

## 1.Kth Smallest

Given an array `arr[]` and an integer `k` where `k` is smaller than the size of the array, the task is to find the  $k^{\text{th}}$  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.

```
class Solution {
public:
    // arr : given array
    // k : find kth smallest element and return using this function
    int kthSmallest(vector<int> &arr, int k) {
        int maxval=0;
        for(int i:arr) maxval=max(maxval+1,i);
        vector<int> res(maxval,0);
        for(int i:arr) res[i]+=1;
        for(int i=0;i<=maxval;i++){
            k-=res[i];
            if(k<=0) return i;
        }
        return maxval;
    }
};
```

### Compilation Completed

For Input:  

7 10 4 3 20 15

3

Your Output:

7

Expected Output:

7

**Time Complexity:**  $O(n+(\text{max\_element}))$

**Auxiliary Space:**  $O(\text{max\_element})$

## 2.Binary Search

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.

```
class Solution {
public:
    int binarysearch(vector<int> &arr, int k) {
        int idx=1e9;
        int mid;
        int n=arr.size();
        int left=0,right=n-1;
        while(left<=right){
            mid=(left+right)/2;
            if(arr[mid]==k) idx=min(idx,mid);
            if(arr[mid]<k) left=mid+1;
            else right=mid-1;
        }
        return idx==1e9? -1:idx;
    }
};
```

### Compilation Completed

For Input:  

4

1 2 3 4 5

Your Output:

3

Expected Output:

3

**Time Complexity :**  $O(\log n)$

**Space Complexity :**  $O(1)$

### 3.Parenthesis Checker

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.

**Input:** *s* = "()"

**Output:** true

**Explanation:**

- This expression contains only one type of bracket, the parentheses ( and ).
- The opening bracket ( is matched with its corresponding closing bracket ).
- Since they form a complete pair, the expression is balanced.

**Input:** *s* = "(["


**Output:** false

**Explanation:**

- This expression contains only one type of bracket, the parentheses ( and ).
- The opening bracket ( is matched with its corresponding closing bracket ).
- Since they form a complete pair, the expression is balanced.

```
class Solution {
public:
    bool isParenthesisBalanced(string& s) {
        // code here
        stack<char> stack;
        for(auto i:s){
            if(!stack.empty() && i=='}' && stack.top()=='{') stack.pop();
            else if(!stack.empty() && i==')' && stack.top()=='(') stack.pop();
            else if(!stack.empty() && i==']' && stack.top()=='[') stack.pop();
            else stack.push(i);
        }
        return stack.empty();
    }
};
```

## Compilation Completed

For Input:  

{{[]}}

Your Output:

true

Expected Output:

true

**Time Complexity :**  $O(n)$

**Space Complexity :**  $O(n)$

#### 4.Next Greater Element

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.

**Input:** `arr[] = [10, 20, 30, 50]`

**Output:** `[20, 30, 50, -1]`

**Explanation:** For a sorted array, the next element is next greater element also except for the last element.



**Input:** `arr[] = [50, 40, 30, 10]`

**Output:** `[-1, -1, -1, -1]`

**Explanation:** There is no greater element for any of the elements in the array, so all are -1.

```
class Solution {
public:
    // Function to find the next greater element for each element
    vector<int> nextLargerElement(vector<int>& arr) {
        int len=arr.size();
        stack<pair<int,int>> stack;
        for(int i=0;i<len;i++){
            while(!stack.empty() && stack.top().first<arr[i])
                auto p=stack.top();
                stack.pop();
                arr[p.second]=arr[i];
            }
            stack.push({arr[i],i});
        }
        while(!stack.empty()){
            arr[stack.top().second]=-1;
            stack.pop();
        }
        return arr;
    }
};
```

## Compilation Completed

For Input:  

1 3 2 4

Your Output:

3 4 4 -1

Expected Output:

3 4 4 -1

**Time Complexity :**  $O(n)$

**Space Complexity :**  $O(n)$

## 5. Minimize the Heights II

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  $8-3 = 5$ .



**Input:** `k = 3, arr[] = {3, 9, 12, 16, 20}`

**Output:** 11

**Explanation:** The array can be modified as  $\{3+k, 9+k, 12-k, 16-k, 20-k\} \rightarrow \{6, 12, 9, 13, 17\}$ . The difference between the largest and the smallest is  $17-6 = 11$ .

```
class Solution {
public:
    int getMinDiff(vector<int>& arr, int k) {
        int n = arr.size();
        if (n == 1) return 0;
        sort(arr.begin(), arr.end());
        int result = arr[n - 1] - arr[0];
        int smallest = arr[0] + k;
        int largest = arr[n - 1] - k;
        for (int i = 0; i < n - 1; i++) {
            int minHeight = min(smallest, arr[i + 1] - k);
            int maxHeight = max(largest, arr[i] + k);
            if (minHeight < 0) continue;
            result = min(result, maxHeight - minHeight);
        }
        return result;
    }
};
```

## Compilation Completed

For Input:  

2

1 5 8 10

Your Output:

5

Expected Output:

5

**Time Complexity :**  $O(n)$

**Space Complexity :**  $O(1)$



## 6. Equilibrium Point

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).

**Input:** arr[] = [1]

**Output:** 1

**Explanation:** Since there's only one element hence it's only the equilibrium point.


**Input:** arr[] = [1, 2, 3]

**Output:** -1

**Explanation:** There is no equilibrium point in the given array.

```
// } Driver Code Ends
class Solution {
public:
    // Function to find equilibrium point in the array.
    int equilibriumPoint(vector<int> &arr) {
        int n=arr.size();
        vector<int> presum(n+1,0);
        presum[0]=0;
        for(int i=1;i<=n;i++){
            presum[i]=arr[i-1]+presum[i-1];
        }
        for(int i=1;i<=n;i++){
            if(presum[i-1]==(presum[n]-presum[i])) return i;
        }
        return -1;
    }
};
// } Driver Code Ends
```

## Compilation Completed

For Input:  

1 3 5 2 2

Your Output:

3

Expected Output:

3

**Time Complexity :**  $O(n)$

**Space Complexity :**  $O(n)$