# Sahana Lakshmipathy\_AI&DS\_Practice\_DAY-2

**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.

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)

**Program:**

class Solution {  
 // Function to return max value that can be put in knapsack of capacity.  
 static int knapSack(int capacity, int val[], int wt[]) {  
 // code here  
 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 class Main {

public static void main(String[] args) {

int capacity = 50; // capacity of knapsack

int[] val = {60, 100, 120}; // values of items

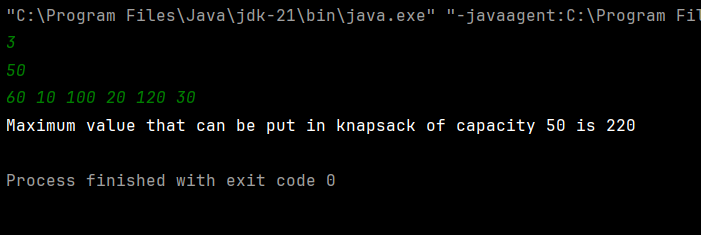
int[] wt = {10, 20, 30}; // weights of items

int maxValue = Solution.knapSack(capacity, val, wt);

System.out.println("Maximum value that can be put in knapsack of capacity " + capacity + " is " + maxValue);

}

}



**Time Complextiy : O(n2)**

**2.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.

**Program:**

class Solution {

// Function to find floor of x

// arr: input array

// n is the size of array

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

int low = 0, high = n - 1;

int res = -1;

while (low <= high) {

int mid = (low + high) / 2;

if (arr[mid] == x)

return mid;

else if (arr[mid] < x) {

res = mid;

low = mid + 1;

} else

high = mid - 1;

}

return res;

}

}  
  
public class Main {

public static void main(String[] args) {

long[] arr = {1, 2, 4, 6, 10, 12, 14}; // sorted array

int n = arr.length;

long x = 7; // element to find the floor of

int floorIndex = Solution.findFloor(arr, n, x);

if (floorIndex != -1) {

System.out.println("The floor of " + x + " is " + arr[floorIndex] + " at index " + floorIndex);

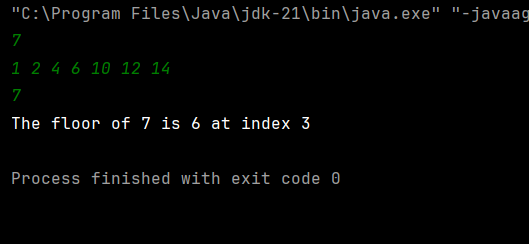
} else {

System.out.println("No floor found for " + x);

}

}

}



**Time Complexity: O(N)**

**3.Check arrays are Equal**

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]

Input: arr1[] = [1, 2, 5], arr2[] = [2, 4, 15]

Output: false

Explanation: arr1[] and arr2[] have only one common value.

**Program:**

class Solution {  
 // Function to check if two arrays are equal or not.  
 public static boolean check(int[] arr1, int[] arr2) {  
 // Your code here  
 if (arr1.length != arr2.length) return false;  
 Map<Integer,Integer> set = new HashMap<>();  
   
 for(int n : arr1){  
 set.put(n, set.getOrDefault(n,0) + 1);  
 }  
   
   
   
 for(int n : arr2){  
 if (!set.containsKey(n)){  
 return false;  
 }  
 set.put(n, set.get(n) -1);  
 }  
   
 for(int count : set.values()){  
 if (count != 0) return false;  
 }  
   
 return true;  
 }  
}

import java.util.HashMap;

import java.util.Map;

public class Main {

public static void main(String[] args) {

int[] arr1 = {1, 2, 3, 4, 5};

int[] arr2 = {5, 4, 3, 2, 1};

boolean areEqual = Solution.check(arr1, arr2);

if (areEqual) {

System.out.println("The arrays are equal.");

} else {

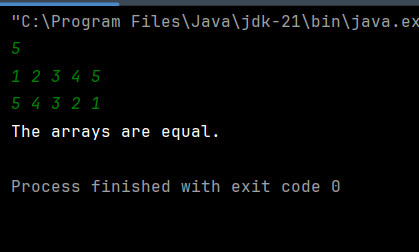
System.out.println("The arrays are not equal.");

}

}

}

**Time Complexity: O(n)**



**4.Balanced Binary Tree or Not**

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.

**Program:**

class Tree {

// Helper method that returns the height of a tree if balanced; -1 if not balanced

private int checkHeight(Node node) {

if (node == null) {

return 0;

}

// Recursively get the height of left and right subtrees

int leftHeight = checkHeight(node.left);

if (leftHeight == -1) return -1; // Not balanced

int rightHeight = checkHeight(node.right);

if (rightHeight == -1) return -1; // Not balanced

// Check if the current node is balanced

if (Math.abs(leftHeight - rightHeight) > 1) {

return -1; // Not balanced

}

// Return height if balanced

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

}

// Function to check whether a binary tree is balanced or not.

boolean isBalanced(Node root) {

return checkHeight(root) != -1;

}

}

class Node {

int data;

Node left, right;

public Node(int data) {

this.data = data;

left = right = null;

}

}

public class Main {

public static void main(String[] args) {

// Creating a balanced binary tree

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

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

root.left.right = new Node(5);

root.right.left = new Node(6);

root.right.right = new Node(7);

Tree tree = new Tree();

// Check if the binary tree is balanced

if (tree.isBalanced(root)) {

System.out.println("The binary tree is balanced.");

} else {

System.out.println("The binary tree is not balanced.");

}

// Creating an unbalanced binary tree

Node unbalancedRoot = new Node(1);

unbalancedRoot.left = new Node(2);

unbalancedRoot.left.left = new Node(3);

if (tree.isBalanced(unbalancedRoot)) {

System.out.println("The unbalanced binary tree is balanced.");

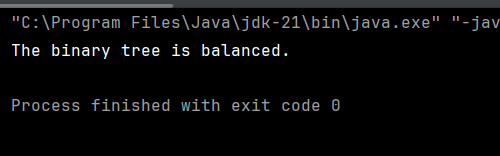
} else {

System.out.println("The unbalanced binary tree is not balanced.");

}

}

}



**Time Complexity: O(n)**

**5.Triplet Sum:**

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.

Input: n = 6, x = 24, arr[] = [40,20,10,3,6,7]

Output: 0

Explanation: There is no triplet with sum 24.

**Program:**

import java.util.Arrays;

class Solution {

// Should return true if there is a triplet with sum equal to x in arr[], otherwise false

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

// Sort the array first

Arrays.sort(arr);

// Iterate over the array

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

// Set up two pointers

int left = i + 1;

int right = n - 1;

// Look for a pair with sum x - arr[i]

while (left < right) {

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

if (currentSum == x) {

return true; // Triplet found

} else if (currentSum < x) {

left++; // Increase sum by moving left pointer to the right

} else {

right--; // Decrease sum by moving right pointer to the left

}

}

}

return false; // No triplet found

}

}

public class Main {

public static void main(String[] args) {

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

int n = arr.length;

int x = 22;

if (Solution.find3Numbers(arr, n, x)) {

System.out.println("Triplet found with sum " + x);

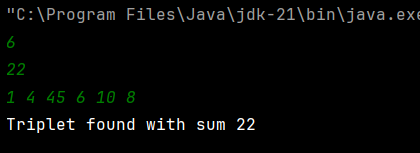
} else {

System.out.println("No triplet found with sum " + x);

}

}

}



**6.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.

**Program:**

class Solution {

// Function to check whether the list is palindrome.

boolean isPalindrome(Node head) {

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

return true;

}

// Step 1: Find the middle of the linked list

Node slow = head, fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

// Step 2: Reverse the second half of the list

Node secondHalfStart = reverseList(slow);

// Step 3: Compare the first and the second half nodes

Node firstHalfStart = head;

Node secondHalfCurrent = secondHalfStart;

boolean isPalindrome = true;

while (secondHalfCurrent != null) {

if (firstHalfStart.data != secondHalfCurrent.data) {

isPalindrome = false;

break;

}

firstHalfStart = firstHalfStart.next;

secondHalfCurrent = secondHalfCurrent.next;

}

// Step 4: Restore the list (optional)

reverseList(secondHalfStart);

return isPalindrome;

}

// Helper function to reverse a linked list

private Node reverseList(Node head) {

Node prev = null;

Node current = head;

while (current != null) {

Node next = current.next;

current.next = prev;

prev = current;

current = next;

}

return prev;

}

}

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

public class Main {

public static void main(String[] args) {

// Creating a palindrome linked list: 1 -> 2 -> 3 -> 2 -> 1

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);

Solution solution = new Solution();

// Check if the linked list is a palindrome

if (solution.isPalindrome(head)) {

System.out.println("The linked list is a palindrome.");

} else {

System.out.println("The linked list is not a palindrome.");

}

// Creating a non-palindrome linked list: 1 -> 2 -> 3 -> 4

Node head2 = new Node(1);

head2.next = new Node(2);

head2.next.next = new Node(3);

head2.next.next.next = new Node(4);

if (solution.isPalindrome(head2)) {

System.out.println("The linked list is a palindrome.");

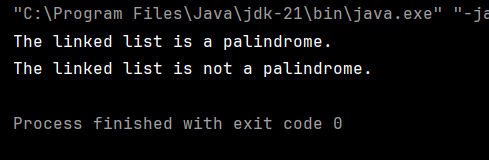
} else {

System.out.println("The linked list is not a palindrome.");

}

}

}



**Time Complexity: O(n)**