1. Given a **1-indexed** array of integers numbers that are already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 < numbers.length.

Return *the indices of the two numbers,* index1, *and* index2*,* ***added by one*** *as an integer array* [index1, index2] *of length 2.*

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

Your solution must use only constant extra space.

Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].

```
int twosum(vector<int> &nums, int target) {
    int n=nums.size();
    vector<int> ans(2,-1);
    for(int i=0;i<n;i++) {
        for(int j=i+1;j<n;i++) {
            if(nums[i]+nums[j]==target) {
                ans[0]=nums[i];
                ans[1]=nums[j];
                break;
            }
        }
    }
    return ans;
}
```

Time Complexity - O(n^2)
Space Complexity - O(2)

---

2. Given an array of integer nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If the target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

```
int find(vector<int> &nums, int target) {

    vector<int> ans={-1,-1};

        if(nums.size()!=0 && binary_search(nums.begin(),nums.end(),target)) {

            ans[0]=lower_bound(nums.begin(),nums.end(),target)-nums.begin();

            ans[1]=upper_bound(nums.begin(),nums.end(),target)-nums.begin()-1;

        }

    return ans;

}
```

Time Complexity - O(logn)

Space Complexity - O(2)

---

3. A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in O(log n) time.

Input: nums = [1,2,3,1]

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

```
int peakelement(vector<int> &nums) {

    int mid, low=0, high=nums.size()-1;

    while(low<high) {

        mid=low+(high-low)/2;
```

```
        if(nums[mid]>nums[mid+1]) {
            high=mid;
        }
        else {
            low=mid+1;
        }
    }
    return low;
}
```

Time Complexity - O(logn)

Space Complexity - O(1)

---

4. Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Input: nums = [1,3,5,6], target = 5
Output: 2

Input: nums = [1,3,5,6], target = 7
Output: 4

```
int position(vector<int> &nums, int element) {
    return(lower_bound(nums.begin(), nums.end(), element)- nums.begin());
}
```

Time Complexity - O(logn)
Space Complexity - O(1)

---

5. Find the majority element in the array. A **majority element** in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Input: A[]={3, 3, 4, 2, 4, 4, 2, 4, 4}

Output: 4
Explanation: The frequency of 4 is 5 which is greater than half of the size of the array size.

```
int majiorityelement(vector<int> &nums) {

    int check = nums.size()/2;

    unordered_map<int, int> mp;

    for(int i: nums) {

        mp[i]++;

        if(mp[i]>check) {

            return i;

        }

    }

    return -1;

}
```

Time Complexity - O(n)

Space Complexity - O(n)

---

6. You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which returns whether the version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Input: n = 5, bad = 4

Output: 4

Explanation:

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

```
int firstbadversion(int n) {

    int mid, low=1, high=n;

    while(low<high) {

        mid=low+(high-low)/2;

        if(isBadVersion(mid)) {

            high=mid;

        }

        else {

            low=mid+1;

        }

    }

    return low;

}
```

Time Complexity - O(logn)

Space Complexity - O(1)

---

7. Given an array of integers, find the inversion of an array. Formally, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

N=5, arr[] = {2, 4, 1, 3, 5}

Output: 3

Explanation: (2,1) (4,1) and (4,3) forms an inversion in an array

```
int countinversions(vector<int> &nums, int n) {

    int ans=0;

    for(int i=0;i<n;i++) {
```

```
        for(int j=i+1;j<n;j++) {
            if(nums[i]>nums[j]) {
                ++ans;
            }
        }
    }
    return ans;
}
```

Time Complexity - O(n^2)

Space Complexity - O(1)

---

8. Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

ar1[] = {1, 5, 10, 20, 40, 80}

ar2[] = {6, 7, 20, 80, 100}

ar3[] = {3, 4, 15, 20, 30, 70, 80, 120}

Output: 20, 80

```
int findcommon(vector<int> &a, vector<int> &b, vector<int> &c) {
    int n1=a.size(), n2=b.size(), n3=c.size();
    vector<int> ans;
    for(int i=0;i<n1;i++) {
        if(i!=0 && a[i]==a[i-1]) {
            continue;
        }
        if(binary_search(b.begin(),b.end(), a[i]) && binary_search(c.begin(), c.end(),
  a[i])) {
            ans.push_back(a[i]);
        }
    }
```

```
    return ans;

}
```

Time Complexity - O(n*logn)

Space Complexity - O(min(n1,n2,n3))
