DSA Practice Solutions – 11.11.24

NAME: N J Meenakshi

DEPARTMENT: CSBS

ROLL NO.: 22CB028

1. **0/1 Knapsack Problem**
   Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.
   Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].
   Input: *N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}*
   Output: *3*
   Explanation: *There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.*

   Input: *N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}*
   Output: *0*

   Code:

```java
import java.util.Scanner;

public class Knapsack {
    public static int knapsack(int[][] t, int[] weights, int[] profit, int n, int w){
        if(n==0 || w==0) return 0;
        if(t[n][w]!=-1) return t[n][w];
        if (weights[n-1]<=w){
            t[n][w]=Math.max(profit[n-1]+knapsack(t, weights, profit, n-1, w-weights[n-
1]),knapsack(t, weights, profit, n-1, w));
            return t[n][w];
        }
        else if(weights[n-1]>w){
            t[n][w]=knapsack(t, weights, profit, n-1, w);
            return t[n][w];
        }
        return 0;
    }
    public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter n: ");
        int n=sc.nextInt();
        System.out.print("Enter w: ");
        int w=sc.nextInt();
        int[] profit = new int[n];
        int[] weights=new int[n];
        System.out.print("Enter profits: ");
        for(int i=0;i<n;i++){
            profit[i]=sc.nextInt();
        }
        System.out.print("Enter weights: ");
        for(int i=0;i<n;i++){
            weights[i]=sc.nextInt();
        }
        int[][] t=new int[weights.length+1][w+1];
        for(int i = 0; i <= n; i++) {
            for(int j = 0; j <= w; j++) {
                t[i][j] = -1;
            }
        }
        System.out.print("The max profit is: "+knapsack(t, weights, profit, n, w));

        sc.close();
    }
}
```

Output:

```
Enter n: 3
Enter w: 4
Enter profits: 1 2 3
Enter weights: 4 5 1
The max profit is: 3

Enter n: 3
Enter w: 3
Enter profits: 1 2 3
Enter weights: 4 5 6
The max profit is: 0
```

Time Complexity: O(2^N)

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.

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

Code:

```java
import java.util.Scanner;

public class Floor_in_sorted_arr {
    public static int solution(int arr[], int k){
        int l=0,r=arr.length-1;
        while(l<=r){
            if (k>=arr[r]){
                return r;
            }
            int mid=(l+r)/2;
            if (arr[mid]==k){
                return mid;
            }
            if (mid>0 && arr[mid-1]<=k && k<arr[mid]){
                return mid-1;
            }
            if(k<arr[mid]) r=mid-1;
            else l=mid+1;
        }
        return -1;
    }
    public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter n: ");
        int n=sc.nextInt();
        System.out.print("Enter k: ");
        int k=sc.nextInt();
        int[] arr=new int[n];
        for(int i=0;i<n;i++){
            arr[i]=sc.nextInt();
        }
        System.out.print("Floor in the array is: "+solution(arr, k));
        sc.close();
    }
}
```

Output:

```
Enter n: 7
Enter k: 0
1 2 8 10 11 12 19
Floor in the array is: -1

Enter n: 7
Enter k: 5
1 2 8 10 11 12 19
Floor in the array is: 1
```

```
Enter n: 3
Enter k: 1
1 2 8
Floor in the array is: 0
```

Time Complexity: O(logN)

3. **Check Equal Arrays**

Given two arrays, **arr1** and **arr2** of equal length **N**, the task is to determine if the given arrays are equal or not. Two arrays are considered equal if:

- Both arrays contain the same set of elements.
- The arrangements (or permutations) of elements may be different.
- If there are repeated elements, the counts of each element must be the same in both arrays.

***Input:** arr1[] = {1, 2, 5, 4, 0}, arr2[] = {2, 4, 5, 0, 1}*
***Output:** Yes*

***Input:** arr1[] = {1, 2, 5, 4, 0, 2, 1}, arr2[] = {2, 4, 5, 0, 1, 1, 2}*
***Output:** Yes*

***Input:** arr1[] = {1, 7, 1}, arr2[] = {7, 7, 1}*
***Output:** No*

Code:

```java
import java.util.Scanner;
import java.util.HashMap;

public class Check_Equal_Arr {
    public static boolean solution(int[] arr1, int[] arr2){
        if(arr1.length!=arr2.length) return false;
        HashMap<Integer,Integer> count= new HashMap<>();
        for(int i:arr1){
            count.put(i, count.getOrDefault(i, 0)+1);
        }
        for(int i:arr2){
            if(!count.containsKey(i) || count.get(i)==0) return false;
            count.put(i, count.get(i)-1);
        }
        return true;
    }
    public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter n: ");
        int n=sc.nextInt();
        int[] arr1=new int[n];
        System.out.print("Enter arr1: ");
        for(int i=0;i<n;i++){
            arr1[i]=sc.nextInt();
        }
        int[] arr2=new int[n];
        System.out.print("Enter arr2: ");
        for(int i=0;i<n;i++){
            arr2[i]=sc.nextInt();
        }
        System.out.print("Equal arrays: "+solution(arr1, arr2));
        sc.close();
    }
}
```

Output:

```
Enter n: 5
Enter arr1: 1 2 5 4 0
Enter arr2: 2 4 5 0 1
Equal arrays: true

Enter n: 7
Enter arr1: 1 2 5 4 0 2 1
Enter arr2: 2 4 5 0 1 1 2
Equal arrays: true

Enter n: 3
Enter arr1: 1 7 1
Enter arr2: 7 7 1
Equal arrays: false
```

Time Complexity: O(N)

4. **Palindrome linked list**

   Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

   Input: head = [1,2,2,1]
   Output: true

   Input: head = [1,2]
   Output: false

   Code:

```
class Node{
    int data;
    Node next;
    Node(int val, Node next){
        this.data=val;
        this.next=next;
    }
    Node(int data){
        this.data=data;
        this.next=null;
    }
}
```

```java
public class Palindromic_LL {
    public static Node reverseLL(Node head){
        if(head==null || head.next==null) return head;
        Node newhead=reverseLL(head.next);
        Node front=head.next;
        front.next=head;
        head.next=null;
        return newhead;
    }
    public static boolean isPal(Node head){
        if(head==null || head.next==null) return true;
        Node slow=head, fast=head;
        while (fast.next!=null && fast.next.next!=null) {
            slow=slow.next;
            fast=fast.next.next;
        }
        Node newhead=reverseLL(slow.next);
        Node first=head;
        Node second=newhead;
        while (second!=null) {
            if(first.data!=second.data){
                reverseLL(newhead);
                return false;
            }
            first=first.next;
            second=second.next;
        }
        reverseLL(newhead);
        return true;
    }
    public static void main(String args[]){
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(2);
        head.next.next.next = new Node(1);
        System.out.print(isPal(head));
    }
}
```

Output:

```
true

false
```

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.

   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.

Code:

```java
class Node {
    int data;
    Node left;
    Node right;
    Node(int val){
        data=val;
        left=null;
        right=null;
    }
}
public class Balanced_Tree {
    public static int dfs(Node root){
        if(root==null) return 0;
        int leftH=dfs(root.left);
        if(leftH==-1) return -1;
        int rightH=dfs(root.right);
        if(rightH==-1) return -1;
        if(Math.abs(leftH-rightH)>1) return -1;
        return Math.max(leftH, rightH)+1;
    }
    public static void main(String args[]){
        Node root = new Node(1);
        root.left = new Node(2);
        root.left.right = new Node(3);
        System.out.print(dfs(root));
    }
}
```

Output:
```
0

1
```
Time Complexity: O(N)

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.

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.

Code:

```java
import java.util.Arrays;
import java.util.Scanner;

public class Triplet_Sum {
    public static boolean solution(int[] arr, int sum){
        int n=arr.length;
        Arrays.sort(arr);
        for(int i=0;i<n;i++){
            int l=i+1;
            int r=n-1;
            while (l<r) {
                int curr_sum=arr[i]+arr[l]+arr[r];
                if(curr_sum==sum) return true;
                else if(curr_sum<sum) l++;
                else r--;
            }
        }
        return false;
    }
    public static void main(String args[]){
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter n: ");
        int n=sc.nextInt();
        System.out.print("Enter x: ");
        int x=sc.nextInt();
        int[] arr=new int[n];
        System.out.print("Enter arr: ");
        for(int i=0;i<n;i++){
            arr[i]=sc.nextInt();
        }
        System.out.print("Triplet sum exists: "+solution(arr, x));
        sc.close();
    }
}
```

Output:

```
Enter n: 6
Enter x: 13
Enter arr: 1 4 45 6 10 8
Triplet sum exists: true

Enter n: 6
Enter x: 10
Enter arr: 1 2 4 3 6 7
Triplet sum exists: true

Enter n: 6
Enter x: 24
Enter arr: 40 20 10 3 6 7
Triplet sum exists: false
```

Time Complexity: O(N^2)