2348. Number of Zero-Filled Subarrays

Medium Array Math

Leetcode Link

Problem Description

The problem requires us to find the total number of subarrays in a given integer array where every element is `0`. A *subarray* is defined as a contiguous sequence of elements within the array. This means that we are looking for all the continuous sequences of zeroes in the array and want to count them. It is important to realize that this includes subarrays of different lengths, starting and ending at any indices where zeros are contiguous.

Intuition

The intuition behind the solution approach is fairly straightforward. We want to iterate through the array and keep track of contiguous sequences of zeros. To achieve this, we can maintain a count (`cnt`) that keeps track of the length of the current sequence of zeros we are in.

Starting from the beginning of the array, for each element, we check whether it is a zero:

- If it is, we increment our current zero sequence length count (`cnt`) by `1`, since we've found an additional zero in our current subarray of zeros.
- If it isn't, we reset our count to `0` as we're no longer in a sequence of zeros.

Now, the key insight is that each new zero we encounter not only forms a subarray of length `1` by itself but also extends the length of all subarrays ending at the previous zero by `1`. Therefore, at each step, we can add the current count of zeros to our answer (`ans`).

For each new zero found, we do not need to recount all the subarrays it could create with all preceding zeros. We just realize that it extends them all by one. Thus, the current running `cnt` can be added directly to `ans` to account for all new subarrays ending at this zero.

This way, we effectively count all zero-filled subarrays without having to explicitly enumerate them all, leading to an efficient solution.

Solution Approach

The implemented solution relies on a simple linear scan of the input array, which implicitly tracks the size of contiguous zero-filled subarrays. Two main variables are introduced: `ans` and `cnt`. No additional data structures are needed, highlighting the simplicity and efficiency of the approach.

Let's delve deeper into the steps:

1. Initialize `ans` and `cnt` to `0`. `ans` will keep the running total of all zero-filled subarray counts, while `cnt` will track the current number of zeros in a contiguous sequence.

2. Iterate through each value `v` in the `nums` array:

   a. If `v` is not `0`, reset `cnt` to `0`. This step is important as it signifies that the current sequence of zeros has ended and any new sequence found later will be independent of any zeros previously encountered.

   b. If `v` is `0`, increment `cnt` by `1`, since we've found an additional zero that extends the current zero-filled subarray.

3. After each iteration, add the value of `cnt` to `ans`. This is because every new zero found (or every zero in a contiguous sequence of zeros) not only represents a subarray of one element (itself) but also extends the length of any zero-filled subarray ending at the previous zero by one. Therefore, `cnt` also represents the number of new zero-filled subarrays that the current `0` contributes to.

4. Continue this process until the end of the array.

5. Once the iteration is complete, `ans` will contain the total number of zero-filled subarrays in the array.

The time complexity of this approach is O(n), where n is the number of elements in the array, since it requires a single pass through the array. The space complexity is O(1), as the solution uses a constant amount of extra space.

Example Walkthrough

To illustrate the solution approach, let's use a small example with the array `nums = [1, 0, 0, 2, 0]`.

1. Initialize `ans` and `cnt` to `0`. There are no zero-filled subarrays counted yet, and no current sequence of zeros.

2. Start iterating through the `nums` array:

   - At the first element `1`, `cnt` remains `0` as it is not a zero.
   - At the second element `0`, increment `cnt` to `1`. This zero forms a new subarray `[0]`. Add `cnt` to `ans`, so `ans` becomes `1`.
   - At the third element `0`, increment `cnt` to `2`. This zero extends the previous subarray to form a new one `[0, 0]` and also makes an individual subarray `[0]`. Add `cnt` to `ans`, so `ans` becomes `1 + 2 = 3`.
   - At the fourth element `2`, reset `cnt` to `0` as the sequence of zeros has been broken.
   - At the fifth element `0`, increment `cnt` to `1`. This zero forms a new subarray `[0]`. Add `cnt` to `ans`, so `ans` becomes `3 + 1 = 4`.

3. By the end of the array, we've counted all zero-filled subarrays: `[0]`, `[0, 0]`, and two separate `[0]` subarrays after the `2`. Therefore, the total number of zero-filled subarrays in this example is `4`.

Thus, `ans` holds the correct total, and the process demonstrates both the resetting of `cnt` when non-zero elements are encountered and the accumulation of the number of subarrays in `ans` as zero elements are processed. The approach ensures we count consecutive sequences of zeros efficiently without unnecessary enumeration.

Python Solution

```python
from typing import List

class Solution:
    def zero_filled_subarray(self, nums: List[int]) -> int:
        # Initialize the counter for current zero sequence and the answer accumulator
        current_zero_count = total_subarrays = 0

        # Iterate over each number in the input list
        for num in nums:
            if num == 0:
                # If the current number is zero, increment the current zero sequence counter
                current_zero_count += 1
            else:
                # If the current number is not zero, reset the counter
                current_zero_count = 0

            # Add the current zero sequence length to the total number of subarrays
            # This works because for each new zero, there are `current_zero_count` new subarrays ending with that zero
            total_subarrays += current_zero_count

        # Return the total count of zero-filled subarrays
        return total_subarrays
```

Java Solution

```java
class Solution {
    // Method to calculate the number of zero-filled contiguous subarrays
    public long zeroFilledSubarray(int[] nums) {
        long totalCount = 0;  // Initialize total count of zero-filled subarrays
        int zeroCount = 0;    // Initialize count of consecutive zeros

        // Iterate through the array elements
        for (int value : nums) {
            // Reset zeroCount to 0 if the current element is not zero
            // Otherwise, increment zeroCount
            zeroCount = (value != 0) ? 0 : zeroCount + 1;
            // Add the current zeroCount to the total count
            totalCount += zeroCount;
        }

        return totalCount;  // Return the total count of zero-filled subarrays
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the number of zero-filled subarrays.
    long long zeroFilledSubarray(vector<int>& nums) {
        long long answer = 0; // Variable to store the final answer.
        int zeroCount = 0; // Counter to keep track of consecutive zeros.

        // Iterate through the numbers in the vector.
        for (int value : nums) {
            // Reset the counter if the current value is not zero, otherwise increase it.
            zeroCount = (value != 0) ? 0 : zeroCount + 1;

            // Add the number of new zero-filled subarrays ending with the current value.
            answer += zeroCount;
        }

        // Return the total number of zero-filled subarrays found.
        return answer;
    }
};
```

Typescript Solution

```typescript
function zeroFilledSubarray(nums: number[]): number {
    let totalSubarrays = 0; // This variable will hold the total count of zero-filled subarrays
    let currentZerosCount = 0; // This variable tracks the current consecutive zeros count

    // Loop through each number in the nums array
    for (const value of nums) {
        if (value === 0) {
            // If the current value is zero, increment the current consecutive zeros count
            currentZerosCount++;
        } else {
            // If it's not zero, reset the count to 0
            currentZerosCount = 0;
        }
        // Add the currentZerosCount to the total subarrays count which also includes
        // all subarrays ending with the current element
        totalSubarrays += currentZerosCount;
    }

    // Return the total number of zero-filled subarrays found in the nums array
    return totalSubarrays;
}
```

Time and Space Complexity

The provided code snippet has a time complexity of `O(n)`, where `n` is the length of the input list `nums`. This is because the code iterates through the list once, performing a constant number of operations for each element.

The space complexity of this code is `O(1)`. No additional space proportional to the input size is used; the variables `ans` and `cnt` use a fixed amount of space regardless of the input size.