1991. Find the Middle Index in Array

Easy Array Prefix Sum

Problem Description

The problem provides an array of integers called `nums`, which is indexed starting from 0. We are tasked with finding the leftmost `middleIndex` in this array, defined as an index where the sum of elements to the left of `middleIndex` is equal to the sum of elements to the right of `middleIndex`. If `middleIndex` is chosen at the very beginning (0th index) or the end (last index), then the sum of elements on the one side without elements is taken as 0.

Our goal is to identify the lowest index that satisfies this condition or determine that no such index exists and return -1. A middle index that divides the array into two parts with equal sums forms a balance point in the array, and finding the leftmost index means we are looking for the first position that creates equilibrium from the start of the array moving towards the end.

Intuition

To solve this problem, the key is to iterate through the array while keeping track of two sums: the sum of elements to the left of the current index and the sum of elements to the right. This can be done efficiently by maintaining two variables, one for the left sum and another for the right sum (which initially is the sum of all elements in the array).

We start iterating through the array from the leftmost index. For each index `i`, we perform the following steps:

1. We subtract the value at index `i` from the right sum, as we are now considering this value as a part of the left sum.
2. We check if the left sum is equal to the right sum. If it is, we have found a `middleIndex`.
3. If the left sum is not equal to the right sum, we add the value at index `i` to the left sum, effectively moving our middleIndex to the next position.

This way, we are continuously updating the sums on either side of the current index in O(1) time, leading to an overall O(n) time complexity for this algorithm, where n is the length of the array. We only need to do a single pass through the array, which makes this approach efficient. If none of the indices satisfies the middle index condition, we return -1, indicating that no such index exists.

Solution Approach

The solution approach involves the following steps:

1. Initialize a variable `left` with a value of `0`. It will hold the sum of elements to the left of the current index during the iteration.
2. Initialize a variable `right` with the sum of all the elements in the `nums` array, which will be the total sum of elements to the right of the current index before iteration starts.
3. Iterate over the array using a loop, where `i` is the current index and `x` is the value at `nums[i]`.

   a. Subtract the current element `x` from `right` because this step effectively updates the sum to the right of the current index to exclude `x` (since `x` either falls on the middle or moves to the left part in the next step).

   b. Check if `left` is equal to `right`. If yes, we have found a `middleIndex` since the sum of elements on either side of `i` is the same. Therefore, return `i`.

   c. If `left` does not equal `right`, it means the current index `i` is not the `middleIndex`. Add `x` to `left` to include it in the sum of elements on the left side for the subsequent iterations.
4. If the loop completes without returning a middle index, this indicates that no such index exists in the array that satisfies the condition. In this case, return `-1`.

This algorithm uses a single loop making it a linear solution with O(n) time complexity, where `n` is the number of elements in `nums`. The space complexity is O(1), as we are only using two extra variables. This approach is efficient because it only needs a single pass over the data, which is a classic example of the prefix sum technique often used in array processing problems.

Here, the `left` variable maintains the prefix sum of elements seen so far (cumulative sum up to the current index), and `right` variable represents the remaining sum of the array not included in the prefix. It is a straightforward solution that easily identifies the middle index by balancing these two sums.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Consider the array `nums = [2, 3, -1, 8, 4]`. We will use this array to walk through the solution approach step by step.

1. We initialize a variable `left` with a value of `0`. This will keep track of the sum of elements to the left of the current index in our loop.
2. We initialize a variable `right` with the sum of all the elements in the `nums` array, which is `2 + 3 + (-1) + 8 + 4 = 16`.
3. Now we begin iterating over the array, looking to find a `middleIndex` where the sum of elements on either side of it are equal:
   - For `i = 0` (`x = nums[0]` is `2`): a. Subtract `x` (which is `2`) from `right`, so `right` becomes `16 - 2 = 14`. b. Check if `left` equals `right`. They are not equal (`0 != 14`), so we move on. c. Finally, add `x` to `left`, making `left` now `0 + 2 = 2`.
   - For `i = 1` (`x = nums[1]` is `3`): a. Subtract `x` (which is `3`) from `right`, so `right` becomes `14 - 3 = 11`. b. Check if `left` equals `right`. They are not equal (`2 != 11`), so we move on. c. Add `x` to `left`, making `left` now `2 + 3 = 5`.
   - For `i = 2` (`x = nums[2]` is `-1`): a. Subtract `x` (which is `-1`) from `right`, so `right` becomes `11 - (-1) = 12`. b. Check if `left` equals `right`. They are not equal (`5 != 12`), so we move on. c. Add `x` to `left`, making `left` now `5 - 1 = 4`.
   - For `i = 3` (`x = nums[3]` is `8`): a. Subtract `x` (which is `8`) from `right`, so `right` becomes `12 - 8 = 4`. b. Check if `left` equals `right`. They are equal (`4 == 4`), so this means the current index `i` (which is `3`) is the `middleIndex`. We can return `3` as the index where the sum of elements to the left and to the right are equal.

   There is no need to proceed further as we have found our solution.
4. If we hadn't found a `middleIndex`, we would return `-1` at the end of the iteration. But in this case, we return the index `3`.

In this example, the `middleIndex` that satisfies our condition is `3`. Thus, using this approach, we efficiently found the balance point in a single pass through the array.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def findMiddleIndex(self, nums: List[int]) -> int:
        # Initialize the prefix sum (sum of elements to the left of the current index)
        prefix_sum = 0
        # Initialize the total sum (sum of all elements in the array)
        total_sum = sum(nums)

        # Iterate over the array
        for index, value in enumerate(nums):
            # Update the total sum by subtracting the current element
            total_sum -= value

            # Compare the prefix sum and the remaining total sum.
            # If they are equal, the current index is the middle index
            if prefix_sum == total_sum:
                return index

            # Update the prefix sum by adding the current element
            prefix_sum += value

        # If no valid middle index is found, return -1
        return -1
```

Java

```java
class Solution {

    /**
     * Find the index such that the sum of the elements to the left of the index
     * is equal to the sum of the elements to the right of the index.
     *
     * @param nums An array of integers to analyze.
     * @return The leftmost middle index if such an index exists, or -1 otherwise.
     */
    public int findMiddleIndex(int[] nums) {
        // Initialize leftSum to keep track of the sum of elements to the left of the current index
        int leftSum = 0;
        // Calculate the total sum of all elements in the array
        int totalSum = Arrays.stream(nums).sum();
        // Initialize rightSum, which will be updated to keep track of the elements' sum to the right of the current index
        int rightSum = totalSum;

        // Iterate over the array to determine the middle index where the leftSum equals rightSum
        for (int i = 0; i < nums.length; ++i) {
            // Exclude the current element from rightSum
            rightSum -= nums[i];

            // Check if the leftSum equals rightSum at the current index
            if (leftSum == rightSum) {
                // If sums are equal, return the current index as the answer
                return i;
            }

            // Include the current element in the leftSum for the next iteration
            leftSum += nums[i];
        }

        // If no such index is found, return -1 to indicate failure
        return -1;
    }
}
```

C++

```cpp
#include <vector>
#include <numeric> // For std::accumulate

class Solution {
public:
    // This method finds the index such that the sum of the elements to the left
    // of it is equal to the sum of the elements to the right of it.
    int findMiddleIndex(vector<int>& nums) {
        int leftSum = 0; // Sum of elements to the left of the current index
        // Calculate the sum of all elements in the vector as the initial right sum.
        int rightSum = std::accumulate(nums.begin(), nums.end(), 0);

        for (int index = 0; index < nums.size(); ++index) {
            // Subtract the current element from right sum because it's not part of the right sum anymore.
            rightSum -= nums[index];
            // If the left sum is equal to the right sum, we have found the middle index.
            if (leftSum == rightSum) {
                return index;
            }
            // Add the current element to the left sum after checking.
            // since it will be part of the left sum when index is incremented.
            leftSum += nums[index];
        }
        // Return -1 if no index can satisfy the condition.
        return -1;
    }
};
```

TypeScript

```typescript
// Function to find the middle index where the sum of the numbers to the left
// is equal to the sum of the numbers to the right.
function findMiddleIndex(nums: number[]): number {
    // Initialize the left sum to 0
    let sumToLeft = 0;
    // Compute the sum for the entire array, which will start as the sum to the right
    let sumToRight = nums.reduce((accumulator, currentValue) => accumulator + currentValue, 0);

    // Iterate through the nums array to find the middle index
    for (let i = 0; i < nums.length; i++) {
        // Subtract the current element from sumToRight before comparing
        sumToRight -= nums[i];

        // If sumToLeft is equal to sumToRight, the middle index is found
        if (sumToLeft === sumToRight) {
            return i;
        }

        // Add the current element to sumToLeft to use it for the next iteration's comparison
        sumToLeft += nums[i];
    }

    // If no index satisfies the condition, return -1
    return -1;
}
```

```python
from typing import List

class Solution:
    def findMiddleIndex(self, nums: List[int]) -> int:
        # Initialize the prefix sum (sum of elements to the left of the current index)
        prefix_sum = 0
        # Initialize the total sum (sum of all elements in the array)
        total_sum = sum(nums)

        # Iterate over the array
        for index, value in enumerate(nums):
            # Update the total sum by subtracting the current element
            total_sum -= value

            # Compare the prefix sum and the remaining total sum.
            # If they are equal, the current index is the middle index
            if prefix_sum == total_sum:
                return index

            # Update the prefix sum by adding the current element
            prefix_sum += value

        # If no valid middle index is found, return -1
        return -1
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input list `nums`. This is because the code iterates through each element of `nums` exactly once in a single for-loop, performing a constant number of operations at each step.

Space Complexity

The space complexity of the provided code is `O(1)` since it uses a fixed amount of extra space: variables `left`, `right`, `i`, and `x`. The total amount of extra space required does not grow with the size of the input list, hence it is constant.