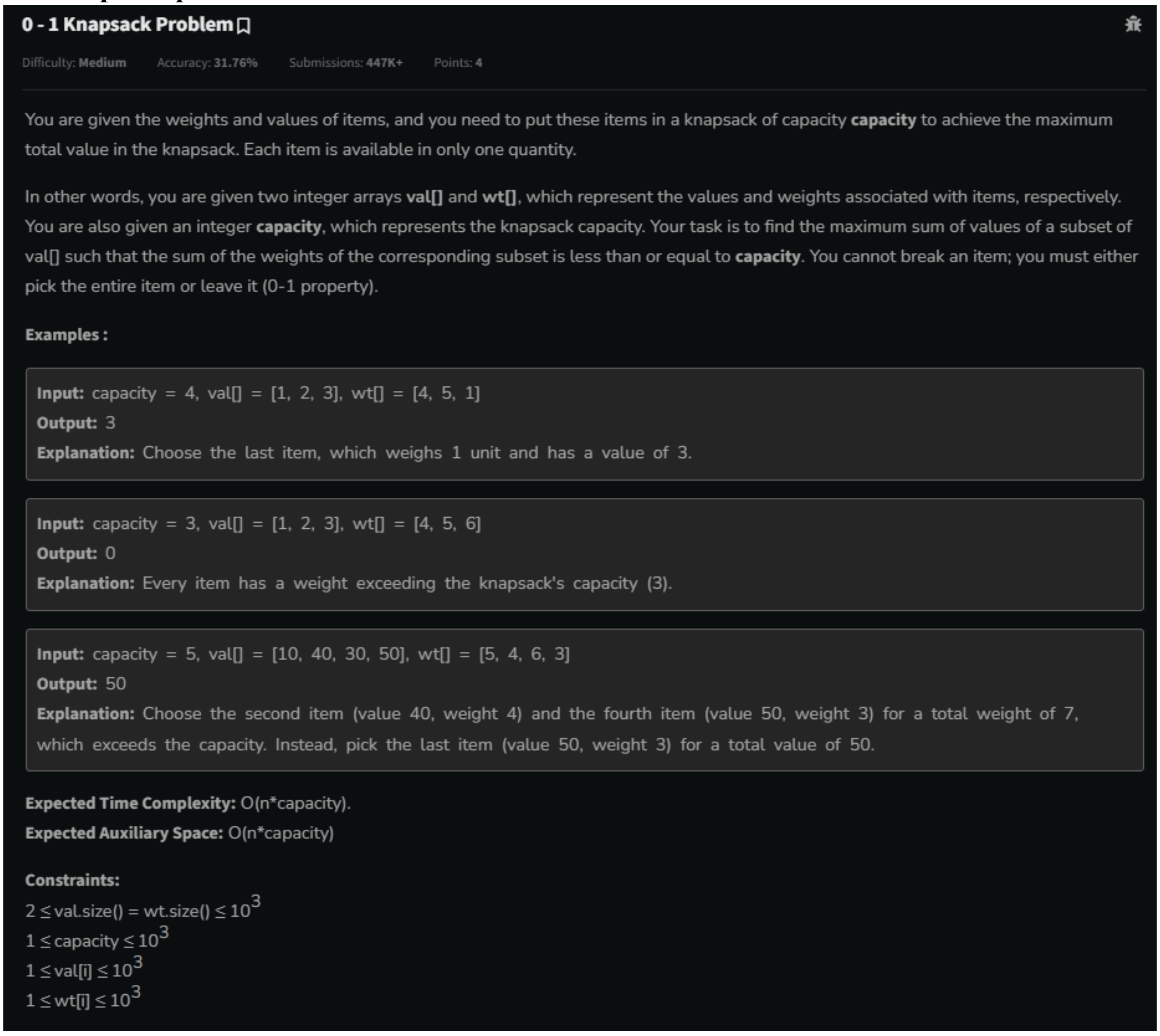
Coding Practice Set-2

1. **0-1 Knapsack problem:**



0 - 1 Knapsack Problem

Difficulty: **Medium** Accuracy: **31.76%** Submissions: **447K+** Points: **4**

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.
```

```
Input: capacity = 3, val[] = [1, 2, 3], wt[] = [4, 5, 6]
Output: 0
Explanation: Every item has a weight exceeding the knapsack's capacity (3).
```

```
Input: capacity = 5, val[] = [10, 40, 30, 50], wt[] = [5, 4, 6, 3]
Output: 50
Explanation: Choose the second item (value 40, weight 4) and the fourth item (value 50, weight 3) for a total weight of 7, which exceeds the capacity. Instead, pick the last item (value 50, weight 3) for a total value of 50.
```

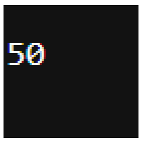
Expected Time Complexity: O(n*capacity).
Expected Auxiliary Space: O(n*capacity)

Constraints:
$2 \le val.size() = wt.size() \le 10^3$
$1 \le capacity \le 10^3$
$1 \le val[i] \le 10^3$
$1 \le wt[i] \le 10^3$

Code:

```
public class Knapsack {
    public static int knapsack(int capacity, int[] val, int[] wt) {
        int n = val.length;
        int[][] dp = new int[n + 1][capacity + 1];

        for (int i = 1; i <= n; i++) {
            for (int w = 0; w <= capacity; w++) {
                if (wt[i - 1] <= w) {
                    dp[i][w] = Math.max(dp[i - 1][w], dp[i - 1][w - wt[i - 1]] + val[i - 1]);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }

        return dp[n][capacity];
    }

    public static void main(String[] args) {
        int capacity = 5;
        int[] val = {10, 40, 30, 50};
        int[] wt = {5, 4, 6, 3};

        System.out.println(knapsack(capacity, val, wt));
    }
}
```

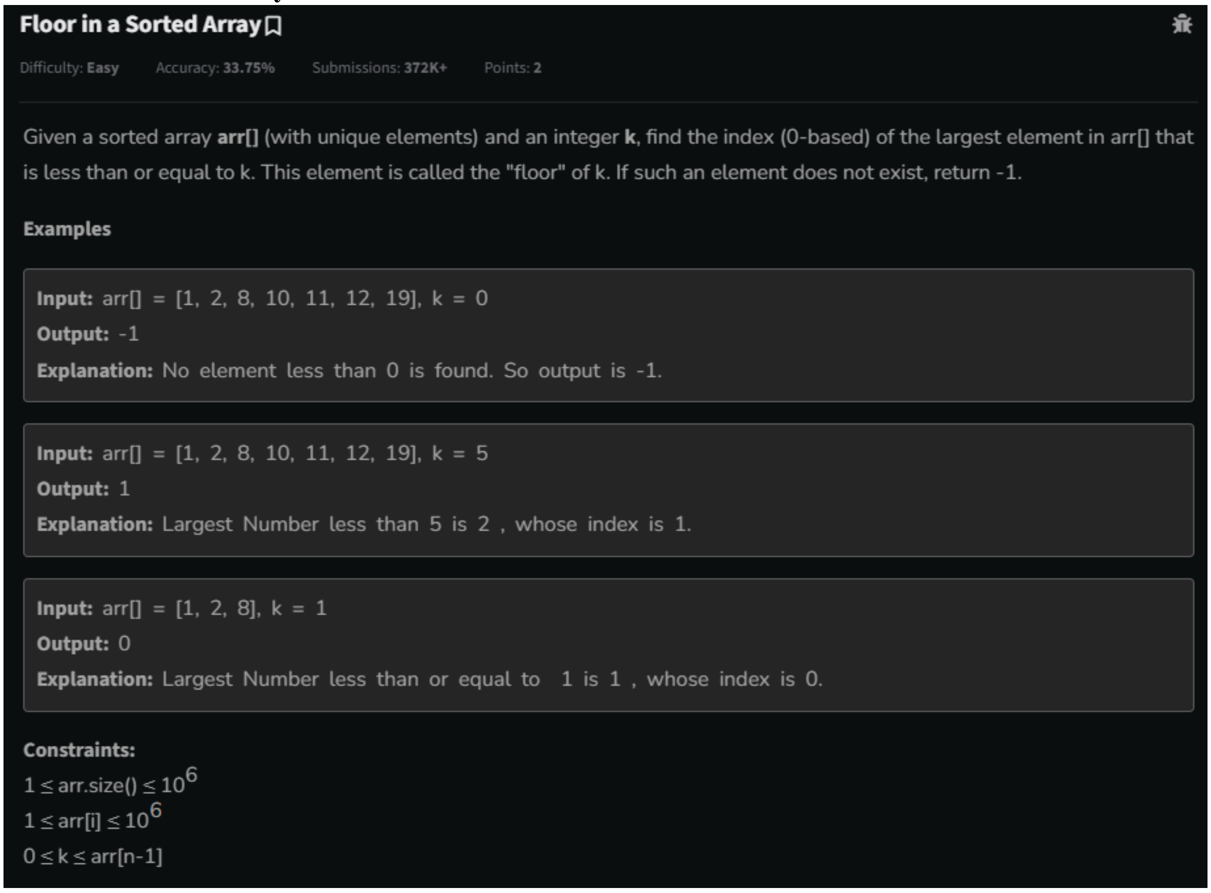
Output:



```
50
```

Time Complexity: O(n*C)

2. Floor in a sorted array



Floor in a Sorted Array

Difficulty: **Easy** Accuracy: **33.75%** Submissions: **372K+** Points: **2**

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.

Input: arr[] = [1, 2, 8, 10, 11, 12, 19], k = 5
Output: 1
Explanation: Largest Number less than 5 is 2 , whose index is 1.

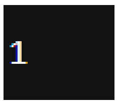
Input: arr[] = [1, 2, 8], k = 1
Output: 0
Explanation: Largest Number less than or equal to 1 is 1 , whose index is 0.

Constraints:
$1 \le arr.size() \le 10^6$
$1 \le arr[i] \le 10^6$
$0 \le k \le arr[n-1]$

Code:

```
public class FloorInSortedArray {
    public static int findFloor(int[] arr, int k) {
        int left = 0, right = arr.length - 1, result = -1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] <= k) {
                result = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return result;
    }
    public static void main(String[] args) {
        int[] arr = {1, 2, 8, 10, 11, 12, 19};
        int k = 5;
        System.out.println(findFloor(arr, k));
    }
}
```

Output:

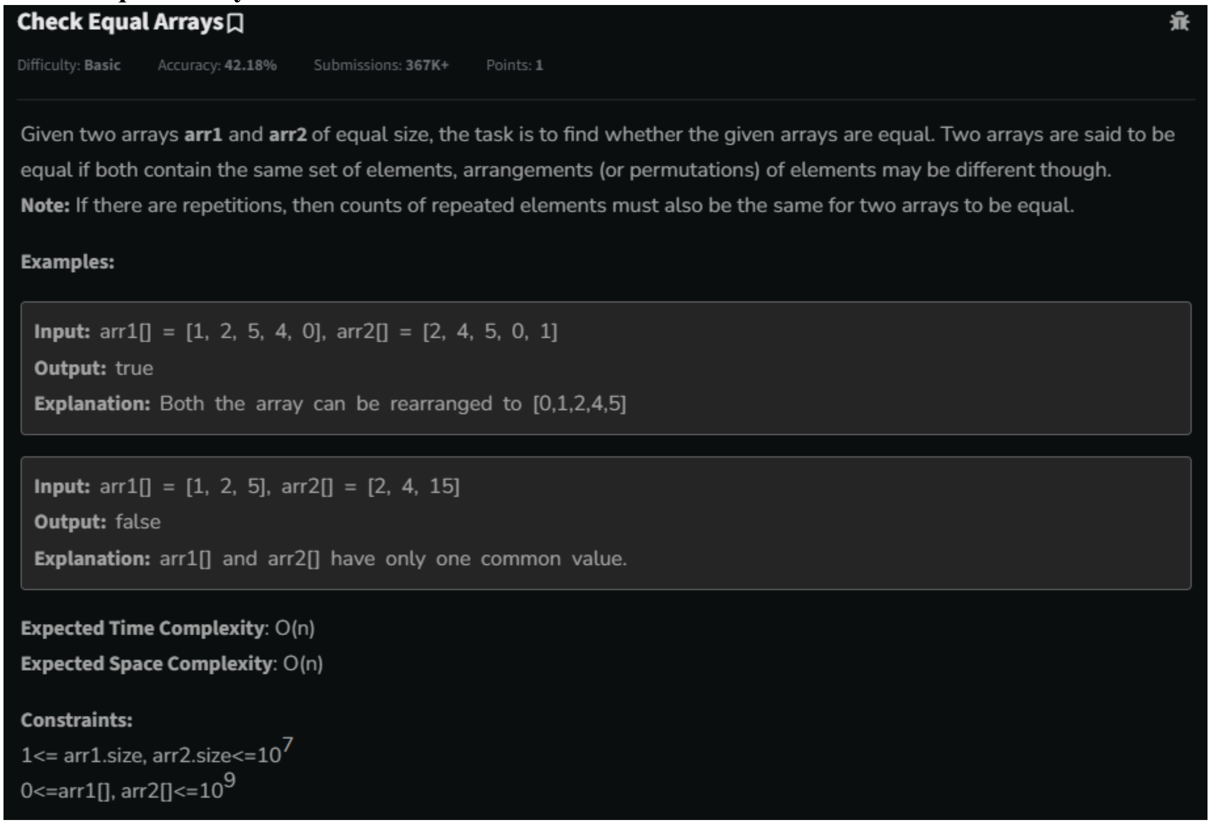


```
1
```

Time Complexity:

Binary Search: O(log n)

3. **Check Equal Arrays:**



Check Equal Arrays

Difficulty: **Basic** Accuracy: **42.18%** Submissions: **367K+** Points: **1**

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]

Input: arr1[] = [1, 2, 5], arr2[] = [2, 4, 15]
Output: false
Explanation: arr1[] and arr2[] have only one common value.

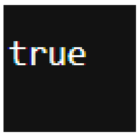
Expected Time Complexity: O(n)
Expected Space Complexity: O(n)

Constraints:
1<= arr1.size, arr2.size<=10^7
0<=arr1[], arr2[]<=10^9

Code:

```
import java.util.HashMap;

public class EqualArrays {
    public static boolean areArraysEqual(int[] arr1, int[] arr2) {
        if (arr1.length != arr2.length) return false;

        HashMap<Integer, Integer> map = new HashMap<>();
        for (int num : arr1) {
            map.put(num, map.getOrDefault(num, 0) + 1);
        }
        for (int num : arr2) {
            if (!map.containsKey(num) || map.get(num) == 0) return false;
            map.put(num, map.get(num) - 1);
        }
        return true;
    }

    public static void main(String[] args) {
        int[] arr1 = {1, 2, 5, 4, 0};
        int[] arr2 = {2, 4, 5, 0, 1};
        System.out.println(areArraysEqual(arr1, arr2));
    }
}
```

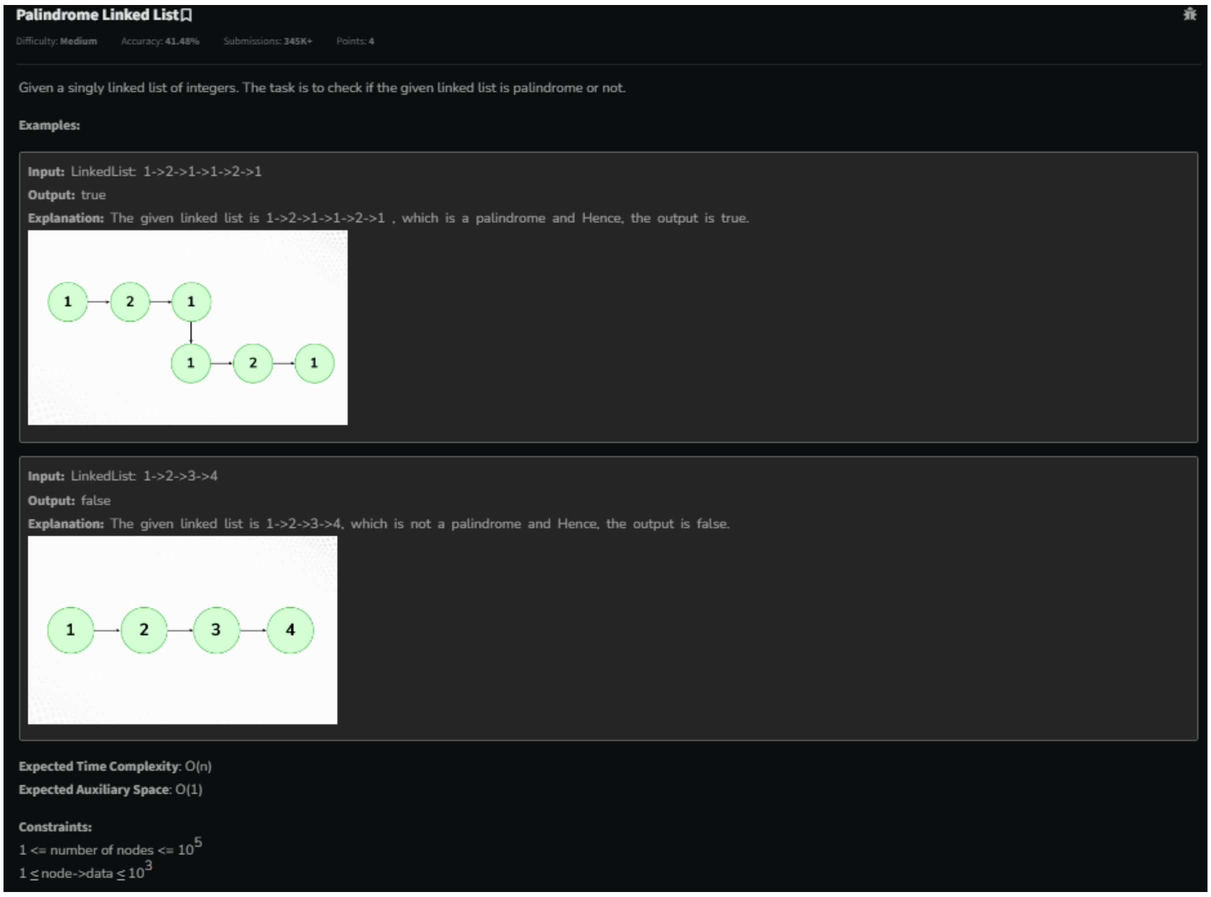
Output:



```
true
```

Time Complexity: O(n)

4. Palindrome linked list



Palindrome Linked List

Difficulty: **Medium** Accuracy: **41.48%** Submissions: **345K+** Points: **4**

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.

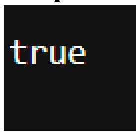
Input: LinkedList: 1->2->3->4
Output: false
Explanation: The given linked list is 1->2->3->4, which is not a palindrome and Hence, the output is false.

Expected Time Complexity: O(n)
Expected Auxiliary Space: O(1)

Constraints:
1 <= number of nodes <= 10^5
1 ≤ node->data ≤ 10^3

Code:

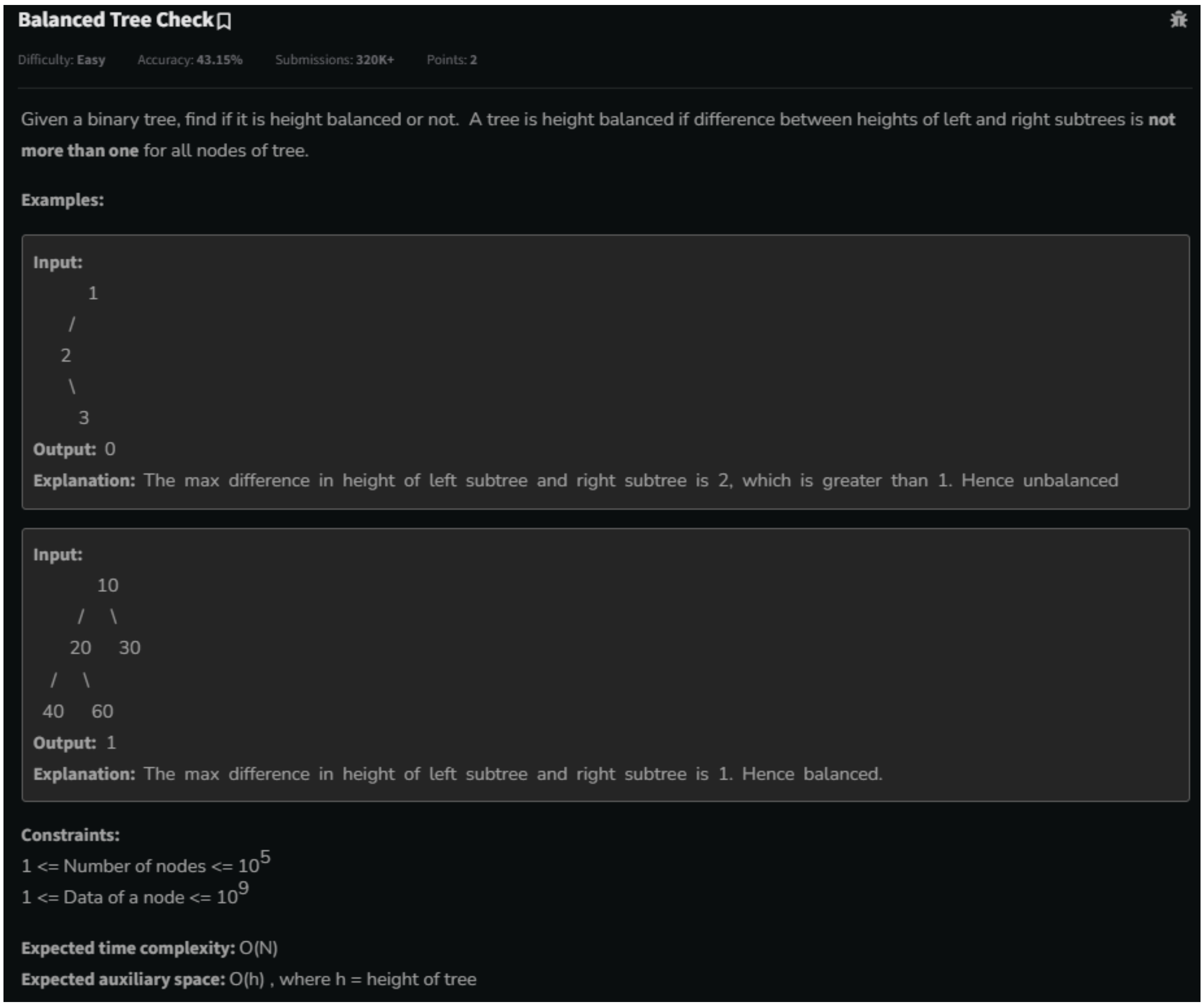
```
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
        this.next = null;
    }
}

public class PalindromeLinkedList {
    public static boolean isPalindrome(Node head) {
        if (head == null || head.next == null) return true;

        Node slow = head, fast = head;

        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        Node secondHalf = reverseList(slow);
        Node firstHalf = head;
        while (secondHalf != null) {
            if (firstHalf.data != secondHalf.data) return false;
            firstHalf = firstHalf.next;
            secondHalf = secondHalf.next;
        }

        return true;
    }

    private static Node reverseList(Node head) {
        Node prev = null;
        while (head != null) {
            Node nextNode = head.next;
            head.next = prev;
            prev = head;
            head = nextNode;
        }
        return prev;
    }

    public static void main(String[] args) {
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(1);
        head.next.next.next = new Node(1);
        head.next.next.next.next = new Node(2);
        head.next.next.next.next.next = new Node(1);

        System.out.println(isPalindrome(head)); // Output: true
    }
}
```
Output:



```
true
```

Time Complexity: O(n)

5. **Balanced Tree Check:**



Balanced Tree Check

Difficulty: **Easy** Accuracy: **43.15%** Submissions: **320K+** Points: **2**

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

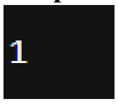
Input:
```
      10
    /   \
   20   30
 /   \
40   60
```
Output: 1
Explanation: The max difference in height of left subtree and right subtree is 1. Hence balanced.

Constraints:
1 <= Number of nodes <= 10^5
1 <= Data of a node <= 10^9

Expected time complexity: O(N)
Expected auxiliary space: O(h) , where h = height of tree

Code:
```
class TreeNode {
   int data;
   TreeNode left, right;
   TreeNode(int data) {
      this.data = data;
      left = right = null;
   }
}

public class Solution {
   public static int height(TreeNode root) {
      if (root == null) return 0;
      int leftHeight = height(root.left);
      int rightHeight = height(root.right);

      if (leftHeight == -1 || rightHeight == -1 || Math.abs(leftHeight - rightHeight) > 1) {
         return -1;
      }

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

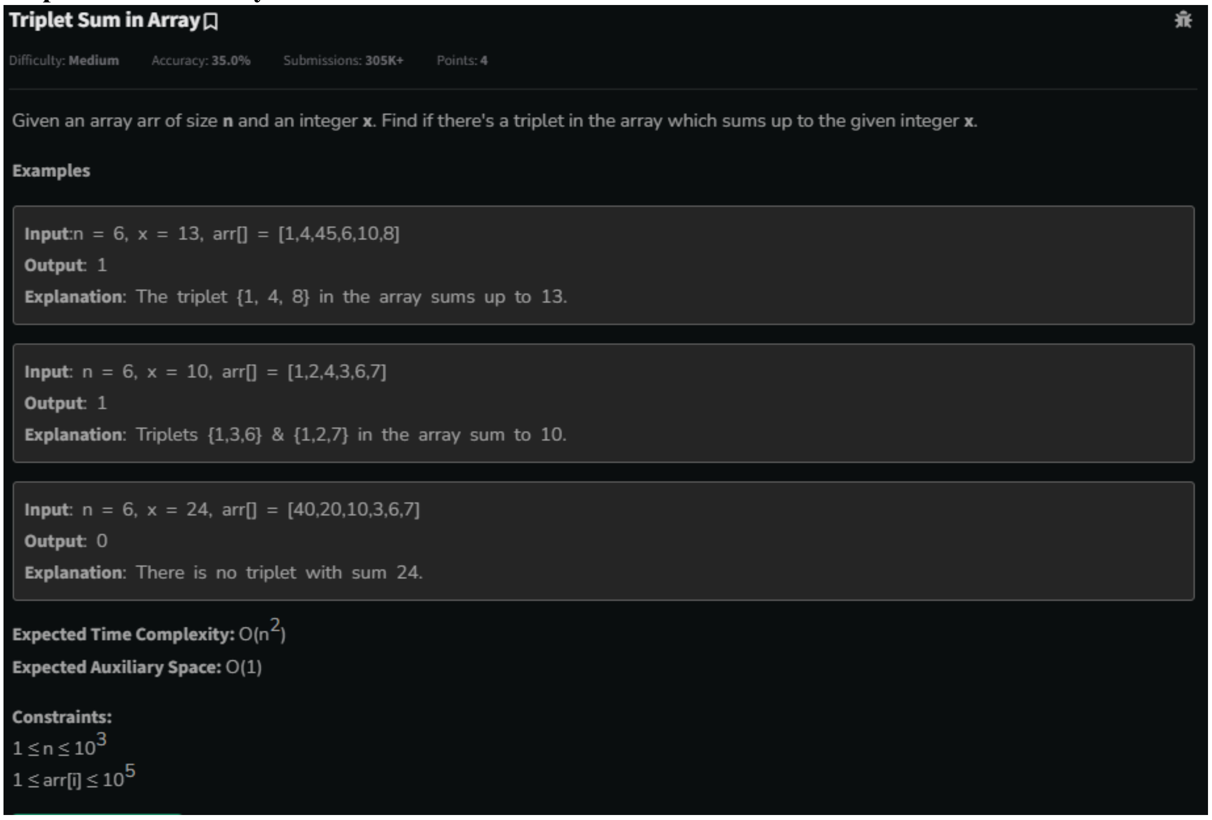
```
    }

    public static boolean isBalanced(TreeNode root) {
        return height(root) != -1;
    }

    public static void main(String[] args) {
        TreeNode root = new TreeNode(10);
        root.left = new TreeNode(20);
        root.right = new TreeNode(30);
        root.left.left = new TreeNode(40);
        root.left.right = new TreeNode(60);

        System.out.println(isBalanced(root));  // Output: true
    }
}
```
Output:



```
1
```

Time Complexity: O(n)

6. Triplet Sum in Array


Triplet Sum in Array

Difficulty: Medium Accuracy: 35.0% Submissions: 305K+ Points: 4

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.
```

```
Input: n = 6, x = 24, arr[] = [40,20,10,3,6,7]
Output: 0
Explanation: There is no triplet with sum 24.
```

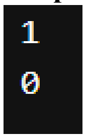
Expected Time Complexity: $O(n^2)$
Expected Auxiliary Space: O(1)

Constraints:
$1 \leq n \leq 10^3$
$1 \leq arr[i] \leq 10^5$

Code:

```
import java.util.Arrays;

public class TripletSum {
    public static int findTriplet(int[] arr, int x) {
        Arrays.sort(arr);
        int n = arr.length;
        for (int i = 0; i < n - 2; i++) {
            int left = i + 1, right = n - 1;
            while (left < right) {
                int sum = arr[i] + arr[left] + arr[right];
                if (sum == x) {
                    return 1;
                } else if (sum < x) {
                    left++;
                } else {
                    right--;
                }
            }
        }
        return 0;
    }

    public static void main(String[] args) {
        int[] arr = {1, 4, 45, 6, 10, 8};
        int x = 13;
        System.out.println(findTriplet(arr, x));

        int[] arr2 = {40, 20, 10, 3, 6, 7};
        int x2 = 24;
        System.out.println(findTriplet(arr2, x2));
    }
}
```

Output:



```
1
0
```

Time Complexity: O(n)