1343. Number of Sub-arrays of Size K and Average Greater than or Equal to Threshold

Medium Array Sliding Window

Leetcode Link

Problem Description

The problem provides an array of integers `arr`, along with two other integers `k` and `threshold`. The task is to determine how many continuous sub-arrays (consecutive elements) of `arr` with size `k` have an average value greater than or equal to `threshold`. A sub-array's average is calculated by adding all the elements in the sub-array and then dividing by `k`. The result should be the count of all sub-arrays that meet the specified average criterion.

Intuition

The intuition behind the solution relies on the concept of a sliding window. The window slides through the array, maintaining a fixed size of `k`, and we compute the sum of the elements within this window. To find the average, we divide the sum by `k`. If the average is greater than or equal to the `threshold`, we increment our answer count.

Instead of recalculating the sum of elements each time the window moves, we can optimize this by subtracting the element that is left behind and adding the new element that comes into the window. This approach is efficient because it updates the sum by removing the exiting element and adding the entering element, thus requiring only a constant number of operations for each shift of the sliding window.

This is achieved in the solution by initially computing the sum of the first `k` elements. Then, as we move the window across the array by iterating starting from the `k`th element, we adjust the sum by simply adding the new element at the front and subtracting the last element of the previous window. After updating the sum, we check if the current average meets the threshold and increment our count accordingly.

Solution Approach

The solution uses a sliding window technique to solve this problem efficiently. Here's how it works:

1. **Initial Window Sum**: We first calculate the sum of the first `k` elements. This sum represents the total of the first window of size `k`. To decide if this window meets the requirements, we check if the average (the sum divided by `k`) is greater than or equal to the `threshold`.
2. **Sliding the Window**: After processing the first window, we iteratively move the window one element to the right. For each move, we perform the following actions:
   - **Add New Element**: Include the new element (rightmost element of the new window) in the sum. This is done by `s += arr[i]`.
   - **Remove Exiting Element**: Exclude the leftmost element of the previous window from the sum. Achieved with `s -= arr[i - k]`.
   - These two operations effectively update the sum to reflect the total of the current window.
3. **Check and Count**: After adjusting the sum for the current window, we check if the average meets or exceeds the `threshold`. If so, we increment our count by one. This is done by `ans += int(s / k >= threshold)`.
4. **Iterating Over the Array**: We continue sliding the window and adjusting our sum and count until we've covered the entire array. The loop starts from `k` and goes till the end of `arr`.
5. **Return the Result**: Once the iteration is complete, the variable `ans` holds the total number of sub-arrays of size `k` with an average greater than or equal to `threshold`. The function then returns `ans`.

The beauty of this solution lies in its simplicity and efficiency. Instead of recalculating the sum for each sub-array from scratch, it smartly updates the existing sum, thus maintaining a constant time operation for each step in the window slide, which makes the algorithm O(n) where n is the number of elements in `arr`.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose we have the following inputs:

- `arr = [2, 5, 5, 6, 3, 2, 2, 1]`
- `k = 3`
- `threshold = 4`

Now, let's walk through the solution step by step:

1. **Initial Window Sum**:
   - First, we calculate the sum of the first `k=3` elements `[2, 5, 5]`. The sum is `2 + 5 + 5 = 12`.
   - To check if this window meets the requirements, we compute the average: `12 / 3 = 4`.
   - The average is equal to the `threshold` of 4, so this window qualifies. Our initial count `ans = 1`.
2. **Sliding the Window**:
   - We slide the window to the right to include the next element (`6`) and exclude the first element (`2`) of the prior window.
   - We update the sum: `12 (previous sum) + 6 (new element) - 2 (exiting element) = 16`.
   - We check if the new window `[5, 5, 6]` has an average greater than or equal to `threshold`: `16 / 3 = 5.33` (greater than 4).
   - Increment our count: `ans = 2`.
3. **Repeat the Process**:
   - Slide the window to the next position: `[5, 6, 3]`.
   - Update the sum: `16 (previous sum) + 3 (new element) - 5 (exiting element) = 14`.
   - Check the average: `14 / 3 = 4.67` (greater than 4). Increment the count: `ans = 3`.
   - Continue this process for the rest of the array: `[6, 3, 2]`, `[3, 2, 2]`, and finally `[2, 2, 1]`.
4. **Final Count**:
   - After processing all windows we find that only the windows `[2, 5, 5]`, `[5, 5, 6]`, and `[5, 6, 3]` meet the threshold average. Therefore, the final count `ans = 3`.
5. **Return the Result**:
   - We return the final count `ans`, which is `3`. Hence, there are three sub-arrays of size `k` with an average value greater than or equal to the threshold.

Python Solution

```python
from typing import List

class Solution:
    def numOfSubarrays(self, arr: List[int], k: int, threshold: int) -> int:
        # Initialize the sum of the first 'k' elements
        current_sum = sum(arr[:k])

        # Calculate the initial average and check if it meets or exceeds the threshold.
        # Convert the comparison result to an integer (True becomes 1, False becomes 0).
        subarrays_count = int(current_sum / k >= threshold)

        # Iterate over the array starting from the k-th element.
        for i in range(k, len(arr)):
            # Update the sum by adding the new element and subtracting the oldest element (sliding window).
            current_sum += arr[i] - arr[i - k]

            # Check if the updated average meets or exceeds the threshold and update the count.
            subarrays_count += int(current_sum / k >= threshold)

        # Return the number of subarrays where the average is greater or equal to the threshold.
        return subarrays_count

# Example usage:
# sol = Solution()
# print(sol.numOfSubarrays([2, 1, 3, 4, 1], 3, 2)) # Output: 3
```

Java Solution

```java
class Solution {
    public int numOfSubarrays(int[] arr, int k, int threshold) {
        // Initialize the sum of the first 'k' elements.
        int sum = 0;
        for (int i = 0; i < k; ++i) {
            sum += arr[i];
        }

        // Calculate the initial average and determine if it meets or exceeds the threshold.
        // Increment the count if it does.
        int count = (sum / k >= threshold) ? 1 : 0;

        // Slide the window of size 'k' across the array and update the sum and count accordingly.
        for (int i = k; i < arr.length; ++i) {
            // Update the sum by adding the new element and subtracting the oldest element.
            sum += arr[i] - arr[i - k];

            // If the updated average meets or exceeds the threshold, increase the count.
            count += (sum / k >= threshold) ? 1 : 0;
        }

        // Return the total count of subarrays that meet the threshold condition.
        return count;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric> // Include necessary headers

class Solution {
public:
    int numOfSubarrays(vector<int>& arr, int k, int threshold) {
        // Calculate the sum of the first 'k' elements
        int windowSum = accumulate(arr.begin(), arr.begin() + k, 0);

        // Initialize the count of subarrays with sum above the threshold
        int count = (windowSum >= k * threshold) ? 1 : 0;

        // Loop over the elements of the array, starting from the 'k'th element
        for (int i = k; i < arr.size(); ++i) {
            // Slide the window by one element to the right: subtract the element leaving the window
            // and add the new element
            windowSum += arr[i] - arr[i - k];

            // If the new window sum satisfies the threshold condition, increment the count
            count += (windowSum >= k * threshold) ? 1 : 0;
        }

        // Return the final count of subarrays
        return count;
    }
};

// Note: To use the updated code, you'll need to have '#include <vector>' and '#include <numeric>'
// at the top of your code to bring in the necessary functionality from the C++ standard library.
```

Typescript Solution

```typescript
function numOfSubarrays(arr: number[], k: number, threshold: number): number {
    // Initialize the sum of the first 'k' elements
    let sum = arr.slice(0, k).reduce((accumulator, current) => accumulator + current, 0);

    // Check if the average of the first 'k' elements meets or exceeds the threshold
    // If it does, we increment our answer count 'subarrayCount'
    let subarrayCount = sum >= k * threshold ? 1 : 0;

    // Iterate over the array starting from the 'k'th element
    for (let i = k; i < arr.length; ++i) {
        // Update the sum to include the next element and exclude the (i-k)'th element
        sum += arr[i] - arr[i - k];

        // Check if the updated sum's average meets or exceeds the threshold,
        // if so, increment 'subarrayCount'
        subarrayCount += sum >= k * threshold ? 1 : 0;
    }

    // Return the total count of subarrays that have met or exceeded the threshold average
    return subarrayCount;
}
```

Time and Space Complexity

The provided code has a time complexity of `O(n)` where `n` is the length of the array. This linear time complexity arises because the code iterates over the array once, performing a constant number of operations for each element in the array after initializing a sum of the first `k` elements.

The key to achieving `O(n)` time complexity lies in the sliding window technique used to calculate the sum of each subarray of length `k`. Instead of recalculating the sum from scratch, the code maintains the sum of the current window by subtracting the element that is leaving the window and adding the new element that enters the window. This update happens in constant time `O(1)`.

The space complexity of the code is `O(1)` since the code only uses a fixed amount of additional space. Variables `s` and `ans` do not depend on the size of the input array and only a constant amount of extra space is used regardless of the input size.