2256. Minimum Average Difference

Medium Array Prefix Sum Leetcode Link

Problem Description

This problem deals with an array of integers where the task is to find the index that minimizes the average difference between the two segments of the array divided by that index. The array is split at every index from 0 to `n - 1`, creating two non-overlapping segments. The left segment includes all elements from the beginning of the array to the current index, and the right segment includes all elements after the current index to the end of the array.

The average difference at an index `i` is the absolute difference between the average of the numbers to the left of (and including) `i` and the average of the numbers to the right of `i`. The averages are calculated as the sum of the elements in the segment divided by the number of elements in the segment, with the result rounded down to the nearest integer. If the right segment is empty (which happens when `i` is the last index), its average is considered to be 0.

The problem asks to determine the index that has the minimum average difference and to return the smallest index if there are multiple such indices.

Intuition

To solve this problem, one intuitive approach is to process each index and calculate the two averages (left and right segment averages) and their absolute difference. However, computing the sum of elements repeatedly for each index would result in a higher time complexity.

A more efficient method involves prefix sums, which can significantly optimize the repeated sum calculations. By maintaining a running total sum from the start of the array (`pre`) and a running total sum from the end of the array (`suf`), it becomes easier to compute the averages quickly at each index. As the index moves from left to right, elements are transferred from the right segment to the left segment. This is done by adding the current element to `pre` and subtracting it from `suf`.

The solution tracks the smallest average difference found (`mi`) and its corresponding index (`ans`). For each index `i`, it calculates the average of the left segment, the average of the right segment (except when the right segment is empty), and their absolute difference (`t`). If `t` is smaller than the smallest difference seen so far, `mi` is updated with `t` and `ans` with `i`. After traversing all indices, the index corresponding to the minimal average difference is returned.

This approach efficiently narrows down the index with the minimum average difference without calculating the sum for each segment more than once, resulting in an overall time complexity of O(n), where n is the length of the array.

Solution Approach

The implementation of the solution uses a simple loop through the array and exploits the concept of prefix sums to efficiently calculate the averages required for the problem statement. The algorithm doesn't require any complex data structures or patterns beyond arrays and basic arithmetic operations.

Here's the step-by-step implementation of the solution:

1. Initialize two variables `pre` and `suf`. Set `pre` to 0 because, initially, there are no elements in the left segment. Set `suf` to the sum of all elements in `nums` - this represents the sum of the right segment before any iteration.

2. Set up variables `ans` for storing the index of the minimum average difference found, and `mi` for storing the minimum average difference itself. `mi` is initialized to `inf`, which is a placeholder for infinity, to ensure that any actual difference calculated will be less than this initial value.

3. Iterate over the elements in `nums`. For each index `i` and its corresponding value `x` in `nums`, do the following:
   - Add `x` to `pre`, as `x` is now part of the left segment.
   - Subtract `x` from `suf`, as `x` is no longer part of the right segment.
   - Calculate the average of the left segment as `a = pre // (i + 1)`.
   - If the right segment is non-empty (determined by `n - i - 1 > 0`), calculate its average as `b = suf // (n - i - 1)`. If the right segment is empty, set `b` to 0.
   - Calculate the absolute difference between the two averages `t = abs(a - b)`.
4. If the absolute difference `t` is less than the current minimum `mi`, update `mi` with `t` and `ans` with the current index `i`.

5. After the loop completes, the variable `ans` holds the index of the minimum average difference. Return `ans` as the result.

The algorithm solely uses integer division and basic arithmetic operations, which creates an efficient way of obtaining the minimum average difference without recalculating the sums for each segment for every index. The time complexity is O(n) because the algorithm requires a single pass through the array, and the space complexity is O(1) as it only uses a fixed number of extra variables.

Example Walkthrough

Let's consider the array `nums = [3, 1, 2, 4, 3]` to illustrate the solution approach.

1. Start by initializing `pre` to 0 and `suf` to the sum of all elements in `nums`, which is `3 + 1 + 2 + 4 + 3 = 13`. Also, initialize `mi` to infinity, and `ans` to -1 (which will later hold the index of the minimum average difference).

2. Now, iteratively process each element of the array:
   - For index `i = 0`:
     - Left Segment `[3]` and Right Segment `[1, 2, 4, 3]`
     - `pre = 0 + 3 = 3`
     - `suf = 13 - 3 = 10`
     - Left average `a = 3 // 1 = 3`
     - Right average `b = 10 // 4 = 2`
     - Absolute difference `t = abs(3 - 2) = 1`
     - Since `t < mi` (1 < infinity), update `mi` to 1 and `ans` to 0.
   - For index `i = 1`:
     - Left Segment `[3, 1]` and Right Segment `[2, 4, 3]`
     - `pre = 3 + 1 = 4`
     - `suf = 10 - 1 = 9`
     - Left average `a = 4 // 2 = 2`
     - Right average `b = 9 // 3 = 3`
     - Absolute difference `t = abs(2 - 3) = 1`
     - Since `t = mi` (1 == 1), no update to `mi` or `ans` as we pick the smallest index.
   - For index `i = 2`:
     - Left Segment `[3, 1, 2]` and Right Segment `[4, 3]`
     - `pre = 4 + 2 = 6`
     - `suf = 9 - 2 = 7`
     - Left average `a = 6 // 3 = 2`
     - Right average `b = 7 // 2 = 3`
     - Absolute difference `t = abs(2 - 3) = 1`
     - Since `t = mi` (1 == 1), no update as `ans` is already at the smallest index 0.
   - For index `i = 3`:
     - Left Segment `[3, 1, 2, 4]` and Right Segment `[3]`
     - `pre = 6 + 4 = 10`
     - `suf = 7 - 4 = 3`
     - Left average `a = 10 // 4 = 2`
     - Right average `b = 3 // 1 = 3`
     - Absolute difference `t = abs(2 - 3) = 1`
     - Because `t = mi` (1 == 1), there's no update needed.
   - For index `i = 4`:
     - Left Segment `[3, 1, 2, 4, 3]` and no Right Segment
     - `pre = 10 + 3 = 13`
     - `suf = 3 - 3 = 0`
     - Left average `a = 13 // 5 = 2`
     - Right average `b = 0` (since there are no elements on the right)
     - Absolute difference `t = abs(2 - 0) = 2`
     - No update required as `t > mi` (2 > 1).
3. After processing all indices, the smallest average difference `mi` is 1, and the `ans` associated with this difference is at index 0.

4. The result returned is `ans = 0`, which is the index of the minimum average difference.

In this example, the process was performed in a single pass (O(n) time complexity), and we did not require extra space aside from a few variables (O(1) space complexity). This walkthrough demonstrates the efficiency and correctness of the proposed solution.

Python Solution

```python
class Solution:
    def minimumAverageDifference(self, nums: List[int]) -> int:
        # Initialize prefix sum, suffix sum, and the number of elements
        prefix_sum, suffix_sum = 0, sum(nums)
        num_elements = len(nums)

        # Initialize variables to track the minimum average difference and its index
        min_index, min_difference = 0, float('inf')

        # Iterate through the list to calculate the differences
        for index, value in enumerate(nums):
            # Update the prefix and suffix sums
            prefix_sum += value
            suffix_sum -= value

            # Calculate the average of the prefix and suffix
            prefix_avg = prefix_sum // (index + 1)
            # Calculate the average of suffix; handle the case when it becomes empty
            suffix_avg = 0 if num_elements - index - 1 == 0 else suffix_sum // (num_elements - index - 1)

            # Calculate the absolute difference of the averages
            current_difference = abs(prefix_avg - suffix_avg)

            # If the current difference is smaller than the recorded minimum, update the result
            if current_difference < min_difference:
                min_index = index
                min_difference = current_difference

        # Return the index of the element that gives the minimum average difference
        return min_index

```

Java Solution

```java
class Solution {
    public int minimumAverageDifference(int[] nums) {
        // Get the length of the input array
        int arrayLength = nums.length;
        // Initialize prefix sum and suffix sum variables
        long prefixSum = 0, suffixSum = 0;

        // Calculate the total sum of the array elements (suffixSum)
        for (int number : nums) {
            suffixSum += number;
        }

        // Initialize the variable to store the index of the result
        int resultIndex = 0;
        // Initialize the minimum difference variable with the maximum value possible
        long minimumDifference = Long.MAX_VALUE;

        // Iterate through the array to calculate prefix and suffix sums dynamically
        for (int i = 0; i < arrayLength; ++i) {
            // Add the current number to the prefix sum
            prefixSum += nums[i];
            // Subtract the current number from the suffix sum
            suffixSum -= nums[i];

            // Calculate the average of the prefix part
            long prefixAvg = prefixSum / (i + 1);
            // Calculate the average of the suffix part, or set to 0 if there are no elements remaining
            long suffixAvg = arrayLength - i - 1 == 0 ? 0 : suffixSum / (arrayLength - i - 1);
            // Calculate the absolute difference between the two averages
            long currentDifference = Math.abs(prefixAvg - suffixAvg);

            // If the current difference is less than the minimum found so far, update the result and minimum
            if (currentDifference < minimumDifference) {
                resultIndex = i;
                minimumDifference = currentDifference;
            }
        }
        // Return the index where the absolute difference between the averages is minimum
        return resultIndex;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <numeric>  // for std::accumulate
#include <cmath>    // for std::abs

class Solution {
public:
    int minimumAverageDifference(std::vector<int>& nums) {
        int numElements = nums.size();                                // Get the total number of elements in nums.
        using ll = long long;                                         // Use 'll' as an alias for 'long long' type for larger numbers.
        ll prefixSum = 0;                                            // Initialize prefix sum of first part of the array.
        ll suffixSum = std::accumulate(nums.begin(), nums.end(), 0LL); // Calculate the sum of all elements in the array.

        int minIndex = 0;                   // Store the index at which the minimum average difference occurs.
        ll minDifference = suffixSum;       // Initially set the minimum difference to the suffix sum.

        for (int i = 0; i < numElements; ++i) {
            prefixSum += nums[i];                             // Add the current element to the prefix sum.
            suffixSum -= nums[i];                             // Remove the current element from the suffix sum.

            ll prefixAvg = prefixSum / (i + 1);               // Calculate average of the prefix part.
            ll suffixAvg = (numElements - i - 1 == 0) ? 0 :   // Calculate average of the suffix part,
                           suffixSum / (numElements - i - 1);  // avoid division by zero by checking the size.

            ll currentDifference = std::abs(prefixAvg - suffixAvg);  // Calculate absolute difference between averages.

            if (currentDifference < minDifference) {              // Check if the current difference is less
                minIndex = i;                                     // than the minimum difference found so far.
                minDifference = currentDifference;                // If so, update minimum difference and index.
            }
        }
        return minIndex;  // Return the index of the minimum average difference.
    }
};

```

Typescript Solution

```typescript
function minimumAverageDifference(nums: number[]): number {
    const lengthOfNums = nums.length; // Length of the input array

    // Prefix sum which starts at zero and will store the sum of the elements before the current index
    let prefixSum = 0;

    // Suffix sum which starts as the sum of all elements and will store the sum of the elements after the current index
    let suffixSum = nums.reduce((acc, current) => acc + current, 0);

    // The index with the smallest average difference
    let minDifferenceIndex = 0;

    // The smallest average difference encountered so far initialized with a value that suffix sum contributes to before any iteratic
    let minimumDifference = suffixSum;

    for (let index = 0; index < lengthOfNums; ++index) {
        prefixSum += nums[index]; // Add current element to the prefix sum
        suffixSum -= nums[index]; // Subtract current element from the suffix sum

        // Calculate the average of the prefix by dividing prefixSum by the number of elements considered
        const prefixAverage = Math.floor(prefixSum / (index + 1));

        // Calculate the average of the suffix, checking for division by zero when index is at the last element
        const suffixAverage = (lengthOfNums - index - 1) === 0 ? 0 : Math.floor(suffixSum / (lengthOfNums - index - 1));

        // Calculate the absolute difference between the prefix and suffix averages
        const difference = Math.abs(prefixAverage - suffixAverage);

        if (difference < minimumDifference) {
            // If the current difference is smaller, update minDifferenceIndex and minimumDifference
            minDifferenceIndex = index;
            minimumDifference = difference;
        }
    }

    return minDifferenceIndex; // Return the index where the minimum average difference occurs
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the array `nums`. The calculation of the sum of the array `sum(nums)` is done in `O(n)`. Inside the loop, only constant time operations are done, and since the loop runs `n` times, the loop operations also amount to `O(n)`. Thus, the entire function runs in linear time with respect to the size of the array.

The space complexity of the code is `O(1)`. Independent of the length of the input array, a fixed number of variables are used (`pre`, `suf`, `n`, `ans`, `mi`, `a`, `b`, and `t`). These require a constant amount of space, and hence the space complexity does not scale with the input size.