2951. Find the Peaks

Easy Array Enumeration

Problem Description

In this problem, we are given a 0-indexed integer array named `mountain`. Our goal is to identify all the elements in the array that qualify as "peaks". A peak in the `mountain` array is an element that is strictly greater than its immediate neighbors to the left and right. We are tasked with returning an array containing the indices of all such peaks. It's important to note that according to the problem rules, the first element (at index 0) and the last element (at the end of the array) cannot be considered as peaks regardless of their value compared to adjacent elements.

Intuition

The intuition behind the solution strategy for this problem involves iterating through the array and examining elements that have neighboring elements on both sides – this means starting our assessment from the second element (index 1) and concluding it with the second-to-last element (as the first and last elements cannot be peaks). For each element at index `i`, where `1 <= i < len(mountain) - 1`, we check the following condition: is the current element greater than the element on its left (`mountain[i - 1]`) and greater than the element on its right (`mountain[i + 1]`)? If both these conditions are met, then the element at index `i` is considered a peak, and we add its index `i` to our list of peaks. This process is repeated for all eligible elements in the array. After we've checked all elements, we return the list of indices that correspond to the peaks we've found.

Solution Approach

The solution for finding all peaks in the `mountain` array is implemented through a simple for-loop that iterates over the indices of the array from index `1` to `len(mountain) - 2`, inclusively. The reasoning behind starting the loop at index `1` and ending at `len(mountain) - 2` is that the first and last elements cannot be peaks by definition, so they are automatically excluded from consideration.

Now, let's discuss the data structures and patterns used in the implementation:

- **Data Structures**: The primary data structure used here is the array (or `List` in Python), both for the input (`mountain`) and output (to store the indices of the peaks). Arrays are ideal for this solution because they allow us to access elements by their index efficiently, which is essential for comparing an element with its neighbors.
- **Algorithm/Pattern**: The core pattern here is a simple linear scan of the array, which is a basic algorithmic strategy. At each index `i` being considered by the loop, we apply the definition of a "peak":
  - To determine if the element at the current index `i` is a peak, we perform two comparisons:
    - Check if the element is greater than the element at index `i - 1` (to the left), denoted as `mountain[i] > mountain[i - 1]`.
    - Check if the element is also greater than the element at index `i + 1` (to the right), denoted as `mountain[i] > mountain[i + 1]`.

If both conditions are true, we conclude that we have found a peak, and the index `i` of this peak is added to the output list.

- **Implementation**: Below is the implementation based on the description provided in the Reference Solution Approach:

```
class Solution:
    def findPeaks(self, mountain: List[int]) -> List[int]:
        return [
            i
            for i in range(1, len(mountain) - 1)
            if mountain[i - 1] < mountain[i] > mountain[i + 1]
        ]
```

This Python implementation makes use of list comprehension, which is a concise way to create lists based on existing lists. It is used here to generate the list of peak indices in a single line of code.

The efficiency of this approach comes from the fact that every element is checked only once, and there are no nested loops, which means that the runtime complexity is linear O(n), where `n` is the number of elements in the `mountain` array.

Example Walkthrough

Let's use a small example `mountain` array to illustrate the solution approach:

Suppose our initial `mountain` array is `[2, 3, 5, 4, 1, 3, 2, 4]`.

Now, following our solution approach, we will look for peaks, which are elements higher than their immediate neighbors:

1. Start at index 1 with the value `3` and compare it to its neighbors. Its left neighbor is `2` and right neighbor is `5`. Since `3` is not greater than `5`, it is not a peak.
2. Move to index 2 with the value `5`. The left neighbor is `3` and the right neighbor is `4`. `5` is greater than both `3` and `4`, so index `2` is a peak.
3. Next, at index 3 with the value `4`, we compare it to its neighbors `5` (left) and `1` (right). As `4` is not greater than `5`, it is therefore not a peak.
4. Proceed to index 4, the value `1` is not a peak since its left neighbor `4` is greater.
5. At index 5 with the value `3`, we compare it to the neighbors `1` (left) and `2` (right). `3` is greater than both `1` and `2`, making index `5` a peak.
6. Then, at index 6 with the value `2`, it has neighbors `3` (left) and `4` (right). Since `2` is less than `3`, it's not a peak.
7. Lastly, we do not consider index 7 as it's the end of the array and by rule can't be a peak.

The output based on our solution approach should, therefore, be a list of indices that are peaks, in this case, `[2, 5]`.

In this example, we demonstrated the linear scan approach mentioned in the solution strategy. We iterated over the array elements (skipping the first and last ones), compared each element to its neighbors, and found the peaks by checking if they are greater than both the element immediately to their left and right. The simplicity of this algorithm allows it to run with O(n) complexity where n is the length of the `mountain` array, because each element is checked only once.

Solution Implementation

Python

```
# Define the Solution class
class Solution:
    def find_peaks(self, mountain: List[int]) -> List[int]:
        # This method finds and returns the indices of all peak elements in the given mountain list.
        # A peak element is one that is strictly greater than its neighbors.

        # Initialize a list to store the indices of the peak elements
        peaks_indices = []

        # Iterate through the mountain list, starting from the second element and ending at the second to last
        for i in range(1, len(mountain) - 1):
            # Check if the current element is a peak, i.e., it is greater than its left and right neighbors
            if mountain[i - 1] < mountain[i] > mountain[i + 1]:
                # If it's a peak, append the index to the peaks_indices list
                peaks_indices.append(i)

        # Return the list of peak indices
        return peaks_indices

# Note: Before using the code, make sure to import the `List` type from the `typing` module like this:
# from typing import List
```

Java

```
import java.util.ArrayList;
import java.util.List;

class Solution {

    /**
     * Finds and returns a list of indices representing peak elements in a given array.
     * A peak element is an element which is greater than its neighbors.
     *
     * @param mountain An array representing the heights of the mountain at each point.
     * @return A list of integers representing the indices of the peak elements.
     */
    public List<Integer> findPeaks(int[] mountain) {
        // The list to store indices of peak elements.
        List<Integer> peaks = new ArrayList<>();

        // Iterate over the array elements, starting from the second element and
        // ending at the second last element, to avoid out-of-bounds situations.
        for (int i = 1; i < mountain.length - 1; ++i) {
            // Compare the current element with its neighbors to check if it's a peak.
            if (mountain[i] > mountain[i - 1] && mountain[i] > mountain[i + 1]) {
                // If it's a peak, add its index to the list.
                peaks.add(i);
            }
        }

        // Return the list of peak indices.
        return peaks;
    }
}
```

C++

```
#include <vector>

class Solution {
public:
    // Function to find all the peak elements of a given vector 'mountain'
    // and return their indices.
    std::vector<int> findPeaks(std::vector<int>& mountain) {
        std::vector<int> peaks;  // Vector to store the indices of the peaks

        // Iterate through the elements of the array, starting from the second element
        // and ending at the second to last element.
        for (int i = 1; i < mountain.size() - 1; ++i) {
            // Check if the current element is larger than the one before it
            // and the one after it, which makes it a peak.
            if (mountain[i - 1] < mountain[i] && mountain[i + 1] < mountain[i]) {
                peaks.push_back(i);  // If it is a peak, add its index to the 'peaks' vector
            }
        }

        return peaks;  // Return the vector with the indices of all peaks
    }
};
```

TypeScript

```
/**
 * Identifies the peak elements in a mountain array.
 * A peak element is defined as an element that is greater than its neighbors.
 * @param {number[]} mountain - The array representing the mountain with heights as elements.
 * @returns {number[]} Indices of all peak elements in the mountain array.
 */
function findPeaks(mountain: number[]): number[] {
    // Initialize an array to store the indices of peak elements.
    const peaks: number[] = [];

    // Iterate through the array starting from the second element and ending at the second to last element.
    for (let i = 1; i < mountain.length - 1; ++i) {
        // Check if the current element is greater than its immediate neighbors.
        if (mountain[i - 1] < mountain[i] && mountain[i + 1] < mountain[i]) {
            // If the current element is a peak, add its index to the peaks array.
            peaks.push(i);
        }
    }

    // Return the array of peak elements' indices.
    return peaks;
}


# Define the Solution class
class Solution:
    def find_peaks(self, mountain: List[int]) -> List[int]:
        # This method finds and returns the indices of all peak elements in the given mountain list.
        # A peak element is one that is strictly greater than its neighbors.

        # Initialize a list to store the indices of the peak elements
        peaks_indices = []

        # Iterate through the mountain list, starting from the second element and ending at the second to last
        for i in range(1, len(mountain) - 1):
            # Check if the current element is a peak, i.e., it is greater than its left and right neighbors
            if mountain[i - 1] < mountain[i] > mountain[i + 1]:
                # If it's a peak, append the index to the peaks_indices list
                peaks_indices.append(i)

        # Return the list of peak indices
        return peaks_indices

# Note: Before using the code, make sure to import the `List` type from the `typing` module like this:
# from typing import List
```

Time and Space Complexity

The given code snippet is designed to find the peaks in a list of integers representing elevations in a mountain sequence. A peak is defined as an element that is greater than its immediate neighbors.

Time Complexity

The function `findPeaks` iterates over the input `mountain` list exactly once, starting from index 1 and ending at `len(mountain) - 1`. For each element, it performs a constant time comparison with its neighbors. This results in a linear time complexity relative to the size of the input list. Therefore, the time complexity of the function is `O(n)`, where `n` is the length of the `mountain` list.

Space Complexity

The space complexity of the function consists of two parts: additional data structures used within the function and the output list. Since no additional data structures are used other than the output list, and ignoring the output list's space, the space complexity is `O(1)`. This implies that the function consumes a constant amount of space, irrespective of the size of the input list.