Harshitha R AIDS A 22AD041 11/11/2024

1. **0 - 1 Knapsack Problem**

Difficulty: MediumAccuracy: 31.76%Submissions: 447K+Points: 4

You are given the weights and values of items, and you need to put these items in a knapsack of 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

**Code**

class Solution {

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

int n=val.length;

if (capacity <= 0 || n == 0 || wt == null || val == null ||

wt.length != val.length) {

return 0;

}

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

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

for (int j = 0; j <= capacity; j++) {

if (i == 0 || j == 0) {

dp[i][j] = 0;

}

else 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];

}

}

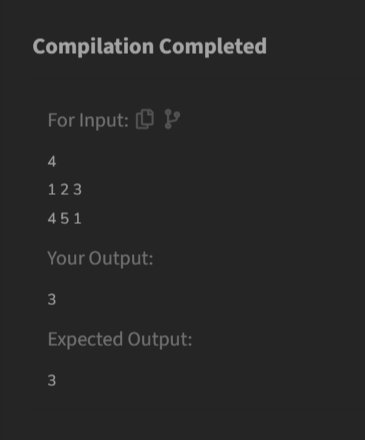
}

return dp[n][capacity];

}

}

**Output**



**Time Complexity**

O(n\*w)

1. **Floor in a Sorted Array**

Difficulty: EasyAccuracy: 33.75%Submissions: 372K+Points: 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]

**Code**

class Solution {

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

if (k < arr[0]) {

return -1;

}

int left = 0;

int right = arr.length - 1;

int result = -1;

while (left <= right) {

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

if (arr[mid] == k) {

return mid;

}

else if (arr[mid] < k) {

result = mid;

left = mid + 1;

}

else {

right = mid - 1;

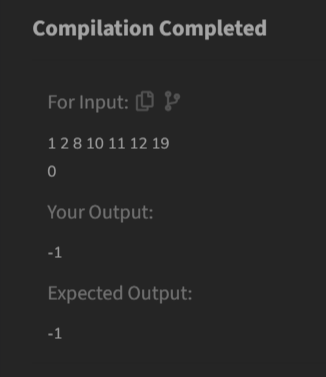
}

}

return result;

}

}

**Output**

**Time Complexity**

O(log n)

1. **Check Equal Arrays**

Difficulty: BasicAccuracy: 42.18%Submissions: 367K+Points: 1

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

**Code**

class Solution {

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

if (arr1.length != arr2.length) {

return false;

}

java.util.HashMap<Integer, Integer> map = new java.util.HashMap<>();

for (int num : arr1) {

map.put(num, map.getOrDefault(num, 0) + 1);

}

for (int num : arr2) {

if (!map.containsKey(num) || map.get(num) == 0) {

return false;

}

map.put(num, map.get(num) - 1);

}

return true;

}

}



**Output**

**Time Complexity**

O(n)

1. **Palindrome Linked List**

Difficulty: MediumAccuracy: 41.48%Submissions: 345K+Points: 4

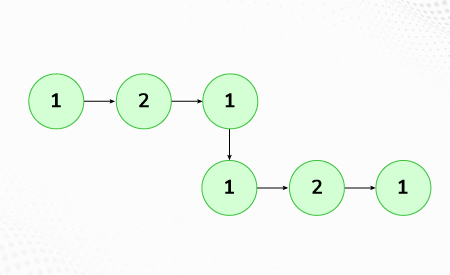
Given a singly linked list of integers. The task is to check if the given linked list is palindrome or not.

Examples:

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

Output: true

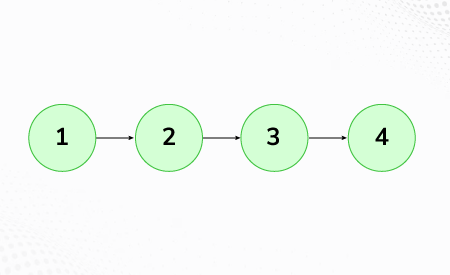
Explanation: The given linked list is 1->2->1->1->2->1 , which is a palindrome and Hence, the output is true.



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 <= 105  
1 ≤ node->data ≤ 103

**Code**

class Solution{

boolean isPalindrome(Node head) {

if (head == null || head.next == null) {

return true;

}

Node slow = head;

Node fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

Node second = reverseList(slow.next);

Node first =head;

while (second!= null) {

if (first.data != second.data) {

return false;

}

first= first.next;

second = second.next;

}

return true;

}

private Node reverseList(Node head) {

Node prev = null;

Node current = head;

Node next = null;

while (current != null) {

next = current.next;

current.next = prev;

prev = current;

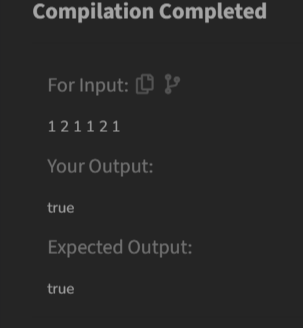
current = next;

}

return prev;

}

}

**Output**

**Time Complexity**

O(n)

1. **Balanced Tree Check**

Difficulty: EasyAccuracy: 43.15%Submissions: 320K+Points: 2

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

**Code**

class Tree {

boolean isBalanced(Node root) {

return checkHeight(root) != -1;

}

private int checkHeight(Node node) {

if (node == null) {

return 0;

}

int left = checkHeight(node.left);

if (left == -1) {

return -1;

}

int right = checkHeight(node.right);

if (right == -1) {

return -1;

}

if (Math.abs(left - right) > 1) {

return -1;

}

return Math.max(left, right) + 1;

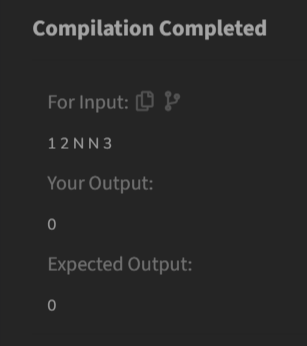
}

}

**Time Complexity**

O(n)

**Output**



1. **Triplet Sum in Array**

Difficulty: MediumAccuracy: 35.0%Submissions: 305K+Points: 4

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

**Code**

class Solution {

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

java.util.Arrays.sort(arr);

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

int left = i + 1;

int right = n - 1;

while (left < right) {

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

if (currentSum == x) {

return true;

}

else if (currentSum < x) {

left++;

}

else {

right--;

}

}

}

return false;

}

}

**Time Complexity**

O(n^2)

**Output**