845. Longest Mountain in Array

Medium Array Two Pointers Dynamic Programming Enumeration Leetcode Link

Problem Description

The problem presents the definition of a "mountain array." An array can be considered a mountain array if it satisfies two conditions:

1. The length of the array is at least 3.
2. There exists an index `i` (0-indexed), which is not at the boundaries of the array (meaning `0 < i < arr.length - 1`), where the elements strictly increase from the start of the array to the index `i`, and then strictly decrease from `i` until the end of the array. In other words, there is a peak element at index `i` with the elements on the left being strictly increasing and the elements on the right being strictly decreasing.

The objective is to find the longest subarray within a given integer array `arr`, which is a mountain, and to return its length. If no such subarray exists, the function should return `0`.

Intuition

To solve this problem, we can apply a two-pointer technique. The idea is to traverse the array while maintaining two pointers, `l` (left) and `r` (right), that will try to identify the bounds of potential mountain subarrays.

We initialize an answer variable, `ans`, to keep track of the maximum length found so far.

Here is a step by step breakdown of the process:

1. Start with a left pointer `l` at the beginning of the array. The right pointer `r` initially points to the element next to `l`.
2. Identify if there's an increasing sequence starting from `l`; we know it's increasing if `arr[l] < arr[r]`. If we don't find an increasing sequence, move the left pointer `l` to the right's current position for the next iteration.
3. If we do find an increasing sequence, advance the right pointer `r` as long as the elements keep increasing to find the peak of the mountain.
4. Once we find the peak (where `arr[r] > arr[r + 1]`), check if there is a decreasing sequence after the peak. Keep advancing the right pointer while the sequence is decreasing.
5. Once we've finished iterating through a decreasing sequence or if we can't find a proper peak, we calculate the length of the mountain subarray (if valid) using the positions of `l` and `r`. We update `ans` with the maximum length found.
6. After processing a mountain or an increasing sequence without a valid peak, set `l` to `r` because any valid mountain subarray must start after the end of the previous one.
7. Repeat the process until we have checked all possible starting points in the array (`l + 2 < n` is used to ensure there's room for a mountain structure).

The function then returns the maximum length of any mountain subarray found during the traversal, as stored in `ans`.

Using this approach, we can find the longest mountain subarray in a single pass through the input array, resulting in an efficient solution with linear time complexity, O(n), as each element is looked at a constant number of times.

Solution Approach

The solution leverages a single pass traversal using a two-pointer approach, an algorithmic pattern that's often used to inspect sequences or subarrays within an array, especially when looking for some optimal subrange. No additional data structures are required, keeping the space complexity to O(1). The steps of the algorithm are as follows:

1. Initialize two pointers, `l` and `r` (`r = l + 1`), and an integer `ans` to zero which will store the maximum length of a valid mountain subarray.
2. Walk through the array starting from the first element, using `l` as the start of a potential mountain subarray:
   - Check if the current element and the next one form an increasing pair. If `arr[l] < arr[r]`, that signifies the start of an upward slope.
   - If an upward slope is detected, move `r` rightwards as long as `arr[r] < arr[r + 1]`, effectively climbing the mountain.
3. Once the peak is reached, which happens when you can no longer move `r` to the right without violating the mountain property (`arr[r] < arr[r + 1]` no longer holds), check if you can proceed downwards:
   - Ensure that the peak isn't the last element (we need at least one element after the peak for a valid mountain array).
   - If `arr[r] > arr[r + 1]`, then we have a downward slope. Now move `r` rightwards as long as `arr[r] > arr[r + 1]`.
4. After climbing up and down the mountain, check if a valid mountain subarray existed:
   - If a peak (greater than the first and last elements of the subarray) and a downward slope were found, update `ans` to the maximum of its current value and the length of the subarray (`r - l + 1`).
   - If the downward slope isn't present following the peak, just increment `r`.
5. Whether you found a mountain or not, set `l` to `r` to start looking for a new mountain. This step avoids overlap between subarrays, as a valid mountain subarray ends before a new one can start.
6. This process is repeated until `l` is too close to the end of the array to possibly form a mountain subarray (specifically, when `l + 2 >= n`), at which point all potential subarrays have been evaluated.

The algorithm ensures we examine each element of the array only a constant number of times as we progressively move our pointers without stepping back except to update `l` to `r`. This ensures a linear time complexity, making the solution efficient for large arrays.

Finally, we return `ans` as the result, which by the end of the traversal will hold the maximum length of the longest mountain subarray found.

Example Walkthrough

Let's apply the solution approach to a small example to illustrate how it works. We will use the following array `arr`:

```
arr = [2, 1, 4, 7, 3, 2, 5]
```

Now, let's follow the steps of the algorithm:

1. Initialize pointers `l = 0` and `r = 1`, and `ans` to `0`.
2. Starting from index `0`, we compare `arr[l]` with `arr[r]`. Since `arr[0] > arr[1]`, we do not have an upward slope, so we move `l` to the right and set `l` to `r` (now `l = 1`, `r = 2`).
3. We check the elements at `arr[l]` and `arr[r]`. Now, `arr[1] < arr[2]`, we have an increasing pair, indicating the start of an upward slope.
4. We increment `r` to `3` because `arr[2] < arr[3]`. We continue this process until `arr[3] > arr[4]`, having found the peak of our mountain.
5. Now, we check if there is a downward slope. Since `arr[3] > arr[4]`, we continue moving `r` to the right as long as the numbers keep decreasing. We now increment `r` again as `arr[4] > arr[5]`. Lastly, since `arr[5] < arr[6]`, the downward slope ends at index `5`.
6. We have found a valid mountain subarray from indices `1` to `5` with length `5 - 1 + 1 = 5`. We update `ans` to `5` because it is greater than the current value of `ans`.
7. After finding this mountain, we set `l` to the current `r` value (`l = 5`) and increment `r` to `l + 1` (now `l = 5`, `r = 6`).
8. However, `l + 2 >= arr.length` is now true, so we stop our process as no further mountain subarrays can start from the remaining elements.

Through this process, we've found that the longest mountain in `arr` is `[1, 4, 7, 3, 2]` with a length of `5`. Since we found no longer mountains during our traversal, `ans` remains `5` and that would be the value returned.

Python Solution

```python
class Solution:
    def longestMountain(self, arr: List[int]) -> int:
        length_of_array = len(arr)
        longest_mountain_length = 0  # This will store the length of the longest mountain found

        # Start exploring from the first element
        left_pointer = 0

        # Iterate over the array to find all possible mountains
        while left_pointer + 2 < length_of_array:  # The smallest mountain has at least 3 elements
            right_pointer = left_pointer + 1

            # Check for strictly increasing sequence
            if arr[left_pointer] < arr[right_pointer]:
                # Move to the peak of the mountain
                while right_pointer + 1 < length_of_array and arr[right_pointer] < arr[right_pointer + 1]:
                    right_pointer += 1

                # Check if it's a peak and not the end of the array
                if right_pointer < length_of_array - 1 and arr[right_pointer] > arr[right_pointer + 1]:
                    # Move down the mountain
                    while right_pointer + 1 < length_of_array and arr[right_pointer] > arr[right_pointer + 1]:
                        right_pointer += 1

                    # Update the longest mountain length found so far
                    mountain_length = right_pointer - left_pointer + 1
                    longest_mountain_length = max(longest_mountain_length, mountain_length)
                else:
                    # If it's not a peak, skip this element
                    right_pointer += 1

            # Move the left pointer to start exploring the next mountain
            left_pointer = right_pointer

        return longest_mountain_length  # Return the largest mountain length found
```

Java Solution

```java
class Solution {
    public int longestMountain(int[] arr) {
        int length = arr.length;
        int longestMountainLength = 0; // This will store the length of the longest mountain seen so far.

        // Iterate over each element in the array to find the mountains.
        for (int start = 0, end = 0; start + 2 < length; start = end) {
            end = start + 1; // Reset the end pointer to the next element.

            // Check if we have an increasing sequence to qualify as the first part of the mountain.
            if (arr[start] < arr[end]) {
                // Find the peak of the mountain.
                while (end + 1 < length && arr[end] < arr[end + 1]) {
                    ++end;
                }

                // Check if we have a decreasing sequence after the peak to qualify as the second part of the mountain.
                if (end + 1 < length && arr[end] > arr[end + 1]) {
                    // Descend the mountain until the sequence is decreasing.
                    while (end + 1 < length && arr[end] > arr[end + 1]) {
                        ++end;
                    }

                    // Update the longest mountain length if necessary.
                    longestMountainLength = Math.max(longestMountainLength, end - start + 1);
                } else {
                    // If not a valid mountain, move to the next position.
                    ++end;
                }
            }
        }

        return longestMountainLength; // Return the length of the longest mountain in the array.
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int longestMountain(vector<int>& arr) {
        int arraySize = arr.size(); // The size of the input array.
        int longestLength = 0; // This will hold the length of the longest mountain found.

        // Loop over the array to find all possible mountains.
        for (int startPoint = 0, endPoint = 0; startPoint + 2 < arraySize; startPoint = endPoint) {

            // Initialize the endPoint for the current mountain.
            endPoint = startPoint + 1;

            // Check if the current segment is ascending.
            if (arr[startPoint] < arr[endPoint]) {
                // Find the peak of the mountain.
                while (endPoint + 1 < arraySize && arr[endPoint] < arr[endPoint + 1]) {
                    ++endPoint;
                }

                // Check if there is a descending part after the peak.
                if (endPoint + 1 < arraySize && arr[endPoint] > arr[endPoint + 1]) {
                    // Find the end of the descending path.
                    while (endPoint + 1 < arraySize && arr[endPoint] > arr[endPoint + 1]) {
                        ++endPoint;
                    }

                    // Calculate the length of the mountain and update the longestLength if necessary.
                    longestLength = max(longestLength, endPoint - startPoint + 1);
                } else {
                    // If there is no descending part, move the endPoint forward.
                    ++endPoint;
                }
            }
        }

        // Return the length of the longest mountain found in the array.
        return longestLength;
    }
};
```

Typescript Solution

```typescript
function longestMountain(arr: number[]): number {
    let arraySize: number = arr.length; // The size of the input array.
    let longestLength: number = 0; // This will hold the length of the longest mountain found.

    // Loop over the array to find all possible mountains.
    for (let startPoint: number = 0, endPoint: number = 0; startPoint + 2 < arraySize; startPoint = endPoint) {
        // Initialize the endPoint for the current mountain.
        endPoint = startPoint + 1;

        // Check if the current segment is ascending.
        if (arr[startPoint] < arr[endPoint]) {
            // Find the peak of the mountain.
            while (endPoint + 1 < arraySize && arr[endPoint] < arr[endPoint + 1]) {
                ++endPoint;
            }

            // Check if there is a descending part after the peak.
            if (endPoint + 1 < arraySize && arr[endPoint] > arr[endPoint + 1]) {
                // Find the end of the descending path.
                while (endPoint + 1 < arraySize && arr[endPoint] > arr[endPoint + 1]) {
                    ++endPoint;
                }

                // Calculate the length of the mountain and update the longest Length if necessary.
                longestLength = Math.max(longestLength, endPoint - startPoint + 1);
            } else {
                // If there is no descending part, move the endPoint forward.
                ++endPoint;
            }
        }
    }

    // Return the length of the longest mountain found in the array.
    return longestLength;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the function `longestMountain` can be analyzed by examining the while loop and nested while loops. The function traverses the array using pointers `l` and `r`. The outer while loop runs while `l + 2 < n`, ensuring at least 3 elements to form a mountain. The first inner while loop executes when a potential ascending part of a mountain is found (`arr[l] < arr[r]`) and continues until the peak is reached. The second inner while loop executes if a peak is found and continues until the end of the descending part.

Each element is visited at most twice: once during the ascent and once during the descent. Hence, the main loop has at most `O(2n)` iterations, which simplifies to `O(n)` where `n` is the length of the array.

Space Complexity

The space complexity of the code is `O(1)`, as it uses a constant amount of extra space. The variables `n`, `ans`, `l`, and `r` do not depend on the input size, and no additional data structures are used that scale with the input size.