# **Sahana Lakshmipathy\_AI&DS\_DSA\_DAY\_4**

**1.Kth Smallest Element**

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.

**Program:**

import java.util.PriorityQueue;

import java.util.Collections;

class Solution {

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

// Create a max-heap with size k

PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

// Iterate over each element in the array

for (int num : arr) {

// Add the current element to the heap

maxHeap.add(num);

// If the heap size exceeds k, remove the largest element

if (maxHeap.size() > k) {

maxHeap.poll();

}

}

// The root of the max-heap is the k-th smallest element

return maxHeap.peek();

}

public static void main(String[] args) {

int[] arr = {7, 10, 4, 3, 20, 15};

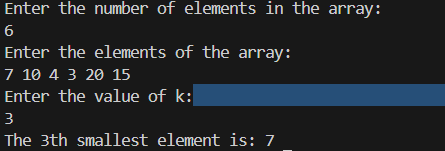
int k = 3;

System.out.println("The " + k + "-th smallest element is " + kthSmallest(arr, k));

}

}

**Time Complexity : O(n)**



**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.

**Program:**

class Solution {  
 int getMinDiff(int[] arr, int k) {  
 int n = arr.length;  
 if (n == 1) return 0;  
  
 // Sort the array to handle heights in increasing order  
 Arrays.sort(arr);  
  
 // Calculate the initial difference between the highest and lowest values  
 int result = arr[n - 1] - arr[0];  
  
 // Initial boundaries after adding and subtracting K  
 int smallest = arr[0] + k;  
 int largest = arr[n - 1] - k;  
  
 // Traverse the sorted array and calculate the min possible difference  
 for (int i = 0; i < n - 1; i++) {  
 // Calculate the new potential minimum and maximum heights after adjustment  
 int minHeight = Math.min(smallest, arr[i + 1] - k);  
 int maxHeight = Math.max(largest, arr[i] + k);  
  
 // If minHeight goes negative, skip as it's invalid  
 if (minHeight < 0) continue;  
  
 // Update the result with the new minimum difference  
 result = Math.min(result, maxHeight - minHeight);  
 }  
  
 return result;  
 }

public static void main(String[] args) {

Solution sol = new Solution();

int[] arr = {1, 5, 8, 10}; int k = 2;

System.out.println("The minimum possible difference is: " + sol.getMinDiff(arr, k));

}   
}

**Time Complexity: O(nlogn)**

**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.

**Program:**

import java.util.Scanner;

import java.util.Stack;

class parenthesesChecker {

    static boolean isParenthesisBalanced(String s) {

        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();

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter a string with brackets to check if it's balanced:");

        String input = scanner.nextLine();

        boolean result = isParenthesisBalanced(input);

        if (result) {

            System.out.println("The brackets are balanced.");

        } else {

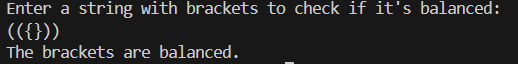
            System.out.println("The brackets are not balanced.");

        }

        scanner.close();

    }

}



**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).

**Program:**

class Solution {

// Function to find equilibrium point in the array.

public static int equilibriumPoint(int arr[]) {

int n = arr.length;

if (n == 1) return 1; // Single element is always the equilibrium point

int totalSum = 0;

for (int num : arr) {

totalSum += num;

}

int leftSum = 0;

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

// Subtract current element from totalSum to get the right sum for index i

totalSum -= arr[i];

// Check if leftSum equals totalSum (right sum)

if (leftSum == totalSum) {

return i + 1; // Return 1-based index

}

// Add current element to leftSum for the next iteration

leftSum += arr[i];

}

return -1; // No equilibrium point found

}

public static void main(String[] args) {

int[] arr1 = {1, 3, 5, 2, 2};

System.out.println(equilibriumPoint(arr1)); // Output: 3

int[] arr2 = {1};

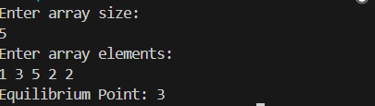
System.out.println(equilibriumPoint(arr2)); // Output: 1

int[] arr3 = {1, 2, 3};

System.out.println(equilibriumPoint(arr3)); // Output: -1

}

}



**Time complexity: O(n)**

**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.

**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; // Target found

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

left = mid + 1; // Search in the right half

} else {

right = mid - 1; // Search in the left half

}

}

return -1; // Target not found

}

public static void main(String[] args) {

Solution sol = new Solution();

int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9};

int k = 5;

System.out.println(sol.binarysearch(arr, k)); // Output: 4 (0-based index)

k = 10;

System.out.println(sol.binarysearch(arr, k)); // Output: -1 (not found)

}

}



**Time complexity:O(1)**

**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.

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:**

import java.util.Stack;

public class NextGreaterElement {

public static int[] nextGreaterElements(int[] nums) {

int n = nums.length;

int[] result = new int[n]; // Array to store results

Stack<Integer> stack = new Stack<>(); // Stack to store indices

// Initialize the result array with -1 (default if no greater element is found)

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

result[i] = -1;

}

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

// While the current number is greater than the number corresponding to the index on the stack top

while (!stack.isEmpty() && nums[i] > nums[stack.peek()]) {

int index = stack.pop();

result[index] = nums[i]; // Update the result for that index

}

// Push the current index onto the stack

stack.push(i);

}

return result;

}

public static void main(String[] args) {

int[] nums = {4, 5, 2, 10, 8};

int[] result = nextGreaterElements(nums);

System.out.print("Next greater elements: ");

for (int res : result) {

System.out.print(res + " ");

}

}

}



**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:**

import java.util.HashSet;

class Solution {

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

// Create a HashSet to store unique elements

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

// Add all elements of array a to the set

for (int num : a) {

set.add(num);

}

// Add all elements of array b to the set

for (int num : b) {

set.add(num);

}

// The size of the set is the count of distinct elements in the union

return set.size();

}

public static void main(String[] args) {

int[] a = {1, 2, 3, 4, 5};

int[] b = {1, 2, 3};

System.out.println(doUnion(a, b)); // Output: 5

}

}



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