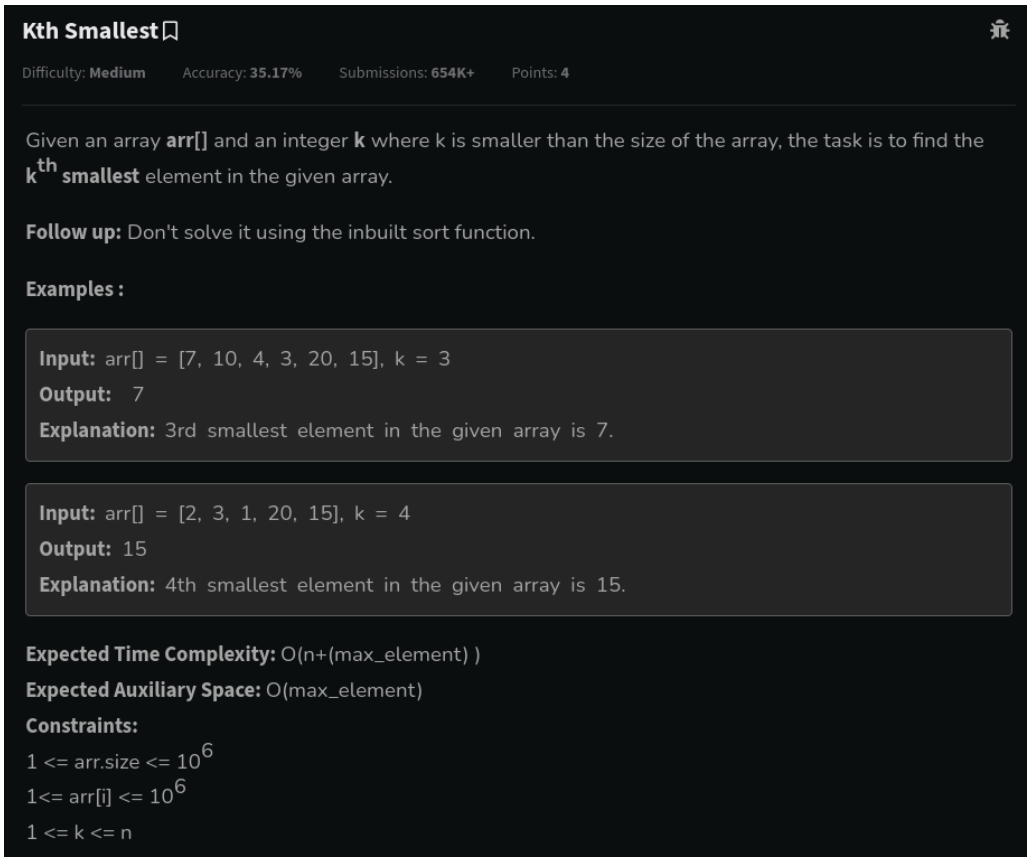
1. Kth Smallest Element



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.

Expected Time Complexity: O(n+(max_element))
Expected Auxiliary Space: O(max_element)
Constraints:
$1 <= arr.size <= 10^6$
$1 <= arr[i] <= 10^6$
$1 <= k <= n$

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

int kthSmallest(vector<int>& arr, int k) {

    priority_queue<int> maxHeap;
    for (int num : arr) {
        maxHeap.push(num);

        if (maxHeap.size() > k) {
            maxHeap.pop();
        }
    }

    return maxHeap.top();
}
```

Time Complexity : O(n)

Space Complexity : O(n)

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.

Note: Try to solve this problem in constant space i.e O(1)

Constraints:
$1 <= arr.size() <= 10^5$
$1 <= arr[i] <= 10^6$
$1 <= k <= 10^6$

```cpp
#include <bits/stdc++.h>
using namespace std;


class Solution {
  public:
    int binarysearch(vector<int> &arr, int k) {

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

        while(low<=high){
            int mid = low+(high-low)/2;

            if(arr[mid] == k){
                return mid;
            }
            else if(arr[mid]<=k){
                low = mid+1;
            }
            else{
                high = mid-1;
            }
        }

        return -1;
    }
};
```

Time Complexity : O(log n)

Space Complexity : O(1)

Paranthesis Checker

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.

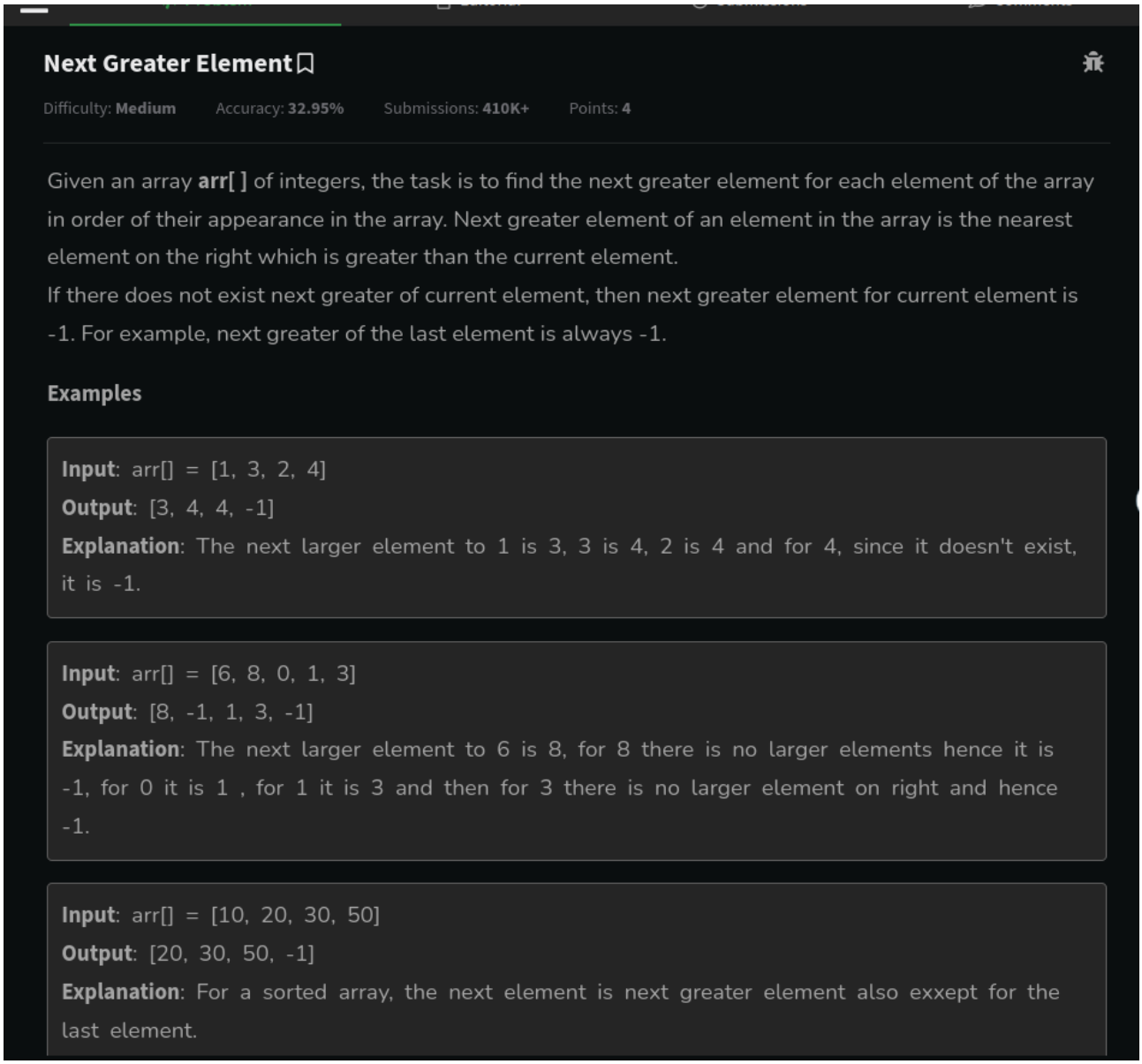
Input: s = "()"
Output: true
Explanation:
- This expression contains only one type of bracket, the parentheses (and).
- The opening bracket (is matched with its corresponding closing bracket).
- Since they form a complete pair, the expression is balanced.

```cpp
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:
    bool isParenthesisBalanced(string& str) {
        stack<char>st;
        for(char ch : str){
            if(ch == '{' || ch == '[' ||  ch == '('){
                st.push(ch);
            }
            else{
                if(st.empty()) {
                    return false;
                }
                else if(ch == ')' && st.top() == '('){
                    st.pop();
                }
                else if(ch == '}' && st.top() == '{'){
                    st.pop();
                }
                else if(ch == ']' && st.top() == '['){
                    st.pop();
                }
                else{
                    return false;
                }
            }
        }
        return st.empty();
    }
};
```

Time Complexity : O(n);

Space Complexity : O(n)

Next Greator Element



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.

Input: arr[] = [10, 20, 30, 50]
Output: [20, 30, 50, -1]
Explanation: For a sorted array, the next element is next greater element also exxept for the last element.

```cpp
void NGE(int arr[], int n) {
    stack<int> s;
    s.push(arr[0]);

    for (int i = 1; i < n; i++) {
        while (!s.empty() && s.top() < arr[i]) {
            cout << s.top() << " --> " << arr[i] << endl;
            s.pop();
        }
        s.push(arr[i]);
    }

    while (!s.empty()) {
        cout << s.top() << " --> " << -1 << endl;
        s.pop();
    }
}

int main() {

    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    NGE(arr, n);
    return 0;
}
```

Time Complexity: O(n)

Space Complexity: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.

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} -> {6, 12, 9, 13, 17}.The difference between the largest and the smallest is 17-6 = 11.

```cpp
int getMinDiff(vector<int>& arr, int k){
    int n = arr.size();
    if (n == 1) return 0;

    sort(arr.begin(), arr.end());
    int ans = arr[n - 1] - arr[0];
    int smallest = arr[0] + k, largest = arr[n - 1] - k;
    for (int i = 0; i < n - 1; i++) {
    int minH = min(smallest, arr[i + 1] - k);
    int maxH = max(largest, arr[i] + k);
    if (minH < 0) continue;
        ans = min(ans, maxH - minH);
    }

    return ans;
}
```

Time Complexity : O(nlogn)
Space complexity : O(1)

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).

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.

Expected Time Complexity: O(n)
Expected Auxiliary Space: O(1)

```cpp
#include <bits/stdc++.h>
using namespace std;
class Solution {
  public:
    int equilibriumPoint(vector<int> &arr) {
        int n = arr.size();

        if(n==1) return 1;

        int sum =0;
        int left_sum =0;

        for(int num:arr){ sum+=num;}

        for(int i=0;i<n;i++){
            int right_sum = sum-left_sum -arr[i];
            if (left_sum == right_sum){
                return i+1;
            }
            left_sum +=arr[i];
        }

        return -1;
    }
};
```

Time Complexity : O(n)

Space Complexity : O(1)