**SANJAY\_M – CSE – DSA – PRACTICE – 3**

**Q1**. **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:

class knapsack {

static int knapSack(int W, int wt[], int val[], int n) {

if (n == 0 || W == 0)

return 0;

if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

else

return Math.max(knapSack(W, wt, val, n - 1),

val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1));

}

public static void main(String args[]) {

int profit[] = new int[] { 1, 2, 3 };

int weight[] = new int[] { 4, 5, 1 };

int W = 4;

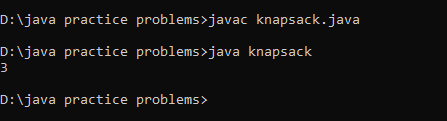
int N = profit.length;

System.out.println(knapSack(W, weight, profit, N));

}

}

OUTPUT:



Time Complexity: O(2^n)

**Q2. Floor in 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.

CODE:

class floorinarray {

static int findFloor(long arr[], int n, long k) {

int low = 0;

int high = n - 1;

int index = -1;

while (low <= high) {

int mid = low + (high - low) / 2;

if (arr[mid] == k) {

return mid;

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

index = mid;

low = mid + 1;

} else {

high = mid - 1;

}

}

return index;

}

public static void main(String[] args) {

long[] arr = {1, 2, 8, 10, 11, 12, 19};

long k = 0;

int n = arr.length;

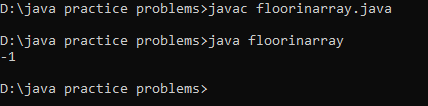
int result = findFloor(arr, n, k);

System.out.println(result);

}

}

OUTPUT:



Time Complexity: O(log n)

**Q3. 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:

class equalarray {

public static boolean check(int[] arr1, int[] arr2) {

int count = 0;

for (int i = 0; i < arr1.length; i++) {

for (int j = 0; j < arr2.length; j++) {

if (arr1[i] == arr2[j]) {

count++;

arr2[j] = -1;

break;

}

}

}

return count == arr1.length;

}

public static void main(String[] args) {

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

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

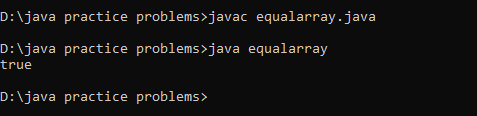
boolean result = check(arr1, arr2);

System.out.println(result);

}

}

OUTPUT:



Time Complexity: O(n \* m)

**Q4. Palindrome linked list**

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

https://assets.leetcode.com/uploads/2021/03/03/pal1linked-list.jpg

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

Output: true

https://assets.leetcode.com/uploads/2021/03/03/pal2linked-list.jpg

Input: head = [1,2]

Output: false

CODE:

import java.util.ArrayList;

import java.util.List;

class ListNode {

int val;

ListNode next;

ListNode(int x) {

val = x;

next = null;

}

}

class Palindromell {

public boolean isPalindrome(ListNode head) {

List<Integer> list = new ArrayList<>();

while (head != null) {

list.add(head.val);

head = head.next;

}

int left = 0;

int right = list.size() - 1;

while (left < right && list.get(left) == list.get(right)) {

left++;

right--;

}

return left >= right;

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

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

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

Palindromell pll = new Palindromell();

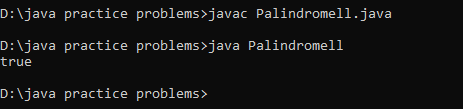
boolean result = pll.isPalindrome(head);

System.out.println(result);

}

}

OUTPUT:



Time Complexity: O(n)

**Q5. 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:

class Node {

int data;

Node left, right;

Node(int data) {

this.data = data;

left = right = null;

}

}

class BalancedTree {

static int check(Node node) {

if (node == null) return 0;

int l = check(node.left);

int r = check(node.right);

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

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

if (Math.abs(l - r) > 1)

return -1;

return Math.max(l, r) + 1;

}

boolean isBalanced(Node root) {

return check(root) != -1 ? 1 : 0;

}

public static void main(String[] args) {

Node root = new Node(1);

root.left = new Node(2);

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

BalancedTree tree = new BalancedTree();

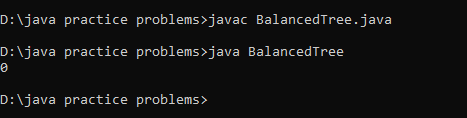
boolean result = tree.isBalanced(root);

System.out.println("Is the tree balanced? " + result);

}

}

OUTPUT:



Time Complexity: O(n)

**Q6. 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.

CODE:

import java.util.Arrays;

class tripletsum {

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

Arrays.sort(arr);

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

if (i > 0 && arr[i] == arr[i - 1]) {

continue;

}

int j = i + 1;

int k = n - 1;

while (j < k) {

int sum = arr[i] + arr[j] + arr[k];

if (sum == x) {

return true;

} else if (sum < x) {

j++;

} else {

k--;

}

}

}

return false;

}

public static void main(String[] args) {

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

int n = arr.length;

int x = 13;

boolean result = find3Numbers(arr, n, x);

if (result) {

System.out.println(1);

} else {

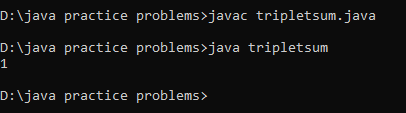
System.out.println(0);

}

}

}

OUTPUT:



Time Complexity: O(n^2)