2090. K Radius Subarray Averages

Medium Array Sliding Window Leetcode Link

Problem Description

In this problem, you are given an array `nums` containing `n` integers, indexed from `0` to `n-1`. You are also given an integer `k`, which defines a radius. The task is to calculate the *k-radius average* for each element in the array. The *k-radius average* for an element at index `i` includes all the elements from index `i - k` to `i + k`, inclusive. To qualify for calculating the average, there must be at least `k` elements before and after index `i`. If an element doesn't have enough neighbors to satisfy the radius `k`, the average for that element will be `-1`.

It is important to note that the average is computed using integer division, meaning you add all the elements within the range and then divide by the total number of elements, truncating the result to remove the decimal part.

For example, for the array `[1,3,5,7,9]` and `k = 1`, the result would be `[-1, 3, 5, 7, -1]` because:

- elements at indices `0` and `4` don't have enough neighbors for radius `1`;
- element at index `1` has an average of `(1+3+5)/3`, which truncates to `3`;
- and so on for the elements at indices `2` and `3`.

Intuition

The straightforward approach would be to calculate the average for each element individually, which would involve summing elements within the radius for every index, resulting in a time-consuming process with a time complexity of O(n*k).

However, a more efficient method is to use a moving sum (or sliding window). By maintaining a running sum of the last `2k+1` elements, we can compute the k-radius average for the current index by simply dividing this sum by `2k+1`. After computing the average, move the window by increasing the index `i`, add the next number in the array to the sum, and subtract the number that just left the window (which would be at index `i - 2k`). This way, we only need one pass through the array, resulting in a time complexity of O(n).

The provided Python function `getAverages` implements this efficient sliding window approach. An array `ans` of the same length as `nums` is initially filled with `-1`. This will store our results. The variable `s` is the sliding sum, which is updated as we iterate over `nums` with index `i`. If `i` is large enough (`i >= k * 2`), it means we can compute an average for the element at index `i - k`. We add the current value `v` to `s`, and if the window is valid, we compute the average and update `ans[i - k]`. One important detail is the use of integer division `s // (k * 2 + 1)` as required by the problem statement. Finally, we return the completed array `ans`.

Solution Approach

The solution provided uses a sliding window technique to efficiently compute each k-radius average in a single pass through the input array `nums`.

Here's an in-depth explanation of the algorithm's steps:

1. Initialize a running sum, `s`, which will keep track of the sum of elements in the current window. We will also initialize an array `ans` with the same length as `nums` filled with `-1`s, to store our results.

2. Iterate through the array `nums` with both index and value (`i` and `v`). Increment the running sum `s` by the value `v`.

3. Check if the current index `i` allows us to have a complete window of `2k+1` elements. This is determined by the condition `i >= k * 2`. If `i` is less than `k * 2`, we cannot calculate the average for `i - k` since we do not have enough elements before index `i`.

4. If we do have enough elements, calculate the average for the element at index `i - k` as the running sum `s` divided by `2k+1`. We use integer division `//` here as specified by the problem constraints. The result is stored in `ans[i - k]`.

5. Then, to maintain the size of the window, subtract the element at index `i - 2k` from the running sum `s`. This is the element that's falling out of our window as we move forward.

6. Continue this process until all elements have been visited, and the `ans` array is fully populated with the k-radius averages where possible or `-1` where the average cannot be calculated.

The algorithm makes use of simple data structures which are a running sum variable `s` and an array `ans` to hold the results. The sliding window pattern here avoids redundantly recalculating the sum for overlapping parts of the window, thus optimizing the process to a time complexity of O(n) where n is the length of the input array.

Here's the core code snippet that illustrates the algorithm:

```python
s = 0
ans = [-1] * len(nums)
for i, v in enumerate(nums):
    s += v
    if i >= k * 2:
        ans[i - k] = s // (k * 2 + 1)
        s -= nums[i - k * 2]
```

By maintaining the sliding window and updating the running sum in this manner, the algorithm efficiently computes the required averages without redundant calculations.

Example Walkthrough

Let's walk through the solution approach with a small example. Assume we have an array `nums = [2, 4, 6, 8, 10]` and `k = 2`. We want to find the k-radius average for each element.

1. **Initialize Variables**: We start by initializing the running sum `s = 0` and an array `ans = [-1, -1, -1, -1, -1]` to store the results.

2. **Iterate Through `nums`**: We begin iterating over `nums`. At the start, our running sum `s` will start accumulating values as we iterate:

   - At index `0`, `s = 0 + 2 = 2`. We cannot compute an average yet, as we don't have enough elements before this index.
   - At index `1`, `s = 2 + 4 = 6`. We still cannot compute an average for `i - k = -1`.
   - At index `2`, `s = 6 + 6 = 12`. We reach a point where we could compute an average for the first element (`i = 2`, `k = 2`), but since it's the first element, we can't.
3. **Check For Complete Window**: Now, we reach the first index where a complete window `2k+1` is available:

   - At index `3`, `s = 12 + 8 = 20`.
   - As `i = 3` is not less than `k * 2 (4)`, we can't calculate the average for `i - k = 1` yet.
4. **Calculate k-Radius Averages**:

   - At index `4`, `s = 20 + 10 = 30`. Since `i = 4 >= k * 2`, we can now calculate the average for `i - k = 2`. We compute it as `s // (2 * 2 + 1) = 30 // 5 = 6`. The running sum at this point includes `nums[0]` through `nums[4]` (values 2, 4, 6, 8, 10). We store `6` in `ans[2]`.
5. **Slide The Window**: Before we move to the next iteration (which we don't have in this example, since we've reached the end of the array), we subtract the number that's falling out of the window. Here that's `nums[4 - 2 * 2]`, which is `nums[0] = 2`. Thus, we update the running sum `s = 30 - 2 = 28`.

If we had more elements in the array, we would continue this process. However, since we've reached the end, we're done, and our result array `ans` looks like this: `[-1, -1, 6, -1, -1]`. The k-radius average is only calculated for the element at index 2 since those at indices 0, 1, 3 and 4 do not have enough neighbors.

This example illustrates the efficiency of the sliding window technique in avoiding redundant calculations. We avoid iterating over each window separately for each element, which significantly reduces the time complexity from O(nk) to O(n).

Python Solution

```python
from typing import List

class Solution:
    def getAverages(self, nums: List[int], k: int) -> List[int]:
        # Initialize the sum variable to keep running total of the elements in the sliding window.
        window_sum = 0
        # Initialize the result list to -1 for all elements as the default value.
        averages = [-1] * len(nums)

        # Loop through the list of numbers while calculating the running sum.
        for index, value in enumerate(nums):
            window_sum += value

            # Check if the window has reached the size of (k * 2 + 1).
            # This means we have enough elements to calculate the average for the middle element.
            if index >= k * 2:
                # Calculate the average for the middle element in the window and store it in the result list.
                averages[index - k] = window_sum // (k * 2 + 1)
                # Subtract the element that's moving out of the sliding window to maintain the correct window size.
                window_sum -= nums[index - k * 2]

        # Return the final list of averages.
        return averages

```

Java Solution

```java
class Solution {
    public int[] getAverages(int[] nums, int k) {
        // Get the length of the input array
        int n = nums.length;

        // Create a new array for storing prefix sums with length of n + 1
        long[] prefixSums = new long[n + 1];

        // Compute the prefix sums
        for (int i = 0; i < n; ++i) {
            prefixSums[i + 1] = prefixSums[i] + nums[i];
        }

        // Initialize the answer array with -1, which signifies positions where
        // the k-range average cannot be computed
        int[] averages = new int[n];
        Arrays.fill(averages, -1);

        // Determine the averages for the k-range for each valid position
        for (int i = 0; i < n; ++i) {
            // Check if current index i allows a full k-range on both sides
            if (i - k >= 0 && i + k < n) {
                // Calculate the sum for this k-range
                long sumForRange = prefixSums[i + k + 1] - prefixSums[i - k];
                // Calculate the average and cast it to int before storing it in the result array
                // (k << 1 | 1) calculates the size of the range, which is (2 * k + 1)
                averages[i] = (int) (sumForRange / (2 * k + 1));
            }
        }
        // Return the completed array with averages and -1 for non-computable positions
        return averages;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate the averages of all possible subarrays of size k*2+1
    std::vector<int> getAverages(std::vector<int>& nums, int k) {
        // Determine the size of the input vector
        int n = nums.size();
        // Initialize the answer vector with -1 for all elements
        std::vector<int> ans(n, -1);
        // Variable to store the sum of the elements within the window
        long sum = 0;

        // Iterate over the nums vector
        for (int i = 0; i < n; ++i) {
            // Add the current element to the sum
            sum += nums[i];

            // Check if the window has enough elements to calculate the average
            if (i >= k * 2) {
                // Calculate the average for the subarray and store it in the answer vector
                ans[i - k] = sum / (k * 2 + 1);
                // Remove the element that's no longer in the window from the sum
                sum -= nums[i - k * 2];
            }
        }
        // Return the answer vector with calculated averages and -1 for rest
        return ans;
    }
};

```

Typescript Solution

```typescript
function getAverages(nums: number[], k: number): number[] {
    // Initialize the length of the input array for easier reference
    const arrayLength = nums.length;

    // Initialize the answer array with -1 for all elements
    // -1 will be used for positions where the average cannot be calculated
    const averages: number[] = new Array(arrayLength).fill(-1);

    // A running sum of the elements in the current window
    let windowSum = 0;

    // Iterate over the array of numbers
    for (let i = 0; i < arrayLength; ++i) {
        // Add the current element to the running sum
        windowSum += nums[i];

        // Check if the window has reached the size of k elements on both sides
        if (i >= k * 2) {
            // Calculate the average for the middle element of the current window
            // The window size is k elements before, the element itself, and k elements after (total 2k + 1 elements)
            averages[i - k] = Math.floor(windowSum / (k * 2 + 1));

            // Subtract the element that is moving out of the window from the running sum
            // This is the element i - 2k where i is the current index of the traversal
            windowSum -= nums[i - k * 2];
        }
    }

    // Return the array of calculated averages and -1 for elements where it could not be calculated
    return averages;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input list `nums`. This is because there is a single loop that iterates over all elements of `nums` once. Within the loop, operations are done in constant time including calculating the sum for the average and updating the sum by subtracting the element that is falling out of the sliding window.

The space complexity of the code is `O(n)`, where `n` is the length of the input list `nums`. This is due to the `ans` list which is initialized to the same length as `nums`. There are no other data structures that depend on the size of the input that would increase the space complexity.