2302. Count Subarrays With Score Less Than K

Hard | Array | Binary Search | Prefix Sum | Sliding Window

Problem Description

The problem provides us with an array of positive integers, `nums`, and a number `k`. Our task is to find the number of contiguous non-empty subarrays within `nums` where the score of the subarray is less than `k`. The score of a subarray is calculated by multiplying the sum of the elements in the subarray by the length of the subarray. A subarray is just a sequence of elements from the array that are adjacent to each other.

Intuition

To solve this problem, we need to find a way to efficiently calculate the score for all possible subarrays in `nums` and count how many of them have a score less than `k`. Doing this directly would require examining every subarray separately, which can be very inefficient for large arrays.

The intuition behind the solution is to use a sliding window approach that allows us to calculate the score of subarrays dynamically as we expand and contract the window. By keeping track of the sum of elements currently in the window, we can determine the score quickly for the current window size.

Here's the general idea:

- We start with a window at the beginning of the array.
- We expand the window to the right by adding elements until the score exceeds or equals `k`.
- When the score is equal to or greater than `k`, we shrink the window from the left by removing elements until the score is less than `k` again.
- For each window size, we count the number of valid subarrays that can be formed, which is equivalent to the number of elements we can add to the right of the current window while maintaining a score less than `k`.

This approach ensures we only calculate the score for relevant subarrays and prevents unnecessary recalculations, leading us to the solution in an efficient manner.

Solution Approach

The implementation of the solution uses a two-pointer (or sliding window) approach that keeps track of the current subarray being considered. These two pointers are denoted as `i` (the right pointer) and `j` (the left pointer), which represent the current bounds of the subarray.

Here is a step-by-step explanation of the algorithm:

1. Initialize `ans` to 0; this will count the number of valid subarrays. Also, initialize `s` to 0; this will hold the sum of the elements of the current subarray. Initialize the left pointer `j` to 0, which represents the start of the current subarray.
2. Iterate over each element `v` in `nums` using its index `i`, which acts as the end of the subarray. This loop will expand the window to the right.
   - Add the value of the current element, `v`, to `s`, which maintains the sum of the subarray from index `j` to `i`.
3. While the score of the current subarray is greater than or equal to `k` (i.e., `s * (i - j + 1) >= k`), remove elements from the start of the subarray to reduce the score.
   - Subtract `nums[j]` from `s` to reduce the sum. This corresponds to "removing" the element at the start of the subarray.
   - Increment `j` to effectively shrink the window from the left.
4. At this point, the sum multiplied by the window length is less than `k`. Therefore, for the current end of the window (`i`), we can count the number of valid subarrays that end at `i` as `i - j + 1`, since we can form a valid subarray by starting from any element between `j` and `i`.
   - Add `i - j + 1` to `ans`, which accumulates the number of valid subarrays.
5. After the iteration is complete, `ans` will hold the total number of valid subarrays, and we return this value as the final answer.

By using this algorithm, we avoid explicitly calculating the score for every possible subarray, and we efficiently count the valid subarrays by maintaining an ongoing sum and adjusting the window size. The algorithm has an overall time complexity of `O(n)` because each element is added to `s` and removed from `s` at most once, keeping the number of operations linear with the size of the input array.

Example Walkthrough

Let's walk through the provided solution with an example. Suppose our array `nums` is `[2, 1, 4, 1]` and the number `k` is `8`.

We will follow the steps described in the solution approach:

1. We initialize `ans` to 0, which will store the total count of valid subarrays, `s` to 0 for the sum of the current subarray elements, and the left pointer `j` to 0.
2. We begin iterating over `nums` with the right pointer `i`. For each element `v` in `nums`, we perform the following steps:
   - `i = 0`, `v = 2`: Add `2` to `s`. `s` becomes 2. `(s * (i - j + 1))`, which is `(2 * (0 - 0 + 1)) = 2`, is less than `8`, so we add `i - j + 1 = 0 - 0 + 1 = 1` to `ans`.
   - `i = 1`, `v = 1`: Add `1` to `s`. `s` becomes 3. `(s * (i - j + 1))`, which is `(3 * (1 - 0 + 1)) = 6`, is less than `8`, so we add `i - j + 1 = 1 - 0 + 1 = 2` to `ans`.
   - `i = 2`, `v = 4`: Add `4` to `s`. `s` becomes 7. `(s * (i - j + 1))`, which is `(7 * (2 - 0 + 1)) = 21`, is greater than `8`, so we start shrinking the window from the left:
     - We subtract `num[j]` which is `2` from `s` and increment `j`. Now `s` is `5` and `j` is `1`. The updated score is `(5 * (2 - 1 + 1)) = 10`, which is still greater than `8`.
     - We again subtract `num[j]`, now `1`, from `s` and increment `j`. Now `s` is `4` and `j` is `2`. The score is `(4 * (2 - 2 + 1)) = 4`, which is less than `8`. Now we add `i - j + 1 = 2 - 2 + 1 = 1` to `ans`.
   - `i = 3`, `v = 1`: Add `1` to `s`. `s` becomes 5. `(s * (i - j + 1))`, which is `(5 * (3 - 2 + 1)) = 10`, is again greater than `8`. We need to shrink the window:
     - We do not need to remove any elements from the window since `j` is already at `2` and score `10` is from subarray `[4, 1]`. Now since we cannot shrink the window and the score is greater than `k`, we simply move on.

At the end of the iteration, we have the total number of valid subarrays which is the value of `ans`. By adding the counts at each step, `ans = 1 + 2 + 1 = 4`. Thus, `[2]`, `[2, 1]`, `[1]` and `[1, 4]` are valid subarrays whose scores are less than `8`, and the final answer is `4`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def countSubarrays(self, nums: List[int], k: int) -> int:
        # Initialize answer, current sum, and start index of the window
        answer = current_sum = start_index = 0

        # Iterate through the array with index and value
        for end_index, value in enumerate(nums):
            # Add the current number to the current sum
            current_sum += value

            # While the product of the sum of the current subarray
            # and the length of the subarray is at least k,
            # shrink the window from the left (increase start_index)
            while current_sum * (end_index - start_index + 1) >= k:
                current_sum -= nums[start_index]
                start_index += 1

            # The number of subarrays ending with the current number
            # is the length of the current window (end_index - start_index + 1)
            answer += end_index - start_index + 1

        # Return the total number of subarrays found
        return answer
```

Java

```java
class Solution {
    public long countSubarrays(int[] nums, long k) {
        long count = 0; // To store the number of subarrays
        long sum = 0; // Sum of the elements in the current subarray
        int start = 0; // Start index for the current subarray

        // Traverse through the array starting from the 0th element
        for (int end = 0; end < nums.length; ++end) {
            sum += nums[end]; // Add the current element to sum

            // Shrink the subarray from the left if the condition is violated
            // sum * length should be less than k
            while (sum * (end - start + 1) >= k) {
                sum -= nums[start]; // Removing the element from the start of subarray
                start++; // Increment the start index
            }

            // At this point, for each element nums[end], we find how many subarrays ending at 'end' are valid
            // The number of valid subarrays is given by the difference btw current end and the new start position
            count += end - start + 1;
        }
        return count; // Return the total count of valid subarrays
    }
}
```

C++

```cpp
class Solution {
public:
    long long countSubarrays(vector<int>& nums, long long k) {
        long long count = 0; // Initialize the count of subarrays
        long long sum = 0;   // Initialize the sum of elements in the current subarray
        int start = 0;       // Start index for our sliding window

        // Iterate through each element in the array
        for (int end = 0; end < nums.size(); ++end) {
            sum += nums[end]; // Add the current element to the sum

            // If the constraint is violated (average * number of elements >= k)
            // increment start index to reduce the number of elements and sum
            while (sum * (end - start + 1) >= k) {
                sum -= nums[start++]; // Remove the element pointed by start from sum and increment start
            }

            // Count the number of subarrays ending with nums[end]. If no elements were removed in the while loop,
            // this will include all subarrays starting from nums[0..end]. If some elements were removed,
            // it will include all subarrays starting from nums[start..end].
            count += end - start + 1;
        }

        return count; // Return the total count of valid subarrays
    }
};
```

TypeScript

```typescript
function countSubarrays(nums: number[], k: number): number {
    let count = 0; // Initialize the count of subarrays
    let sum = 0;   // Initialize the sum of the elements in the current subarray
    let start = 0; // Start index for our sliding window

    // Iterate through each element in the array
    for (let end = 0; end < nums.length; end++) {
        sum += nums[end]; // Add the current element to the sum

        // While the average of the subarray multiplied by the number of elements
        // is greater than or equal to k, adjust the start index to shrink the subarray
        while (sum * (end - start + 1) >= k) {
            sum -= nums[start]; // Remove the element pointed to by start from the sum
            start++; // Increment start index to reduce the subarray size
        }

        // Count the number of subarrays ending with nums[end]. This includes all
        // subarrays starting from nums[start..end]. If the while loop above didn't
        // run, it would also include all subarrays from nums[0..end].
        count += end - start + 1;
    }

    return count; // Return the total count of valid subarrays
}
```

```python
from typing import List

class Solution:
    def countSubarrays(self, nums: List[int], k: int) -> int:
        # Initialize answer, current sum, and start index of the window
        answer = current_sum = start_index = 0

        # Iterate through the array with index and value
        for end_index, value in enumerate(nums):
            # Add the current number to the current sum
            current_sum += value

            # While the product of the sum of the current subarray
            # and the length of the subarray is at least k,
            # shrink the window from the left (increase start_index)
            while current_sum * (end_index - start_index + 1) >= k:
                current_sum -= nums[start_index]
                start_index += 1

            # The number of subarrays ending with the current number
            # is the length of the current window (end_index - start_index + 1)
            answer += end_index - start_index + 1

        # Return the total number of subarrays found
        return answer
```

Time and Space Complexity

Time Complexity

The given code uses a sliding window technique to count the number of subarrays whose elements product is less than `k`. In this code, two pointers (`i` and `j`) are used, which move forward through the array without stepping backwards. This results in each element being considered only once by each pointer, resulting in a linear traversal.

Thus, the time complexity of this algorithm is `O(n)`, where `n` is the number of elements in the array `nums`. This is because both pointers `i` and `j` can only move from the start to the end of the array once, and the operations inside the for-loop and while-loop are all constant time operations.

Space Complexity

The space complexity is determined by the extra space used aside from the input. In this case, only a fixed number of variables (`ans`, `s`, `j`, `i`, `v`) are used. These do not depend on the size of the input array. Therefore, the space complexity of the code is `O(1)`, which is constant space complexity since no additional space that grows with the input size is used.