**CODING PRACTICE PROBLEMS DATE:11/11/24**

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).

**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 util;

public class knapsack {

public static 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 w = 0; w <= capacity; w++) {

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

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

} else {

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

}

}

}

return dp[n][capacity];

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

int capacity = 4;

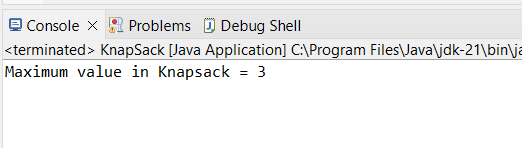
int[] val = {1, 2, 3};

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

System.***out***.println("Maximum value in Knapsack = " + *knapSack*(capacity, val, wt));

}

}



Time Complexity:O(n\*capacity)

**2**.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 util;

public class floored {

public static int findFloor(int[] arr, int x) {

int left = 0, right = arr.length - 1;

int floor = -1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == x) {

return arr[mid];

} else if (arr[mid] < x) {

floor = arr[mid];

left = mid + 1;

} else {

right = mid - 1;

}

}

return floor;

}

public static void main(String[] args) {

int[] arr = {1, 2, 8, 10, 10, 12, 19};

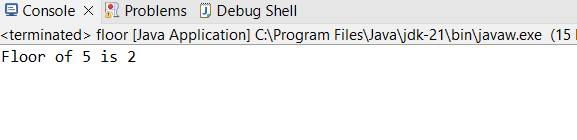
int x = 5;

int result = *findFloor*(arr, x);

System.***out***.println("Floor of " + x + " is " + result);

}

}



Time Complexity: O(logn)

**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.

**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 util;

import java.io.\*;

import java.util.\*;

public class equalarrays {

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[] = { 3, 5, 2, 5, 2 };

int arr2[] = { 2, 3, 5, 5, 2 };

if (*areEqual*(arr1, arr2))

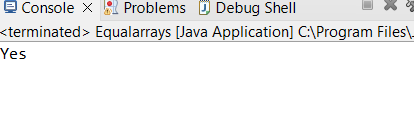
System.***out***.println("Yes");

else

System.***out***.println("No");

}

}

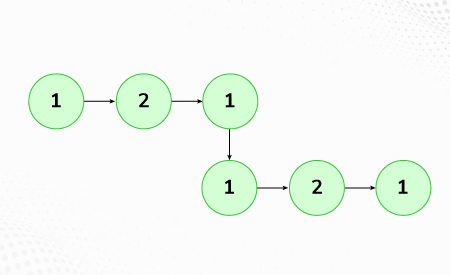


Time Complexity:  O(N\*log(N))

**4.**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:

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

public class Main {

Node head;

public boolean isPalindrome() {

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

Node slow = head;

Node fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

Node secondHalfStart = reverseList(slow);

Node firstHalf = head;

Node secondHalf = secondHalfStart;

boolean isPalindrome = true;

while (isPalindrome && secondHalf != null) {

if (firstHalf.data != secondHalf.data) {

isPalindrome = false;

}

firstHalf = firstHalf.next;

secondHalf = secondHalf.next;

}

reverseList(secondHalfStart);

return isPalindrome;

}

private Node reverseList(Node head) {

Node prev = null;

Node current = head;

while (current != null) {

Node nextNode = current.next;

current.next = prev;

prev = current;

current = nextNode;

}

return prev;

}

public void append(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

return;

}

Node last = head;

while (last.next != null) {

last = last.next;

}

last.next = newNode;

}

public static void main(String[] args) {

Main list = new Main(); // Use Main here

int[] elements = {1, 2, 1, 1, 2, 1};

for (int element : elements) {

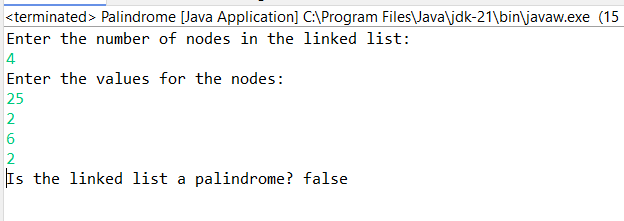
list.append(element);

}

System.out.println("Is the linked list a palindrome? " + list.isPalindrome());

}

}



Time Complexity:O(n)

**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.

**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:

class TreeNode {

int data;

TreeNode left, right;

public TreeNode(int data) {

this.data = data;

left = right = null;

}

}

public class Main {

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) {

Main tree = new Main();

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 {

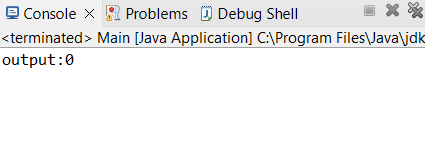
System.out.print("output:");

System.out.println("0");

}

}

}



Time Complexity:O(n)

**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.

CODE:

class Solution {

public static boolean find3Numbers(int arr[], int n, int x) {

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]==x){

return true;

}

}

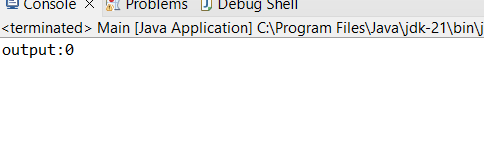
}

}

return false;

}

}



Time Complexity:O(n^2)