941. Valid Mountain Array

Easy Array

Problem Description

The problem requires us to determine if a given array `arr` of integers represents a mountain array or not. By definition, an array is considered a mountain array if it meets the following criteria:

- Its length is at least 3.
- There exists a peak element `arr[i]`, such that:
  - Elements before `arr[i]` are in strictly increasing order. This means `arr[0] < arr[1] < ... < arr[i - 1] < arr[i]`.
  - Elements after `arr[i]` are in strictly decreasing order. This means `arr[i] > arr[i + 1] > ... > arr[arr.length - 1]`.

In essence, the array should rise to a single peak and then consistently decrease from that point, simulating the shape of a mountain.

Intuition

The idea for solving this problem is to find the peak of the potential mountain and to verify that there is a clear increasing sequence before the peak and a clear decreasing sequence after the peak.

To ascertain whether the array `arr` is a mountain array, we can walk from both ends of the array towards the center:

- We start from the beginning of the array and keep moving right as long as each element is greater than the previous one (the array is ascending). We do this to find the increasing slope of the mountain.
- Simultaneously, we start from the end of the array and keep moving left as long as each element is less than the previous one (the array is descending). We do this to find the decreasing slope of the mountain.

If after traversing, we find that both pointers meet at the same index (the peak), and it's neither the first nor the last index (since the peak cannot be at the ends), we can conclude that it is a valid mountain array. Otherwise, if the pointers do not meet or the peak is at the first or last index, the array is not a mountain array.

Solution Approach

The solution approach utilizes a two-pointer technique, which is an algorithmic pattern where we use two indices to iterate over the data structure—in this case, the array—to solve the problem with better time or space complexity. The solution does not require any additional data structures and operates directly on the input array, which makes it an in-place algorithm with `O(1)` extra space complexity.

Here's the step-by-step process of the solution approach:

1. **Initial Check**: First, we check if the array `arr` has at least three elements. If not, it cannot form a mountain array, so we return `False`.
2. **Finding the Increasing Part**: We set a pointer `l` at the start of the array. We move `l` to the right (`l += 1`) as long as `arr[l] < arr[l + 1]`. This loop will stop when `l` points to the peak of the mountain or when `arr` is no longer strictly increasing.
3. **Finding the Decreasing Part**: We set a pointer `r` at the end of the array. Similarly, we move `r` to the left (`r -= 1`) as long as `arr[r] < arr[r - 1]`. This loop will stop when `r` points to the peak of the mountain or when `arr` is no longer strictly decreasing.
4. **Checking the Peak**: After exiting both loops, if `l` and `r` are pointing to the same index, it means that we have potentially found the peak of the mountain. But for a valid mountain array, the peak cannot be the first or last element. Therefore, if `l` and `r` meet at the same index and it's not at the ends of the array, we can conclude that the array is a valid mountain array and return `True`.
5. **Result**: If the above conditions are not met, we return `False` since the array does not satisfy the criteria for being a mountain array.

The solution has a time complexity of `O(n)` where `n` is the length of the array since in the worst case we could traverse the entire array once with each pointer.

Example Walkthrough

Let's take a small example with the array `arr = [1, 3, 5, 4, 2]` to illustrate the solution approach.

1. **Initial Check**: First, we check if `arr` has at least three elements. In this case, the array has five elements, so we can proceed.
2. **Finding the Increasing Part**: We set a pointer `l` at the start of the array (`l = 0`). We move `l` to the right while `arr[l] < arr[l + 1]`.
   - `l = 0`: we check if `arr[0] < arr[1]` which means `1 < 3`. It's true, so we increment `l` to `1`.
   - `l = 1`: we check if `arr[1] < arr[2]` which means `3 < 5`. It's true, so we increment `l` to `2`.
   - `l = 2`: we check if `arr[2] < arr[3]` which means `5 < 4`. It's false, so we stop. Now `l` is pointing to the peak at index `2`.
3. **Finding the Decreasing Part**: Similarly, we set a pointer `r` at the end of the array (`r = 4`). We move `r` to the left while `arr[r] < arr[r - 1]`.
   - `r = 4`: we check if `arr[4] < arr[3]` which means `2 < 4`. It's true, so we decrement `r` to `3`.
   - `r = 3`: we check if `arr[3] < arr[2]` which means `4 < 5`. It's true, so we decrement `r` to `2`.
   - Now `r` is pointing to the same peak index as `l`, which is index `2`.
4. **Checking the Peak**: Both `l` and `r` are pointing to the same index (`2`). We check if this is not the first or last index of the array. Since `2` is neither `0` nor `4`, the conditions are satisfied.
5. **Result**: Since all conditions are met (`l` and `r` met at the same index, not at the ends, and ascending/descending sequences are correct), we conclude that the given array `arr = [1, 3, 5, 4, 2]` is a valid mountain array and return `True`.

This example demonstrated a successful case where the array meets the criteria for a mountain array. If, for instance, `arr` had been `[1, 2, 2, 1]`, the solution would have returned `False` because there is no strictly increasing or strictly decreasing sequence due to the repeated `2`s.

Solution Implementation

Python

```python
class Solution:
    def validMountainArray(self, arr: List[int]) -> bool:
        # Get the length of the array
        num_elements = len(arr)

        # A valid mountain array requires at least 3 elements
        if num_elements < 3:
            return False

        # Initialize two pointers starting from the beginning and the end of the array
        left_index, right_index = 0, num_elements - 1

        # Move the left index towards the right, stop before the peak
        while left_index + 1 < num_elements - 1 and arr[left_index] < arr[left_index + 1]:
            left_index += 1

        # Move the right index towards the left, stop before the peak
        while right_index - 1 > 0 and arr[right_index] < arr[right_index - 1]:
            right_index -= 1

        # The mountain peak is valid if left and right pointers meet at the same index,
        # implying exactly one peak is present.
        return left_index == right_index
```

Java

```java
class Solution {

    // Method to check if the given array represents a valid mountain array.
    public boolean validMountainArray(int[] arr) {
        int length = arr.length; // Get the length of the array

        // A mountain array must have at least 3 elements
        if (length < 3) {
            return false;
        }

        int left = 0;   // Starting index from the beginning of the array
        int right = length - 1; // Starting index from the end of the array

        // Move the left index towards the right as long as the next element is greater
        // which is analogous to climbing up the mountain
        while (left + 1 < length - 1 && arr[left] < arr[left + 1]) {
            left++;
        }

        // Move the right index towards the left as long as the previous element is greater
        // which is analogous to walking down the mountain
        while (right - 1 > 0 && arr[right] < arr[right - 1]) {
            right--;
        }

        // For the array to be a mountain array, we must have climbed up and then walked down the mountain,
        // which means the indexes should meet at the peak, and the peak can't be the first or last element.
        return left == right;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    bool validMountainArray(std::vector<int>& arr) {
        int arrSize = arr.size();

        // A mountain array must have at least 3 elements
        if (arrSize < 3) return false;

        // Initialize pointers to start and end of the array
        int leftIndex = 0, rightIndex = arrSize - 1;

        // Move leftIndex rightward as long as the next element is greater
        while (leftIndex + 1 < arrSize - 1 && arr[leftIndex] < arr[leftIndex + 1]) {
            ++leftIndex;
        }

        // Move rightIndex leftward as long as the previous element is greater
        while (rightIndex - 1 > 0 && arr[rightIndex] < arr[rightIndex - 1]) {
            --rightIndex;
        }

        // A valid mountain array will have both pointers meet at the same peak index
        return leftIndex == rightIndex;
    }
};
```

TypeScript

```typescript
function validMountainArray(arr: number[]): boolean {
    let arrSize = arr.length;

    // A mountain array must have at least 3 elements.
    if (arrSize < 3) {
        return false;
    }

    // Initialize pointers to start and end of the array.
    let leftIndex = 0;
    let rightIndex = arrSize - 1;

    // Move leftIndex rightward as long as the next element is greater.
    while (leftIndex + 1 < arrSize && arr[leftIndex] < arr[leftIndex + 1]) {
        leftIndex++;
    }

    // Move rightIndex leftward as long as the previous element is greater.
    while (rightIndex > 0 && arr[rightIndex - 1] > arr[rightIndex]) {
        rightIndex--;
    }

    // A valid mountain array will have both pointers meet at the same peak index
    // and that index cannot be the first or last index of the array.
    return leftIndex == rightIndex && leftIndex !== 0 && rightIndex !== arrSize - 1;
}
```

```python
class Solution:
    def validMountainArray(self, arr: List[int]) -> bool:
        # Get the length of the array
        num_elements = len(arr)

        # A valid mountain array requires at least 3 elements
        if num_elements < 3:
            return False

        # Initialize two pointers starting from the beginning and the end of the array
        left_index, right_index = 0, num_elements - 1

        # Move the left index towards the right, stop before the peak
        while left_index + 1 < num_elements - 1 and arr[left_index] < arr[left_index + 1]:
            left_index += 1

        # Move the right index towards the left, stop before the peak
        while right_index - 1 > 0 and arr[right_index] < arr[right_index - 1]:
            right_index -= 1

        # The mountain peak is valid if left and right pointers meet at the same index,
        # implying exactly one peak is present.
        return left_index == right_index
```

Time and Space Complexity

The time complexity of the given `validMountainArray` function can be considered to be O(n), where n is the length of the input array `arr`. This is because the function uses two while loops that iterate through the array from both ends, but each element is visited at most once. The first loop increments the variable `l` until the ascending part of the mountain is traversed, and the second loop decrements the variable `r` until the descending part of the mountain is traversed. In the worst case, these two loops together will scan all elements of the array once.

The space complexity of the function is O(1), as it only uses a fixed number of extra variables (`n, l, r`) that do not depend on the size of the input array.