162. Find Peak Element

Medium Array Binary Search

Problem Description

The problem presents an integer array called `nums` which is 0-indexed. We are tasked with finding an index of a peak element. A peak element is defined as an element that is strictly greater than its neighboring elements. Here the term "strictly" means that the peak element must be greater than its neighbors, not equal to them.

Furthermore, the problem scenario extends the array conceptually, so that if you look at the start or end of the array, it's as if there's an invisible `-∞` to the left of the first element and to the right of the last element. This means that if the first or last element is greater than their one real neighbor, they also count as peak elements.

We're also given a constraint on the time complexity: the solution needs to run in `O(log n)` time, which implies that a simple linear scan of the array is not efficient enough. We need to use an algorithm that repeatedly divides the array into smaller segments—binary search is an example of such an algorithm.

Intuition

Given the requirement to complete the task in `O(log n)` time, we must discard a linear scan approach that would take `O(n)` time. Instead, we adopt a binary search method due to its logarithmic time complexity, which fits our constraint.

Binary search is a technique often used for searching a sorted array by repeatedly dividing the search interval in half. Although in this case the entire array isn't sorted, we can still use binary search because of the following key insight: if an element is not a peak (meaning it's less than either of its neighbors), then a peak must exist to the side of the greater neighbor.

The reason this works is because of our `-∞` bounds at both ends of the array. We imagine a slope up from `-∞` to a non-peak element and then down from the non-peak element towards `-∞`. Somewhere on that rising or falling slope there must be a peak, which is a local maximum.

So in our modified binary search, instead of looking for a specific value, we look for any peak as follows:

1. We take the middle element of the current interval and compare it with its right neighbor.
2. If the middle element is greater than its right neighbor, we've found a descending slope, and there must be a peak to the left. Hence, we restrict our search to the left half of the current interval.
3. If the middle element is less than its right neighbor, we've found an ascending slope, and a peak exists to the right. We then do our next search on the right half.
4. We continue this process of narrowing down our search interval by half until we've isolated a peak element.

This binary search on a wave-like array ensures we find a peak in `O(log n)` time, satisfying the problem's constraints.

Solution Approach

The solution leverages a binary search algorithm, which is ideal for situations where we need to minimize the time complexity to `O(log n)`. The essence of binary search in this context is to reduce the search space by half after each comparison, making it much faster than a linear approach.

Here's a step-by-step explanation of the implementation:

1. We initialize two pointers, `left` and `right`, which represent the boundaries of our current search interval. `left` is set to `0`, and `right` is set to the length of the array minus one (`len(nums) - 1`).

2. We enter a while loop that continues as long as `left` is less than `right`, ensuring that we are still considering a range of possible positions for a peak element.

3. Inside the loop, we calculate the midpoint of the current search interval as `mid = (left + right) >> 1`. The `>> 1` operation is a bitwise shift to the right by 1 bit, which is equivalent to dividing by two, but it's often faster.

4. We then compare the element at the `mid` position with the element immediately to its right (`nums[mid + 1]`).

5. If `nums[mid]` is greater than `nums[mid + 1]`, the peak must be at `mid` or to the left of `mid`. Thus, we set `right` to `mid`, effectively narrowing the search interval to the left half.

6. On the other hand, if `nums[mid]` is less than or equal to `nums[mid + 1]`, then a peak lies to the right. Thus, we set `left` to `mid + 1`, narrowing the search interval to the right half.

7. The loop continues, halving the search space each time, until `left` equals `right`. At this point, we have found the peak because it means that `nums[left]` cannot be smaller than both its neighbors (as per the `nums[-1] = nums[n] = -∞` rule).

8. We exit the loop and return `left`, which is the index of the peak element.

This approach guarantees finding a peak, if not the highest peak, in logarithmic time.

Here's the code snippet that follows this approach:

```python
class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1
        while left < right:
            mid = (left + right) >> 1
            if nums[mid] > nums[mid + 1]:
                right = mid
            else:
                left = mid + 1
        return left
```

The simplicity of binary search in conjunction with the described logic yields an efficient and elegant solution for finding a peak element.

Example Walkthrough

Let's walk through the solution with a small example.

Suppose our input array `nums` is `[1, 2, 3, 1]`. We want to find an index of a peak element.

Initial Setup

- The initial value of `left` is `0`.
- The initial value of `right` is `len(nums) - 1`, which is `3`.

Iteration 1

- Calculate the midpoint: `mid = (left + right) >> 1` which is `(0 + 3) >> 1 = 1`.
- Compare `nums[mid]` and `nums[mid + 1]`: `nums[1]` is `2`, and `nums[2]` is `3`.
- Since `2` is less than `3`, we are on an ascending slope. We should move right.
- Update `left` to `mid + 1`: `left` becomes `2`.

Iteration 2

- `left` is now `2` and `right` is `3`.
- Calculate the new midpoint: `mid = (left + right) >> 1` which is `(2 + 3) >> 1 = 2`.
- Compare `nums[mid]` and `nums[mid + 1]`: `nums[2]` is `3`, and `nums[3]` is `1`.
- Since `3` is greater than `1`, we're on a descending slope. We should move left.
- Update `right` to `mid`: `right` becomes `2`.

Conclusion

- The loop ends when `left` equals `right`, which is now the case (`left` and `right` are both `2`).
- Therefore, we have found our peak at index `2` where the element is `3`, and it is greater than both its neighbors (where the neighbor on the right is `1`, and the neighbor on the left is `2`, and conceptually `-∞` on both ends).

Thus, the index `2` is returned.

Using this approach with the given example, we can see how the binary search algorithm rapidly narrows down the search to find the peak element, satisfying the `O(log n)` time complexity constraint.

Python Solution

```python
from typing import List

class Solution:
    def findPeakElement(self, nums: List[int]) -> int:
        # Initialize the start and end pointers.
        start, end = 0, len(nums) - 1

        # Binary search to find the peak element.
        while start < end:
            # Find the middle index.
            mid = (start + end) // 2

            # If the middle element is greater than its next element,
            # it means a peak element is on the left side(inclusive of mid).
            if nums[mid] > nums[mid + 1]:
                end = mid
            # Otherwise, the peak is in the right half of the array.
            else:
                start = mid + 1

        # When start and end pointers meet, we've found a peak element.
        return start

```

Java Solution

```java
class Solution {
    public int findPeakElement(int[] nums) {
        int left = 0; // Initialize the left boundary of the search space
        int right = nums.length - 1; // Initialize the right boundary of the search space

        // Continue the loop until the search space is reduced to one element
        while (left < right) {
            // Calculate the middle index of the current search space
            int mid = left + (right - left) / 2;

            // If the middle element is greater than its next element, then a peak must be to the left (including mid)
            if (nums[mid] > nums[mid + 1]) {
                // Narrow the search space to the left half
                right = mid;
            } else {
                // Otherwise, the peak exists in the right half (excluding mid)
                // Narrow the search space to the right half
                left = mid + 1;
            }
        }

        // When left == right, we have found the peak element's index, return it
        return left;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include vector header for using the vector container

class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        // Initialize the left and right pointers
        int left = 0;
        int right = nums.size() - 1;

        // Perform binary search
        while (left < right) {
            // Find the middle index
            // Using (left + (right - left) / 2) avoids potential overflow of integer addition
            int mid = left + (right - left) / 2;

            // If the middle element is greater than the next element,
            // the peak must be in the left half (including mid)
            if (nums[mid] > nums[mid + 1]) {
                right = mid;
            } else {
                // If the middle element is smaller than the next element,
                // the peak must be in the right half (excluding mid)
                left = mid + 1;
            }
        }
        // At the end of the loop, left == right, which points to the peak element
        return left;
    }
};

```

Typescript Solution

```typescript
function findPeakElement(nums: number[]): number {
    // Initialize the search boundaries to the start and end of the array
    let leftBoundary: number = 0;
    let rightBoundary: number = nums.length - 1;

    // Continue searching as long as the search space contains more than one element
    while (leftBoundary < rightBoundary) {
        // Find the middle index using bitwise operator
        const middleIndex: number = leftBoundary + ((rightBoundary - leftBoundary) >> 1);

        // Compare the middle element to its next element
        if (nums[middleIndex] > nums[middleIndex + 1]) {
            // If the middle element is greater than the next element,
            // then a peak element is in the left half (including middle)
            rightBoundary = middleIndex;
        } else {
            // Otherwise, the peak element is in the right half (excluding middle)
            leftBoundary = middleIndex + 1;
        }
    }

    // When leftBoundary equals rightBoundary, we found the peak element.
    // Return its index.
    return leftBoundary;
}

```

Time and Space Complexity

The time complexity of the provided algorithm is `O(log n)`, where `n` is the length of the input array `nums`. The algorithm uses a binary search approach, whereby at each step, it halves the search space. This halving continues until the peak element is found, requiring at most `log2(n)` iterations to converge on a single element.

The space complexity of the algorithm is `O(1)` as it only uses a constant amount of extra space. The variables `left`, `right`, and `mid`, along with a few others for storing intermediate results, do not vary with the size of the input array `nums`, ensuring that the space used remains fixed.