Day 31 / 100 :

Topic - Binary Tree

1 Problem statement: Peak Index in a mountain Array (Medium)

An array arr a mountain if the following properties hold:
arr.length >= 3
There exists some i with 0 < i < arr.length - 1 such that:
arr[0] < arr[1] < ... < arr[i - 1] < arr[i]
arr[i] > arr[i + 1] > ... > arr[arr.length - 1]
Given a mountain array arr, return the index i such that arr[0] < arr[1] < ... < arr[i - 1] < arr[i] > arr[i + 1] > ... > arr[arr.length - 1].
You must solve it in O(log(arr.length)) time complexity.

Example 1:
Input: arr = [0,1,0]
Output: 1

Example 2:
Input: arr = [0,2,1,0]
Output: 1

Solutions :

Approach 1 - Binary Search

Intuition:
We have to find the element which is peak in the whole array.
So we can see that this [0,10,5,2] array is first increasing in a sorted way then starts decreasing which is also in a sorted way.
So by this we can see that we have to find the element which is the maximum and it's also present somewhere in the middle of the array {because first increasing then decreasing so it has to be somewhere in the middle}.

Approach:

- So by the intuation we can be clicked with a approach similer in which we have to find any value that is present in the middle somewhere, Yes! you are right binary search.
- By this we can find out where is the element which is more that out mid or less than our mid. That's the whole game.
- we start using normal binary search and let's think about the if else cases of our binary search.
- now if out arr[mid] is less than arr[mid+1] than we can understand that the array is increasing right now so our moutain element must be somewhere ahead then out current mid, So we update out start= mid+1.
- now if out arr[mid] is more than arr[mid+1] then its deacresing right now so out mountain element must be somewhere behind so we have to pull our end = mid.
- now we are ready with bothh condition so when we find the place where we can go no further {s<e} than we will return start position because starting point is the place which will be peak cause we are updating it when our mid is shorter.
- I hope you got the logic behind it.

Complexity:

Time complexity: O(log(n))

Space complexity: O(1)

```
class Solution {
public:
    int peakIndexInMountainArray(vector<int>& arr) {
        int s = 0;
        int e = arr.size()-1;
        int mid = s + (e-s)/2;
        while(s<e){
            if(arr[mid] < arr[mid+1]){
                s = mid+1;
            }
            else{
                e = mid;
```

```
            }
            mid = s + (e-s)/2;
        }
        return s;
    }
};
```

1 Problem statement: [Find 1st and last element in sorted array](#) (Medium)

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.
If target is not found in the array, return [-1, -1].
You must write an algorithm with O(log n) runtime complexity.

Example 1:
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]

Example 2:
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]

Solutions :

Approach 1 - Binary Search

Explanation:

1. Use a variable to store the temporary result during the search(let's call it ans). Every time you hit a not found condition AND if there is a possibility of one of the mid or its neighbours being the desired answer store it.

Reason : storing can't harm you if its constant space, the later points will show you how it reduces a burden while writing binary search. Also it ensures an answer.

2. Whenever you hit not found condition do s = mid+1 or e = mid-1 according to the condition without worrying.

Reason : You have stored your value if the loop ends, can't go wrong. This also ensures you don't go into infinite loops.

3. Default to writing mid = s + (e-s)/2 and NOT mid = (s+e)/2.

Reason : The later can result in an overflow when you are not searching in an array but rather on an answer space, specially because binary search if it can be applied is a great tool for huge numbers.

4. Write your if and else well, and don't worry about adding extra conditions with the neighbours, at the same time don't forget to check whether the neighbour exists.

Reason : This is the only thing you should be worrying about coding right, because this depends on the problem. If it is a easy or medium, sometimes even medium-hard, this part shouldn't cause much trouble.

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int
target) {
        vector<int> ans ={-1,-1};
        // Find the starting position
        int left = 0, right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] < target) {
                left = mid + 1;
            } else if (nums[mid] >= target) {
                right = mid - 1;
            }
        }
```

```
        if (left < nums.size() && nums[left] == target)
{
            ans[0] = left;
        }

        // Find the ending position
        left = 0;
        right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] <= target) {
                left = mid + 1;
            } else if (nums[mid] > target) {
                right = mid - 1;
            }
        }

        if (right >= 0 && nums[right] == target) {
            ans[1] = right;
        }

        return ans;
    }
};
```