448. Find All Numbers Disappeared in an Array

Easy Array Hash Table

Problem Description

The given problem describes a scenario where we have an array `nums` containing `n` integers. Each integer is within the range from `1` to `n`. The goal is to find all the numbers that are missing from the array. A number is missing if it is within the given range but does not appear in `nums`. The task is to return an array of all such missing integers.

Intuition

To efficiently find the missing numbers without using extra space, we can harness the fact that the integers in `nums` are in the range `[1, n]` and use the index as a way to flag whether a number is present. By traversing the `nums` array, for each value `x` we encounter, we can mark that `x` is present by flipping the sign of the element at index `x-1`. Note that we use `abs(x) - 1` because the values in `nums` are 1-based, and Python uses 0-based indexing. Moreover, to ensure we don't flip the sign back if we encounter the same number twice, we only flip the sign if it's positive.

Once we've completed marking the presence of numbers, we can make a second pass through the `nums` array. This time, for each index `i`, if `nums[i]` is positive, it indicates that the number `i+1` (since `i` is 0-based) was never marked and thus is missing from `nums`. We collect all these missing numbers and return them in the final array.

Solution Approach

The solution utilizes a clever trick that takes advantage of the fact that the array `nums` is mutable and that all elements are initially positive. This trick uses the sign of each element to indicate whether a number corresponding to an index has been encountered. Here is a step-by-step explanation:

1. **Loop Through `nums`**: We iterate over each element `x` in the array `nums`.
2. **Calculate Index**: For each number `x`, we compute the index `i = abs(x) - 1`. The `abs` function ensures that we are always working with a positive index, which is important since we might have previously negated some elements in `nums`.
3. **Negate Elements**: We negate the number at index `i` in `nums` if it's positive (`if nums[i] > 0`). Negating an element marks that the number `i + 1` is present in the array. If the element at index `i` is already negative, we do nothing since this implies the presence of `i + 1` has already been recorded.
4. **Identify Missing Numbers**: After negating elements in step 3, numbers corresponding to indices with positive values in `nums` are the ones missing from the array. Thus, we create a list of such numbers by checking for positive values in `nums` and adjusting the index `i` to the actual missing number `i + 1`.

The algorithm employs no additional data structures for tracking purposes, so the space complexity remains constant, i.e., `O(1)` (not counting the output array). The time complexity is `O(n)` because the array is traversed twice: once for marking presence and once for identifying missing numbers. The in-place marking is a pattern that exploits the known range of input values to use the array itself as a direct-access data structure for tracking the presence of numbers.

In summary, the algorithm leverages in-place array manipulation to solve the problem efficiently.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose our `nums` array is `[4, 3, 2, 7, 8, 2, 3, 1]`, which has `n = 8` numbers, and we are looking to find the missing numbers in the range from `1` to `8`.

1. **First Loop**:
   - We begin by traversing the array `nums`. The first number is `4`. We calculate the index for `4`, which is `abs(4)-1 = 3`. The number at index `3` is `7`, and since it's positive, we negate it to mark that `4` is present.
   - Continuing, we encounter `3`, so `abs(3)-1 = 2`. We negate the number at index `2`, which is now `-2` (it was `2` before negating `7`, but the value doesn't matter as we're using the absolute value).
   - We repeat this process for each number, and our array looks like this after the first loop: `[-4, -3, -2, -7, 8, 2, -3, -1]`. Notably, the elements at indices `4` and `5` remain positive, signaling that the corresponding numbers `5` and `6` are missing.
2. **Second Loop**:
   - In this pass, we look for indices with positive numbers. At index `4`, the value is `8`, which is positive, so we know that `5` is missing (`4 + 1`).
   - Similarly, at index `5`, the value is `2`, which is positive, so `6` is missing (`5 + 1`).

The final output array containing all missing numbers is `[5, 6]`, as these are the numbers not marked in the `nums` array. This example demonstrates how, by negating numbers and using the original array as a marker, we can efficiently identify missing elements with only two traversals of `nums`, achieving `O(n)` time complexity with constant `O(1)` space complexity, excluding the output array.

Solution Implementation

Python

```python
class Solution:
    def findDisappearedNumbers(self, nums: List[int]) -> List[int]:
        # Iterate through each number in the input list
        for number in nums:
            # Use the absolute value to find the index (since we might have negative values due to marking)
            index = abs(number) - 1

            # Mark the number at this index as visited by making it negative
            # Only mark it if it is positive (to handle duplicates, ensure only marked once)
            if nums[index] > 0:
                nums[index] *= -1

        # After marking all numbers, the positive numbers' indices are the missing ones
        # Iterate through the list, and for each positive number, the index + 1 is a missing number
        missing_numbers = [i + 1 for i, x in enumerate(nums) if x > 0]

        return missing_numbers
```

Java

```java
class Solution {

    /**
     * Finds all the elements of [1, n] inclusive that do not appear in the array.
     *
     * @param nums Array of integers ranging from 1 to n, possibly containing duplicates.
     * @return A list of integers that are missing from the array.
     */
    public List<Integer> findDisappearedNumbers(int[] nums) {
        int n = nums.length;

        // Iterate over each number in the array.
        for (int num : nums) {
            // Use absolute value in case nums[i] has been marked negative already.
            int index = Math.abs(num) - 1;

            // Mark the number at index i as negative if it's not already.
            if (nums[index] > 0) {
                nums[index] = -nums[index];
            }
        }

        // Create a list to hold the result of missing numbers.
        List<Integer> missingNumbers = new ArrayList<>();

        // Check for numbers that were not marked negative.
        for (int i = 0; i < n; i++) {
            // If the number is positive, the number (i + 1) is missing.
            if (nums[i] > 0) {
                missingNumbers.add(i + 1);
            }
        }

        return missingNumbers;
    }
}
```

C++

```cpp
class Solution {
public:
    vector<int> findDisappearedNumbers(vector<int>& nums) {
        int size = nums.size(); // Get the size of the input vector

        // Iterate over all elements in the input vector
        for (int& num : nums) {
            int index = abs(num) - 1; // Calculate the index from the value

            // If the value at the calculated index is positive, negate it
            // Negation marks that the number at 'index + 1' exists in the array
            if (nums[index] > 0) {
                nums[index] = -nums[index];
            }
        }

        vector<int> result; // Initialize an empty vector to store missing numbers

        // Iterate over the numbers from 1 to n and find which indices have positive values
        for (int i = 0; i < size; ++i) {
            // If the value at index 'i' is positive, it means 'i + 1' is missing
            if (nums[i] > 0) {
                result.push_back(i + 1); // Add the missing number to the result vector
            }
        }

        return result; // Return the vector of missing numbers
    }
};
```

TypeScript

```typescript
function findDisappearedNumbers(nums: number[]): number[] {
    const size = nums.length; // The size of the input array 'nums'.
    // Mark each number that appears in the array by negating the value at the index
    // corresponding to that number.
    for (const num of nums) {
        const index = Math.abs(num) - 1; // Calculate the correct index.
        // Negate the number at the index if it is positive.
        if (nums[index] > 0) {
            nums[index] = -nums[index];
        }
    }

    // Initialize an array to hold the numbers that did not appear in 'nums'.
    const missindNumbers: number[] = [];
    // Traverse the array to find the indices that contain a positive number, which
    // indicates that the number corresponding to that index did not appear in 'nums'.
    for (let i = 0; i < size; i++) {
        if (nums[i] > 0) {
            missingNumbers.push(i + 1); // Add the missing number to the result array.
        }
    }

    // Return the array containing all the missing numbers.
    return missingNumbers;
}
```

```python
class Solution:
    def findDisappearedNumbers(self, nums: List[int]) -> List[int]:
        # Iterate through each number in the input list
        for number in nums:
            # Use the absolute value to find the index (since we might have negative values due to marking)
            index = abs(number) - 1

            # Mark the number at this index as visited by making it negative
            # Only mark it if it is positive (to handle duplicates, ensure only marked once)
            if nums[index] > 0:
                nums[index] *= -1

        # After marking all numbers, the positive numbers' indices are the missing ones
        # Iterate through the list, and for each positive number, the index + 1 is a missing number
        missing_numbers = [i + 1 for i, x in enumerate(nums) if x > 0]

        return missing_numbers
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`. This is because the code consists of a single loop that goes through the `nums` array with `n` elements exactly once, marking each number that has been seen by negating the value at the index corresponding to that number. The loop to create the output list also runs for `n` elements, so the entire operation is linear.

The space complexity of the code is `O(1)`, as it only uses constant extra space. The input list is modified in place to keep track of which numbers have been seen. The list of disappeared numbers is the only additional space used, and it's the output of the function, which typically isn't counted towards space complexity.