**SDE ASSIGNMENT 11-11-2024**

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.

CODE:

**package** src;

**public** **class** Knapsack {

**public** **static** **int** findMax(**int** n1, **int** n2) {

**if**(n1>n2) {

**return** n1;

} **else** {

**return** n2;

}

}

**public** **static** **int** knapsack(**int** W, **int** wt[], **int** val[], **int** n) {

**int** K[][] = **new** **int**[n+1][W+1];

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

**for**(**int** w = 0; w<=W; w++) {

**if**(i == 0 || w == 0) {

K[i][w] = 0;

} **else** **if**(wt[i-1] <= w) {

K[i][w] = *findMax*(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);

} **else** {

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

}

}

}

**return** K[n][W];

}

**public** **static** **void** main(String[] args) {

**int**[] val = {10, 40, 30, 50};

**int**[] wt = {5, 4, 6, 3};

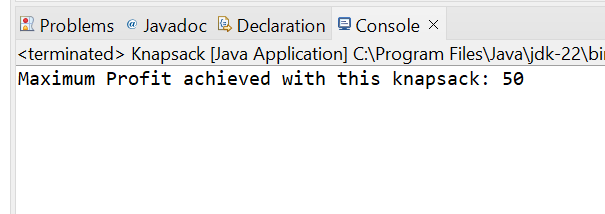
**int** capacity = 5;

**int** len = val.length;

System.***out***.print("Maximum Profit achieved with this knapsack: " + *knapsack*(capacity, wt, val, len));

}

}



TIME COMPLEXITY:O(n\*capacity)

2. **FLOOR SORTED ARRAY**

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 .

Input: arr[] = {1, 2, 8, 10, 10, 12, 19}, x = 5

Output: 2

Explanation: 2 is the largest element in arr[] smaller than 5

CODE:

**package** src;

**public** **class** floor {

**static** **int** floorSearch(**int** arr[], **int** n, **int** x)

{

**if** (x >= arr[n - 1])

**return** n - 1;

**if** (x < arr[0])

**return** -1;

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

**if** (arr[i] > x)

**return** (i - 1);

**return** -1;

}

**public** **static** **void** main(String[] args)

{

**int** arr[] = { 1, 2, 4, 6, 10, 12, 14 };

**int** n = arr.length;

**int** x = 7;

**int** index = *floorSearch*(arr, n - 1, x);

**if** (index == -1)

System.***out***.print("Floor of " + x

+ " doesn't exist in array ");

**else**

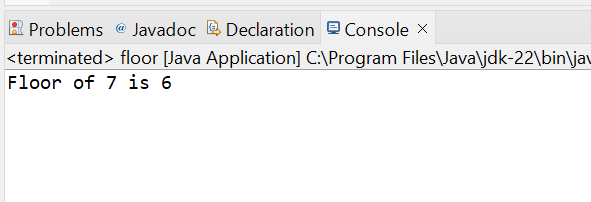
System.***out***.print("Floor of " + x + " is "

+ arr[index]);

}

}

OUTPUT:



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]

CODE:

package src;

import java.util.Arrays;

public class Equalarray {

public static boolean areEqual(int arr1[], int arr2[])

{

int n = arr1.length;

int m = arr2.length;

if (n != m)

return false;

Arrays.sort(arr1);

Arrays.sort(arr2);

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

if (arr1[i] != arr2[i])

return false;

return true;

}

public static void main(String[] args)

{

int arr1[] = { 1, 2, 5, 4, 0 };

int arr2[] = {2, 4, 5, 0, 1};

if (areEqual(arr1, arr2))

System.out.println("Yes");

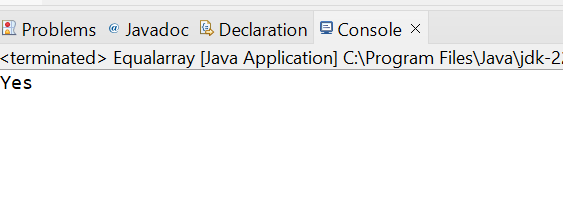
else

System.out.println("No");

}

}

OUTPUT:



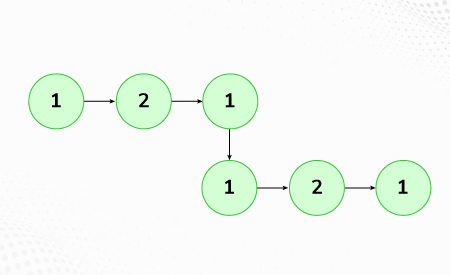
TIME COMPLEXITY:O(N\*log(N))

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.  


CODE:

**package** src;

**import** java.util.Stack;

**class** Node {

**int** data;

Node next;

Node(**int** d) {

data = d;

next = **null**;

}

}

**public** **class** Palindrome {

**static** **boolean** isPalindrome(Node head) {

Node currNode = head;

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

**while** (currNode != **null**) {

s.push(currNode.data);

currNode = currNode.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(3);

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

head.next.next.next.next = **new** Node(1);

**boolean** result = *isPalindrome*(head);

**if** (result)

System.***out***.println("true");

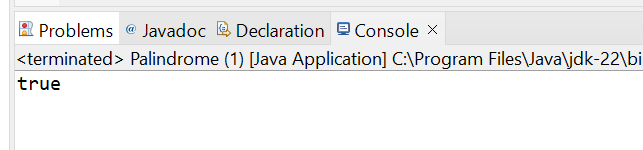
**else**

System.***out***.println("false");

}

}

OUTPUT:

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

CODE:

package src;

class TreeNode {

int data;

TreeNode left, right;

public TreeNode(int data) {

this.data = data;

left = right = null;

}

}

public class Balanced {

public boolean isBalanced(TreeNode root) {

return checkHeight(root) != -1;

}

private 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(leftHeight - rightHeight) > 1) return -1;

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

}

public static void main(String[] args) {

Balanced tree = new Balanced();

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

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

if (tree.isBalanced(root)) {

System.out.println("1");

} else {

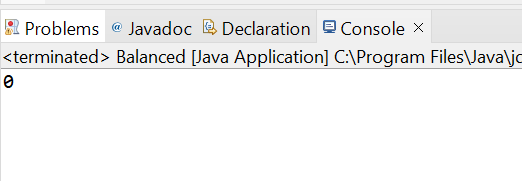
System.out.println("0");

}

}

}

OUTPUT:



TIME COMPLEXITY:O(N)

6.**TRIPLET SUM IN A 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. 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.

CODE:

package src;

public class Triplet {

static boolean find3Numbers(int[] arr, int sum)

{

int n = arr.length;

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

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

for (int k = j + 1; k < n; k++) {

if (arr[i] + arr[j] + arr[k] == sum) {

System.out.println(

"Triplet is " + arr[i] + ", "

+ arr[j] + ", " + arr[k]);

return true;

}

}

}

}

return false;

}

public static void main(String[] args)

{

int[] arr = { 1, 4, 45, 6, 10, 8 };

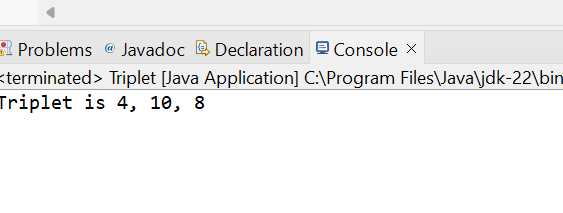
int sum = 22;

find3Numbers(arr, sum);

}

}

OUTPUT:



TIME COMPLEXITY:O(N^3)