Q1. Find the first and last occurance of an element in a sorted array.

Approach: Finding First Occurance.

→ Applying binary search, if at any instance mid is such that arr[mid] == key, then:

① Either this is the first occurance itself.
② Or there are more `key` towards the left of mid.

So we'll temporarily store mid and then search in the left half by making high = mid - 1.

If arr[mid] < key, then search in right half by making low = mid + 1.

If arr[mid] > key, then search in left half by making high = mid - 1.

Example: arr[] = {1, 1, 2, 3, 4, 4, 4, 4}   key = 4.

① arr[] = {1, 1, 2, 3, 4, 4, 4, 4}
            ↑   ↑   ↑
            ℓ   mid   h

arr[mid] < key ⇒ Definitely in the right half.

low = mid + 1.   // low = 4

② arr[] = {1, 1, 2, 3, 4, 4, 4, 4}
                     ↑ ↑   ↑
                     ℓ mid   h

arr[mid] == key ⇒ Either mid or before mid.
store mid in a variable ans.   // ans = 5.
Then make high = mid - 1.   // high = 4

                        mid
                         ↓
③ arr[] = {1, 1, 2, 3, 4, 4, 4, 4}

③ arr[] = {1, 1, 2, 3, 4, 4, 4, 4}

mid ↓
l↑ h↑

arr[mid] == key ⟹ Either mid or before mid.
store mid in a variable ans. // ans = 4
Then make high = mid - 1. // high = 3

④ arr[] = {1, 1, 2, 3, 4, 4, 4, 4}

h↑ l↑

low > high ⟹ break out. Last value of $\boxed{ans = 4}$

Note: low, mid, high and ans are storing the indices and
not values.

Finding Last Occurance.

→ Applying binary search, if at any instance mid is such
that arr[mid] == key, then :

① Either this is the last occurance itself.
② Or there are more 'key' towards the right of mid.

So we'll temporarily store mid and then search in the
right half by making low = mid+1.

If arr[mid] < key, then search in right half. by making
low = mid + 1.

If arr[mid] > key, then search in left half. by making
high = mid - 1.

Example: arr[] = {1, 1, 2, 3, 4, 4, 4, 4}   key = 4.

① arr[] = {1, 1, 2, 3, 4, 4, 4, 4}

l↑   mid↑   h↑

arr[mid] < key ⇒ Definitely in the right half.

low = mid + 1.   // low = 4

② arr[] = {1, 1, 2, 3, 4, 4, 4, 4}
                     ↑ ↑   ↑
                     l mid   h

arr[mid] == key ⇒ Either mid or after mid.
store mid in a variable ans.   // ans = 5.
Then make low = mid+1.   // low = 6

③ arr[] = {1, 1, 2, 3, 4, 4, 4, 4}
                        mid↓
                         ↑   ↑
                         l   h

arr[mid] == key ⇒ Either mid or after mid.
store mid in a variable ans.   // ans = 6
Then make low = mid+1.   // low = 7

④ arr[] = {1, 1, 2, 3, 4, 4, 4, 4}
                             mid
                             ↑↑
                             l h

arr[mid] == key ⇒ Either the last occurance is at index mid
or to the right of mid.
Store mid in a variable ans.   // ans = 7
Then make low = mid+1.

⑤ arr[] = {1, 1, 2, 3, 4, 4, 4, 4}
                            ↑ ↑
                            h l

low > high ⇒ break out. Last value of ans = 7

Code:

```
int firstOcc(int arr[], int n, int key) {

    int s = 0, e = n-1;
    int mid = s + (e-s)/2;
    int ans = -1;
    while(s<=e) {

        if(arr[mid] == key){
            ans = mid;
            e = mid - 1;
        }
        else if(key > arr[mid]) {//Right me jao
            s = mid + 1;
        }
        else if(key < arr[mid]) {//left me jao
            e = mid - 1;
        }

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

```
int lastOcc(int arr[], int n, int key) {

    int s = 0, e = n-1;
    int mid = s + (e-s)/2;
    int ans = -1;
    while(s<=e) {

        if(arr[mid] == key){
            ans = mid;
            s = mid + 1;
        }
        else if(key > arr[mid]) {//Right me jao
            s = mid + 1;
        }
        else if(key < arr[mid]) {//left me jao
            e = mid - 1;
        }

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

Follow - up question :

Q. Count the number of occurances of an element in a sorted array.

→ (Last Occurance Index) − (Fist Occurance Index) + 1
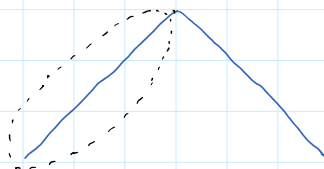
Q. Peak in mountain array.

Eg: arr[] : {0, 10, 5, 2}   ⟶
     ans = 10.

Approach: If for `mid` index,

1. arr[mid] < arr[mid+1] :
This element lies in the increasing slope / ascend. Search in the right half. ⟹ low = mid + 1
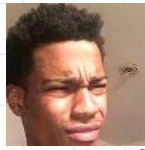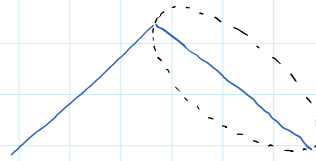
2. In all other cases.
This element lies in the decreasing slope / descend. or is the peak element itself. Since mid can be the peak element, search in the left half including mid.

element, search in the left half including mid.

$\Rightarrow$ high = mid.

Confused?

Example: arr[] : {1, 10, 5, 2, 0}

① low = 0, high = 4, mid = 2

   arr[] : {1, 10, 5, 2, 0}
             l   mid   h

   arr[mid] is not smaller than arr[mid+1]. $\Rightarrow$ high = mid

② low = 0, high = 2, mid = 1

   arr[] : {1, 10, 5, 2, 0}
             l mid h

   arr[mid] is not smaller than arr[mid+1]. $\Rightarrow$ high = mid

③ low = 0, high = 1, mid = 0
                mid
   arr[] : {1, 10, 5, 2, 0}
             l   h

   arr[mid] is smaller than arr[mid+1] $\Rightarrow$ low = mid+1.

④ low = 1, high = 1, mid = 1
                mid
   arr[] : {1, 10, 5, 2, 0}
              l
               h

Do this till ⌈ low < high ⌉

Code:

```cpp
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;
}
```

Homework: Find pivot in a rotated sorted array.

Discussed in the next Lecture. 😛