2393. Count Strictly Increasing Subarrays

Medium Array Math Dynamic Programming

Problem Description

You are given an array `nums` with positive integers. The task is to find out how many subarrays within this array are in strictly increasing order. A subarray is defined as a consecutive sequence of elements from the given array.

Intuition

The intuition behind the solution is to use a two-pointer approach to iterate through the array and identify all the consecutive, strictly increasing subarrays. An incrementing counter is maintained to track the length of each found increasing subarray.

As the outer loop moves through the array, the inner loop expands the current subarray window as long as the next element is greater than the previous one, indicating an increasing sequence. When the inner loop finds that the sequence is no longer increasing, it stops, and the number of subarrays that can be formed with the identified sequence is calculated.

To calculate the number of strictly increasing subarrays that fit within the length of the contiguous sequence, we use the formula for the sum of the first `n` natural numbers, which is `n(n + 1) / 2`. This formula is applied because every element in the increasing subarray can be the start of a new subarray, and the number of subarrays ending at each element increases by one for each additional element.

Once counted, the outer loop moves to the end of the previously identified increasing subarray to begin the search for the next sequence. This process continues until the end of the array is reached. The solution has linear time complexity since each element is visited only once by the inner loop.

Solution Approach

The given solution uses a simple algorithm with the help of two pointers, `i` and `j`. The Python code uses a `while` loop to iterate over the array with its index `i`. Inside this loop, another `while` loop is initiated with index `j` set to `i + 1`. Here's a breakdown of the approach:

1. Initialize an answer variable `ans` to zero. This will keep the count of strictly increasing subarrays.
2. Start the outer loop with `i` at zero, and it will run until it reaches the length of `nums`.
3. Inside the outer loop, immediately start the inner loop with `j = i + 1`.
4. Continue the inner loop as long as `j` is less than the length of `nums` and the current element `nums[j]` is greater than the previous element `nums[j - 1]` (the condition for a strictly increasing subarray).
5. If the condition satisfies inside the inner loop, increment `j` to expand the current window of the increasing subarray.
6. Calculate the count `cnt` of consecutive increasing numbers as `j - i`.
7. Use the formula `(1 + cnt) * cnt // 2` to add the number of subarrays that can be formed within the window `i` to `j - 1` to `ans`. This formula derives from the sum of the first `n` natural numbers formula `n * (n + 1) / 2`, as each element of the subarray can be the starting element for a new subarray, adding to the total count.
8. Assign `i` to `j` to move to the end of the current strictly increasing sequence and then look for the next one in the outer loop.
9. After the outer loop concludes, return the cumulative count `ans` as the result.

This approach does not require any additional data structures and relies on a simple calculation and pointer manipulation to arrive at the solution.

Example Walkthrough

Let's walk through the solution approach with a small example array `nums = [1, 2, 3, 1, 2]`.

1. Initialize `ans` to `0`. This is our counter for strictly increasing subarrays.
2. Begin with `i = 0`. The value at `nums[i]` is `1`.
3. Start the inner loop by setting `j = i + 1`, which is `1` (the second element, which has a value of `2`).
4. Now, `nums[j] > nums[j - 1]` because `2` (at `j`) is greater than `1` (at `i`). Hence, there is a strictly increasing subarray from index `0` to index `1`.
5. Continue incrementing `j`. It becomes `2`, and `nums[j]` is `3` which still satisfies `nums[j] > nums[j - 1]`.
6. Incrementing `j` again to `3`, we get `nums[j] = 1`. This is not greater than `3` (the previous value), so the inner loop stops here.
7. Calculate the count `cnt` as `j - i`, which gives `3 - 0 = 3`. We've found a strictly increasing subarray of length `3`.
8. We can form `(1 + cnt) * cnt // 2`, which is `(1 + 3) * 3 // 2 = 6` strictly increasing subarrays from index `0` to index `2`.
9. Add `6` to `ans`. Now, `ans` is `6`.
10. Move `i` to `j`, so `i = 3`. The value at `nums[i]` is `1`.
11. Repeat the process with the new value of `i`. Start the inner loop with `j = i + 1`, which is `4`. We have `nums[j] = 2`.
12. Continue as `nums[j] > nums[j - 1]` (`2 > 1`), but since `j` is now pointing to the last element, the inner loop will stop after this.
13. Calculate the count `cnt` as `1` because only one pair `i, 2` satisfies the condition of a strictly increasing subarray.
14. We can form `(1 + cnt) * cnt // 2`, which is `(1 + 1) * 1 // 2 = 1` additional strictly increasing subarray from index `3` to index `4`.
15. Add `1` to `ans`. Now, `ans` is `6 + 1 = 7`.
16. There's no more elements beyond `j = 4` to check, so we finish iteration.

At the end of this process, we have determined that there are `7` strictly increasing subarrays within `nums`.

Solution Implementation

Python

```python
class Solution:
    def countSubarrays(self, nums: List[int]) -> int:
        # Initialize the answer and the starting index
        subarray_count = start_index = 0

        # Loop over the array to find all increasing subarrays
        while start_index < len(nums):
            # Move to the next index to compare with the current element
            end_index = start_index + 1

            # Check if the next element is greater than the current
            # Continue moving end_index as long as we have an increasing subarray
            while end_index < len(nums) and nums[end_index] > nums[end_index - 1]:
                end_index += 1

            # Calculate the length of the current increasing subarray
            length = end_index - start_index

            # Compute the number of possible contiguous subarrays in the increasing subarray
            # and add to the total count
            subarray_count += (1 + length) * length // 2

            # Move start_index to the next point to start checking for a new subarray
            start_index = end_index

        # Return the total count of increasing subarrays
        return subarray_count
```

Java

```java
class Solution {
    // Method to count strictly increasing contiguous subarrays within an array.
    public long countSubarrays(int[] nums) {
        // Initialize count of subarrays as 0
        long totalSubarrays = 0;
        // Start index of the current subarray
        int startIdx = 0;
        // Total number of elements in the array
        int arraySize = nums.length;

        // Iterate over the array
        while (startIdx < arraySize) {
            // Initialize end index of the current subarray to be just after startIdx
            int endIdx = startIdx + 1;
            // Extend the end index until it no longer forms a strictly increasing subarray
            while (endIdx < arraySize && nums[endIdx] > nums[endIdx - 1]) {
                ++endIdx;
            }
            // Calculate the count of increasing subarrays that can be formed between startIdx and endIdx
            long currentSubarrayCount = endIdx - startIdx;
            // Use the formula for the sum of first n numbers to calculate combinations
            totalSubarrays += (currentSubarrayCount + 1) * currentSubarrayCount / 2;
            // Move the start index to the end index for the next iteration
            startIdx = endIdx;
        }

        // Return the total count of strictly increasing subarrays
        return totalSubarrays;
    }
}
```

C++

```cpp
class Solution {
public:
    long long countSubarrays(vector<int>& nums) {
        long long count = 0; // This will hold the total count of valid subarrays
        int start = 0, n = nums.size(); // `start` is the index to mark the beginning of a new subarray

        // Iterate over the array
        while (start < n) {
            int end = start + 1; // `end` is the index for the end of a growing subarray

            // Extend the subarray until the current element is not greater than the previous one
            while (end < n && nums[end] > nums[end - 1]) {
                ++end;
            }

            // Calculate the number of possible subarrays within the current valid range
            int length = end - start; // Length of the current subarray
            // Use the arithmetic series sum formula to count subarrays: n * (n + 1) / 2
            count += 1ll * (1 + length) * length / 2;

            // Move the start for the next potential subarray
            start = end;
        }

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

TypeScript

```typescript
function countSubarrays(nums: number[]): number {
    // Initialize the count of subarrays
    let count = 0;
    // Index 'i' is used to iterate through the array with initialized value of 0
    let i = 0;
    // 'n' stands for the length of the input array 'nums'
    const n = nums.length;

    // Loop over the elements of the array using 'i'
    while (i < n) {
        // Initialize 'j' for the inner loop as the next index of 'i'
        let j = i + 1;

        // Inner loop to find increasing consecutive subarray starting at 'i'
        while (j < n && nums[j] > nums[j - 1]) {
            ++j; // Increase 'j' if the current element is greater than the previous
        }

        // Calculate the length of the current increasing subarray
        const length = j - i;

        // Calculate the count of subarrays for this segment using the formula for sum of first 'length' natural numbers
        count += ((1 + length) * length) / 2;

        // Set 'i' to 'j' to look for the next segment
        i = j;
    }

    // Return the total count of increasing consecutive subarrays
    return count;
}
```

```python
class Solution:
    def countSubarrays(self, nums: List[int]) -> int:
        # Initialize the answer and the starting index
        subarray_count = start_index = 0

        # Loop over the array to find all increasing subarrays
        while start_index < len(nums):
            # Move to the next index to compare with the current element
            end_index = start_index + 1

            # Check if the next element is greater than the current
            # Continue moving end_index as long as we have an increasing subarray
            while end_index < len(nums) and nums[end_index] > nums[end_index - 1]:
                end_index += 1

            # Calculate the length of the current increasing subarray
            length = end_index - start_index

            # Compute the number of possible contiguous subarrays in the increasing subarray
            # and add to the total count
            subarray_count += (1 + length) * length // 2

            # Move start_index to the next point to start checking for a new subarray
            start_index = end_index

        # Return the total count of increasing subarrays
        return subarray_count
```

Time and Space Complexity

Time Complexity

The given code consists of a while loop that iterates over the length of the array `nums`. Inside this loop, there is another while loop that continues as long as the current element is greater than the previous element. This inner loop increments `j`, effectively counting the length of a monotonically increasing subarray starting at index `i`. The time complexity of this algorithm depends on the number and size of the monotonically increasing subarrays in `nums`.

In the best-case scenario, when all elements are in decreasing order, the outer loop runs `n` times (where `n` is the length of the array), and the inner loop runs only once for each element, thus giving a time complexity of `O(n)`.

In the worst-case scenario, when all elements are in increasing order, the first inner loop runs `n-1` times, the second `n-2` times, and so on. Therefore, the number of iterations would resemble the sum of the first `n-1` integers, which is `(n-1)*n/2`, resulting in a time complexity of `O(n^2)` for this scenario.

However, because each element is visited at most twice (once as the end of a previous subarray and once as the beginning of a new subarray), the actual time complexity, in general, is `O(n)`.

Space Complexity

The space complexity of the provided code is `O(1)`. This is because the code uses a constant amount of extra space for variables `ans`, `i`, and `j`, regardless of the input size. There are no additional data structures that grow with the size of the input.