852. Peak Index in a Mountain Array

Medium Array Binary Search

Problem Description

In this problem, we're given an array `arr` that represents a mountain, which means it has the shape of a peak: the elements first increase and then decrease. The constraints that define a mountain are:

1. The length of the array should be at least 3.
2. There exists an index `i` where `0 < i < arr.length - 1`, and the elements of the array increase from `arr[0]` to `arr[i - 1]`, reach a peak at `arr[i]`, and then decrease from `arr[i + 1]` to `arr[arr.length - 1]`.

Our goal is to find the peak of the mountain, which is the index `i` at the top of the mountain. Importantly, we're asked to find this peak index within a logarithmic time complexity, more precisely `O(log(arr.length))`, which suggests that a straightforward linear scan won't be sufficient as it would take linear time.

Intuition

To find the peak index in an efficient way that meets the time complexity requirement of `O(log(arr.length))`, we immediately think of the binary search algorithm. Binary search cuts the problem space in half each time it makes a comparison, which results in the logarithmic time complexity.

The intuition behind using binary search for this problem lies in the properties of a mountain array. Since the elements strictly increase up to the peak and then strictly decrease, if we pick a point in the array and look at its immediate neighbor(s), we can decide which half of the array the peak lies in:

1. If the current element is greater than its right neighbor, we know that the peak is at this element or to its left.
2. Conversely, if the current element is less than its right neighbor, the peak must be to the right of this element.

We can keep narrowing down our search space using these comparisons until we converge on the peak element.

The provided solution in Python performs this algorithm efficiently using a while loop to iterate through the potential peak indices within the boundaries defined by `left` and `right`. We intentionally start `left` at 1 and `right` at `len(arr) - 2` to ensure we do not consider the endpoints of the array, which cannot be peaks based on the mountain array definition. The use of the bitwise shift operation `>> 1` is a common technique to quickly divide `mid` by 2, which is a part of the binary search approach to find the middle index between `left` and `right`.

Solution Approach

The solution to finding the peak index in the mountain array employs the binary search algorithm. Here's a detailed walkthrough of the implementation based on the provided code:

1. We initialize two pointers, `left` and `right`, that define the range within which we'll conduct our search. As the peak cannot be the first or last element (by the definition of a mountain array), we set `left` to 1 and `right` to `len(arr) - 2`.

2. We enter a while loop that will run as long as `left` is less than `right`, ensuring that we consider at least two elements. This is necessary because we compare the element at the midpoint with its immediate right neighbor to determine the direction of the search.

3. Within the loop, we calculate the midpoint `mid` using `(left + right) >> 1`, which is equivalent to `(left + right) / 2` but faster computationally as it is a bit-shift operation that divides `right` and `left` by 2.

4. We then perform a comparison between the elements at `arr[mid]` and `arr[mid + 1]`. If `arr[mid]` is greater than `arr[mid + 1]`, this implies we are currently at a descending part of the array, or we may have found the peak. Thus, the peak index must be at `mid` or to its left. In this case, we move the `right` pointer to `mid` to narrow the search range.

5. If `arr[mid]` is not greater than `arr[mid + 1]`, it means we're on an ascending part of the array, and the peak lies to the right of `mid`. Consequently, we move the `left` pointer to `mid + 1` to adjust the search range.

6. The loop continues until `left` and `right` converge, which happens when `left` equals `right`. At this point, both pointers are indicating the peak's index, so we return the value of `left`.

By repeatedly halving the range of possible indices, we ensure the logarithmic time complexity `O(log(arr.length))`, as required by the problem.

One thing to note about this implementation is that it uses a half-interval search. This means we always move one of our boundaries to the midpoint itself, not past it. This approach guarantees that the search space is reduced after each iteration and that we don't overshoot the peak and miss it in our search.

Lastly, the algorithm doesn't use any auxiliary data structures, it operates directly on the input array, which ensures space complexity of `O(1)` - only constant extra space is used.

Example Walkthrough

Let's consider a sample mountain array `arr = [1, 3, 5, 4, 2]` to illustrate the solution approach:

1. Initialize two pointers, `left = 1` and `right = len(arr) - 2 = 3`, to avoid checking the first and last elements as they cannot be the peak by definition.

2. Start the while loop since `left < right` (1 < 3 is true).

3. Calculate the midpoint `mid` using `(left + right) >> 1`. For our example, it's `(1 + 3) >> 1`, which equals 2.

4. Compare `arr[mid]` to `arr[mid + 1]`. For `mid = 2`, `arr[mid]` is 5 and `arr[mid + 1]` is 4. Because 5 > 4, we update `right` to `mid`. Now `left = 1` and `right = 2`.

5. The loop continues because `left < right` is still true.

6. Recalculate the midpoint with the updated pointers. Now, `mid` is `(1 + 2) >> 1`, which equals 1.

7. Compare `arr[mid]` to `arr[mid + 1]` again. For `mid = 1`, `arr[mid]` is 3 and `arr[mid + 1]` is 5. Because 3 < 5, we update `left` to `mid + 1`. Now `left = 2` and `right = 2`.

8. The loop ends because `left` equals `right`, indicating convergence at the peak's index, which is `2` in our example array.

9. Return `left`, which is the peak index. In our array, `arr[2]` is indeed the peak with a value of 5.

Through each iteration, the search space is narrowed down until the peak is found, fulfilling the `O(log(arr.length))` time complexity for this logarithmic search approach.

Python Solution

```python
class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        # Starting the search from second elemeent to second last element
        # because the peak can't be the first or last element.
        left_index, right_index = 1, len(arr) - 2

        # Use binary search to find the peak element
        while left_index < right_index:
            # Calculate the middle index of the current subarray
            mid_index = (left_index + right_index) // 2  # Use '//' for floor division in Python 3

            # If the middle element is greater than its next element,
            # then the peak is in the left side of mid. Update the right_index.
            if arr[mid_index] > arr[mid_index + 1]:
                right_index = mid_index
            else:
                # If the middle element is less than its next element,
                # then the peak is in the right side of mid. Update the left_index.
                left_index = mid_index + 1

        # When left_index==right_index, we have found the peak index
        return left_index

```

Java Solution

```java
class Solution {
    public int peakIndexInMountainArray(int[] arr) {
        // Initialize left and right pointers, excluding the first and last elements
        // because the peak cannot be at the ends of the array.
        int left = 1;
        int right = arr.length - 2;

        // Perform a binary search to find the peak element
        while (left < right) {
            // Calculate the middle index
            int mid = left + (right - left) / 2;

            // If the middle element is greater than its successor,
            // the peak is in the left half, including the mid element
            if (arr[mid] > arr[mid + 1]) {
                right = mid;
            } else {
                // If the middle element is less than or equal to its successor,
                // the peak is in the right half
                left = mid + 1;
            }
        }

        // When left equals right, we have found the peak index
        return left;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to find the peak index in a mountain array
    int peakIndexInMountainArray(vector<int>& arr) {
        // Initialize the search range within the array
        int left = 1; // Starting from 1 because the peak cannot be the first element
        int right = arr.size() - 2; // Ending at size - 2 because the peak cannot be the last element

        // Continue searching as long as the left index is less than the right index
        while (left < right) {
            // Calculate the middle index using bitwise right shift (equivalent to dividing by 2)
            int mid = (left + right) >> 1;

            // If the middle element is greater than its next element,
            // we are in the descending part of the mountain
            // therefore, we continue to search on the left side
            if (arr[mid] > arr[mid + 1]) {
                right = mid;
            } else {
                // Else, we are in the ascending part of the mountain,
                // we continue to search on the right side
                left = mid + 1;
            }
        }

        // Since 'left' and 'right' converge to the peak index,
        // we return 'left' as the peak index of the mountain array
        return left;
    }
};

```

Typescript Solution

```typescript
function peakIndexInMountainArray(arr: number[]): number {
    // Initialize the search range within the boundaries of the potential peak.
    // We avoid the first and last elements as they cannot be the peak.
    let left = 1;
    let right = arr.length - 2;

    // Use binary search to find the peak of the mountain array.
    while (left < right) {
        // Calculate the mid index by shifting right bitwise by 1 (equivalent to dividing by 2).
        const mid = left + ((right - left) >> 1);

        // If the mid element is greater than its next element, we continue in the left portion.
        // This is because the peak must be to the left of mid or at mid.
        if (arr[mid] > arr[mid + 1]) {
            right = mid;
        } else {
            // Otherwise, the peak lies to the right of mid, so we continue searching in the right portion.
            left = mid + 1;
        }
    }
    // After the loop, left will be pointing at the peak element's index
    return left;
}

```

Time and Space Complexity

The given Python code performs a binary search to find the peak index in a mountain array. The complexity analysis is:

- **Time Complexity**: The `while` loop keeps halving the search interval until `left` and `right` meet. This halving process continues for at most `O(log n)` iterations, where `n` is the length of the array. Hence, the time complexity of this code is `O(log n)`.

- **Space Complexity**: The space complexity is `O(1)` because the code uses a fixed amount of additional memory (a few variables for left, right, and mid indices) regardless of the input array size.