**Coding practice Problems 09-11-2024**

**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:

#include<iostream>

#include<vector>

#include<climits>

using namespace std;

int maximumSubarray(vector<int>& nums){

int maxSum=INT\_MIN;

int sum=0;

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

sum+=nums[i];

if(sum<0){

sum=0;

}

maxSum=max(maxSum,sum);

}

return maxSum;

}

int main(){

vector<int> nums;

nums={2,3,-8,7,-1,2,3};

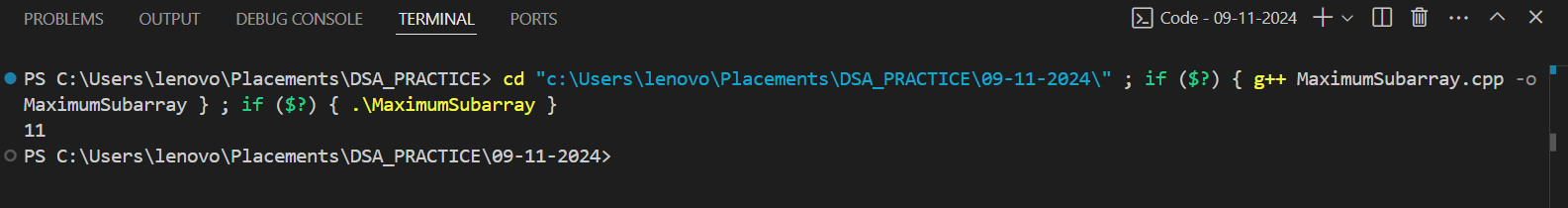
int b=maximumSubarray(nums);

cout<<b;

return 0;

}

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:

#include<iostream>

#include<vector>

#include<climits>

using namespace std;

int maximumProductSubarray(vector<int>& nums){

int maxProduct=INT\_MIN;

int product=1;

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

product\*=nums[i];

maxProduct=max(product,maxProduct);

if(product==0){

product=1;

}

}

product=1;

for(int i=nums.size()-1;i>=0;i--){

product\*=nums[i];

maxProduct=max(product,maxProduct);

if(product==0){

product=1;

}

}

return maxProduct;

}

int main(){

vector<int> nums;

nums={-2, 6, -3, -10, 0, 2};

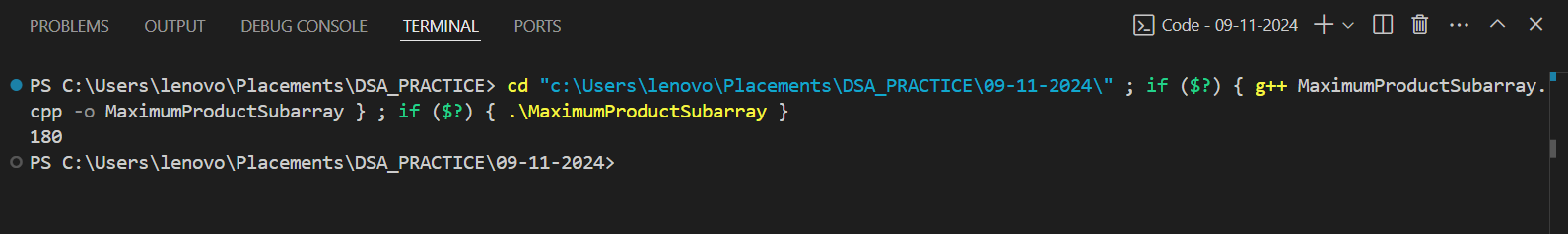
int b=maximumProductSubarray(nums);

cout<<b;

return 0;

}

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:

#include<iostream>

#include<vector>

using namespace std;

int searchElement(vector<int> &nums, int target){

int left=0,right=nums.size()-1;

while(left<=right){

int mid=(left+right)/2;

if(nums[mid]==target)return mid;

else if(nums[mid]>=nums[left]){

if(nums[left]<=target&&target<=nums[mid])right=mid-1;

else left=mid+1;

}else{

if(nums[mid]<=target&&target<=nums[right])left=mid+1;

else right=mid-1;

}

}

return -1;

}

int main(){

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

int target=0;

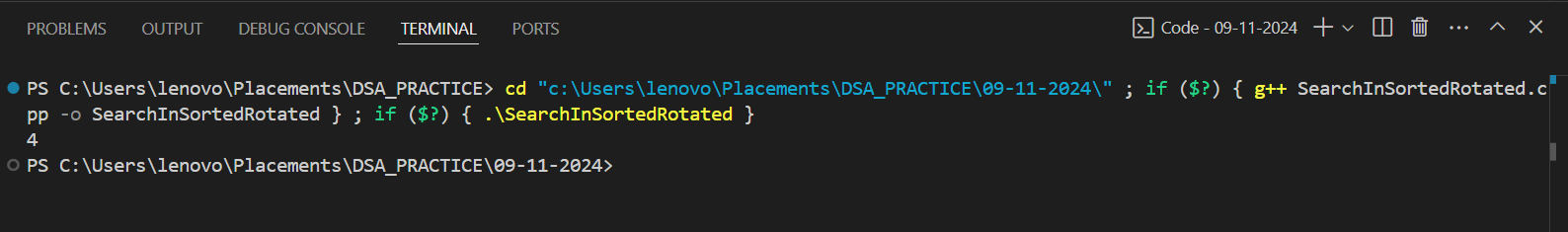
int b=searchElement(nums,target);

cout<<b;

return 0;

}

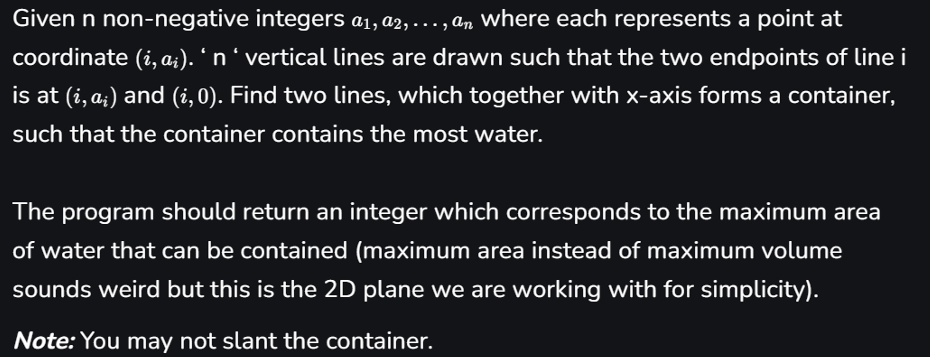
OUTPUT:



TIME COMPLEXITY: O(logN)

SPACE COMPLEXITY: O(1)

**4. 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

Input: arr = [3, 1, 2, 4, 5]

Output: 12

Explanation: 5 and 3 are distance 4 apart. So the size of the base = 4. Height of container = min(5, 3) = 3. So total area = 4 \* 3 = 12

CODE:

#include<iostream>

#include<vector>

#include<climits>

using namespace std;

int maxContainer(vector<int>& nums){

int low=0;

int high=nums.size()-1;

int maxArea=INT\_MIN;

while(low<=high){

int area=min(nums[low],nums[high])\*(high-low);

maxArea=max(area,maxArea);

if(nums[low]<nums[high]) low++;

else high--;

}

return maxArea;

}

int main(){

vector<int> nums;

nums={3,1,2,4,5};

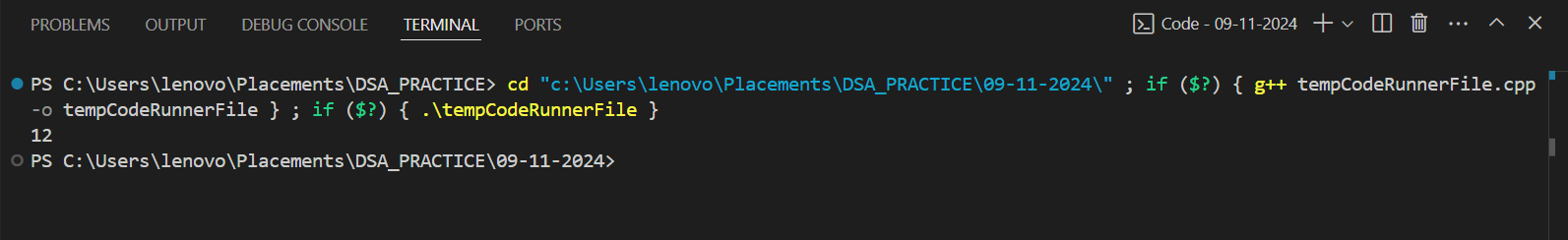
int b=maxContainer(nums);

cout<<b;

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(n)  
SPACE COMPLEXITY: O(1)

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

Input: 100

Output: 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

Input: 50

Output: 30414093201713378043612608166064768844377641568960512000000000000

CODE:  
#include <iostream>

using namespace std;

#define MAX 500

int multiply(int x, int res[], int res\_size) {

int carry = 0;

for (int i = 0; i < res\_size; i++) {

int prod = res[i] \* x + carry;

res[i] = prod % 10;

carry = prod / 10;

}

while (carry) {

res[res\_size] = carry % 10;

carry /= 10;

res\_size++;

}

return res\_size;

}

string factorial(int n) {

int res[MAX];

res[0] = 1;

int res\_size = 1;

for (int x = 2; x <= n; x++)

res\_size = multiply(x, res, res\_size);

string result = "";

for (int i = res\_size - 1; i >= 0; i--)

result += to\_string(res[i]);

return result;

}

int main() {

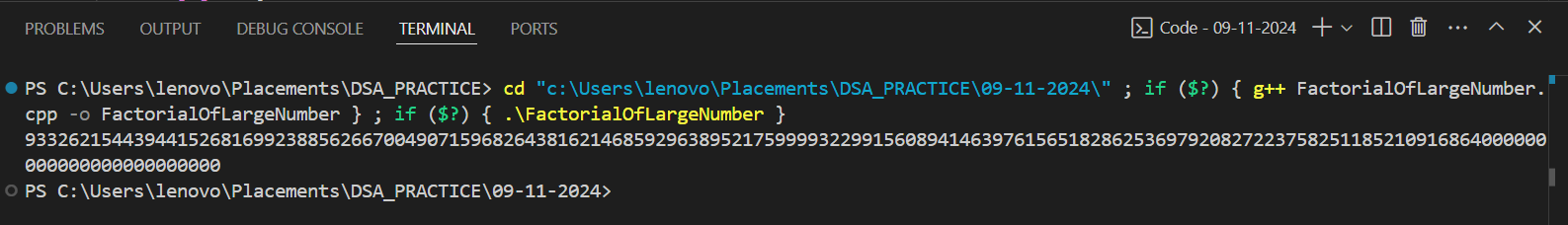
string b=factorial(100);

cout<<b;

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(N log(N!))

SPACE COMPLEXITY: O(max(no.of digits in Factorial)

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:

#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

int trappingWater(vector<int>&nums){

int n=nums.size(),res=0;

vector<int>left(n),right(n);

left[0]=nums[0];

for(int i=1;i<n;i++) left[i]=max(left[i-1],nums[i]);

right[n-1]=nums[n-1];

for(int i=n-2;i>=0;i--) right[i]=max(right[i+1],nums[i]);

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

int minOf2=min(left[i-1],right[i+1]);

if(minOf2>nums[i]) res+=minOf2-nums[i];

}

return res;

}

int main(){

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

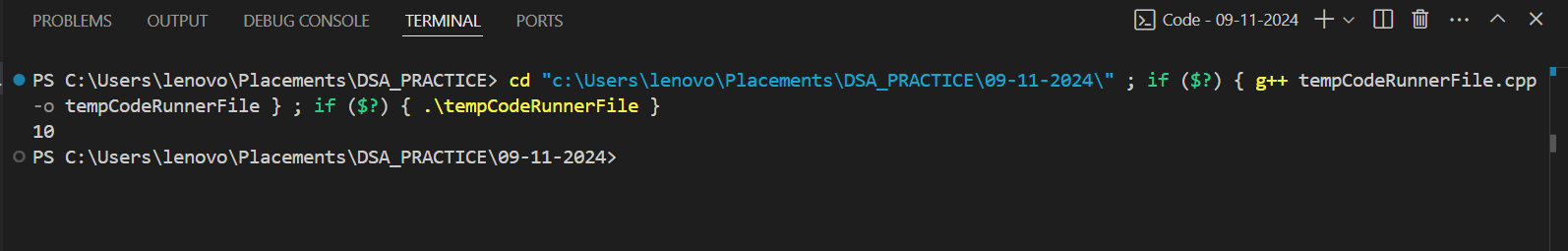
int b=trappingWater(nums);

cout<<b;

return 0;

}

OUTPUT:



TIME COMPLEXITY: 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.

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:

#include<iostream>

#include<vector>

#include<algorithm>

#include<climits>

using namespace std;

int findMinDiff(vector<int>&arr, int m){

int n=arr.size();

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

int minDiff=INT\_MAX;

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

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

if(diff<minDiff) minDiff=diff;

}

return minDiff;

}

int main(){

vector<int>arr={7,3,2,4,9,12,56};

int m=3;

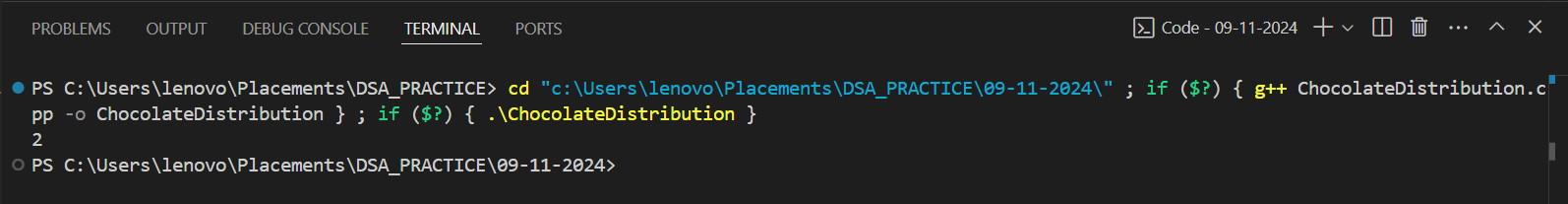
int b=findMinDiff(arr,m);

cout<<b;

return 0;

}

OUTPUT:



TIME COMPLEXITY: 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]].

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:

#include<iostream>

#include<vector>

#include<algorithm>

using namespace std;

vector<vector<int>> mergeIntervals(vector<vector<int>>& nums){

vector<vector<int>> solution;

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

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

if(solution.empty() || solution.back()[1]<nums[i][0]){

solution.push\_back(nums[i]);

}

else{

solution.back()[1]=max(solution.back()[1],nums[i][1]);

}

}

return solution;

}

int main(){

vector<vector<int>> nums={{7,8},{1,5},{2,4},{4,6}};

vector<vector<int>> b=mergeIntervals(nums);

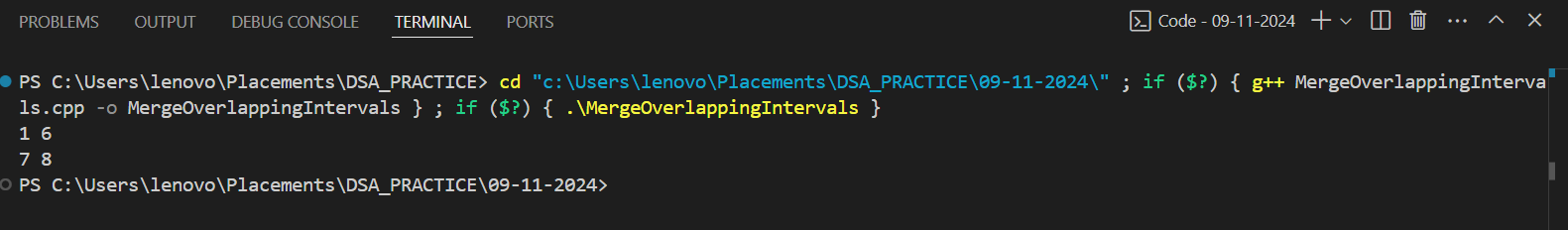
for(vector<int> num: b){

cout<<num[0]<<" "<<num[1]<<endl;

}

return 0;

}

OUTPUT:  


TIME COMPLEXITY: 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}}

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:

#include<iostream>

#include<vector>

using namespace std;

void fillZeroes(vector<vector<int>>&mat){

int rows=mat.size(),cols=mat[0].size();

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

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

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

int ind=i-1;

while(ind>=0){

if(mat[ind][j]!=1)mat[ind][j]=-1;

ind--;

}

ind=i+1;

while(ind<rows){

if(mat[ind][j]!=1)mat[ind][j]=-1;

ind++;

}

ind=j-1;

while(ind>=0){

if(mat[i][ind]!=1)mat[i][ind]=-1;

ind--;

}

ind=j+1;

while(ind<cols){

if(mat[i][ind]!=1)mat[i][ind]=-1;

ind++;

}

}

}

}

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

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

if(mat[i][j]<0)mat[i][j]=1;

}

}

}

int main(){

vector<vector<int>>mat={{1,0,0,1},{0,0,1,0},{0,0,0,0}};

fillZeroes(mat);

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

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

cout<<mat[i][j]<<" ";

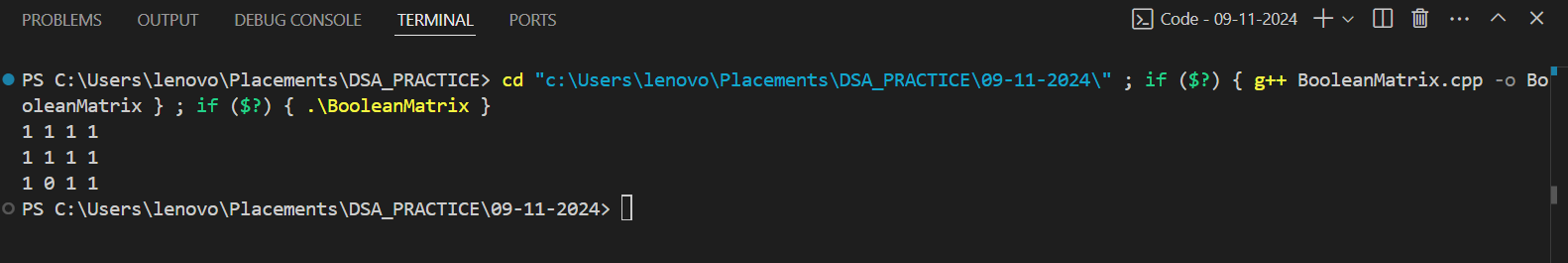
}

cout<<endl;

}

}

OUTPUT:



TIME COMPLEXITY:  
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:

#include<iostream>

#include<vector>

using namespace std;

void spiralPrint(int m,int n,vector<vector<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++) cout<<a[top][i]<<" ";

top++;

for(int i=top;i<=bottom;i++) cout<<a[i][right]<<" ";

right--;

if(top<=bottom){

for(int i=right;i>=left;i--) cout<<a[bottom][i]<<" ";

bottom--;

}

if(left<=right){

for(int i=bottom;i>=top;i--) cout<<a[i][left]<<" ";

left++;

}

}

}

int main(){

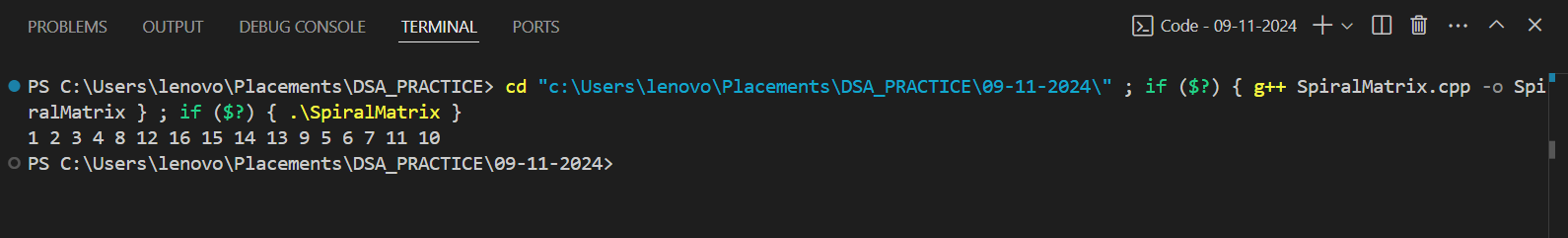
vector<vector<int>>matrix={{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};

spiralPrint(matrix.size(),matrix[0].size(),matrix);

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(m\*n)

SPACE COMPLEXITY: O(1)

**11. 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:

#include<iostream>

#include<stack>

using namespace std;

bool checkParantheses(string s){

stack<char> st;

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

if(s[i]=='(' || s[i]=='[' || s[i]=='{'){

st.push(s[i]);

}

else if(s[i]==']' || s[i]=='}' || s[i]==')'){

if(st.empty()){

return false;

}

else if(s[i]==')'){

if(st.top()=='('){

st.pop();

}

else{

return false;

}

}

else if(s[i]==']'){

if(st.top()=='['){

st.pop();

}

else{

return false;

}

}

else if(s[i]=='}'){

if(st.top()=='{'){

st.pop();

}

else{

return false;

}

}

}

}

return true;

}

int main(){

string s="“())((())";

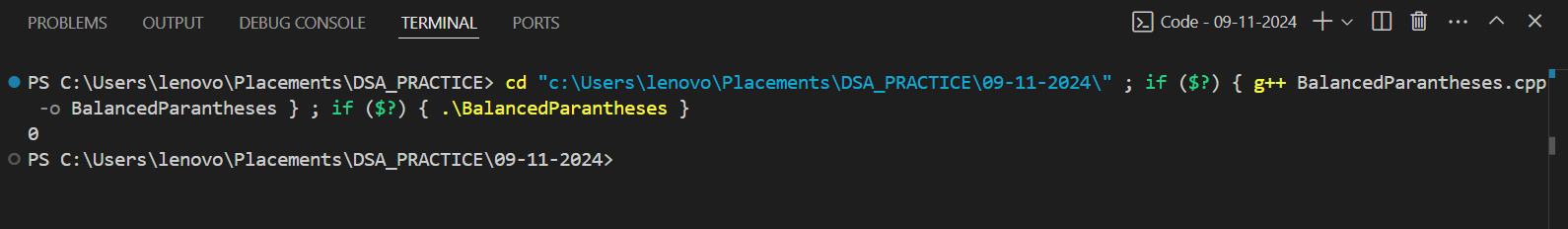
bool ans=checkParantheses(s);

cout<<ans;

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(n)  
SPACE COMPLEXITY: 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 = “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:

#include<iostream>

#include<algorithm>

#include<string>

using namespace std;

bool isAnagram(string s1, string s2){

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

return false;

}

sort(s1.begin(),s1.end());

sort(s2.begin(),s2.end());

if(s1==s2) return true;

else return false;

}

int main(){

string s1="allergy";

string s2="allergic";

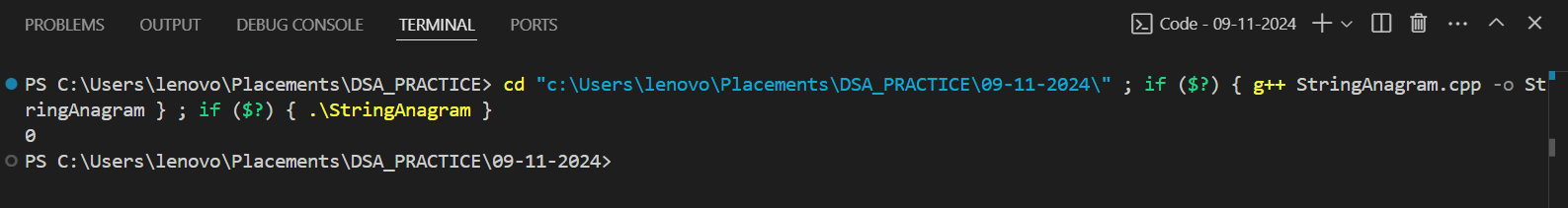
bool ans=isAnagram(s1,s2);

cout<<ans;

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(n log(n))

SPACE COMPLEXITY: O(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.

Input: str = “Geeks”

Output: “ee”

Input: str = “abc”

Output: “a”

Input: str = “”

Output: “”

#include <iostream>

#include <string>

using namespace std;

string longestPalindrome(string& s){

int n = s.length();

if(n == 0) return "";

int start = 0, maxLen = 1;

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

for(int j = 0; j <= 1; j++){

int low = i, hi = i + j;

while(low >= 0 && hi < n && s[low] == s[hi]){

int currLen = hi - low + 1;

if(currLen > maxLen){

start = low;

maxLen = currLen;

}

low--;

hi++;

}

}

}

return s.substr(start, maxLen);

}

int main(){

string s = "forgeeksskeegfor";

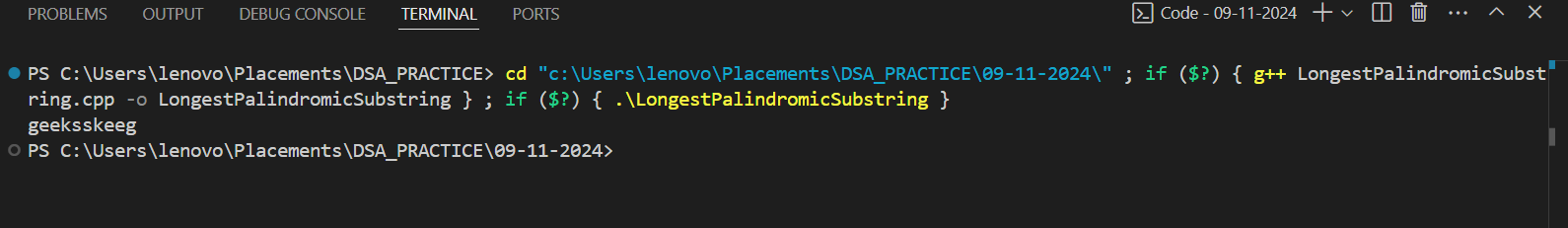
string b=longestPalindrome(s);

cout<<b;

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(n^2)

SPACE COMPLEXITY: O(1)

**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.

Input: arr[] = [“hello”, “world”]

Output: -1

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

CODE:

**include<iostream>**

**#include<algorithm>**

**#include<vector>**

**using namespace std;**

**string longestPrefix(vector<string>& s){**

**string ans="";**

**sort(s.begin(),s.end());**

**int i=0;**

**int j=0;**

**while(i<s[0].size() && j<s[s.size()-1].size()){**

**if(s[0][i]==s[s.size()-1][i]){**

**ans+=s[0][i];**

**i++;**

**j++;**

**}**

**else{**

**break;**

**}**

**}**

**return ans;**

**}**

**int main(){**

**vector<string> s={"geeksforgeeks", "geeks", "geek", "geezer"};**

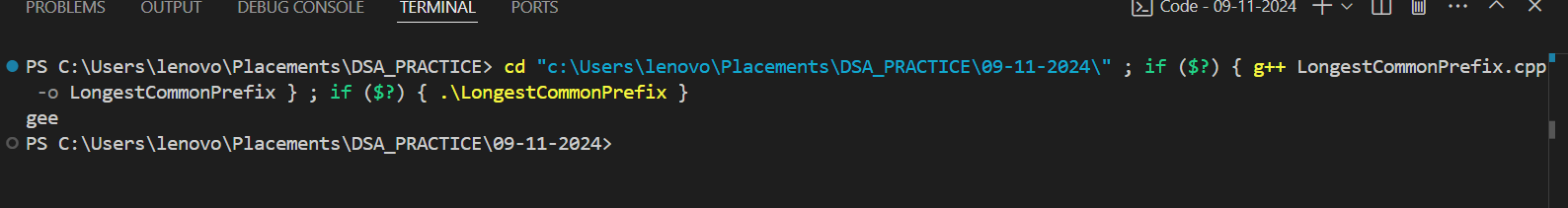
**string b=longestPrefix(s);**

**cout<<b;**

**return 0;**

**}**

OUTPUT:



TIME COMPLEXITY: O(n\*m\*log(n))  
SPACE COMPLEXITY: O(n\*m)

**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]

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

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

CODE:

#include <iostream>

#include <stack>

using namespace std;

void deleteMid(stack<char>& st) {

int n = st.size();

stack<char> tempSt;

int count = 0;

while (count < n / 2) {

char c = st.top();

st.pop();

tempSt.push(c);

count++;

}

st.pop();

while (!tempSt.empty()) {

st.push(tempSt.top());

tempSt.pop();

}

}

int main() {

stack<char> st;

st.push('1');

st.push('2');

st.push('3');

st.push('4');

st.push('5');

deleteMid(st);

while (!st.empty()) {

char p = st.top();

st.pop();

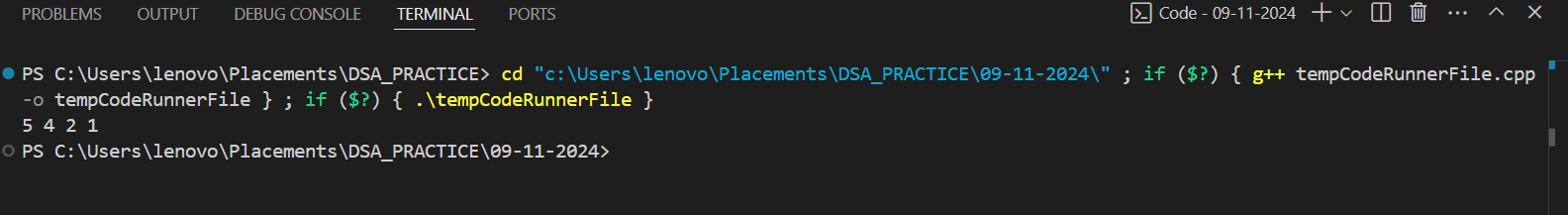
cout << p << " ";

}

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(n)

SPACE 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 2 –> 5 –> 25 –> 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 –> 7 -1 –> 12 6 12 –> 12 –> -1

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

CODE:

#include <iostream>

#include <stack>

using namespace std;

void printNGE(int arr[], int n) {

stack<int> s;

s.push(arr[0]);

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

if (s.empty()) {

s.push(arr[i]);

continue;

}

while (!s.empty() && s.top() < arr[i]) {

cout << s.top() << " > " << arr[i] << endl;

s.pop();

}

s.push(arr[i]);

}

while (!s.empty()) {

cout << s.top() << " > " << -1 << endl;

s.pop();

}

}

int main() {

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

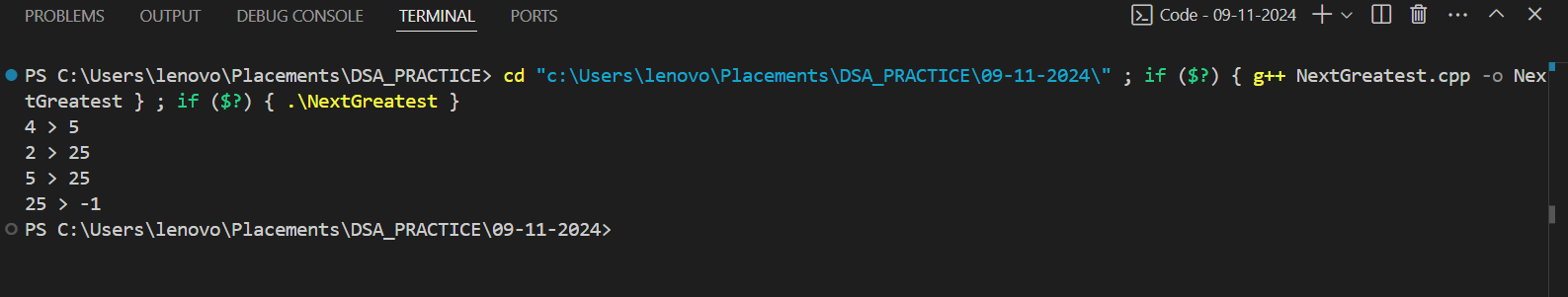
int n = sizeof(arr) / sizeof(arr[0]);

printNGE(arr, n);

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(n)  
SPACE COMPLEXITY: O(n)

**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:

#include <iostream>

#include <vector>

using namespace std;

class Node {

public:

int data;

Node \*left, \*right;

Node(int x) {

data = x;

left = right = nullptr;

}

};

void RecursiveRightView(Node\* root, int level, int& maxLevel, vector<int>& result) {

if (!root) return;

if (level > maxLevel) {

result.push\_back(root->data);

maxLevel = level;

}

RecursiveRightView(root->right, level + 1, maxLevel, result);

RecursiveRightView(root->left, level + 1, maxLevel, result);

}

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

vector<int> result;

int maxLevel = -1;

RecursiveRightView(root, 0, maxLevel, result);

return result;

}

void printArray(vector<int>& arr) {

for (int val : arr) {

cout << val << " ";

}

cout << endl;

}

int main() {

Node \*root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

root->right->left = new Node(4);

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

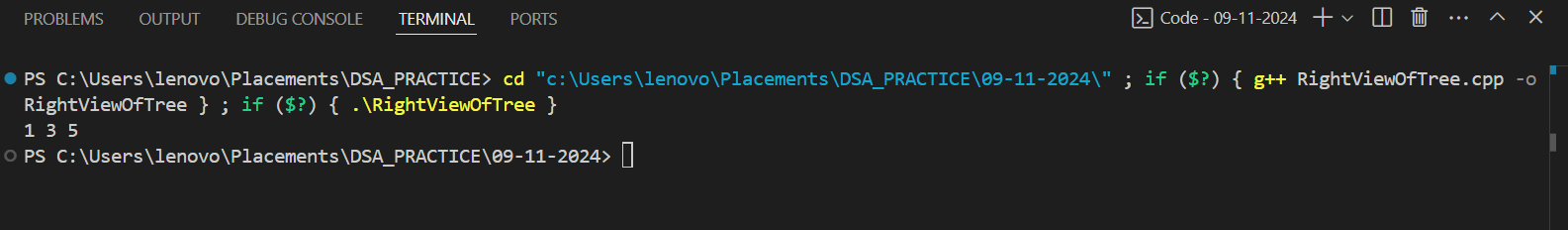
vector<int> result = rightView(root);

printArray(result);

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(h) h–>height of tree

**18. Maximum Depth or Height of Binary Tree**

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

CODE:

#include <iostream>

#include <algorithm>

using namespace std;

struct TreeNode {

int value;

TreeNode \*leftChild;

TreeNode \*rightChild;

TreeNode(int val) {

value = val;

leftChild = nullptr;

rightChild = nullptr;

}

};

int findMaxDepth(TreeNode \*node) {

if (node == nullptr)

return 0;

int leftDepth = findMaxDepth(node->leftChild);

int rightDepth = findMaxDepth(node->rightChild);

return max(leftDepth, rightDepth) + 1;

}

int main() {

TreeNode \*rootNode = new TreeNode(1);

rootNode->leftChild = new TreeNode(2);

rootNode->rightChild = new TreeNode(3);

rootNode->leftChild->leftChild = new TreeNode(4);

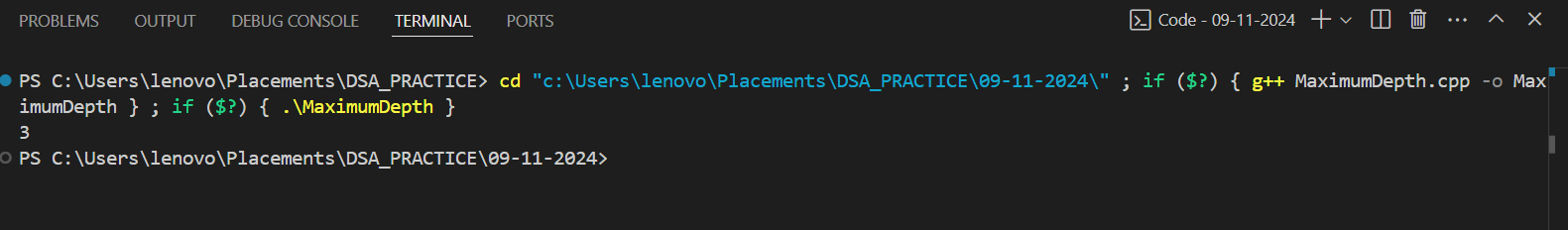
rootNode->leftChild->rightChild = new TreeNode(5);

cout << findMaxDepth(rootNode);

return 0;

}

OUTPUT:



TIME COMPLEXITY: O(n)

SPACE COMPLEXITY: O(h) h–>height of tree

.