**Practice Set-1**

**1)Maximum Subarray Sum – Kadane‟s Algorithm:**

Given an array arr[], the task is to find the subarray that has the maximum sum and return its sum. Input: arr[] = {2, 3, -8, 7, -1, 2, 3} Output: 11 Explanation: The subarray {7, -1, 2, 3} has the largest sum 11.

Code:

import java.util.\*;

public class Sum{

public static void main(String[] args){

Sum obj=new Sum();

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

int[] arr=new int[n];

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

arr[i]=sc.nextInt();

}

int res=obj.maxsum(arr);

System.out.println(res);

}

public int maxsum(int[] arr){

int currsum=0;

int maxsum=Integer.MIN\_VALUE;

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

currsum+=arr[i];

maxsum=Math.max(maxsum,currsum);

if(currsum<0){

currsum=0;

}

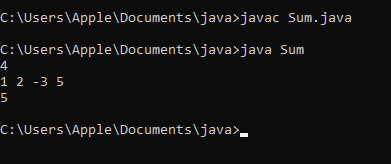
}

return currsum;

}

}

Output:



**Time Complexity**:O(n) **Spacecomplexity**:O(1)

**2. Maximum Product Subarray**

Given an integer array, the task is to find the maximum product of any subarray. Input: arr[] = {-2, 6, -3, -10, 0, 2} Output: 180 Explanation: The subarray with maximum product is {6, -3, -10} with product = 6 \* (-3) \* (-10) = 180 Input: arr[] = {-1, -3, -10, 0, 60} Output: 60

Program:

import java.util.\*;

public class Productsum {

public static void main(String[] args){

Productsum obj=new Productsum();

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

int[] arr=new int[n];

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

arr[i]=sc.nextInt();

}

int res=obj.product(arr);

System.out.println(res);

}

public int product(int[] arr){

int min=arr[0];

int max=arr[0];

int result=arr[0];

for(int array:arr){

if(array<0){

int temp=max;

max=min;

min=temp;

}

max=Math.max(array,max\*array);

min=Math.min(array,min\*array);

result=Math.max(result,max);

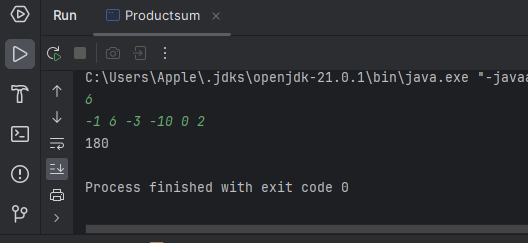
}

return result;

}

}

Output:



**Time Complexity**:O(n) **Space complexit**y:O(1)

**3)Search in a sorted and rotated Array**

Given a sorted and rotated array arr[] of n distinct elements, the task is to find the index of given key in the array. If the key is not present in the array, return -1. Input : arr[] = {4, 5, 6, 7, 0, 1, 2}, key = 0 Output : 4

Input : arr[] = { 4, 5, 6, 7, 0, 1, 2 }, key = 3 Output : -1

Program:

import java.util.Scanner;

public class SortandRotatedarray {

public static void main (String[] args){

SortandRotatedarray obj=new SortandRotatedarray();

Scanner sc=new Scanner(System.in);

System.out.println("Enter no of input");

int n=sc.nextInt();

System.out.println("Enter key");

int key=sc.nextInt();

int[] arr=new int[n];

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

arr[i]=sc.nextInt();

}

int res=obj.FindKey(arr, key);

System.out.println(res);

}

public int FindKey(int[] arr,int key){

int low=0;

int high=arr.length-1;

while(low<=high){

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

if(arr[mid]==key){

return mid;

}

if(arr[low]<=arr[mid]){

if(key>=arr[low] && key<arr[mid]){

high=mid-1;

else{

low=mid+1;

}

}

else{

if(key<=arr[high] && key>arr[mid]){

low=mid+1;

}

else{

high=mid-1;

}

}

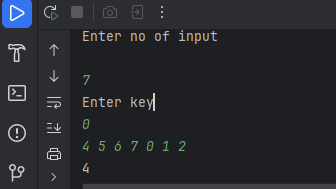
}

return -1;

}

}

Output:



**Time complexity**:O(log n) **Space complexity**=O(1)

**4) container with most water**

Given n non-negative integers a1,a2,…,an*a*1​,*a*2​,…,*an*​ where each represents a point at coordinate (i,ai)(*i*,*ai*​). ‘ n ‘ vertical lines are drawn such that the two endpoints of line i is at (i,ai) (*i*,*ai*​) and (i,0)(*i*,0). Find two lines, which together with x-axis forms a container, such that the container containthemostwater.  
The program should return an integer which corresponds to the maximum area of water that can be contained (maximum area instead of maximum volume sounds weird but this is the 2D plane we are working with for simplicity).

Program:

import java.util.Scanner;

public class ContainerWithMostWater {

public static void main(String[] args){

ContainerWithMostWater obj=new ContainerWithMostWater();

Scanner sc=new Scanner(System.in);

System.out.println("Enter no of input");

int n=sc.nextInt();

int[] arr=new int[n];

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

arr[i]=sc.nextInt();

}

int res=obj.mostWater(arr);

System.out.println(res);

}

public int mostWater(int[] arr){

int left=0;

int right=arr.length-1;

int max=0;

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

int height=Math.min(arr[left],arr[right]);

int width=right-left;

int area=height\*width;

max=Math.max(max,area);

if(arr[left]<arr[right]){

left++;

}

else{

right--;

}

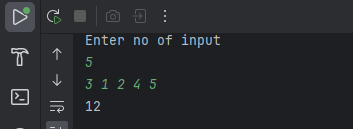
}

return max;

}

}

Output:



**Time Complexity** :O(n) **Space Complexity**: O(1)

**5. Find the Factorial of a large number**

Input: 100

Output: 933262154439441526816992388562667004907159682643816214685929638952175999932299 156089414639761565182862536979208272237582511852109168640000000000000000000000 00

Program:

import java.math.BigInteger;

import java.util.Scanner;

public class BigFactorial {

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

System.out.println("enter num");

int num=sc.nextInt();

BigFactorial obj=new BigFactorial();

BigInteger res=obj.fact(num);

System.out.println(res);

}

public BigInteger fact(int num){

if(num==0){

return BigInteger.ONE;

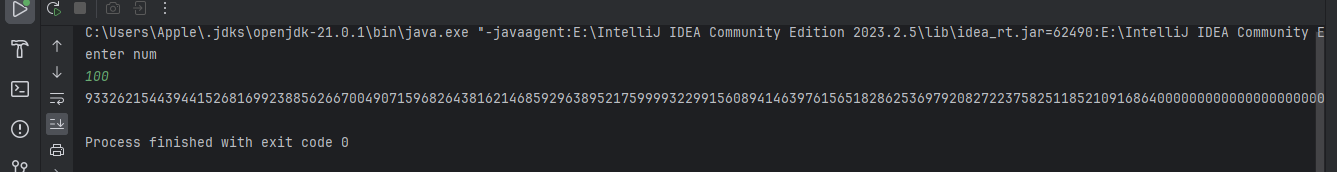
}

return BigInteger.valueOf(num).multiply(fact(num-1));

}

}

Output:



**TimeComplexity** :O(n\*logn) **Space Complexity**:O(n)

**6. Trapping Rainwater Problem**

states that given an array of n non-negative integers arr[] representing an elevation map where the width of each bar is 1, compute how much water it can trap after rain.

Input: arr[] = {3, 0, 1, 0, 4, 0, 2} Output: 10

Explanation: The expected rainwater to be trapped is shown in the above image.

Program:

import java.util.Scanner;

public class TrappingRainWater {

public static void main(String[] args){

TrappingRainWater obj=new TrappingRainWater();

Scanner sc=new Scanner(System.in);

System.out.println("Enter no of input");

int n=sc.nextInt();

int[] arr=new int[n];

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

arr[i]=sc.nextInt();

}

int res=obj.rainWater(arr);

System.out.println(res);

}

public int rainWater(int[] arr){

int l=arr.length;

int[] left=new int[l];

int[] right=new int[l];

left[0]=arr[0];

for(int i=1;i<l;i++){

left[i]=Math.max(left[i-1],arr[i]);

}

right[l-1]=arr[l-1];

for(int i=l-2;i>=0;i--){

right[i]=Math.max(right[i+1],arr[i]);

}

int sum=0;

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

sum+=Math.min(left[i],right[i]-arr[i]);

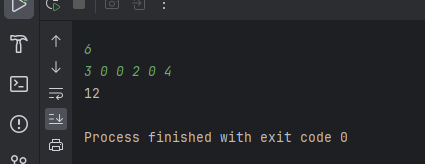
}

return sum;

}

}

Output:



**TimeComplexity** :O(n) **Space Complexity**:O(n)

**7. Chocolate Distribution Problem**

Given an array arr[] of n integers where arr[i] represents the number of chocolates in ith packet. Each packet can have a variable number of chocolates. There are m students, the task is to distribute chocolate packets such that:

Each student gets exactly one packet. The difference between the maximum and minimum number of chocolates in the packets given to the students is minimized. Input: arr[] = {7, 3, 2, 4, 9, 12, 56}, m = 3 Output: 2 Explanation: If we distribute chocolate packets {3, 2, 4}, we will get the minimum difference, that is 2.

Program:

import java.util.Arrays;

import java.util.Scanner;

public class DistributeChocolate {

public static void main(String[] args){

DistributeChocolate obj=new DistributeChocolate();

Scanner sc=new Scanner(System.in);

System.out.println("Enter no of input");

int n=sc.nextInt();

System.out.println("student");

int key=sc.nextInt()

int[] arr=new int[n];

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

arr[i]=sc.nextInt();

}

int res=obj.chocolate(arr, key);

System.out.println(res);

}

public int chocolate(int[] arr,int m){

if(m>arr.length){

return 0;

}

Arrays.sort(arr);

int min=Integer.MAX\_VALUE;

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

int diff=arr[i+m-1]-arr[i];

min=Math.min(min,diff);

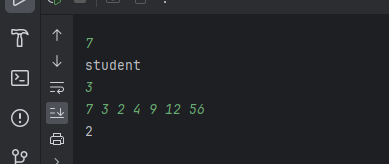
}

return min;

}

}

Output



**TimeComplexity** :O(n log n) **Space Complexity**:O(1)

**8. Merge Overlapping Intervals**

Given an array of time intervals where arr[i] = [starti, endi], the task is to merge all the overlapping intervals into one and output the result which should have only mutually exclusive intervals.

Input: arr[] = [[1, 3], [2, 4], [6, 8], [9, 10]]

Output: [[1, 4], [6, 8], [9, 10]]

Explanation: In the given intervals, we have only two overlapping intervals [1, 3] and [2, 4]. Therefore, we will merge these two and return [[1, 4}], [6, 8], [9, 10]].

Program:

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.Scanner;

public class MergeIntervals {

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of intervals: ");

int n = sc.nextInt();

int[][] arr=new int[n][2];

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

arr[i][0]=sc.nextInt();

arr[i][1]=sc.nextInt();

}

MergeIntervals obj=new MergeIntervals();

int[][] res= obj.intervals(arr);

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

System.out.println(Arrays.toString(res[i]));

}

}

public int[][] intervals(int[][] arr){

Arrays.sort(arr,(a,b)->Integer.compare(a[0] ,b[0]));

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

for(int[] interval:arr){

if(list.isEmpty() || list.get(list.size()-1)[1]<interval[0]){

list.add(interval);

}

else{

list.get(list.size()-1)[1]=Math.max(list.get(list.size()-1)[1],interval[1]);

}

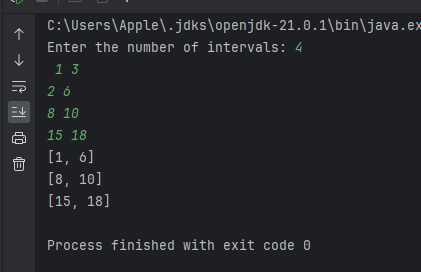
}

return list.toArray(new int[list.size()][]);

}

}

Output:



**TimeComplexity** :O(n log n) **Space Complexity**:O(n)

**9. A Boolean Matrix Question**

Given a boolean matrix mat[M][N] of size M X N, modify it such that if a matrix cell mat[i][j] is 1 (or true) then make all the cells of ith row and jth column as 1.

Input: {{1, 0}, {0, 0}}

Output: {{1, 1} {1, 0}}

Input: {{0, 0, 0}, {0, 0, 1}}

Output: {{0, 0, 1}, {1, 1, 1}}

Program:

import java.util.Scanner;

public class MatrixZero {

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

System.out.println("Enter the length of the matrix");

int n=sc.nextInt();

int m=sc.nextInt();

int[][] matrix=new int[n][m];

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

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

matrix[i][j]=sc.nextInt();

}

}

MatrixZero obj=new MatrixZero();

int[][] res = obj.zero(matrix);

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

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

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

}

System.out.println();

}

}

public int[][] zero(int[][] arr){

int m=arr.length;

int n=arr[0].length;

boolean[] row=new boolean[m];

boolean[] col=new boolean[n];

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

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

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

row[i]=true;

col[j]=true;

}

}

}

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

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

if(row[i] || col[j]) {

arr[i][j] = 1;

}

}

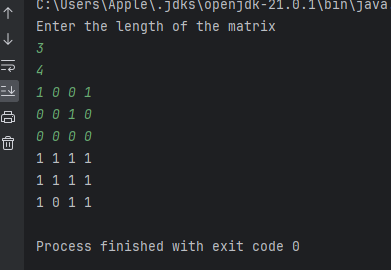
}

return arr;

}

}

Output:



**TimeComplexity** :O(n\*m) **Space Complexity**:O(n+m)

**10. Print a given matrix in spiral form**

Given an m x n matrix, the task is to print all elements of the matrix in spiral form.

Input: matrix = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16 }}

Output: 1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10 Input: matrix = { {1, 2, 3, 4, 5, 6}, {7, 8, 9, 10, 11, 12}, {13, 14, 15, 16, 17, 18}}

Program:

import java.util.Scanner;

public class SpiralMatrix {

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

System.out.println("Enter the length of the matrix");

int n=sc.nextInt();

int m=sc.nextInt();

int[][] matrix=new int[n][m];

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

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

matrix[i][j]=sc.nextInt();

}

}

SpiralMatrix obj=new SpiralMatrix();

obj.matrix(matrix);

}

public void matrix(int[][] matrix){

int left=0;

int right=matrix[0].length-1;

int top=0;

int bottom=matrix.length-1;

while(top<=bottom && left<=right){

for(int i=left;i<=right;i++){

System.out.print(matrix[top][i]+" ");

}

for(int i=top;i<=bottom;i++){

System.out.print(matrix[i][right]+" ");

}

right--;

if(top<=bottom){

for(int i=right;i>=left;i--){

System.out.print(matrix[bottom][i]+" ");

}

bottom--;

}

if(left<=right){

for (int i=bottom;i>=top;i--){

System.out.print(matrix[i][left]+" ");

}

left++;

}

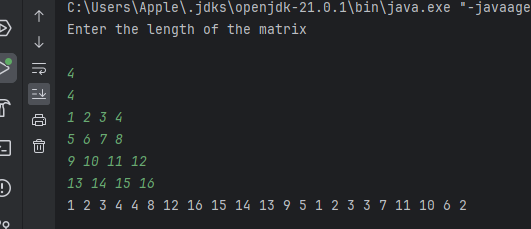
}

System.out.println();

}

}

Output;



**TimeComplexity** :O(n\*m) **Space Complexity**:O(1)

**13. Check if given Parentheses expression is balanced or not**

Given a string str of length N, consisting of „(„ and „)„ only, the task is to check whether it is balanced or not

Input: str = “((()))()()”

Output: Balanced

Input: str = “())((())”

Output: Not Balanced

Program:

import java.util.Scanner;

public class BalancedParanthesis {

public static void main(String[] args){

Scanner sc =new Scanner(System.in);

String s=sc.nextLine();

BalancedParanthesis obj=new BalancedParanthesis();

obj.balanced(s);

}

public void balanced(String str){

int count=0;

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

char c=str.charAt(i);

if(c=='('){

count++;

}

else if(c==')'){

count--;

}

}

if(count<0){

System.out.println("Not Balanced");

}

else{

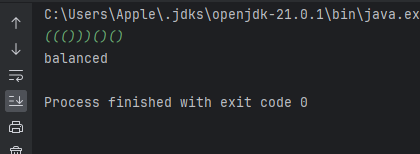
System.out.println("balanced");

}

}

}

Output:



**TimeComplexity** :O(n) **Space Complexity**:O(1)

**14. Check if two Strings are Anagrams of each other**

Given two strings s1 and s2 consisting of lowercase characters, the task is to check whether the two given strings are anagrams of each other or not. An anagram of a string is another string that contains the same characters, only the order of characters can be different.

Input: s1 = “geeks” s2 = “kseeg”

Output: true

Explanation: Both the string have same characters with same frequency. So, they are anagrams.

Program:

import java.util.Arrays;

import java.util.Scanner;

public class Anagram {

public static void main(String[] args){

Scanner sc =new Scanner(System.in);

String s=sc.nextLine();

Anagram obj=new Anagram();

String s2=sc.nextLine();

obj.equal(s,s2);

}

public void equal(String str1,String str2){

if(str1.length()!= str2.length()){

System.out.println("false");

}

char[] c=str1.toCharArray();

char[] c2=str2.toCharArray();

Arrays.sort(c);

Arrays.sort(c2);

if(Arrays.equals(c,c2)){

System.out.println("true");

}

else {

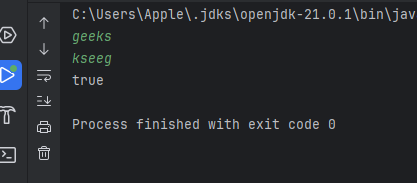
System.out.println("false");

}

}

}

Output:



**TimeComplexity** :O(n\*log n) **Space Complexity**:O(n)

**15. Longest Palindromic Substring**

Given a string str, the task is to find the longest substring which is a palindrome. If there are multiple answers, then return the first appearing substring.

Input: str = “forgeeksskeegfor”

Output: “geeksskeeg”

Explanation: There are several possible palindromic substrings like “kssk”, “ss”, “eeksskee” etc. But the substring “geeksskeeg” is the longest among all.

Program:

import java.util.Scanner;

public class longestPalindromeSubString {

public static void main(String[] args){

Scanner sc =new Scanner(System.in);

String s=sc.nextLine();

int start=0;

int end=0;

longestPalindromeSubString obj=new longestPalindromeSubString();

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

int l1=obj.subString(s,i,i);

int l2=obj.subString(s,i,i+1);

int max=Math.max(l1,l2);

if(max>(end-start)){

start=i-(max-1)/2;

end=i+max/2;

}

}

System.out.println(s.substring(start,end+1));

}

public int subString(String str,int start,int end){

while(start>=0 && end <str.length() && str.charAt(start)==str.charAt(end)){

start--;

end++;

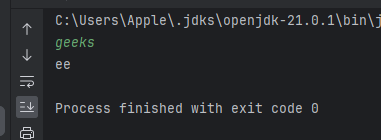
}

return end-start-1;

}

}

Output:



**TimeComplexity** :O(n\*2) **Space Complexity**:O(1)

**16. Longest Common Prefix using Sorting**

Given an array of strings arr[]. The task is to return the longest common prefix among each and every strings present in the array. If there‟s no prefix common in all the strings, return “-1”.

Input: arr[] = [“geeksforgeeks”, “geeks”, “geek”, “geezer”]

Output: gee

Explanation: “gee” is the longest common prefix in all the given strings

Program:

import java.util.Arrays;

import java.util.Scanner;

public class LongestCommonPrefix {

public static void main(String[] args){

Scanner sc =new Scanner(System.in);

int n=sc.nextInt();

sc.nextLine();

String[] str=new String[n];

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

str[i]=sc.nextLine();

}

Arrays.sort(str);

String str1=str[0];

String str2=str[n-1];

int index=0;

while(index<n){

if(str1.charAt(index)==str2.charAt(index)){

index++;

}

else{

break;

}

}

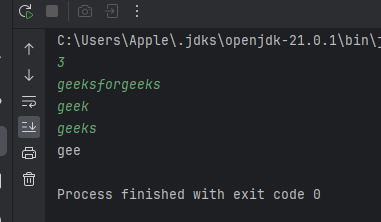
String res=index==0?"":str1.substring(0,index);

System.out.println(res);

}

}

Output:



**TimeComplexity** :O(n\*log n+m) **Space Complexity**:O(1)

**17. Delete middle element of a stack**

Given a stack with push(), pop(), and empty() operations, The task is to delete the middle element of it without using any additional data structure.

Input : Stack[] = [1, 2, 3, 4, 5] Output : Stack[] = [1, 2, 4, 5] Input : Stack[] = [1, 2, 3, 4, 5, 6] Output : Stack[] = [1, 2, 4, 5, 6]

Program: import java.util.Scanner;

import java.util.Stack;

public class PoptheMiddleElement {

public static void main(String[] args){

Scanner sc =new Scanner(System.in);

int n=sc.nextInt();

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

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

stack.push(sc.nextInt());

}

PoptheMiddleElement obj=new PoptheMiddleElement();

int mid=n/2;

obj.delete(stack,mid);

System.out.println(stack);

}

public void delete(Stack<Integer> stack, int mid){

if(mid==0){

stack.pop();

return;

}

int top=stack.pop();

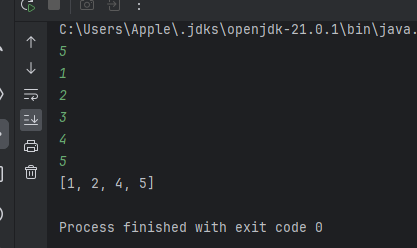
delete(stack,mid-1);

stack.push(top);

}

}

Output:



**TimeComplexity** :O(n) **Space Complexity**:O(n)

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

Given an array, print the Next Greater Element (NGE) for every element. Note: The Next greater Element for an element x is the first greater element on the right side of x in the array. Elements for which no greater element exist, consider the next greater element as -1.

Input: arr[] = [ 4 , 5 , 2 , 25 ]

Output: 4 –> 5 5 –> 25 2 –> 25 25 –> -1 Explanation: Except 25 every element has an element greater than them present on the right side Input: arr[] = [ 13 , 7, 6 , 12 ] Output: 13 –> -1 7 –> 12 6 –> 12 12 –> -1

Program:

import java.util.Scanner;

import java.util.Stack;

public class NextGreaterElement {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int[] arr = new int[n];

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

arr[i] = sc.nextInt();

}

findNGE(arr, n);

}

public static void findNGE(int[] arr, int n) {

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

for (int i = n - 1; i >= 0; i--) {

while (!stack.isEmpty() && stack.peek() <= arr[i]) {

stack.pop();

}

if (stack.isEmpty()) {

System.out.println(arr[i] + " -> -1");

} else {

System.out.println(arr[i] + " -> " + stack.peek());

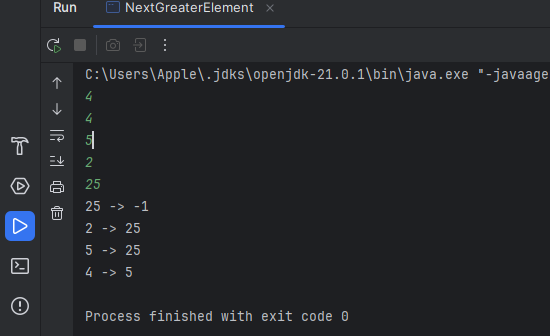
}

stack.push(arr[i]);

}

}

}



Output:

**TimeComplexity** :O(n) **Space Complexity**:O(n)

**19. Print Right View of a Binary Tree**

Given a Binary Tree, the task is to print the Right view of it. The right view of a Binary Tree is a set of rightmost nodes for every level

Program:

import java.util.\*;

class TreeNode {

int data;

TreeNode left, right;

TreeNode(int data) {

this.data = data;

left = right = null;

}

}

public class BinaryTree { // Make this class public

TreeNode root;

public void rightView(TreeNode root) {

if (root == null) return;

Queue<TreeNode> queue = new LinkedList<>();

queue.add(root);

while (!queue.isEmpty()) {

int size = queue.size();

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

TreeNode current = queue.poll();

if (i == size) {

System.out.print(current.data + " ");

}

if (current.left != null) queue.add(current.left);

if (current.right != null) queue.add(current.right);

}

}

}

public static void main(String[] args) {

BinaryTree tree = new BinaryTree();

tree.root = new TreeNode(1);

tree.root.left = new TreeNode(2);

tree.root.right = new TreeNode(3);

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

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

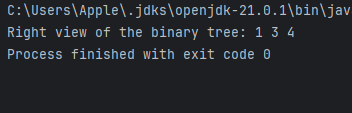
System.out.print("Right view of the binary tree: ");

tree.rightView(tree.root);

}

}

Output:



**TimeComplexity** :O(n) **Space Complexity**:O(n)

**20. 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

Program:

import java.util.\*;

class Tree {

int data;

Tree left, right;

Tree(int data) {

this.data = data;

left = right = null;

}

}

public class BinaryTree2 {

Tree root;

public int maxDepth(Tree node) {

if (node == null) {

return 0;

}

int leftDepth = maxDepth(node.left);

int rightDepth = maxDepth(node.right);

return 1 + Math.max(leftDepth, rightDepth);

}

public static void main(String[] args) {

BinaryTree2 tree = new BinaryTree2();

tree.root = new Tree(1);

tree.root.left = new Tree(2);

tree.root.right = new Tree(3);

tree.root.left.left = new Tree(4);

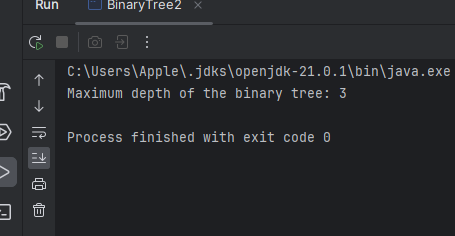
tree.root.left.right = new Tree(5);

System.out.println("Maximum depth of the binary tree: " + tree.maxDepth(tree.root));

}

}

Output:



**TimeComplexity** :O(n) **Space Complexity**:O(n)