

DAY 2 DSA TRAINING

Name: SUNDARA VINAYAGAM V

Dept: 3rd year CSBS

1. Floor in the 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.

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.

Solution:

```
import java.util.*;
class SortedArray{
    public static void main(String[] ar){
        Scanner s=new Scanner(System.in);
        System.out.println("Enter the length of an array:");
        int n=s.nextInt();
        int arr[]=new int[n];
        System.out.println("Enter the array elements:");
        for(int i=0;i<n;i++){
            arr[i]=s.nextInt();
        }
        System.out.println("Enter k value:");
        int k=s.nextInt();
        int max = Integer.MIN_VALUE;
        int index = -1;

        for (int i = 0; i < arr.length; i++) {
            if (arr[i] <= k && arr[i] > max) {
                max = arr[i];
                index = i;
            }
        }

        System.out.println("The index less than k is: "+ index);
    }
}
```

```

2
PS F:\cse\java> java .\SortedArray.java
Enter the length of an array:
5
Enter the array elements:
12
14
16
22
25
Enter k value:
15
The index less than k is: 1
PS F:\cse\java> |

```

Time Complexity: $O(n)$

Space Complexity: $O(n)$

2. 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]

Input: arr1[] = [1, 2, 5], arr2[] = [2, 4, 15]

Output: false

Explanation: arr1[] and arr2[] have only one common value.

Solution

```

import java.util.*;

class CheckArray {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the size of the first array: ");
        int n1 = sc.nextInt();
        int[] arr1 = new int[n1];
        System.out.println("Enter elements of the first array:");
        for (int i = 0; i < n1; i++) {
            arr1[i] = sc.nextInt();
        }

        System.out.print("Enter the size of the second array: ");
        int n2 = sc.nextInt();
        int[] arr2 = new int[n2];
    }
}

```

```

System.out.println("Enter elements of the second array:");
for (int i = 0; i < n2; i++) {
    arr2[i] = sc.nextInt();
}

if (arr1.length != arr2.length) {
    System.out.println("The arrays are not equal.");
    sc.close();
    return;
}

HashMap<Integer, Integer> h1 = new HashMap<>();
HashMap<Integer, Integer> h2 = new HashMap<>();
for (int n : arr1) {
    h1.put(n, h1.getOrDefault(n, 0) + 1);
}
for (int n : arr2) {
    h2.put(n, h2.getOrDefault(n, 0) + 1);
}

if (h1.equals(h2)) {
    System.out.println("The arrays are equal.");
} else {
    System.out.println("The arrays are not equal.");
}
}
}

```

Output:

```

PS F:\cse\java> javac .\CheckArray.java
PS F:\cse\java> java .\CheckArray.java
Enter the size of the first array: 5
Enter elements of the first array:
1
2
3
4
5
Enter the size of the second array: 5
Enter elements of the second array:
2
3
4
0
1
The arrays are not equal.
PS F:\cse\java> |

```

Time Complexity: $O(n)$

Space Complexity: $O(n)$

3.Triplet Sum:

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.

Solution:

```
import java.util.*;
```

```
class TripletSum {
    public static boolean find3Numbers(int arr[], int n, int x) {
        Arrays.sort(arr);
        int count = 0;
        for (int i = 0; i < n - 2; i++) {
            int left = i + 1;
            int right = n - 1;
            while (left < right) {
                int currentSum = arr[i] + arr[left] + arr[right];
                if (currentSum == x) {
                    count++;
                    left++;
                    right--;
                } else if (currentSum < x) {
                    left++;
                } else {
                    right--;
                }
            }
        }
        return count >= 1;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the number of elements in the array: ");
        int n = sc.nextInt();
        int[] arr = new int[n];
        System.out.println("Enter the elements of the array:");
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        System.out.print("Enter the target sum: ");
        int x = sc.nextInt();
    }
}
```

```

        boolean result = find3Numbers(arr, n, x);
        if (result) {
            System.out.println("Triplet found.");
        } else {
            System.out.println("No triplet found.");
        }
        sc.close();
    }
}

```

```

PS F:\cse\java> javac .\TripletSum.java
PS F:\cse\java> java .\TripletSum.java
Enter the number of elements in the array: 6
Enter the elements of the array:
2
5
4
7
8
9
Enter the target sum: 15
Triplet found.
PS F:\cse\java> |

```

Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

4. Palindrome Linked List:

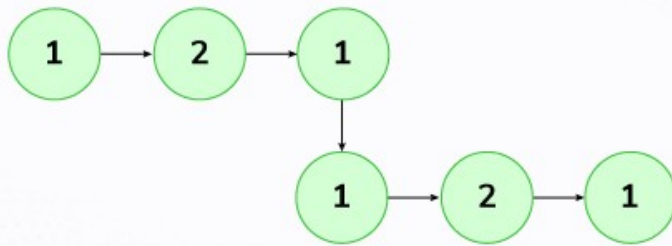
Given a singly linked list of integers. The task is to check if the given linked list is palindrome or not.

Examples:

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

Output: true

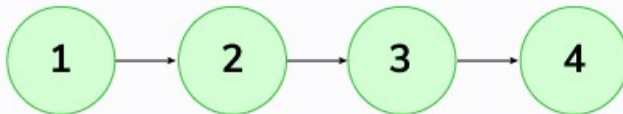
Explanation: The given linked list is 1->2->1->1->2->1 , which is a palindrome and Hence, the output is true.



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.

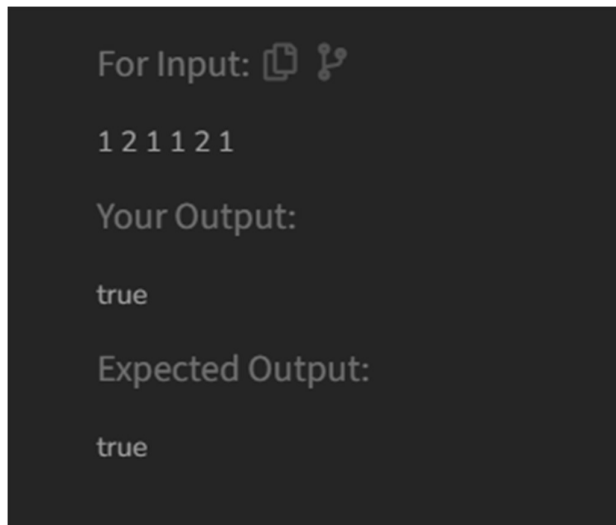


```
class Solution {
    // Function to check whether the list is palindrome.
    boolean isPalindrome(Node head) {
        // Your code here
        Stack<Integer> stack =new Stack();
        Node curr=head;
        while(curr != null)
        {
            stack.push(curr.data);
            curr=curr.next;
        }
        curr=head;

        while(curr != null){
            if(curr.data !=stack.pop()){
                return false;
            }
            curr=curr.next;
        }
        return true;
    }
}
```

```
}
```

Output:



Time Complexity: $O(n)$

Space Complexity: $O(1)$

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

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:

$1 \leq \text{Number of nodes} \leq 10^5$

$1 \leq \text{Data of a node} \leq 10^9$

Solution:

```
class Tree
{
    boolean a=true;
```

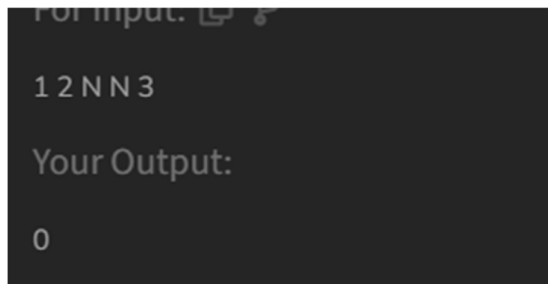
```

int solve(Node root){
    if(root==null) return 0;
    int l=solve(root.left);
    int r=solve(root.right);
    if(Math.abs(l-r)>1)a=false;
    return Math.max(l,r)+1;
}
//Function to check whether a binary tree is balanced or not.
boolean isBalanced(Node root)

{
    // Your code here
    solve(root);
    return a;
}
}

```

Output:



Time Complexity: $O(n)$
 Space Complexity: $O(h)$

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.

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.

Constraints:

$$2 \leq \text{val.size()} = \text{wt.size()} \leq 10^3$$

$$1 \leq \text{capacity} \leq 10^3$$

$$1 \leq \text{val}[i] \leq 10^3$$

$$1 \leq \text{wt}[i] \leq 10^3$$

Solution:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int testCases = Integer.parseInt(br.readLine());

        while (testCases-- > 0) {
            int capacity = Integer.parseInt(br.readLine());
            String[] valInput = br.readLine().split(" ");
            int[] val = new int[valInput.length];
            for (int i = 0; i < valInput.length; i++) {
                val[i] = Integer.parseInt(valInput[i]);
            }
            String[] wtInput = br.readLine().split(" ");
            int[] wt = new int[wtInput.length];
            for (int i = 0; i < wtInput.length; i++) {
                wt[i] = Integer.parseInt(wtInput[i]);
            }

            System.out.println(Solution.knapSack(capacity, val, wt));
            System.out.println("~");
        }
    }
}

class Solution {
    static int knapSack(int capacity, int val[], int wt[]) {
        int n = val.length;
        int dp[][] = new int[n][capacity+1];



        for(int w = wt[0]; w<=capacity; w++)
            dp[0][w] = val[0];
        for(int i = 0; i<n; i++)
            dp[i][0] = 0;

        for(int ind = 1; ind<n; ind++){
            for(int w = 1; w<=capacity; w++){
                int nTake = dp[ind-1][w];
                int take = (int)(-1e9);
                if(wt[ind]<=w)
                    take = val[ind]+dp[ind-1][w-wt[ind]];

                dp[ind][w] = Math.max(take, nTake);
            }
        }
    }
}
```

```
    }  
    }  
    return dp[n-1][capacity];  
    }  
}
```

Output:

For Input:  

4

1 2 3

4 5 1

Your Output:

3

Expected Output:

3

Time Complexity: $O(n \cdot \text{capacity})$.

Space Complexity: $O(n \cdot \text{capacity})$