DSA Practice 2 (11-11-2024)

Name: Atchayaa T

Department: B.Tech CSBS

Register No.: 22CB004

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].

Examples:

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:

```
class Knapsack01 {
    public static int solution(int w, int wt[], int val[], int n)
    {
        if (n == 0 || w == 0)
            return 0;
        if (wt[n - 1] > w)
            return solution(w, wt, val, n - 1);
        else
            return Math.max(solution(w, wt, val, n - 1),
             val[n - 1] + solution(w - wt[n-1], wt, val, n-1));
    }

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

```
        int w = 5;
        int n = pr.length;
        System.out.println(solution(w, wt, pr, n));
    }
}
```

Output:

50

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.

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

Code:

```
import java.io.*;
```

```
class FloorInASortedArray {

    static int solution(int arr[], int l, int h,
                        int x)
    {
        if (l > h)
            return -1;
        if (x >= arr[h])
            return h;
        int mid = (l + h) / 2;
        if (arr[mid] == x)
            return mid;
        if (mid > 0 && arr[mid - 1] <= x && x < arr[mid])
            return mid - 1;
        if (x < arr[mid])
            return solution(arr, l, mid - 1, x);

        return solution(arr, mid + 1, h, x);
    }


    public static void main(String[] args)
    {
        int arr[] = {1, 2, 8, 10, 11, 12, 19};
        int n = arr.length;
        int x = 5;
        int index = solution(arr, 0, n - 1, x);
        if (index == -1)
            System.out.println(
                "Floor of " + x+ " doesn't exist in array ");
        else
            System.out.println("Floor of " + x + " is at "+ index);
    }
}
```

Output:

Floor of 5 is at 1

Time Complexity: O(log n)

3. Check Equal Arrays

Given two given arrays of equal length, the task is to find if given arrays are equal or not. Two arrays are said to be equal if both of them contain the same set of elements and in the same order.

Examples:

Input : arr1[] = {1, 2, 5, 4, 0}; arr2[] = {1, 2, 5, 4, 0};

Output : Yes

Input : arr1[] = {1, 2, 5, 4, 0, 2}; arr2[] = {2, 4, 5, 0};

Output : No

Input : arr1[] = {1, 7, 7}; arr2[] = {7, 7, 1};

Output : No

Code:

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

class CheckEqualArrays {

    public static boolean solution(int arr1[], int arr2[])
    {
        int n = arr1.length;
        int m = arr2.length;

        if (n != m)
            return false;

        Map<Integer, Integer> h= new HashMap<Integer, Integer>();
        int count = 0;
        for (int i = 0; i < n; i++) {
            if (h.get(arr1[i]) == null)
                h.put(arr1[i], 1);
            else {
                count = h.get(arr1[i]);
                count++;
                h.put(arr1[i], count);
            }
        }

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

            if (!h.containsKey(arr2[i]))
                return false;
            if (h.get(arr2[i]) == 0)
                return false;

            count = h.get(arr2[i]);
             --count;
            h.put(arr2[i], count);
        }

        return true;
    }
```

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

        if (solution(arr1, arr2))
            System.out.println("Yes");
        else
            System.out.println("No");
    }
}
```

Output:

Yes

Time Complexity: O(n)

4. Palindrome linked list

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

Examples:
Input: head: 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: head: 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.

Code:

```
import java.util.Stack;
class Node {
    int data;
    Node next;
    Node(int d) {
        data = d;
        next = null;
    }
}

class PalindromeLinkedList{

    static boolean solution(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;
    }

    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);

        boolean r = solution(head);

        if (r)
            System.out.println(true);
        else
            System.out.println(false);
    }
}
```

Output:

true

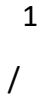
Time Complexity: O(n)

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:



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:

```
import java.io.*;
import java.lang.*;
import java.util.*;

class Node {
    int key;
    Node left;
    Node right;
    Node(int k)
    {
        key = k;
        left = right = null;
    }
}

class BalancedTreeCheck {

    public static int solution(Node root)
    {
        if (root == null)
            return 0;
        int lh = solution(root.left);
        if (lh == -1)
            return -1;
        int rh = solution(root.right);
        if (rh == -1)
```

```
            return -1;

        if (Math.abs(lh - rh) > 1)
            return -1;
        else
            return Math.max(lh, rh) + 1;
    }

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

        if (solution(root) > 0)
            System.out.print("Balanced");
        else
            System.out.print("Not Balanced");
    }
}
```

Output:

Balanced

Time Complexity: O(n)

6. Triplet Sum in 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**.

Examples:

Input: arr = {12, 3, 4, 1, 6, 9}, sum = 24;
Output: 12, 3, 9
Explanation: There is a triplet (12, 3 and 9) present in the array whose sum is 24.

Input: arr = {1, 2, 3, 4, 5}, sum = 9
Output: 5, 3, 1
Explanation: There is a triplet (5, 3 and 1) present in the array whose sum is 9.

Input: arr = {2, 10, 12, 4, 8}, sum = 9
Output: No Triplet
Explanation: We do not print in this case and return false.

Code:

```
import java.util.Arrays;

class TripletSum {
    static boolean solution(int[] arr, int sum)
    {
        int n = arr.length;

        Arrays.sort(arr);

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

            int l = i + 1;
            int r = n - 1;

            while (l < r) {
                int curr = arr[i] + arr[l] + arr[r];
                if (curr == sum) {
                    System.out.println("Triplet is " + arr[i] + ", "+ arr[l] + ", " +
arr[r]);
                    return true;
                }
                else if (curr < sum) {
                    l++;
                }
                else {
                    r--;
                }
            }
        }
        return false;
    }

    public static void main(String[] args)
    {
        int[] arr = {12, 3, 4, 1, 6, 9};
        int sum = 24;

        solution(arr, sum);
    }
}
```

Output:

Triplet is 3, 9, 12

Time Complexity: O(n^2)