Peak Index in a Mountain Array

(Divide-and-Conquer)

Problem Statement

- The **Peak Index in a Mountain Array** refers to the index of the maximum element in a *mountain array*.
- A **mountain array** is an array that satisfies the following properties:
  - It contains at least three elements.
  - There exists an index i (the peak index) such that:
    - arr[0]<arr[1]<...<arr[i −1]<arr[i] (strictly increasing sequence up to the peak).
    - arr[i]>arr[i +1]>...>arr[arr.length−1] (strictly decreasing sequence after the peak).
- Thus, a mountain array has one peak element where the array values first increase and then decrease.
- The **peak index** is the position of this maximum element.

Examples

- For the array [1, 3, 5, 4, 2]:
  - The peak element is 5, and its index is 2.
  - So, the peak index is 2.
- Array: [10, 20, 30, 40, 35, 25, 15, 5]
  - Peak Element: 40
  - Peak Index: 3
- Array: [0, 10, 5, 2]
  - Peak Element: 10
  - Peak Index: 1

Brute force approach

Brute force approach

- Initialize a variable to keep track of the maximum value found so far (max_value) and its corresponding index (peak_index).
- Traverse the array from left to right:Compare each element with max_value.
- If the current element is greater than max_value, update max_value and set peak_index to the current index.
- After traversing the entire array, the peak_index will hold the index of the peak element
- Time Complexity: O(n)

Divide-and-Conquer approach

DAC Approach

- Start with two pointers, $low=0$ and $high=arr.length-1$
- Compute the mid-point: $mid=(low+high)/2$
- If $arr[mid]<arr[mid+1]$, the peak is in the right half ($low=mid+1$)
- Otherwise, the peak is in the left half ($high=mid$).
- Continue until $low==high$, and that index is the peak index.
- Time Complexity: O(log n)

Pseudocode

```
#include <vector>
class Solution {
public:
int peakIndexInMountainArray(vector<int>& arr) {
        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
        while (left < right) {
           int mid = (left + right) >> 1;
           if (arr[mid] > arr[mid + 1]) {
              right = mid;
           } else {
              left = mid + 1;}}
        return left;}};
```

Example

Example

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

1	3	5	4	2

- 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.

1	3	5	4	2

- Start the while loop since left < right (1 < 3 is true).
- Calculate the midpoint mid using (left + right) >> 1. For our example, it's (1 + 3) >> 1, which equals 2.

1	3	5	4	2

Example

- 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.

1	3	5	4	2

- The loop continues because left < right is still true.
- Recalculate the midpoint with the updated pointers. Now, mid is (1 + 2) >> 1, which equals 1.

1	3	5	4	2

Example

- 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.

		l,r		
1	3	5	4	2

- The loop ends because left equals right, indicating convergence at the peak's index, which is 2 in our example array.
- Return left, which is the peak index. In our array, arr[2] is indeed the peak with a value of 5.