283. Move Zeroes

Easy | Array | Two Pointers

Leetcode Link

Problem Description

The problem presents us with a challenge to modify an array `nums` such that all of the zeros are moved to the end, while the relative order of the non-zero elements is preserved. The important constraints for this task are that the changes must be made to the original array `nums` directly, without using an additional array or making a copy of the original array. This requirement ensures an in-place solution with no extra space overhead, which tests one's ability to manipulate array elements efficiently.

Intuition

The intuition behind the solution is based on the two-pointer technique. The goal is to iterate through the array and process elements one by one, swapping non-zero elements towards the front while effectively moving zero elements towards the end.

We initialize a pointer `i` at the position just before the start of the array (i.e., `i = -1`). This pointer will keep track of the position to place the next non-zero element.

As we iterate over the elements of the array with another pointer `j`, we look for non-zero elements. When we encounter a non-zero element (`x`), we increment the pointer `i`, indicating that we have found another non-zero element. Then we swap the elements at indices `i` and `j`. This action has two consequences:

1. It moves the non-zero element `x` to the front of the array at the next available position.
2. It moves the zero (if `i` and `j` are not the same) to the position `j`.

The process ensures that all non-zero elements are shifted to the front, maintaining their relative order because we swap non-zero elements with zeroes (or with themselves) without affecting previously positioned non-zero elements. Since `i` increases only when we find a non-zero, it always points to the first zero in the sequen`ce of elements processed so far, ensuring the relative order is maintained.

When the loop is complete, the array is effectively partitioned into two segments: the non-zero elements from the start to index `i`, followed by zeros from index `i + 1` to the end.

Solution Approach

The solution uses a straightforward algorithm that leverages the two-pointer technique, which is a common pattern used to modify arrays in-place.

Here's a step-by-step walk through the implementation of the solution:

- Start by initializing a pointer called `i` to `-1`. This pointer will serve as a marker for the position of the last non-zero element found.
- Begin iterating through the array with a `for` loop, using another pointer `j` to keep track of the current index in the array.
- For each element `x` in the array, check if it is a non-zero.
- If a non-zero element is encountered, increment `i`. This step means you have found a non-zero element to place at an earlier position in the array (specifically, at index `i + 1`).
- Swap the elements at the positions pointed by `i` and `j`. The swapping is done using tuple unpacking in Python: `nums[i], nums[j] = nums[j], nums[i]`.
  - If `i` is different from `j`, this has the effect of moving a non-zero element to the front (at the position `i`) and a zero element toward the back (at the position `j`).
  - If `i` is the same as `j`, this action does nothing as we're swapping an element with itself, but it's an essential part of the loop as it maintains the relative order of the non-zero elements.
- This process is repeated for every element in the array. Since the positions of non-zero elements are incremented and swapped in sequence, their relative order is preserved.
- The zeros are automatically moved to the end of the array as non-zero elements are pulled forward.
- This continues until the entire array is traversed, resulting in all non-zero elements being placed at the beginning of the array while all zeros are moved to the end.

The simplicity of this approach is highlighted by the fact that it does not use any additional data structures and fulfills the in-place requirement of the problem. It is also time-efficient, running in `O(n)` time complexity since it only requires a single pass through the array, and space-efficient with `O(1)` space complexity as it only uses additional constant space.

Example Walkthrough

Let's go through a small example to illustrate the solution approach using the given problem description:

Assume our input array `nums` is `[0, 1, 0, 3, 12]`. We need to move all zeros to the end while maintaining the order of the non-zero elements.

1. Start with pointer `i` set to `-1`, which will track the placement of non-zero elements.
2. Begin a `for` loop with pointer `j` iterating through the indices of `nums`.
   - On the first iteration, `j` is `0` and `nums[j]` is `0`. Since this is a zero, we do not move the pointer `i`. The array remains unchanged.
   - On the second iteration, `j` is `1` and `nums[j]` is `1`. This is non-zero, so we increment `i` to `0`, and swap `nums[i]` with `nums[j]`, but since `i` is now the same as `j`, the array still remains `[0, 1, 0, 3, 12]`.
   - On the third iteration, `j` is `2` and `nums[j]` is `0`. The pointer `i` stays the same since this is a zero. No changes are made to the array.
   - On the fourth iteration, `j` is `3` and `nums[j]` is `3`. This is non-zero, so we increment `i` to `1`, and swap `nums[i]` with `nums[j]`, resulting in the array `[0, 3, 0, 1, 12]`.
   - On the fifth iteration, `j` is `4` and `nums[j]` is `12`. This is non-zero, so we increment `i` to `2`, and swap `nums[i]` with `nums[j]`, which leads to `[0, 3, 12, 1, 0]`.
3. Continuing this process until the end of the array will not result in any more changes since we've reached the last element.
4. After the loop concludes, we have successfully moved all non-zero elements to the front of the array in their original order, with the zeros shifted to the end: `[1, 3, 12, 0, 0]`. This is our final modified `nums` array.

In this example, we can see how the two-pointer technique is applied to solve the problem by using one pointer `i` to track the position of the non-zero element to place next, and another pointer `j` to iterate through the elements, performing swaps as needed without the use of any additional space, thus preserving the original order of non-zero elements and accomplishing the task efficiently.

Python Solution

```python
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        This function takes a list of numbers and moves all the zeros to the end,
        maintaining the relative order of the other elements.
        """
        # last_non_zero_found_at keeps track of the last non-zero index found in the array
        last_non_zero_found_at = 0

        # Iterate over the array
        for current, value in enumerate(nums):
            # When a non-zero element is found
            if value != 0:
                # Swap the current non-zero element with the element at last_non_zero_found_at index
                nums[last_non_zero_found_at], nums[current] = nums[current], nums[last_non_zero_found_at]
                # Move the last_non_zero_found_at index forward
                last_non_zero_found_at += 1

        # Note: This function does not return anything as it is supposed to modify the nums list in-place.
```

Java Solution

```java
class Solution {

    /**
     * Method to move all zeros in the array to the end while maintaining the relative order of
     * the non-zero elements.
     */
    public void moveZeroes(int[] nums) {
        // Initialize a pointer to keep track of the position of the last non-zero element found.
        int lastNonZeroFoundAt = -1;
        // Variable to store the array's length to avoid recalculating it.
        int arrayLength = nums.length;

        // Iterate over the array.
        for (int currentIndex = 0; currentIndex < arrayLength; ++currentIndex) {
            // If the current element is not zero,
            if (nums[currentIndex] != 0) {
                // Increment the lastNonZeroFoundAt.
                lastNonZeroFoundAt++;

                // Swap the current element with the element at the lastNonZeroFoundAt position.
                int temp = nums[lastNonZeroFoundAt];
                nums[lastNonZeroFoundAt] = nums[currentIndex];
                nums[currentIndex] = temp;
            }
        }
    }
}
```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to move all zeros in the array to the end while maintaining the relative order of non-zero elements.
    void moveZeroes(vector<int>& nums) {
        // Initialize a pointer `lastNonZeroFoundAt` to keep track of the position of the last non-zero element found.
        int lastNonZeroFoundAt = -1;
        // Get the size of the input vector `nums`.
        int n = nums.size();

        // Iterate over the vector.
        for (int currentIndex = 0; currentIndex < n; ++currentIndex) {
            // Check if the current element is non-zero.
            if (nums[currentIndex] != 0) {
                // Increment `lastNonZeroFoundAt` and swap the current element with the element at the `lastNonZeroFoundAt` index.
                // This moves all non-zero elements to the front of the array in their original order.
                swap(nums[++lastNonZeroFoundAt], nums[currentIndex]);
            }
            // If the current element is zero, nothing needs to be done; continue to the next iteration.
        }
        // After the loop, all non-zero elements are at the beginning of the array, and all zeros are moved to the end.
    }
};
```

Typescript Solution

```typescript
/**
 * This function moves all the zero elements in an array to the end of it
 * while maintaining the relative order of the non-zero elements.
 * @param nums - The array of numbers to be rearranged in-place
 */
function moveZeroes(nums: number[]): void {
    // Get the length of the input array.
    const length = nums.length;
    // Initialize the position for the non-zero element to be placed.
    let insertPosition = 0;

    // Iterate over each element in the array.
    for (let currentIndex = 0; currentIndex < length; currentIndex++) {
        // Check if the current element is non-zero.
        if (nums[currentIndex] !== 0) {
            // If the current index is greater than the insert position,
            // swap the elements and place zero at the current index.
            if (currentIndex > insertPosition) {
                nums[insertPosition] = nums[currentIndex];
                nums[currentIndex] = 0;
            }
            // Move to the next insert position for the next non-zero element.
            insertPosition++;
        }
    }
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the `nums` array. This is because the code uses a single loop that traverses the elements in the array once. Each iteration involves a constant time operation of checking a condition and possibly swapping elements, which does not depend on the size of the array.

Space Complexity

The space complexity of the provided code is `O(1)`. This is due to the fact that no additional space that scales with the input size is used. The only extra variables used are `i` and `j`, which are pointers that help in iterating and swapping elements within the array in-place.