DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

LAB ASSIGNMENT

(Problems for Fast Learners)

Student Name: Sameer **UID:** 22BCS15631
Branch: Computer Science & Engineering **Section/Group:** IOT-614/B
Semester: 6th **Date of Performance:** 07/04/2025
Subject Name: Advanced Programming Lab-2 **Subject Code:** 22CSP-351

Q.1. Set Matrix Zeroes

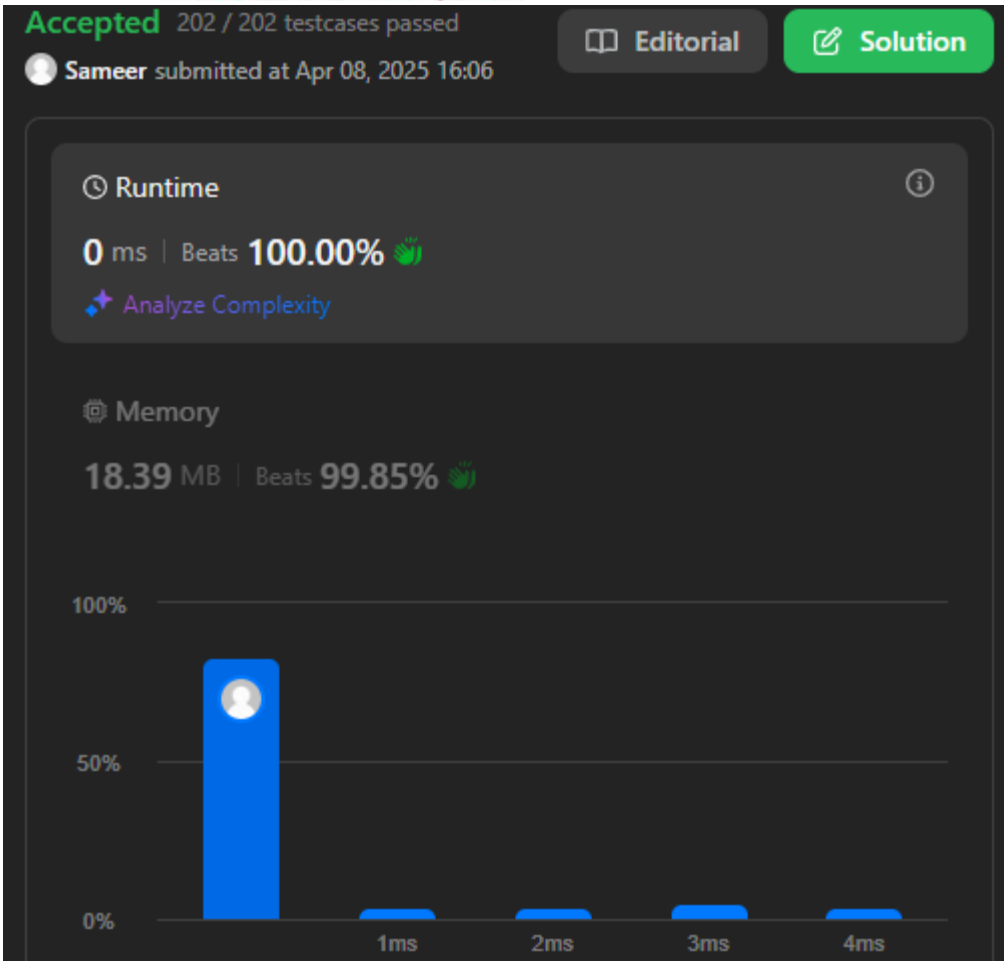
Given an m x n matrix, if an element is 0, set its entire row and column to 0. The modification must be done in place without using additional storage for another matrix.

Code:

```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
    int m = matrix.size();
    int n = matrix[0].size();
    bool firstRowZero = false, firstColZero = false;

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

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

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

    if (firstColZero)
        for (int i = 0; i < m; i++)
            matrix[i][0] = 0;
    }
};
```

Output:


Accepted 202 / 202 testcases passed
Sameer submitted at Apr 08, 2025 16:06
Editorial
Solution
Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
18.39 MB | Beats 99.85%
100%
50%
0%
1ms
2ms
3ms
4ms

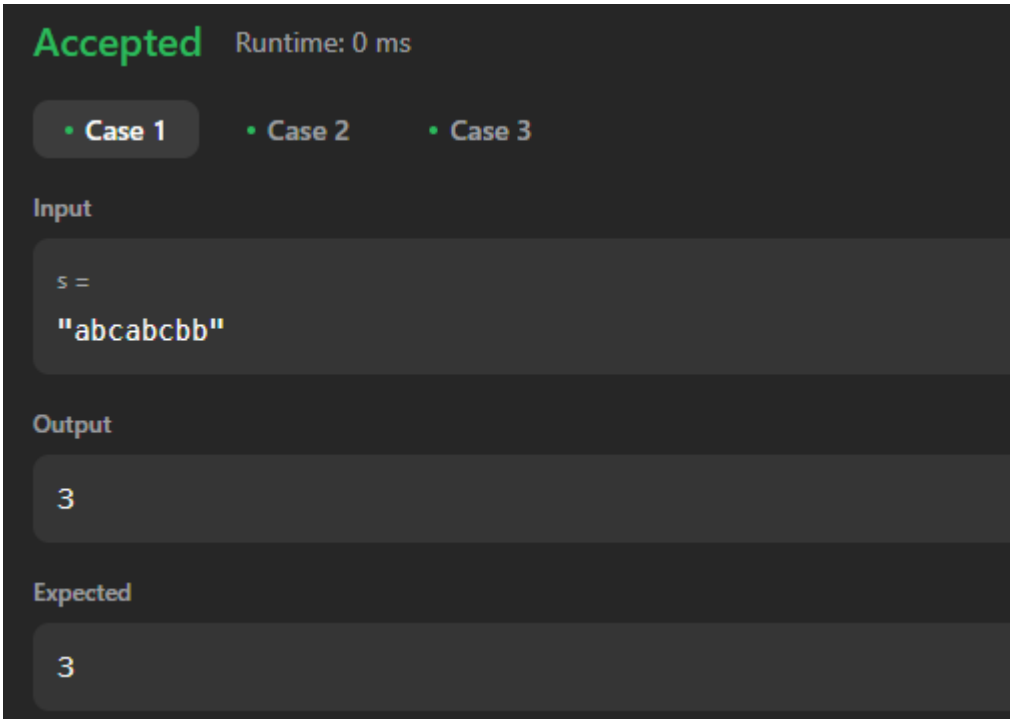
Q.2. Longest Substring Without Repeating Characters

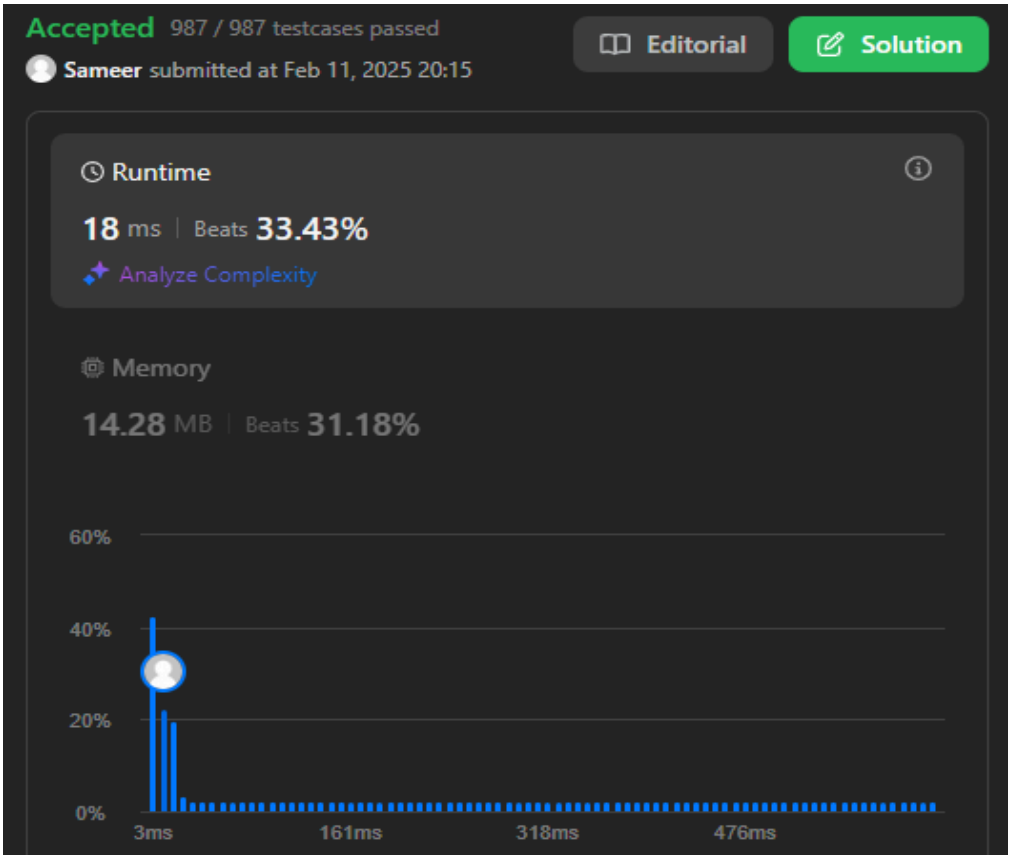
Given a string s, find the length of the longest substring that does not contain any repeating characters.

Code:

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_set<char> seen;
        int left = 0, right = 0, maxLen = 0;

        while (right < s.length()) {
            if (seen.find(s[right]) == seen.end()) {
                seen.insert(s[right]);
                maxLen = max(maxLen, right - left + 1);
                right++;
            }

            else {
                seen.erase(s[left]);
                left++;
            }
        }

        return maxLen;
    }
};
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
s =
"abcabcbb"
Output
3
Expected
3


Accepted
987 / 987 testcases passed
Sameer submitted at Feb 11, 2025 20:15
Editorial
Solution
Runtime
18 ms | Beats 33.43%
Analyze Complexity
Memory
14.28 MB | Beats 31.18%
60%
40%
20%
0%
3ms
161ms
318ms
476ms

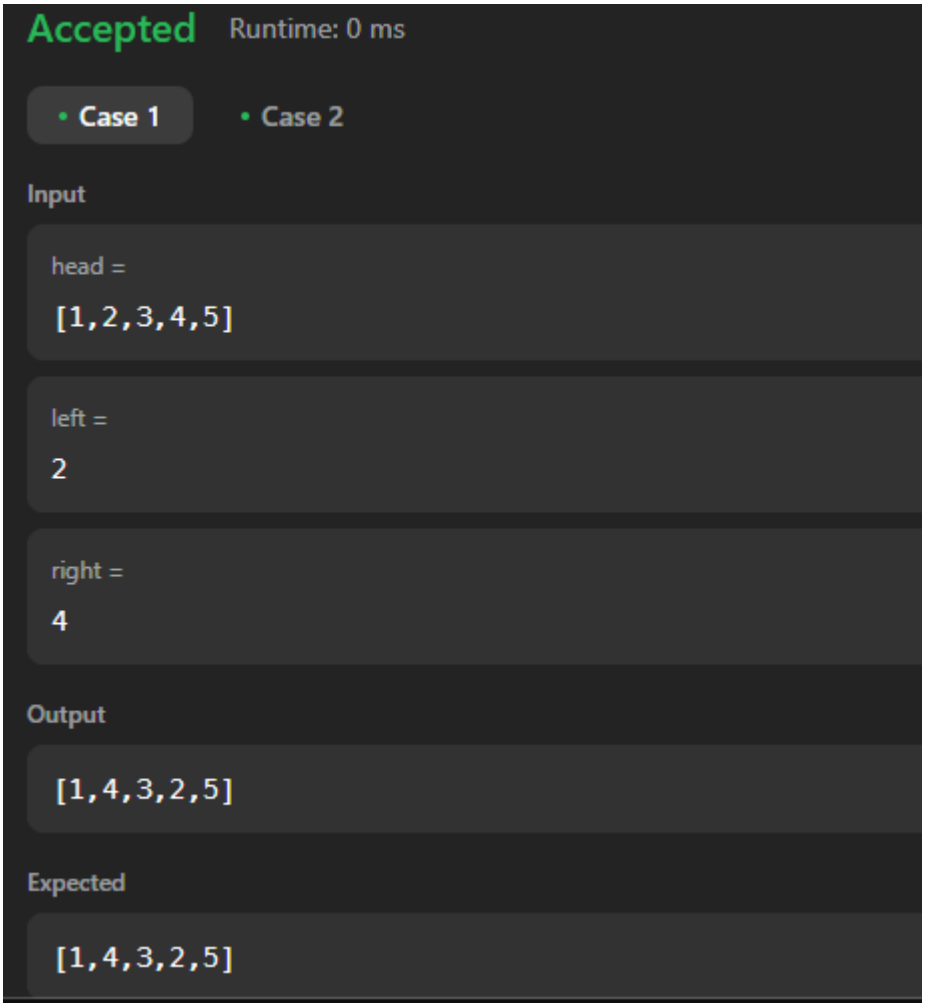
Q.3. Reverse Linked List II

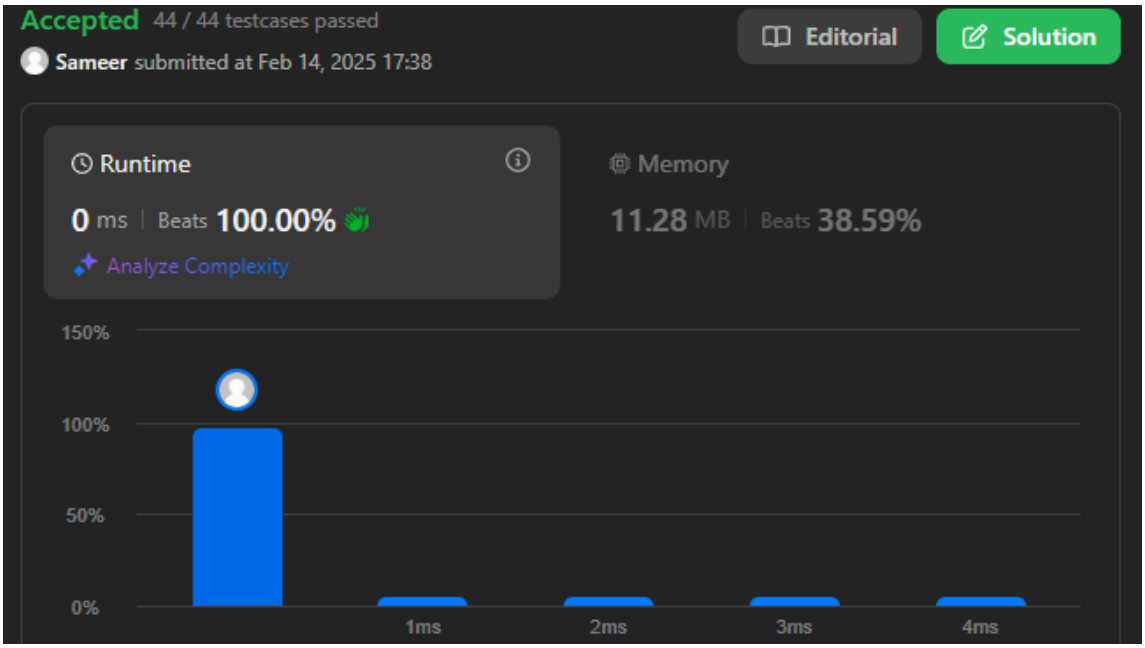
Given the head of a singly linked list and two integers left and right, reverse the nodes of the list from position left to right, and return the modified list.

Code:

```
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* start = prev->next;
      ListNode* then = start->next;

      for (int i = left; i < right; ++i)
      {
         start->next = then->next;
         then->next = prev->next;
         prev->next = then;
         then = start->next;
      }

      return dummy->next;
   }
};
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Input
head =
[1,2,3,4,5]
left =
2
right =
4
Output
[1,4,3,2,5]
Expected
[1,4,3,2,5]


Accepted 44 / 44 testcases passed
Sameer submitted at Feb 14, 2025 17:38
Editorial
Solution
Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
11.28 MB | Beats 38.59%
150%
100%
50%
0%
1ms
2ms
3ms
4ms

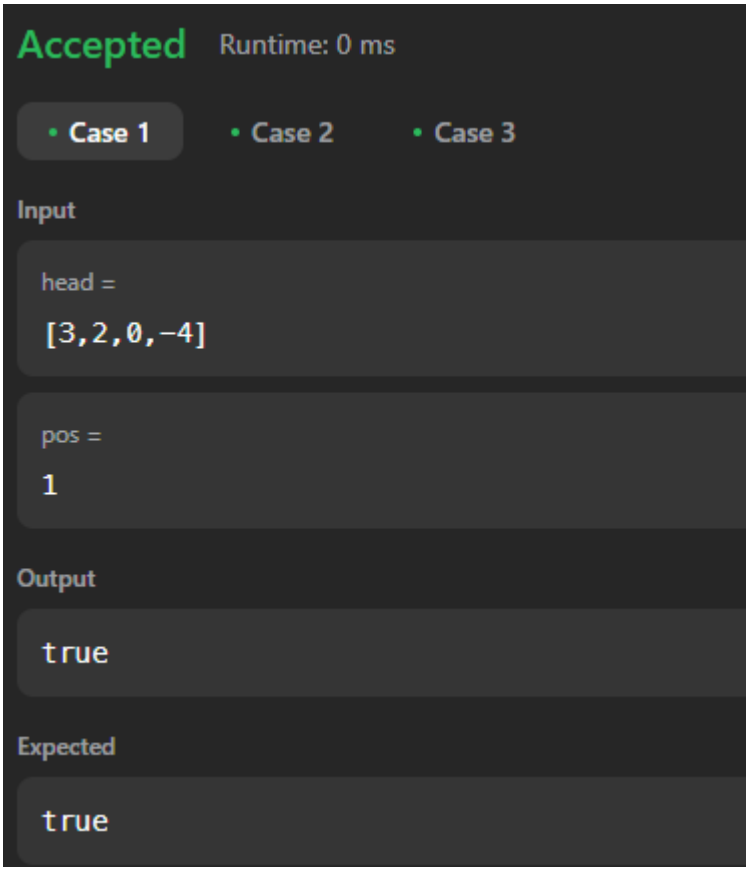
Q.4. Detect a Cycle in a Linked List

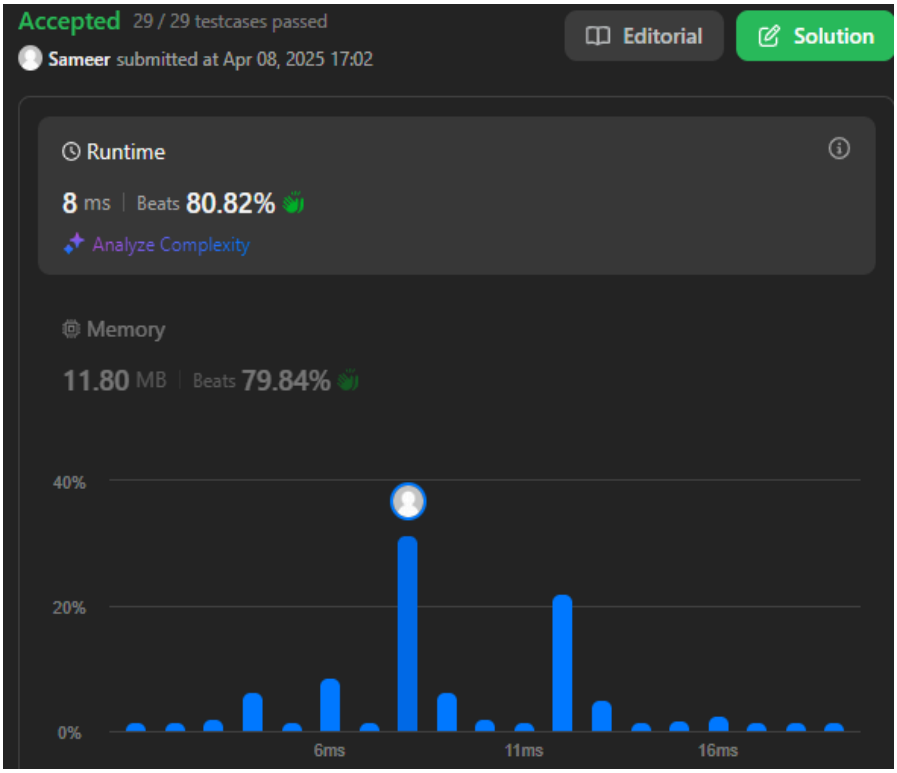
Given the head of a linked list, determine whether the linked list contains a cycle. A cycle occurs if a node's next pointer points to a previous node in the list.

Code:

```
class Solution {
public:
   bool hasCycle(ListNode *head)
   {
      if (head == NULL  ||  head->next == NULL)
      {
         return false ;
      }

      ListNode* slow = head ;
      ListNode* fast = head ;

      while (slow != NULL  &&  fast != NULL  &&  fast->next != NULL)
      {
         slow = slow->next ;
         fast = fast->next->next ;

         if (slow == fast)
         {
            return true ;
         }
      }

      return false ;
   }
};
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
head =
[3,2,0,-4]
pos =
1
Output
true
Expected
true


Accepted
29 / 29 testcases passed
Sameer submitted at Apr 08, 2025 17:02
Editorial
Solution
Runtime
8 ms | Beats 80.82%
Analyze Complexity
Memory
11.80 MB | Beats 79.84%
40%
20%
0%
6ms
11ms
16ms

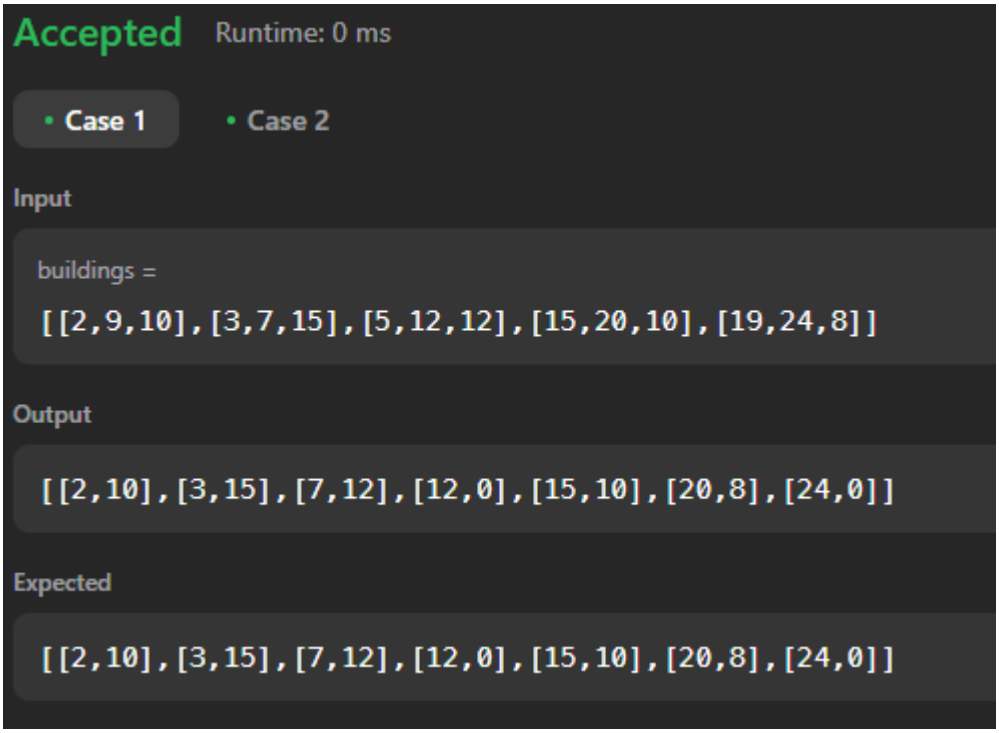
Q.5. The Skyline Problem

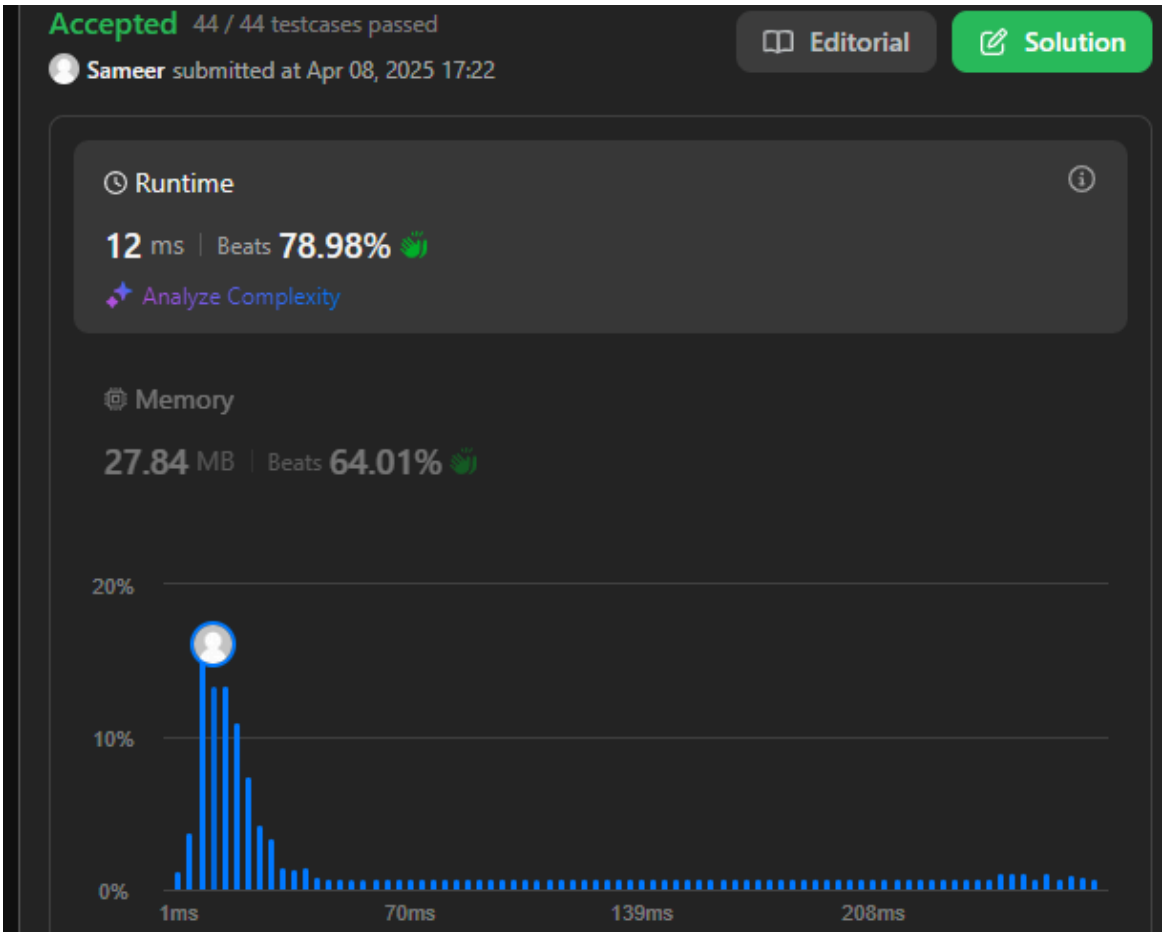
Given a list of buildings represented as [left, right, height], where each building is a rectangle, return the key points of the skyline. A key point is represented as [x, y], where x is the x-coordinate where the height changes to y.

Code:

```
class Solution {
public:
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
        vector<pair<int, int>> events;
        for (auto& b : buildings) {
            events.emplace_back(b[0], -b[2]);
            events.emplace_back(b[1], b[2]);
        }

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

        multiset<int> heights = {0};
        int prev = 0;
        vector<vector<int>> result;

        for (auto& [x, h] : events) {
            if (h < 0)
                heights.insert(-h);
            else
                heights.erase(heights.find(h));

            int curr = *heights.rbegin();
            if (curr != prev) {
                result.push_back({x, curr});
                prev = curr;
            }
        }
        return result;
    }
};
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Input
buildings =
[[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
Output
[[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
Expected
[[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]


Accepted
44 / 44 testcases passed
Sameer submitted at Apr 08, 2025 17:22
Editorial
Solution
Runtime
12 ms | Beats 78.98%
Analyze Complexity
Memory
27.84 MB | Beats 64.01%
20%
10%
0%
1ms
70ms
139ms
208ms

Q.6. Longest Increasing Subsequence II

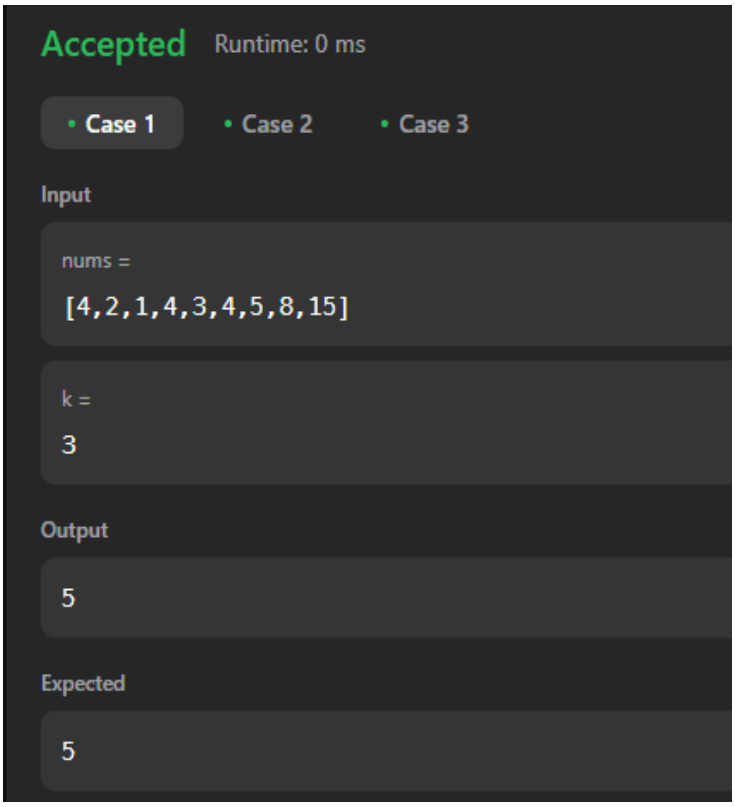
Given an integer array nums, find the length of the longest strictly increasing subsequence. A subsequence is derived from the array by deleting some or no elements without changing the order of the remaining elements.

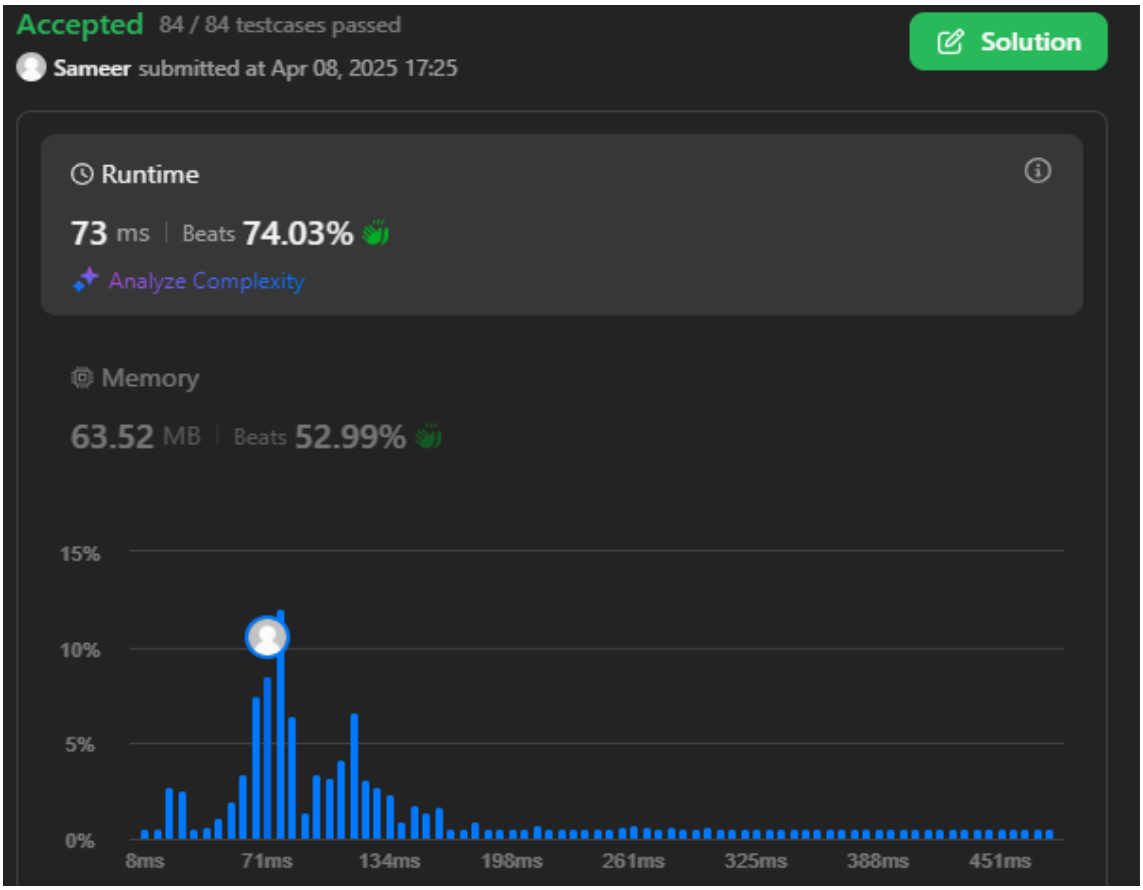
Code:

```
class SegmentTree {
   vector<int> tree;
   int size;

public:
   SegmentTree(int n) {
      size = n + 2;
      tree.resize(4 * size, 0);
   }

   void update(int index, int value, int node, int l, int r) {
      if (l == r) {
         tree[node] = max(tree[node], value);
         return;
      }

      int mid = (l + r) / 2;

      if (index <= mid){
         update(index, value, 2 * node, l, mid);
      }

      else{
         update(index, value, 2 * node + 1, mid + 1, r);
      }

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

   int query(int ql, int qr, int node, int l, int r) {
      if (ql > r || qr < l)
         return 0;

      if (ql <= l && r <= qr)
         return tree[node];
```

```
        int mid = (l + r) / 2;
        return max(query(ql, qr, 2 * node, l, mid),

            query(ql, qr, 2 * node + 1, mid + 1, r));
    }

    void update(int index, int value) { update(index, value, 1, 0, size - 1); }

    int query(int l, int r) { return query(l, r, 1, 0, size - 1); }
};

class Solution {
public:
    int lengthOfLIS(vector<int>& nums, int k) {
        int maxVal = *max_element(nums.begin(), nums.end());
        SegmentTree seg(maxVal);
        int result = 0;

        for (int num : nums) {
            int left = max(0, num - k);
            int right = num - 1;
            int best = seg.query(left, right);
            seg.update(num, best + 1);
            result = max(result, best + 1);
        }

        return result;
    }
};
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[4,2,1,4,3,4,5,8,15]
k =
3
Output
5
Expected
5


Accepted
84 / 84 testcases passed
Sameer submitted at Apr 08, 2025 17:25
Solution
Runtime
73 ms | Beats 74.03%
Analyze Complexity
Memory
63.52 MB | Beats 52.99%
15%
10%
5%
0%
8ms
71ms
134ms
198ms
261ms
325ms
388ms
451ms

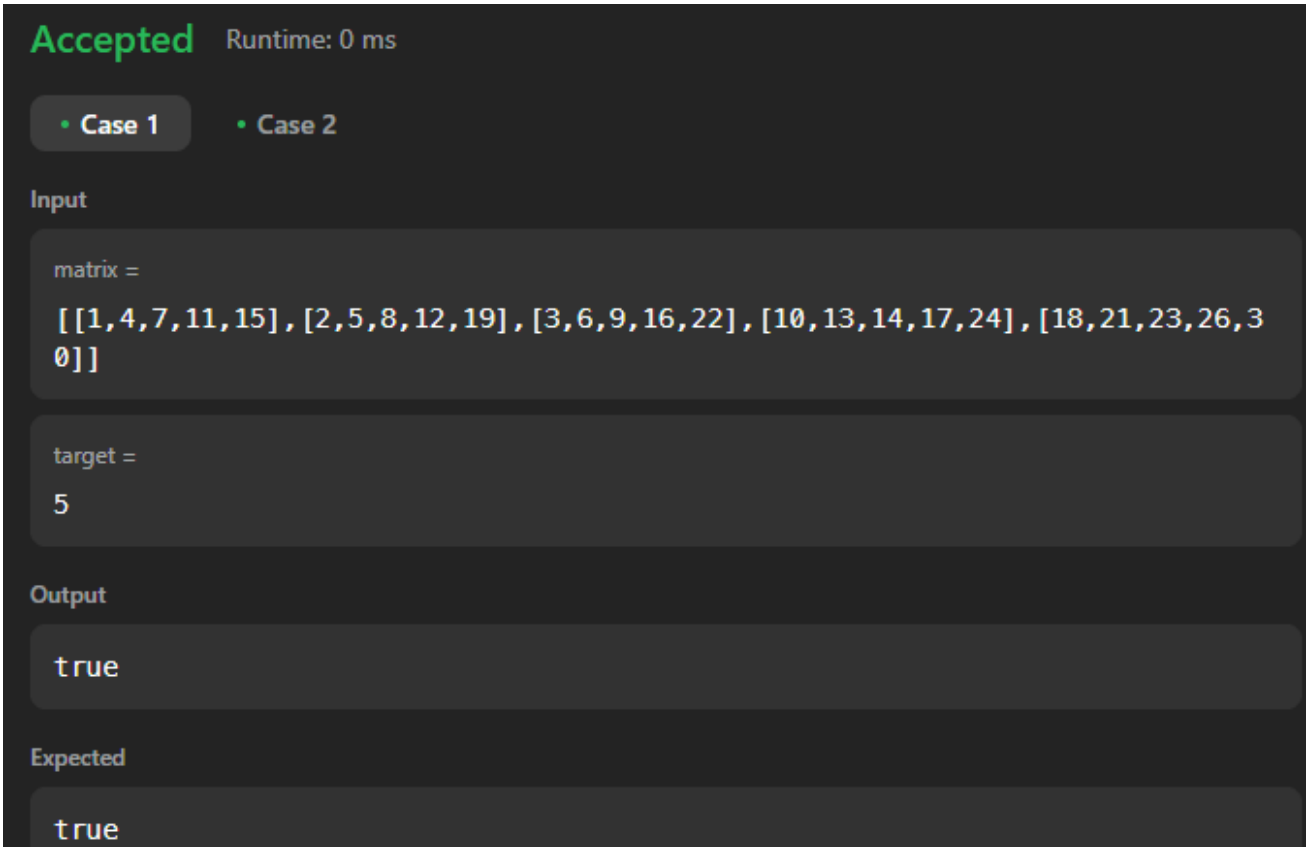
Q.7. Search a 2D Matrix II

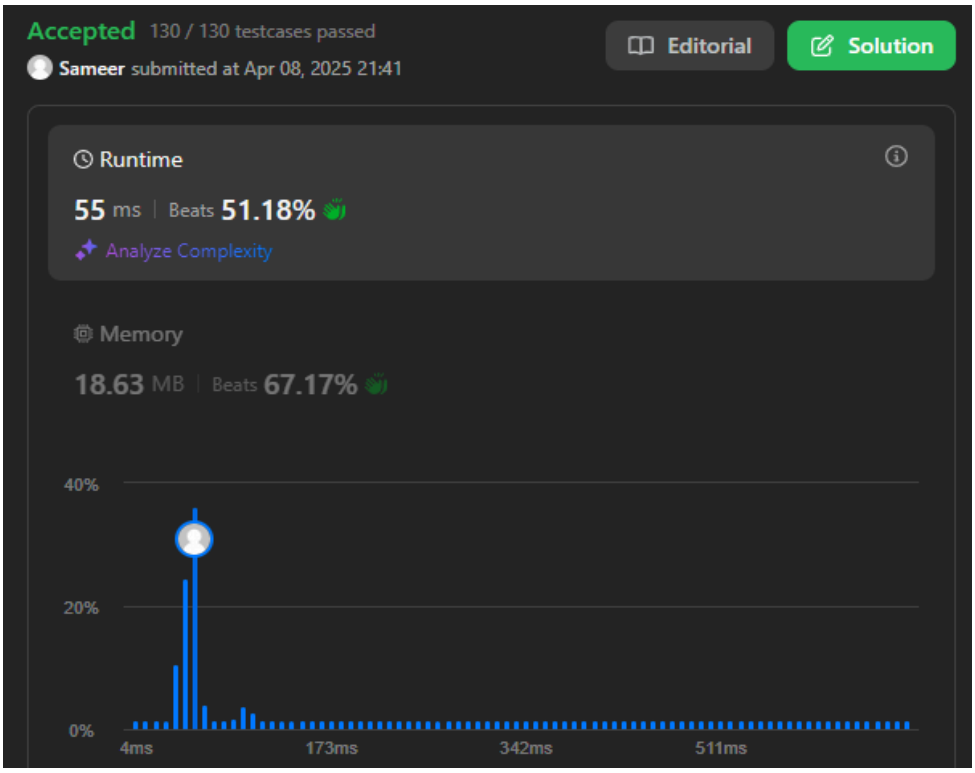
Given an m x n matrix where each row is sorted in ascending order from left to right and each column is sorted in ascending order from top to bottom, and an integer target, determine if the target exists in the matrix.

Code:

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size(), n = matrix[0].size();
        int row = 0, 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:


Accepted
Runtime: 0 ms
Case 1
Case 2
Input
matrix =
[[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]]
target =
5
Output
true
Expected
true


Accepted 130 / 130 testcases passed
Sameer submitted at Apr 08, 2025 21:41
Editorial
Solution
Runtime
55 ms | Beats 51.18%
Analyze Complexity
Memory
18.63 MB | Beats 67.17%
40%
20%
0%
4ms
173ms
342ms
511ms

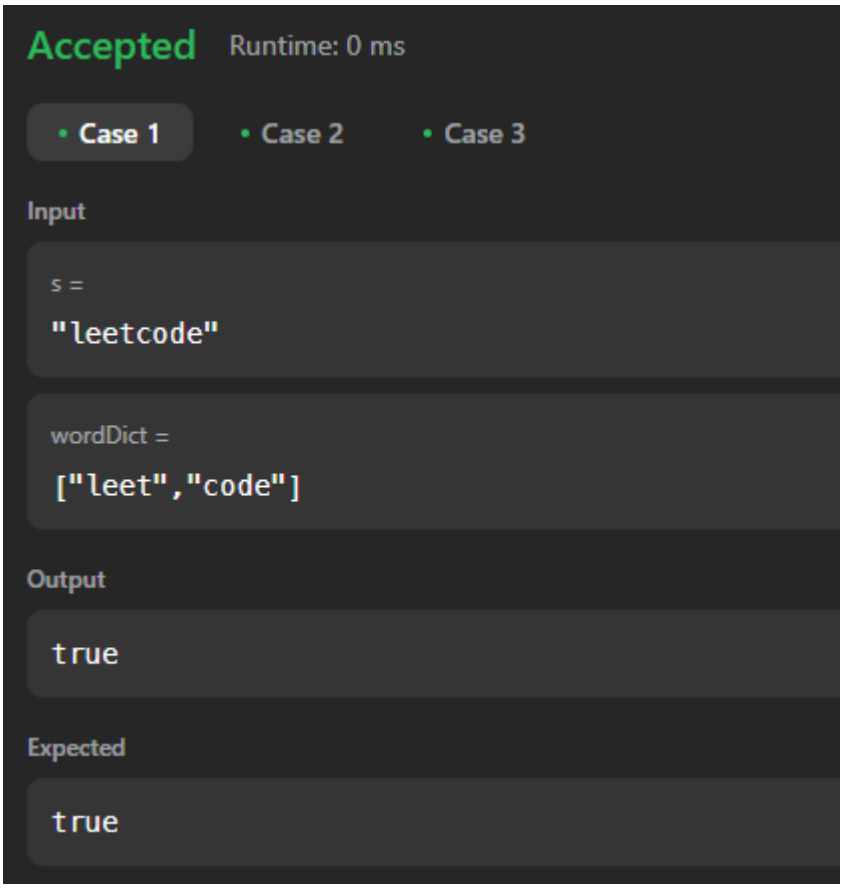
Q.8. Word Break

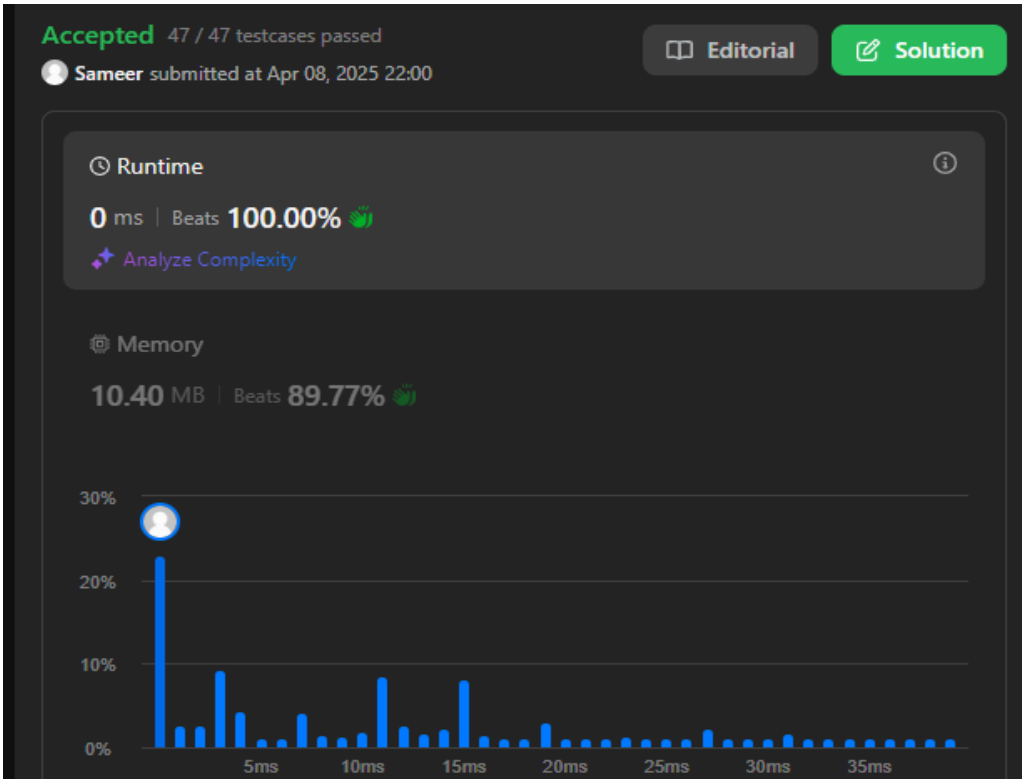
Given a string s and a dictionary wordDict containing a list of words, determine if s can be segmented into a space-separated sequence of one or more dictionary words. The same word can be reused multiple times.

Code:

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> wordSet(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 = i - 1; j >= max(0, i - 20); j--) {
                if (dp[j] && wordSet.find(s.substr(j, i - j)) != wordSet.end()) {
                    dp[i] = true;
                    break;
                }
            }
        }

        return dp[n];
    }
};
```

Output:


Accepted
Runtime: 0 ms
• Case 1
• Case 2
• Case 3
Input
s =
"leetcode"
wordDict =
["leet","code"]
Output
true
Expected
true


Accepted 47 / 47 testcases passed
Sameer submitted at Apr 08, 2025 22:00
Editorial
Solution
Runtime
0 ms | Beats 100.00%
Analyze Complexity
Memory
10.40 MB | Beats 89.77%
30%
20%
10%
0%
5ms
10ms
15ms
20ms
25ms
30ms
35ms

Q.9. Longest Increasing Path in a Matrix

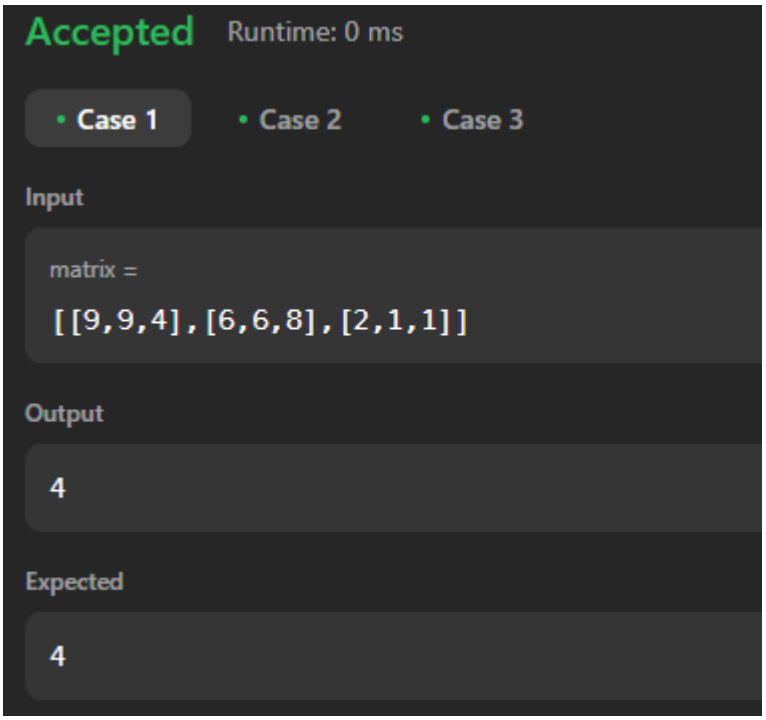
Given an m x n integer matrix, find the length of the longest strictly increasing path. You can move up, down, left, or right from each cell. Diagonal moves and moves outside the boundaries are not allowed.

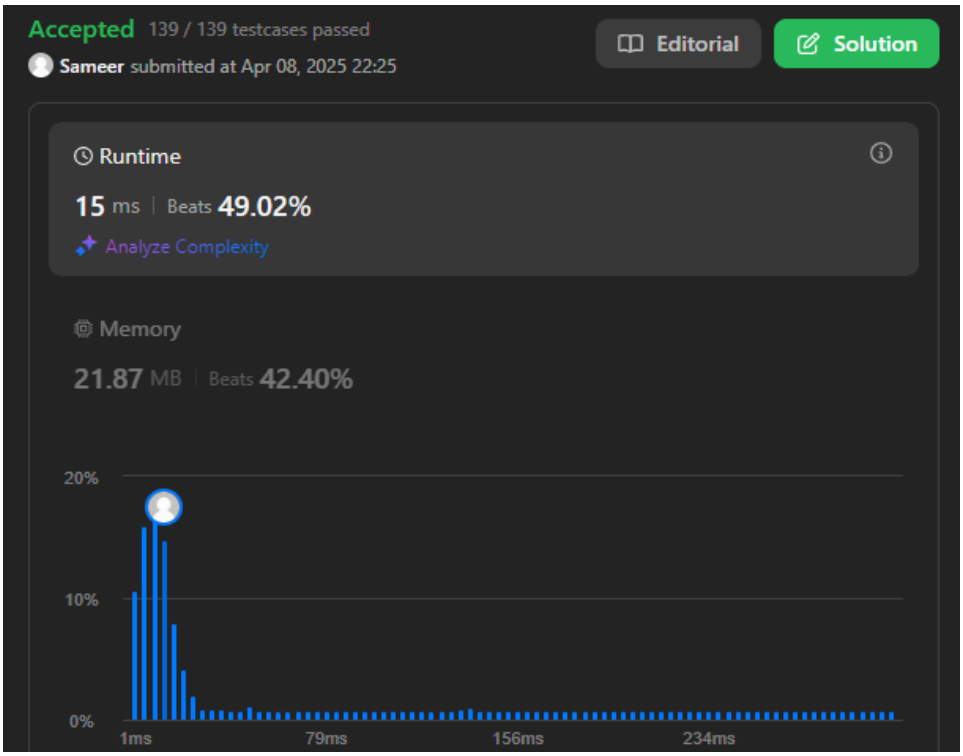
Code:

```
class Solution {
public:
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        int m = matrix.size(), n = matrix[0].size();
        vector<vector<int>> dp(m, vector<int>(n, 0));
        int res = 0;
        vector<int> dirs = {0, 1, 0, -1, 0};

        function<int(int, int)> dfs = [&](int i, int j) {
            if (dp[i][j]){
                return dp[i][j];
            }

            int maxLen = 1;
            for (int d = 0; d < 4; ++d) {
                int x = i + dirs[d], y = j + dirs[d + 1];
                if (x >= 0 && x < m && y >= 0 && y < n &&
                    matrix[x][y] > matrix[i][j]) {
                    maxLen = max(maxLen, 1 + dfs(x, y));
                }
            }
            return dp[i][j] = maxLen;
        };

        for (int i = 0; i < m; ++i){
            for (int j = 0; j < n; ++j){
                res = max(res, dfs(i, j));
            }
        }
```

```
        return res;
    }
};
```

Output:





Q.10. 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:

```
class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size();
        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: