2460. Apply Operations to an Array

Easy Array Simulation Leetcode Link

Problem Description

In this problem, you are given an array of non-negative integers. Your task is to perform a series of operations on this array, specifically `n - 1` operations where `n` is the size of the array. Each operation is performed on the `i`th element of the array (considering 0-indexing), following these rules:

- Check if the `i`th element of the array (`nums[i]`) is equal to the next element (`nums[i + 1]`).
- If they are equal, double the `i`th element (`nums[i] = nums[i] * 2`) and set the `i + 1`th element to `0`.
- If they are not equal, move on to the next operation without making changes to the current `i`th element.

After you have performed all the operations, you need to shift all the `0`'s in the array to the end, while preserving the order of the non-zero elements. The challenge is to perform these operations sequentially and then return the modified array.

Example: Given an array `[1,0,2,0,0,1]`, after performing all the operations and shifting the `0`'s, the resulting array would be `[1,2,1,0,0,0]`.

Intuition

The solution involves a two-step approach. First, we perform the `n - 1` operations as specified, inspecting each pair of adjacent elements and applying the doubling and zeroing rules. After all operations are complete, we're left with an array with some elements set to `0` that now should be moved to the end.

The intuition behind the first step is straightforward: loop through the array, compare each element with its neighbor, and if they are the same, apply the operation. We have to remember that these operations should be applied sequentially, meaning the result of one operation may affect subsequent operations. Therefore, careful in-place manipulation of the array is necessary.

For the second step, the intuition is to keep track of where the next non-zero element should be placed. Essentially, this involves a second pass through the array, where we move each non-zero element leftwards to "fill in" the non-zero portion of the array. This is why a separate counter (`i` in the provided solution) is maintained to keep track of the index at which the next non-zero element should be inserted. Non-zero elements are placed in the array in their original order until all non-zero elements have been accounted for. Finally, the rest of the array is filled with `0`s.

This two-pass approach ensures that the operations are applied correctly and that the output array maintains the proper order of non-zero elements, concluding with the zeros shifted to the end.

Solution Approach

The provided solution follows a straightforward two-pass approach which efficiently addresses the requirements with simple array manipulation techniques. The key steps in this approach are as follows:

1. *Doubling and Zeroing in Place*: The first pass goes through the array from the start to the second-to-last element. At each index `i`, the algorithm checks if `nums[i]` equals `nums[i + 1]`. If they are equal, it doubles `nums[i]` using the left shift operator (`<<= 1` is equivalent to multiplying by 2) and sets `nums[i + 1]` to `0`. Using the bitwise shift here is a more efficient way of doubling integers.

2. *Shifting Non-zero Elements*: In the second pass, the algorithm traverses the array only once more and maintains a separate index `i` which keeps track of the position where the next non-zero element should go. Hence, for each element `x` in the array, if `x` is non-zero, it is placed at `nums[i]` and the index `i` is incremented. This effectively compacts all non-zero elements towards the beginning of the array.

3. *Filling Remaining with Zeros*: Because the original array is modified in place during step 1, and the non-zero elements are moved forward in step 2, the remaining elements in the array (from the current index `i` to the end) are already implicitly `0`. If there was any need to explicitly set them to `0`, it could be done in a final pass; however, the code efficiency is improved by realizing this step is not necessary given the initial array manipulation.

This approach takes `O(n)` time due to the two sequential passes through the array, where `n` is the number of elements in `nums`. No additional data structures are needed, so the space complexity is `O(1)` as the solution uses only a fixed amount of extra space to store the counters and temporary values.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following array:

```
Input array: [2, 2, 3, 3, 3]
```

Here is how the solution approach would be applied to this array:

Doubling and Zeroing in Place:

We start at the first element and compare it with the next element.

1. `nums[0]` is 2, and `nums[1]` is also 2. They are equal, so we double `nums[0]` (2 becomes 4) and set `nums[1]` to 0. Now the array looks like this: `[4, 0, 3, 3, 3]`.
2. We move to the next non-zero pair. `nums[2]` is 3, and `nums[3]` is also 3. Doubling `nums[2]` we get 6, and set `nums[3]` to 0: `[4, 0, 6, 0, 3]`.
3. `nums[4]` is the last element and doesn't have a pair to compare with, so the array remains `[4, 0, 6, 0, 3]` after the first pass.

Shifting Non-zero Elements:

Now, we make a second pass through the array and move all non-zero elements to the front.

1. We set up a separate index `i` starting at 0 to track where to place non-zero elements.
2. Starting from the beginning of the array, when we find a non-zero element, we move it to the `nums[i]` position and increment `i`.
3. For the first non-zero element, `4` stays in its original position, and `i` is incremented to 1.
4. Skipping over the zero, we come to `6`. `6` is placed at `nums[i]` (which is `nums[1]` now), and `i` becomes 2.
5. Next, the `3` is placed at `nums[i]` (now `nums[2]`), and `i` is incremented to 3.

So, the non-zero part of the array is now `[4, 6, 3]`. Since the array's length is 5 and we already have the remaining elements implicitly set to `0` due to the first pass, we get:

```
Output array: [4, 6, 3, 0, 0]
```

We have now successfully completed all operations as defined by the problem statement, and the zeros are shifted to the end of the array while preserving the order of the non-zero elements.

Python Solution

```python
from typing import List

class Solution:
    def apply_operations(self, nums: List[int]) -> List[int]:
        # Get the length of the list 'nums'.
        length = len(nums)

        # Iterate over the list elements, except for the last element.
        for i in range(length - 1):
            # If the current element is the same as the next element,
            # double its value and set the next element to 0.
            if nums[i] == nums[i + 1]:
                nums[i] *= 2
                nums[i + 1] = 0

        # Create a new list 'result' with the same size filled with zeros.
        result = [0] * length

        # Initialize a pointer for the index of 'result'.
        result_index = 0

        # Iterate over 'nums' to populate non-zero elements in the 'result'.
        for num in nums:
            # If the element is non-zero, put it in the next position of 'result'.
            if num:
                result[result_index] = num
                result_index += 1

        # Return the 'result' list containing the processed numbers.
        return result

```

Java Solution

```java
class Solution {

    // Method to apply operations on an array of integers
    public int[] applyOperations(int[] nums) {
        int length = nums.length; // Get the length of the array

        // Loop through each element, except the last one
        for (int i = 0; i < length - 1; ++i) {
            // Check if the current element is equal to the next element
            if (nums[i] == nums[i + 1]) {
                // If so, double the current element
                nums[i] <<= 1; // Same as nums[i] = nums[i] * 2
                // And set the next element to zero
                nums[i + 1] = 0;
            }
        }

        int[] result = new int[length]; // Create a new array to store the results
        int index = 0; // Initialize result array index

        // Iterate through the original array
        for (int num : nums) {
            // Copy non-zero elements to the result array
            if (num > 0) {
                result[index++] = num; // Assign and then increment the index
            }
        }

        return result; // Return the resulting array
    }
}

```

C++ Solution

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

        // Loop through each pair of adjacent numbers
        for (int idx = 0; idx < size - 1; ++idx) {
            // If adjacent numbers are equal, double the current number and set next number to zero
            if (nums[idx] == nums[idx + 1]) {
                nums[idx] <<= 1; // double the number (same as nums[idx] *= 2)
                nums[idx + 1] = 0; // set the next number to zero
            }
        }
        // Create a new vector to store the resulting numbers after applying operations
        vector<int> result(size);
        int resultIndex = 0; // Initiate a result index to populate result vector with non-zero values

        // Iterate over the modified nums array to filter out the zeros
        for (int& num : nums) {
            // If the current number is non-zero, add it to the result vector
            if (num) {
                result[resultIndex++] = num; // Add to result and increment the position
            }
        }

        // Return the result vector (which doesn't contain zeros between non-zero numbers)
        return result;
    }
};

```

Typescript Solution

```typescript
function applyOperations(nums: number[]): number[] {
    const length = nums.length; // The total number of elements in the array 'nums'

    // Double the current number and set the next one to 0 if they're equal
    for (let index = 0; index < length - 1; ++index) {
        if (nums[index] === nums[index + 1]) {
            nums[index] *= 2; // Double the current number
            nums[index + 1] = 0; // Set the next number to 0
        }
    }

    // Initialize a new array 'result' with the same length as 'nums' and fill it with 0s
    const result: number[] = Array(length).fill(0);

    // Pointer to the position in 'result' where the next non-zero element will be placed
    let resultIndex = 0;

    // Move all non-zero elements to the 'result' array
    for (const number of nums) {
        if (number !== 0) {
            result[resultIndex++] = number; // Assign non-zero element and move to the next index
        }
    }

    return result; // Return the transformed array
}

```

Time and Space Complexity

Time Complexity

The given function `applyOperations` consists of two separate for-loops that are not nested.

The first for-loop iterates through the list `nums`, except for the last element, performing constant-time operations. The iteration occurs exactly `n - 1` times, where `n` is the length of `nums`. Since no other operations are nested inside this loop, the time complexity for this portion is `O(n-1)` which simplifies to `O(n)`.

The second for-loop iterates through each element in `nums` once. It fills in the non-zero elements to the list `ans`. The assignment and increment operations are constant time operations, and since this loop iterates `n` times, the time complexity for this loop is also `O(n)`.

Combining both loops, which are sequential and not nested, the overall time complexity of the function is `O(n) + O(n)` which simplifies to `O(n)`.

Space Complexity

Regarding space complexity, a new list `ans` of the same size as the input list `nums` is created, which denotes the extra space used by the algorithm. This implies a space complexity of `O(n)`.

No additional data structures are used that grow with the input size, hence the total space complexity of the function is `O(n)`.