2972. Count the Number of Incremovable Subarrays II Two Pointers Binary Search

Problem Description

Array

Hard

important point to note is that an empty array is always considered strictly increasing. Here is a quick example to illustrate this: If the array is [5, 3, 4, 6, 7], then the subarray [3, 4] is "incremovable" because upon its removal, we are left with [5, 6, 7], which is strictly increasing.

You are given an array of positive integers called nums, where the array is indexed starting with 0. The challenge is to find the

number of subarrays that can be removed such that the remaining elements of nums form a strictly increasing sequence. A

subarray is a contiguous non-empty sequence of elements within an array, and you are tasked with determining all such

"incremovable" subarrays. An "incremovable" subarray is one whose removal makes the sequence strictly increasing. An

Intuition

To solve this problem, we approach it by considering different segments of the array that can independently be strictly increasing — the prefix (beginning part) and the suffix (ending part) of the array.

Initially, we find the longest strictly increasing prefix of the nums array. This prefix itself cannot be part of an "incremovable" subarray because its removal would not contribute to making the entire array strictly increasing. However, any subarray that

starts immediately after this prefix and extends to the end of the array can be removed to leave a strictly increasing sequence. Thus, we count all subarrays that start at an index greater than the last index of this prefix.

In the next step, we focus on the suffix. We look at each subarray that ends just before a strictly increasing suffix. Here, we must

ensure that for any given end index of a subarray, the element just before the subarray must be less than the start of the suffix to keep the entire sequence strictly increasing. We count such arrangements as we identify them. By carefully examining these two scenarios — looking at the suffixes and prefixes that can remain after removing a subarray we ensure that all possible "incremovable" subarrays are counted. The final count comprises all valid subarray removals that leave us with a strictly increasing sequence.

the algorithm, using the Python code provided as a reference: Initialize a pointer i to start from the beginning of the array. The first step is to increment i until it reaches the end of the longest strictly increasing prefix of nums. This is achieved by a while loop that continues as long as nums[i] is less than

The solution uses a two-pointer technique to efficiently find all "incremovable" subarrays. Here's a step-by-step break down of

If the entire array is strictly increasing (which means i is at the last index after the loop), then every subarray can be

considered "incremovable". The total number is calculated using the formula n * (n + 1) // 2 where n is the length of the

array nums. This follows from the property that there are n choose 2 plus n subarrays (including the empty subarrays at each position).

nums[i + 1].

Solution Approach

If the array is not entirely strictly increasing, the count starts with 1 + 2, which represents the number of strictly increasing subarrays that include the longest prefix identified in step 1.

A second pointer j is initialized to point to the end of the array. The intention is to decrease j until the strictly increasing suffix is broken (when nums[j - 1] is no longer less than nums[j]). For each valid position of j, move the pointer i as far back as necessary so that nums[i] is less than nums[j]. At every such

and the suffix from j onwards) will be strictly increasing. So, we add i + 2 to our count for each such arrangement.

from the array's end moving back towards the start (or until the strictly increasing property for the suffix is broken).

repeated work by using two pointers to dynamically adjust the range being considered for possible removal.

forward, meaning each element is considered at most twice.

subarrays that can be removed to leave a strictly increasing sequence.

backward as long as the strictly increasing suffix is being extended.

nums [4] = 4 is not strictly greater than nums [3] = 5.

def incremovable subarray count(self, nums: List[int]) -> int:

Find the longest increasing prefix of the array

return length * (length + 1) // 2

subarray_count += start_index + 2

start_index, length = 0, len(nums)

start_index += 1

if start index == length - 1:

subarray_count = start_index + 2

start index -= 1

increasingSequenceEnd++;

// integers starting from 1).

long long count = currentPos + 2;

--currentPos;

break;

return count;

};

TypeScript

let index = 0;

index++;

if (index === n - 1) {

let count = index + 2;

--index;

count += index + 2;

start_index += 1

end index = length - 1

break

end_index -= 1

while end index:

if start index == length - 1:

subarray_count = start_index + 2

start index -= 1

return length * (length + 1) // 2

subarray_count += start_index + 2

break;

return count;

return (n * (n + 1)) / 2;

count += currentPos + 2;

if (currentPos == n - 1) {

if (increasingSequenceEnd == numsLength - 1) {

// Loop from the end of the array to the start

for (int $j = numsLength - 1; j > 0; --j) {$

break

end_index -= 1

return subarray_count

Java

Initialize starting index and get the length of the input list

Count the subarrays including the longest increasing prefix

Add subarrays count based on the current start_index

If we find a non-increasing pair, break the loop

if nums[end_index - 1] >= nums[end_index]:

Move the end index one step to the left

Return the total count of incremovable subarrays

while start index + 1 < length and nums[start_index] < nums[start_index + 1]:</pre>

If entire array is increasing, return the sum of length choose 2 plus length

while start index >= 0 and nums[start_index] >= nums[end_index]:

step, it represents a possibility where the subarray starting at i + 1 can be removed, and what remains (the prefix up to i

Continue decrementing j and adjusting i as needed until you've considered all possible endpoints j for the suffix starting

This method efficiently finds all the subarrays that can be removed to leave a strictly increasing sequence, as it takes into

account all the increasing sequences that can be formed by removing an "incremovable" subarray. It avoids unnecessary

- This approach is particularly effective because it reduces the problem to a series of decisions about where to start and stop removing elements, which can be determined in O(n) time. The pointer i moves only backward, and the pointer j moves only
- Identifying the Longest Strictly Increasing Prefix: We set a pointer i at the start of the array and move it forward as long as nums[i] is less than nums[i + 1]. Initially, nums [0] = 2 and nums [1] = 1, which is not strictly increasing, so the pointer does not move forward.

The longest strictly increasing prefix is of length 0, indicating that there are no strictly increasing elements from the start.

Let's take a small example array nums = [2, 1, 3, 5, 4, 6] and walk through the solution approach to find all "incremovable"

∘ We don't need to use the formula n * (n + 1) // 2 here, because we found that the first two elements don't form a strictly increasing sequence. • The count starts at i + 2, which in this case, is 0 + 2 = 2. Identifying the Longest Strictly Increasing Suffix: We create a second pointer j at the end of the array and move it

• Starting at j = 5 (last index), since nums[5] = 6 is strictly greater than nums[4] = 4, we decrement j. But we stop at j = 4 because

Checking if the Whole Array is Strictly Increasing: Since our pointer i did not reach the last index, the array is not entirely

Counting the Possibilities for Removal: For each valid position of j in the strictly increasing suffix, we move i backwards to

endpoint.

add i + 2 = 4 to our count again.

that can be removed.

Solution Implementation

Example Walkthrough

strictly increasing.

satisfy nums[i] < nums[j].</pre> \circ At j = 4, we find that i = 2, and nums[2] = 3 < nums[4] = 4. So, subarray nums[3:4] (which is [5]) can be removed. Here, i + 2 = 2 + 12 = 4, so we add 4 to our count.

Repeat for Each Valid j Position: We continue to decrement j and adjust i for each potential "incremovable" subarray

• Next, we check j = 3 (since nums[3] = 5 is strictly greater than nums[2] = 3). We cannot move i back any further from i = 2, so we

Aggregate Count and Return: We add up our counts from steps 3 to 5 to get the total number of "incremovable" subarrays

 \circ Our count is now 2 (base count from step 2) + 4 (from j = 4) + 4 (from j = 3) = 10. Thus, by using the solution approach, we can efficiently determine that there are 10 "incremovable" subarrays for the array nums

= [2, 1, 3, 5, 4, 6]. Each "incremovable" subarray removal would leave a remaining sequence that is strictly increasing.

- **Python** class Solution:
 - # Start from the end and look for potential non-increasing elements end index = length - 1while end index: # Decrease start index till nums[start index] is less than nums[end_index]
- class Solution { public long incremovableSubarrayCount(int[] nums) { int increasingSequenceEnd = 0; // Initialize a pointer to track the end of the initial increasing sequence int numsLength = nums.length; // Assign the length of the array to a variable for easy reference

// If the entire array is strictly increasing, return the sum of all possible

return static_cast<long long>(n) * (n + 1) / 2;

// Start counting the strictly increasing subarrays

// from the back and count all the possible subarrays.

for (int reversePos = n - 1; reversePos > 0; --reversePos) {

// Stop if the next sequence is not strictly increasing.

if (nums[reversePos - 1] >= nums[reversePos]) {

function incremovableSubarrayCount(nums: number[]): number {

// Find the length of the initial increasing subarray

while (index + 1 < n && nums[index] < nums[index + 1]) {</pre>

const n = nums.length; // Number of elements in the array

// subarray lengths (which follows the formula n * (n + 1) / 2 for consecutive

// Traverse the array from the end, looking for strictly increasing sequences

while (currentPos >= 0 && nums[currentPos] >= nums[reversePos]) {

// Add possible subarrays starting at positions up to currentPos.

// Function to count the number of incrementally removable subarrays in a given array

// If the whole array is increasing, return the sum of all subarray counts

// Initial count including subarrays ending at the first non—increasing element

// Iterate from the end of the array to find all other incremovable subarrays

// If we find a non-increasing pair from the end, we stop the process

while start index + 1 < length and nums[start_index] < nums[start_index + 1]:</pre>

If entire array is increasing, return the sum of length choose 2 plus length

Decrease start index till nums[start index] is less than nums[end_index]

Count the subarrays including the longest increasing prefix

Add subarrays count based on the current start_index

If we find a non-increasing pair, break the loop

if nums[end_index - 1] >= nums[end_index]:

Move the end index one step to the left

Return the total count of incremovable subarrays

Start from the end and look for potential non-increasing elements

while start index >= 0 and nums[start_index] >= nums[end_index]:

for (let reverseIndex = n - 1; reverseIndex > 0; --reverseIndex) {

while (index >= 0 && nums[index] >= nums[reverseIndex]) {

// Increment count by the number of incremovable subarrays

// which can be created ending with nums[reverseIndex]

if (nums[reverseIndex - 1] >= nums[reverseIndex]) {

// Return the total count of incremovable subarrays

// than the value at reverseIndex to maintain increasing order

// Decrease index while the current value is not less

// Decrement currentPos until a valid subarray (strictly increasing) emerges.

// Loop to find the end of the initially increasing sequence

```
// Decrease increasingSequenceEnd pointer until the value at it is less than the value at index 'j'.
            while (increasingSequenceEnd >= 0 && nums[increasingSequenceEnd] >= nums[j]) {
                --increasingSequenceEnd;
            // Update the answer with the possible subarray counts calculated using the current index 'j'.
            answer += increasingSequenceEnd + 2;
            // If the previous number is not less than the current, there's no point in continuing.
            if (nums[i - 1] >= nums[j]) {
                break;
        return answer; // Return the total count of incremovable subarrays.
C++
class Solution {
public:
    // Function to count the number of increasing subarrays that can be formed such that
    // any subarray can only be removed as a whole if it is strictly increasing.
    long long incremovableSubarrayCount(vector<int>& nums) {
        int currentPos = 0:
        int n = nums.size();
        // Finding the initial strictly increasing sequence
        while (currentPos + 1 < n && nums[currentPos] < nums[currentPos + 1]) {</pre>
            ++currentPos;
```

while (increasing Sequence End + 1 < nums Length && nums[increasing Sequence End] < nums[increasing Sequence End + 1]) {

return numsLength * (numsLength + 1L) / 2; // n(n+1)/2 is the formula for the sum of all natural numbers up to n.

long answer = increasing Sequence End + 2; // Initialize the answer and start counting from increasing Sequence End + 2

// If the entire array is an increasing sequence, return the count of all possible subarrays.

class Solution: def incremovable subarray count(self. nums: List[int]) -> int: # Initialize starting index and get the length of the input list start_index, length = 0, len(nums) # Find the longest increasing prefix of the array

return subarray_count Time and Space Complexity

The time complexity of the given code is O(n). Here's why: • The first while loop increments i until the sequence is no longer strictly increasing. In the worst case scenario, this will take 0(n) time if nums

operation.

The space complexity of the code is 0(1). This is because:

• The second while loop (starting with while j:) potentially iterates from the end of the array down to the beginning. Each nested while loop

• The if condition checks if the entire array is increasing and, if so, returns a calculation based on the length of nums, which is an 0(1)

- (under while j:) will reduce i until a suitable condition is met. Together, both loops will at most go through the array once in reverse, leading again to O(n) time.
- Combining both loops, our time complexity remains O(n) since both loops can't exceed n iterations in total for the length of the

Space Complexity

array.

Time Complexity

is an increasing array.

No additional data structures that grow with input size are used.

• The variables i, n, ans, and j use a constant amount of space that does not depend on the size of the input nums.

• Thus, the space used remains constant irrespective of the length of nums.