2970. Count the Number of Incremovable Subarrays I

Easy Array Two Pointers Prefix Sum Enumeration

Problem Description

In this problem, you are given a 0-indexed array of positive integers named `nums`. A subarray (which is a continuous portion of the array) is considered **incremovable** if, by removing this subarray, the rest of the elements in the array form a strictly increasing sequence. An array is strictly increasing if each element is greater than the one before it. The challenge is to calculate the total number of such incremovable subarrays in the given array. An interesting note from the problem is that an empty array is also viewed as strictly increasing, implying that the boundary conditions where the subarray to remove is either at the beginning or the end of the array should also be considered strictly increasing. The aim is to find out all possible subarrays that, when removed, would leave the array strictly increasing.

Intuition

The intuition behind the solution to this problem begins with understanding what a strictly increasing array is and how removing certain subarrays can help us achieve this condition. Firstly, we note that if the array itself is already strictly increasing, then any subarray, including an empty one, is potentially incremovable. On the other hand, when the array is not strictly increasing, only particular subarrays can be removed to achieve this state. This observation leads us to the notion that we should look for the longest strictly increasing prefix and the longest strictly increasing suffix in the array. Once these are identified, there are a few key points that guide our approach to finding the solution:

1. If we have a strictly increasing prefix, any subarray starting immediately after this prefix and extending to the array's end can be removed to maintain a strictly increasing sequence.
2. If we have a strictly increasing suffix, we can similarly remove any subarray that ends immediately before this suffix and starts anywhere before the suffix (even from the array's start).
3. For cases where there are elements in between the longest prefix and suffix that prevent the entire array from being strictly increasing, we need to check how many subarrays can be removed to preserve both the strictly increasing prefix and suffix.

Combining these insights, we can traverse the array from both ends: finding the longest increasing subsequence (LIS) from the start and the longest increasing subsequence from the end (maintaining two pointers, `i` and `j`). The solution involves counting all the possible subarrays that can be removed in these scenarios. When `i` is moving from the end towards the start, `i` needs to be adjusted such that `nums[i]` is less than `nums[j]` to maintain the strictly increasing property. This way, as `j` moves and `i` is adjusted, we continue to count the number of incremovable subarrays until the suffix of the array (at position `j`) is no longer strictly increasing.

This two-pointer technique, combined with careful enumeration through the different ways the prefix and suffix can be combined, allows us to count all the potential incremovable subarrays.

Solution Approach

The solution uses a two-pointers approach to iterate through the array and identify incremovable subarrays. Here's a walkthrough of how the implementation works by diving into the algorithm, data structures, or patterns used:

1. **Initialization**: We start by initializing two pointers, `i` and `j`. Pointer `i` starts at the beginning of the array `nums` and is used to find the end of the longest strictly increasing prefix. Pointer `j` starts at the end of the array and is used to find the beginning of the longest strictly increasing suffix.
2. **Finding the Longest Increasing Prefix**: The algorithm begins by moving pointer `i` forward as long as the elements are in strictly increasing order. That is, while `nums[i] < nums[i + 1]` holds, `i` is incremented.
3. **Checking for a Fully Increasing Array**: If pointer `i` reaches the end of the array (`i == n - 1`, where `n` is the length of `nums`), this implies that the entire array is strictly increasing. The total number of subarrays in a fully increasing array of size `n` is given by the formula `n * (n + 1) / 2`. This takes into account all possible starting and ending points for a subarray, including the empty subarray, and would be returned immediately as the answer.
4. **Enumerating Subarrays with Increasing Prefix**: If the entire array is not strictly increasing, we add `i + 2` to the answer variable `ans`. This is because all subarrays that can be removed to maintain the increasing prefix, as described by situations 1 to 6 in the reference solution. It captures all the continuous segments from the end of the longest increasing prefix to the end of the array.
5. **Enumerating Subarrays with Increasing Suffix**: Next, we focus on the second pointer `j` to find all incremovable subarrays that include the longest increasing suffix. The pointer `j` is decremented as long as the suffix from `nums[j]` is strictly increasing.
6. **Adjusting Pointer `i` and Counting Subarrays**: While enumerating the suffix, for each `j`, we need to ensure that the elements to the left of `j` (the prefix) are strictly less than `nums[j]` to ensure the remaining array parts are strictly increasing. As a result, pointer `i` needs to be moved backwards (`i -= 1`) while `nums[i]` is not less than `nums[j]`.
7. **Incrementing Answer**: After adjusting `i`, we update the answer `ans` by adding `i + 2`, which counts the number of subarrays that can be removed to maintain the incremovable property considering the suffix starting at `j`.
8. **Termination Condition**: The process of moving pointer `j` and adjusting pointer `i` is repeated until either we've considered all elements in `nums` or once we encounter a part of `nums` that's not strictly increasing (i.e., when `nums[j - 1] >= nums[j]`). At this point, the loop stops, and we have counted all possible incremovable subarrays.

The final result is stored in the variable `ans`, which is then returned as the total count of incremovable subarrays in the input array `nums`.

Example Walkthrough

Let's consider a small example array `nums = [1, 2, 5, 3, 4]` to illustrate the solution approach.

1. **Initialization**: We start with two pointers `i = 0` and `j = 4` (since `nums` has 5 elements, `j` starts at index 4).
2. **Finding the Longest Increasing Prefix**: We move pointer `i` from left to right. For our example, `i` moves forward as `nums[0] < nums[1]` and `nums[1] < nums[2]`. However, `i` stops moving when `nums[2] > nums[3]` is encountered. Therefore, `i` stops at index 2 which is the element `5`.
3. **Checking for a Fully Increasing Array**: Since `i != n - 1` (`i` is not at the last index), the array is not fully increasing. We do not return `n * (n + 1) / 2`.
4. **Enumerating Subarrays with Increasing Prefix**: Our current pointer `i = 2` marks the end of the longest increasing prefix `[1, 2, 5]`. Including the subsequent subarrays starting right after this prefix leads us to `ans = i + 2 = 2 + 2 = 4`. These are the subarrays `[5, 3, 4]`, `[3, 4]`, `[4]`, and `[]` (empty array is also considered).
5. **Enumerating Subarrays with Increasing Suffix**: We begin to move `j` from right to left to find the longest increasing suffix. The pointer `j` moves from `4` to `3` as `nums[3] < nums[4]`.
6. **Adjusting Pointer `i` and Counting Subarrays**: While `j` is at `3`, we see that `nums[i] > nums[j]` (`5 > 3`). So we decrement `i` until `nums[i] < nums[j]`. In our case, `i` becomes `1` since `nums[1] < nums[3]`.
7. **Incrementing Answer**: Now that `i` is to the right place, we increment `ans` by `i + 2`, which is `1 + 2 = 3`. It accounts for the subarrays ending in `3` from the longest increasing suffix, which are `[2, 5, 3]`, `[5, 3]`, and `[3]`.
8. **Termination Condition**: We decrement `j` to `2` and try to adjust `i` again. However, `nums[2] <= nums[2]` fails to strictly increase, so we terminate the loop.

In total, the array `[1, 2, 5, 3, 4]` has `4 + 3 = 7` possible incremovable subarrays, which are `[5, 3, 4]`, `[3, 4]`, `[4]`, `[]`, `[2, 5, 3]`, `[5, 3]`, and `[3]`. Therefore, the algorithm's output for the number of incremovable subarrays would be `7`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def incremovableSubarrayCount(self, nums: List[int]) -> int:
        # Initialize the index for traversing the array from the beginning
        left_index = 0
        # Get the total length of the array 'nums'
        array_size = len(nums)

        # Find the first decreasing point in the array
        # Increment left_index as long as the next element is greater than the current
        while left_index + 1 < array_size and nums[left_index] < nums[left_index + 1]:
            left_index += 1

        # If the whole array is strictly increasing, then all subarrays are "incremovable"
        if left_index == array_size - 1:
            # Calculate the total number of subarrays using the formula for the sum of the first N natural numbers
            return array_size * (array_size + 1) // 2

        # Initialize answer with the case where we consider the longest increasing subarray from the start
        answer = left_index + 2

        # Start iterating the array from the end
        right_index = array_size - 1
        while right_index:
            # Decrease left_index until we find an element smaller than the one at right_index
            while left_index >= 0 and nums[left_index] >= nums[right_index]:
                left_index -= 1

            # Add the number of "incremovable" subarrays ending at current right_index
            answer += left_index + 2

            # Break the loop if the subarray is not strictly increasing anymore
            if nums[right_index - 1] >= nums[right_index]:
                break

            # Move the right index to the left
            right_index -= 1

        # Return the total count of "incremovable" subarrays
        return answer

# Example usage:
# solution = Solution()
# result = solution.incremovableSubarrayCount([1, 2, 3, 4])
# print(result)  # Output should be the total count of "incremovable" subarrays
```

Java

```java
class Solution {
    public int incremovableSubarrayCount(int[] nums) {
        int leftIndex = 0, n = nums.length;

        // Increment leftIndex until the current element is no longer smaller than the next
        while (leftIndex + 1 < n && nums[leftIndex] < nums[leftIndex + 1]) {
            ++leftIndex;
        }

        // If the entire array is already increasing, return the sum
        // of all subarrays possible
        if (leftIndex == n - 1) {
            return n * (n + 1) / 2;
        }

        // Otherwise, start the answer by counting subarrays including
        // nums[0] to nums[leftIndex] and one element after that
        int result = leftIndex + 2;

        // Iterate from the end of the array towards the start
        for (int rightIndex = n - 1; rightIndex > 0; --rightIndex) {
            // Move leftIndex backwards as long as it is smaller than
            // nums[rightIndex] to extend the subarray
            while (leftIndex >= 0 && nums[leftIndex] >= nums[rightIndex]) {
                --leftIndex;
            }

            // Count the subarrays which include nums[rightIndex] and all possible starts to its left
            result += leftIndex + 2;

            // If the array is not increasing from nums[rightIndex - 1] to nums[rightIndex],
            // break the loop as we cannot have incremovable subarrays to the right
            if (nums[rightIndex - 1] >= nums[rightIndex]) {
                break;
            }
        }
        return result;
    }
}
```

C++

```cpp
class Solution {
public:
    // Method to calculate the number of incrementally removable subarrays.
    int incremovableSubarrayCount(vector<int>& nums) {
        int currentIndex = 0; // Start from the first index
        int size = nums.size(); // Get the size of the array
        // Iterate until you find a non-increasing pair.
        while (currentIndex + 1 < size && nums[currentIndex] < nums[currentIndex + 1]) {
            ++currentIndex;
        }
        // If the whole array is increasing, every subarray is incrementally removable.
        if (currentIndex == size - 1) {
            // Return the total number of subarrays using the formula for the sum of first n natural numbers,
            return size * (size + 1) / 2;
        }
        // If not the whole array, include the subarrays found so far.
        int result = currentIndex + 2;
        // Iterate backwards starting from the end of the array.
        for (int backwardIndex = size - 1; backwardIndex > 0; --backwardIndex) {
            // Move currentIndex backwards until a stable element is found.
            while (currentIndex >= 0 && nums[currentIndex] >= nums[backwardIndex]) {
                --currentIndex;
            }
            // Add the number of possible subarrays ending at 'backwardIndex'.
            result += currentIndex + 2;
            // If we find a non-increasing pair while iterating backwards, break the loop.
            if (nums[backwardIndex - 1] >= nums[backwardIndex]) {
                break;
            }
        }
        // Return the total count of incrementally removable subarrays.
        return result;
    }
};
```

TypeScript

```typescript
function incremovableSubarrayCount(nums: number[]): number {
    // Calculate the length of the input array.
    const length = nums.length;

    // Initialize the index pointer to 0.
    let index = 0;

    // Find the length of the initial strictly increasing sequence.
    while (index + 1 < length && nums[index] < nums[index + 1]) {
        index++;
    }

    // If the entire array is increasing, return the sum of all lengths of subarrays possible.
    if (index === length - 1) {
        return (length + length * (length + 1)) / 2;
    }

    // Initialize answer with the case where we remove the first element.
    let answer = index + 2;

    // Iterate over the array from the end to the start.
    for (let currentIndex = length - 1; currentIndex; currentIndex--) {
        // Move the index backwards as long as nums[index] is less than nums[currentIndex].
        while (index >= 0 && nums[index] >= nums[currentIndex]) {
            --index;
        }
        // Add the number of subarrays ending at 'currentIndex'.
        answer += index + 2;

        // Check if current sequence is still increasing, if not, break out of the loop.
        if (nums[currentIndex - 1] >= nums[currentIndex]) {
            break;
        }
    }

    // Return the total count of non-removable subarrays.
    return answer;
};
```

Python

```python
from typing import List

class Solution:
    def incremovableSubarrayCount(self, nums: List[int]) -> int:
        # Initialize the index for traversing the array from the beginning
        left_index = 0
        # Determine the length of the array 'nums'
        array_size = len(nums)

        # Find the first decreasing point in the array
        # Increment left_index as long as the next element is greater than the current
        while left_index + 1 < array_size and nums[left_index] < nums[left_index + 1]:
            left_index += 1

        # If the whole array is strictly increasing, then all subarrays are "incremovable"
        if left_index == array_size - 1:
            # Calculate the total number of subarrays using the formula for the sum of the first N natural numbers
            return array_size * (array_size + 1) // 2

        # Initialize answer with the case where we consider the longest increasing subarray from the start
        answer = left_index + 2

        # Start iterating the array from the end
        right_index = array_size - 1
        while right_index:
            # Decrease left_index until we find an element smaller than the one at right_index
            while left_index >= 0 and nums[left_index] >= nums[right_index]:
                left_index -= 1

            # Add the number of "incremovable" subarrays ending at current right_index
            answer += left_index + 2

            # Break the loop if the subarray is not strictly increasing anymore
            if nums[right_index - 1] >= nums[right_index]:
                break

            # Move the right index to the left
            right_index -= 1

        # Return the total count of "incremovable" subarrays
        return answer

# Example usage:
# solution = Solution()
# result = solution.incremovableSubarrayCount([1, 2, 3, 4])
# print(result)  # Output should be the total count of "incremovable" subarrays
```

Time and Space Complexity

The time complexity of the given code is `O(n)`. This complexity arises because the code primarily consists of two while loops (not nested) that each iterate over sections of the array `nums`, where `n` is the length of `nums`. Each loop runs in a linear fashion relative to the portion of the array it traverses, hence the overall time required scales linearly with the input size `n`.

The first while loop increases `i` until it finds a non-increasing element or reaches the end of the array, and the second while loop decreases `j` while ensuring certain conditions are met. In the worst case, each will run through the entire array once, making the time complexity `O(n)`.

The space complexity of the code is `O(1)`. This is because the code uses a fixed number of integer variables to keep track of indices and the answer, which does not depend on the size of the input array, thus using a constant amount of additional space.