**ASSIGNMENT 9 (Fast Learner)**

**Student Name: Ramit Chaturvedi UID:22BCS15597**

**Branch: CSE Section/Group:614-B**

**Semester: 6 Date of Performance:10-04-25**

**Subject Name: AP LAB-II Subject Code: 22CSP-351**

Q 1) [**Set Matrix Zeroes**](https://leetcode.com/problems/set-matrix-zeroes/)

Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's.

 Code)

class Solution {

 public:

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

    const int m = matrix.size();

    const int n = matrix[0].size();

    bool shouldFillFirstRow = false;

    bool shouldFillFirstCol = false;

    for (int j = 0; j < n; ++j)

      if (matrix[0][j] == 0) {

        shouldFillFirstRow = true;

        break;

      }

    for (int i = 0; i < m; ++i)

      if (matrix[i][0] == 0) {

        shouldFillFirstCol = true;

        break;

      }

    // Store the information in the first row and the first column.

    for (int i = 1; i < m; ++i)

      for (int j = 1; j < n; ++j)

        if (matrix[i][j] == 0) {

          matrix[i][0] = 0;

          matrix[0][j] = 0;

        }

    // Fill 0s for the matrix except the first row and the first column.

    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;

    // Fill 0s for the first row if needed.

    if (shouldFillFirstRow)

      for (int j = 0; j < n; ++j)

        matrix[0][j] = 0;

    // Fill 0s for the first column if needed.

    if (shouldFillFirstCol)

      for (int i = 0; i < m; ++i)

        matrix[i][0] = 0;

  }

};

Output)



Q 2) [**Longest Substring Without Repeating Characters**](https://leetcode.com/problems/longest-substring-without-repeating-characters/)

Given a string s, find the length of the **longest** **substring** without duplicate characters.

Code)

class Solution {

public:

int lengthOfLongestSubstring(string s) {

int ans = 0;

vector<int> count(128);

for (int l = 0, r = 0; r < s.length(); ++r) {

++count[s[r]];

while (count[s[r]] > 1)

--count[s[l++]];

ans = max(ans, r - l + 1);

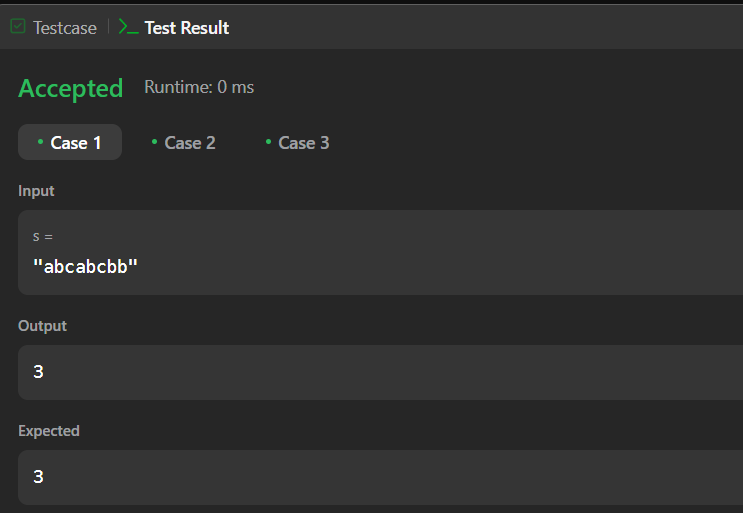
}

return ans;

}

};

Output)



3) [**Reverse Linked List II**](https://leetcode.com/problems/reverse-linked-list-ii/)

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return *the reversed list*.

Return *the****minimum****possible total number of stones remaining after applying the*k*operations*.

floor(x) is the **greatest** integer that is **smaller** than or **equal** to x (i.e., rounds x down).

Code)

class Solution {

public:

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

if (left == 1)

return reverseN(head, right);

head->next = reverseBetween(head->next, left - 1, right - 1);

return head;

}

private:

ListNode\* reverseN(ListNode\* head, int n) {

if (n == 1)

return head;

ListNode\* newHead = reverseN(head->next, n - 1);

ListNode\* headNext = head->next;

head->next = headNext->next;

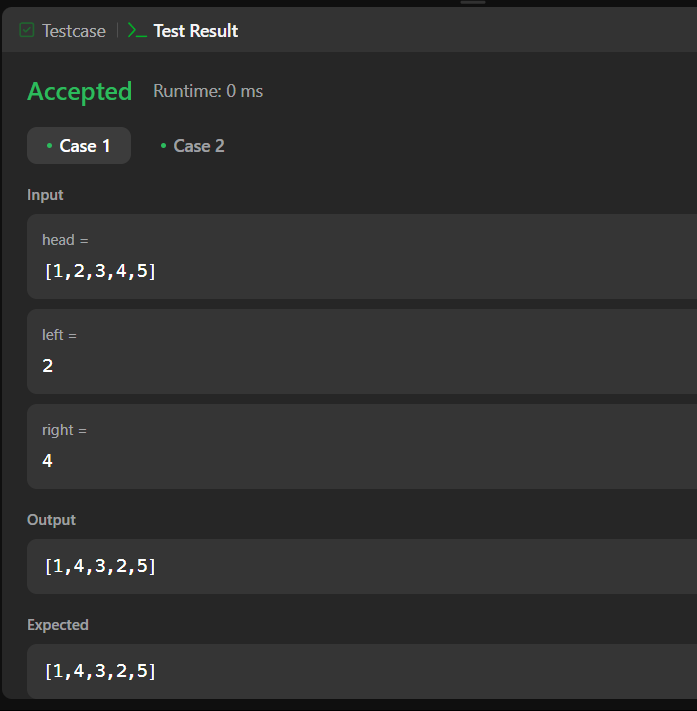
headNext->next = head;

return newHead;

}

};

Output)



4) [**Linked List Cycle**](https://leetcode.com/problems/linked-list-cycle/)

Given head, the head of a linked list, determine if the linked list has a cycle in it.

Code)

/\*\*

 \* Definition for singly-linked list.

 \* struct ListNode {

 \*     int val;

 \*     ListNode \*next;

 \*     ListNode(int x) : val(x), next(NULL) {}

 \* };

 \*/

class Solution {

public:

    bool hasCycle(ListNode\* head) {

        if (head == NULL || head->next == NULL) {

            return false;

        }

        ListNode\* slow = head;

        ListNode\* fast = head->next;

        while (fast != slow) {

            if (fast->next == NULL || fast->next->next == NULL) {

                return false;

            }

            slow = slow->next;

            fast = fast->next->next;

        }

        return true;

    }

};

Output)



5) [**The Skyline Problem**](https://leetcode.com/problems/the-skyline-problem/)

A city's **skyline** is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return *the****skyline****formed by these buildings collectively*.

Code)

class Solution {

public:

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

const int n = buildings.size();

if (n == 0)

return {};

if (n == 1) {

const int left = buildings[0][0];

const int right = buildings[0][1];

const int height = buildings[0][2];

return {{left, height}, {right, 0}};

}

const vector<vector<int>> left =

getSkyline({buildings.begin(), buildings.begin() + n / 2});

const vector<vector<int>> right =

getSkyline({buildings.begin() + n / 2, buildings.end()});

return merge(left, right);

}

private:

vector<vector<int>> merge(const vector<vector<int>>& left,

const vector<vector<int>>& right) {

vector<vector<int>> ans;

int i = 0; // left's index

int j = 0; // right's index

int leftY = 0;

int rightY = 0;

while (i < left.size() && j < right.size())

// Choose the point with the smaller x.

if (left[i][0] < right[j][0]) {

leftY = left[i][1]; // Update the ongoing `leftY`.

addPoint(ans, left[i][0], max(left[i++][1], rightY));

} else {

rightY = right[j][1]; // Update the ongoing `rightY`.

addPoint(ans, right[j][0], max(right[j++][1], leftY));

}

while (i < left.size())

addPoint(ans, left[i][0], left[i++][1]);

while (j < right.size())

addPoint(ans, right[j][0], right[j++][1]);

return ans;

}

void addPoint(vector<vector<int>>& ans, int x, int y) {

if (!ans.empty() && ans.back()[0] == x) {

ans.back()[1] = y;

return;

}

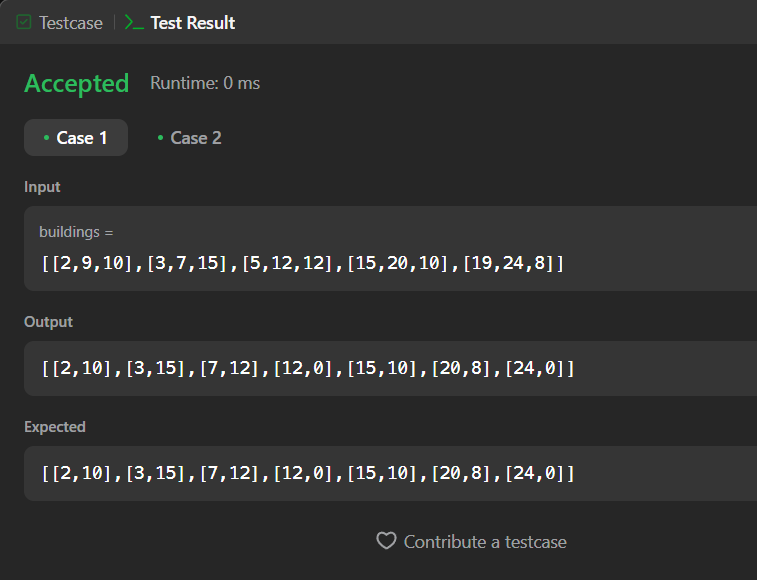
if (!ans.empty() && ans.back()[1] == y)

return;

ans.push\_back({x, y});

}

};Output)



6) 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

Code)

struct SegmentTreeNode {

int lo;

int hi;

int maxLength;

std::unique\_ptr<SegmentTreeNode> left;

std::unique\_ptr<SegmentTreeNode> right;

// maxLength := the maximum length of LIS ending in [lo..hi]

SegmentTreeNode(int lo, int hi, int maxLength,

std::unique\_ptr<SegmentTreeNode> left = nullptr,

std::unique\_ptr<SegmentTreeNode> right = nullptr)

: lo(lo),

hi(hi),

maxLength(maxLength),

left(std::move(left)),

right(std::move(right)) {}

};

class SegmentTree {

public:

explicit SegmentTree() : root(make\_unique<SegmentTreeNode>(0, 1e5 + 1, 0)) {}

void updateRange(int i, int j, int maxLength) {

update(root, i, j, maxLength);

}

// Returns the maximum length of LIS ending in [i..j].

int queryRange(int i, int j) {

return query(root, i, j);

}

private:

std::unique\_ptr<SegmentTreeNode> root;

void update(std::unique\_ptr<SegmentTreeNode>& root, int i, int j,

int maxLength) {

if (root->lo == i && root->hi == j) {

root->maxLength = maxLength;

root->left = nullptr;

root->right = nullptr;

return;

}

const int mid = root->lo + (root->hi - root->lo) / 2;

if (root->left == nullptr) {

root->left = make\_unique<SegmentTreeNode>(root->lo, mid, root->maxLength);

root->right =

make\_unique<SegmentTreeNode>(mid + 1, root->hi, root->maxLength);

}

if (j <= mid)

update(root->left, i, j, maxLength);

else if (i > mid)

update(root->right, i, j, maxLength);

else {

update(root->left, i, mid, maxLength);

update(root->right, mid + 1, j, maxLength);

}

root->maxLength = merge(root->left->maxLength, root->right->maxLength);

}

int query(std::unique\_ptr<SegmentTreeNode>& root, int i, int j) {

if (root->left == nullptr)

return root->maxLength;

if (root->lo == i && root->hi == j)

return root->maxLength;

const int mid = root->lo + (root->hi - root->lo) / 2;

if (j <= mid)

return query(root->left, i, j);

if (i > mid)

return query(root->right, i, j);

return merge(query(root->left, i, mid), query(root->right, mid + 1, j));

}

int merge(int left, int right) const {

return max(left, right);

};

};

class Solution {

public:

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

int ans = 1;

SegmentTree tree;

for (const int num : nums) {

const int left = max(1, num - k);

const int right = num - 1;

// the maximum length of LIS ending in [left..right] + the current number

const int maxLength = tree.queryRange(left, right) + 1;

ans = max(ans, maxLength);

tree.updateRange(num, num, maxLength);

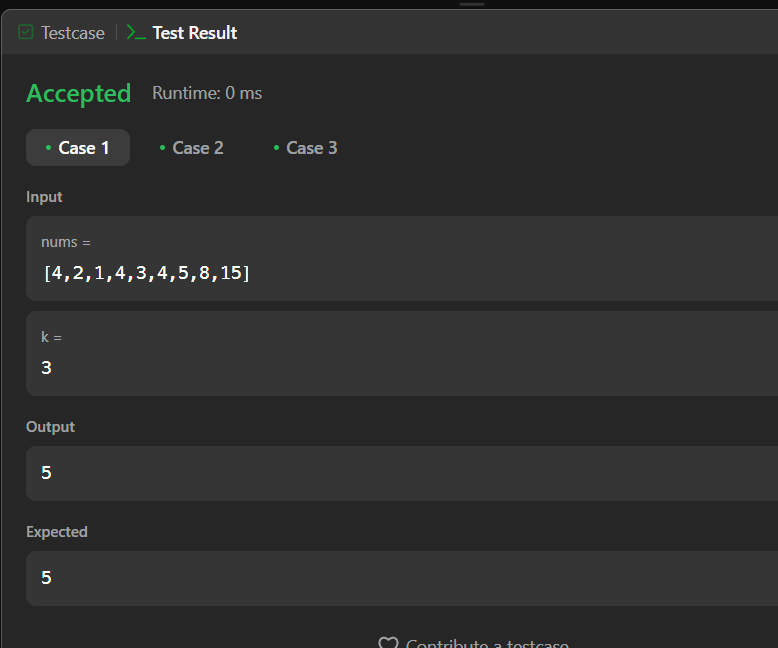
}

return ans;

}

};

Output)



7) 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.

Code)

class Solution {

public:

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

int r = 0;

int c = matrix[0].size() - 1;

while (r < matrix.size() && c >= 0) {

if (matrix[r][c] == target)

return true;

if (matrix[r][c] > target)

--c;

else

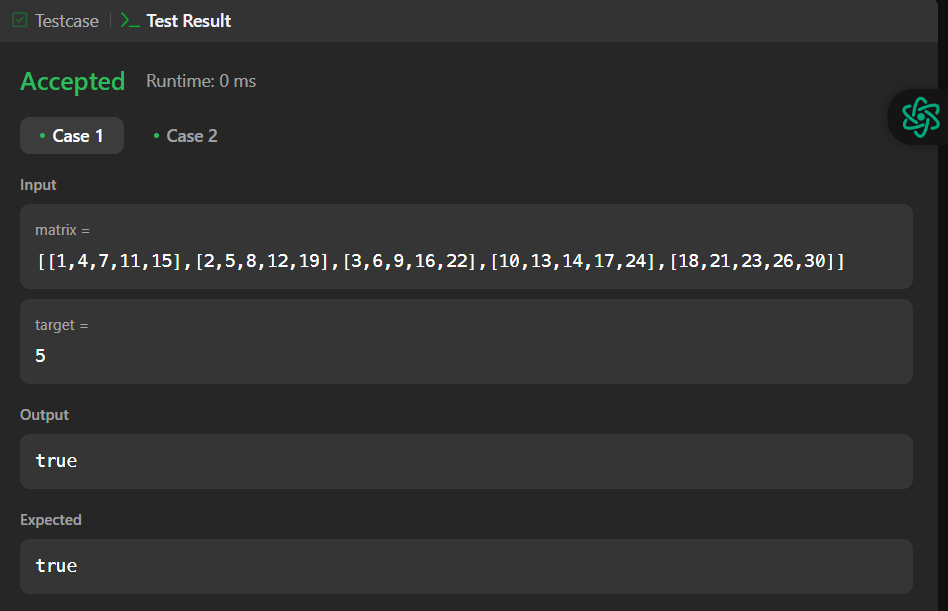
++r;

}

return false;

}

};



8) 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.

Code)

class Solution {

public:

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

return wordBreak(s, {wordDict.begin(), wordDict.end()}, {});

}

private:

bool wordBreak(const string& s, const unordered\_set<string>&& wordSet,

unordered\_map<string, bool>&& mem) {

if (wordSet.contains(s))

return true;

if (const auto it = mem.find(s); it != mem.cend())

return it->second;

// 1 <= prefix.length() < s.length()

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

const string& prefix = s.substr(0, i);

const string& suffix = s.substr(i);

if (wordSet.contains(prefix) &&

wordBreak(suffix, std::move(wordSet), std::move(mem)))

return mem[s] = true;

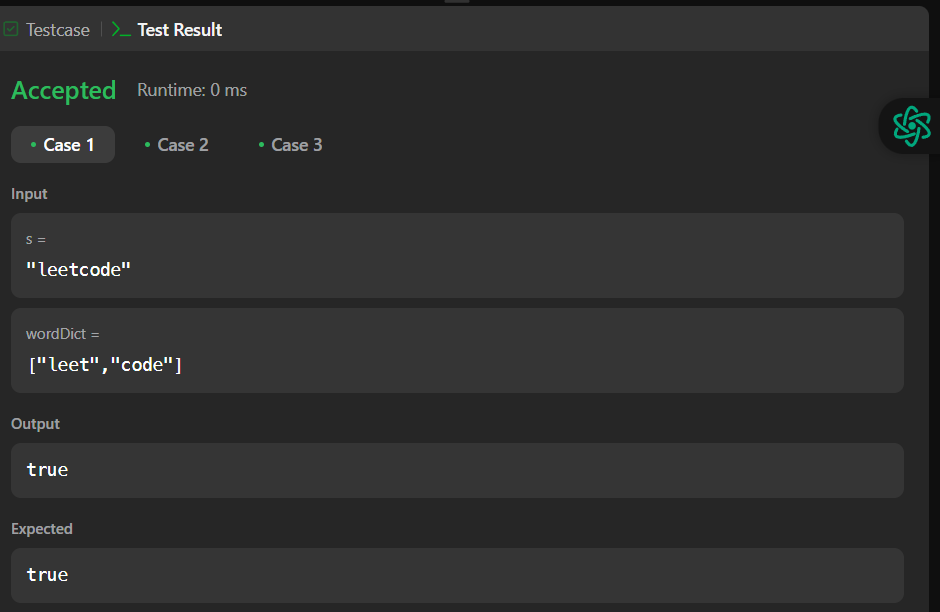
}

return mem[s] = false;

}

};

Output)



9) 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.

Code)

class Solution {

public:

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

const int m = matrix.size();

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

int ans = 0;

vector<vector<int>> mem(m, vector<int>(n));

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

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

ans = max(ans, dfs(matrix, i, j, INT\_MIN, mem));

return ans;

}

private:

// mem[i][j] := the LIP starting from matrix[i][j]

int dfs(const vector<vector<int>>& matrix, int i, int j, int prev,

vector<vector<int>>& mem) {

if (i < 0 || i == matrix.size() || j < 0 || j == matrix[0].size())

return 0;

if (matrix[i][j] <= prev)

return 0;

int& ans = mem[i][j];

if (ans > 0)

return ans;

const int curr = matrix[i][j];

return ans = 1 + max({dfs(matrix, i + 1, j, curr, mem),

dfs(matrix, i - 1, j, curr, mem),

dfs(matrix, i, j + 1, curr, mem),

dfs(matrix, i, j - 1, curr, mem)});

}

};

Output)

