2750. Ways to Split Array Into Good Subarrays

Medium Array Math

Problem Description

You are given an array of binary numbers, `nums`. A "good" subarray is defined as a contiguous part of `nums` that contains exactly one `1`. Your task is to calculate the number of ways you can split the given array into "good" subarrays. Because the resulting number could be very large, you have to return this number modulo `10**9 + 7`. It's important to remember that a subarray is considered non-empty, meaning it must have at least one element.

Intuition

The solution's intuitive approach relies on counting contiguous segments of zeros between ones. For each segment of zeros between ones, you can create multiple subarrays that include the single one. The number of ways to split on the left side of a '1' depends on the number of zeros before it. More precisely, you multiply `ans` with the number of elements between the current '1' and the previous one, because each of these elements represents a point where the subarray could start.

Here's how we approach the solution:

- Initialize a variable to keep track of the previous index where `1` was found. Let's call this `j`, and we start with `j = -1`, which indicates there's no `1` found yet.
- Iterate through the given array, and each time we find a `1`, we calculate the number of ways using the difference between the current index `i` and `j`.
- If `j` is set (not `-1`), it means we have found a `1` previously, and we should multiply the current count (`ans`) with the number of elements between the current and the previous `1` (which is `i - j`).
- Take modulo `10**9 + 7` with the result of the multiplication to ensure the number does not exceed the limit after each operation.
- If we never find a `1` (`j` stays `-1`), return `0` since it's impossible to form any "good" subarrays.
- Otherwise, if at least one `1` is found, return the final count `ans`.

Solution Approach

The code implements a simple yet effective approach to solve the problem using a single pass through the input array and arithmetic operations. The core idea is based on combinatorics, considering the number of zeros between ones.

Here's a step-by-step breakdown of the algorithm:

1. Variables:
   - `mod`: A constant representing the value `10**9 + 7`, used for taking the modulo to keep the count within integer limits.
   - `ans`: It acts as a running total for the number of ways we can split the array into "good" subarrays. Initialized to `1`.
   - `j`: An index marker to remember the position of the last `1` encountered in the array. Initialized to `-1` to indicate that no `1` has been seen at the start.
2. Loop:
   - We iterate over the array with an index `i` and value `x` using `enumerate(nums)`.
   - Within the loop, if we encounter a zero (`x == 0`), we simply `continue` to the next iteration as zeros between ones do not change the count directly but affect the number of ways to split before the next one.
3. When encountering a `1` (`x == 1`):
   - Check if `j` is greater than `-1`, which would mean this is not the first `1`. If so, compute the product of the current `ans` and the difference `i - j`, representing the number of zeroes between the previous `1` and the current. This product represents the number of new "good" subarrays we can now form.
   - We then take the result modulo `mod` to ensure the number stays within the required limit.
4. Update `j`:
   - Regardless of whether it's the first `1` or a subsequent one, we update `j` to the current index `i` right after the above computation.
5. Final check and return:
   - After the loop, we check if `j` is `-1`. If it is, it implies that the array contained no `1` to create "good" subarrays, so we return `0`.
   - If `j` is not `-1`, then we return the value of `ans` as the final result as it contains the total count modulo `10**9 + 7`.

This solution leverages the pattern that each one in the array "resets" the possibility counter, and the zeros preceding one contribute to the total count of subarray splits by providing multiple starting points for the subarrays. By updating the count at each `1`, we harness the potential of previous zeros, thus avoiding the need for nested loops and reducing the overall time complexity.

Example Walkthrough

Let's illustrate the solution with a small example. Consider the input array `nums = [0, 1, 0, 0, 1]`. We want to calculate the number of ways we can split this array into "good" subarrays, which are subarrays containing exactly one `1`.

Following the solution approach:

1. Initialize `mod = 10**9 + 7`, `ans = 1`, and `j = -1`.
2. Begin looping through `nums` with index `i` and value `x`:
   - At `i = 0`, `x = 0`. Since it's a zero, we continue to the next iteration.
   - At `i = 1`, `x = 1`. This is the first `1` we've encountered, so we don't multiply `ans` by `(i - j)`. We simply update `j = 1`.
   - At `i = 2`, `x = 0`. It's a zero, so again, we continue to the next iteration.
   - At `i = 3`, `x = 0`. Since it's a zero, we proceed to the next iteration.
   - At `i = 4`, `x = 1`. We now have `j = 1` from the previous `1`. So, `ans` is multiplied by `(i - j)`, which is `(4 - 1) == 3`. This gives us `ans = ans * 3 % mod = 1 x 3 % mod = 3`, since there are three ways to split with zeros between the two `1`s. We then update `j = 4`.
3. Looping is complete, now we check `j`:
   - In our example, `j` is not `-1` (it's `4`), which means we have "good" subarrays.
4. Finally, we would return `ans`. The final answer is `3`, representing the number of ways to split the array into "good" subarrays:
   - `[1]` from the subarray `[0, 1]`
   - `[1, 0]` from the subarray `[0, 1, 0]`
   - `[1, 0, 0]` from the subarray `[0, 1, 0, 0]`

The key to understanding the approach is recognizing how the "good" subarrays can only start after the previous `1` up until the next `1`. Hence, the `(i - j)` multiplication effectively captures the valid "good" subarray starts. This example demonstrates the principle behind the algorithm in a straightforward manner.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def numberOfGoodSubarraySplits(self, nums: List[int]) -> int:
        # Define the modulus constant to avoid magic numbers,
        # as required in some competitive programming problems.
        MODULUS = 10**9 + 7

        # Initialize the answer with 1 since there's always at least one "good" subarray
        # if there's at least one non-zero number in the input.
        answer = 1
        # Initialize the previous non-zero index as -1 (indicating not found yet).
        prev_non_zero_index = -1

        # Loop through the list of numbers with their indices.
        for current_index, number in enumerate(nums):
            # Skip zero values as they do not contribute to "good" subarray splits.
            if number == 0:
                continue

            # If we've found a non-zero before, we calculate the product of the answer
            # and the distance between the current non-zero and the previous non-zero.
            # We apply the modulus to keep the numbers manageable.
            if prev_non_zero_index != -1:
                answer = (answer * (current_index - prev_non_zero_index)) % MODULUS

            # Update the index of the previous non-zero number.
            prev_non_zero_index = current_index

        # If no non-zero was found (prev_non_zero_index is still -1), return 0.
        # Otherwise, return the computed answer.
        return 0 if prev_non_zero_index == -1 else answer
```

Java

```java
class Solution {

    // Method to calculate the number of good subarray splits in a given array.
    public int numberOfGoodSubarraySplits(int[] nums) {
        // Modulus to ensure the result fits within integer limits
        final int mod = (int) 1e9 + 7;

        // Initialize the result with 1 since there's always at least one way to split (i.e., the whole array itself)
        int result = 1;

        // Initialize 'lastNonZeroIndex' to keep track of the last non-zero element seen
        int lastNonZeroIndex = -1;

        // Iterate over the array
        for (int i = 0; i < nums.length; ++i) {
            // Skip if the current element is zero
            if (nums[i] == 0) {
                continue;
            }

            // If we have encountered a non-zero element before, update the result
            if (lastNonZeroIndex > -1) {
                // The number of ways the array can be split increases by the distance between
                // the current non-zero element and the immediate previous non-zero element
                result = (int) ((long) result * (i - lastNonZeroIndex) % mod);
            }

            // Update the 'lastNonZeroIndex' with the current index
            lastNonZeroIndex = i;
        }

        // If no non-zero elements were found, return 0 (no good splits)
        // Otherwise, return the computed result
        return lastNonZeroIndex == -1 ? 0 : result;
    }
}
```

C++

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

class Solution {
public:
    // Function to compute the number of good subarray splits in the given array.
    int numberOfGoodSubarraySplits(vector<int>& nums) {
        const int MODULO = 1e9 + 7; // A constant for modulo operation to avoid overflow.

        int totalGoodSplits = 1; // Initialize the result to start from 1.
        int lastNonZeroIndex = -1; // Keep track of the last non-zero index encountered.

        // Iterate through all elements in the array.
        for (int i = 0; i < nums.size(); ++i) {
            // Skip processing if the current element is 0.
            if (nums[i] == 0) {
                continue;
            }

            // If this is not the first non-zero element,
            // multiply the total number of good splits by the distance to the previous non-zero element.
            if (lastNonZeroIndex > -1) {
                totalGoodSplits = static_cast<long long>(totalGoodSplits) * (i - lastNonZeroIndex) % MODULO;
            }

            // Update the last non-zero index to the current index.
            lastNonZeroIndex = i;
        }

        // If there were no non-zero elements found, return 0. Otherwise, return the number of good splits.
        return lastNonZeroIndex == -1 ? 0 : totalGoodSplits;
    }
};
```

TypeScript

```typescript
function numberOfGoodSubarraySplits(nums: number[]): number {
    // Initialize the answer with 1 as a default for the first valid split
    let answer = 1;
    // Initialize the previous non-zero index 'j' with -1
    let prevNonZeroIndex = -1;
    // Define the modulus value for large number handling
    const MODULUS = 10 ** 9 + 7;
    // Get the length of the input array
    const arrayLength = nums.length;

    // Iterate through every element in the given array
    for (let currentIndex = 0; currentIndex < arrayLength; ++currentIndex) {
        // Continue to next iteration if the current element is zero
        if (nums[currentIndex] === 0) {
            continue;
        }
        // If a non-zero element was already found,
        // calculate the product of the number of elements between the current and previous non-zero elements.
        if (prevNonZeroIndex > -1) {
            answer = (answer * (currentIndex - prevNonZeroIndex)) % MODULUS;
        }
        // Update the previous non-zero index to the current index
        prevNonZeroIndex = currentIndex;
    }
    // Return 0 if no non-zero elements were found; otherwise, return the answer
    return prevNonZeroIndex === -1 ? 0 : answer;
}
```

```python
from typing import List

class Solution:
    def numberOfGoodSubarraySplits(self, nums: List[int]) -> int:
        # Define the modulus constant to avoid magic numbers,
        # as required in some competitive programming problems.
        MODULUS = 10**9 + 7

        # Initialize the answer with 1 since there's always at least one "good" subarray
        # if there's at least one non-zero number in the input.
        answer = 1
        # Initialize the previous non-zero index as -1 (indicating not found yet).
        prev_non_zero_index = -1

        # Loop through the list of numbers with their indices.
        for current_index, number in enumerate(nums):
            # Skip zero values as they do not contribute to "good" subarray splits.
            if number == 0:
                continue

            # If we've found a non-zero before, we calculate the product of the answer
            # and the distance between the current non-zero and the previous non-zero.
            # We apply the modulus to keep the numbers manageable.
            if prev_non_zero_index != -1:
                answer = (answer * (current_index - prev_non_zero_index)) % MODULUS

            # Update the index of the previous non-zero number.
            prev_non_zero_index = current_index

        # If no non-zero was found (prev_non_zero_index is still -1), return 0.
        # Otherwise, return the computed answer.
        return 0 if prev_non_zero_index == -1 else answer
```

Time and Space Complexity

Time Complexity

The given Python code defines a function `numberOfGoodSubarraySplits` that iterates through the entire list `nums` just once. The operations performed within the loop are constant-time operations: basic arithmetic operations, a modulus operation, a conditional statement, and variable assignments.

- **Enumerating over the list `nums`:** $O(n)$, where n is the number of elements in `nums`.
- **Arithmetic operations**: These are constant-time operations within the loop, i.e., $O(1)$.
- **Modulus operation**: Also a constant-time operation, i.e., $O(1)$.
- **Conditional statement check (if statements)**: Checking a condition is a constant-time operation, i.e., $O(1)$.

Since these are all done in a single pass over the list, we are looking at $O(n) * O(1)$, which simplifies to $O(n)$.

Therefore, the overall time complexity of the function is $O(n)$.

Space Complexity

The space complexity of the function is determined by the amount of additional memory used by the function as the size of the input changes.

- **Variables `ans`, `j`, `mod`:** All use a fixed amount of space regardless of the input size, i.e., $O(1)$.
- **The input list `nums`:** Since the input is not altered and no additional data structures are used that grow with the input size, this does not add to the space complexity.
- **Loop variables `i`, `x`:** they also use a fixed amount of space, i.e., $O(1)$.

Since there are no additional data structures used that scale with the input size, the space complexity is simply the space required for the fixed-size variables, which is $O(1)$.

Overall, the space complexity of the function is $O(1)$.