## 581. Shortest Unsorted Continuous Subarray

### **Problem Description**

**Greedy** Array

subarray) of this array that, if we sort it, the entire array becomes sorted in non-decreasing order (i.e., ascending order with duplicates allowed). The goal is to find the length of this smallest segment. We are looking for the subarray where once sorted, the values before and after this selected part fit in order and don't disrupt the

In this LeetCode problem, we are given an array of integers (nums). The task is to identify the smallest segment (continuous

Two Pointers Sorting Monotonic Stack

non-decreasing sequence. To solve this problem, we have to distinguish between the parts of the array which are already in the correct position and those that are not.

### There are different strategies that we could use to solve this problem, some being more optimal than others. The initial, more

Intuition

Medium

straightforward approach is to sort a copy of the array and to compare it to the original. The first and last positions where they differ will give us the boundaries of the subarray that needs to be sorted. However, the actual implemented solution follows a more optimal and intuitive approach that avoids using extra space and reduces the time complexity. The solution involves iterating over the array while keeping track of the maximum and minimum

values seen from the left and the right, respectively. While scanning from the left, we maintain the maximum value encountered so far - let's call it mx. If at any index i, the value x is less than mx, this suggests x is out of its required sorted position because everything before it is already part of a non-decreasing

sequence. Hence, x should be included in the subarray that needs sorting. The rightmost index r where this occurs marks the endpoint of the unsorted subarray.

Simultaneously, we scan from the right and maintain the minimum value encountered so far - referenced as mi. Upon finding an element greater than mi at index n - i - 1 from the end, it indicates this element is also out of order. The leftmost index l where this situation arises becomes the beginning of the unsorted subarray. Initially, 1 and r are set to -1, and if they remain unchanged after the iterations, it means the entire array is already sorted, and we

difference r - l + 1. The intuition behind updating r and 1 only when we find values out of their required position is that only these values define the boundaries of the shortest unsorted subarray—and only by sorting them can we achieve a fully sorted array.

return length 0. If they have been updated, the length of the minimum subarray required to be sorted is calculated by the

The solution involves a single pass approach that cleverly makes use of two properties, namely tracking the maximum and minimum values from the left and right, respectively. By doing this, we can identify the bounds of the unsorted subarray without

# Let's understand this algorithm step by step:

Solution Approach

Initialize Variables: Two pointers 1 and r are initialized to -1 which will represent the left and right bounds of the unsorted subarray. We also initialize two variables, mi to +infinity, representing the minimum value seen from the right; and mx to infinity, representing the maximum value seen from the left.

Iterate Over the Array: We loop over all elements in the array using a variable i that goes from ∅ to n − 1, where n is the

length of the array. While iterating, we perform the following: • Update Right Boundary (r): For the i-th element x, we compare it with the maximum value seen so far mx. If x is smaller than mx, it means x

having to sort the array or use extra space.

- is out of place and should be part of the subarray to be sorted, so we update r to be the current index i. • Update Maximum (mx): If x is greater than or equal to mx, we update mx to be x because it's the new maximum value seen so far.  $\circ$  At each iteration of i, we also consider the mirror position from the right end of the array: index n - i - 1. ○ Update Left Boundary (1): We compare nums [n - i - 1] with the minimum value seen from the right mi. If nums [n - i - 1] is bigger than
- mi, it's out of place and should be included in the subarray to be sorted, so we update 1 to be n i 1. ∘ Update Minimum (mi): If nums [n - i - 1] is less than or equal to mi, we update mi to this value because it's now the minimum seen from the right.
  - means the array was already sorted, and we return 0. Otherwise, we return the length of the subarray that needs to be sorted, which is r - l + 1.

By iterating from both ends towards the center and updating 1 and r on-the-go, we are leveraging the information of where the

decreasing order up to that point, thus efficiently finding the shortest unsorted window in one single pass with a time complexity

sequence breaks from being non-decreasing. mx and mi act as sentinels to detect any number that doesn't fit into the non-

Calculate the Subarray Length: After iterating through the array, we check if r was updated. If r is not updated (r == -1), it

of O(n) and space complexity of O(1). **Example Walkthrough** 

**Step 1**: Initialize Variables • l = -1• r = -1• mi = +infinity mx = -infinity

Let's walk through a small example using the described solution approach. Consider the array nums = [2, 6, 4, 8, 10, 9, 15].

### We start iterating from the first element to the last. For each element, we perform the checks and updates as specified.

**Step 2**: Iterate Over the Array

Let's see this in action:

```
• mx = max(-infinity, 2) = 2.
• The mirror index from the right is n - 0 - 1 = 6. Element nums [6] = 15.
```

```
• Element 6 is not less than mx (2). Thus, mx becomes 6.
• The mirror index from the right is n - 1 - 1 = 5. Element nums [5] = 9.
• mi becomes min(15, 9) = 9.
```

ii. Moving to i = 1, nums [i] = 6.

• mi = min(+infinity, 15) = 15.

i. When i = 0, element nums [i] = 2.

- iii. Next, i = 2, nums [i] = 4. • 4 is less than mx (6), hence we update r = 2.

## • The mirror index from the right is n - 2 - 1 = 4. Element nums [4] = 10.

• mi is now the min(9, 10) = 9.

iv. For i = 3, nums [i] = 8. • 8 is not less than mx (6). So, mx becomes 8.

• The mirror index is n - 3 - 1 = 3. Element nums [3] = 8.

• The mirror index is n - 4 - 1 = 2. Element nums [2] = 4.

 mi stays at 9 since 8 is less than mi. v. When i = 4, nums [i] = 10.

## • 4 is less than mi (9), so we update 1 = 2.

descending sequence.

from typing import List

**Python** 

• 10 is not less than mx. Update mx to 10.

**Step 3**: Calculate the Subarray Length

 We need to find the full range of the unsorted segment. We already know r should be at least 5, because nums [5] = 9 was out of place. • Correcting for the left boundary due to the two-pointer check in the algorithm, 1 should be 1, not 2. The element nums [1] = 6 initiated the

After completing the iteration, 1 is found to be 2 and r is also 2.

• Thus, the length of the subarray is r - l + 1 = 5 - 1 + 1 = 5.

sorted. The length of this segment is 5. **Solution Implementation** 

# Initializing the left and right boundaries of the subarray

# Update the maximum value seen so far

# Update the minimum value seen so far

# Otherwise return the length of the unsorted subarray

minimum = nums[length - i - 1]

# Iterate over the list to find the right boundary of the subarray

# If the right boundary is still -1, the array is already sorted, hence return 0

return 0 if right\_boundary == −1 else right\_boundary - left\_boundary + 1

left\_boundary, right\_boundary = -1, -1

# Length of the input list

for i in range(length):

if maximum > nums[i]:

right\_boundary = i

maximum = nums[i]

for (int i = 0; i < n; ++i) {

right = i;

} else {

if (maxVal > nums[i]) {

length = len(nums)

greater than all seen max values and less than all seen min values, respectively.

• They both referred to the same element because we performed the 1 update after the r update.

class Solution: def findUnsortedSubarray(self, nums: List[int]) -> int: # Initializing minimum and maximum values found while traversing minimum, maximum = float('inf'), float('-inf')

# Found a new right boundary if current number is less than the maximum seen so far

The subarray [6, 4, 8, 10, 9] from indices 1 to 5 is the smallest segment that, once sorted, the entire array nums becomes

As we continue, mx and mi will not update any further because the remaining elements (nums [5] = 9 and nums [6] = 15) are

#### # Checking the left boundary using the mirrored index (from the end) if minimum < nums[length - i - 1]:</pre> # Found a new left boundary if current number is more than the minimum seen so far $left_boundary = length - i - 1$ else:

Java

class Solution {

else:

```
public int findUnsortedSubarray(int[] nums) {
       // Initialize variables
        final int MAX_INT = Integer.MAX_VALUE; // Use MAX_VALUE constant for clarity
        int n = nums.length; // Length of the input array
        int leftIndex = -1, rightIndex = -1; // Track the left and right boundaries of the subarray
        int minUnsorted = MAX_INT, maxUnsorted = Integer.MIN_VALUE; // Set initial values for min and max of the unsorted subarra
       // Loop through the array to find the unsorted subarray's boundaries
        for (int i = 0; i < n; ++i) {
           // If current max is greater than the current element, it might belong to the unsorted part
           if (maxUnsorted > nums[i]) {
                rightIndex = i; // Update the right boundary
            } else {
                maxUnsorted = nums[i]; // Update the current max for the sorted part
            // Simultaneously, check the unsorted subarray from the end of the array
            if (minUnsorted < nums[n - i - 1]) {</pre>
                leftIndex = n - i - 1; // Update the left boundary
            } else {
                minUnsorted = nums[n - i - 1]; // Update the current min for the sorted part
       // If rightIndex is not updated, the array is fully sorted, return 0
       // Otherwise, return the length of the subarray that must be sorted
        return rightIndex == -1 ? 0 : rightIndex - leftIndex + 1;
C++
#include <vector>
#include <algorithm>
using namespace std;
class Solution {
public:
    int findUnsortedSubarray(vector<int>& nums) {
        const int INF = 1e9; // Use uppercase for constant values
        int n = nums.size(); // Size of the input vector
        int left = -1, right = -1; // Initialize the left and right pointers of the subarray
```

int minVal = INF, maxVal = -INF; // Initialize the minimum and maximum values seen so far

// Iterate forward through the vector to find the right boundary of the unsorted subarray

```
maxVal = nums[i];
            // Iterate backward through the vector to find the left boundary of the unsorted subarray
            if (minVal < nums[n - i - 1]) {
                left = n - i - 1;
            } else {
                minVal = nums[n - i - 1];
        // If right is -1, the array is already sorted; otherwise, calculate the length of the unsorted subarray
        return right == -1 ? 0 : right - left + 1;
};
TypeScript
/**
* Finds the length of the smallest contiguous subarray that, if sorted, results in the whole array being sorted.
* @param nums Array of numbers to be checked.
* @return The length of such subarray.
*/
function findUnsortedSubarray(nums: number[]): number {
    let left = -1;
    let right = -1;
    let minOutOfOrder = Infinity;
    let maxOutOfOrder = -Infinity;
    const length = nums.length;
    // Traverse the array to find the boundaries where the order is incorrect.
    for (let i = 0; i < length; ++i) {</pre>
       // From the left, identify the rightmost index that is out of order.
        if (maxOutOfOrder > nums[i]) {
            right = i;
       } else {
            maxOutOfOrder = nums[i];
       // From the right, identify the leftmost index that is out of order.
        if (minOutOfOrder < nums[length - i - 1]) {</pre>
            left = length -i - 1;
        } else {
            minOutOfOrder = nums[length - i - 1];
```

// If no out-of-order elements are found, the array is already sorted (return 0).

// Otherwise, return the length of the unsorted subarray.

# Initializing minimum and maximum values found while traversing

# Iterate over the list to find the right boundary of the subarray

# Checking the left boundary using the mirrored index (from the end)

return 0 if right\_boundary == -1 else right\_boundary - left\_boundary + 1

# Found a new right boundary if current number is less than the maximum seen so far

# Found a new left boundary if current number is more than the minimum seen so far

# Initializing the left and right boundaries of the subarray

# Update the maximum value seen so far

# Otherwise return the length of the unsorted subarray

return right === -1 ? 0 : right - left + 1;

def findUnsortedSubarray(self, nums: List[int]) -> int:

minimum, maximum = float('inf'), float('-inf')

left\_boundary, right\_boundary = -1, -1

**# Length of the input list** 

for i in range(length):

if maximum > nums[i]:

right\_boundary = i

maximum = nums[i]

if minimum < nums[length - i - 1]:</pre>

length = len(nums)

else:

from typing import List

class Solution:

```
left_boundary = length - i - 1
   else:
        # Update the minimum value seen so far
        minimum = nums[length - i - 1]
# If the right boundary is still -1, the array is already sorted, hence return 0
```

Time and Space Complexity

The time complexity of the given code is O(n). This is because the code contains only one for loop that iterates over all elements of the array, nums, exactly once. Inside the loop, it performs a constant number of operations for each element, which does not depend on the size of the array.

The space complexity of the code is 0(1) which means constant space. No additional space is utilized that is dependent on the input size n, besides a fixed number of individual variables (mi, mx, l, r, and n) that occupy space which does not scale with the size of the input array.