<https://github.com/doocs/leetcode/blob/main/solution/2300-2399/2393.Count%20Strictly%20Increasing%20Subarrays/README_EN.md>

You are given an array nums consisting of positive integers.

Return the number of **subarrays** of nums that are in strictly increasing order.

**A subarray is a contiguous part of an array.**

**Example 1:**

Input: nums = [1,3,5,4,4,6]

Output: 10

Explanation: The strictly increasing subarrays are the following:

Subarrays of length 1: [1], [3], [5], [4], [4], [6].

Subarrays of length 2: [1,3], [3,5], [4,6].

Subarrays of length 3: [1,3,5].

The total number of subarrays is 6 + 3 + 1 = 10.

**Example 2:**

Input: nums = [1,2,3,4,5]

Output: 15

Explanation: Every subarray is strictly increasing. There are 15 possible subarrays that we can take.

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^6

**Attempt 1: 2024-08-30**

**Solution 1: Math (10 min)**

public class Solution {

public long countIncreasingSubarrays(int[] arr) {

int n = arr.length;

long count = 0;

int length = 1;

for (int i = 1; i < n; i++) {

if (arr[i] > arr[i - 1]) {

length++;

} else {

count += (long) length \* (length + 1) / 2;

length = 1;

}

}

// For the last segment

count += (long) length \* (length + 1) / 2;

return count;

}

public static void main(String[] args) {

Solution solution = new Solution();

int[] arr = {1, 3, 5, 4, 7};

System.out.println(solution.countIncreasingSubarrays(arr)); // Output: 7

}

}

Time Complexity: O(N)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/2393>

**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:

Initialize an answer variable ans to zero. This will keep the count of strictly increasing subarrays.

Start the outer loop with i at zero, and it will run until it reaches the length of nums.

Inside the outer loop, immediately start the inner loop with j = i + 1.

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).

If the condition satisfies inside the inner loop, increment j to expand the current window of the increasing subarray.

Calculate the count cnt of consecutive increasing numbers as j - i.

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.

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.

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].

Initialize ans to 0. This is our counter for strictly increasing subarrays.

Begin with i = 0. The value at nums[i] is 1.

Start the inner loop by setting j = i + 1, which is 1 (the second element, which has a value of 2).

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.

Continue incrementing j. It becomes 2, and nums[j] is 3 which still satisfies nums[j] > nums[j - 1].

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.

Calculate the count cnt as j - i, which gives 3 - 0 = 3. We've found a strictly increasing subarray of length 3.

We can form (1 + cnt) \* cnt // 2, which is (1 + 3) \* 3 // 2 = 6 strictly increasing subarrays from index 0 to index 2.

Add 6 to ans. Now, ans is 6.

Move i to j, so i = 3. The value at nums[i] is 1.

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.

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.

Calculate the count cnt as 1 because only one pair 1, 2 satisfies the condition of a strictly increasing subarray.

We can form (1 + cnt) \* cnt // 2, which is (1 + 1) \* 1 // 2 = 1 additional strictly increasing subarray from index 3 to index 4.

Add 1 to ans. Now, ans is 6 + 1 = 7.

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**

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;

}

}

**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.

**Refer to chatGPT**

Leetcode problem 2393, titled "Count Strictly Increasing Subarrays," involves finding the number of strictly increasing subarrays within a given array. Here's a Java solution for this problem:

**Problem Statement:**

Given an array arr of n integers, you need to count the number of strictly increasing subarrays of arr.

**Java Solution:**

To solve this problem, we can iterate through the array and count the length of each strictly increasing subarray. For each subarray of length L, there are (L \* (L + 1)) / 2 strictly increasing subarrays that can be formed. This is because a subarray of length L has:

L subarray of length 1,

L - 1 subarrays of length 2,

L - 2 subarrays of length 3,

...,

1 subarrays of length L.

**Code Implementation:**

public class Solution {

public long countIncreasingSubarrays(int[] arr) {

int n = arr.length;

long count = 0;

int length = 1;

for (int i = 1; i < n; i++) {

if (arr[i] > arr[i - 1]) {

length++;

} else {

count += (long) length \* (length + 1) / 2;

length = 1;

}

}

// For the last segment

count += (long) length \* (length + 1) / 2;

return count;

}

public static void main(String[] args) {

Solution solution = new Solution();

int[] arr = {1, 3, 5, 4, 7};

System.out.println(solution.countIncreasingSubarrays(arr)); // Output: 7

}

}

**Explanation:**

Initialization: We initialize a variable count to keep track of the total number of strictly increasing subarrays. We also initialize length to 1 to represent the length of the current strictly increasing subarray.

Loop Through the Array: We iterate through the array starting from the second element:

**If the current element is greater than the previous element (arr[i] > arr[i - 1]), it means we are still in a strictly increasing subarray, so we increment length.**

**If not, we add the count of strictly increasing subarrays for the current length to count using the formula (length \* (length + 1)) / 2 and reset length to 1.**

Add Remaining Subarray Count: After exiting the loop, we add the count of strictly increasing subarrays for the last segment.

Return Result: Finally, we return the value of count as the result.

**Example:**

For the input array {1, 3, 5, 4, 7}, the strictly increasing subarrays are: [1], [3], [5], [1, 3], [3, 5], [4], [7], making a total of 7.

This solution efficiently counts the number of strictly increasing subarrays with a time complexity of O(n) , where n is the number of elements in the input array.

**Refer to**

[L845.Longest Mountain in Array (Ref.L821)](note://WEB927061d55e7df1331b08a8ea60797df9)