Find Duplicate Element CodeStudio

You are given an array of size n containing integers from 1 to n-1, inclusive. There is exactly one element in the array that appears more than once. You need to find and return the duplicate element.

Note:

You must solve the problem in linear time complexity and constant space complexity.

Input: The input array arr contains elements {3, 1, 1, 2, 4}.

Output: The output will be: The duplicate element is: 1

Approach 1: Brute Force Approach

This approach uses a nested loop to compare each element with every other element in the array. When a duplicate element is found, it is immediately returned.

This approach has a **time complexity of O(n^2)**, where n is the size of the array.

The **space complexity is O(1)** since no additional data structures are used.

Approach 2: Hash Map Approach

This approach utilizes an unordered map to count the occurrences of each element in the array. It iterates over the array and updates the frequency of each element in the unique unordered map. Afterward, it checks the frequency of each element in the map and returns the first element with a count greater than 1, indicating a duplicate.

The **time complexity of this approach is O(n)** for iterating over the array and O(1) for checking the frequency in the map. Thus, **the overall time complexity is O(n),**

The **space complexity is O(n)** to store the elements and their frequencies in the unordered map.

Approach 3: Floyd's Tortoise and Hare algorithm, or the Fast and Slow Pointers technique.

This solution uses two pointers: a slow pointer that moves one step at a time and a fast pointer that moves two steps at a time. By traversing the array, the two pointers will eventually meet at a common element within the cycle.

Afterward, we reset the slow pointer to the starting point and move both the slow and fast pointers one step at a time until they meet again. The element at this meeting point is the duplicate element.

The optimized solution has a time complexity of O(n) and a space complexity of O(1), making it more efficient than the previous approaches.