1. **0-1 Knapsack Problem**

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:**

public:

int solve(int w, vector<int>& wt, vector<int>& val,int n){

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

if(w >= wt[n-1])

return max(val[n-1] + solve(w-wt[n-1], wt, val, n-1) , solve(w, wt, val, n-1));

else return solve(w, wt, val, n-1);

}

int knapSack(int w, vector<int>& val, vector<int>& wt) {

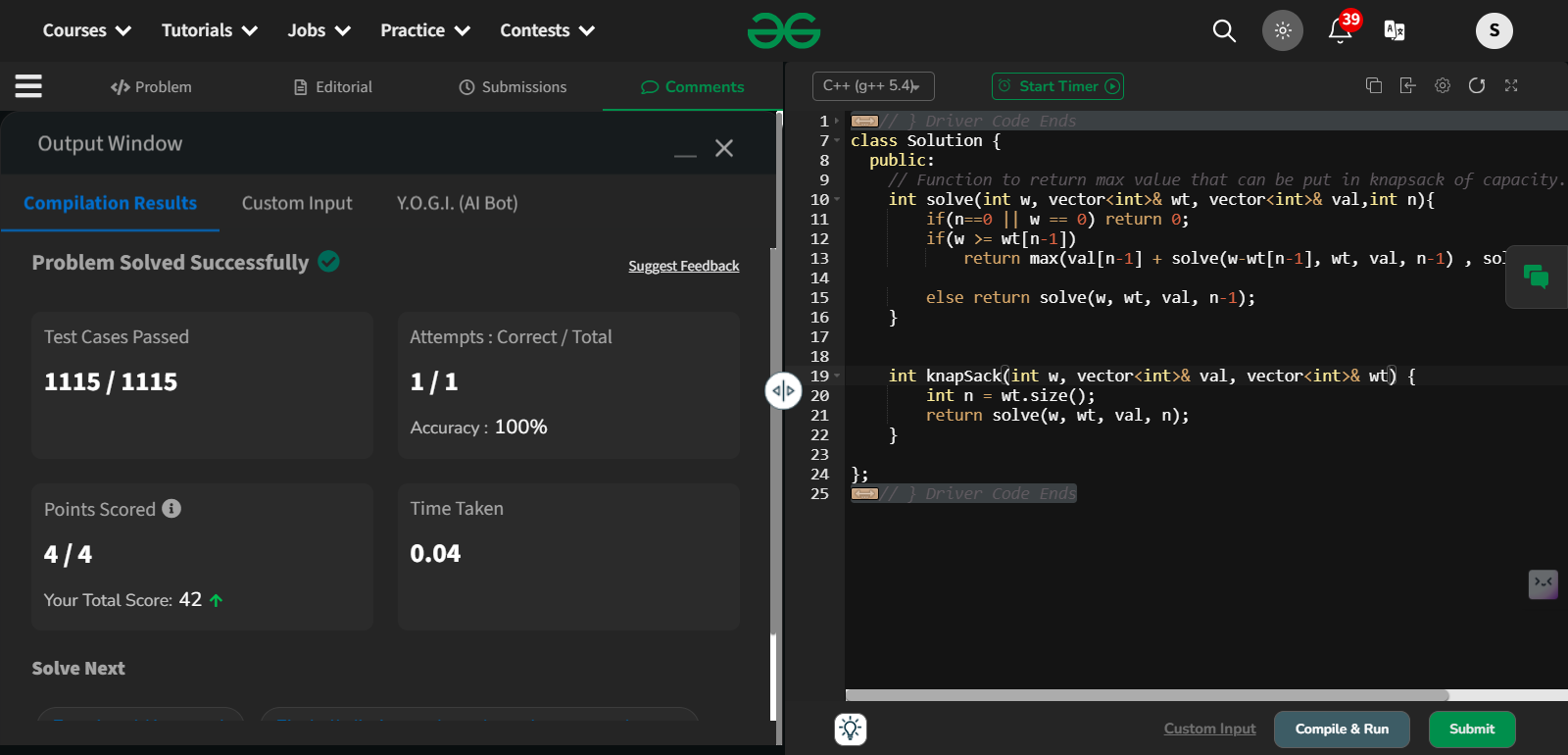
int n = wt.size();

return solve(w, wt, val, n);

}

};

**Output:**



1. **Floor in a sorted array**

**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.**

**Examples:**

Input: arr[] = [1, 2, 8, 10, 11, 12, 19], k = 0

Output: -1

Explanation: No element less than 0 is found. So output is -1.

**Program:**

class Solution {

public:

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

int n=arr.size();

int low=0;

int high=arr.size()-1;

bool flag=false;

int floorind=-1;

while(low<=high)

{

int mid=(low+high)/2;

if(arr[mid]==k)

{

flag=true;

return mid;

}

if(arr[mid]<k)

{

floorind=mid;

low=mid+1;

}

else

{

high=mid-1;

}

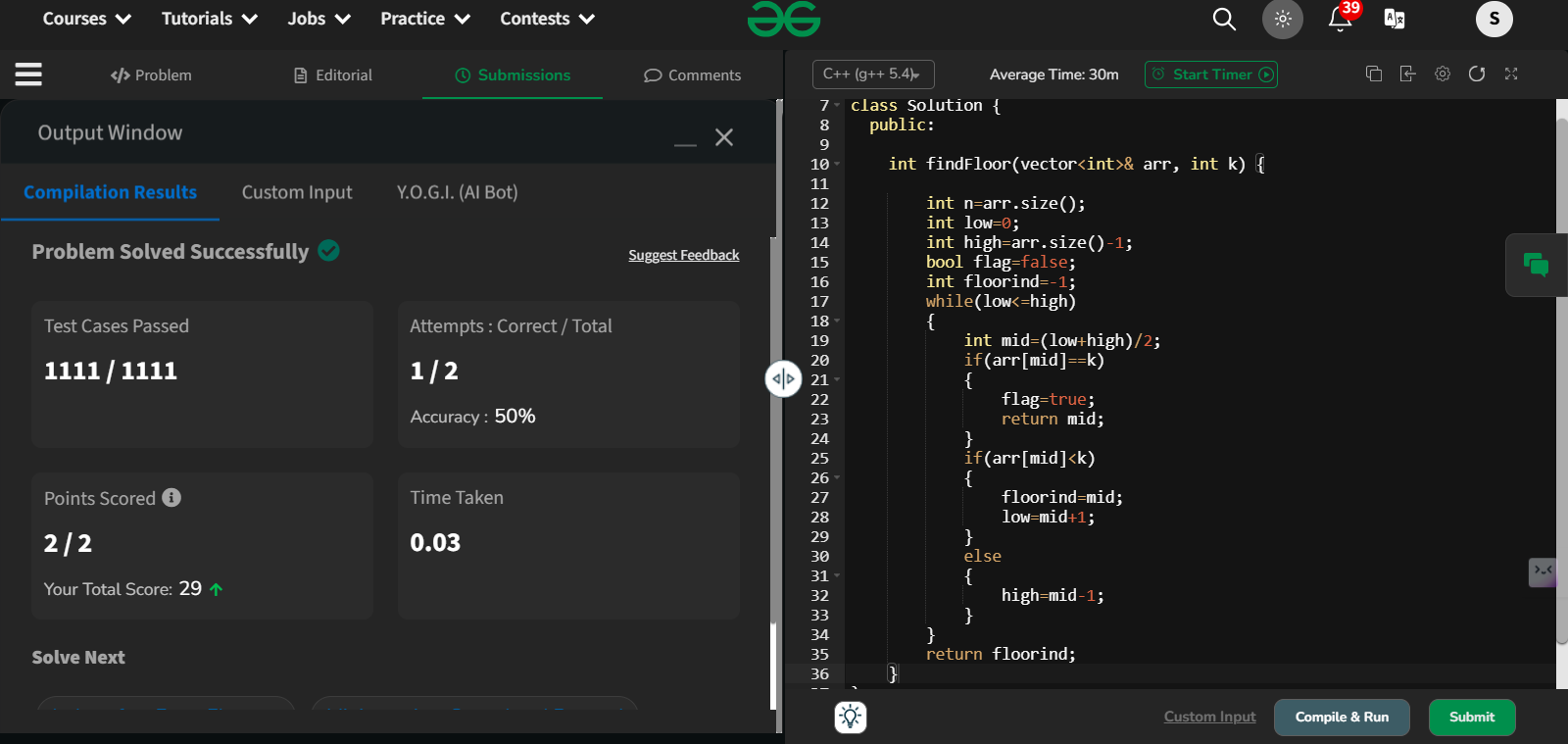
}

return floorind;

}

};

Output:



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 {

public:

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

bool check(vector<int>& arr1, vector<int>& arr2) {

int x=arr1.size();

int y=arr2.size();

if(x!=y)

{

return false;

}

sort(arr1.begin(),arr1.end());

sort(arr2.begin(),arr2.end());

for(int i=0;i<x;i++)

{

if(arr1[i]!=arr2[i])

{

return false;

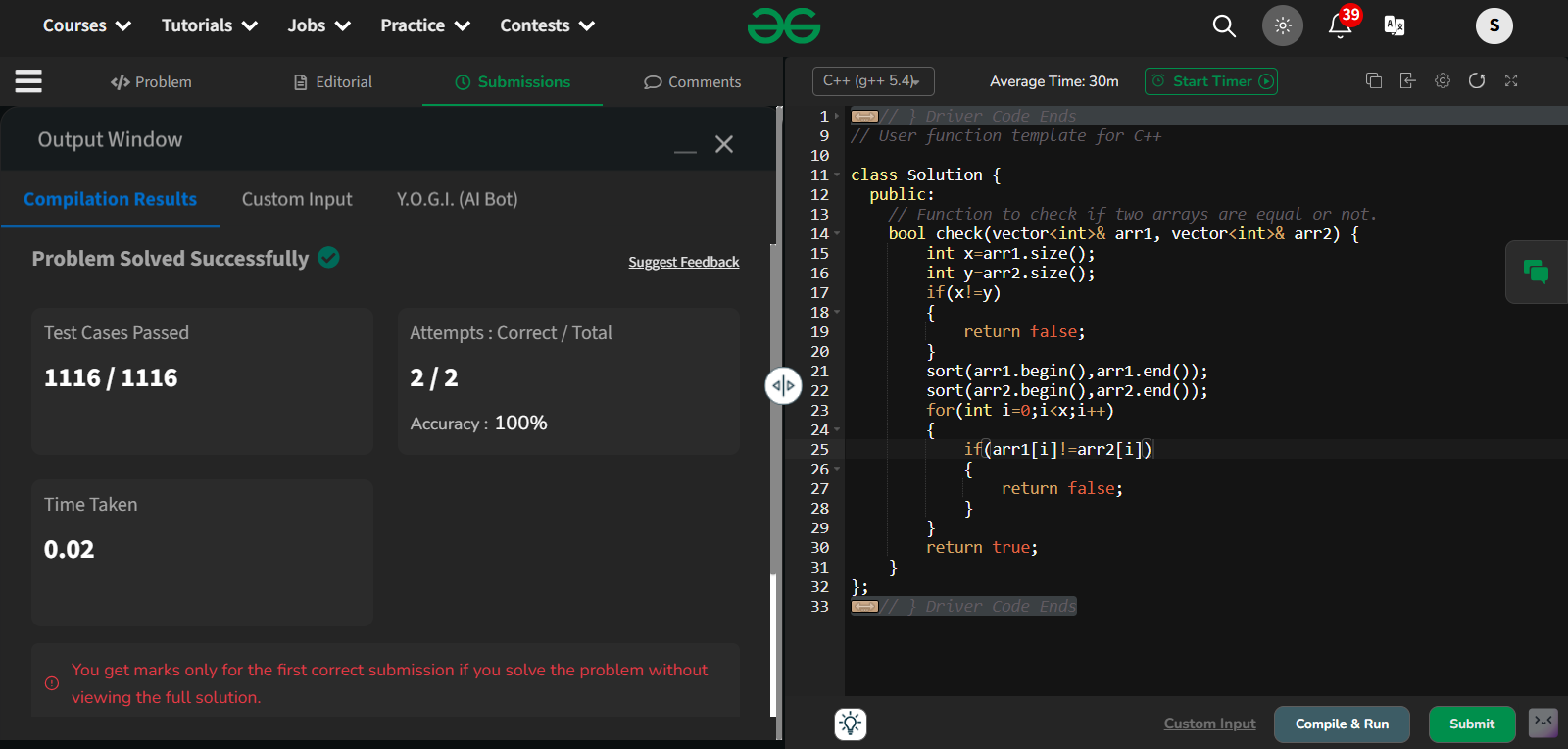
}

}

return true;

}};

**Output:**

****

**4.Palindrome linked list**

**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:**

class Solution {

public:

// Function to check whether the list is palindrome.

bool isPalindrome(Node \*head) {

vector<int> s;

if(head=='\0') return true;

int si=0;

Node\* curr=head;

while(curr!=nullptr)

{

int x=curr->data;

curr=curr->next;

s.push\_back(x);

si+=1;

}

int left=0;

int right=si-1;

while(left<right)

{

if(s[left]!=s[right])

{

return false;

}

left++;

right--;

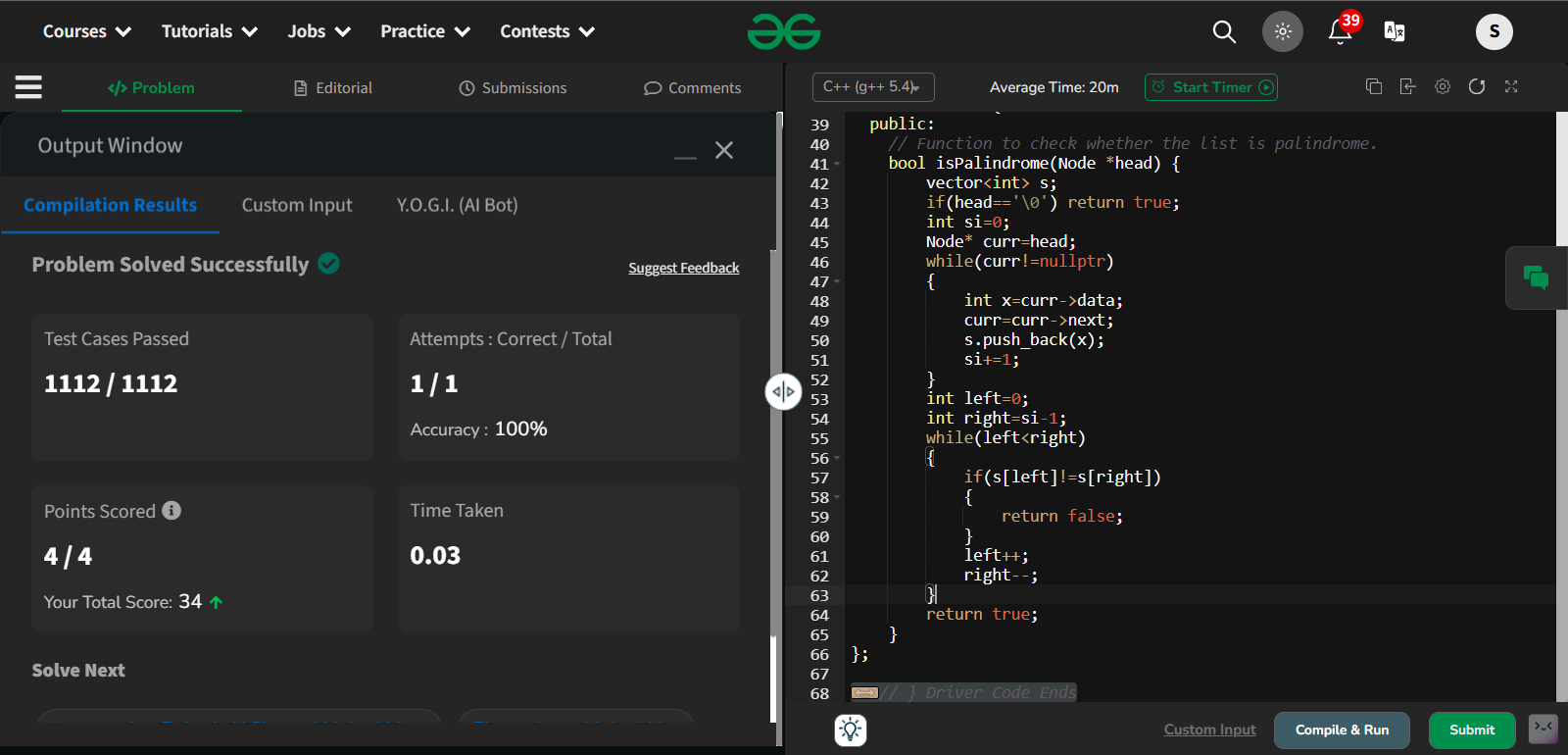
}

return true;

}

};

**Output:**



**5.Balanced tree check**

**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 Solution{**

**public:**

**pair<bool,int> fastcheck(Node\* root){**

**//base case**

**if(root==NULL){**

**pair<bool,int> p = make\_pair(true,0);**

**return p;**

**}**

**pair<bool,int> left = fastcheck(root->left);**

**pair<bool,int> right = fastcheck(root->right);**

**bool diff = abs(left.second-right.second)<=1;**

**pair<bool,int> ans;**

**ans.second = max(left.second,right.second)+1;**

**if(left.first && right.first && diff){**

**ans.first= 1;**

**}**

**else{**

**ans.first= 0;**

**}**

**return ans;**

**}**

**bool isBalanced(Node \*root)**

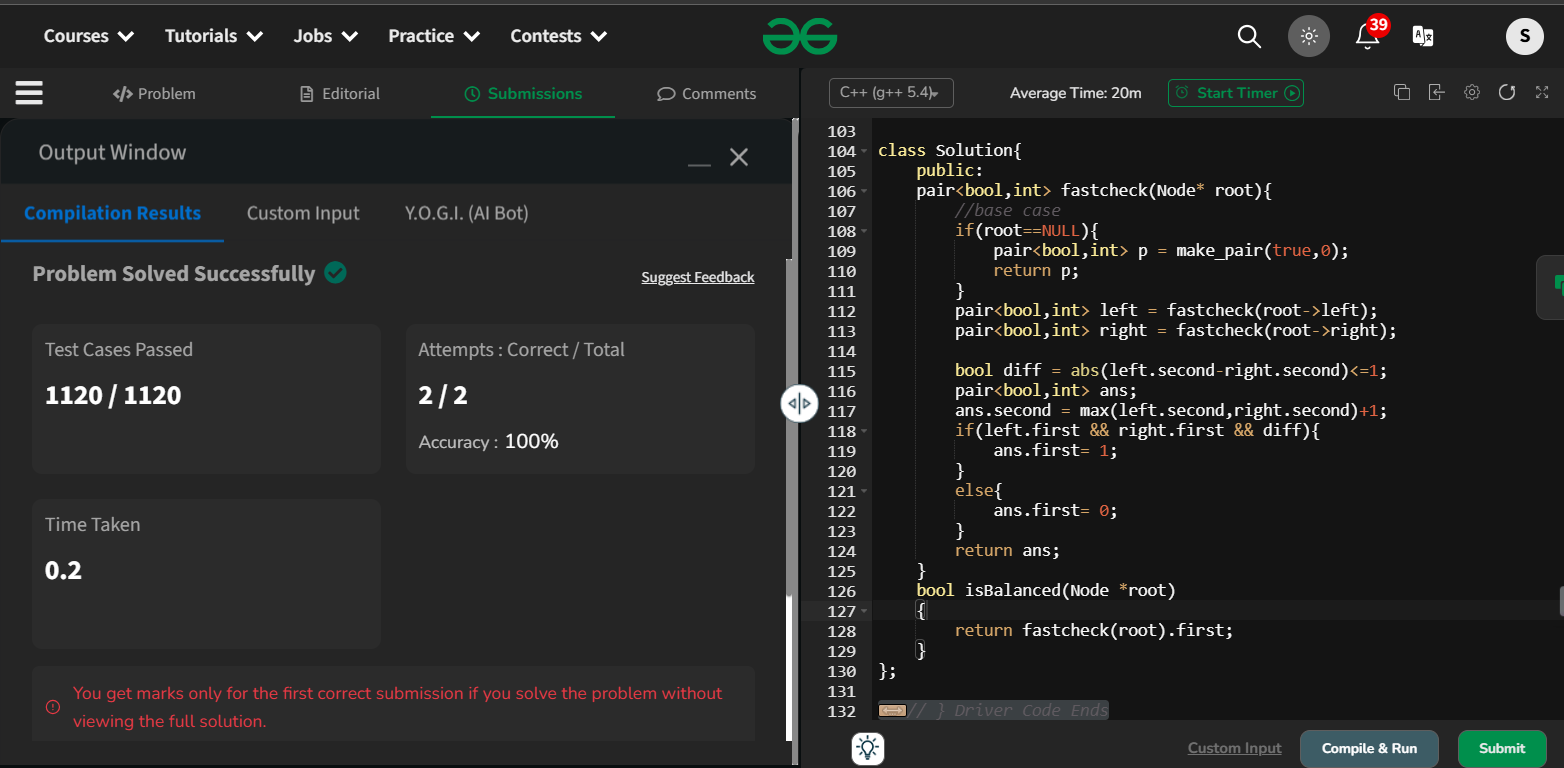
**{**

**return fastcheck(root).first;**

**}**

**};**

**Output:**

****

**6.Triplet sum in array**

**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:**

class Solution {

public:

// Should return true if there exists a triplet in the

// array arr[] which sums to x. Otherwise false

bool find3Numbers(int arr[], int n, int x) {

sort(arr,arr+n);

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

{

int start=i+1;

int end=n-1;

while(start<end)

{

int total=arr[i]+arr[start]+arr[end];

if(total==x)

{

return true;

}

if(total>x) end--;

else start++;

}

}

return false;

}

};

**Output:**

