**1.Maximum Subarray Sum – Kadane‟s Algorithm:**  
  
Solution:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int maxsubarray\_sum(vector<int> arr) {

int n = arr.size();

int sum = 0;

int maximum = 0;

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

sum += arr[i];

if (sum < 0) {

sum = 0;

}

maximum = max(sum, maximum);

}

return maximum;

}

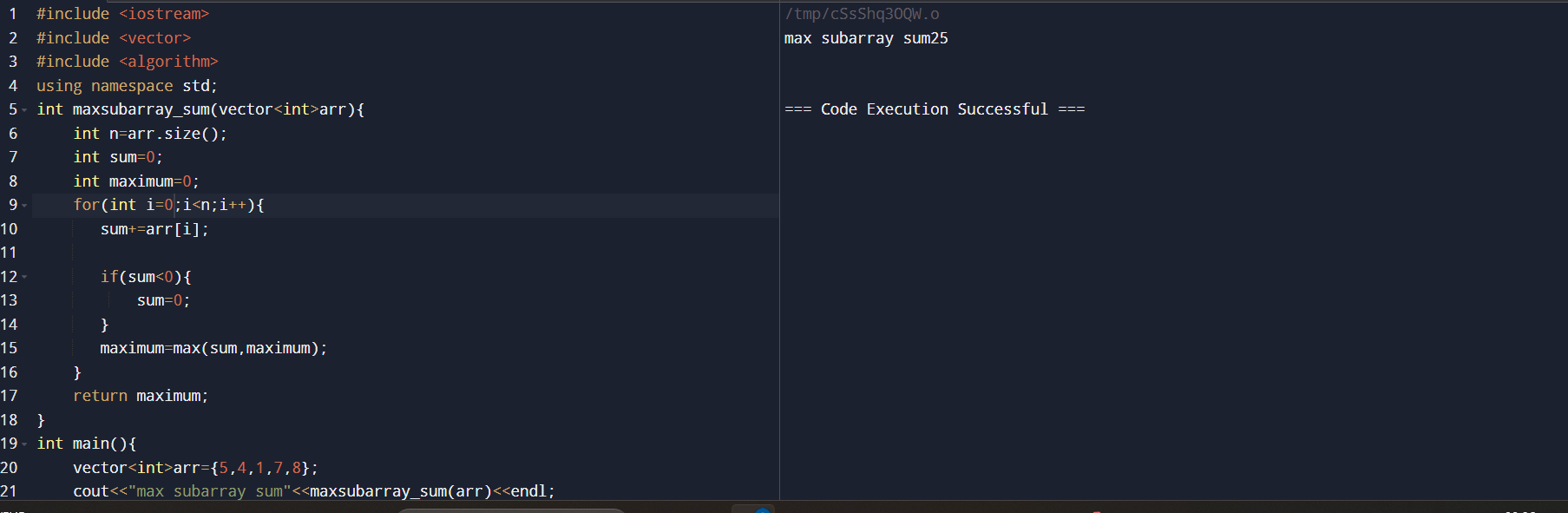
int main() {

vector<int> arr = {5, 4, -1, 7, 8};

cout << "Max subarray sum: " << maxsubarray\_sum(arr) << endl;

return 0;

}

t.c:O(n)  


**2.Maximum Product Subarray**#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int maxsubarray\_prdt(vector<int> arr) {

int n = arr.size();

int min\_prdt = arr[0];

int max\_prdt = arr[0];

int res = arr[0];

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

if (arr[i] < 0) swap(max\_prdt, min\_prdt);

max\_prdt = max(arr[i], max\_prdt \* arr[i]);

min\_prdt = min(arr[i], min\_prdt \* arr[i]);

res = max(res, max\_prdt);

}

return res;

}

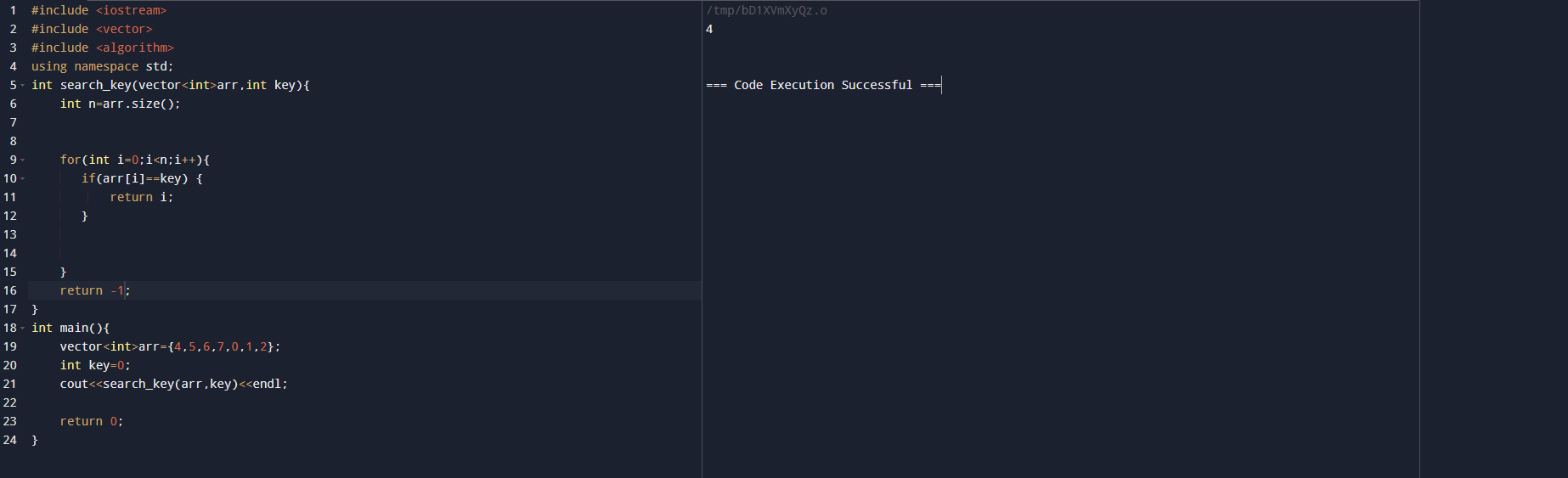
int main() {

vector<int> arr = {-2, 6, -3, -10, 0, 2};

cout << "Max subarray product: " << maxsubarray\_prdt(arr) << endl;

return 0;

}

TC:O(N)  


**3.Search in a sorted and rotated Array**  
  
#include <iostream>

#include <vector>

using namespace std;

int search\_key(vector<int> arr, int key) {

int n = arr.size();

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

if (arr[i] == key) {

return i;

}

}

return -1;

}

int main() {

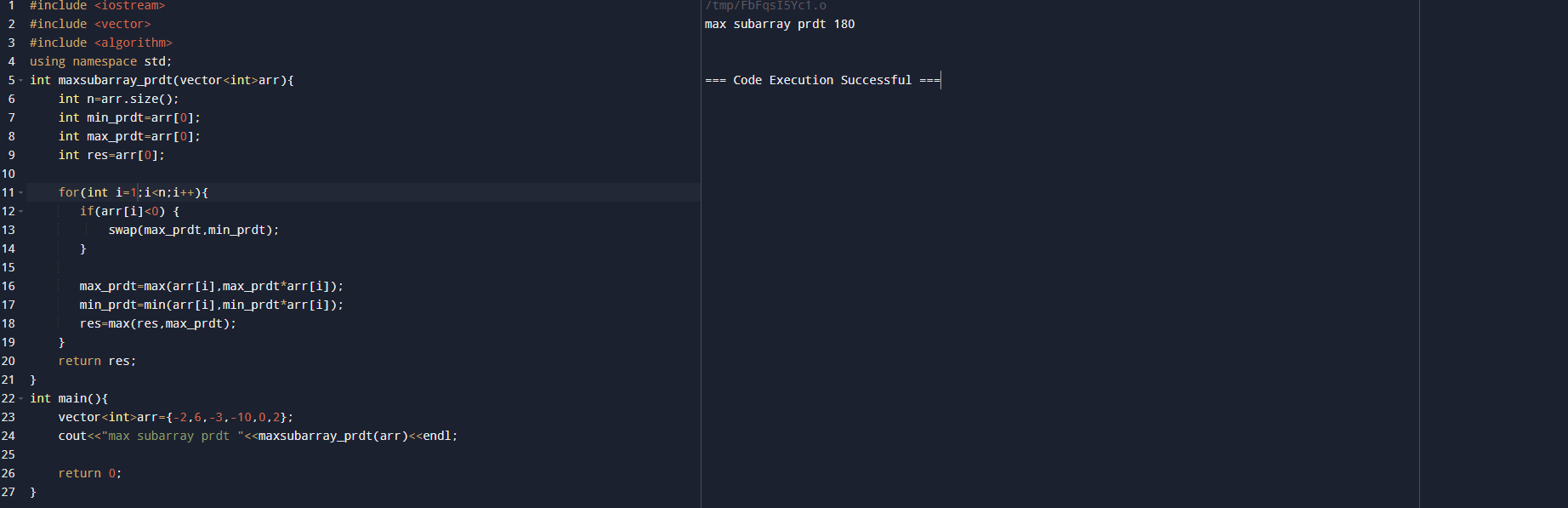
vector<int> arr = {4, 5, 6, 7, 0, 1, 2};

int key = 0;

cout << search\_key(arr, key) << endl;

return 0;

}

T.C:O(N)  
  
  
  
**4.Container with Most Water**  
  
  
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

int container\_with\_mostwater(vector<int> arr) {

int maxarea = 0;

int l = 0, r = arr.size() - 1;

while (l < r) {

int width = r - l;

int h = min(arr[l], arr[r]);

int water = width \* h;

maxarea = max(maxarea, water);

if (arr[l] < arr[r]) {

l++;

} else {

r--;

}

}

return maxarea;

}

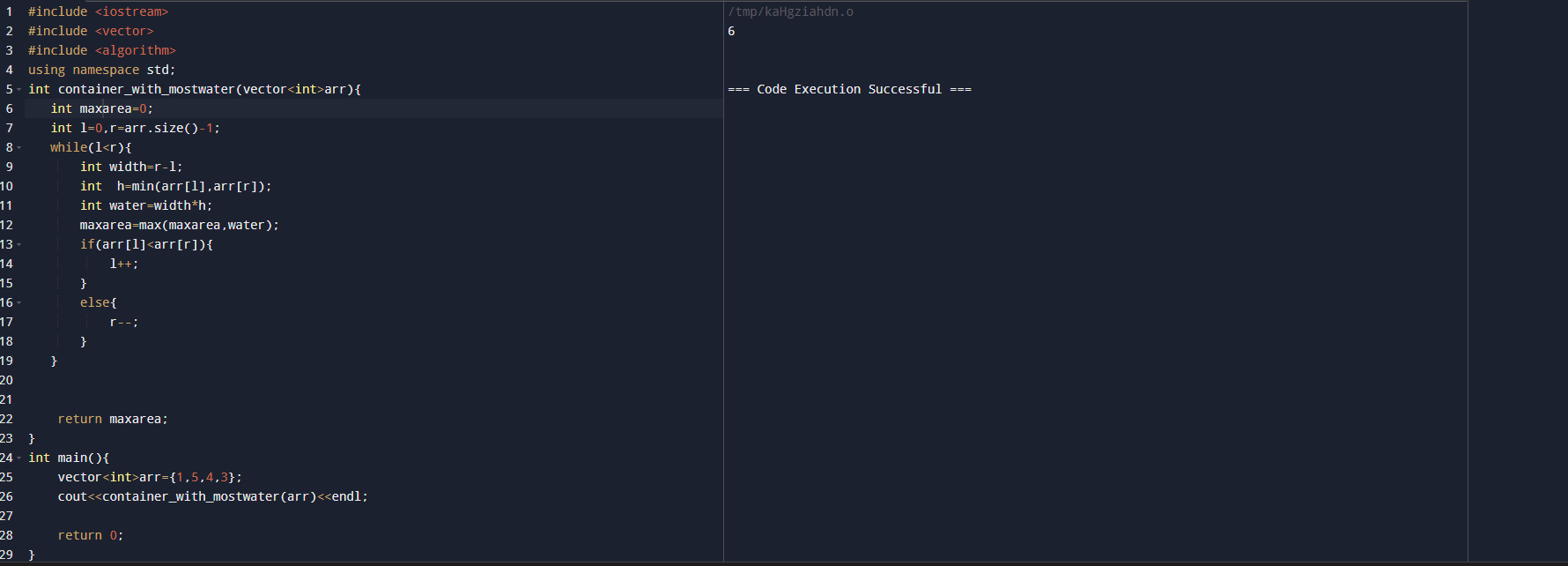
int main() {

vector<int> arr = {1, 5, 4, 3};

cout << container\_with\_mostwater(arr) << endl;

return 0;

}

TC:O(N)  


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

Input: 100

Output:

933262154439441526816992388562667004907159682643816214685929638952175999932299

156089414639761565182862536979208272237582511852109168640000000000000000000000

00

Input: 50

Output: 30414093201713378043612608166064768844377641568960512000000000000

Solution:  
package test;

import java.math.BigInteger;

import java.util.Scanner;

public class factorial {

public static void main(String[] args) {

Scanner num = new Scanner(System.***in***);

int n = num.nextInt();

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

}

public static BigInteger fact(int n) {

BigInteger f = new BigInteger("1");

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

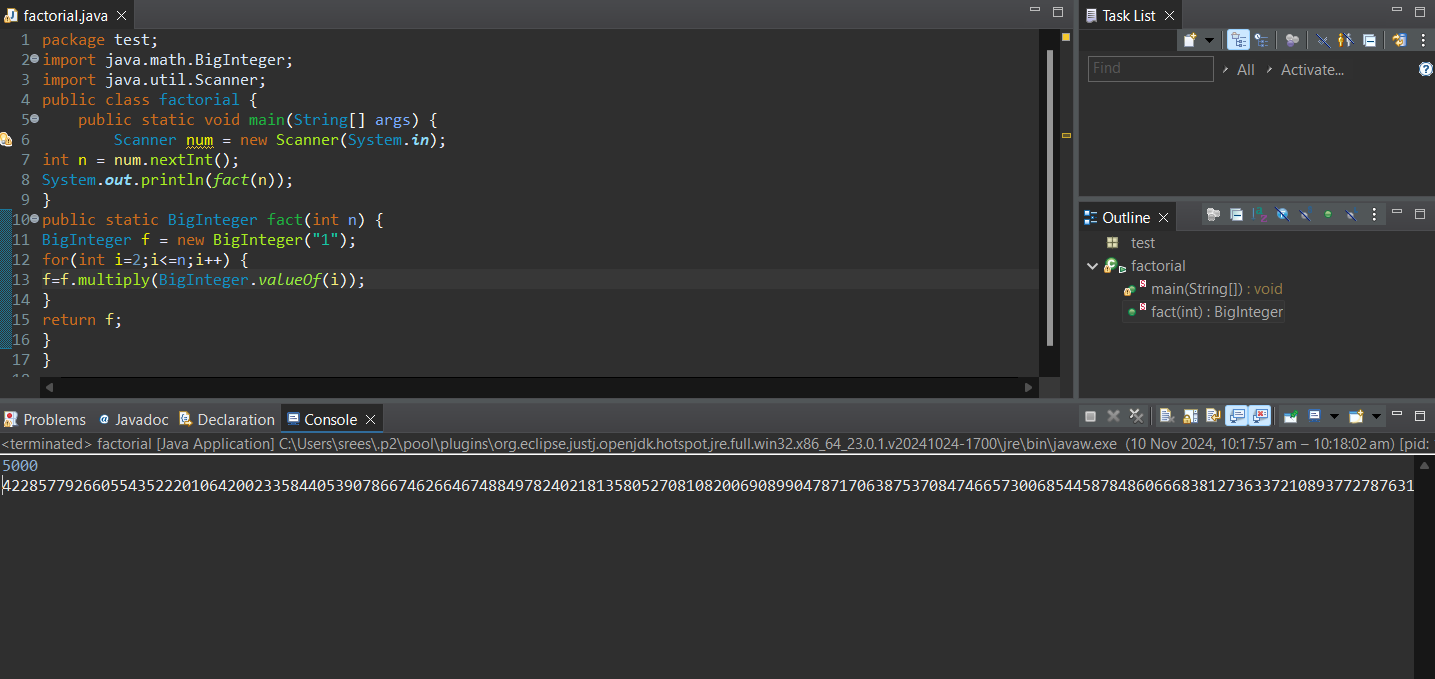
f=f.multiply(BigInteger.*valueOf*(i));

}

return f;

}

}

TC: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

Solution:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

    int trap(vector<int>& arr) {

        if(arr.size()<3) return 0;

        int l=0;

        int r=arr.size()-1;

        int l\_max=arr[l];

        int r\_max=arr[r];

        int tot\_water=0;

        while(l<r){

            if(arr [l]<arr[r]){

                if(arr[l]<l\_max){

                    tot\_water+=l\_max-arr[l];

                }else{

                l\_max=arr[l];

                }

            l++;

        }else{

            if(arr[r]<r\_max){

                    tot\_water+=r\_max-arr[r];

                }else{

                r\_max=arr[r];

                }

            r--;

        }

    }

    return tot\_water;

}

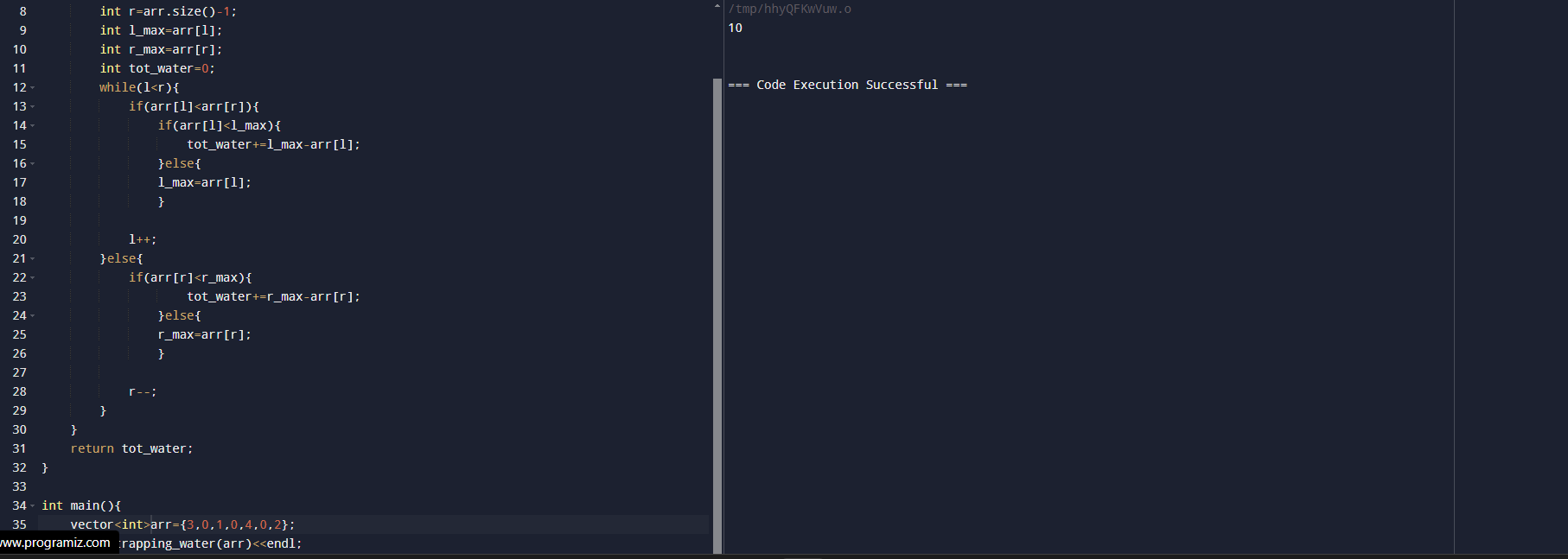
};

int main(){

    vector<int> arr={3, 0, 1, 0, 4, 0, 2};

    cout<<trapping\_water(arr)<<endl;

    return 0;

}  


**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.

Solution:  
  
int findMinDiff(vector<int>& a, int m) {

// code here

sort(a.begin(),a.end());

int n=a.size();

int mindiff=INT\_MAX;

for(int i=0;i+m-1<n;i++){

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

if(diff<mindiff){

mindiff=diff;

}

}

return mindiff;

}  
int main(){

    vector<int> arr={ 3, 4, 1, 9, 56, 7, 9, 12};

int m=5;

    cout<< findMinDiff (arr)<<endl;

    return 0;

}

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].

Solution:  
vector<vector<int>> merge(vector<vector<int>>& intervals) {

        int n=intervals.size();

        sort(intervals.begin(),intervals.end());

        vector<vector<int>>ans;

        for(int i=0;i<n;i++){

            if(ans.empty()||intervals[i][0]>ans.back()[1]){

                ans.push\_back(intervals[i]);

            }else{

                ans.back()[1]=max(intervals[i][1],ans.back()[1]);

            }

        }

        return ans;

    }

};

int main(){

    vector<int> intervals{ [1,3],[2,6],[8,10],[15,18] };

    cout<< merge(arr)<<endl;

    return 0;

}

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}}

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}}  
  
Solution:  
void setOnes(vector<vector<int>>& matrix) {

        int col0=1;

        int m=matrix.size();

        int n=matrix[0].size();

        for(int i=0;i<m;i++){

            for(int j=0;j<n;j++){

                if(matrix[i][j]==1){

                    matrix[i][0]=1;

                    if(j!=0){

                        matrix[0][j]=1;

                    }

                    else{

                        col0=1;

                    }

                }

            }

        }

        for(int i=1;i<m;i++){

            for(int j=1;j<n;j++){

                if(matrix[i][j]!=1){

                    if(matrix[i][0]==1|| matrix[0][j]==1){

                         matrix[i][j]=1;

                    }

                }

            }

        }

        if(matrix[0][0]==1){

            for(int j=0;j<n;j++){

                matrix[0][j]=0;

            }

        }

        if(col0==1){

            for(int i=0;i<m;i++){

                matrix[i][0]=1;

            }

        }

    }

};  
int main(){

vector<vector<int>> matrix= {[1,1,1],[1,0,1],[1,1,1]}

    cout<< setOnes(arr)<<endl;

    return 0;

}

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.  
  
Solution:  
vector<int> spiralOrder(vector<vector<int>>& matrix) {

        int n=matrix.size();

        int m=matrix[0].size();

        int left=0;

        int right=m-1;

        int top=0;

        int bottom=n-1;

        vector<int>res;

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

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

                res.push\_back(matrix[top][i]);

            }

            top++;

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

                res.push\_back(matrix[i][right]);

            }

            right--;

            if(top<=bottom){

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

                res.push\_back(matrix[bottom][i]);

            }

            bottom--;

            }

            if(left<=right){

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

                res.push\_back(matrix[i][left]);

            }

            left++;

            }

        }

        return res;

    }

};

int main(){

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

    cout<< spiralOrder(arr)<<endl;

    return 0;  
  
  
**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  
  
Solution:

#include <iostream>

#include <string>

using namespace std;

bool paranthesis\_balancing(const string& str) {

    int count = 0;

    for (char ch : str) {

        if (ch == '(') {

            count++;

        } else if (ch == ')') {

            count--;

            if (count < 0) {

                return false;

            }

        }

    }

    return count == 0;

}

int main() {

    string str = "((()))()()";

    if (paranthesis\_balancing(str)) {

        cout << "Balanced" << endl;

    } else {

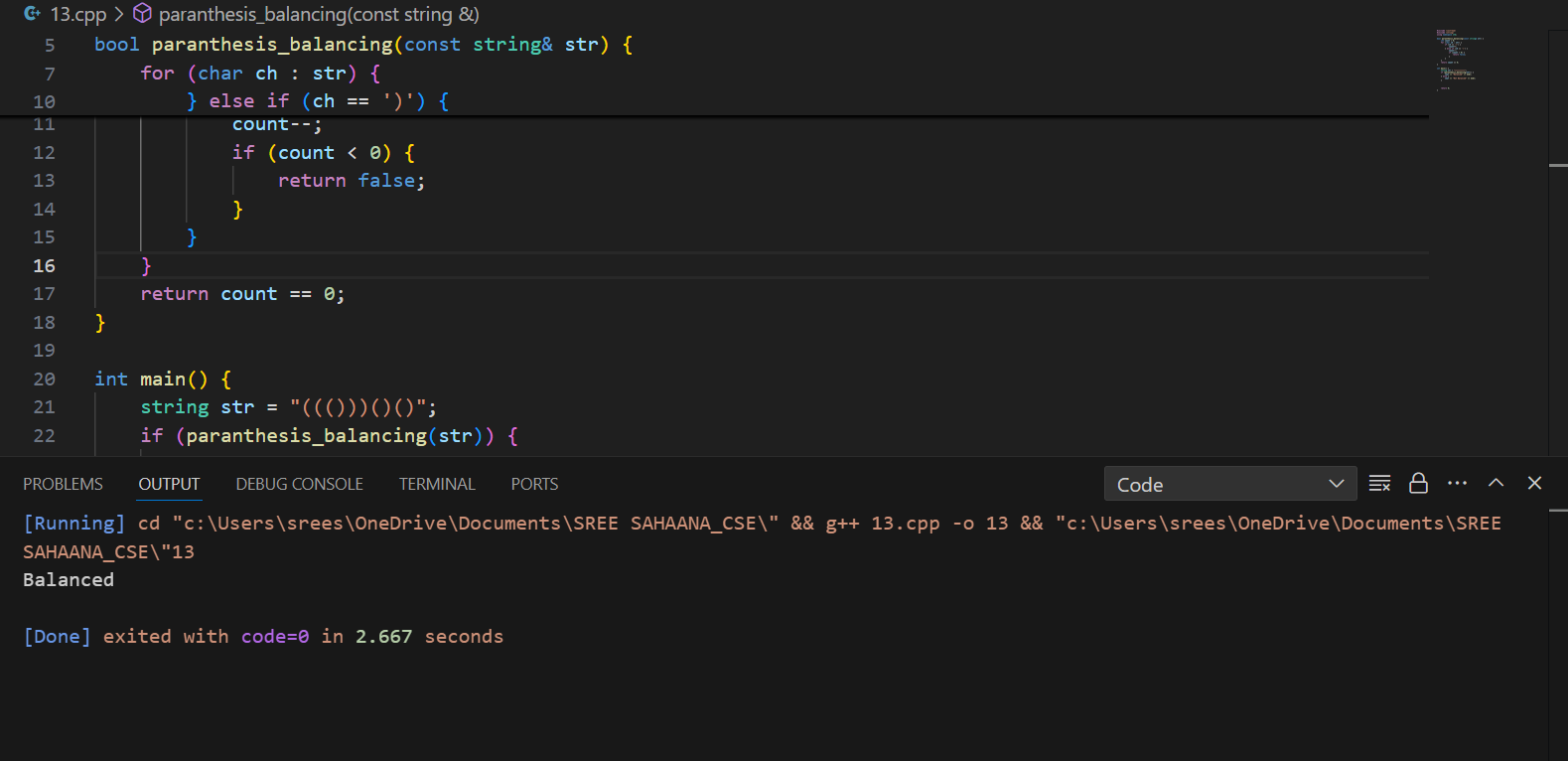
        cout << "Not Balanced" << endl;

    }

    return 0;

}

T.C:O(N)



**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

Solution:  
#include <iostream>

#include <string>

#include <vector>

using namespace std;

bool areAnagrams(const string& s1, const string& s2) {

    if (s1.length() != s2.length()) {

        return false;

    }

    vector<int> count(26, 0);

    for (int i = 0; i < s1.length(); i++) {

        count[s1[i] - 'a']++;

        count[s2[i] - 'a']--;

    }

    for (int c : count) {

        if (c != 0) {

            return false;

        }

    }

    return true;

}

int main() {

    string s1 = "geeks";

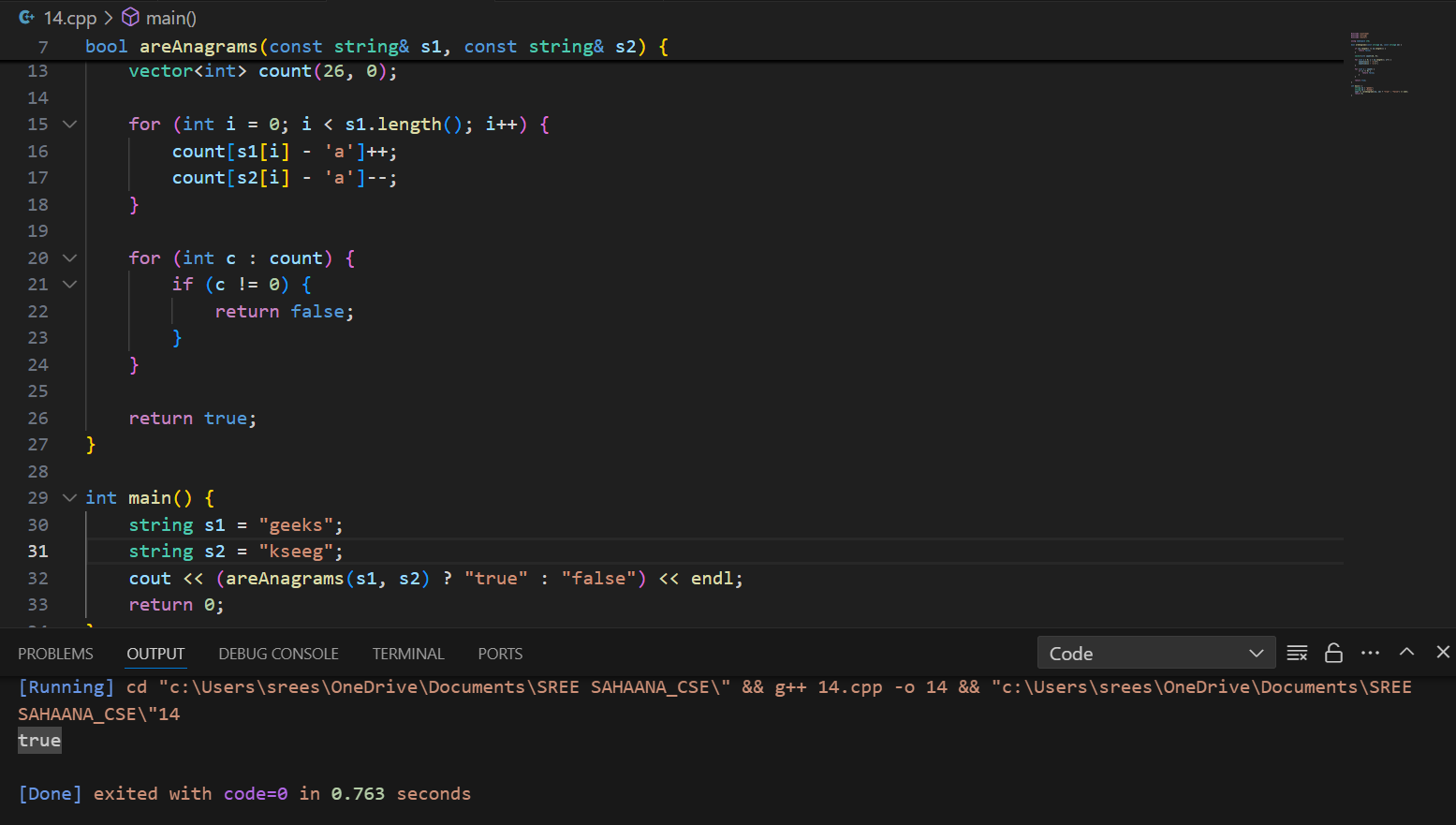
    string s2 = "kseeg";

    cout << (areAnagrams(s1, s2) ? "true" : "false") << endl;

    return 0;

}

T.C: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.

Input: str = “Geeks”

Output: “ee”

Input: str = “abc”

Output: “a”

Input: str = “”

Output: “”

Solution:  
#include <iostream>

#include <string>

using namespace std;

string expanding\_Around\_Center(const string& s, int l, int r) {

    while (l >= 0 && r < s.size() && s[l] == s[r]) {

        l--;

        r++;

    }

    return s.substr(l + 1, r - l - 1);

}

string longestPalindrome(string s) {

    if (s.empty()) return "";

    string longest;

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

        string oddPalindrome = expanding\_Around\_Center(s, i, i);

        if (oddPalindrome.size() > longest.size()) {

            longest = oddPalindrome;

        }

        string evenPalindrome = expanding\_Around\_Center(s, i, i + 1);

        if (evenPalindrome.size() > longest.size()) {

            longest = evenPalindrome;

        }

    }

    return longest;

}

int main() {

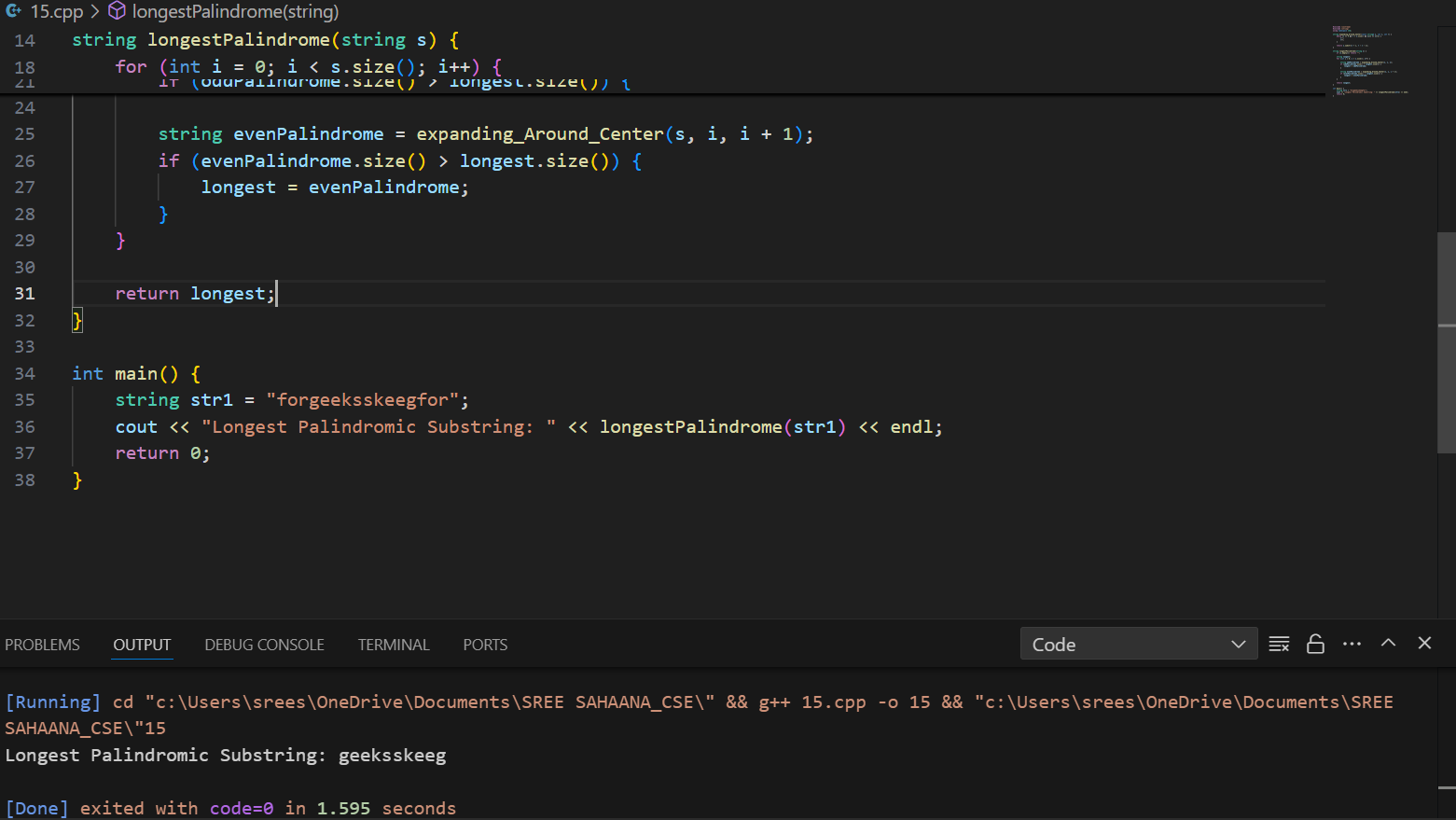
    string str1 = "forgeeksskeegfor";

    cout << "Longest Palindromic Substring: " << longestPalindrome(str1) << endl;

    return 0;

}

T.C:O(N^2)



**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

Solution:  
#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

string longestCommonPrefix(vector<string>& arr) {

    if (arr.empty()) return "-1";

    sort(arr.begin(), arr.end());

    string first = arr[0];

    string last = arr[arr.size() - 1];

    int minlen = min(first.size(), last.size());

    int i = 0;

    while (i < minlen && first[i] == last[i]) {

        i++;

    }

    string prefix = first.substr(0, i);

    return prefix.empty() ? "-1" : prefix;

}

int main() {

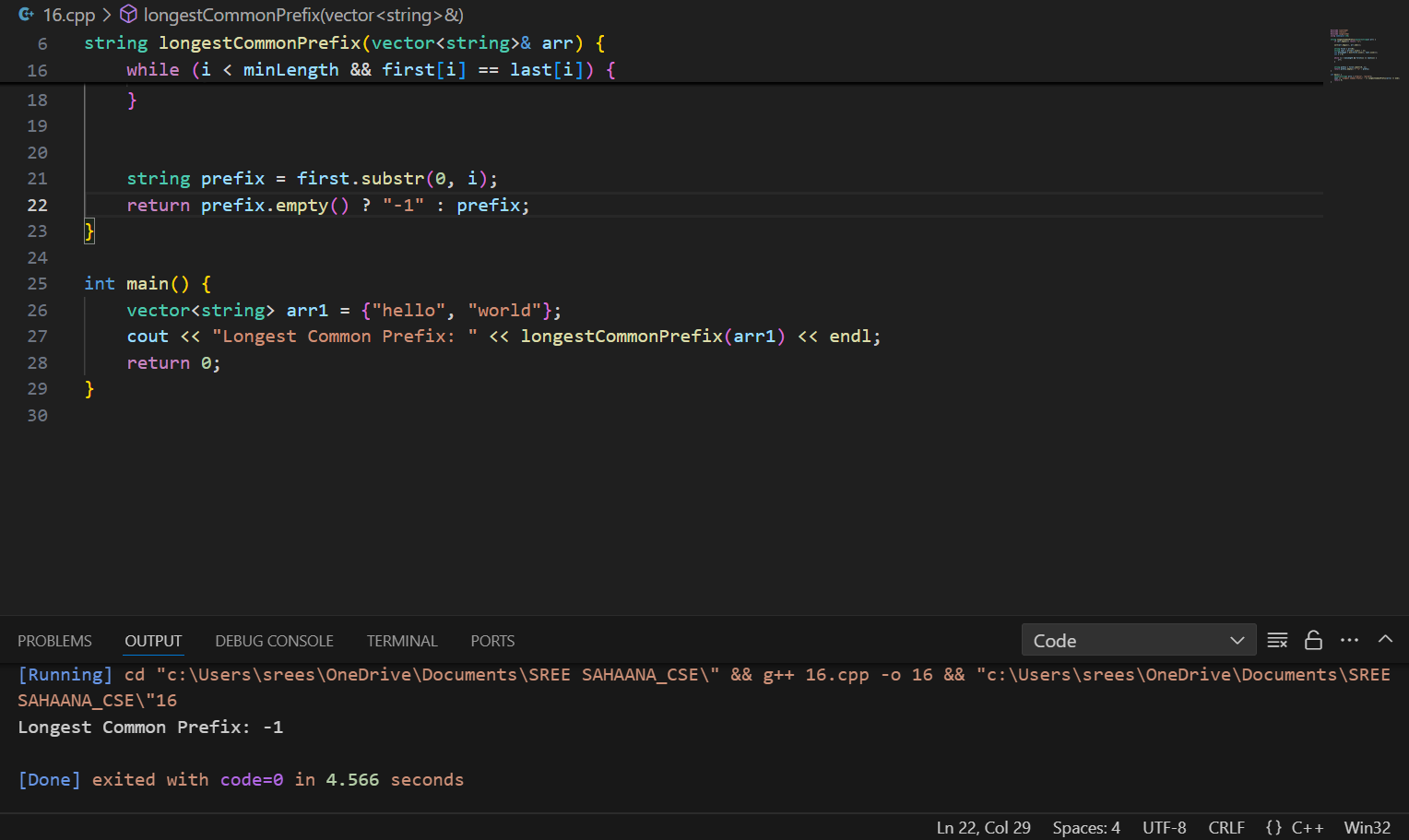
    vector<string> arr1 = {"hello", "world"};

    cout << "Longest Common Prefix: " << longestCommonPrefix(arr1) << endl;

    return 0;

}

T.C:O(nlogn+m)



**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]

Solution:  
#include <iostream>

#include <stack>

using namespace std;

void delete\_Middle\_element(stack<int>& st, int current, int middle) {

    if (current == middle) {

        st.pop();

        return;

    }

    int topElement = st.top();

    st.pop();

    delete\_Middle\_element(st, current + 1, middle);

    st.push(topElement);

}

void deleteMiddleElement(stack<int>& st) {

    if (!st.empty()) {

        int middle = st.size() / 2;

        delete\_Middle\_element(st, 0, middle);

    }

}

void printStack(stack<int>& st) {

    stack<int> temp;

    while (!st.empty()) {

        temp.push(st.top());

        st.pop();

    }

    cout << "Stack[] = [";

    while (!temp.empty()) {

        cout << temp.top();

        st.push(temp.top());

        temp.pop();

        if (!temp.empty()) cout << ", ";

    }

    cout << "]" << endl;

}

int main() {

    stack<int> st;

    st.push(1);

    st.push(2);

    st.push(3);

    st.push(4);

    st.push(5);

    cout << "Original ";

    printStack(st);

    deleteMiddleElement(st);

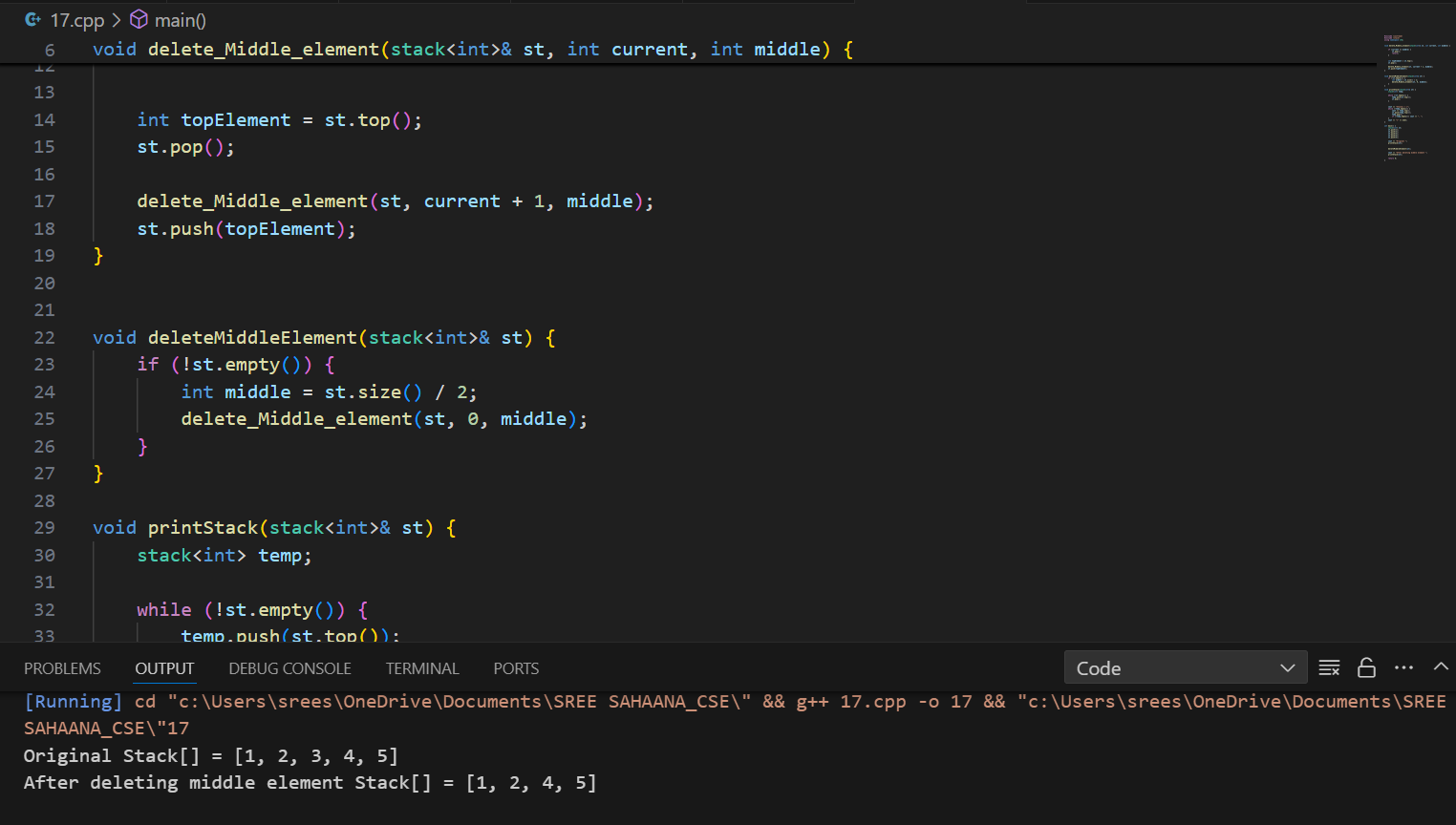
    cout << "After deleting middle element ";

    printStack(st);

    return 0;

}

T.CO(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

Solution:  
#include <iostream>

#include <vector>

using namespace std;

void nextGreaterElement(const vector<int>& arr) {

    int n = arr.size();

    for (int i = 0; i < n; i++) {

        int nextgreater = -1;

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

            if (arr[j] > arr[i]) {

                nextgreater = arr[j];

                break;

            }

        }

        cout << arr[i] << " -> " << nextgreater << endl;

    }

}

int main() {

    vector<int> arr = {4, 5, 2, 25};

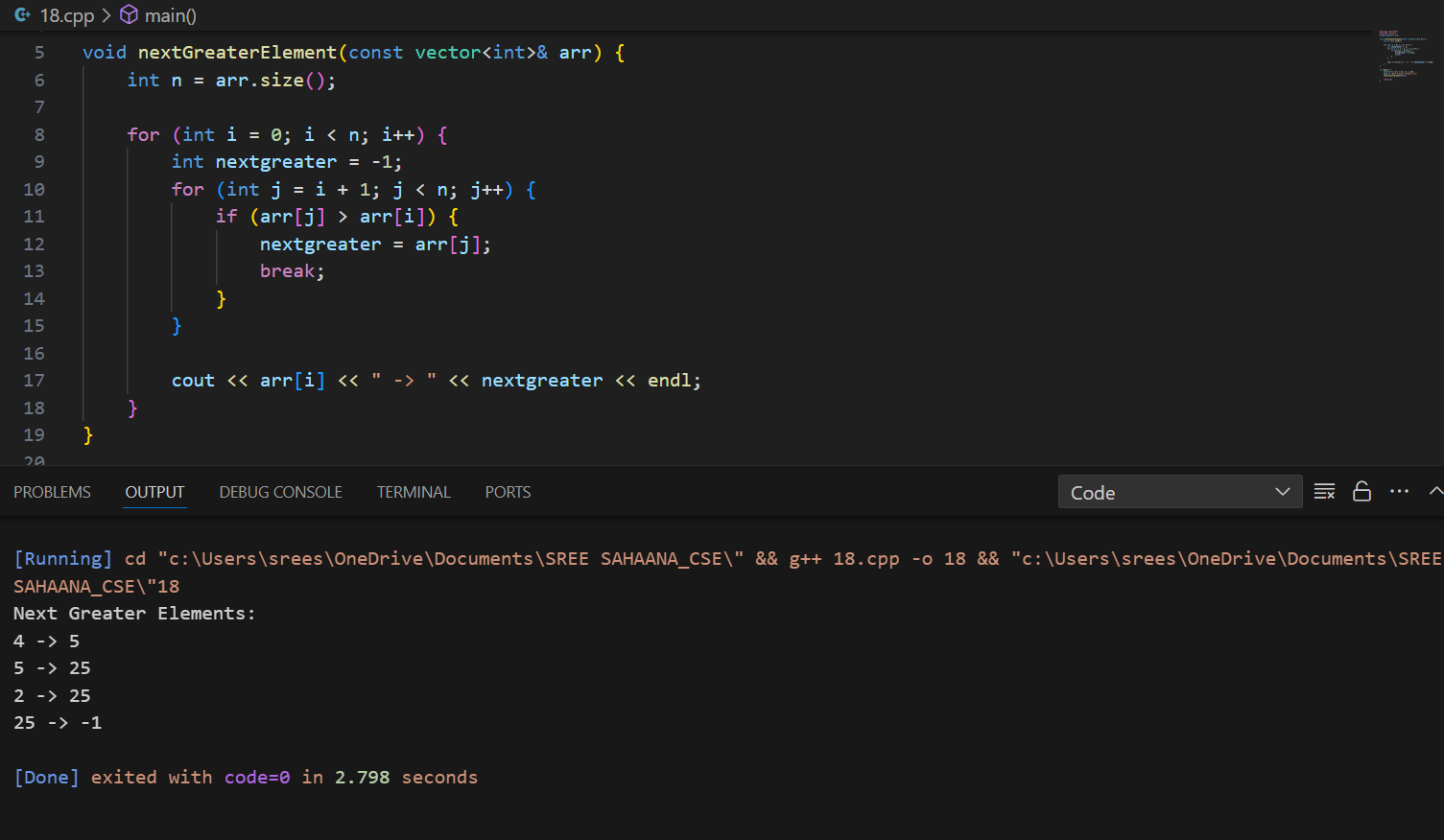
    cout << "Next Greater Elements:\n";

    nextGreaterElement(arr);

    return 0;

}

T.C:O(N^2)



**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.  
Solution:  
#include <iostream>

#include <vector>

#include <queue>

using namespace std;

struct TreeNode {

    int val;

    TreeNode \*left;

    TreeNode \*right;

    TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

vector<int> rightView(TreeNode\* root) {

    vector<int> result;

    if (!root) return result;

    queue<TreeNode\*> q;

    q.push(root);

    while (!q.empty()) {

        int n = q.size();

        for (int i = 0; i < n; i++) {

            TreeNode\* node = q.front();

            q.pop();

            if (i == n - 1) {

                result.push\_back(node->val);

            }

            if (node->left) q.push(node->left);

            if (node->right) q.push(node->right);

        }

    }

    return result;

}

TreeNode\* createTreeExample1() {

    TreeNode\* root = new TreeNode(1);

    root->left = new TreeNode(2);

    root->right = new TreeNode(3);

    root->left->left = new TreeNode(4);

    root->right->right = new TreeNode(5);

    return root;

}

TreeNode\* createTreeExample2() {

    TreeNode\* root = new TreeNode(1);

    root->left = new TreeNode(2);

    root->right = new TreeNode(3);

    root->left->left = new TreeNode(4);

    root->left->left->left = new TreeNode(5);

    return root;

}

int main() {

    TreeNode\* root1 = createTreeExample1();

    TreeNode\* root2 = createTreeExample2();

    vector<int> result1 = rightView(root1);

    vector<int> result2 = rightView(root2);

    cout << "Right view of Example 1: ";

    for (int val : result1) cout << val << " ";

    cout << endl;

    cout << "Right view of Example 2: ";

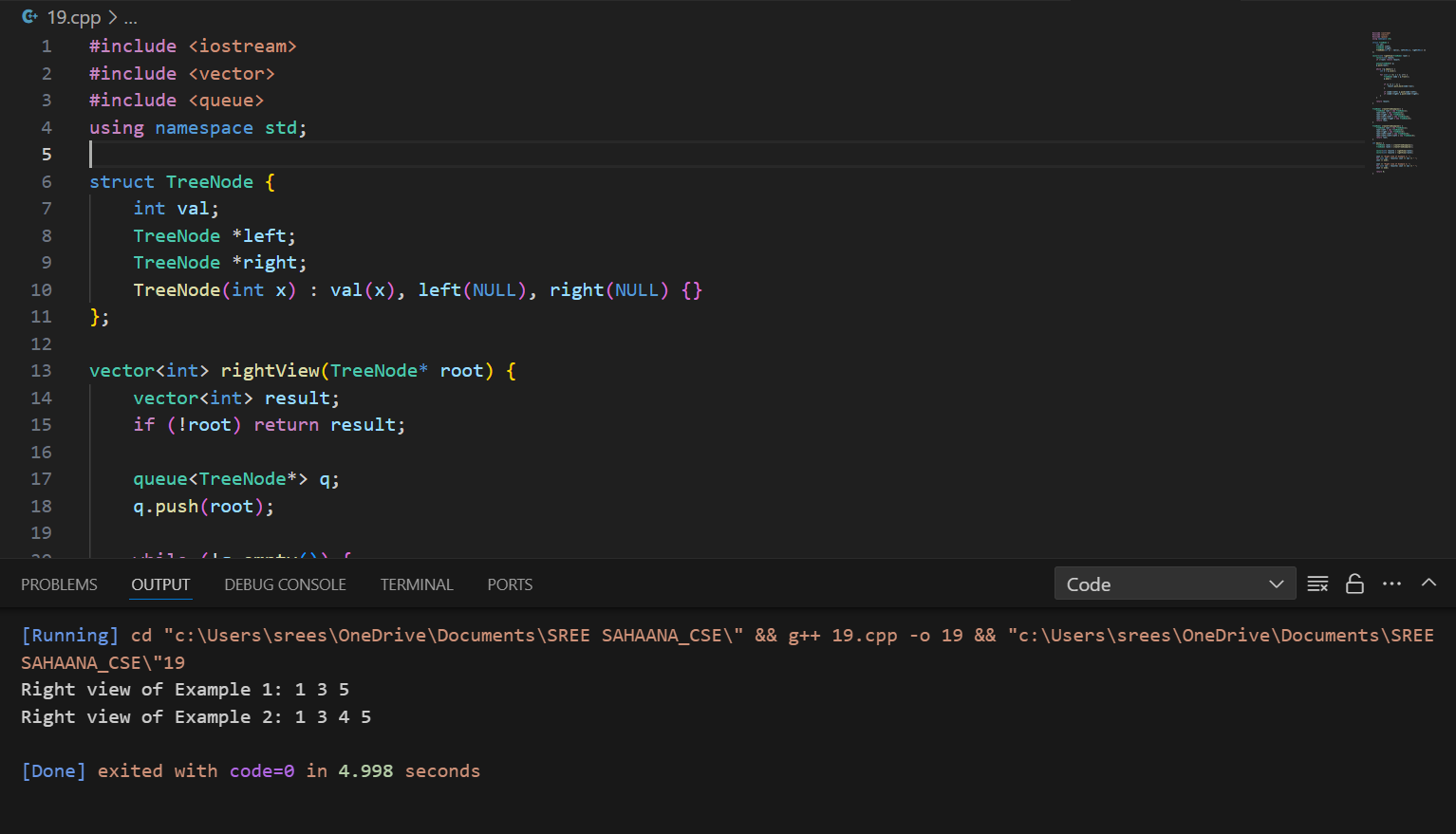
    for (int val : result2) cout << val << " ";

    cout << endl;

    return 0;

}

T.C: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.

Solution:  
#include <iostream>

using namespace std;

struct TreeNode {

    int val;

    TreeNode\* left;

    TreeNode\* right;

    TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

// Function to find the maximum depth of the binary tree

int maxDepth(TreeNode\* root) {

    // Base case: if the tree is empty, the depth is 0

    if (root == NULL) {

        return 0;

    }

    // Recursive case: the depth is 1 + the maximum of the depths of the left and right subtrees

    int leftDepth = maxDepth(root->left);

    int rightDepth = maxDepth(root->right);

    return 1 + max(leftDepth, rightDepth);

}

   TreeNode\* createTreeExample1() {

    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);

    return root;

}

TreeNode\* createTreeExample2() {

    TreeNode\* root = new TreeNode(1);

    root->left = new TreeNode(2);

    root->right = new TreeNode(3);

    root->left->left = new TreeNode(4);

    root->right->right = new TreeNode(5);

    root->right->right->left = new TreeNode(6);

    root->right->right->right = new TreeNode(6);

    return root;

}

int main() {

    TreeNode\* root1 = createTreeExample1();

    TreeNode\* root2 = createTreeExample2();

    int result1 = maxDepth(root1);

    int  result2 = maxDepth(root2);

    cout << "Maximum depth of Example 1: " << result1 << endl;

    cout << "Maximum depth of Example 2: " << result2 << endl;

    return 0;

}  
T.C:O(N)  
