**Date: 09.11.2024**

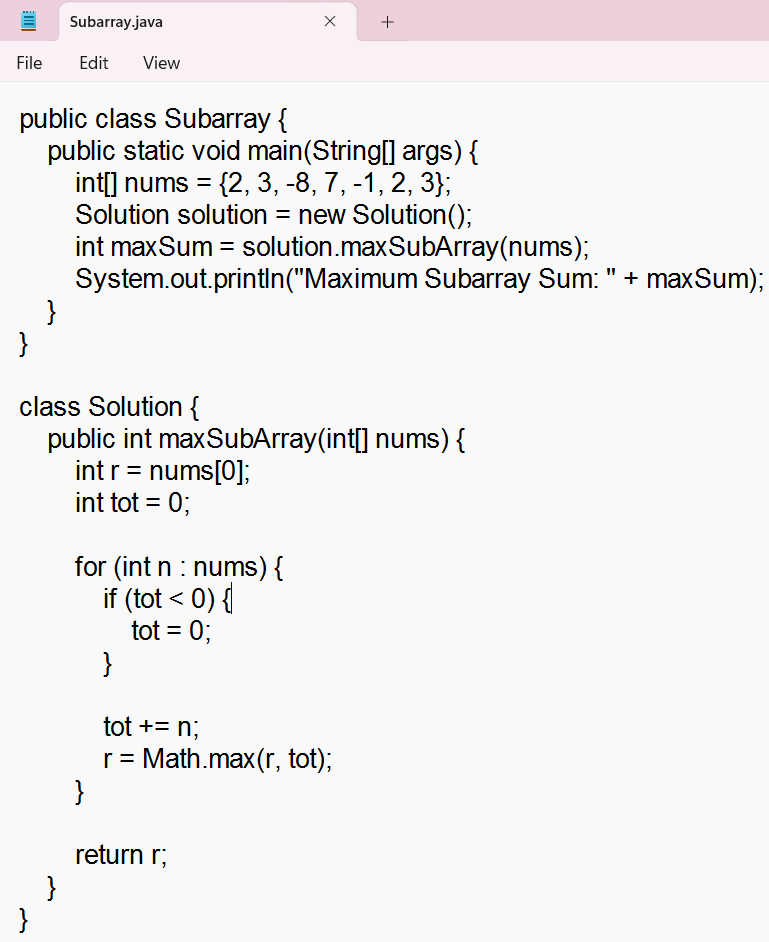
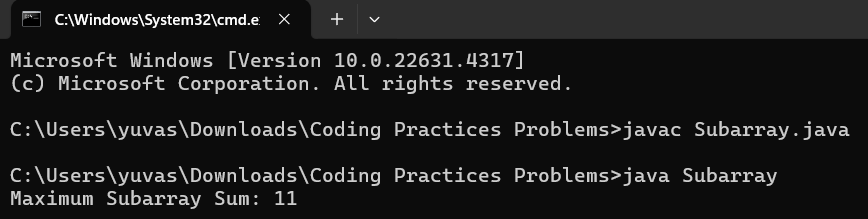
**Sri Vaigunth.M-22AD134**

**Coding Practice Problems**

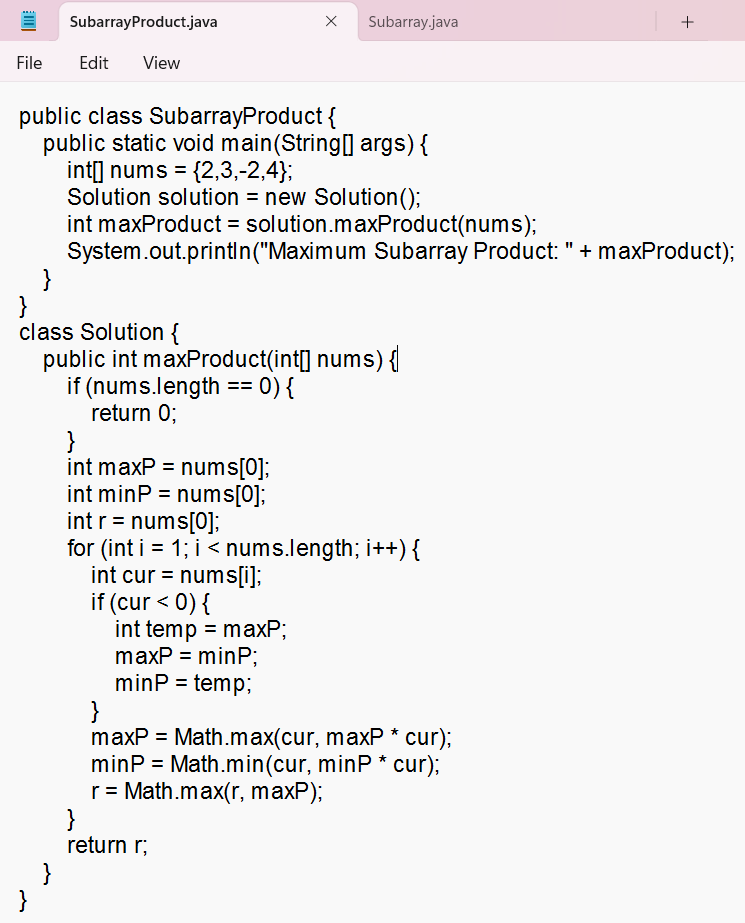
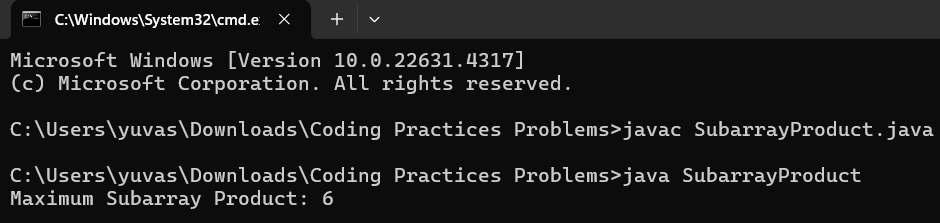
Set - 1

1. Maximum Subarray Sum – Kadane‟s Algorithm:

Time Complexity : o(n)

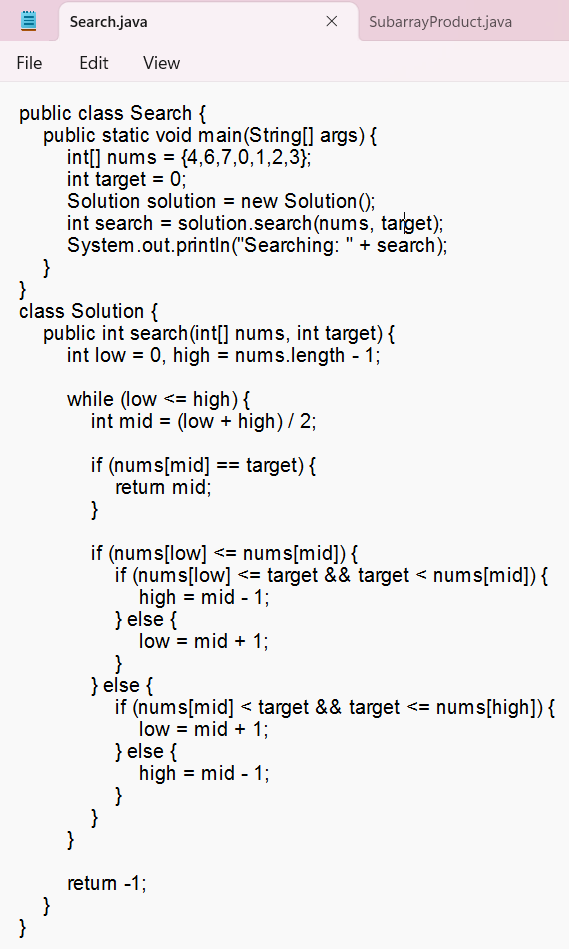
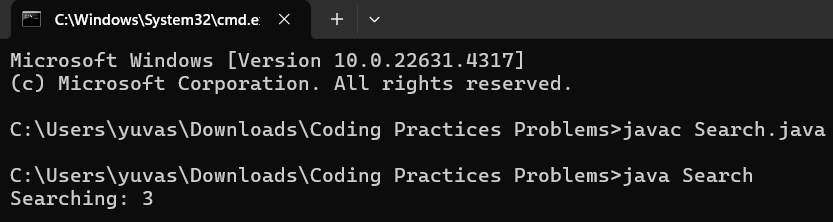


1. Maximum Product Subarray



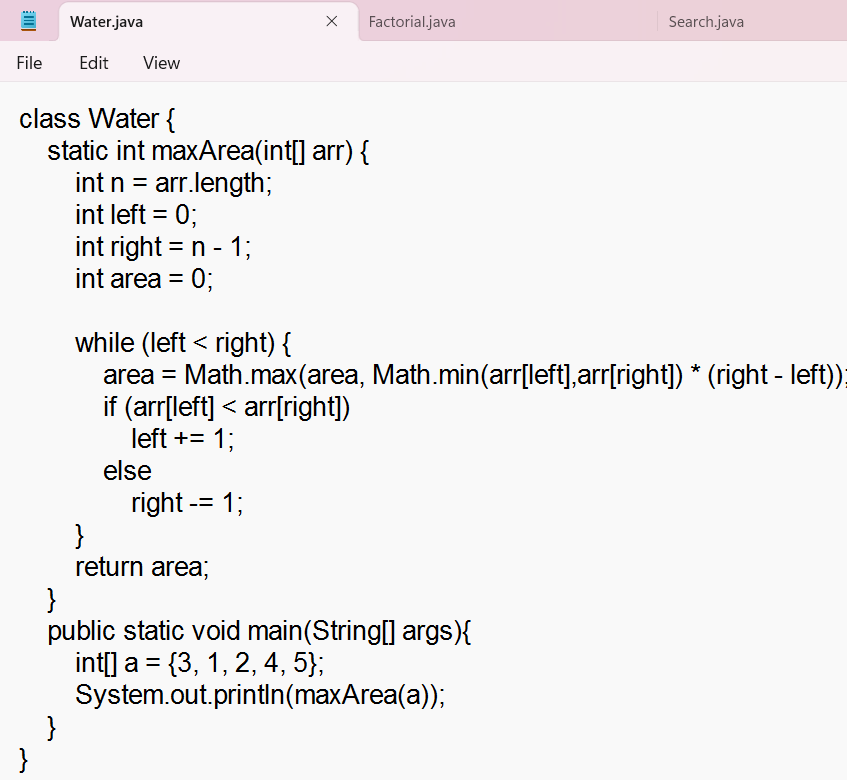
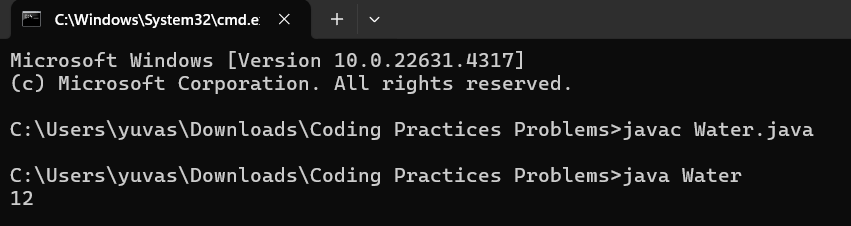
Time Complexity: o(n)

1. Search in a sorted and rotated Array

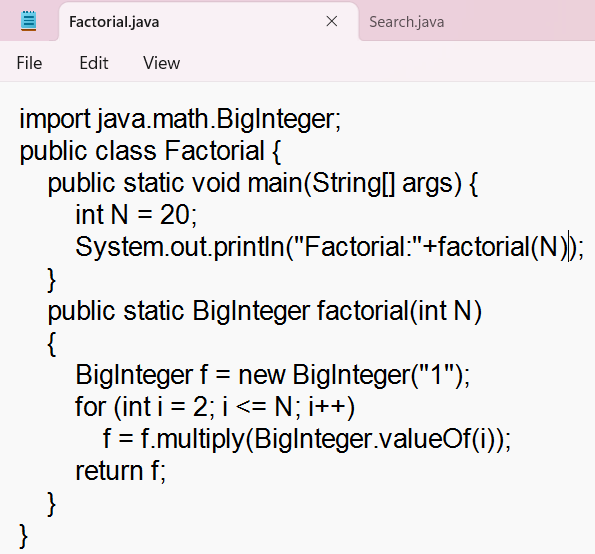
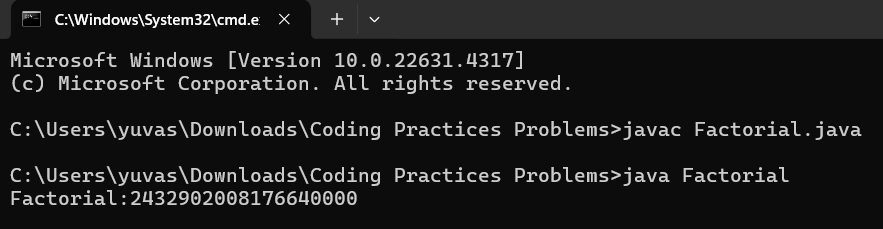


Time Complexity: o(log n)

1. Container with Most Water



1. Find the Factorial of a large number



Time Complexity: o(n)

1. Trapping Rainwater Problem:

**Time Complexity: O(n)**

import java.util.\*;

class TRP {

// Function to find the maximum amount of water that can be trapped

static int maxWater(int[] arr) {

int left = 1;

int right = arr.length - 2;

// lMax : Maximum in subarray arr[0..left-1]

// rMax : Maximum in subarray arr[right+1..n-1]

int lMax = arr[left - 1];

int rMax = arr[right + 1];

int res = 0;

while (left <= right) {

// If rMax is smaller, then we can decide the amount of water for arr[right]

if (rMax <= lMax) {

// Add the water for arr[right]

res += Math.max(0, rMax - arr[right]);

// Update right max

rMax = Math.max(rMax, arr[right]);

// Update right pointer as we have decided the amount of water for this

right -= 1;

} else {

// Add the water for arr[left]

res += Math.max(0, lMax - arr[left]);

// Update left max

lMax = Math.max(lMax, arr[left]);

// Update left pointer as we have decided water for this

left += 1;

}

}

return res;

}

// Driver code

public static void main(String[] args) {

int[] arr = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};

System.out.println(maxWater(arr));

}

}

**Output:**

****

1. Chocolate Distribution Problem:

**Time Complexity : O(nlog )**

import java.util.Arrays;

public class ChocolateDistribution {

public static int findMinDifference(int[] chocolates, int m) {

// Edge case: if there are no chocolates or fewer packets than children

if (chocolates.length == 0 || m == 0) return 0;

if (chocolates.length < m) return -1; // Not enough packets for each child

// Step 1: Sort the array

Arrays.sort(chocolates);

// Step 2: Find the minimum difference by checking each subset of m packets

int minDifference = Integer.MAX\_VALUE;

for (int i = 0; i + m - 1 < chocolates.length; i++) {

int difference = chocolates[i + m - 1] - chocolates[i];

minDifference = Math.min(minDifference, difference);

}

return minDifference;

}

public static void main(String[] args) {

int[] chocolates = {7, 3, 2, 4, 9, 12, 56};

int m = 3; // Number of children

int result = findMinDifference(chocolates, m);

if (result == -1) {

System.out.println("Not enough chocolate packets for each child.");

} else {

System.out.println("Minimum difference is " + result);

}

}

}

Output:



1. Merge Overlapping Intervals

TimeComplexity:O(n)

Code:

package ArrayList;

import java.util.\*;

public class intervels {

public static void main(String[] args) {

int[][] arr= {{7,8}, {1,5}, {2,4}, {4,6}};

HashMap<Integer,Integer> h1=new HashMap<>();

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

h1.put(arr[i][0],arr[i][1]);

}

List<Map.Entry<Integer,Integer>> li=new ArrayList<>(h1.entrySet());

li.sort(Map.Entry.*comparingByKey*());

List<int[]> merged = new ArrayList<>();

int start = li.get(0).getKey();

int end = li.get(0).getValue();

for (int i = 1; i < li.size(); i++) {

int next = li.get(i).getKey();

int last = li.get(i).getValue();

if (end >= next) {

end = Math.*max*(end, last);

} else {

merged.add(new int[]{start, end});

start = next;

end = last;

}

}

merged.add(new int[]{start, end});

int[][] nums=new int[merged.size()][2];

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

nums[i]=merged.get(i);

}

for (int[] interval : merged) {

System.***out***.println("[" + interval[0] + ", " + interval[1] + "]");

}

}

}

Output:



1. A Boolean Matrix Question

Time Complexity: O(n\*m)

Code:

package ArrayList;

public class booleanMatrix {

public static void main(String[] args) {

int[][] mat = {{0, 0, 0},

{0, 0, 1}};

int[][] updat=new int[mat.length][mat[0].length];

int n=mat.length;int m=mat[0].length;

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

for(int j=0;j<mat[i].length;j++) {

if(mat[i][j]==1) {

for(int x=0;x<m;x++) {

updat[i][x]=1;

}

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

updat[x][j]=1;

}

}

}

}

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

for(int j=0;j<mat[i].length;j++) {

System.***out***.print(updat[i][j]+ " ");

}

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

}

}

}

Output:



1. Print a given matrix in spiral form

Time complexity: O(n∗m)

Code:

package ArrayList;

import java.util.\*;

public class spiral {

public static void main(String[] args) {

int[][] matrix= {{1,2,3},{4,5,6},{7,8,9}};

int rows = matrix.length;

int cols = matrix[0].length;

int x = 0;

int y = 0;

int dx = 1;

int dy = 0;

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

for (int i = 0; i < rows \* cols; i++) {

res.add(matrix[y][x]);

matrix[y][x] = -101;

if (!(0 <= x + dx && x + dx < cols && 0 <= y + dy && y + dy < rows) || matrix[y+dy][x+dx] == -101) {

int temp = dx;

dx = -dy;

dy = temp;

}

x += dx;

y += dy;

}

System.***out***.println(res);

}

}

Output:



1. Check if given Parentheses expression is balanced or not

Time Complexity:O(n)

Code:

package ArrayList;

import java.util.\*;

public class parantheses {

public static void main(String[] args) {

String str1 = "((()))()()";

Stack <Character> stack = new Stack<Character>();

for(int i=0;i<str1.length();i++) {

char ch=str1.charAt(i);

if(ch=='(') {

stack.push(ch);

}else if(stack.isEmpty()) {

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

return;

}else {

stack.pop();

}

}

if(stack.isEmpty()) {

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

}else {

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

}

}

}

Output:



1. Check if two Strings are Anagrams of each other

TimeComplexity: O(nlogn)

Code:

package ArrayList;

import java.util.\*;

public class string {

public static void main(String[] args) {

String str1="allergy";

String str2="allergic";

char[] ch1=str1.toCharArray();

Arrays.*sort*(ch1);

str1=new String(ch1);

char[] ch2=str2.toCharArray();

Arrays.*sort*(ch2);

str2=new String(ch2);

System.***out***.println(str1+" "+str2);

if(str1==str2) {

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

return;

}

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

}

}

Output:



1. Longest Palindromic Substring

TimeComplexity:O(N^2)

Code:

package ArrayList;

public class longestCommon {

private static boolean isPalindrome(String str) {

int left = 0;

int right = str.length() - 1;

while (left < right) {

if (str.charAt(left) != str.charAt(right)) {

return false;

}

left++;

right--;

}

return true;

}

public static void main(String[] args) {

String s="forgeeksskeegfor";

if (s.length() <= 1) {

System.***out***.println(s);

return;

}

int maxLen = 1;

String maxStr = s.substring(0, 1);

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

for (int j = i + maxLen; j <= s.length(); j++) {

if (j - i > maxLen && *isPalindrome*(s.substring(i, j))) {

maxLen = j - i;

maxStr = s.substring(i, j);

}

}

}

System.***out***.println(maxStr);

}

}

Output:



1. Longest Common Prefix using Sorting

TimeComplexity:o(nlogn)

Code:

package ArrayList;

import java.util.\*;

public class commonPrefix {

public static void main(String[] args) {

String[] str = {"geeksforgeeks", "geeks", "geek", "geezer"};

Arrays.*sort*(str);

System.***out***.println(Arrays.*toString*(str));

String s1 = str[0];

String s2 = str[str.length-1];

int i = 0;

while(i< s1.length() && i < s2.length()){

if(s1.charAt(i) == s2.charAt(i)){

i++;

} else {

break;

}

}

System.***out***.println(s1.substring(0,i));

}

}

Output:



1. Delete middle element of a stack

Time Complexity:O(n)

Code:

package ArrayList;

import java.util.\*;

public class stack1 {

public static void main(String[] args) {

Stack<Integer> st=new Stack<>();

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

if(arr.length<=1) {

System.***out***.println(st);

return;

}

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

st.push(arr[i]);

}

Stack<Integer> s2=new Stack<>();

for(int i=0;i<=st.size()/2;i++) {

s2.push(st.pop());

}

st.pop();

int len=s2.size();

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

st.push(s2.pop());

}

System.***out***.println(st);

}

}

Output:



1. Next Greater Element (NGE) for every element in given Array

Time Complexity: O(N)

package util;

import java.util.Stack;

import java.util.HashMap;

public class greaterele {

public static void printNextGreaterElements(int[] arr) {

Stack<Integer> stack = new Stack<>();

HashMap<Integer, Integer> ngeMap = new HashMap<>();

for (int i = arr.length - 1; i >= 0; i--) {

int currentElement = arr[i];

while (!stack.isEmpty() && stack.peek() <= currentElement) {

stack.pop();

}

int nextGreater = stack.isEmpty() ? -1 : stack.peek();

ngeMap.put(currentElement, nextGreater);

stack.push(currentElement);

}

for (int element : arr) {

System.***out***.println(element + " -> " + ngeMap.get(element));

}

}

public static void main(String[] args) {

int[] arr = {4, 5, 2, 25};

*printNextGreaterElements*(arr);

}

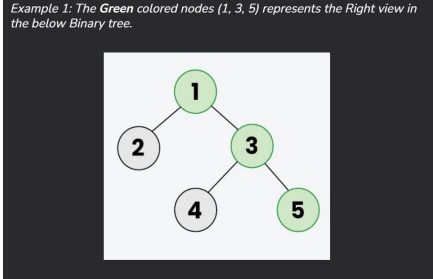
}

Output:



1. Print Right View of a Binary Tree

**Time Complexity**: O(n)



Code:

import java.util.ArrayList;

import java.util.List;

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) { this.val = val; }

}

class Solution {

int maxlevel = 0;

public List<Integer> rightSideView(TreeNode root) {

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

right(root, 1, list);

return list;

}

void right(TreeNode root, int level, List<Integer> list) {

if (root == null) {

return;

}

if (maxlevel < level) {

list.add(root.val);

maxlevel = level;

}

right(root.right, level + 1, list);

right(root.left, level + 1, list);

}

public static void main(String[] args) {

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

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

root.right.right = new TreeNode(4);

Solution solution = new Solution();

List<Integer> result = solution.rightSideView(root);

System.out.println(result);

}

}

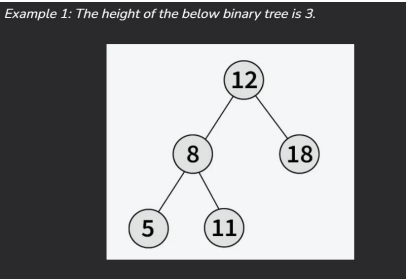
Output:



18. Maximum Depth or Height of Binary Tree Given a binary tree, the task is to find the maximum depth or height of the tree. The height of the tree is the number of vertices in the tree from the root to the deepest node

Time Complexity:O(n)

TestCase:



Code:

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) { this.val = val; }

}

class Solution {

public int maxDepth(TreeNode root) {

if (root == null) return 0;

int left = maxDepth(root.left);

int right = maxDepth(root.right);

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

}

public static void main(String[] args) {

TreeNode root = new TreeNode(3);

root.left = new TreeNode(9);

root.right = new TreeNode(20);

root.right.left = new TreeNode(15);

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

Solution solution = new Solution();

int result = solution.maxDepth(root);

System.out.println(result);

}

}

Output:

