2574. Left and Right Sum Differences

Easy Array Prefix Sum

Problem Description

The problem presents us with an array of integers called `nums`. Our goal is to construct another array called `answer`. The length of `answer` must be the same as `nums`. Each element in `answer`, `answer[i]`, is defined as the absolute difference between the sum of elements to the left and to the right of `nums[i]`. Specifically,

- `leftSum[i]` is the sum of all elements before `nums[i]` in `nums`. If `i` is 0, `leftSum[i]` is 0 since there are no elements to the left.
- `rightSum[i]` is the sum of all elements after `nums[i]` in `nums`. If `i` is at the last index of `nums`, `rightSum[i]` becomes 0 since there are no elements after it.

We need to calculate this absolute difference for every index in the `nums` array and return the resulting array `answer`.

Intuition

To solve this problem, we look for an efficient way to calculate the left and right sums without repeatedly summing over subsets of the array for each index, which would lead to an inefficient solution with a high time complexity.

We start by observing that for each index `i`, `leftSum[i]` is simply the sum of all previous elements in `nums` up to but not including `nums[i]`, and `rightSum[i]` is the sum of all elements after `nums[i]`.

We can keep track of the `leftSum` as we iterate through the array by maintaining a running sum of the elements we've seen so far. Similarly, we'll keep track of the `rightSum` by starting with the sum of the entire array and subtracting elements as we iterate through the array.

Here's the step by step approach:

- Initialize `left` as 0 to represent the initial `leftSum` (since there are no elements to the left of index 0).
- Calculate the initial `rightSum` as the sum of all elements in `nums`.
- Traverse each element `x` in `nums` from left to right.
- For each element, calculate the current `rightSum` by subtracting the value of the current element (`x`) from `rightSum`.
- Calculate the absolute difference between `left` and `rightSum` and append it to `ans` (the `answer` array).
- Update `left` by adding the value of `x` to include it in the left sum for the next element's computation.
- Return the `answer` array after the loop ends.

By doing this, we efficiently compute the required absolute differences, resulting in a linear time complexity solution (i.e., O(n)) - which is optimal given that we need to look at each element at least once.

Solution Approach

The implementation of the solution follows a straightforward approach using a single pass through the array which uses the two-pointer technique effectively. Here is a breakdown of this approach, including the algorithms, data structures, or patterns used in the solution:

- Initialize a variable called `left` to 0. This variable will hold the cumulative sum of elements to the left of the current index `i` as the array is iterated through.
- Compute the total sum of all elements in `nums` and store this in a variable called `right`. This represents the sum of all elements to the right of the current index `i` before we start the iteration.
- Create an empty list called `ans` that will store the absolute differences between the `left` and `right` sums for each index `i`.
- Start iterating through each element `x` in the `nums` array:
  1. Before we update `left`, `right` still includes the value of the current element `x`. Hence, subtract `x` from `right` to update the `rightSum` for the current index.
  2. Compute the absolute difference between the updated sums `left` and `right` as `abs(left - right)`.
  3. Append the computed absolute difference to the `ans` list.
  4. Add the value of the current element `x` to `left`. After this addition, `left` correctly represents the sum of elements to the left of the next index.
- At the end of this single pass, all elements in `nums` have been considered, and `ans` contains the computed absolute differences.
- Return the list `ans` as the solution.

This algorithm is efficient since it traverses the array only once (`O(n)` time complexity), and uses a constant amount of extra space for the variables `left`, `right`, and the output list `ans` (`O(1)` space complexity, excluding the output list). It avoids the need for nested loops or recursion, which could result in higher time complexity, by cleverly updating `left` and `right` at each step, thereby considering the impact of each element on the `leftSum` and `rightSum` without explicitly computing these each time.

Here's how the pseudo-code might look like:

```
left = 0
right = sum(nums)
ans = []

for x in nums:
    right -= x
    ans.append(abs(left - right))
    left += x

return ans
```

Each step progresses logically from understanding the problem to implementing a solution that addresses the problem's requirements efficiently.

Example Walkthrough

Let's go through an illustrative example using the array `nums = [1, 2, 3, 4]`. We will apply the solution approach step by step.

1. We initialize `left` to `0` and `right` is the sum of all elements in `nums`, which is `1 + 2 + 3 + 4 = 10`.
2. Start with an empty answer array `ans = []`.
3. For the first element `x = 1`:
   - Subtract `x` from `right`: `right = 10 - 1 = 9`.
   - Calculate the absolute difference `abs(left - right) = abs(0 - 9) = 9` and append to `ans`: `ans = [9]`.
   - Add `x` to `left`: `left = 0 + 1 = 1`.
4. For the second element `x = 2`:
   - Subtract `x` from `right`: `right = 9 - 2 = 7`.
   - Calculate the absolute difference `abs(left - right) = abs(1 - 7) = 6` and append to `ans`: `ans = [9, 6]`.
   - Add `x` to `left`: `left = 1 + 2 = 3`.
5. For the third element `x = 3`:
   - Subtract `x` from `right`: `right = 7 - 3 = 4`.
   - Calculate the absolute difference `abs(left - right) = abs(3 - 4) = 1` and append to `ans`: `ans = [9, 6, 1]`.
   - Add `x` to `left`: `left = 3 + 3 = 6`.
6. For the fourth and final element `x = 4`:
   - Subtract `x` from `right`: `right = 4 - 4 = 0`.
   - Calculate the absolute difference `abs(left - right) = abs(6 - 0) = 6` and append to `ans`: `ans = [9, 6, 1, 6]`.
   - `left` would be updated, but since this is the last element it's not necessary for the calculation.
7. The iteration is complete, and the answer array `ans` is `[9, 6, 1, 6]`.

This array represents the absolute differences between the sum of elements to the left and to the right for each element in `nums`. The algorithm has linear time complexity, O(n), where n is the number of elements in `nums`, because it processes each element of the array exactly once.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def leftRightDifference(self, nums: List[int]) -> List[int]:
        # Initialize left_sum as 0 and right_sum as the sum of all elements in nums
        left_sum, right_sum = 0, sum(nums)

        # This list will store the absolute differences
        result = []

        # Iterate through numbers in nums
        for num in nums:
            # Subtract current number from the right sum (as it is moving to the left)
            right_sum -= num

            # Append the absolute difference between left_sum and right_sum
            result.append(abs(left_sum - right_sum))

            # Add the current number to the left sum (as it moved from right to left)
            left_sum += num

        # Return the result list containing absolute differences
        return result
```

Java

```java
import java.util.Arrays; // Import Arrays class to use the stream method

class Solution {
    // Method to find the absolute difference between the sum of numbers to the left
    // and the sum of numbers to the right of each index in an array
    public int[] leftRightDifference(int[] nums) {
        int leftSum = 0; // Initialize sum of left numbers
        int rightSum = Arrays.stream(nums).sum(); // Initialize sum of right numbers using stream
        int n = nums.length;
        int[] differences = new int[n]; // Array to store the differences at each position

        // Iterate through the array elements
        for (int i = 0; i < n; ++i) {
            rightSum -= nums[i]; // Subtract the current element from the right sum
            differences[i] = Math.abs(leftSum - rightSum); // Store the absolute difference at the current position
            leftSum += nums[i]; // Add the current element to the left sum
        }

        return differences; // Return the array containing the differences
    }
}
```

C++

```cpp
#include <vector> // Include necessary header for vector
#include <numeric> // Include numeric header for accumulate function

class Solution {
public:
    // Function to calculate the absolute difference between the sum of elements to the left and right
    // of each element in the array 'nums'
    vector<int> leftRightDifference(vector<int>& nums) {
        int leftSum = 0; // Initialize the left sum to 0
        // Calculate the initial right sum as the total sum of elements in 'nums'
        int rightSum = accumulate(nums.begin(), nums.end(), 0);
        vector<int> differences; // Create a vector to store the differences

        // Iterate through each element in the input 'nums' vector
        for (int num : nums) {
            rightSum -= num; // Decrement rightSum by the current element value
            // Calculate the absolute difference between leftSum and rightSum and push to the 'differences' vector
            differences.push_back(abs(leftSum - rightSum));
            leftSum += num; // Increment leftSum by the current element value
        }

        return differences; // Return the vector containing the differences
    }
};
```

TypeScript

```typescript
function leftRightDifference(nums: number[]): number[] {
    // Initialize left sum as 0. It will represent the sum of elements to the left of the current element
    let leftSum = 0;

    // Calculate the initial right sum, which is the sum of all elements in the array
    let rightSum = nums.reduce((total, currentValue) => total + currentValue);

    // Initialize an empty array to hold the difference between left and right sums at each index
    const differences: number[] = [];

    // Iterate over each element in the input array
    for (const num of nums) {
        // Subtract the current element from the right sum, since it will now be included in the left sum
        rightSum -= num;

        // Calculate the absolute difference between left and right sums and add it to the differences array
        differences.push(Math.abs(leftSum - rightSum));

        // Add the current element to the left sum, as we move to the next index
        leftSum += num;
    }

    // Return the array of differences
    return differences;
}
```

Python3

```python
from typing import List

class Solution:
    def leftRightDifference(self, nums: List[int]) -> List[int]:
        # Initialize left sum as 0 and right sum as the sum of all elements in nums
        left_sum, right_sum = 0, sum(nums)

        # This list will store the absolute differences
        result = []

        # Iterate through numbers in nums
        for num in nums:
            # Subtract current number from the right sum (as it is moving to the left)
            right_sum -= num

            # Append the absolute difference between left_sum and right_sum
            result.append(abs(left_sum - right_sum))

            # Add the current number to the left sum (as it moved from right to left)
            left_sum += num

        # Return the result list containing absolute differences
        return result
```

Time and Space Complexity

The given code snippet calculates the absolute difference between the sum of the numbers to the left and right of the current index in an array. Here is the analysis of its computational complexity:

Time Complexity

The time complexity of this code is primarily dictated by the single loop that iterates through the array. It runs for every element of the array `nums`. Inside the loop, basic arithmetic operations are performed (addition, subtraction, and assignment), which are constant time operations, denoted by `O(1)`. Therefore, the time complexity of the function is `O(n)`, where `n` is the number of elements in the array `nums`.

Space Complexity

The space complexity of the code involves both the input and additional space used by the algorithm. The input space for the array `nums` does not count towards the space complexity of the algorithm. The additional space used by the algorithm is for the variables `left`, `right`, and the array `ans` which stores the result. Since `left` and `right` are single variables that use constant space `O(1)` and the size of `ans` scales linearly with the size of the input array, the space complexity is `O(n)` where `n` is the length of the input array `nums`.