Problems: Sriram N  
 22AD139

1. **Palindrome Linked List: (TC: O(n) SC:O(n))**

Given a singly linked list of integers. The task is to check if the given linked list is palindrome or not.

Examples:

Input: LinkedList: 1->2->1->1->2->1

Output: true

Explanation: The given linked list is 1->2->1->1->2->1 , which is a palindrome and Hence, the output is true.

**Program:**

import java.io.\*;

import java.util.\*;

public class PalindromeLinkedList {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

ListNode head = new ListNode(sc.nextInt());

ListNode current = head;

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

current.next = new ListNode(sc.nextInt());

current = current.next;

}

Solution sol = new Solution();

System.out.println(sol.isPalindrome(head) ? "Palindrome" : "Not a palindrome");

}

}

class Solution {

public boolean isPalindrome(ListNode head) {

StringBuilder ans = new StringBuilder();

while (head != null) {

ans.append(head.val);

head = head.next;

}

return ans.toString().equals(ans.reverse().toString());

}

}

class ListNode {

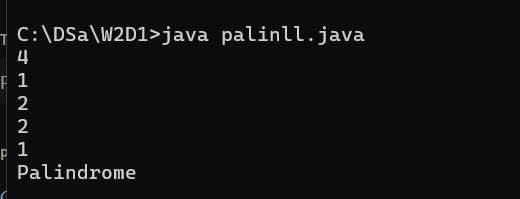
int val;

ListNode next;

ListNode(int x) { val = x; }

}

O/P:



1. **Floor in sorted array: (TC:O(logn))**

Given a sorted array arr[] (with unique elements) and an integer k, find the index (0-based) of the largest element in arr[] that is less than or equal to k. This element is called the "floor" of k. If such an element does not exist, return -1.

**Program:**

import java.io.\*;

import java.util.\*;

public class fisa {

public static void main(String[] fisa) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int[] nums = new int[n];

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

nums[i] = sc.nextInt();

}

int target = sc.nextInt();

Solution sol = new Solution();

System.out.println(sol.searchInsert(nums, target));

}

}

class Solution {

public int searchInsert(int[] nums, int target) {

int low = 0, high = nums.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (nums[mid] == target) {

return mid;

} else if (nums[mid] < target) {

low = mid + 1;

} else {

high = mid - 1;

}

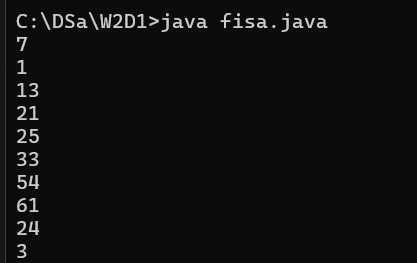
}

return low;

}

}

O/P:



1. **3SUM:**

Given an array arr of size n and an integer x. Find if there's a triplet in the array which sums up to the given integer x.

Examples:

Input:n = 6, x = 13, arr[] = [1,4,45,6,10,8]

Output: 1

Explanation: The triplet {1, 4, 8} in the array sums up to 13.

Input: n = 6, x = 10, arr[] = [1,2,4,3,6,7]

Output: 1

Explanation: Triplets {1,3,6} & {1,2,7} in the array sum to 10.

**Program:**

import java.io.\*;

import java.util.\*;

public class tsum {

public static void main(String tsum[]) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int[] nums = new int[n];

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

nums[i] = sc.nextInt();

}

Solution sol = new Solution();

List<List<Integer>> result = sol.threeSum(nums);

for (List<Integer> triplet : result) {

System.out.println(triplet);

}

}

}

class Solution {

public List<List<Integer>> threeSum(int[] nums) {

Set<List<Integer>> ans = new HashSet<>();

Arrays.sort(nums);

int n = nums.length;

for (int i = 0; i < n - 2; i++) {

int j = i + 1, k = n - 1;

while (j < k) {

int sum = nums[i] + nums[j] + nums[k];

if (sum == 0) {

ans.add(Arrays.asList(nums[i], nums[j], nums[k]));

j++;

k--;

} else if (sum > 0) {

k--;

} else {

j++;

}

}

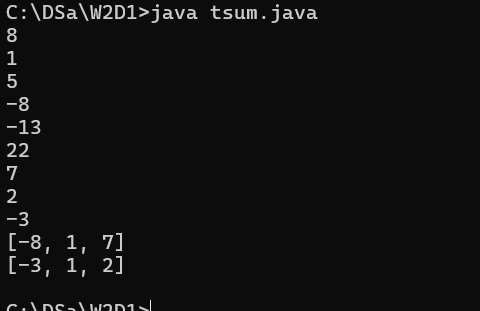
}

return new ArrayList<>(ans);

}

}

**O/P:**

****

1. **Balanced Tree:**

Given a binary tree, find if it is height balanced or not. A tree is height balanced if difference between heights of left and right subtrees is not more than one for all nodes of tree.

Examples:

Input:

1

/

2

\

3

Output: 0

Explanation: The max difference in height of left subtree and right subtree is 2, which is greater than 1. Hence unbalanced

**Program:**

class Tree

{

boolean isBalanced(Node root)

{

Boolean[] ans = {true};

isBalanced(root, ans);

return ans[0];

}

static int isBalanced(Node root, Boolean[] ans) {

if (root == null) return 0;

int leftHeight = isBalanced(root.left, ans);

int rightHeight = isBalanced(root.right, ans);

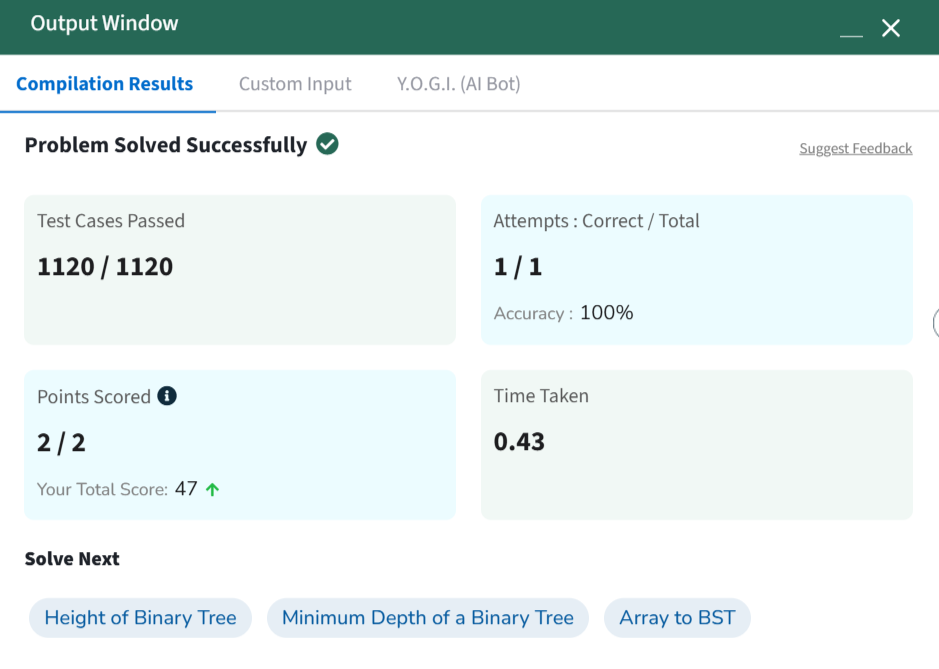
if (Math.abs(leftHeight - rightHeight) > 1) ans[0] = false;

return 1 + Math.max(leftHeight, rightHeight);

}

}

**O/P:**



1. **Check equal arrays**

Given two arrays arr1 and arr2 of equal size, the task is to find whether the given arrays are equal. Two arrays are said to be equal if both contain the same set of elements, arrangements (or permutations) of elements may be different though.

Note: If there are repetitions, then counts of repeated elements must also be the same for two arrays to be equal.

Examples:

Input: arr1[] = [1, 2, 5, 4, 0], arr2[] = [2, 4, 5, 0, 1]

Output: true

Explanation: Both the array can be rearranged to [0,1,2,4,5]

**Program:**

class Solution {

// Function to check if two arrays are equal or not.

public static boolean check(int[] a, int[] b) {

if (a.length!=b.length) return false;

Arrays.sort(a);

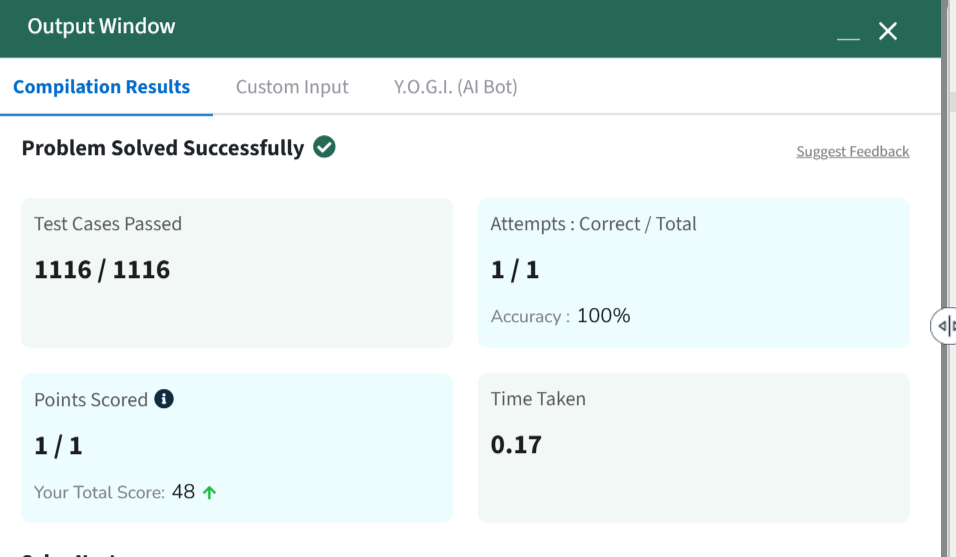
Arrays.sort(b);

return Arrays.equals(a, b);

}

}

**O/P:**

****

1. **0,1 Knapsack:**

You are given the weights and values of items, and you need to put these items in a knapsack of capacity capacity to achieve the maximum total value in the knapsack. Each item is available in only one quantity.

In other words, you are given two integer arrays val[] and wt[], which represent the values and weights associated with items, respectively. You are also given an integer capacity, which represents the knapsack capacity. Your task is to find the maximum sum of values of a subset of val[] such that the sum of the weights of the corresponding subset is less than or equal to capacity. You cannot break an item; you must either pick the entire item or leave it (0-1 property).

Examples :

Input: capacity = 4, val[] = [1, 2, 3], wt[] = [4, 5, 1]

Output: 3

Explanation: Choose the last item, which weighs 1 unit and has a value of 3.

**Program:**

class Solution {

static int knapSack(int W, int val[], int wt[]) {

return knapSackRecursive(W, val, wt, val.length);

}

static int knapSackRecursive(int W, int val[], int wt[], int n) {

if (n == 0 || W == 0) return 0;

if (wt[n-1] > W) return knapSackRecursive(W, val, wt, n-1);

else return Math.max(val[n-1] + knapSackRecursive(W - wt[n-1], val, wt, n-1), knapSackRecursive(W, val, wt, n-1));

}

**O/P:**

