DATE:09-11-2024

CODING PROBLEMS 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:**

package util;

import java.util.\*;

public class subarray {

public static void main(String[] args) {

int[] arr1={-2,-4};

System.out.println(calculate(arr1));

int[] arr2 = {2, 3, -8, 7, -1, 2, 3};

System.out.println(calculate(arr2));

int[] arr3 = {5, 4, 1, 7, 8};

System.out.println(calculate(arr3));

}

private static int calculate(int[] arr) {

int n = arr.length;

int res = arr[0];

int end = arr[0];

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

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

res = Math.max(res,end);

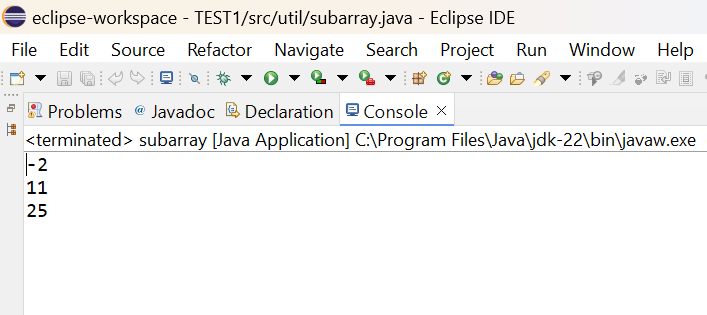
}

return res;

}

}

**OUTPUT:**



**TIME COMPLEXITY: O(n)**

1. **MAXIMUM PRODUCT SUBARRAY:**

**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**

**CODE:**

package util;

public class product {

public static void main(String[] args) {

int[] arr = {-2, 6, -3, -10, 0, 2};

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

int[] arr1 = {-1, -3, -10, 0, 60};

System.out.println(calculate(arr1));

}

public static int calculate(int[] arr) {

if(arr == null) return 0;

int min = arr[0];

int max = arr[0];

int res = arr[0];

for(int i=1;i<arr.length;i++) {

int tempmax = max;

max = Math.max(Math.max(max\*arr[i],min\*arr[i]),arr[i]);

min = Math.min(Math.min(tempmax\*arr[i], min\*arr[i]),arr[i]);

res = Math.max(max,res);

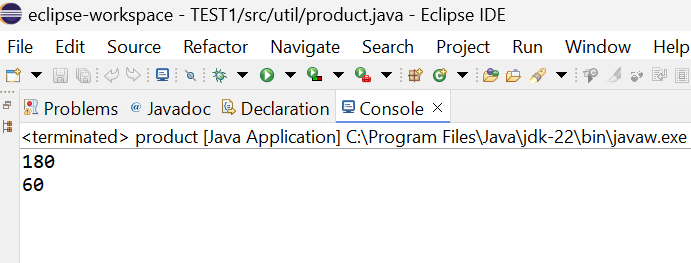
}

return res;

}

}

**OUTPUT:**



**TIME COMPLEXITY:O(n)**

1. **SEARCH IN A SORTED AND ROTATED ARRAY**

**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**

**Input : arr[] = {50, 10, 20, 30, 40}, key = 10 Output : 1**

**CODE:**

package util;

public class searcharray {

public static void main(String[] args) {

int[] nums = {4, 5, 6, 7, 0, 1, 2};

int key = 0;

System.out.println(search(nums,key));

}

public static int search(int[] nums, int target) {

int n = nums.length;

int start = 0;

int end = n-1;

while(start<end){

int mid =(start+end)/2;

if(nums[0]<=nums[mid]){

if(nums[0]<=target && target<=nums[mid]){

end = mid;

}else{

start =mid+1;

}

}

else{

if(nums[mid]<target && target<=nums[n-1]){

start = mid+1;

}else{

end = mid;

}

}

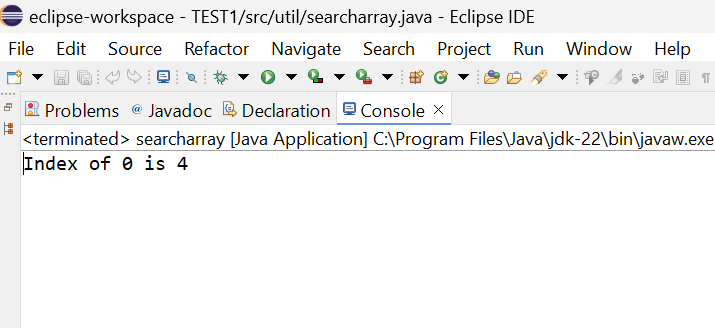
}

return nums[start]==target ? start:-1;

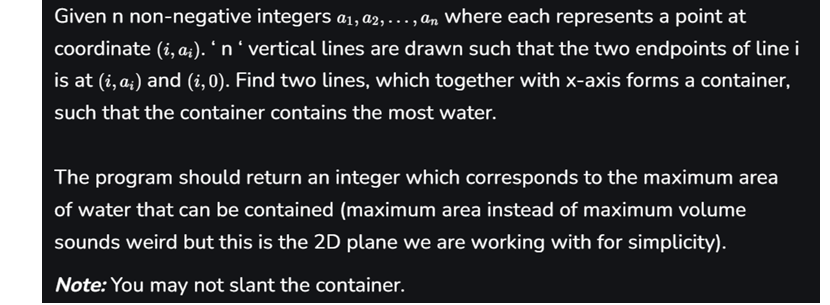
}

}

**OUTPUT:**



**TIME COMPLEXITY:**O(log n)

1. **CONTAINER WITH MOST WATER**

**Input: arr = [1, 5, 4, 3] Output: 6 Explanation: 5 and 3 are distance 2 apart. So the size of the base = 2. Height of container = min(5, 3) = 3. So total area = 3 \* 2 = 6**

**CODE:**

package util;

import java.util.Scanner;

public class Container {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int[] height = new int[n];

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

height[i] = sc.nextInt();

}

System.out.println(maxArea(height));

}

public static int maxArea(int[] height) {

int max = 0;

int left = 0;

int right = height.length-1;

while(left<right){

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

int width = right-left;

int current = heights\*width;

max = Math.max(max,current);

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

left++;

}else{

right--;

}

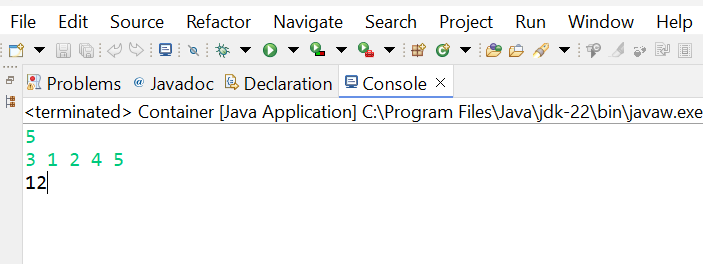
}

return max;

}

}

**OUTPUT:**



**TIME COMPLEXITY:**O(n)

1. **FIND THE FACTORIAL OF A LARGE NUMBER**

**Input: 100**

**Output: 933262154439441526816992388562667004907159682643816214685929638952175999932299 156089414639761565182862536979208272237582511852109168640000000000000000000000 00**

**Input: 50**

**Output: 30414093201713378043612608166064768844377641568960512000000000000**

**CODE:**

package util;

import java.math.BigInteger;

public class Factorial {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

System.out.println(fact(n));

}

public static BigInteger fact(int n) {

BigInteger f = new BigInteger("1");

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

f=f.multiply(BigInteger.valueOf(i));

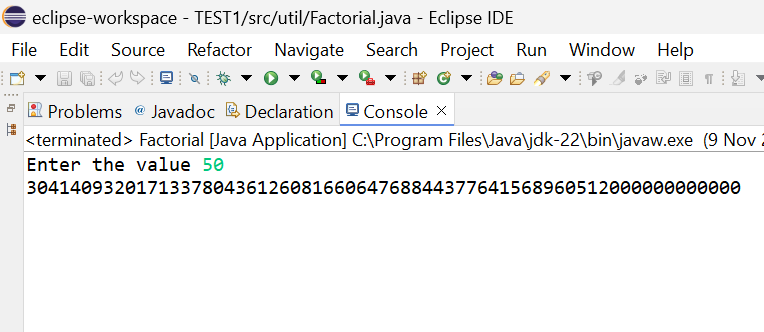
}

return f;

}

}

**OUTPUT:**



**TIME COMPLEXITY:** O(n)

**6.TRAPPPING RAIN WATER PROBLEM**

**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.**

**CODE:**

package util;

import java.util.Scanner;

public class trapping {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int[] height = new int[n];

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

height[i] = sc.nextInt();

}

System.out.println(trap(height));

}

public static int trap(int[] height) {

int n = height.length;

if(n==0) return 0;

int[] leftmax = new int[n];

int[] rightmax = new int[n];

leftmax[0] = height[0];

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

leftmax[i] = Math.max(leftmax[i-1],height[i]);

}

rightmax[n-1]=height[n-1];

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

rightmax[i]=Math.max(rightmax[i+1],height[i]);

}

int water = 0;

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

water+=Math.min(leftmax[i],rightmax[i])-height[i];

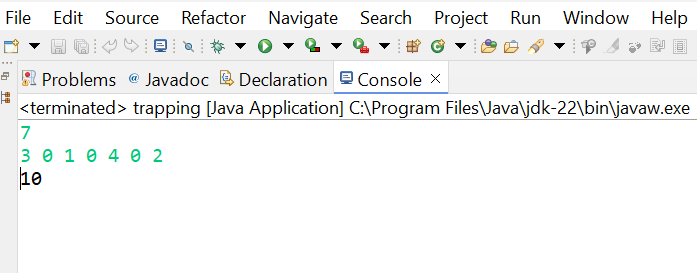
}

return water;

}

}

**OUTPUT:**



**TIME COMPLEXITY:** O(n)

1. **CHOCOLATE DISTRIBUTION PROBLEM**

**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.**

**CODE:**

package util;

import java.util.Arrays;

import java.util.Scanner;

public class Chocolate {

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();

}

int m = sc.nextInt();

System.out.println(difference(arr,m));

}

public static int difference(int[] arr,int m) {

Arrays.sort(arr);

int min = Integer.MAX\_VALUE;

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

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

if(diff<min) {

min = diff;

}

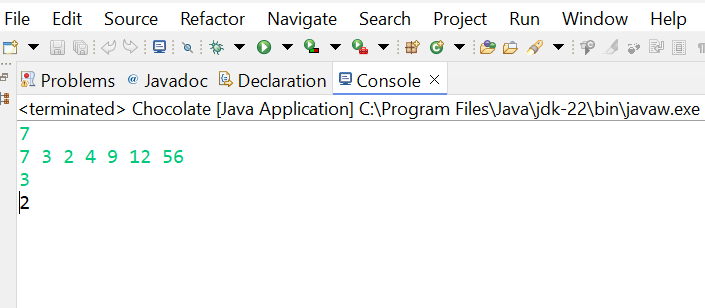
}

return min;

}

}

**OUTPUT:**



**TIME COMPLEXITY :** O(n log n)

1. **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]].**

**CODE:**

package util;

import java.util.\*;

public class Mergeinterval {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

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

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

{

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

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

}

int[][] result = Mergeinterval.merge(intervals);

for (int[] interval : result) {

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

}

}

public static int[][] merge(int[][] intervals) {

Arrays.sort(intervals, (a, b) -> a[0] - b[0]);

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

mergedintervals.add(intervals[0]);

for (int i = 1; i < intervals.length; ++i) {

int start = intervals[i][0];

int end = intervals[i][1];

int[] lastmerged = mergedintervals.get(mergedintervals.size() - 1);

if (lastmerged[1] < start) {

mergedintervals.add(intervals[i]);

} else {

lastmerged[1] = Math.max(lastmerged[1], end);

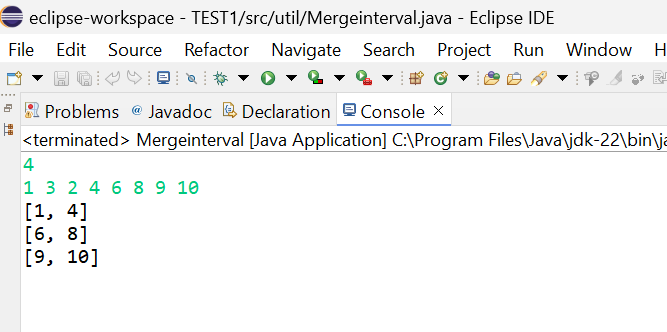
}

}

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

}}

**OUTPUT:**



**TIME COMPLEXITY:O( n log n)**

1. **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}}**

**CODE:**

package util;

import java.util.Scanner;

public class matrix1 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of rows:");

int r = sc.nextInt();

System.out.println("Enter the number of columns:");

int c = sc.nextInt();

int[][] mat = new int[r][c];

System.out.println("Enter the elements of the matrix (row by row):");

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

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

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

}

}

matrix(mat, r, c);

System.out.println("Matrix after modification:");

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

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

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

}

System.out.println();

}

}

public static void matrix(int mat[][],int r,int c) {

int row[] = new int[r];

int col[] = new int[c];

int i,j;

for(i=0;i<r;i++)

row[i]=0;

for(i=0;i<c;i++)

col[i]=0;

for(i=0;i<r;i++) {

for(j=0;j<c;j++) {

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

row[i]=1;

col[i]=1;

}

}

}

for(i=0;i<r;i++) {

for(j=0;j<c;j++) {

if(row[i]==1||col[j]==1) {

mat[i][j]=1;

}

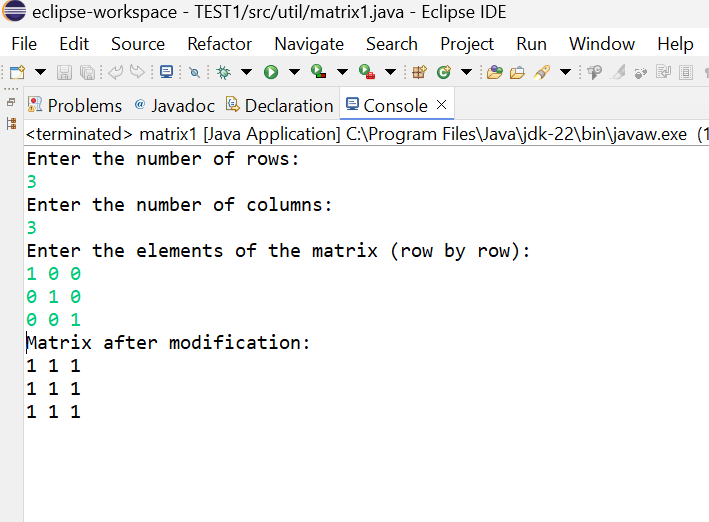
}

}

}

}

**OUTPUT:**



**TIME COMPLEXITY:O(N)**

**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**

**CODE:**

package util;

import java.util.Scanner;

public class spiralmatrix {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of rows:");

int m = sc.nextInt();

System.out.println("Enter the number of columns:");

int n = sc.nextInt();

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

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

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

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

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

}

}

System.out.println("Matrix in spiral order:");

spiralPrint(m, n, a);

}

public static void spiralPrint(int m, int n, int[][] a)

{

int top = 0, bottom = m - 1, left = 0,

right = n - 1;

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

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

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

}

top++;

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

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

}

right--;

if (top <= bottom) {

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

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

}

bottom--;

}

if (left <= right) {

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

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

}

left++;

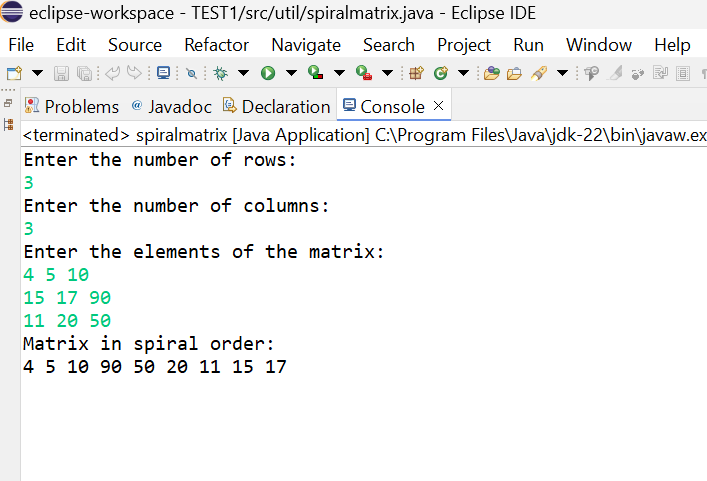
}

}

}

}

**OUTPUT:**



**TIME COMPLEXITY:**O(N)

11**.CHECK IF GIVEN PARENTHESES EXPRESSION IS BALANCED OR NOT**

**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**

**CODE:**

package util;

import java.util.Scanner;

import java.util.Stack;

public class Balancedparantheses {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

String str = sc.nextLine();

if (isBalanced(str)) {

System.out.println("Balanced");

} else {

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

}

}

public static boolean isBalanced(String str) {

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

for (char c : str.toCharArray()) {

if (c == '(') {

stack.push(c);

} else if (c == ')') {

if (stack.isEmpty()) {

return false;

}

stack.pop();

}

}

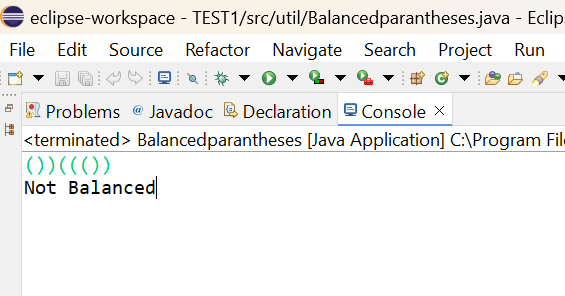
return stack.isEmpty();

}

}

}

**OUTPUT:**



**TIME COMPEXITY:**O(N)

**12. 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.**

**Input: s1 = “g”, s2 = “g”**

**Output: true**

**Explanation: Characters in both the strings are same, so they are anagrams**.

**CODE:**

package util;

import java.util.\*;

public class Anagram {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the first string:");

String s1 = sc.nextLine();

System.out.println("Enter the second string:");

String s2 = sc.nextLine();

System.out.println((areAnagrams(s1, s2)));

}

public static boolean areAnagrams(String s1, String s2) {

if (s1.length() != s2.length()) {

return false;

}

char[] str1 = s1.toCharArray();

char[] str2 = s2.toCharArray();

Arrays.sort(str1);

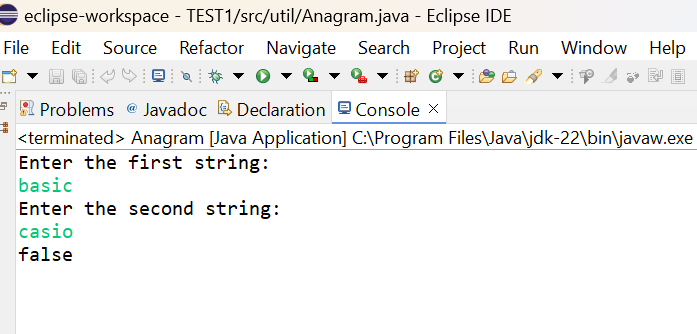
Arrays.sort(str2);

return Arrays.equals(str1, str2);

}

}

**OUTPUT:**



**TIME COMPLEXITY**:O(n log n)

**13. 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.**

**CODE:**

package util;

import java.util.Scanner;

public class Palindrome {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the string:");

String s = sc.nextLine();

Palindrome obj = new Palindrome();

String longestPalindromicSubstring = obj.longestPalindrome(s);

System.out.println("Longest Palindromic Substring: " + longestPalindromicSubstring);

}

public String longestPalindrome(String s) {

int start = 0;

int end =0;

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

int len1 = palindrome(s,i,i);

int len2 = palindrome(s,i,i+1);

int len = Math.max(len1,len2);

if(len>end-start){

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

end = i+len/2;

}

}

return s.substring(start,end+1);

}

private int palindrome(String s,int left,int right){

while(left>=0 && right<s.length() && s.charAt(left)==s.charAt(right)){

left--;

right++;

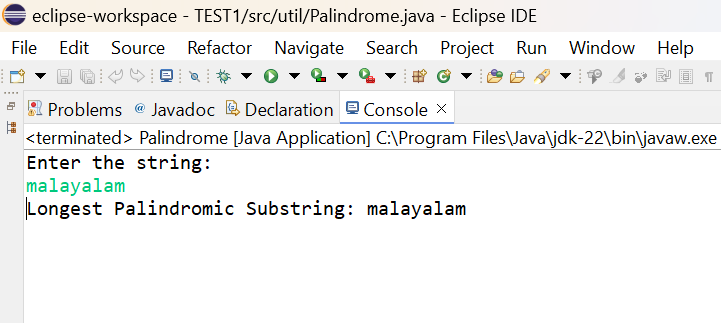
}

return right-left-1;

}

}

**OUTPUT:**



**TIME COMPLEXITY**:O(n^2)

**14. 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.**

**CODE:**

package util;

import java.util.\*;

public class Commonprefix {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of strings:");

int n = sc.nextInt();

sc.nextLine();

List<String> strs = new ArrayList<>();

System.out.println("Enter the strings:");

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

strs.add(sc.nextLine());

}

Commonprefix obj = new Commonprefix();

String result = obj.longestCommonPrefix(strs);

System.out.println("Longest Common Prefix: " + result);

}

public String longestCommonPrefix(List<String> strs) {

if (strs == null || strs.isEmpty()) {

return "-1";

}

String ans = strs.get(0);

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

while (!strs.get(i).startsWith(ans)) {

ans = ans.substring(0, ans.length() - 1);

if (ans.isEmpty()) {

return "-1";

}

}

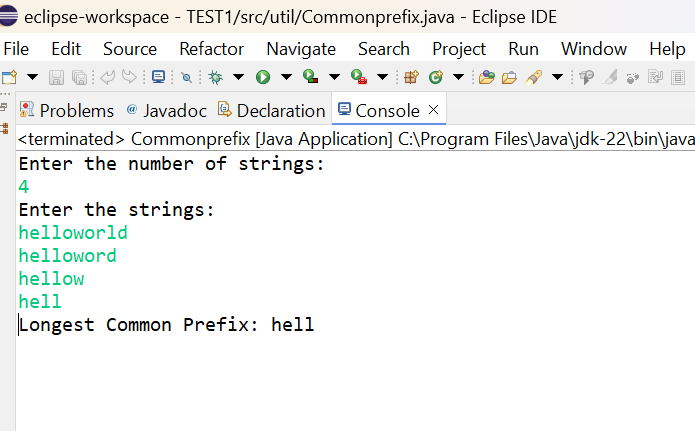
}

return ans;

}

}

**OUTPUT:**



**TIME COMPLEXITY:** O(N)

15**. 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]**

**CODE:**

package util;

import java.util.\*;

public class DeleteMiddleElement {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the number of elements in the stack:");

int n = sc.nextInt();

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

System.out.println("Enter the elements of the stack:");

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

stack.push(sc.nextInt());

}

System.out.println("Original Stack: " + stack);

if (!stack.isEmpty()) {

deleteMiddle(stack, n / 2);

}

System.out.println("Stack after deleting middle element: " + stack);

}

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

if (mid == 0) {

stack.pop();

return;

}

int temp = stack.pop();

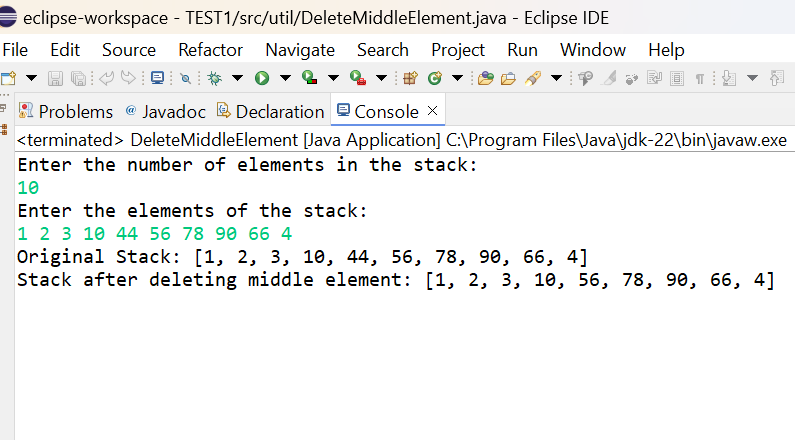
deleteMiddle(stack, mid - 1);

stack.push(temp);

}

}

**OUTPUT:**



**TIME COMPLEXITY:**O(N)

16**. 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**

**CODE:**

package util;

import java.util.\*;

public class nge {

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();

}

int[] res = next(arr);

for(int i:res) {

System.out.print(i+" ");

}

}

private static int[] next(int[] arr) {

int n = arr.length;

int[] element = new int[n];

Arrays.fill(element,-1);

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

for(int j=i+1;j<n;j++) {

if(arr[j]>arr[i]) {

element[i]=arr[j];

break;

}

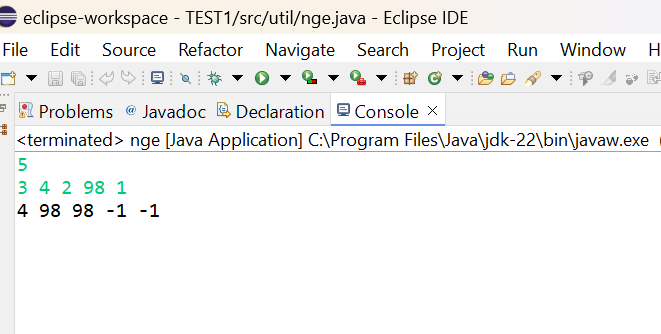
}

}

return element;

}}

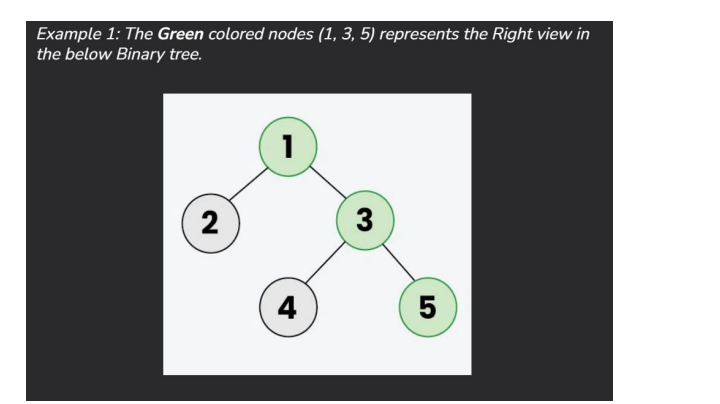
**OUTPUT:**



**TIME COMPLEXITY:** O(N^2)

**17. 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*.***

**

CODE:

package util;

public class Tree1 {

class Node {

int data;

Node left;

Node right;

Node(int val) {

data = val;

left = null;

right = null;

}

}

public Node create(int data) {

return new Node(data);

}

static int max\_level;

public void printRightView(Node root, int level) {

if (root != null) {

if (level > max\_level) {

max\_level = level;

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

}

printRightView(root.right, level + 1);

printRightView(root.left, level + 1);

}

}

public static void main(String[] args) {

Tree1 tree = new Tree1();

Node root = tree.create(2);

root.left = tree.create(3);

root.right = tree.create(7);

root.left.left = tree.create(5);

root.left.right = tree.create(4);

root.left.left.right = tree.create(6);

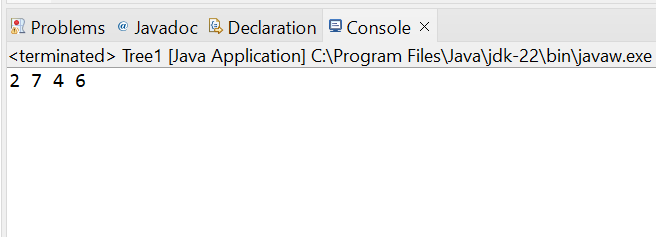
max\_level = 0;

tree.printRightView(root, 1);

}

}

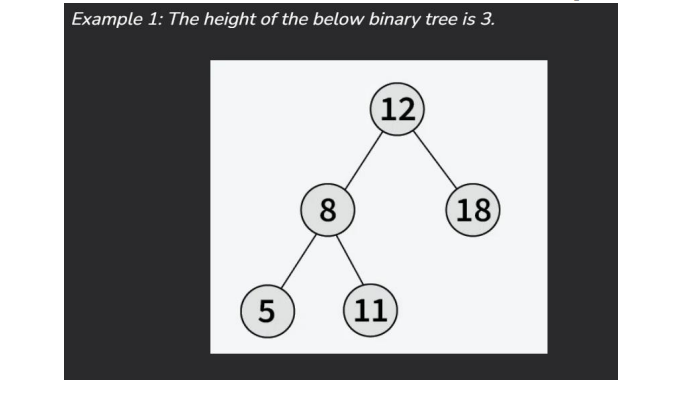
**OUTPUT:**



**TIME COMPLEXITY**:O(N)

**18.MAXIMUM DEPTH OR HEIGHT OF A 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**



**CODE:**

package util;

public class Heightoftree {

public static void main(String[] args) {

Heightoftree tree = new Heightoftree();

Node root = tree.new Node(1);

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

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

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

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

System.out.println("Height of tree is: " + maxDepth(root));

}

class Node {

int data;

Node left, right;

Node(int val) {

data = val;

left = null;

right = null;

}

}

static int maxDepth(Node node) {

if (node == null)

return 0;

int l = maxDepth(node.left);

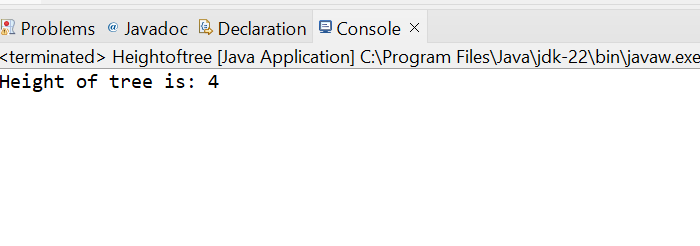
int r = maxDepth(node.right);

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

}

}

**OUTPUT:**



**TIME COMPLEXITY:** O(N)