442. Find All Duplicates in an Array

Medium Array Hash Table Leetcode Link

Problem Description

This problem asks us to find all the numbers that appear twice in an array `nums`, where the array has a length `n`, and all elements in the array are within the range `[1, n]`. The elements in the array can either appear once or twice. The challenge is to come up with an algorithm that finds the numbers appearing twice without using more than constant extra space, and it should have a linear run time, i.e., `O(n)`.

Intuition

The solution utilizes the properties of the array elements being within the range from 1 to n. The algorithm works by placing each number in its correct position, i.e., the number 1 should be in index 0, 2 should be in index 1, and so forth. This is done by swapping the elements until each index `i` contains the number `i+1`.

The intuition here is that if all numbers appear only once, then after this process, each index `i` should hold the value `i+1`. If a number appears twice, then there will be at least one discrepancy where the value at `nums[i]` wouldn't match `i+1`.

1. Iterate through each element in the array.
2. For each element, check if it is not in its correct position, i.e., `nums[i]` is not equal to `nums[nums[i] - 1]`. If it's not, swap the elements in the current index `i` and the index that the current number should be at, which is `nums[i] - 1`.
3. Repeat this step until every number in the array is either at its correct index or there's a duplicate that prevents it from being placed correctly.
4. Once the array is sorted in this manner, we can easily find duplicates because the number will not match its index +1. The list comprehension `[v for i, v in enumerate(nums) if v != i + 1]` does exactly that, creating a list of values that do not match `i+1`, which are the duplicates we're looking for.

By using array indices to store the state of whether a number has been seen or not, we achieve the goal using constant space `(O(1))`, while the swapping ensures we can detect duplicates efficiently.

Solution Approach

The algorithm operates based on the notion that if we could place each number at its correct index in the array, we can then just iterate through the array to find numbers that are not at their correct indices.

The steps involved in solving this problem are:

1. Loop over each index `i` in the array `nums`.
2. Inside the loop, we initiate another loop that swaps the current element `nums[i]` with the element at the index it should be at, which is `nums[nums[i] - 1]`. This continues as long as `nums[i]` is not already at the correct position.
3. To swap, we perform a tuple assignment `nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]`, which swaps the elements in-place without the need for extra storage.
4. After completing the outer loop, we know that for each index `i`, if `nums[i]` does not equal `i + 1`, it must be a duplicate because there can only be one such case per number, considering that elements are strictly in the range `[1, n]`.
5. We construct the result array by iterating over `nums` with the condition `if v != i + 1`, using a list comprehension that iterates over `nums` and its indices (using `enumerate`), and creates a list of the values `v` that do not match their supposed index `i+1`.

This approach utilizes in-place swapping to achieve the sorting, which ensures that we are not using any additional space; thus, it adheres to the constant space complexity restriction. The solution guarantees that we do not re-visit any number more than twice, maintaining the `O(n)` time complexity. Once an element is in its correct position, it's not touched again, and discovering the duplicate takes a single pass through the sorted array.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Consider the array `nums = [4, 3, 2, 7, 8, 2, 3, 1]`.

According to the algorithm:

1. We start with the first element: `4`. Since `4` should be at index `3` (`nums[3]`), we swap it with the element at index `3`, which is `7`. The array now looks like `[7, 3, 2, 4, 8, 2, 3, 1]`.

2. We still are at index `0`, and now we have `7` there, which should be at index `6`. After swapping `7` with `3` (at index `6`), the array is `[3, 3, 2, 4, 8, 2, 7, 1]`.

3. At index `0`, there's `3` that should be at index `2`. But index `2` also has `2`, which is the correct element for that position. Hence, we proceed to the next index.

4. At index `1`, we also have `3`, which is out of place, reflecting a duplicate has been found. However, `3`'s correct spot (index `2`) already has a `2` positioned correctly, so we move on.

5. Proceed with the other elements until each index `i` contains the number `i+1` or it's determined that a duplication prevents it from being in the correct position.

After the outer loop is completed, the array is `[1, 2, 3, 4, 3, 2, 7, 8]`. Following step 5, we will iterate through the array and identify the numbers that are not in their correct positions. Here, `3` at index `4` should have been `5`, and `2` at index `5` should have been `6`. Hence, these are our duplicates.

So, the output according to the algorithm is `[3, 2]`, which accurately reflects the numbers that appear twice in the array.

Python Solution

```python
from typing import List

class Solution:
    def findDuplicates(self, numbers: List[int]) -> List[int]:
        n = len(numbers)
        # Iterate over the numbers
        for i in range(n):
            # Place each number at its correct position (number-1)
            # since the numbers are from 1 to n
            while numbers[i] != numbers[numbers[i] - 1]:
                # Swap the elements to their correct positions
                correct_idx = numbers[i] - 1
                numbers[i], numbers[correct_idx] = numbers[correct_idx], numbers[i]

        # Now, find all the numbers that are not at their correct position
        # which will be our duplicates since those have been
        # placed correctly during the previous loop
        return [number for i, number in enumerate(numbers) if number != i + 1]

```

Java Solution

```java
import java.util.ArrayList;
import java.util.List;

class Solution {

    // Main method to find all the duplicates in the array.
    public List<Integer> findDuplicates(int[] nums) {
        int n = nums.length;

        // Place each number in its correct position such that the number i is at index i-1.
        for (int i = 0; i < n; ++i) {
            // While the current number is not at its correct position, swap it.
            while (nums[i] != nums[nums[i] - 1]) {
                swap(nums, i, nums[i] - 1);
            }
        }

        // Initialize the list to hold the duplicates.
        List<Integer> duplicates = new ArrayList<>();

        // Scan the array for duplicates; a duplicate is found if the number is not at its correct position.
        for (int i = 0; i < n; ++i) {
            if (nums[i] != i + 1) {
                duplicates.add(nums[i]);
            }
        }

        // Return the list of duplicates.
        return duplicates;
    }

    // Helper method to swap two elements in the array.
    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For std::swap

class Solution {
public:
    // Function to find all duplicates in an array.
    // Each element appears either once or twice, and elements are in the range [1, n].
    std::vector<int> findDuplicates(std::vector<int>& nums) {
        int size = nums.size();

        // Reorder the array such that the number `i` is placed at the index `i - 1`
        for (int i = 0; i < size; ++i) {
            // Swap elements until the current element is at its correct position.
            while (nums[i] != nums[nums[i] - 1]) {
                std::swap(nums[i], nums[nums[i] - 1]);
            }
        }

        std::vector<int> duplicates; // Vector to hold the duplicates found
        for (int i = 0; i < size; ++i) {
            // If current element is not at its correct position, it must be a duplicate
            if (nums[i] != i + 1) {
                duplicates.push_back(nums[i]); // Record the duplicate
            }
        }

        // Return the vector containing all the duplicates
        return duplicates;
    }
};

```

Typescript Solution

```typescript
function findDuplicates(nums: number[]): number[] {
  const size = nums.length;

  // Reorder the array such that the number `i` will be placed at the index `i - 1`
  for (let i = 0; i < size; ++i) {
    // Keep swapping elements until the current element is at its correct position
    while (nums[i] !== nums[nums[i] - 1]) {
      // Swap nums[i] with the element at its target position
      const temp = nums[i];
      nums[i] = nums[nums[i] - 1];
      nums[temp - 1] = temp;
    }
  }

  const duplicates: number[] = []; // Array to hold the duplicates found
  for (let i = 0; i < size; ++i) {
    // If the element is not at its correct position, it is a duplicate
    if (nums[i] !== i + 1) {
      duplicates.push(nums[i]); // Record the duplicate
    }
  }

  // Return the array containing all the duplicates
  return duplicates;
}

```

Time and Space Complexity

The given code follows the approach of cyclic sort where every element is placed in its correct position, i.e., the element `1` goes to index `0`, element `2` goes to index `1`, and so on.

Time Complexity

The time complexity of this function can be analyzed as follows:

- We iterate through each of the `n` elements of the array once, giving us an initial time complexity of `O(n)`.
- Inside the while loop, we perform the operation of placing each element in its correct location.
- Even though there is a nested loop (the while loop), the runtime still remains `O(n)`. The reason is that each number is swapped at most once because once a number is in its correct index, it's no longer part of the while loop operation.
- Therefore, every element is moved at most once, and the inner loop can run a maximum of `n` times for all elements in total, not for every individual element.

Given the above, the overall time complexity of the function is `O(n)`.

Space Complexity

The space complexity is considered as follows:

- Since we only use constant extra space (a few variables for the indices), the space complexity is `O(1)`.
- The output array does not count towards space complexity for the purpose of this analysis, as it is part of the function's output.

In conclusion, the time complexity is `O(n)` and the space complexity is `O(1)`.