Find intersection of two Arrays CodeStudio

Given two arrays arr1 and arr2, find their intersection, i.e., the elements that are common to both arrays.

Input:

1 2 2 2 3 4

2 2 3 3

Output: 2 2 3

Input:

1 2 3

3 4

Output: 3

Solution 1: Brute Force Approach

The arrayIntersection function implements a brute force approach to find the intersection. It iterates over each element of arr1 and arr2 and checks for equality. When a match is found, the element is added to the ans vector.

The time complexity of this approach is O(n * m), where n and m are the sizes of arr1 and arr2, respectively.

The space complexity is O(1) as the ans vector stores the intersection elements.

Solution 2: Two-Pointer Approach

The arrayIntersectionOptimized function uses a two-pointer approach to find the intersection. It assumes that both input arrays are sorted in non-decreasing order. The two pointers, i and j, traverse arr1 and arr2 respectively. If the elements at the current pointers are equal, it means they are part of the intersection, so they are added to the ans vector. If the element in arr1 is smaller, i is incremented. Otherwise, j is incremented.

This approach eliminates unnecessary comparisons, resulting in a **time complexity of O(n + m), where n and m are the sizes of arr1 and arr2**, respectively.

The space complexity remains O(1) as the ans vector stores the intersection elements.

Solution 3: Hash Map Approach

The arrayIntersectionHashMap function utilizes a hash map to find the intersection. It counts the frequency of elements in arr1 using an unordered map named count. Then, it iterates over arr2 and checks if each element exists in the count map. If the element is present, it is added to the ans vector, and the count is decremented.

This approach has a time complexity of O(n + m) since iterating over both arr1 and arr2 takes O(n + m) time.

The space complexity is O(n) as the hash map stores the frequency of elements in arr1.

Among the given solutions, the Two-Pointer Approach (arrayIntersectionOptimized) is the most optimal one, offering a time complexity of O(n + m) and a space complexity of O(1). It performs well when both arrays are sorted, as it eliminates unnecessary comparisons by taking advantage of the sorted order.