1574. Shortest Subarray to be Removed to Make Array Sorted

Medium Stack Array Two Pointers Binary Search Monotonic Stack Leetcode Link

Problem Description

The goal is to find the minimum length of a subarray that, when removed from an array `arr`, leaves the remaining elements in a non-decreasing order (i.e., each element is less than or equal to the next). In other words, after removing the subarray, the resulting array should be sorted in non-decreasing order. This problem also includes the possibility of not removing any subarray at all if `arr` is already non-decreasing. The term "subarray" refers to a sequence of elements that are contiguous within the `arr`.

Intuition

The key to solving this problem lies in identifying parts of the array that are already sorted in non-decreasing order. Once we've identified such parts, we can find the minimum subarray to be removed. The solution approach can be broken down into the following steps:

1. Find the longest non-decreasing subarray from the start (`left` sorted subarray).
2. Find the longest non-decreasing subarray from the end (`right` sorted subarray).
3. Evaluate if the entire array is already non-decreasing by checking if the `left` and `right` overlap or touch each other. If they do, the shortest subarray to remove would be of length 0, which means we don't have to remove anything.
4. If a removal is needed, we can consider two potential solutions:
   - Remove the elements from the end of the `left` sorted subarray to the beginning of the array, leaving only the `right` sorted subarray.
   - Remove the elements from the start of the `right` sorted subarray to the end of the array, leaving only the `left` sorted subarray.
5. It's possible that by combining some portion of the `left` subarray with some portion of the `right` subarray, we could actually remove a shorter subarray in between and still maintain the non-decreasing order. Therefore, we iterate through the `left` sorted subarray and try to match its end with the beginning of the `right` sorted subarray, minimizing the length of the subarray to be removed.

Following these steps, we can determine the shortest subarray to remove, ensuring the array remains sorted in non-decreasing order after the removal.

Solution Approach

The solution approach consists of several key steps that use loops and variables to track the progress through the array `arr`. Here's how the implementation works:

1. Initialize two pointers, `i` at the beginning of the array and `j` at the end. These pointers are used to find the `left` and `right` sorted subarrays, respectively.
2. Progress `i` forward through the array until we find the first element that is not in non-decreasing order. Until that point, the elements rest in a sorted subarray from the start.
3. Similarly, move `j` backwards through the array to find the first element from the end that breaks the non-decreasing order. Until that point, the elements are in a sorted subarray from the end.
4. If `i` has passed `j`, return 0, as the entire array is already non-decreasing or it has only one element that is out of order, which can be removed by itself.
5. Compute the initial potential answers:
   - The length of a subarray from `i` to the end of the array: `n - i - 1`
   - The length of a subarray from the start of the array to `j`: `j`

   We are interested in the minimal length of the subarray to be removed, so we take the minimum of these two potential answers.
6. Then comes the crucial step: trying to find the shortest subarray for removal that possibly lies between the sorted subarrays identified in steps 2 and 3. Initialize a new pointer `r` (short for right) to `j`.
7. Now, iterate through the array `arr` using the `left` pointer from 0 to `i` (inclusive). For each position of the `left` pointer, progress the `right` pointer until `arr[r]` is not less than `arr[l]`, ensuring that elements to the left and right are in non-decreasing order.
8. Update answer `ans` each time to reflect the minimal value: the current `ans` and the number of elements between the `left` and `right` pointers, denoted by `r - l - 1`.

By following these steps, the function concludes by returning `ans`, which represents the length of the shortest subarray to remove to achieve a non-decreasing array after its removal.

This implementation is efficient and makes clever use of two-pointer technique along with a simple `for` loop and while loop constructs to keep track of the non-decreasing subarrays from both the start and end of the input array and to calculate the minimum length of the subarray that needs to be removed.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider the array `arr = [1, 3, 2, 3, 5]`.

1. Starting from the left, we see that `1 <= 3`, but `3 > 2`, so the longest non-decreasing subarray from the start is `[1, 3]` with `i = 1`.
2. Starting from the right, we see that `5 >= 3`, `3 >= 2`, but `2 < 3`, so the longest non-decreasing subarray from the end is `[2, 3, 5]` with `j = 2`.
3. Since `i < j`, they do not overlap, and we must remove a subarray to make the entire array non-decreasing.
4. If we remove elements starting from the end of the `left` sorted subarray to the beginning of the array, we would remove `[1, 3]`, leaving `[2, 3, 5]`, which is in non-decreasing order. However, this results in removing 2 elements.
5. Conversely, if we remove elements from the start of the `right` sorted subarray to the end of the array, we would remove `[3, 5]`, leaving `[1, 3, 2]`, which is not in non-decreasing order, so this is not a valid option.
6. Now, we check to see if it's possible to maintain a part of the `left` subarray `[1, 3]` and combine it with the `right` subarray `[2, 3, 5]` to minimize the length of the subarray to be removed. To find the shortest subarray for removal, initialize pointer `r` (short for right) to `j`, which is 2 at the moment.
7. We iterate through the array from the left pointer `l = 0` to `i = 1`. When `l = 0`, `arr[l] = 1` is less than `arr[r] = 2` (since `r` is at `j`), so we don't need to move `r`. Next, when `l = 1`, `arr[l] = 3` is greater than `arr[r] = 2`, so we increment `r` to ensure that `arr[r]` is not less than `arr[l]`. Since `arr[r] = 3` is now greater than `arr[l] = 3`, we can stop.
8. The minimal length of the subarray to be removed lies in between pointer `l` and pointer `r`, which in this case is the subarray `[2]` (since `r = 3` and `l = 1`, we have `r - l - 1 = 3 - 1 - 1 = 1` element to be removed).

Thus, by following the solution steps, the smallest subarray we need to remove to make `arr` sorted in non-decreasing order is `[2]` of length 1. Hence, the function returns 1 as the answer.

Python Solution

```python
from typing import List

class Solution:
    def findLengthOfShortestSubarray(self, arr: List[int]) -> int:
        # Length of the array
        length = len(arr)

        # Initialize two pointers for the beginning and end of the array
        left = 0
        right = length - 1

        # Move the left pointer to the right as long as the subarray is non-decreasing
        while left + 1 < length and arr[left] <= arr[left + 1]:
            left += 1

        # Move the right pointer to the left as long as the subarray is non-decreasing
        while right - 1 >= 0 and arr[right - 1] <= arr[right]:
            right -= 1

        # If the whole array is already non-decreasing, return 0
        if left >= right:
            return 0

        # Calculate the length of the remaining array to be removed
        min_length_to_remove = min(length - left - 1, right)

        # Reinitialize the right pointer for the next loop
        new_right = right

        # Check for the shortest subarray from the left side to the midpoint
        for new_left in range(left + 1):
            # Increment the right pointer until the elements on both sides are non-decreasing
            while new_right < length and arr[new_right] < arr[new_left]:
                new_right += 1
            # Update the minimum length if a shorter subarray is found
            min_length_to_remove = min(min_length_to_remove, new_right - new_left - 1)

        # Return the minimum length of the subarray to remove to make array non-decreasing
        return min_length_to_remove

```

Java Solution

```java
class Solution {
    public int findLengthOfShortestSubarray(int[] arr) {
        int n = arr.length;
        // Find the length of the non-decreasing starting subarray.
        int left = 0, right = n - 1;
        while (left + 1 < n && arr[left] <= arr[left + 1]) {
            left++;
        }
        // If the whole array is already non-decreasing, return 0.
        if (left == n - 1) {
            return 0;
        }

        // Find the length of the non-decreasing ending subarray.
        while (right > 0 && arr[right - 1] <= arr[right]) {
            right--;
        }

        // Compute the length of the subarray to be removed,
        // considering only one side (either starting or ending subarray).
        int minLengthToRemove = Math.min(n - left - 1, right);

        // Try to connect a prefix of the starting non-decreasing subarray
        // with a suffix of the ending non-decreasing subarray.
        for (int leftIdx = 0, rightIdx = right; leftIdx <= left; leftIdx++) {
            // Move the rightIdx pointer to the right until we find an element
            // that is not less than the current element from the left side.
            while (rightIdx < n && arr[rightIdx] < arr[leftIdx]) {
                rightIdx++;
            }
            // Update the answer with the minimum length found so far.
            minLengthToRemove = Math.min(minLengthToRemove, rightIdx - leftIdx - 1);
        }
        return minLengthToRemove;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    int findLengthOfShortestSubarray(std::vector<int>& arr) {
        int n = arr.size(); // The size of the input array
        int left = 0, right = n - 1; // Pointers to iterate through the array

        // Expand the left pointer as long as the current element is smaller or equal than the next one
        // This means the left part is non-decreasing
        while (left + 1 < n && arr[left] <= arr[left + 1]) {
            ++left;
        }

        // If the whole array is non-decreasing, no removal is needed
        if (left == n - 1) {
            return 0;
        }

        // Expand the right pointer inwards as long as the next element leftwards is smaller or equal
        // This means the right part is non-decreasing
        while (right > 0 && arr[right - 1] <= arr[right]) {
            --right;
        }

        // Calculate the initial length of the subarray that we can potentially remove
        int minSubarrayLength = std::min(n - left - 1, right);

        // Attempt to merge the non-decreasing parts on the left and the right
        for (int l = 0, r = right; l <= left; ++l) {
            // Find the first element which is not less than arr[l] in the right part to merge
            while (r < n && arr[r] < arr[l]) {
                ++r;
            }
            // Update the answer with the minimum length after merging
            minSubarrayLength = std::min(minSubarrayLength, r - l - 1);
        }

        // Return the answer which is the length of the shortest subarray to remove
        return minSubarrayLength;
    }
};

```

Typescript Solution

```typescript
function findLengthOfShortestSubarray(arr: number[]): number {
    const n: number = arr.length; // The size of the input array
    let left: number = 0; // Pointer to iterate from the start
    let right: number = n - 1; // Pointer to iterate from the end

    // Expand the left pointer as long as the current element is smaller than or equal to the next one
    // This means the left part is non-decreasing
    while (left + 1 < n && arr[left] <= arr[left + 1]) {
        left++;
    }

    // If the whole array is non-decreasing, no removal is needed
    if (left === n - 1) {
        return 0;
    }

    // Expand the right pointer inward as long as the next element to the left is smaller than or equal
    // This means the right part is non-decreasing
    while (right > 0 && arr[right - 1] <= arr[right]) {
        right--;
    }

    // Calculate the initial length of the subarray that we can potentially remove
    let minSubarrayLength: number = Math.min(n - left - 1, right);

    // Attempt to merge the non-decreasing parts on the left and the right
    for (let l: number = 0, r: number = right; l <= left; l++) {
        // Find the first element which is not less than arr[l] in the right part to merge
        while (r < n && arr[r] < arr[l]) {
            r++;
        }
        // Update the minimum length after merging
        minSubarrayLength = Math.min(minSubarrayLength, r - l - 1);
    }

    // Return the minimum length, which is the length of the shortest subarray to remove
    return minSubarrayLength;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be broken down as follows:

1. Two while loops (before the `if` statement) are executed sequentially, each advancing at most `n` steps. The worst-case complexity for this part is O(`n`).
2. The `if` statement is a constant time check O(1).
3. The minimum of `n - i - 1` and `j` is also a constant time operation O(1).
4. A for loop runs from `0` to `i + 1`, and inside it, there is a while loop that could iterate from `j` to `n` in the worst case. In the worst-case scenario, this nested loop could run O(`n`^2) times because for each iteration of the for loop (at most `n` times), the while loop could also iterate `n` times.

Thus, the overall time complexity is dominated by the nested loop, giving us a worst-case time complexity of O(`n`^2).

Space Complexity

The space complexity is determined by the extra space used by the algorithm besides the input. In this case:

1. Variables `i`, `j`, `n`, `ans`, and `r` use constant space O(1).
2. There are no additional data structures used that grow with the size of the input.

Therefore, the space complexity is O(1), which corresponds to constant space usage.