map<Integer, Integer> sumMap = new HashMap<>();
        sumMap.put(0, 1);

        int sum = 0, count = 0;

        for (int num : nums) {
            sum += num;
            if (sumMap.containsKey(sum - k)) {
                count += sumMap.get(sum - k);
            }
            sumMap.put(sum, sumMap.getOrDefault(sum, 0) + 1);
        }

        return count;
    }

    public static void main(String[] args) {
        SubarraySumK obj = new SubarraySumK();
        int[] nums = {1, 1, 1};
        System.out.println(obj.subarraySum(nums, 2)); // Output: 2
```

```
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

6. Find Missing Number

Problem: Given an array containing `n` distinct numbers taken from `0, 1, 2, ..., n`, find the missing number.
Example:

- **Input**: `arr = [3, 0, 1]`
- **Output**: `2`

Approach: Use the formula for the sum of first `n` numbers: `n * (n + 1) / 2`.

```java
public class MissingNumber {
    public int findMissingNumber(int[] nums) {
        int n = nums.length;
        int expectedSum = n * (n + 1) / 2;
        int actualSum = 0;

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

        return expectedSum - actualSum;
    }

    public static void main(String[] args) {
        MissingNumber obj = new MissingNumber();
        int[] nums = {3, 0, 1};
        System.out.println(obj.findMissingNumber(nums)); // Output: 2
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

7. Check if Two Strings are Isomorphic

Problem: Determine if two strings are isomorphic, i.e., they can be transformed into each other by character replacement.
Example:

- **Input**: `s = "egg", t = "add"`
- **Output**: `true`

Approach: Use two hash maps to map characters from `s` to `t` and vice versa.

```java
import java.util.*;

public class IsomorphicStrings {
```

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

        Map<Character, Character> mapS = new HashMap<>();
        Map<Character, Character> mapT = new HashMap<>();

        for (int i = 0; i < s.length(); i++) {
            char c1 = s.charAt(i);
            char c2 = t.charAt(i);

            if (mapS.containsKey(c1)) {
                if (mapS.get(c1) != c2) return false;
            } else {
                mapS.put(c1, c2);
            }

            if (mapT.containsKey(c2)) {
                if (mapT.get(c2) != c1) return false;
            } else {
                mapT.put(c2, c1);
            }
        }

        return true;
    }

    public static void main(String[] args) {
        IsomorphicStrings obj = new IsomorphicStrings();
        System.out.println(obj.isIsomorphic("egg", "add")); // Output: true
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the strings.

---

8. Longest Palindromic Substring

Problem: Find the longest substring that is a palindrome.
Example:

- **Input**: `"babad"`
- **Output**: `"bab"`

Approach: Use dynamic programming to find the longest palindromic substring.

```java
public class LongestPalindromicSubstring {
    public String longestPalindrome(String s) {
        int n = s.length();
        boolean[][] dp = new boolean[n][n];
        int start = 0, maxLen = 1;

        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }

        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                if (s.charAt(i) == s.charAt(j) && (len == 2 || dp[i + 1][j - 1])) {
                    dp[i][j] = true;
                    if (len > maxLen) {
                        start = i;
                        maxLen = len;
                    }
                }
```

```
                }
            }

            return s.substring(start, start + maxLen);
        }

        public static void main(String[] args) {
            LongestPalindromicSubstring obj = new LongestPalindromicSubstring();
            System.out.println(obj.longestPalindrome("babad")); // Output: "bab"
        }
    }
```

Time Complexity: `O(n^2)` where `n` is the length of the string.

---

9. Count Subarrays with Zero Sum

Problem: Count the number of subarrays whose sum is zero.
Example:

- **Input**: `arr = [0, 0, 5, 5, 0, 0]`
- **Output**: `6`

Approach: Use a hash map to keep track of cumulative sums and count zero sum subarrays.

java

```
import java.util.*;

public class CountZeroSumSubarrays {
    public int countSubarraysWithZeroSum(int[] nums) {
        Map<Integer, Integer> sumMap = new HashMap<>();
        sumMap.put(0, 1);

        int sum = 0, count = 0;

        for (int num : nums) {
            sum += num;
            count += sumMap.getOrDefault(sum, 0);
            sumMap.put(sum, sumMap.getOrDefault(sum, 0) + 1);
        }

        return count;
    }

    public static void main(String[] args) {
        CountZeroSumSubarrays obj = new CountZeroSumSubarrays();
        int[] nums = {0, 0, 5, 5, 0, 0};
        System.out.println(obj.countSubarraysWithZeroSum(nums)); // Output: 6
    }
}
```

Time Complexity: `O(n)` where `n` is the length of the array.

---

10. Find the Intersection of Two Arrays

Problem: Given two arrays, find their intersection.
Example:

- **Input**: `arr1 = [1, 2, 2, 1], arr2 = [2, 2]`

- **Output**: `[2]`

Approach: Use a hash set to store the unique elements from one array and find the common elements.

```java
import java.util.*;

public class IntersectionOfArrays {
    public int[] intersection(int[] nums1, int[] nums2) {
        Set<Integer> set1 = new HashSet<>();
        for (int num : nums1) {
            set1.add(num);
        }

        Set<Integer> intersection = new HashSet<>();
        for (int num : nums2) {
            if (set1.contains(num)) {
                intersection.add(num);
            }
        }

        int[] result = new int[intersection.size()];
        int i = 0;
        for (int num : intersection) {
            result[i++] = num;
        }

        return result;
    }

    public static void main(String[] args) {
        IntersectionOfArrays obj = new IntersectionOfArrays();
        int[] nums1 = {1, 2, 2, 1};
        int[] nums2 = {2, 2};
        System.out.println(Arrays.toString(obj.intersection(nums1, nums2))); // Output:
[2]
    }
}
```

Time Complexity: `O(n + m)` where `n` is the length of the first array and `m` is the length of the second array.

These solutions offer clear and optimized Java implementations for the requested problems, with explanations and time complexities provided. Let me know if you need further clarifications!