Naresh M\_AI&DS\_Day-4\_Practice

**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 **kth 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.

**Program:**

class Solution {

public static int kthSmallest(int[] arr, int k) {

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

for (int num : arr) {

minHeap.add(num);

}

for (int i = 0; i < k - 1; i++) {

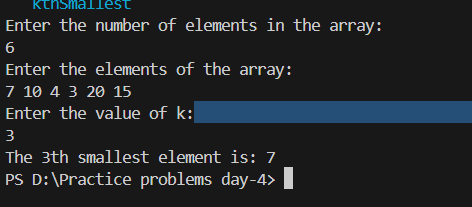
minHeap.poll();

}

return minHeap.peek();

}

}

  
**Time complexity: O(klogn)**

**2. Parentheses 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

Input: s = "()"

Output: true

Input: s = "([]"

Output: false

Constraints:  
1 ≤ s.size() ≤ 106s[i] ∈ {'{', '}', '(', ')', '[', ']'}

**Program:**

class Solution {

// Function to check if brackets are balanced or not.

static boolean isParenthesisBalanced(String s) {

if(s.length()%2!=0){

return false;

}

Stack<Character> stack=new Stack<>();

for(char c:s.toCharArray()){

if(c=='(' || c=='[' || c=='{'){

stack.push(c);

}else if (c==')'&& !stack.isEmpty() && stack.peek()=='('){

stack.pop();

}else if (c==']'&& !stack.isEmpty() && stack.peek()=='['){

stack.pop();

}else if (c=='}'&& !stack.isEmpty() && stack.peek()=='{'){

stack.pop();

}else{

return false;

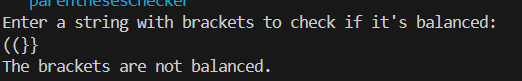
}

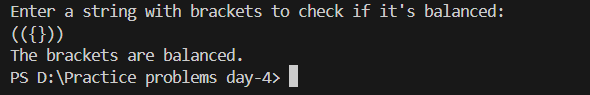
}

return stack.isEmpty();

}

}



****

**Time complexity: O(N)**

**3.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.

**Program:**

class Solution {

public int binarysearch(int[] arr, int k) {

int left = 0;

int right = arr.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == k) {

return mid;

}

if (arr[mid] > k) {

right = mid - 1;

} else {

left = mid + 1;

}

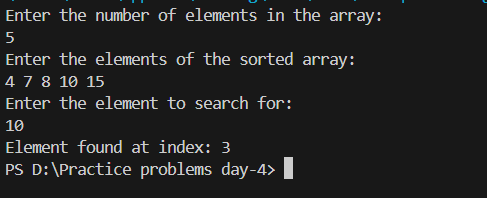
}

return -1;

}

}

**Time Complexity: O(1)**

****

**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.

**Program:**

class Solution {

// Function to find the next greater element for each element of the array.

public ArrayList<Integer> nextLargerElement(int[] arr) {

int n = arr.length;

ArrayList<Integer> l = new ArrayList<>();

int[] nge = new int[n];

Stack<Integer> stack = new Stack<>();

for (int i = n - 1; i >= 0; i--) {

while (!stack.isEmpty() && stack.peek() <= arr[i]) {

stack.pop();

}

nge[i] = stack.isEmpty() ? -1 : stack.peek();

stack.push(arr[i]);

}

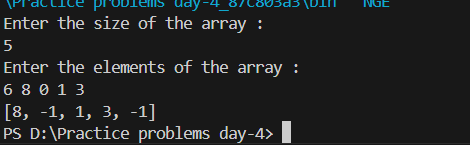
for (int i = 0; i < n; i++) {

l.add(nge[i]);

}

return l;

}

}

**Time complexity: O(n)**

**5.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).

**Program**:

class Solution {

// Function to find equilibrium point in the array.

public static int equilibriumPoint(int arr[]) {

int totalSum = 0;

for (int num : arr) {

totalSum += num;

}

int leftSum = 0;

for (int i = 0; i < arr.length; i++) {

totalSum -= arr[i];

if (leftSum == totalSum) {

return i + 1;

}

leftSum += arr[i];

}

return -1;

}

}



**Time complexity : O(n)**

**6.Union of two Arrays with Duplicates**

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.

**Program:**

class Solution {

public static int findUnion(int a[], int b[]) {

HashSet<Integer> unionSet = new HashSet<>();

for (int num : a) {

unionSet.add(num);

}

for (int num : b) {

unionSet.add(num);

}

return unionSet.size();

}

}



**Time complexity: O(n+m)**

**7. 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](https://practice.geeksforgeeks.org/problems/minimize-the-heights-i/1/).  
**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.

**Expected Time Complexity:** O(n\*logn)  
**Expected Auxiliary Space:** O(n)

**Program:**

class Solution {

int getMinDiff(int[] arr, int k) {

int n=arr.length;

if (n == 1) {

return 0;

}

Arrays.sort(arr);

int initialDiff = arr[n - 1] - arr[0];

int minHeight = arr[0] + k;

int maxHeight = arr[n - 1] - k;

int minDiff = initialDiff;

for (int i = 1; i < n; i++) {

if (arr[i] - k < 0) {

continue;

}

int minCandidate = Math.min(minHeight, arr[i] - k);

int maxCandidate = Math.max(maxHeight, arr[i - 1] + k);

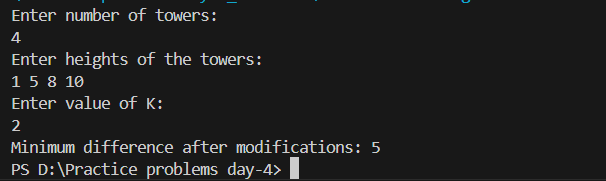
minDiff = Math.min(minDiff, maxCandidate - minCandidate);

}

return minDiff;

}

}



**Time complexity: O(nlogn)**