**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.

Examples :

Input: arr[] = [7, 10, 4, 3, 20, 15], k = 3

Output: 7

Explanation: 3rd smallest element in the given array is 7.

**Code**

class Solution {

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

PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

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

maxHeap.add(arr[i]);

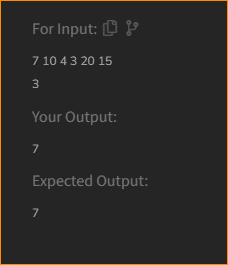
if (maxHeap.size() > k) {

maxHeap.poll();

}}

return maxHeap.peek();}}

**Output**



**Time Complexity**

O( n logk )

**2. 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.

**Code**

class Solution {

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

int n = arr.length;

Arrays.sort(arr);

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

int tempmin, tempmax;

tempmin = arr[0];

tempmax = arr[n - 1];

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

if (arr[i] - k < 0)

continue;

tempmin = Math.min(arr[0] + k, arr[i] - k);

tempmax

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

ans = Math.min(ans, tempmax - tempmin);

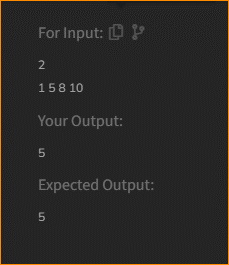
}

return ans;

}

}

**Output**



**Time Complexity**

O ( n logn )

**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**.**

**Code**

class Solution {

static boolean isParenthesisBalanced(String s) {

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

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

char currentChar = s.charAt(i);

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

stack.push(currentChar);

} else if (currentChar == ')' || currentChar == '}' || currentChar == ']') {

if (stack.isEmpty()) {

return false;

}

char top = stack.peek();

if ((currentChar == ')' && top == '(') ||

(currentChar == '}' && top == '{') ||

(currentChar == ']' && top == '[')) {

stack.pop();

} else {

return false;

}

}

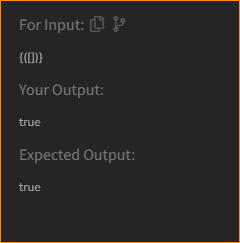
}

return stack.isEmpty();

}

}

**Output**



**Time Complexity**

O(n)

**4. 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).

**Code**

class Solution {

public static int equilibriumPoint(int arr[]) {

int n = arr.length;

long sum = 0;

for (int i = 0; i < n; i++) sum += arr[i];

long sum2 = 0;

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

sum = sum - arr[i];

if (sum2 == sum) {

return (i + 1);

}

sum2 = sum2 + arr[i];

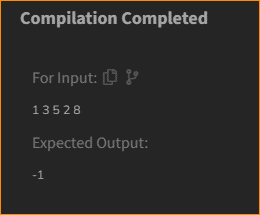
}

return -1;

}

}

**Output**



**Time Complexity**

O(n\*\*2)

**5. 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.

**Code**

class Solution {

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

int low = 0;

int high = arr.length - 1;

while (low <= high) {

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

if (arr[mid] == k) {

return mid;

} else if (k > arr[mid]) {

low = mid + 1;

} else {

high = mid - 1;

}

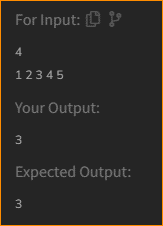
}

return -1;

}

}

**Output**



**Time Complexity**

O( logn )

**6. 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.

**Code**

class Solution {

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

int n = arr.length;

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

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

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

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

stack.pop();

}

if (stack.isEmpty()) {

result.add(-1);

} else {

result.add(stack.peek());

}

stack.push(arr[i]);

}

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

for (int i = result.size() - 1; i >= 0; i--) {

finalResult.add(result.get(i));

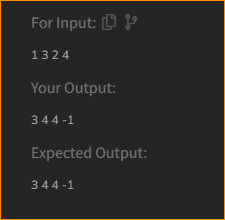
}

return finalResult;

}

}

**Output**



**Time Complexity**

O(n)

**7. Union of 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.

**Code**

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);

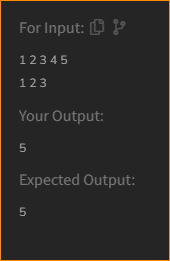
}

return unionSet.size();

}

}

**Output**



**Time Complexity**

O(n+m)