Binary Search

Difficulty: **Easy** Accuracy: **44.32%** Submissions: **530K+** Points: **2**

Given a sorted array **arr** and an integer **k**, find the position(0-based indexing) at which k is present in the array using binary search.

Note: If multiple occurrences are there, please return the smallest index.

Examples:

Input: arr[] = [1, 2, 3, 4, 5], k = 4
Output: 3
Explanation: 4 appears at index 3.

Input: arr[] = [11, 22, 33, 44, 55], k = 445
Output: -1
Explanation: 445 is not present.

Code:

```cpp
int binarysearch(vector<int> &arr, int k) {
        int start=0,end=arr.size()-1,ans=-1;
        while(start<=end){
            int mid=(start+end)/2;
            if(arr[mid]==k){
                ans=mid;
                break;
            }
            if(arr[mid]<k){
                start=mid+1;
            }
            else{
                end=mid-1;
            }
        }
        return ans;
}
```

TimeComplexity:O(log n)
SpaceComplexity:O(1)

Next Greater Element

Difficulty: **Medium** Accuracy: **32.95%** Submissions: **410K+** Points: **4**

Given an array **arr[ ]** of integers, the task is to find the next greater element for each element of the array in order of their appearance in the array. Next greater element of an element in the array is the nearest element on the right which is greater than the current element.
If there does not exist next greater of current element, then next greater element for current element is -1. For example, next greater of the last element is always -1.

Examples

Input: arr[] = [1, 3, 2, 4]
Output: [3, 4, 4, -1]
Explanation: The next larger element to 1 is 3, 3 is 4, 2 is 4 and for 4, since it doesn't exist, it is -1.

Input: arr[] = [6, 8, 0, 1, 3]
Output: [8, -1, 1, 3, -1]
Explanation: The next larger element to 6 is 8, for 8 there is no larger elements hence it is -1, for 0 it is 1 , for 1 it is 3 and then for 3 there is no larger element on right and hence -1.

Code:

```
vector<int> nextLargerElement(vector<int>& arr) {
        stack<int> st;
        vector<int> ans(arr.size(),-1);
        st.push(arr[arr.size()-1]);
        for(int i=arr.size()-2;i>=0;i--){
            while(!st.empty() and st.top()<=arr[i]){
                st.pop();
            }
            if(!st.empty()){
                ans[i]=st.top();
            }
            st.push(arr[i]);
        }
        return ans;
}
```

TimeComplexity:O(n)
SpaceComplexity:O(n)

Union of Two Arrays with Duplicate Elements

Difficulty: **Easy** Accuracy: **42.22%** Submissions: **387K+** Points: **2**

Given two arrays **a[]** and **b[]**, the task is to find the number of elements in the union between these two arrays.

The Union of the two arrays can be defined as the set containing distinct elements from both arrays. If there are repetitions, then only one element occurrence should be there in the union.

Note: Elements are not necessarily distinct.

Examples

Input: a[] = [1, 2, 3, 4, 5], b[] = [1, 2, 3]
Output: 5
Explanation: 1, 2, 3, 4 and 5 are the elements which comes in the union setof both arrays. So count is 5.

Input: a[] = [85, 25, 1, 32, 54, 6], b[] = [85, 2]
Output: 7
Explanation: 85, 25, 1, 32, 54, 6, and 2 are the elements which comes in the union set of both arrays. So count is 7.

Code:

```cpp
int findUnion(vector<int>& a, vector<int>& b) {
        unordered_map<int,int> hash;
        for(auto t:a){
            hash[t]++;
        }
        for(auto t:b){
            hash[t]++;
        }
        return hash.size();
}
```

TimeComplexity:O(n)
SpaceComplexity:O(n)

Parenthesis Checker

Difficulty: **Easy** Accuracy: **28.56%** Submissions: **617K+** Points: **2**

You are given a string **s** representing an expression containing various types of brackets: {}, (), and []. Your task is to determine whether the brackets in the expression are balanced. A balanced expression is one where every opening bracket has a corresponding closing bracket in the correct order.

Examples :

Input: s = "{([])}"
Output: true
Explanation:
- In this expression, every opening bracket has a corresponding closing bracket.
- The first bracket { is closed by }, the second opening bracket (is closed by), and the third opening bracket [is closed by].
- As all brackets are properly paired and closed in the correct order, the expression is considered balanced.

Code:

```
bool isParenthesisBalanced(string& s) {
        stack<char> st;
        for(int i=0;i<s.size();i++){
            if(s[i]=='{' || s[i]=='(' || s[i]=='['){
                st.push(s[i]);
            }
            else{
                if(st.empty())return false;
                if(s[i]==']' and st.top()!='['){
                    return false;
                }
                else if(s[i]=='}' and st.top()!='{'){
                    return false;
                }
                else if(s[i]==')' and st.top()!='('){
                    return false;
                }
                st.pop();
            }
        }
        if(!st.empty())return false;
        return true;
}
```

TimeComplexity:O(n)
SpaceComplexity:O(n)

Equilibrium Point

Difficulty: **Easy** Accuracy: **28.13%** Submissions: **593K+** Points: **2**

Given an array **arr** of non-negative numbers. The task is to find the first **equilibrium point** in an array. The equilibrium point in an array is an index (or position) such that the sum of all elements before that index is the same as the sum of elements after it.

Note: Return equilibrium point in 1-based indexing. Return -1 if no such point exists.

Examples:

Input: arr[] = [1, 3, 5, 2, 2]
Output: 3
Explanation: The equilibrium point is at position 3 as the sum of elements before it (1+3) = sum of elements after it (2+2).

Code:

```
int equilibriumPoint(vector<int> &arr){
        int n=arr.size();
        if(n==1)return n;
        vector<int> StartSum(n,0),EndSum(n,0);
        int start=0,end=0;
        for(int i=0;i<n;i++){
            start+=arr[i];
            end+=arr[n-i-1];
            StartSum[i]=start;
            EndSum[n-i-1]=end;
        }
        for(int i=0;i<n;i++){
            if(i==0){
                if(EndSum[i+1]==0){
                    return i+1;
                }
            }
            else if(i==n-1){
                if(StartSum[n-2]==0){
                    return i+1;
                }
            }
            else{
                if(StartSum[i-1]==EndSum[i+1]){
                    return i+1;
                }
            }
        }
        return -1;
}
```

TimeComplexity:O(n)
SpaceComplexity:O(n)

Minimize the Heights II

Difficulty: **Medium** Accuracy: **15.06%** Submissions: **620K+** Points: **4**

Given an array **arr[]** denoting heights of **N** towers and a positive integer **K.**

For **each** tower, you must perform **exactly one** of the following operations **exactly once**.

- **Increase** the height of the tower by **K**
- **Decrease** the height of the tower by **K**

Find out the **minimum** possible difference between the height of the shortest and tallest towers after you have modified each tower.

You can find a slight modification of the problem here.
Note: It is **compulsory** to increase or decrease the height by K for each tower. **After** the operation, the resultant array should **not** contain any **negative integers**.

Examples :

Input: k = 2, arr[] = {1, 5, 8, 10}
Output: 5
Explanation: The array can be modified as {1+k, 5-k, 8-k, 10-k} = {3, 3, 6, 8}.The difference between the largest and the smallest is 8-3 = 5.

Code:

```
int getMinDiff(vector<int> &arr, int k) {
        sort(arr.begin(),arr.end());
        int n=arr.size();
        int diff=arr[n-1]-arr[0];
        int mini=0,maxi=0;
        for(int i=0;i<n-1;i++){
            maxi=max(arr[n-1]-k,arr[i]+k);
            mini=min(arr[0]+k,arr[i+1]-k);
            if(mini>=0){
                diff=min(diff,maxi-mini);
            }
        }
        return diff;
}
```

TimeComplexity:O(n+logn)
SpaceComplexity:O(1)

Kth Smallest

Difficulty: **Medium** Accuracy: **35.17%** Submissions: **654K+** Points: **4**

Given an array **arr[]** and an integer **k** where k is smaller than the size of the array, the task is to find the **k^{th} smallest** element in the given array.

Follow up: Don't solve it using the inbuilt sort function.

Examples :

Input: arr[] = [7, 10, 4, 3, 20, 15], k = 3
Output: 7
Explanation: 3rd smallest element in the given array is 7.

Input: arr[] = [2, 3, 1, 20, 15], k = 4
Output: 15
Explanation: 4th smallest element in the given array is 15.

Code:

```
int kthSmallest(vector<int> &arr, int k) {
        priority_queue<int> pq;
        for(int i=0;i<k;i++){
            pq.push(arr[i]);
        }
        for(int i=k;i<arr.size();i++){
            int num=pq.top();
            pq.pop();
            if(arr[i]<num){
                pq.push(arr[i]);
            }
            else{
                pq.push(num);
            }
        }
        return pq.top();
}
```

TimeComplexity:O(nlogn)
SpaceComplexity:O(n)