**ASSIGNMENT 9 (Fast Learner)**

**AP LAB**

**Lavanya Saini – 22BCS15497**

**22BCS IOT 614-B**

1. **Set Matrix Zeroes   
   Problem Statement:** Given an m x n matrix, if an element is 0, set its entire row and column to 0. The modification must be done in place without using additional storage for another matrix.

**Code:**

#include <iostream>

#include <vector>

using namespace std;

void setZeroes(vector<vector<int>>& matrix) {

    int m = matrix.size(), n = matrix[0].size();

    bool firstRow = false, firstCol = false;

    for (int i = 0; i < m; i++)

        if (matrix[i][0] == 0) firstCol = true;

    for (int j = 0; j < n; j++)

        if (matrix[0][j] == 0) firstRow = true;

    for (int i = 1; i < m; i++)

        for (int j = 1; j < n; j++)

            if (matrix[i][j] == 0)

                matrix[i][0] = matrix[0][j] = 0;

    for (int i = 1; i < m; i++)

        for (int j = 1; j < n; j++)

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

                matrix[i][j] = 0;

    if (firstCol)

        for (int i = 0; i < m; i++) matrix[i][0] = 0;

    if (firstRow)

        for (int j = 0; j < n; j++) matrix[0][j] = 0;

}

void printMatrix(const vector<vector<int>>& matrix) {

    for (const auto& row : matrix) {

        for (int val : row)

            cout << val << " ";

        cout << endl;

    }

}

int main() {

    vector<vector<int>> matrix = {

        {1, 1, 1},

        {1, 0, 1},

        {1, 1, 1}

    };

    cout << "Original Matrix:\n";

    printMatrix(matrix);

    setZeroes(matrix);

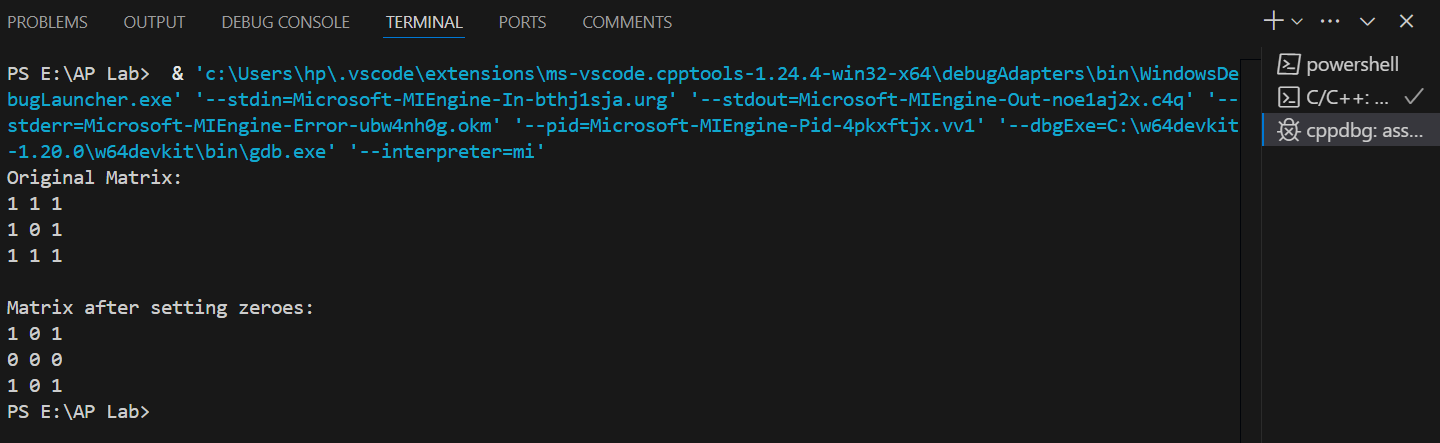
    cout << "\nMatrix after setting zeroes:\n";

    printMatrix(matrix);

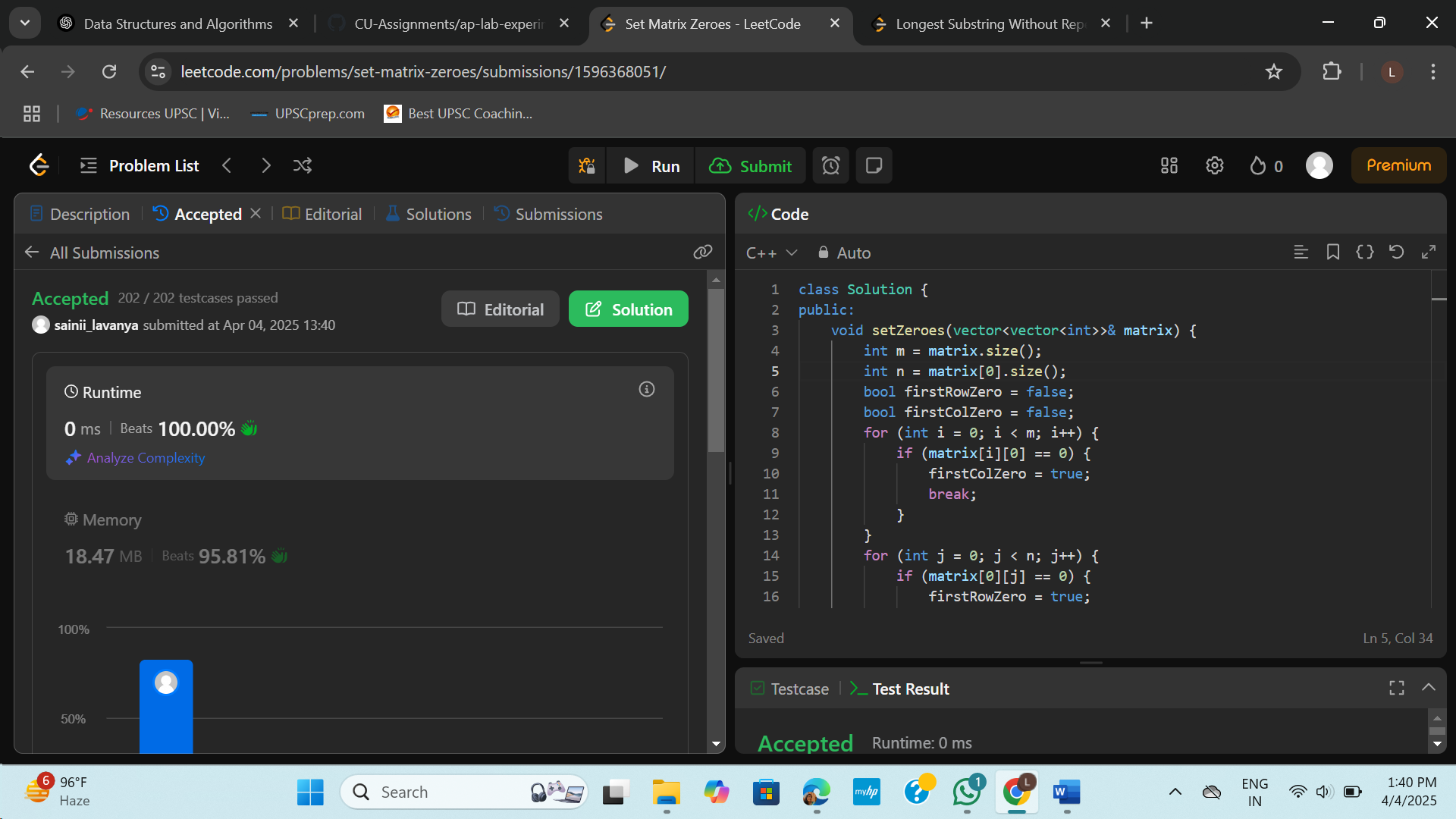
    return 0;

}

**Output:**

****

**Leetcode:-**



1. **Longest Substring Without Repeating Characters**

**Problem Statement:** Given a string s, find the length of the longest substring that does not contain any repeating characters.

**Code:**

#include <iostream>

#include <unordered\_map>

#include <string>

using namespace std;

int lengthOfLongestSubstring(string s) {

    unordered\_map<char, int> charIndex;

    int maxLen = 0, start = 0;

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

        if (charIndex.count(s[end]) && charIndex[s[end]] >= start)

            start = charIndex[s[end]] + 1;

        charIndex[s[end]] = end;

        maxLen = max(maxLen, end - start + 1);

    }

    return maxLen;

}

int main() {

    string s;

    cout << "Enter a string: ";

    cin >> s;

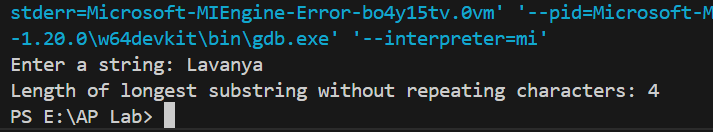
    cout << "Length of longest substring without repeating characters: "

         << lengthOfLongestSubstring(s) << endl;

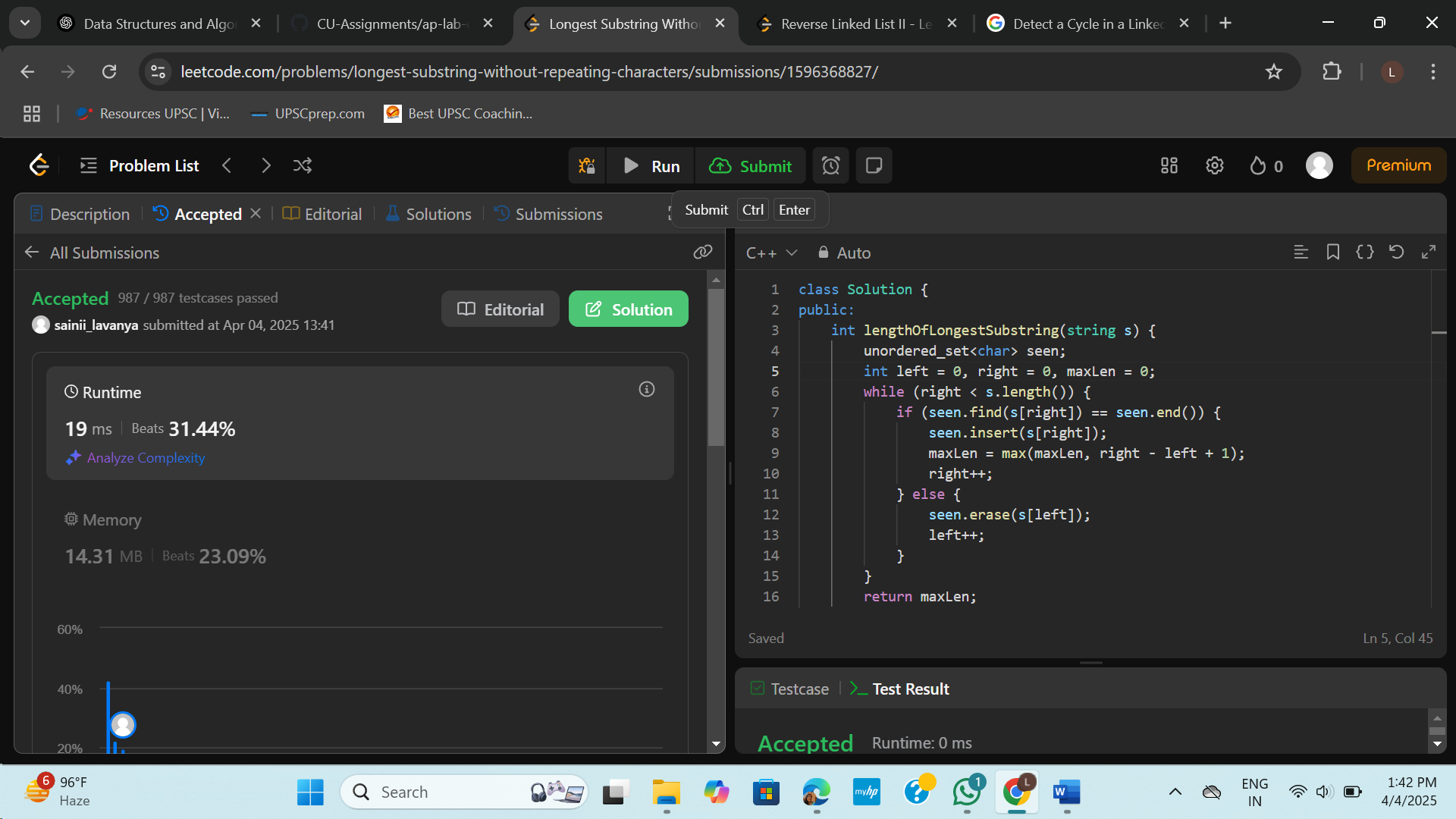
    return 0;

}

**Output:**



LeetCode:-



1. **Reverse Linked List II**

**Problem Statement:** Given the head of a singly linked list and two integers left and right, reverse the nodes of the list from position left to right, and return the modified list.

**Code:**

#include <iostream>

using namespace std;

struct ListNode {

    int val;

    ListNode\* next;

    ListNode(int x) : val(x), next(nullptr) {}

};

ListNode\* reverseBetween(ListNode\* head, int left, int right) {

    if (!head || left == right) return head;

    ListNode dummy(0);

    dummy.next = head;

    ListNode\* prev = &dummy;

    for (int i = 1; i < left; i++)

        prev = prev->next;

    ListNode\* curr = prev->next;

    for (int i = 0; i < right - left; i++) {

        ListNode\* temp = curr->next;

        curr->next = temp->next;

        temp->next = prev->next;

        prev->next = temp;

    }

    return dummy.next;

}

void printList(ListNode\* head) {

    while (head) {

        cout << head->val;

        if (head->next) cout << " -> ";

        head = head->next;

    }

    cout << "\n";

}

ListNode\* createList(int size) {

    if (size == 0) return nullptr;

    cout << "Enter " << size << " elements: ";

    int val;

    cin >> val;

    ListNode\* head = new ListNode(val);

    ListNode\* tail = head;

    for (int i = 1; i < size; i++) {

        cin >> val;

        tail->next = new ListNode(val);

        tail = tail->next;

    }

    return head;

}

int main() {

    int n, left, right;

    cout << "Enter the number of elements in the linked list: ";

    cin >> n;

    ListNode\* head = createList(n);

    cout << "Enter left and right positions to reverse: ";

    cin >> left >> right;

    cout << "Original List: ";

    printList(head);

    head = reverseBetween(head, left, right);

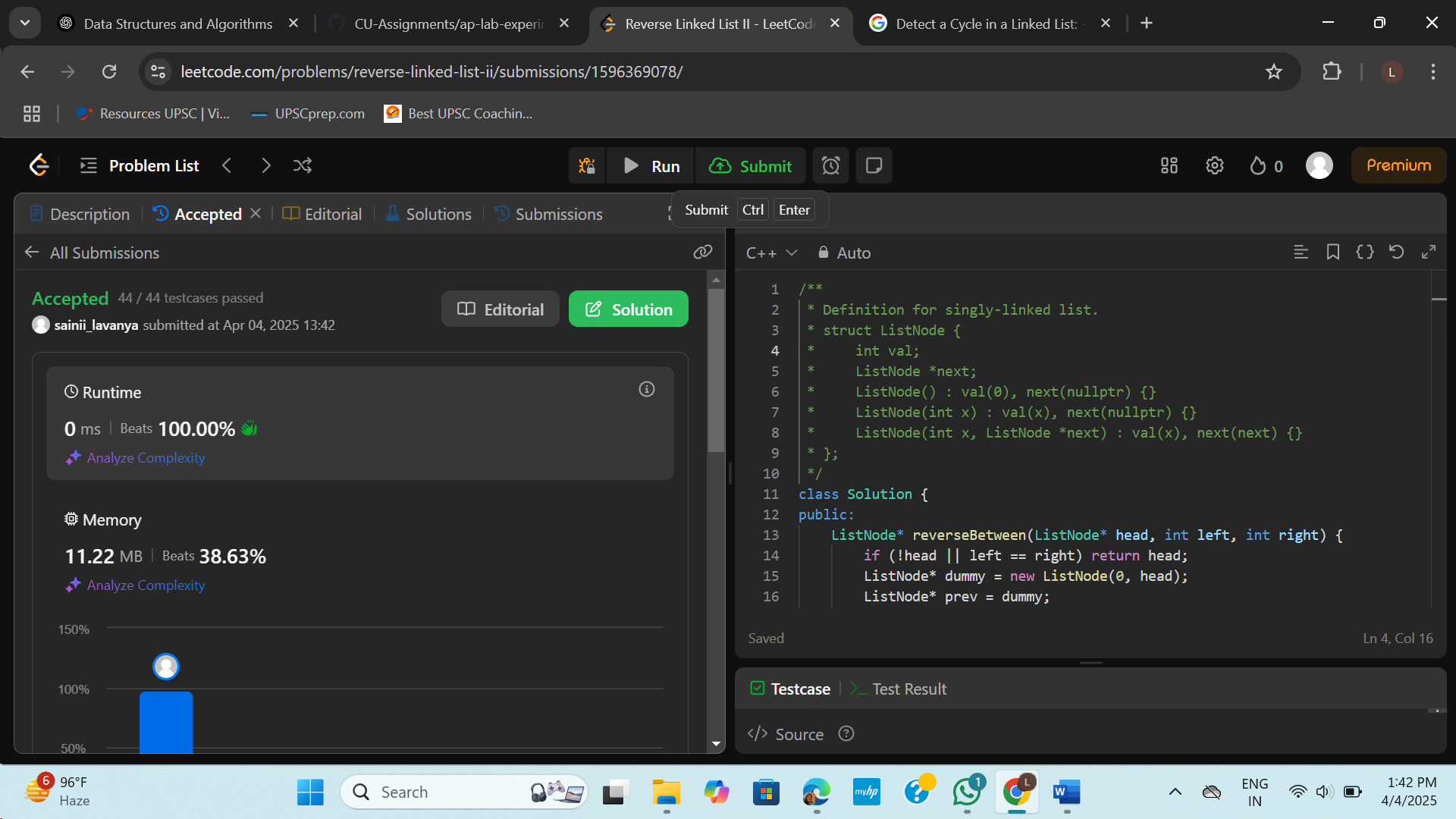
    cout << "Reversed List: ";

    printList(head);

    return 0;

}

LeetCode:-



1. **Detect a Cycle in a Linked List:** Given the head of a linked list, determine whether the linked list contains a cycle. A cycle occurs if a node's next pointer points to a previous node in the list.

class Solution {

public:

    bool hasCycle(ListNode \*head) {

        ListNode \*n1 = head, \*n2 = head;

        while (n2 && n2->next) {

            n1 = n1->next;

            n2 = n2->next->next;

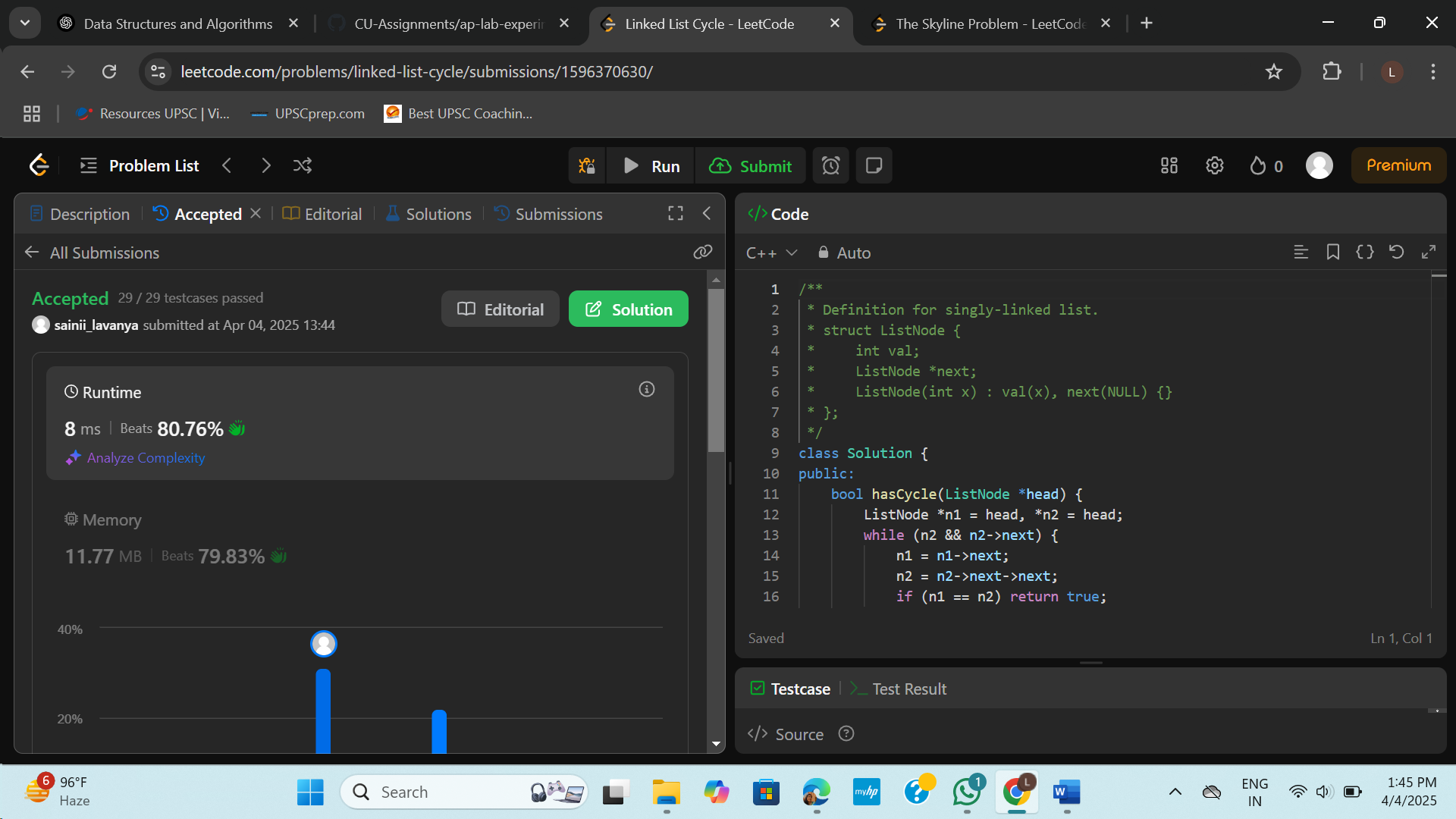
            if (n1 == n2) return true;

        }

        return false;

    }

};



1. **The Skyline Problem:** Given a list of buildings represented as [left, right, height], where each building is a rectangle, return the key points of the skyline. A key point is represented as [x, y], where x is the x coordinate where the height changes to y.

class Solution {

public:

    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {

        vector<pair<int, int>> events;

        for (auto& b : buildings) {

            events.push\_back({b[0], -b[2]});

            events.push\_back({b[1], b[2]});

        }

        sort(events.begin(), events.end(), [](auto& a, auto& b) {

            if (a.first != b.first) return a.first < b.first;

            return a.second < b.second;

        });

        multiset<int> heights = {0};

        int prevMax = 0;

        vector<vector<int>> result;

        for (auto& [x, h] : events) {

            if (h < 0) {

                heights.insert(-h);

            } else {

                heights.erase(heights.find(h));

            }

            int currMax = \*heights.rbegin();

            if (currMax != prevMax) {

                result.push\_back({x, currMax});

                prevMax = currMax;

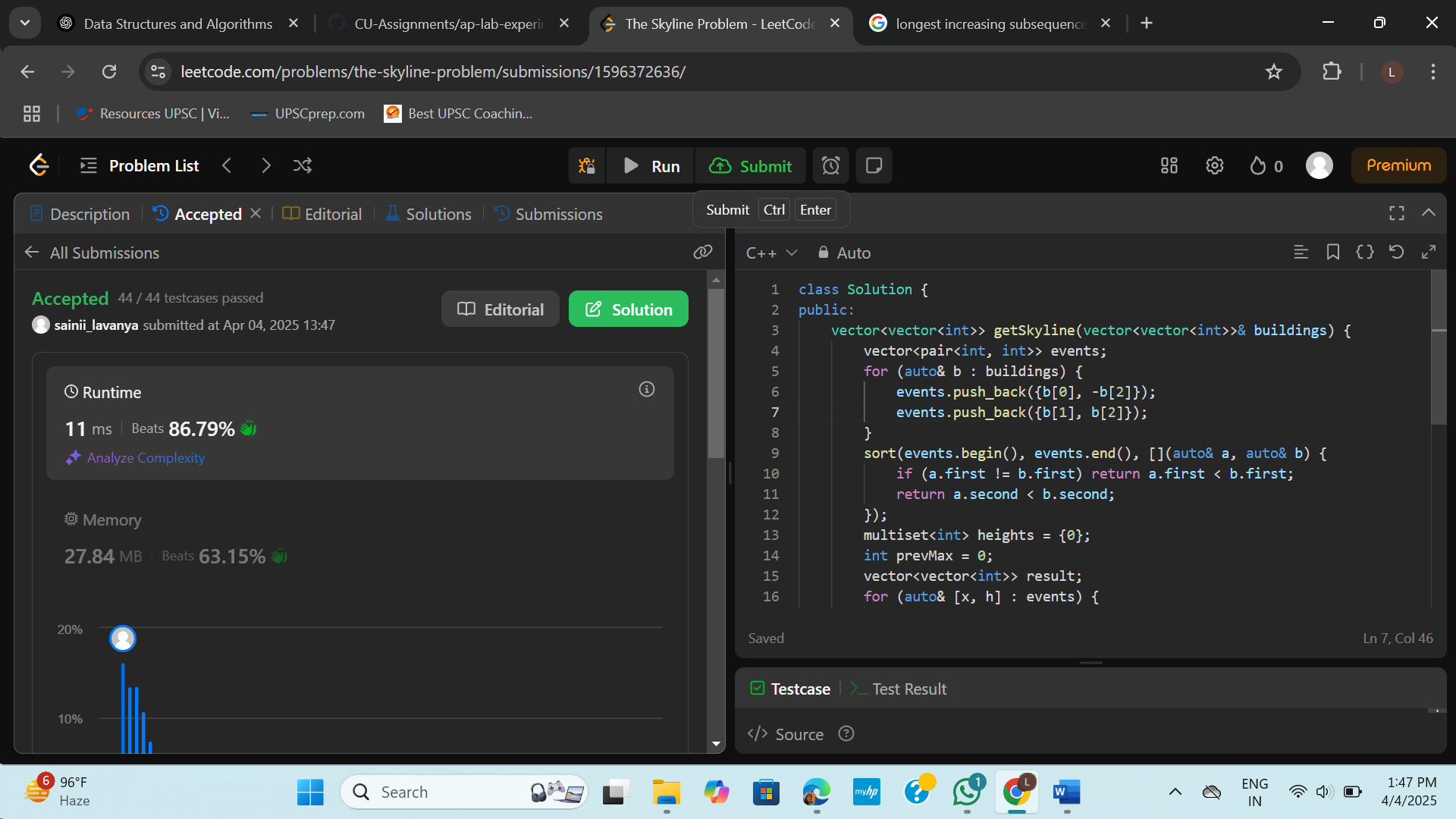
            }

        }

        return result;

    }

};



1. **Longest Increasing Subsequence II:** Given an integer array nums, find the length of the longest strictly increasing subsequence. A subsequence is derived from the array by deleting some or no elements without changing the order of the remaining elements.

class SegmentTree {

    vector<int> tree;

    int size;

public:

    SegmentTree(int n) {

        size = n;

        tree.resize(4 \* n, 0);

    }

    int query(int l, int r, int start, int end, int node) {

        if (r < start || l > end) return 0;

        if (l <= start && end <= r) return tree[node];

        int mid = (start + end) / 2;

        return max(

            query(l, r, start, mid, 2 \* node),

            query(l, r, mid + 1, end, 2 \* node + 1)

        );

    }

    void update(int index, int value, int start, int end, int node) {

        if (start == end) {

            tree[node] = max(tree[node], value);

            return;

        }

        int mid = (start + end) / 2;

        if (index <= mid)

            update(index, value, start, mid, 2 \* node);

        else

            update(index, value, mid + 1, end, 2 \* node + 1);

        tree[node] = max(tree[2 \* node], tree[2 \* node + 1]);

    }

    int query(int l, int r) {

        return query(l, r, 1, size, 1);

    }

    void update(int index, int value) {

        update(index, value, 1, size, 1);

    }

};

class Solution {

public:

    int lengthOfLIS(vector<int>& nums, int k) {

        int maxVal = \*max\_element(nums.begin(), nums.end());

        SegmentTree seg(maxVal + 2);

        int res = 0;

        for (int num : nums) {

            int maxLen = seg.query(max(1, num - k), num - 1);

            seg.update(num, maxLen + 1);

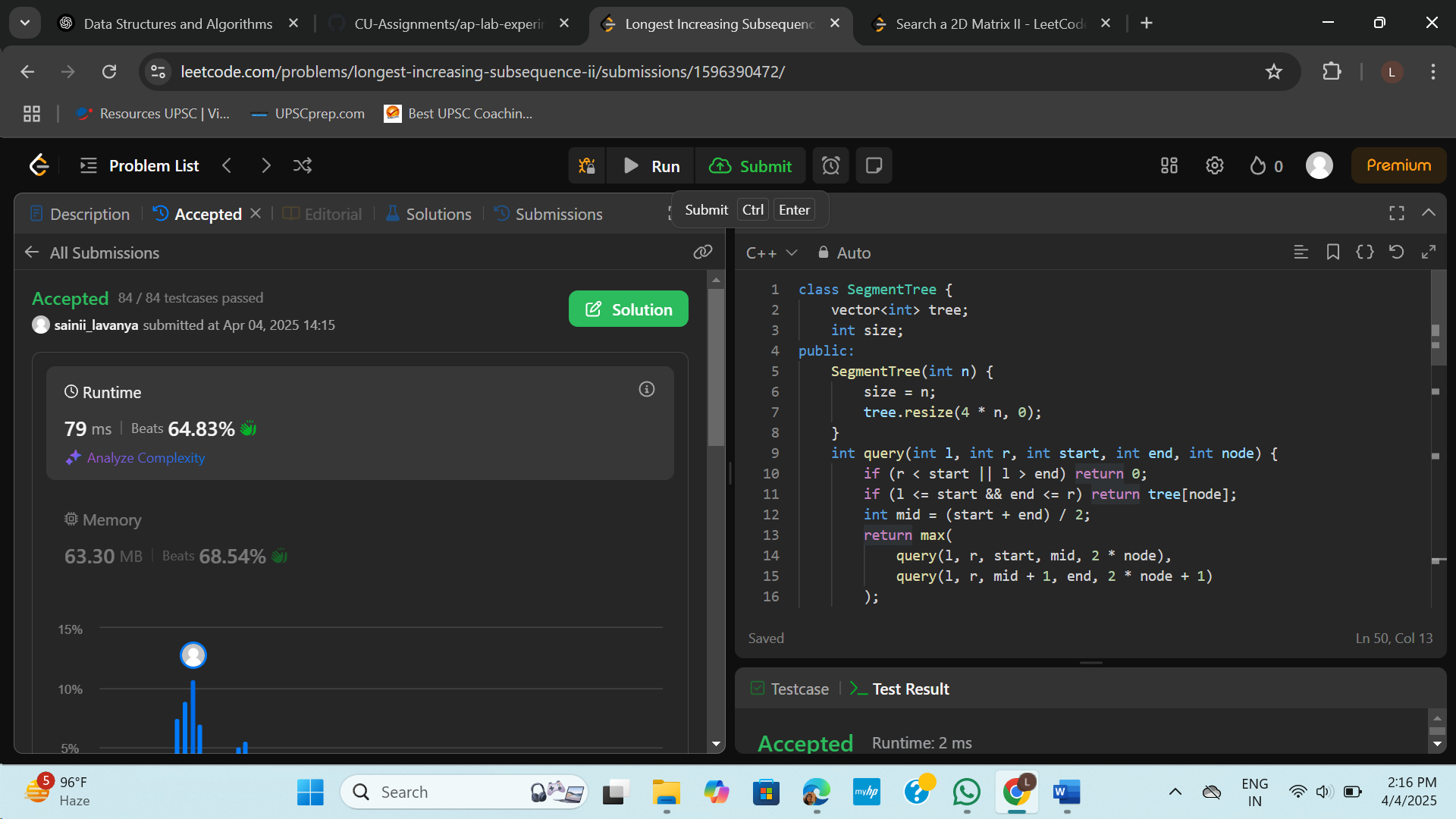
            res = max(res, maxLen + 1);

        }

        return res;

    }

};



1. **Search a 2D Matrix II:** Given an m x n matrix where each row is sorted in ascending order from left to right and each column is sorted in ascending order from top to bottom, and an integer target, determine if the target exists in the matrix.

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

int m = matrix.size();

int n = matrix[0].size();

int row = 0;

int col = n - 1;

while (row < m && col >= 0) {

if (matrix[row][col] == target) {

return true;

} else if (matrix[row][col] > target) {

col--;

} else {

row++;

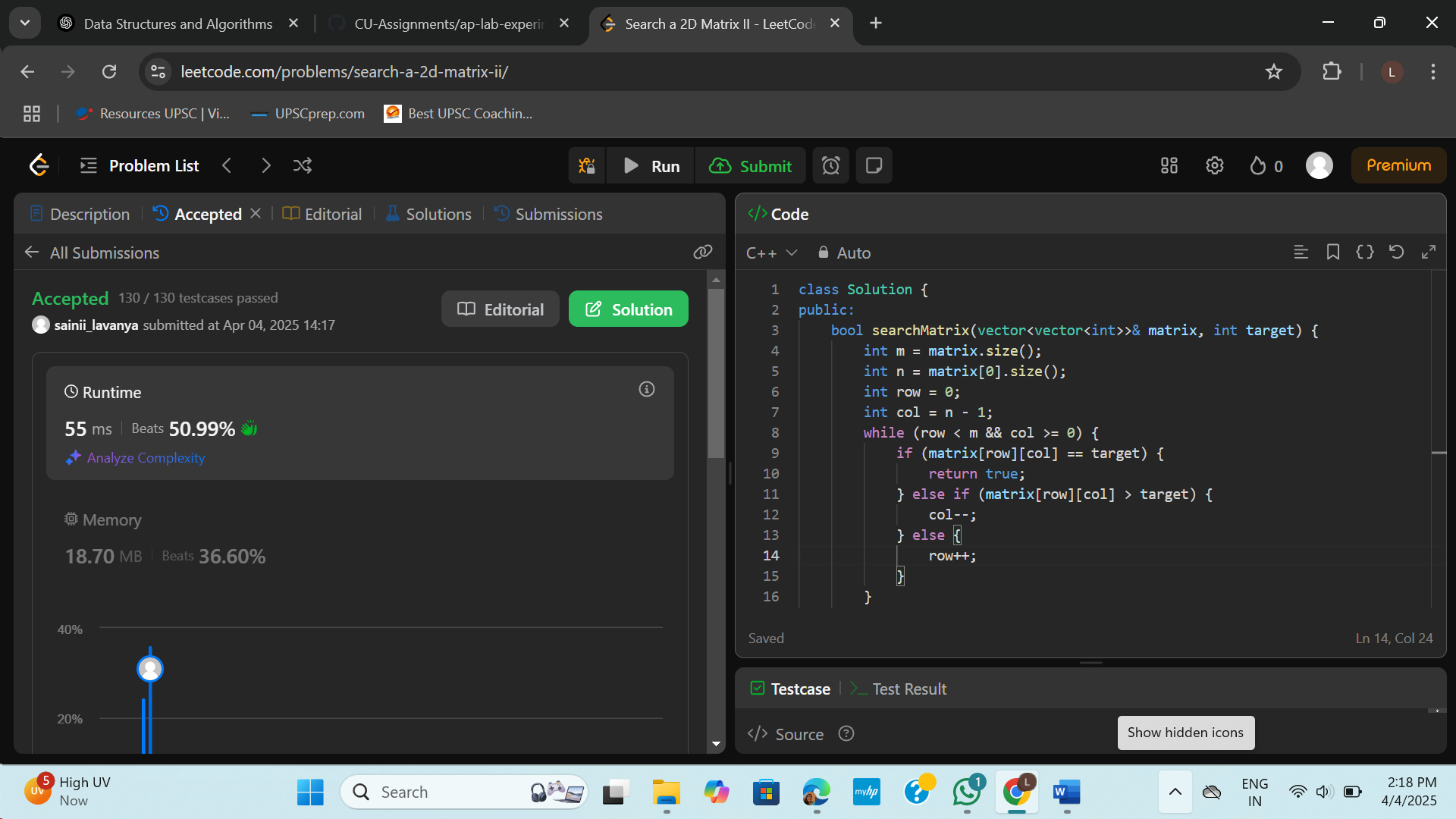
}

}

return false;

}

};



1. **Word Break:** Given a string s and a dictionary wordDict containing a list of words, determine if s can be segmented into a space-separated sequence of one or more dictionary words. The same word can be reused multiple times.

class Solution {

public:

bool wordBreak(string s, vector<string>& wordDict) {

unordered\_set<string> dict(wordDict.begin(), wordDict.end());

int n = s.length();

vector<bool> dp(n + 1, false);

dp[0] = true;

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

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

if (dp[j] && dict.find(s.substr(j, i - j)) != dict.end()) {

dp[i] = true;

break;

}

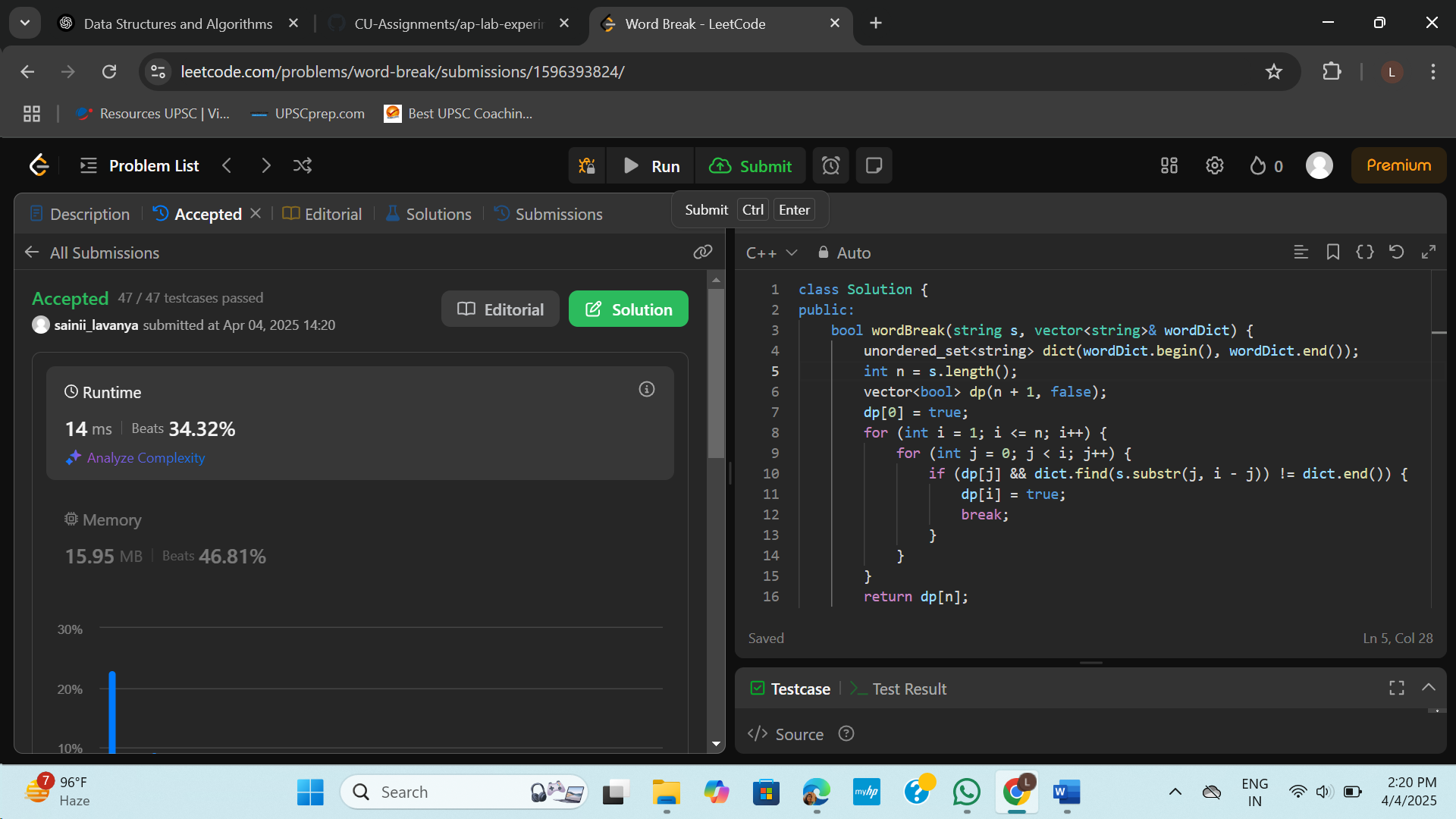
}

}

return dp[n];

}

};



1. **Longest Increasing Path in a Matrix:** Given an m x n integer matrix, find the length of the longest strictly increasing path. You can move up, down, left, or right from each cell. Diagonal moves and moves outside the boundaries are not allowed.

class Solution {

public:

int longestIncreasingPath(vector<vector<int>>& matrix) {

if (matrix.empty()) return 0;

int m = matrix.size();

int n = matrix[0].size();

vector<vector<int>> dp(m, vector<int>(n, 0));

int maxLen = 0;

vector<int> dirs = {0, 1, 0, -1, 0};

function<int(int, int)> dfs = [&](int i, int j) -> int {

if (dp[i][j]) return dp[i][j];

int maxPath = 1;

for (int d = 0; d < 4; ++d) {

int ni = i + dirs[d];

int nj = j + dirs[d + 1];

if (ni >= 0 && ni < m && nj >= 0 && nj < n && matrix[ni][nj] > matrix[i][j]) {

maxPath = max(maxPath, 1 + dfs(ni, nj));

}

}

return dp[i][j] = maxPath;

};

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

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

maxLen = max(maxLen, dfs(i, j));

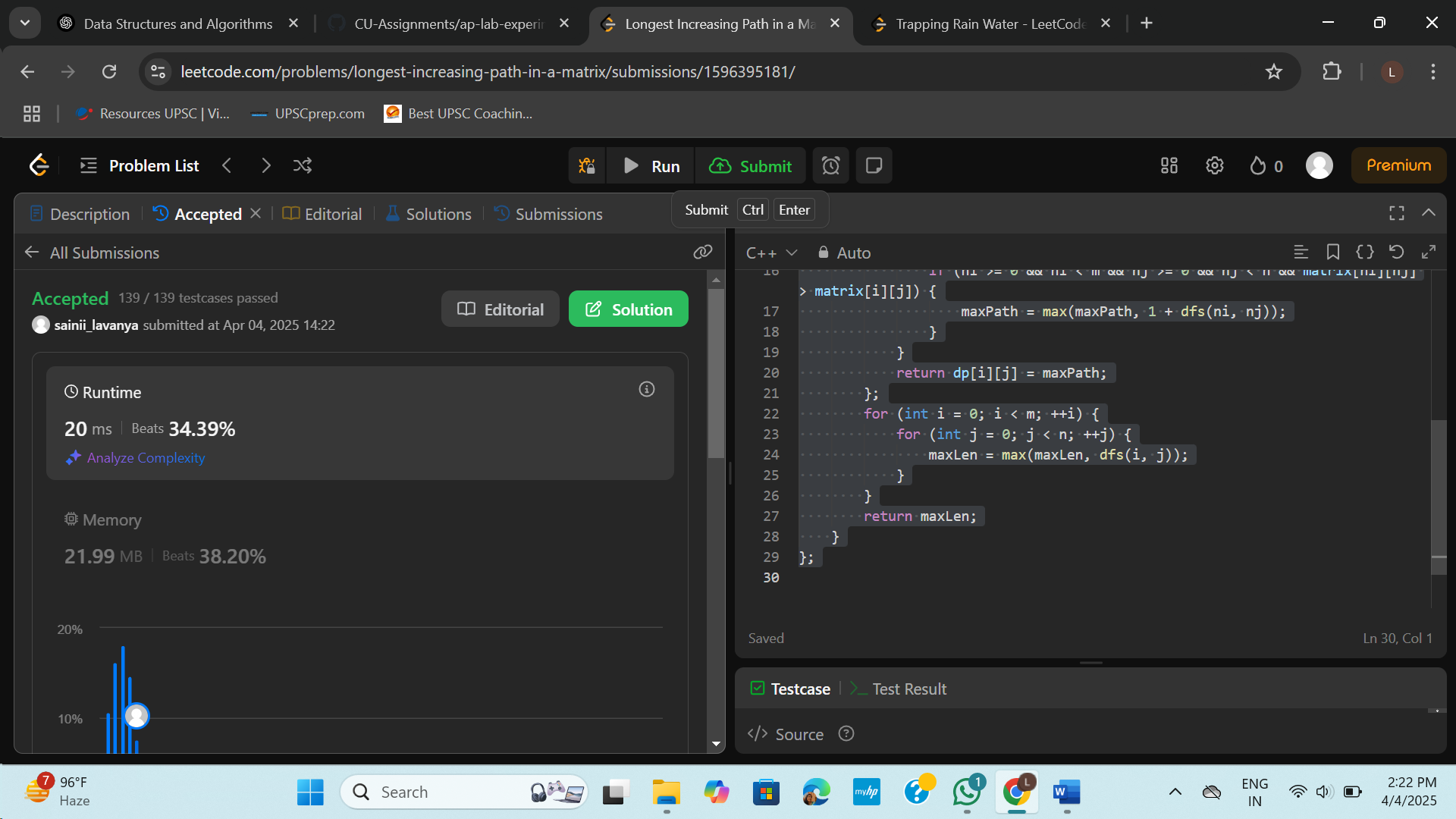
}

}

return maxLen;

}

};



1. **Trapping Rain Water:** Given n non-negative integers representing an elevation map where the width of each bar is 1, compute the total amount of water that can be trapped after raining.

class Solution {

public:

int trap(vector<int>& height) {

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

int leftMax = 0, rightMax = 0;

int water = 0;

while (left < right) {

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

if (height[left] >= leftMax)

leftMax = height[left];

else

water += leftMax - height[left];

left++;

} else {

if (height[right] >= rightMax)

rightMax = height[right];

else

water += rightMax - height[right];

right--;

}

}

return water;

}

};

