724. Find Pivot Index

Easy Array Prefix Sum

Leetcode Link

Problem Description

The problem presents an array of integers and asks us to find what is known as the **pivot index**. The pivot index is when the sum of the numbers to the left of that index is equal to the sum of the numbers to the right of it. The problem clarifies a couple of edge cases:

- If the pivot index is at the very beginning of the array (index 0), the sum of the numbers to the left is considered to be 0 since there are no numbers left of it.
- Likewise, if the pivot is at the very end of the array, the sum of the numbers to the right is 0 for similar reasons.

Our task is to return the **leftmost pivot index**. In other words, we're looking for the first position from the left where this balance occurs. If no such index can be found where the sums to the left and right are equal, we must return `-1`.

Intuition

The challenge is to find the pivot index without checking each possible index with brute force, which would be time-consuming for very large arrays. To make it more efficient, we can look for a way to find the balance as we go through the array just once.

To solve this problem, we need to track the sums of the numbers to the left and to the right of each index. An initial idea might be to calculate the sum of all numbers to the left and all those to the right for each index in the array. However, this would involve a lot of repeated work because we'd be summing over many of the same elements multiple times.

In the solution provided, we use a clever technique to avoid this repetition. We start by calculating the sum of the entire array and assign it to `right`. As we iterate over each element with the index `i`, we do the following:

1. Subtract the current element's value from `right` because we're essentially moving our 'pivot point' one step to the right, so everything that was once to the right is now either the pivot or to its left except the current element.
2. Check if the current sums to the left and to the right of `i` (not including `i`) are equal. If they are, `i` is our pivot index.
3. If they're not equal, we add the current element's value to `left` because if we continue to the next element, our 'pivot point' will have moved, and the current element will now be considered part of the left sum.

By iterating over the array only once and updating `left` and `right` appropriately, we efficiently find the pivot index if it exists or determine that no pivot index is present.

Solution Approach

The implementation of the solution involves a single pass over the array, which makes use of two helpful constructs:

1. **Cumulative Sums**: We don't calculate the sum on the fly for each pivot candidate; instead, we maintain a running total (`left`) that we build up as we iterate through the array.
2. **Total Sum**: Before we start iterating, we calculate the total sum of the array (`right`). This sum represents everything that would initially be to the right of a hypothetical pivot at the start of the array.

Algorithmically, the steps are:

- We initialize `left` to `0`, since at the start of the array, there is nothing to the left.
- We initialize `right` to the sum of all elements in `nums` using the `sum` function, representing the sum to the right of our starting index.
- We then iterate over each element `x` of the array alongside its index `i` using Python's `enumerate` function, which gives us both the index and the value at that index in the array.
- For each element `x`, we deduct `x` from `right`, which now represents the sum of elements to the right of our current index `i`.
- We compare `left` and `right`; if they are equal, we've found a pivot index, and `i` is returned immediately.
- If `left` does not equal `right`, we then add the value of `x` to `left`, preparing for the next iteration where `i` will be one index to the right.
- If we complete the entire loop without finding equal `left` and `right` sums, which means there is no pivot index, we return `-1`.

Thus, the approach cleverly manages the sum totals on either side of the current index by simply moving the index from left to right through the array, updating the sums by adding or subtracting the current element as appropriate. It does not require additional data structures or complex patterns but uses arithmetic operations judiciously to keep track of potential pivot indices.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the array `nums = [1, 7, 3, 6, 5, 6]`.

1. **Initial Setup**: Start with `left` as `0` and `right` as the sum of the entire array, which is `1 + 7 + 3 + 6 + 5 + 6 = 28`.
2. **First Iteration (i = 0)**:
   - We have `nums[i] = 1`, so update `right` to be the sum of numbers to the right of index 0.
     `right = right - nums[i] = 28 - 1 = 27`.
   - Compare `left` and `right`. Since `left` is `0` and `right` is `27`, they are not equal, so index 0 is not a pivot.
   - Update `left` to include `nums[i]`.
     `left = left + nums[i] = 0 + 1 = 1`.
3. **Second Iteration (i = 1)**:
   - `nums[i] = 7`, so update `right` to be the sum of numbers to the right of index 1.
     `right = right - nums[i] = 27 - 7 = 20`.
   - Compare `left` and `right`. `left` is `1` and `right` is `20`, not equal, no pivot yet.
   - Update `left` to include `nums[i]`.
     `left = left + nums[i] = 1 + 7 = 8`.
4. **Third Iteration (i = 2)**:
   - `nums[i] = 3`, adjust `right`.
     `right = right - nums[i] = 20 - 3 = 17`.
   - Compare `left` and `right`, `left` is `8` and `right` is `17`, not equal.
   - Update `left`.
     `left = left + nums[i] = 8 + 3 = 11`.
5. **Fourth Iteration (i = 3)**:
   - `nums[i] = 6`, adjust `right`.
     `right = right - nums[i] = 17 - 6 = 11`.
   - Compare `left` and `right`. Here both are `11`, so we found a pivot index `i = 3`.

Since we've found a pivot index, we would return it (`3`). If we hadn't found a pivot index at this point, we would've continued with the process for the rest of the array. If no pivot index was found after the last iteration, we'd return `-1`.

This example walk-through demonstrates how the approach scans through the array, updating the sums of values to the left and right of the current index, and checks equality at each step without needing to sum over large subarrays repeatedly.

Python Solution

```python
from typing import List

class Solution:
    def pivotIndex(self, nums: List[int]) -> int:
        # Initialize the left sum and right sum.
        # Right sum is the sum of all elements initially.
        left_sum, right_sum = 0, sum(nums)

        # Iterate through the list
        for index, element in enumerate(nums):
            # Subtract the element from right sum as it will not be part of the right segment
            right_sum -= element

            # If left sum and right sum are equal, the current index is the pivot
            if left_sum == right_sum:
                return index  # Return the pivot index

            # Add the element to left sum as it becomes part of the left segment
            left_sum += element

        # If no pivot index is found, return -1.
        return -1
```

Java Solution

```java
class Solution {
    public int pivotIndex(int[] nums) {
        // Initialize the sum of numbers to the left of the pivot to 0
        int sumLeft = 0;

        // Calculate the total sum of the array elements
        int totalSum = 0;
        for (int num : nums) {
            totalSum += num;
        }

        // Iterate through the array to find the pivot index
        for (int i = 0; i < nums.length; i++) {
            // Right sum is total sum minus the current element
            // since the current element is to be the pivot
            int sumRight = totalSum - sumLeft - nums[i];

            // If the sum of numbers to the left of the pivot is equal to
            // the sum of numbers to the right of the pivot, return current index
            if (sumLeft == sumRight) {
                return i;
            }

            // Update the sumLeft by adding the current element's value
            sumLeft += nums[i];
        }

        // If no pivot index is found, return -1
        return -1;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric> // Required for std::accumulate function

class Solution {
public:
    // Function to find the pivot index of the array
    int pivotIndex(vector<int>& nums) {
        int sumLeft = 0; // Initialize sum of elements to the left of the pivot
        // Compute the total sum of the array elements with std::accumulate
        int sumRight = std::accumulate(nums.begin(), nums.end(), 0);

        // Iterate over the array elements
        for (int i = 0; i < nums.size(); ++i) {
            sumRight -= nums[i]; // Subtract the current element from the right sum as it's under consideration

            // If left sum is equal to right sum, the current index is the pivot index
            if (sumLeft == sumRight) {
                return i; // Return the current index as the pivot index
            }

            sumLeft += nums[i]; // Add the current element to the left sum before moving to the next element
        }

        return -1; // If no pivot index is found, return -1
    }
};
```

Typescript Solution

```typescript
// Function to find the pivot index of an array
// The pivot index is where the sum of the numbers to the left of the index
// is equal to the sum of the numbers to the right of the index
function pivotIndex(nums: number[]): number {
    let sumLeft = 0; // Initialize sum of elements to the left
    let sumRight = nums.reduce((a, b) => a + b, 0); // Initialize sum of all elements

    // Iterate through the array elements
    for (let i = 0; i < nums.length; ++i) {
        sumRight -= nums[i]; // Subtract the current element from the right sum

        // Check if left sum and right sum are equal
        if (sumLeft === sumRight) {
            return i; // Return the current index as pivot index
        }

        sumLeft += nums[i]; // Add the current element to the left sum
    }

    return -1; // If no pivot index is found, return -1
}
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of elements in the `nums` list. This is because we iterate over all elements of the array exactly once in a single loop to find the pivot index.

The space complexity of the code is `O(1)` since we use only a constant amount of extra space for the variables `left`, `right`, `i`, and `x` irrespective of the input size.