Documentation: Find All Duplicates in an Array

Table of Contents

1. **Problem Statement**
2. **Intuition**
3. **Key Observations**
4. **Approach**
5. **Edge Cases**
6. **Complexity Analysis**
   - Time Complexity
   - Space Complexity
7. **Alternative Approaches**
8. **Test Cases**
9. **Final Thoughts**

1. Problem Statement

Given an integer array nums of length n, where each element is in the range [1, n], and each integer appears at most twice, the task is to find all elements that appear exactly twice in the array and return them as an array.

The algorithm must run in O(n) time and use only constant auxiliary space (excluding the space needed to store the output).

2. Intuition

To solve this problem efficiently, we need to find a way to track the frequency of elements without using extra space. A clever way is to leverage the given constraint that all elements are within the range [1, n] by marking elements in the array itself.

3. **Key Observations**

- All elements in the array are in the range [1, n], meaning the value of any element can be used as an index to track whether it has been seen before.
- If an element appears twice, the second time we encounter it, we can identify it as a duplicate.
- Modifying the array itself allows us to avoid using additional space for counting the elements.

4. **Approach**

The approach follows these steps:

   a. Traverse through the array: For each element x, we calculate its corresponding index x - 1.
   b. Marking visited elements:
      i. If the number at the index x - 1 is negative, this means we have already encountered the element x before, so it is a duplicate.
      ii. If the number at the index x - 1 is positive, we mark it as visited by negating the number at that index.
   c. Return duplicates: If the number at index x - 1 is negative when we first encounter x, it means x is a duplicate, and we add it to the result list.

This solution uses the array itself for marking visited indices, which ensures that the space complexity remains constant.

5. **Edge Cases**

- Empty Array: If the array is empty, the result should be an empty list as there are no elements to duplicate.
- Array with all unique elements: If there are no duplicates, the result should also be an empty list.
- Array with exactly two identical elements: The result should return that single element as the only duplicate.
- Array with all elements duplicated: In the case where every element appears twice, the result should include all elements.

6. **Complexity Analysis**

Time Complexity: O(n)

- We iterate over the array once, making constant-time operations for each element.
- Hence, the time complexity is O(n), where n is the length of the array.

Space Complexity: O(1)

- The solution uses constant space, excluding the space needed for the output array.
- We modify the input array in place and use only a constant amount of extra space.

7. **Alternative Approaches**

Using a Hash Map:

- One alternative approach could be using a hash map to store the frequency of each element. After processing all elements, we can iterate through the hash map and return the elements with a count of 2.
- Time Complexity: O(n) (for both insertion and lookup in the hash map).
- Space Complexity: O(n) (to store the frequency of each element).

While this approach works, it uses extra space (a hash map), which violates the problem's requirement of constant auxiliary space.

8. **Test Cases**

Test Case 1: Example Case 1

Input: [4, 3, 2, 7, 8, 2, 3, 1]
Output: [2, 3]

Test Case 2: Example Case 2

Input: [1, 1, 2]
Output: [1]

Test Case 3: Example Case 3

Input: [1]
Output: []

Test Case 4: Edge Case (All unique elements)

Input: [1, 2, 3, 4, 5]
Output: []

Test Case 5: Edge Case (All elements are duplicates)

Input: [1, 1, 2, 2, 3, 3]
Output: [1, 2, 3]

Test Case 6: Edge Case (Empty array)

Input: []
Output: []

9. Final Thoughts

This solution provides an efficient way to find duplicates in an array with a time complexity of O(n) and constant extra space. By cleverly using the array itself to mark visited indices, we avoid the need for additional data structures like hash maps. The approach is not only optimal in terms of space but also simple and easy to implement. For large inputs, this solution is highly scalable and adheres to the problem's constraints.