Gopika shree M

22AD034

AI&DS

1) 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.

**Expected Time Complexity:** O(n\*capacity).  
**Expected Auxiliary Space:** O(n\*capacity)

**Constraints:**  
2 ≤ val.size() = wt.size() ≤ 103  
1 ≤ capacity ≤ 103  
1 ≤ val[i] ≤ 103  
1 ≤ wt[i] ≤ 103

class Solution {

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

int n = val.length;

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

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= capacity; j++) {

if (wt[i - 1] <= j) {

dp[i][j] = Math.*max*(val[i - 1] + dp[i - 1][j - wt[i - 1]], dp[i - 1][j]);

} else {

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

}

}

}

return dp[n][capacity];

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

int capacity1 = 4;

int[] val1 = {1, 2, 3};

int[] wt1 = {4, 5, 1};

System.***out***.println(solution.knapSack(capacity1, val1, wt1));

}

}

2) 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 ≤ arr.size() ≤ 106  
1 ≤ arr[i] ≤ 106  
0 ≤ k ≤arr[n-1]

public class Main {

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

int left = 0;

int right = arr.length -1;

int index = -1;

while (left <= right){

int mid = (left+right)/2;

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

else if (k < arr[mid]) {

right = mid-1;

}

else {

index = mid;

left = mid+1;

}

}

return index;

}

public static void main(String[] args) {

Main obj = new Main();

int res = obj.*findFoor*(new int[]{1,2,8,10,11,12,19}, 0);

System.***out***.println(res);

}

}

3) 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.

**Expected Time Complexity**: O(n)  
**Expected Space Complexity**: O(n)

**Constraints:**  
1<= arr1.size, arr2.size<=107  
0<=arr1[], arr2[]<=109

package problems;

import java.util.Map;

import java.util.HashMap;

public class Main {

public static boolean CheckArrays(int[] a, int[] b) {

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

for (int i: a) {

set.put(i, set.getOrDefault(set, 0)+1);

}

for (int j: b) {

if (!set.containsKey(j)) {

return false;

}

set.put(j, set.get(j)-1);

if (set.getOrDefault(j,0) == 0){

set.remove(j);

}

}

if (set.isEmpty()) return true;

return false;

}

public static void main(String[] args) {

Main obj = new Main();

boolean res = obj.*CheckArrays*(new int[]{1,2,5}, new int[]{5,0,1});

System.***out***.println(res);

}

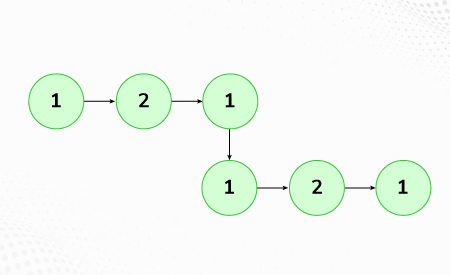
}

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

**Examples:**

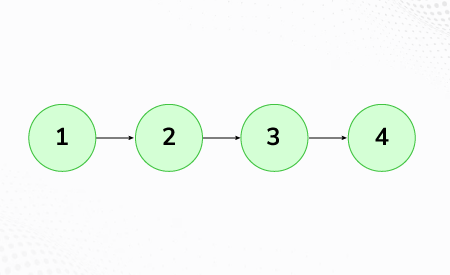
**Input:** LinkedList: 1->2->1->1->2->1

**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.  


**Expected Time Complexity**: O(n)  
**Expected Auxiliary Space**: O(1) 

**Constraints:**  
1 <= number of nodes <= 1051 ≤ node->data ≤ 103

public class Main{

public static class ListNode {

int val;

ListNode next;

ListNode(int x) {

val = x;

next = null;

}

}

public boolean isPalindrome(ListNode head) {

if (head == null || head.next == null) return true;

ListNode slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

ListNode secondHalfStart = reverseList(slow);

ListNode fStart = head;

ListNode Iter = secondHalfStart;

boolean True = true;

while (True && Iter != null) {

if (fStart.val != Iter.val) {

True = false;

}

fStart = fStart.next;

Iter = Iter.next;

}

return True;

}

private ListNode reverseList(ListNode head) {

ListNode prev = null;

while (head != null) {

ListNode nextNode = head.next;

head.next = prev;

prev = head;

head = nextNode;

}

return prev;

}

public static void main(String[] args) {

Main solution = new Main();

ListNode head = new ListNode(1);

head.next = new ListNode(2);

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

head.next.next.next = new ListNode(1);

head.next.next.next.next = new ListNode(1);

System.***out***.println(solution.isPalindrome(head));

}

}

5) 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 <= Number of nodes <= 105  
1 <= Data of a node <= 109

**Expected time complexity:**O(N)  
**Expected auxiliary space:**O(h) , where h = height of tree

public class Main{

public static class TreeNode{

int val;

TreeNode left;

TreeNode right;

TreeNode(int x){

val = x;

left = null;

right = null;

}

}

public boolean isBalanced(TreeNode root) {

return checkHeight(root) != -1;

}

public int checkHeight(TreeNode node) {

if (node == null) return 0;

int leftHeight = checkHeight(node.left);

if (leftHeight == -1) return -1;

int rightHeight = checkHeight(node.right);

if (rightHeight == -1) return -1;

if (Math.*abs*(rightHeight-leftHeight) > 1) return -1;

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

}

public static void main(String[] args) {

Main solution = new Main();

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

// root.right = new TreeNode(3);

root.left.left = new TreeNode(4);

// root.left.right = new TreeNode(5);

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

System.***out***.println(solution.isBalanced(root));

}

}

6) 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.

**Expected Time Complexity:**O(n2)  
**Expected Auxiliary Space:**O(1)

**Constraints:**  
1 ≤ n ≤ 103  
1 ≤ arr[i] ≤ 105

import java.util.Arrays;

public class Main {

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

Arrays.*sort*(arr);

int n = arr.length;

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

int left = i +1;

int right = n-1;

while (left < right){

int curr = arr[i]+arr[left]+arr[right];

if (curr == k) return 1;

else if (curr < k) {

left += 1;

}

else{

right -= 1;

}

}

}

return 0;

}

public static void main(String[] args) {

Main obj = new Main();

int res = obj.*TripleSum*(new int[]{1,4,45,6,10,8}, 13);

System.***out***.println(res);

}

}