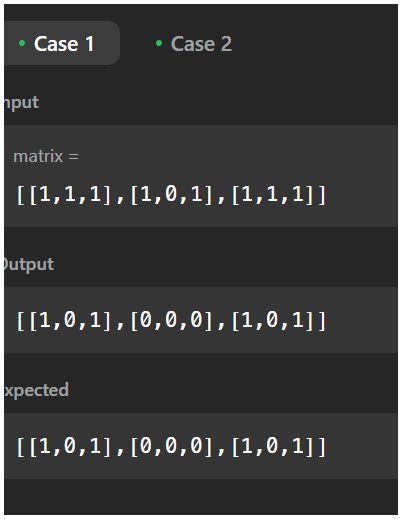
AP 9TH Assignment

Q1. **Set Matrix Zeroes**: Given an m x n matrix, if an element is 0, set its entire row and column to 0.

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;  }
    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;  }
    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 (shouldFillFirstRow)
      for (int j = 0; j < n; ++j)
        matrix[0][j] = 0;
    if (shouldFillFirstCol)
      for (int i = 0; i < m; ++i)
        matrix[i][0] = 0;  }   };
```

Output:



Q2. **Longest Substring Without Repeating Characters**: Given a string s, find the length of the longest substring that does not contain any repeating characters.

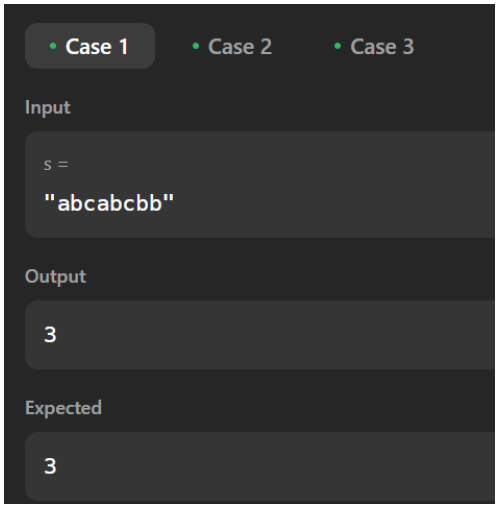
Code:

```
public:
    int lengthOfLongestSubstring(string s) {
        int n = s.length();
        int maxLength = 0;
        unordered_set<char> charSet;
        int left = 0;

        for (int right = 0; right < n; right++) {
            if (charSet.count(s[right]) == 0) {
                charSet.insert(s[right]);
                maxLength = max(maxLength, right - left + 1);
            } else {
                while (charSet.count(s[right])) {
                    charSet.erase(s[left]);
                    left++;
                }
                charSet.insert(s[right]);
            }
        }

        return maxLength;
    }
};
```

Output:

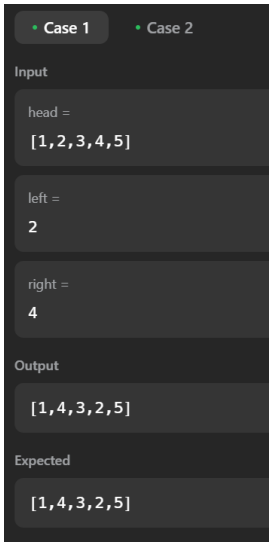


Q3. **Reverse Linked List II**: Given the head of a singly linked list and two integers left and right, reverse the nodes of the list from position left to right.

Code:

```cpp
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        if (!head || left == right) return head;
        ListNode* dummy = new ListNode(0);
        dummy->next = head;
        ListNode* prev = dummy;
        for (int i = 1; i < left; ++i) {
            prev = prev->next;
        }
        ListNode* curr = prev->next;
        ListNode* next = nullptr;
        for (int i = left; i < right; ++i) {
            next = curr->next;
            curr->next = next->next;
            next->next = prev->next;
            prev->next = next;
        }
        return dummy->next;
    }
};
```

Output:

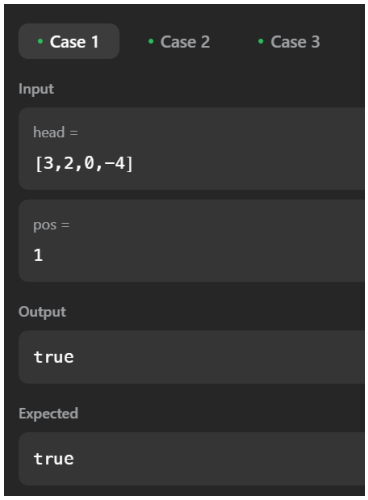


Q4. **Detect a Cycle in a Linked List**: Given the head of a linked list, determine whether the linked list contains a cycle. A cycle occurs if a node's next pointer points to a previous node in the list.

Code:

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* slow = head, *fast = head;
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) return true;
        }
        return false;
    }
};
```

Output:

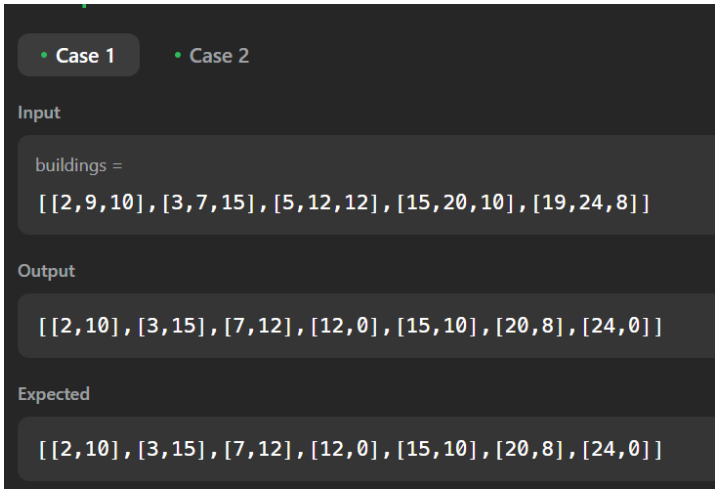


Q5. **The Skyline Problem**: Given a list of buildings represented as [left, right, height], where each building is a rectangle, return the key points of the skyline. A key point is represented as [x, y], where x is the xcoordinate where the height changes to y.

Code:

```
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());
    multiset<int> heights = {0};
    vector<vector<int>> result;
    int lastHeight = 0;
    for (auto& [x, h] : events) {
        if (h < 0) {
            heights.insert(-h);
        } else {
            heights.erase(heights.find(h));
        }
        int currHeight = *heights.rbegin();
        if (currHeight != lastHeight) {
            result.push_back({x, currHeight});
            lastHeight = currHeight;
        }
    }
    return result;
```

Output:

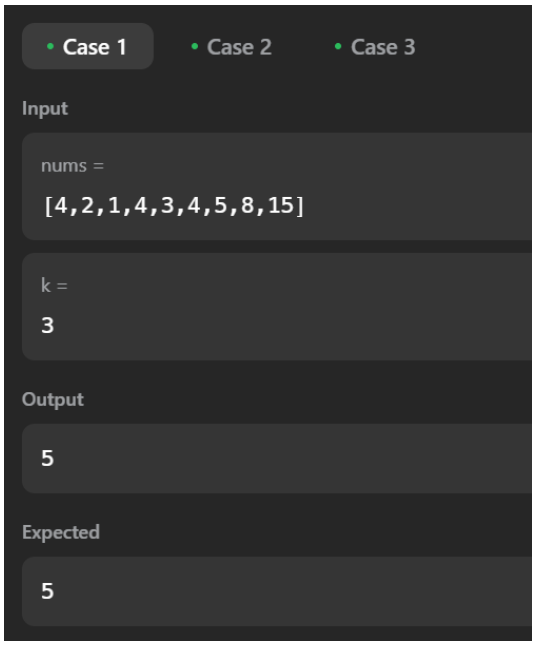


Q6. **Longest Increasing Subsequence II**: Given an integer array nums, find the length of the longest strictly increasing subsequence. A subsequence is derived from the array by deleting some or no elements without changing the order of the remaining elements.

Code:

```
public:
    vector<int> tree;
    int size;
    SegmentTree(int n) {
        size = n + 2;
        tree.resize(size * 4); }
    int query(int l, int r, int s = 1, int e = -1, int node = 1) {
        if (e == -1) e = size - 1;
        if (r < s || l > e) return 0;
        if (l <= s && e <= r) return tree[node];
        int m = (s + e) / 2;
        return max(query(l, r, s, m, node * 2), query(l, r, m + 1, e, node *
2 + 1));  }
    void update(int idx, int val, int s = 1, int e = -1, int node = 1) {
        if (e == -1) e = size - 1;
        if (s == e) {
            tree[node] = max(tree[node], val);
            return;  }
        int m = (s + e) / 2;
        if (idx <= m) update(idx, val, s, m, node * 2);
        else update(idx, val, m + 1, e, node * 2 + 1);
        tree[node] = max(tree[node * 2], tree[node * 2 + 1]); }
};
```

Output:

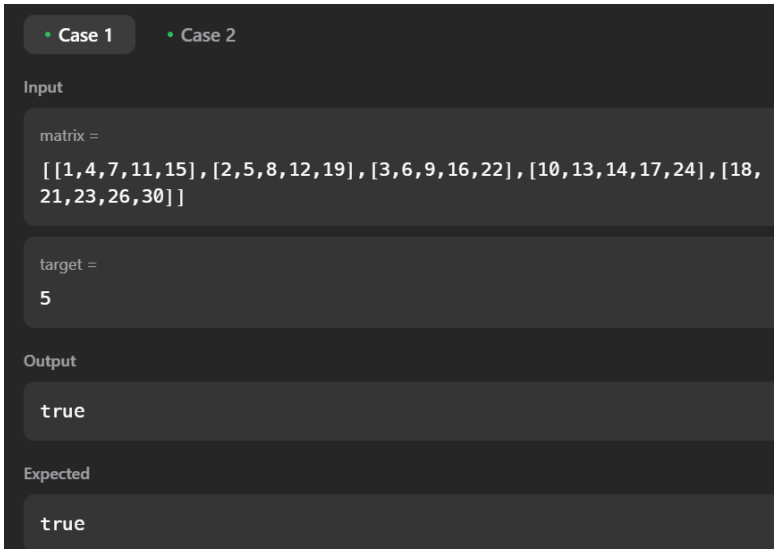


Q7. **Search a 2D Matrix II**: Given an m x n matrix where each row is sorted in ascending order from left to right and each column is sorted in ascending order from top to bottom, and an integer target, determine if the target exists in the matrix.

Code:

```cpp
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;
    }
};
```

Output:



Q8. **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:

```cpp
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> dict(wordDict.begin(), wordDict.end());
        int n = s.size();
        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.count(s.substr(j, i - j))) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[n];
    }
};
```

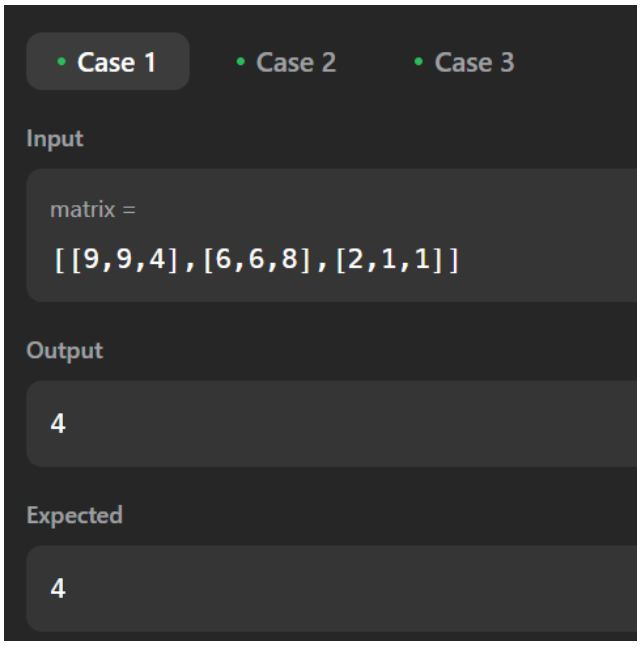
Output:

Q9. **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 go(vector<vector<int>>& mat, int i, int j, vector<vector<int>>& dp) {
        if (dp[i][j] != 0) return dp[i][j];
        int m = mat.size();
        int n = mat[0].size();
        int best = 1;
        int dir[4][2] = {{0,1}, {1,0}, {0,-1}, {-1,0}};
        for (int d = 0; d < 4; d++) {
            int x = i + dir[d][0];
            int y = j + dir[d][1];
            if (x >= 0 && y >= 0 && x < m && y < n && mat[x][y] > mat[i][j])
{
                best = max(best, 1 + go(mat, x, y, dp));
            } }
        dp[i][j] = best;
        return best;  }
    int longestIncreasingPath(vector<vector<int>>& mat) {
        if (mat.empty()) return 0;
        int m = mat.size();
        int n = mat[0].size();
        vector<vector<int>> dp(m, vector<int>(n, 0));
        int ans = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                ans = max(ans, go(mat, i, j, dp));
            } }
        return ans;  }   };
```

Output:



Q10. **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.

Code:

```cpp
class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size();
        if (n == 0) return 0;
        int left = 0, right = n - 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;
    }
};
```

Output: