643. Maximum Average Subarray I

Easy | Array | Sliding Window

Problem Description

The problem presents an array of integers, `nums`, with `n` elements, and another integer, `k`. The objective is to find a contiguous subarray within `nums` that has a length exactly equal to `k` and which has the maximum average value compared to all other possible subarrays of the same length.

Constraints that might apply to this problem, such as the range of values for `n`, `k`, and the individual elements in `nums`, are not stated here. The answer needs to be sufficiently accurate such that a calculation error less than `10^-5` is acceptable.

Intuition

To solve this problem effectively, we can use the "sliding window" technique. The key idea behind this technique is to maintain a "window" of size `k` that moves through the array, one element at a time, updating the sum of the window as it goes. By doing this, we avoid recalculating the sum for each subarray from scratch, which leads to significant time savings.

Here's the intuition step-by-step:

- First, calculate the sum of the initial window, which is the sum of the first `k` elements.
- This sum is our current maximum sum (and, as it stands, the maximum average since we would divide this by `k`).
- Now, slide the window by one element to the right. This means adding the element that comes after the current window and subtracting the element that is leaving the window.
- Update the maximum sum if the new window's sum is greater.
- Repeat this process of sliding the window and updating the maximum sum until we have visited all possible windows of size `k`.
- Since the maximum sum will have been found by the time we've checked all windows, we can simply take the maximum sum and divide it by `k` to find the maximum average.

This approach is efficient because it calculates the necessary sum in constant time for each step of the sliding window instead of linear time if we were to add up `k` elements again and again. Therefore, the overall time complexity is O(n), where `n` is the number of elements in the array.

Solution Approach

The solution is implemented using the sliding window pattern as mentioned:

1. **Initialization**: The first step is to calculate the sum of the first `k` elements. This is done via the expression `sum(nums[:k])`. We initialize a variable `s` with this initial sum, and also we initialize an `ans` variable with the same value because, at this point, this is the maximum sum we have found, and hence the maximum average as well.
2. **Sliding the Window**:
   - The for loop with `for i in range(k, len(nums)):` begins the process of sliding the window across the array. The variable `i` represents the right boundary of the current window.
   - As the window slides, we add the new rightmost element to our sum with `s += nums[i]` and remove the leftmost element of the previous window with `- nums[i - k]`. This enables us to maintain the sum of exactly `k` elements.
   - After adjusting the sum, we then check if the new sum (`s`) is greater than the previously recorded maximum sum (`ans`). If it is, we update `ans` with `ans = max(ans, s)`.
3. **Calculating the Maximum Average**:
   - Once the window has slid from the start to the end of the array, the maximum sum of any subarray with length `k` will be stored in `ans`.
   - The final step is to divide `ans` by `k` to find the maximum average value, which is returned as the result of the function: `return ans / k`.

By following this approach, the algorithm ensures that each element is added and subtracted exactly once from the sum, leading to an efficient O(n) time complexity. No extra space is needed apart from a few variables, so the space complexity is O(1).

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the array `nums = [1, 12, -5, -6, 50, 3]` and `k = 4`. We want to find the contiguous subarray of length `k` with the maximum average value.

Here's how the sliding window approach can be applied:

1. **Initialization**: First, we calculate the sum of the first `k` elements, which are `1 + 12 - 5 - 6 = 2`. We set `s` to `2`, and also `ans` is initialized to `2`.
2. **Sliding the Window**:
   - Begin the process with `i` at index `k`, or `4`, which corresponds to the first number outside of our initial window, `50`.
   - The for loop starts, i.e., `for i in range(4, 6)`.
   - At `i = 4`, we add `nums[4]` (which is `50`) to `s`, and subtract `nums[4 - 4]` (which is `1`, the leftmost element of the previous window). Now, `s = s + 50 - 1 = 51`.
   - We now check if `s` is greater than `ans`. Since `51` is greater than `2`, we update `ans` to `51`.
   - The window is now `[12, -5, -6, 50]`.
3. **Continue Sliding the Window**:
   - Increment `i` to `5`, add `nums[5]` to `s`, and subtract `nums[5 - 4]`. So `s = 51 + 3 - 12 = 42`.
   - We check if `s` is greater than the current `ans`. It is not (`42` is less than `51`), so `ans` remains `51`.
   - The window is now `[-5, -6, 50, 3]`.
4. **Calculating the Maximum Average**:
   - We have now checked all windows of size `k`. `ans` is holding the maximum sum, which is `51`.
   - The final step is to divide `ans` (which is `51`) by `k` (which is `4`) to find the maximum average. `maximum average = ans / k = 51 / 4 = 12.75`

Therefore, the subarray `[12, -5, -6, 50]` returns the maximum average of `12.75` for the given array when the subarray length is `4`. This example demonstrates that we only pass through the array once, updating our sum as we go and only carrying out constant work for each element, thereby giving us an O(n) time complexity.

Python Solution

```python
from typing import List

class Solution:
    def findMaxAverage(self, nums: List[int], k: int) -> float:
        # Initial sum of the first 'k' elements
        current_sum = sum(nums[:k])
        # Starting with the initial sum as the max sum
        max_sum = current_sum

        # Iterate over the list starting from the k-th element to the end
        for i in range(k, len(nums)):
            # Update the current sum by adding the next element and
            # subtracting the (i-k)-th element, sliding the window forward
            current_sum += nums[i] - nums[i - k]
            # Update the max sum if the current sum is greater
            max_sum = max(max_sum, current_sum)

        # Calculate the maximum average by dividing the max sum by k
        return max_sum / k
```

Java Solution

```java
class Solution {
    public double findMaxAverage(int[] nums, int k) {
        // Sum of the first 'k' elements.
        int currentSum = 0;
        for (int i = 0; i < k; ++i) {
            currentSum += nums[i];
        }

        // Initialize the max sum as the sum of the first 'k' elements.
        int maxSum = currentSum;

        // Iterate through the array starting from the k-th element.
        for (int i = k; i < nums.length; ++i) {
            // Update the current window sum by adding the new element
            // and subtracting the first element of the previous window.
            currentSum += (nums[i] - nums[i - k]);

            // Update the max sum if the current window sum is greater.
            maxSum = Math.max(maxSum, currentSum);
        }

        // Return the maximum average, which is the max sum divided by 'k',
        // converted to a double for precision.
        return maxSum * 1.0 / k;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

// Function to find the maximum average of any subarray of size k
double findMaxAverage(std::vector<int>& nums, int k) {
    int totalNumbers = nums.size(); // Total count of numbers in the input array
    int maxSum = 0; // This will hold the maximum sum of any subarray of size k
    int currentSum = 0; // This holds the sum of the current subarray size k

    // Calculate the sum of the first subarray of size k
    for (int index = 0; index < k; ++index) {
        currentSum += nums[index];
    }
    maxSum = currentSum;

    // Slide the window of size k through the array to find the max sum
    // The loop starts from k since we already computed the sum for the first k elements
    for (int index = k; index < totalNumbers; ++index) {
        // Add the next element and remove the first element of the previous window
        currentSum += nums[index] - nums[index - k];
        // Update maxSum if the new currentSum is greater than maxSum
        maxSum = std::max(maxSum, currentSum);
    }

    // The result should be the average, so divide maxSum by k
    return static_cast<double>(maxSum) / k;
}
```

Typescript Solution

```typescript
function findMaxAverage(nums: number[], k: number): number {
    let totalNumbers = nums.length;
    let maxSum = 0;  // This will hold the maximum sum of any subarray of size k
    let currentSum = 0;  // This holds the sum of the current subarray of size k

    // Calculate the sum of the first subarray of size k
    for (let index = 0; index < k; index++) {
        currentSum += nums[index];
    }
    maxSum = currentSum;

    // Slide the window of size k through the array to find the max sum
    // The loop starts from k since we already computed the sum for the first `k` elements
    for (let index = k; index < totalNumbers; index++) {
        // Add the next element and remove the first element of the previous window
        currentSum += nums[index] - nums[index - k];
        // Update maxSum if the new currentSum is greater than the existing maxSum
        maxSum = Math.max(maxSum, currentSum);
    }

    // The result should be the average, so divide by k
    return maxSum / k;
}
```

Time and Space Complexity

The time complexity of the provided code is `O(N)` where `N` is the number of elements in the input list `nums`. This is because the code iterates through the list once after calculating the initial sum of the first `k` elements. Each iteration performs a constant number of operations: one addition, one subtraction, and one comparison.

The space complexity of the code is `O(1)` because it uses a fixed amount of extra space: variables `s` and `ans` to keep the rolling sum and the maximum sum, respectively. The space used does not depend on the size of the input list `nums`.