Naresh M\_AIDS\_Practice\_Day-2

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).

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)

**Program:**

class Solution {

    // Function to return max value that can be put in knapsack of capacity.

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

        // code here

        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],val[i-1] + dp[i-1][w-wt[i-1]]);

                } else {

                    dp[i][w] = dp[i-1][w];

                }…..

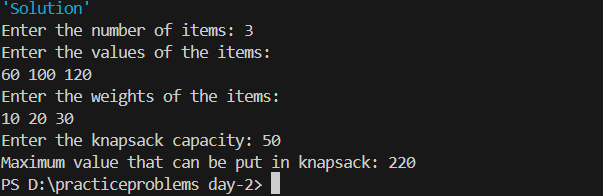
            }

        }

        return dp[n][capacity];

    }

}

**Time complexity(n\*w)**

**2.**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.

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.

**Program:**

import java.util.Scanner;

public class floor {

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

        int left = 0, right = arr.length - 1;

        int max=-1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (arr[mid] <= k) {

                max= mid;

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return max;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of elements in the array: ");

        int n = scanner.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements of the array:");

        for (int i = 0; i < n; i++) {

            arr[i] = scanner.nextInt();

        }

        System.out.print("Enter the value of k: ");

        int k = scanner.nextInt();

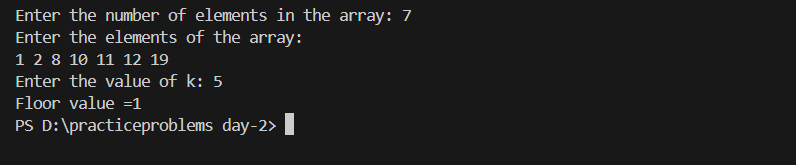
        scanner.close();

        int floorvalue= findFloor(arr, k);

        System.out.println("Floor value =" + floorvalue);

    }

}



**Time Complexity: O(n)**

**3.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.

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.

**Program:**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class checkEqualArray {

    public static boolean check(int[] arr1, int[] arr2) {

        if (arr1.length != arr2.length) return false;

        Map<Integer, Integer> map = new HashMap<>();

        for (int n : arr1) {

            map.put(n, map.getOrDefault(n, 0) + 1);

        }

        for (int n : arr2) {

            if (!map.containsKey(n)) {

                return false;

            }

            map.put(n, map.get(n) - 1);

        }

        for (int count : map.values()) {

            if (count != 0) return false;

        }

        return true;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the size of the arrays: ");

        int n = scanner.nextInt();

        int[] arr1 = new int[n];

        System.out.println("Enter elements of the first array:");

        for (int i = 0; i < n; i++) {

            arr1[i] = scanner.nextInt();

        }

        int[] arr2 = new int[n];

        System.out.println("Enter elements of the second array:");

        for (int i = 0; i < n; i++) {

            arr2[i] = scanner.nextInt();

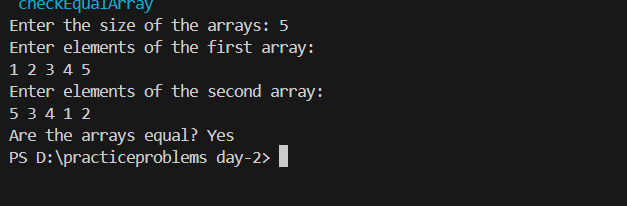
        }

        boolean result = check(arr1, arr2);

        System.out.println("Are the arrays equal? " + (result ? "Yes" : "No"));

    }

}



**Time complexity: O(n)**

**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.**

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.lang.\*;

import java.util.\*;

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

class LinkedListPalindrome {

static Node insert(Node head, int data) {

Node temp = new Node(data);

if (head == null) {

head = temp;

return head;

} else {

Node t = head;

while (t.next != null) {

t = t.next;

}

t.next = temp;

}

return head;

}

static void printList(Node head) {

Node temp = head;

while (temp != null) {

System.out.print(temp.data + " ");

temp = temp.next;

}

}

public static void main(String[] args) throws IOException {

BufferedReader read = new BufferedReader(new InputStreamReader(System.in));

int t = Integer.parseInt(read.readLine());

while (t-- > 0) {

Node head = null;

String str[] = read.readLine().trim().split(" ");

int listSize = str.length;

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

head = insert(head, Integer.parseInt(str[i]));

}

boolean f = new Solution().isPalindrome(head);

System.out.println(f ? "true" : "false");

}

}

}

class Solution {

boolean isPalindrome(Node head) {

Node curr = head;

Stack<Integer> s = new Stack<>();

while(curr != null){

s.push(curr.data);

curr = curr.next;

}

while(head != null){

int c = s.pop();

if(head.data != c){

return false;

}

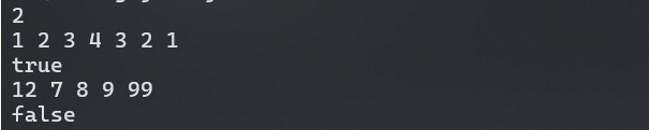
head = head.next;

}

return true;

}

}



**Time complexity: O(n)**

**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.

**Program:**

// Definition for a binary tree node.

class Node {

    int data;

    Node left, right;

    // Constructor to initialize the node

    Node(int d) {

        data = d;

        left = right = null;

    }

}

class Solution {

    // Function to check if the tree is balanced

    public int isBalanced(Node root) {

        if (checkHeight(root) == -1) {

            return 0; // Tree is unbalanced

        }

        return 1;

    }

    private int checkHeight(Node root) {

        if (root == null) {

            return 0;

        }

        int leftHeight = checkHeight(root.left);

        int rightHeight = checkHeight(root.right);

        if (leftHeight == -1 || rightHeight == -1) {

            return -1;

        }

        if (Math.abs(leftHeight - rightHeight) > 1) {

            return -1;

        }

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

    }

}

public class BalancedTreeCheck {

    public static void main(String[] args) {

        // Create a tree for testing

        Node root = new Node(10);

        root.left = new Node(20);

        root.right = new Node(30);

        root.left.left = new Node(40);

        root.left.right = new Node(60);

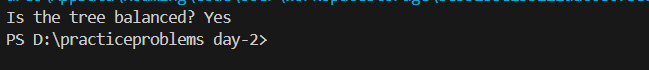
        Solution solution = new Solution();

        int result = solution.isBalanced(root);

        System.out.println("Is the tree balanced? " + (result == 1 ? "Yes" : "No"));

    }

}



**Time Complexity: O(n)**

**6.Triplet Sum of an 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.

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.

**Program:**

import java.util.Arrays;

import java.util.Scanner;

public class tripletSum {

    public static int find3Numbers(int[] arr, int n, int x) {

        Arrays.sort(arr);

        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) {

                    return 1;

                } else if (currentSum < x) {

                    left++;

                } else {

                    right--;

                }

            }

        }

        return 0;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the size of the array: ");

        int n = scanner.nextInt();

        int[] arr = new int[n];

        System.out.println("Enter the elements of the array:");

        for (int i = 0; i < n; i++) {

            arr[i] = scanner.nextInt();

        }

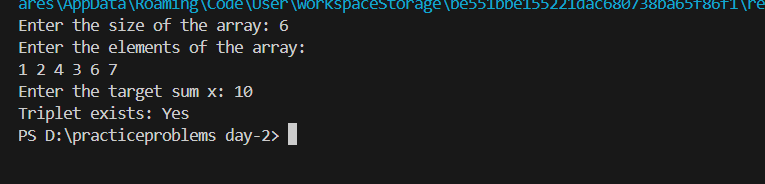
        System.out.print("Enter the target sum x: ");

        int x = scanner.nextInt();

        int result = find3Numbers(arr, n, x);

        System.out.println("Triplet exists: " + (result == 1 ? "Yes" : "No"));

    }

}  


**Time Complexity: O(n^2)**