9th Nov 2024

**DSA Practice Test – 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.

Input: arr[] = {-2, -4}

Output: –2

Explanation: The subarray {-2} has the largest sum -2.

Input: arr[] = {5, 4, 1, 7, 8}

Output: 25

Explanation: The subarray {5, 4, 1, 7, 8} has the largest sum 25.

**Code:**

public class SubarraySum {

    public int findSum(int[] arr){

        int max\_sum=arr[0];

        int res=arr[0];

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

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

            res=Math.max(res,max\_sum);

        }

    return res;

    }

    public static void main(String ar[]){

        SubarraySum ob=new SubarraySum();

        int[] arr={-2,-4};

        int res=ob.findSum(arr);

        System.out.println(res);

    }

}

**Output**:



**Time Complexity:** O (n)

**Space Complexity:** 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

Explanation: The subarray with maximum product is {60}.

**Code:**

**Brute Force Approach:**

public class SubarrayProduct {

int findMax(int a,int b, int c){

int m=Math.max(a,Math.max(b,c));

return m;

}

int findMin(int a,int b, int c){

int m=Math.min(a,Math.min(b,c));

return m;

}

int findProd(int[] arr){

int max\_prod=arr[0];

int x=arr[0];

int y=arr[0];

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

int t=findMax(arr[i],x\*arr[i],y\*arr[i]);

y=findMin(arr[i],x\*arr[i],y\*arr[i]);

x=t;

max\_prod=Math.max(max\_prod,x);

}

return max\_prod;

}

public static void main(String ar[]){

SubarrayProduct ob=new SubarrayProduct();

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

int res=ob.findProd(arr);

System.out.println(res);

}

}

**Optimized Code:**

public class SubarrayProduct {

int findProd(int[] arr) {

int max\_prod = arr[0];

int max\_current = arr[0];

int min\_current = arr[0];

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

if (arr[i] < 0) {

int temp = max\_current;

max\_current = min\_current;

min\_current = temp;

}

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

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

max\_prod = Math.max(max\_prod, max\_current);

}

return max\_prod;

}

public static void main(String[] args) {

SubarrayProduct ob = new SubarrayProduct();

int[] arr = {2,-2,1};

int result = ob.findProd(arr);

System.out.println(result);

}

}

**Output:**



**Time Complexity:** O (n)

**Space Complexity:** 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

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

Output : 1

**Code:**

public class Rotatedarray {

int findind(int[] arr, int key){

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

if(key==arr[i])

return i;

}

return -1;

}

public static void main(String ar[]){

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

int key=3;

Rotatedarray ob=new Rotatedarray();

int res=ob.findind(arr,key);

System.out.println(res);

}

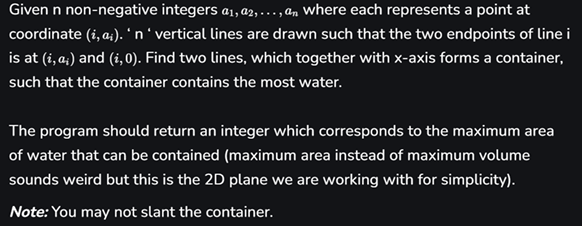
}

Output:



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

**4. Container with Most Water**



**Code:**

public class AreaofC {

public int maxArea(int[] height) {

int a=0;

int b=height.length-1;

int area=0;

while(a<b){

int curr\_a=Math.min(height[a],height[b])\*(b-a);

area=Math.max(area,curr\_a);

if(height[a]<height[b]){

a++;

}

else{

b--;

}

}

return area;

}

public static void main(String ar[]){

AreaofC ob=new AreaofC();

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

int res=ob.maxArea(height);

System.out.print(res);

}

}

**Output:**



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

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

Input: 100

Output: 933262154439441526816992388562667004907159682643816214685929638952175999932299

156089414639761565182862536979208272237582511852109168640000000000000000000000

00

Input: 50

Output: 30414093201713378043612608166064768844377641568960512000000000000

**Code:**import java.math.BigInteger;

public class Fac {

static BigInteger fac(int n)

{ if(n==0||n==1)

return BigInteger.ONE;

else

return BigInteger.valueOf(n).multiply(fac(n-1));

}

public static void main(String ar[])

{

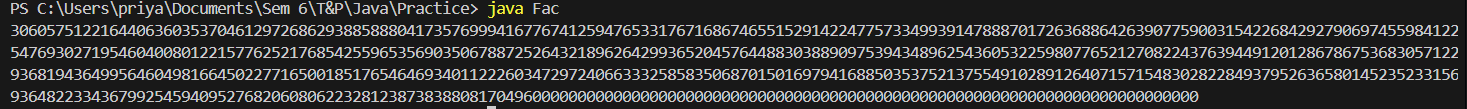
int n=300;

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

}

}

**Output:**

****

**Time Complexity:** O (n)

**Space Complexity:** O (1)

**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.

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

Output: 7

Explanation: We trap 0 + 3 + 1 + 3 + 0 = 7 units.

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

Output: 0

Explanation : We cannot trap water as there is no height bound on both sides

Input: arr[] = {10, 9, 0, 5}

Output: 5

Explanation : We trap 0 + 0 + 5 + 0 = 5

**Code:**

public class TrappingRainWater {

static int trap(int[] height) {

if (height == null || height.length == 0) {

return 0;

}

int left = 0;

int right = height.length - 1;

int left\_max = 0;

int right\_max = 0;

int waterTrapped = 0;

while (left < right) {

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

if (height[left] >= left\_max) {

left\_max = height[left];

} else {

waterTrapped += left\_max - height[left];

}

left++;

} else {

if (height[right] >= right\_max) {

right\_max = height[right];

} else {

waterTrapped += right\_max - height[right];

}

right--;

}

}

return waterTrapped;

}

public static void main(String[] args) {

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

System.out.println("Water trapped: " + trap(height));

}

}

**Output:**

****

**Time Complexity:** O (n log 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.

Input: arr[] = {7, 3, 2, 4, 9, 12, 56}, m = 5

Output: 7

Explanation: If we distribute chocolate packets {3, 2, 4, 9, 7}, we will get the minimum

difference, that is 9 – 2 = 7.

**Code:**

import java.util.Arrays;

public class Chocolates {

static int minDiff(int[] arr,int m){

if(m==0||arr.length==0||arr.length<m){

return -1;

}

int mi=Integer.MAX\_VALUE;

Arrays.sort(arr);

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

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

mi=Math.min(mi,a);

}

return mi;

}

public static void main(String ar[]){

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

int m=3;

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

}

}

**Output:**



**Time Complexity:** O (n)

**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]].

Input: arr[] = [[7, 8], [1, 5], [2, 4], [4, 6]]

Output: [[1, 6], [7, 8]]

Explanation: We will merge the overlapping intervals [[1, 5], [2, 4], [4, 6]] into a single interval

[1, 6].

**Code:**

import java.util.\*;

class MergeIntervals {

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

int n = intervals.length;

if (n == 1) return intervals;

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

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

int l = intervals[0][0];

int r = intervals[0][1];

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

if (r < intervals[i][0]) {

ans.add(new int[]{l, r});

l = intervals[i][0];

r = intervals[i][1];

} else if (r >= intervals[i][0] && r <= intervals[i][1]) {

r = intervals[i][1];

}

}

ans.add(new int[]{l, r});

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

}

public static void main(String ar[]){

int[][] a={

{1,3},

{2,6,},

{8,10},

{15,18}

};

int[][] ans=merge(a);

for(int[] i:ans){

System.out.print(Arrays.toString(i));

}

}

}

**Output:**



**Time Complexity:** O (n)

**Space Complexity:** O (1)

**9. A Boolean Matrix**

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}}

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

{0, 0, 1, 0},

{0, 0, 0, 0}}

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

{1, 1, 1, 1},

{1, 0, 1, 1}}

**Code:**

import java.util.HashSet;

import java.util.Scanner;

import java.util.Set;

class Bool {

public void modifyMatrix(int[][] mat, int M, int N) {

Set<Integer> rows = new HashSet<>();

Set<Integer> cols = new HashSet<>();

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

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

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

rows.add(i);

cols.add(j);

}

}

}

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

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

if (rows.contains(i) || cols.contains(j)) {

mat[i][j] = 1;

}

}

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int M = scanner.nextInt();

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

int N = scanner.nextInt();

int[][] mat = new int[M][N];

System.out.println("Enter the elements of the matrix (0 or 1) row by row:");

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

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

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

}

}

Bool solution = new Bool();

solution.modifyMatrix(mat, M, N);

System.out.println("Modified matrix:");

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

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

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

}

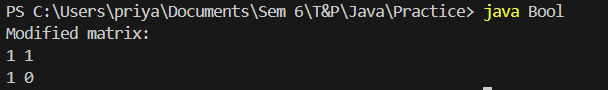
System.out.println();

}

scanner.close();

}

}

**Output:  
**

**Time Complexity:** O(n2)

**Space Complexity:** O (1)

**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}}

Output: 1 2 3 4 5 6 12 18 17 16 15 14 13 7 8 9 10 11

Explanation: The output is matrix in spiral format.

**Code:**

public class SpiralMat {

static void spir(int[][] arr){

int m=arr.length;

int n=arr[0].length;

int top=0, bot=m-1, l=0, r=n-1;

while(top<=bot&&l<=r){

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

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

}

top++;

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

System.out.print(arr[i][r] + " ");

}

r--;

if(top <= bot){

for(int i=r;i>=l;i--){

System.out.print(arr[bot][i] + " ");

}

bot--;

}

if(l<=r){

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

System.out.print(arr[i][l] + " ");

}

l++;

}

}

}

public static void main(String ar[]){

int[][] arr={

{1,2,3,4},

{5,6,7,8},

{9,10,11,12},

{13,14,15,16}

};

spir(arr);

}

}

**Output:**

****

**Time Complexity:** O (n2)

**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

**Code:**

import java.util.Stack;

public class Parentheses {

static void balanced(String s){

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

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

char c=s.charAt(i);

if(c=='('){

st.push(c);

}

else if(c==')'){

char b=st.peek();

if(b=='('){

st.pop();

}

else{

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

}

}

System.out.print("Balanced");

}

}

public static void main(String ar[]){

String s="((()))()()";

balanced(s);

}

}

**Output:**



**Time Complexity:** 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.

Input: s1 = “allergy” s2 = “allergic”

Output: false

Explanation: Characters in both the strings are not same. s1 has extra character „y‟ and s2 has

extra characters „i‟ and „c‟, so they are not anagrams.

Input: s1 = “g”, s2 = “g”

Output: true

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

**Code:**

import java.util.HashMap;

import java.util.Scanner;

public class Anagram {

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

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

return false;

}

HashMap<Character, Integer> charCountMap = new HashMap<>();

for (char c : s1.toCharArray()) {

charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);

}

for (char c : s2.toCharArray()) {

if (!charCountMap.containsKey(c) || charCountMap.get(c) == 0) {

return false;

}

charCountMap.put(c, charCountMap.get(c) - 1);

}

return true;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter first string: ");

String s1 = scanner.nextLine();

System.out.print("Enter second string: ");

String s2 = scanner.nextLine();

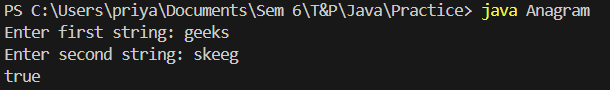
System.out.println(isAnagram(s1, s2));

scanner.close();

}

}

**Output:**



**Time Complexity:** O (n)

**Space Complexity:** O (1)

**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.

Input: str = “Geeks”

Output: “ee”

Input: str = “abc”

Output: “a”

Input: str = “”

Output: “”

**Code:**

public class LongestPalindrome {

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;

}

static String longestPalindrome(String s) {

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

return s;

}

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);

}

}

}

return maxStr;

}

public static void main(String ar[]){

String str="forgeeksskeegfor";

System.out.print(longestPalindrome(str));

}

}

**Output:**



**Time Complexity:** O (n2)

**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.

Input: arr[] = [“hello”, “world”]

Output: -1

Explanation: There‟s no common prefix in the given strings.

Code:

class LongestComPre {

static String longestCommonPrefix(String[] strs) {

if (strs == null || strs.length == 0) return "";

String prefix = strs[0];

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

while (!strs[i].startsWith(prefix)) {

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

if (prefix.isEmpty()) return "";

}

}

return prefix;

}

public static void main(String ar[]){

String[] str={

"geeksforgeeks",

"geeks",

"geek",

"geezer"

};

System.out.print(longestCommonPrefix(str));

}

}

**Output:**



**Time Complexity:** O (n log n)

**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]

**Code:**

import java.util.Stack;

public class DeleteMiddleElement {

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

if (stack.isEmpty()) {

return;

}

int middle = size / 2;

if (current == middle) {

stack.pop();

return;

}

int temp = stack.pop();

deleteMiddle(stack, current + 1, size);

stack.push(temp);

}

public static void deleteMiddleElement(Stack<Integer> stack) {

deleteMiddle(stack, 0, stack.size());

}

public static void main(String[] args) {

Stack<Integer> stack1 = new Stack<>();

stack1.push(1);

stack1.push(2);

stack1.push(3);

stack1.push(4);

stack1.push(5);

deleteMiddleElement(stack1);

System.out.println(stack1);

Stack<Integer> stack2 = new Stack<>();

stack2.push(1);

stack2.push(2);

stack2.push(3);

stack2.push(4);

stack2.push(5);

stack2.push(6);

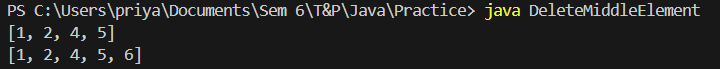
deleteMiddleElement(stack2);

System.out.println(stack2);

}

}

**Output:**



**Time Complexity:** O(n)

**Space Complexity:** O(1)

**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

Explanation: 13 and 12 don‟t have any element greater than them present on the right side

**Code:**

import java.util.Stack;

import java.util.HashMap;

public class NextGreaterElement {

public static void printNextGreaterElement(int[] arr) {

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

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

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

int current = arr[i];

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

stack.pop();

}

if (stack.isEmpty()) {

ngeMap.put(current, -1);

} else {

ngeMap.put(current, stack.peek());

}

stack.push(current);

}

for (int num : arr) {

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

}

}

public static void main(String[] args) {

int[] arr1 = {4, 5, 2, 25};

System.out.println("Output for arr1:");

printNextGreaterElement(arr1);

int[] arr2 = {13, 7, 6, 12};

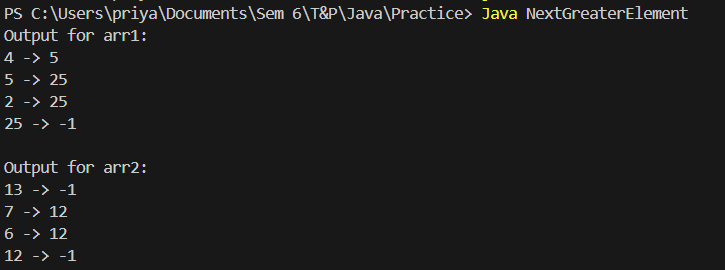
System.out.println("\nOutput for arr2:");

printNextGreaterElement(arr2);

}

}

**Output:**

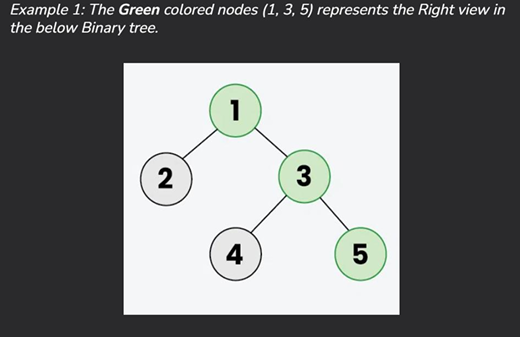


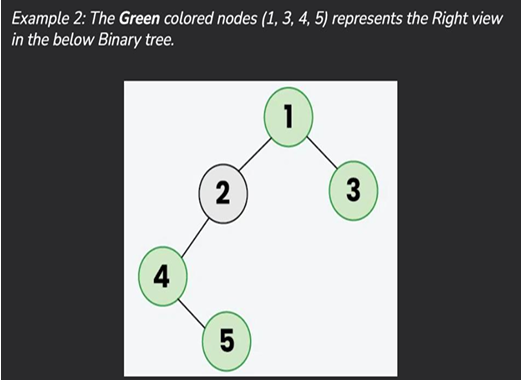
**Time Complexity: O (n)**

**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.





**Code:**

import java.util.\*;

class TreeNode {

int val;

TreeNode left, right;

TreeNode(int val) {

this.val = val;

left = right = null;

}

}

public class RightViewBinaryTree {

public static void printRightView(TreeNode root) {

if (root == null) {

return;

}

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

queue.add(root);

while (!queue.isEmpty()) {

int levelSize = queue.size();

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

TreeNode currentNode = queue.poll();

if (i == levelSize - 1) {

System.out.print(currentNode.val + " ");

}

if (currentNode.left != null) {

queue.add(currentNode.left);

}

if (currentNode.right != null) {

queue.add(currentNode.right);

}

}

}

}

public static void main(String[] args) {

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

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

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

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

printRightView(root);

}

}

**Output:**



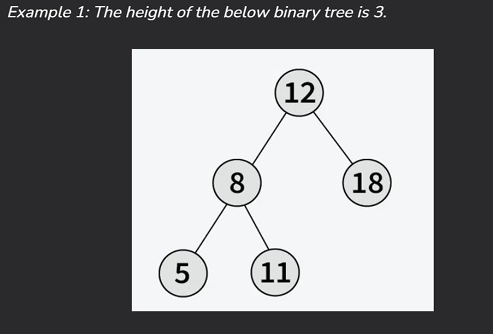
**Time Complexity:** O (n)

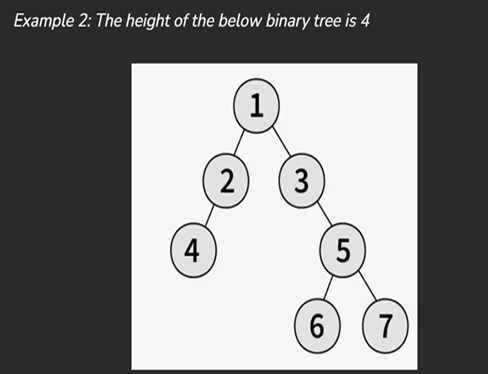
**Space Complexity:** O(1)

**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.





**Code:**

class TreeNode {

int val;

TreeNode left, right;

TreeNode(int val) {

this.val = val;

left = right = null;

}

}

public class BinaryTreeHeight {

public static int maxDepth(TreeNode root) {

if (root == null) {

return 0;

}

int leftDepth = maxDepth(root.left);

int rightDepth = maxDepth(root.right);

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

}

public static void main(String[] args) {

TreeNode root = new TreeNode(12);

root.left = new TreeNode(8);

root.right = new TreeNode(18);

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

root.left.right = new TreeNode(11);

System.out.println("Height of the binary tree: " + maxDepth(root)); // Expected Output: 3

}

}

**Output:**



**Time Complexity:** O (n)