Date: 11.11.2024

DSA PRACTICE DAY 2

1. 0/1 Knapsack Problem

Solution:

import java.util.Scanner;

public class Knapsack {

    public static void main(String[] args) {

        Scanner sc=new Scanner(System.in);

        System.out.println("Enter the number of objects: ");

        int n=sc.nextInt();

        System.out.print("Enter the weight of objects: ");

        int[] weight=new int[n];

        for(int i=0;i<n;i++) weight[i]=sc.nextInt();

        System.out.print("Enter the profit of objects: ");

        int[] profit=new int[n];

        for(int i=0;i<n;i++) profit[i]=sc.nextInt();

        System.out.println("Enter the net capacity of the sack: ");

        int capacity=sc.nextInt();

        sc.close();

        int[][] K=new int[n+1][capacity+1];

        for (int i=0;i<=n;i++){

            for (int w=0;w<=capacity;w++){

                if (i==0 || w==0) K[i][w]=0;

                else if (weight[i-1] <= w){

                    K[i][w]=Math.max(profit[i-1]+K[i-1][w-weight[i-1]],K[i-1][w]);

                }

                else K[i][w]=K[i-1][w];

            }

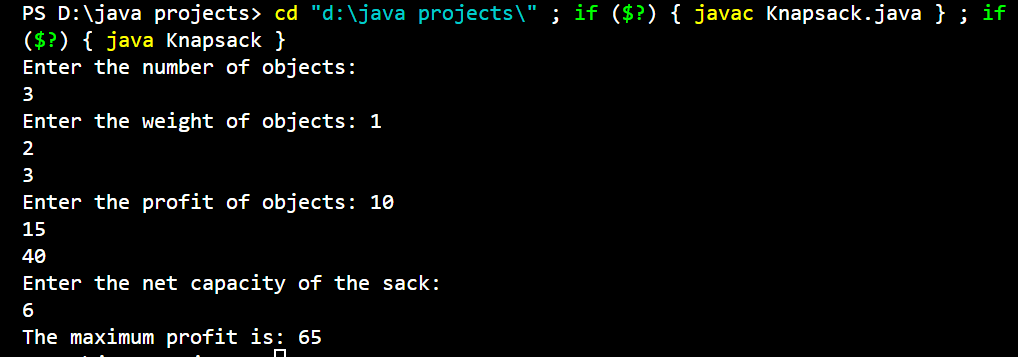
        }

        System.out.println("The maximum profit is: "+K[n][capacity]);

    }

}

Output:



Time Complexity: O(N\*W)

Space Complexity: O(N\*W)

1. Floor in Sorted Arrays

Given a sorted array and a value **x**, the floor of x is the largest element in the array smaller than or equal to x. Write efficient functions to find the floor of x

Solution:

import java.util.Scanner;

public class FloorInSortedArrays {

    public static void main(String[] args) {

        Scanner sc=new Scanner(System.in);

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

        int n=sc.nextInt();

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

        int[] arr=new int[n];

        for(int i=0;i<n;i++) arr[i]=sc.nextInt();

        System.out.print("Enter the element to search: ");

        int x=sc.nextInt();

        sc.close();

        System.out.println(Search(arr,x));

    }

    static int Search(int[] arr,int x){

        if (x<arr[0]) return -1;

        if (x>=arr[arr.length-1]) return arr[arr.length-1];

        int start=0;

        int end=arr.length-1;

        while(start<end){

            int mid=(start+end)/2;

            if(arr[mid]==x) return arr[mid];

            else if(x>arr[mid]){

                if (x<arr[mid+1]) return arr[mid];

                start=mid+1;

            }

            else end=mid-1;

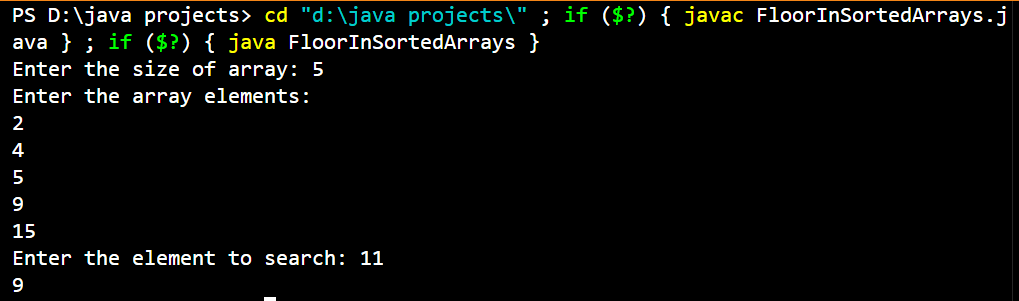
        }

        return -1;

    }

}

Solution:



Time Complexity: O(log N)

Space Complexity: O(1)

1. Check Equal Arrays

Given two arrays, arr1and arr2of equal lengthN, the task is to determine if the given arrays are equal or not.

Solution:

import java.util.Arrays;

import java.util.Scanner;

public class CheckEqualArrays {

    public static void main(String[] args) {

        Scanner sc=new Scanner(System.in);

        System.out.print("Enter the size of array 1: ");

        int n=sc.nextInt();

        System.out.println("Enter the array 1 elements: ");

        int[] arr1=new int[n];

        for(int i=0;i<n;i++) arr1[i]=sc.nextInt();

        System.out.print("Enter the size of array 2: ");

        int m=sc.nextInt();

        System.out.println("Enter the array 2 elements: ");

        int[] arr2=new int[m];

        for(int i=0;i<n;i++) arr2[i]=sc.nextInt();

        sc.close();

        if (CheckArrays(arr1,arr2)) System.out.println("YES.. the arrays are equal");

        else System.out.println("NO...the arrays are not equal");

    }

    static boolean CheckArrays(int[] arr1,int[] arr2){

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

        Arrays.sort(arr1);

        Arrays.sort(arr2);

        for(int i=0;i<arr1.length;i++) {

            if (arr1[i]!=arr2[i]) return false;

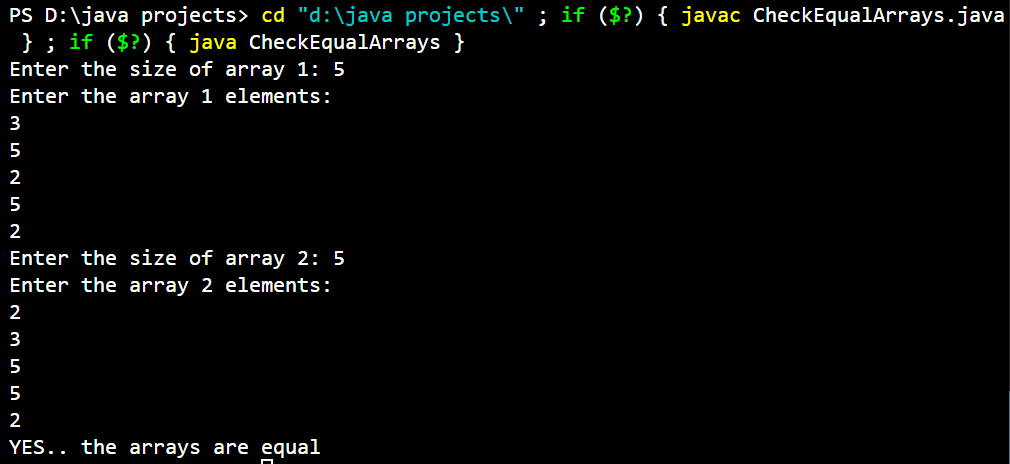
        }

        return true;

    }

}

Output:



Time Complexity: O(N\*log N)

Space Complexity: O(1)

1. Palindrome Linked List

Given a singlylinked list. The task is to check if the given linked list is palindromeor not.

Solution:

class Node {

    int data;

    Node next;

    Node(int d) {

        data = d;

        next = null;

    }

}

public class PalindromeLinkedList {

    static Boolean isPalindrome(Node head){

        Node slow=head,fast=head;

        while(fast.next!=null && fast.next.next!=null){

            slow=slow.next;

            fast=fast.next.next;

        }

        Node head2=reverse(slow.next);

            slow.next=null;

            return checkEqual(head,head2);

    }

    static Node reverse(Node head){

        Node prev=null,next,curr=head;

        while(curr!=null){

            next=curr.next;

            curr.next=prev;

            prev=curr;

            curr=next;

        }

        return prev;

    }

    static Boolean checkEqual(Node n1,Node n2){

        while(n1!=null && n2!=null){

            if(n1.data!=n2.data){

                return false;

            }

            n1=n1.next;

            n2=n2.next;

        }

        return true;

    }

    public static void main(String[] args) {

        Node head=new Node(2);

        head.next=new Node(4);

        head.next.next=new Node(5);

        head.next.next.next=new Node(4);

        head.next.next.next.next=new Node(2);

        Boolean res=isPalindrome(head);

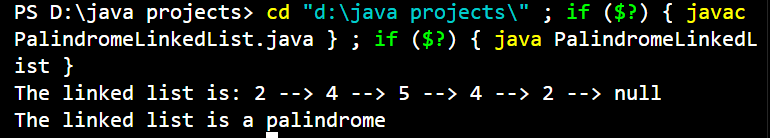
        if (res) System.out.println("The linked list is a palindrome");

        else System.out.println("The linked list is not a palindrome");

    }

}

Output:



Time Complexity: O(N)

Space Complexity: O(1)

1. Triplet sum of array

Given an array arr[] of size n and an integer sum. Find if there’s a triplet in the array which sums up to the given integer sum.

Solution:

import java.util.Scanner;

public class TripletSum {

    static Boolean CheckTriplet(int[] arr, int sum){

        int n = arr.length;

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

            int l=i+1;

            int r=n-1;

            while (l < r) {

                int s = arr[i] + arr[l] + arr[r];

                if (s==sum){

                    System.out.println("Found Triplet : " + arr[i] + " " + arr[l] + " " + arr[r]);

                    return true;

                }

                else if(s<sum) l++;

                else r--;

            }

        }

        return false;

    }

    public static void main(String[] args) {

        Scanner sc=new Scanner(System.in);

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

        int n=sc.nextInt();

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

        int[] arr=new int[n];

        for(int i=0;i<n;i++) arr[i]=sc.nextInt();

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

        int sum=sc.nextInt();

        sc.close();

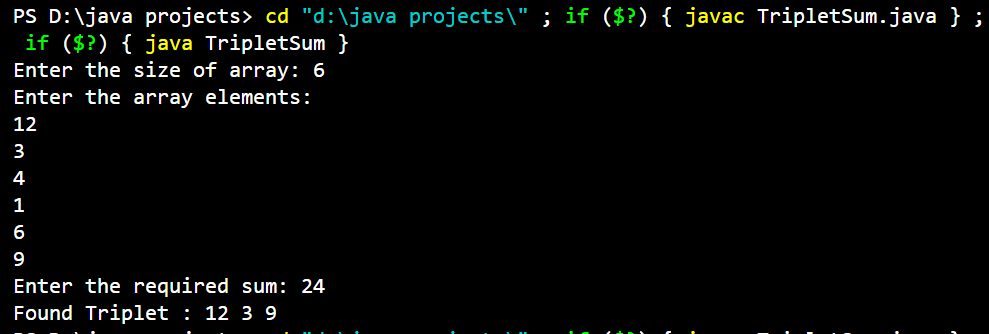
        boolean result=CheckTriplet(arr,sum);

        if(!result) System.out.println("No possible Triplet Sum");

    }

}

Output:



Time Complexity: O(N^2)

Space Complexity: O(1)

1. Balanced Tree Check

Solution:

class Node{

    int data;

    Node left,right;

    Node(int d){

        data = d;

        left=null;

        right=null;

    }

}

public class BalancedBinaryTree {

    static int isBalanced(Node root) {

        if (root == null) return 1;

        int lh=isBalanced(root.left);

        if(lh==-1) return -1;

        int rh=isBalanced(root.right);

        if(rh==-1) return -1;

        if(Math.abs(lh-rh)>1) return -1;

        return Math.max(lh,rh)+1;

    }

    public static void main(String[] args) {

        //Node root = null;

        Node root = new Node(1);

        root.left = new Node(2);

        root.right = new Node(3);

        root.left.left = new Node(4);

        root.left.right = new Node(5);

        root.right.left = new Node(6);

        root.right.right = new Node(7);

        root.left.left.left = new Node(8);

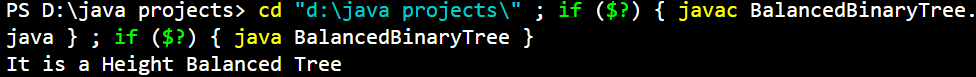
        if(isBalanced(root)>0) System.out.println("It is a Height Balanced Tree");

        else System.out.println("It is not a Height Balanced Tree");

    }

}

Output:



Time Complexity: O(N)

Space Complexity: O(h)