2270. Number of Ways to Split Array

Medium Array Prefix Sum

Leetcode Link

Problem Description

The problem presents us with an integer array `nums` where our goal is to find the number of "valid splits." A split is considered valid if it meets two conditions:

1. The sum of elements to the left of index `i` (including the element at `i`) is at least as large as the sum of elements to the right of `i`.
2. There is at least one element to the right of index `i`; meaning that `i` can range between `0` and `n - 2`, where `n` is the length of `nums`.

We are asked to return the count of all such valid splits.

Intuition

To find the number of valid splits, we need to check each possible split, and see if it satisfies the two conditions mentioned. While we could calculate the sum of elements on each side of the split every time, this would result in an inefficient solution with a high time complexity.

A more efficient approach is to realize that instead of recalculating sums for every potential split, we can maintain a running total as we iterate through the array. Specifically, we:

- Calculate the total sum `s` of all elements in the array `nums` initially, which will be used to derive the sum of elements on the right side of the split.
- As we move from left to right, we keep adding the values to a running total `t`, which keeps track of the sum of elements on the left side of the split.
- At each index (except the last one to satisfy the second condition), we compare the running total `t` to the remaining sum, `s - t`, which represents the sum of elements on the right side of the split.
- If `t` is greater than or equal to `s - t`, the split is valid, and we increment our answer counter `ans`.
- We continue the aforementioned steps until we reach the second-to-last element of the array.

The intuition behind this approach is that calculating running totals allows us to efficiently update the sums of elements on each side of a potential split without redundant operations. Thus, we can determine the number of valid splits with just a single pass through the array, achieving a time complexity of O(n), where n is the length of the array.

Solution Approach

The implementation of the solution, as presented in the reference code, primarily utilizes a single-pass approach through the `nums` array, leveraging a running cumulative sum to perform the comparison required for determining valid splits.

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

1. Calculate the total sum `s` of the `nums` array. This will allow us to easily compute the sum of elements on the right side of any potential split point.
2. Initialize two variables `ans` and `t` to `0`. The variable `ans` will hold the final count of valid splits, while `t` will represent the cumulative sum of elements from the start of the array up to the current index (initially set to `0` since we haven't started processing elements).
3. Loop over the array `nums` up to the second-to-last element (`nums[:-1]`) to satisfy the condition that there must be at least one element to the right of the split point. Within this loop:
   - Include the current value `v` in the cumulative sum `t` by updating `t += v`.
   - Check if the current split is valid by comparing `t` with `s - t`. If `t >= s - t`, increment `ans` to count the valid split.
4. Continue until the loop ends, then return the total count `ans` as the number of valid splits in the array.

The solution uses a single loop which iterates through the array once, making the time complexity O(n), where `n` is the length of the array `nums`. No additional data structures are used; only a few variables are needed to maintain the sums and the count. This leads to a space complexity of O(1), as the extra space used does not grow with the size of the input array.

This algorithm is straightforward and efficient because it avoids recalculating the sum of elements for each potential split point by keeping a running total. It effectively splits the array into two parts at each index and compares the sums directly. The last value is always excluded from the calculation of `t` as there needs to be an element on the right; this is handled implicitly by using `nums[:-1]` in the loop.

Example Walkthrough

Consider the following small example to illustrate the solution approach: Let's say we have an integer array `nums` with elements `[1, 2, 3, 4, 10]`.

Now, let's go through the solution step-by-step:

1. **Calculate the total sum `s` of the `nums` array.**
   - Here, `s` would be `1+2+3+4+10 = 20`.
2. **Initialize variables `ans` and `t` to `0`.**
   - `ans` will hold the number of valid splits (initially `0` because we haven't found any splits yet).
   - `t` will represent the cumulative sum from the start of the array.
3. **Loop over the array `nums` up to the second-to-last element.** We will go through each element and perform the following steps:
   - i = 0
     - Include the current value (1) in the cumulative sum `t`. Now, `t = 1`.
     - Compare `t` to `s - t` (20 - 1 = 19). The split is not valid because `t < s - t`.
   - i = 1
     - Add the current value (2) to `t`. Now, `t = 1 + 2 = 3`.
     - Compare `t` to `s - t` (20 - 3 = 17). Again, the split is not valid because `t < s - t`.
   - i = 2
     - Include the current value (3) to `t`. Now, `t = 3 + 3 = 6`.
     - Compare `t` to `s - t` (20 - 6 = 14). Still, the split is not valid because `t < s - t`.
   - i = 3
     - Add the current value (4) to `t`. Now, `t = 6 + 4 = 10`.
     - Compare `t` with `s - t` (20 - 10 = 10). The split is valid because `t >= s - t`.
     - Increment `ans` to count this valid split. Now, `ans = 1`.
4. **Continue until the loop ends, then return `ans` as the number of valid splits.**
   - We've reached the end of our loop. Since there are no more elements to process, we stop here and return the total count of valid splits `ans`, which in this case is `1`.

With the provided array `[1, 2, 3, 4, 10]`, our algorithm correctly determines that there is only one valid split, which occurs before the last element (10). The sum of the elements to the left at that point (1+2+3+4) is equal to the sum of the element on the right (10), so the conditions for a valid split are met.

Python Solution

```python
class Solution:
    def waysToSplitArray(self, nums: List[int]) -> int:
        # Calculate the sum of all numbers in the given list
        total_sum = sum(nums)

        # Initialize the prefix sum (prefix_sum) and the count of valid splits (split_count)
        prefix_sum = 0
        split_count = 0

        # Loop through the numbers, except the last one
        for index, value in enumerate(nums[:-1]):  # Exclude the last item as per the problem requirement
            # Add the current value to the prefix sum
            prefix_sum += value

            # Split is valid if prefix sum is greater than or equal to the suffix sum
            if prefix_sum >= total_sum - prefix_sum:
                split_count += 1

        # Return the number of valid splits
        return split_count

```

Java Solution

```java
class Solution {
    public int waysToSplitArray(int[] nums) {
        // Calculate the total sum of the array elements
        long totalSum = 0;
        for (int num : nums) {
            totalSum += num;
        }

        // Initialize the count of ways to split the array
        int countWays = 0;
        // Temporary sum to keep track of the sum of the first part of the array
        long tempSum = 0;

        // Iterate through the array elements, except the last one
        for (int i = 0; i < nums.length - 1; ++i) {
            // Add the current element to the temporary sum
            tempSum += nums[i];
            // Compare the temporary sum with the sum of the remaining elements
            if (tempSum >= totalSum - tempSum) {
                // If the temporary sum is greater than or equal to the remaining sum,
                // increment the count of ways to split the array
                ++countWays;
            }
        }

        // Return the total count of ways to split the array
        return countWays;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int waysToSplitArray(vector<int>& nums) {
        // Use long long for the sum to avoid integer overflow
        long long totalSum = accumulate(nums.begin(), nums.end(), 0LL); // Calculate the total sum of the array
        long long leftSum = 0; // Initialize left part sum
        int count = 0; // Initialize the count of ways to split the array

        // Iterate over the array, stopping at the second to last element
        for (int i = 0; i < nums.size() - 1; ++i) {
            leftSum += nums[i]; // Add current element to the left part sum
            // If left part sum is greater or equal to the right part sum, increment count
            if (leftSum >= totalSum - leftSum) {
                count++;
            }
        }

        // Return the total number of ways to split the array where the left part sum is >= right part sum
        return count;
    }
};

```

Typescript Solution

```typescript
// Define the method to calculate the number of ways to split the array
function waysToSplitArray(nums: number[]): number {
    // Use number type for the sum as TypeScript automatically handles large integers with its number type
    let totalSum: number = nums.reduce((a, b) => a + b, 0); // Calculate the total sum of the array
    let leftSum: number = 0; // Initialize left part sum
    let count: number = 0; // Initialize the count of ways to split the array

    // Iterate over the array, stopping at the second to last element
    for(let i = 0; i < nums.length - 1; ++i) {
        leftSum += nums[i]; // Add current element to the left part sum
        // If left part sum is greater or equal to the right part sum, increment the count
        if (leftSum >= totalSum - leftSum) {
            count++;
        }
    }

    // Return the total number of ways to split the array where the left part sum is greater than or equal to the right part sum
    return count;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the `nums` list. This is because there is a single for loop that iterates through all the elements of the `nums` list except the last one. Within this for loop, we are doing a constant amount of work: adding the current value to the total `t`, and checking a condition that compares `t` with the sum of the remaining elements. Since we calculate the sum of all elements `s` once at the beginning and use it in constant-time comparisons, the running time of the whole loop is linear with respect to the list size.

Space Complexity

The space complexity of the provided code is `O(1)`. No additional space is required that grows with the input size. We use a fixed number of variables (`s`, `ans`, `t`, and `v`) regardless of the input size. The input list is not being copied or modified, and no extra data structures are being used, so the space consumption remains constant.