Advance Problem

Student Name: Karan Goyal **UID:-22BCS15864**
Branch: BE-CSE **Group-Ntpp_IOT_602-A**
Semester: 5TH **Dateof Performance:8-4-25**
Subject Name:-Advance Programming lab-2 **Subject Code:-22CSP-351**

Problem 1

Aim:- You 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.

Example 1: Input: matrix = [[1, 1, 1], [1, 0, 1], [1, 1, 1]] Output: [[1, 0, 1], [0, 0, 0], [1, 0, 1]] Explanation: The element at position (1,1) is 0. Therefore, the entire row 1 and column 1 are set to 0.

Example 2: Input: matrix = [[0, 1, 2, 0], [3, 4, 5, 2], [1, 3, 1, 5]] Output: [[0, 0, 0, 0], [0, 4, 5, 0], [0, 3, 1, 0]] Explanation: The zeros in the first row (positions (0,0) and (0,3)) cause the entire first row and their corresponding columns to be set to 0.

Objective:- The objective of this problem is to modify a given 2D matrix such that if any cell contains the value 0, its entire row and column are set to 0. The challenge is to perform this in-place without using additional matrix storage. This problem enhances understanding of array manipulation, efficient space usage, and in-place updates, which are crucial for memory-optimized programming in real-world applications.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm: Set Matrix Zeroes

Input: A 2D matrix matrix of size m x n
Output: Modify the matrix *in-place* such that if an element is 0, its entire row and column are set to 0.

1. **Initialize Variables**
   - rows = number of rows in matrix
   - cols = number of columns in matrix
   - firstRowHasZero = false → To check if the first row needs to be zeroed
   - firstColHasZero = false → To check if the first column needs to be zeroed
2. **Check First Row for Zeroes**
   - Traverse the first row (matrix[0][c] for all columns c)
   - If any element is 0, set firstRowHasZero = true
3. **Check First Column for Zeroes**
   - Traverse the first column (matrix[r][0] for all rows r)
   - If any element is 0, set firstColHasZero = true
4. **Mark Zeroes in First Row & First Column**
   - For all elements excluding the first row and first column (r=1 to rows-1, c=1 to cols-1)

    - If matrix[r][c] == 0, mark its row and column by:
        - matrix[r][0] = 0
        - matrix[0][c] = 0

5. **Set Marked Rows to Zero**
    - For each row r = 1 to rows-1
    - If matrix[r][0] == 0, set all elements in that row (from column 1 to cols-1) to 0
6. **Set Marked Columns to Zero**
    - For each column c = 1 to cols-1
    - If matrix[0][c] == 0, set all elements in that column (from row 1 to rows-1) to 0
7. **Zero Out the First Row (if needed)**
    - If firstRowHasZero is true, set all elements in the first row to 0
8. **Zero Out the First Column (if needed)**
    - If firstColHasZero is true, set all elements in the first column to 0

Time Complexity:

- **O(m × n)** — Full matrix is traversed multiple times.

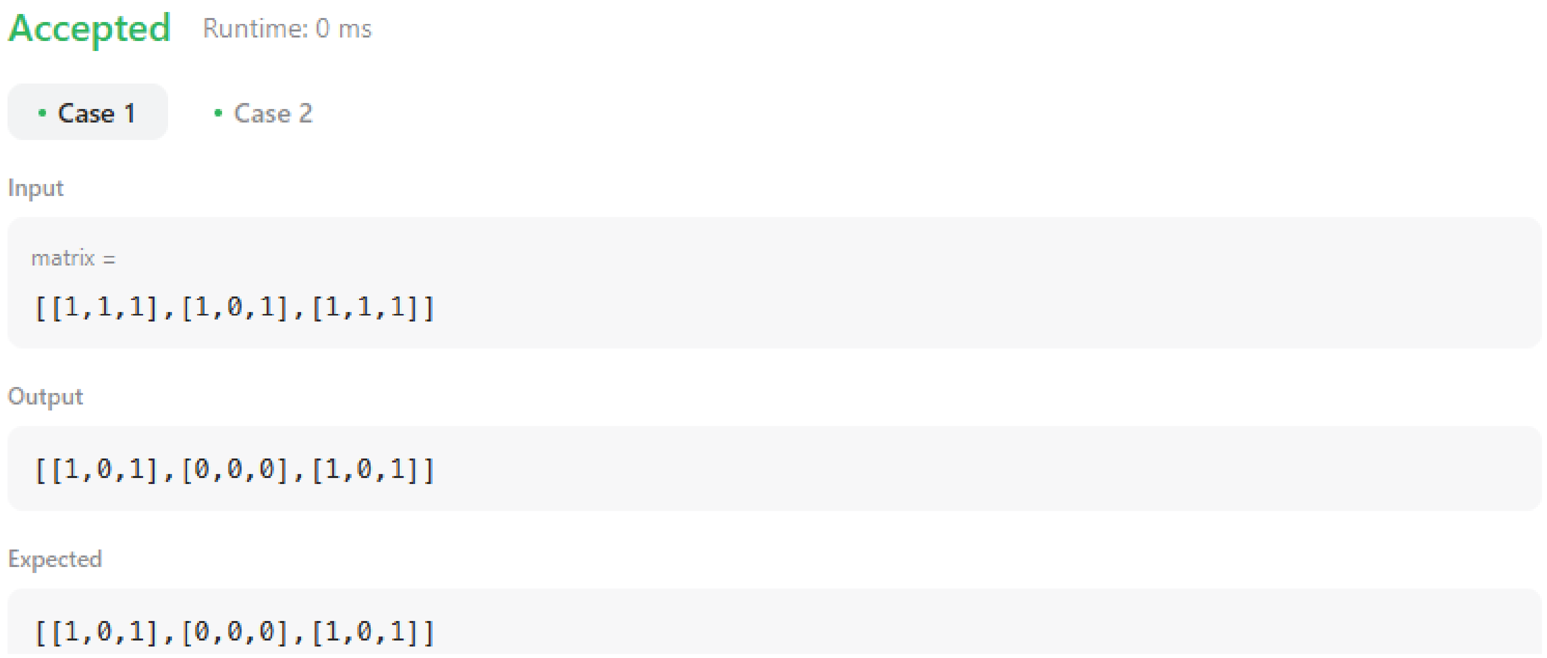
Space Complexity:

- **O(1)** — No extra space used (modifies matrix in-place).

Code:

```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int rows = matrix.size();
        int cols = matrix[0].size();

        bool firstRowHasZero = false;
        bool firstColHasZero = false;

        // Check if the first row contains zero
        for (int c = 0; c < cols; c++) {
            if (matrix[0][c] == 0) {
                firstRowHasZero = true;
                break;
            }
        }

        // Check if the first column contains zero
        for (int r = 0; r < rows; r++) {
            if (matrix[r][0] == 0) {
                firstColHasZero = true;
                break;
            }
    }

    // Use the first row and column as markers
    for (int r = 1; r < rows; r++) {
        for (int c = 1; c < cols; c++) {
            if (matrix[r][c] == 0) {
                matrix[r][0] = 0;
                matrix[0][c] = 0;
            }
        }
    }

    // Set the marked rows to zero
    for (int r = 1; r < rows; r++) {
        if (matrix[r][0] == 0) {
            for (int c = 1; c < cols; c++) {
                matrix[r][c] = 0;
            }
        }
    }

    // Set the marked columns to zero
    for (int c = 1; c < cols; c++) {
        if (matrix[0][c] == 0) {
            for (int r = 1; r < rows; r++) {
                matrix[r][c] = 0;
            }
        }
    }

    // Set the first row to zero if needed
    if (firstRowHasZero) {
        for (int c = 0; c < cols; c++) {
            matrix[0][c] = 0;
        }
    }

    // Set the first column to zero if needed
    if (firstColHasZero) {
        for (int r = 0; r < rows; r++) {
            matrix[r][0] = 0;
        }
    }
}};
```

Output- All the test cases passed



Input

matrix =

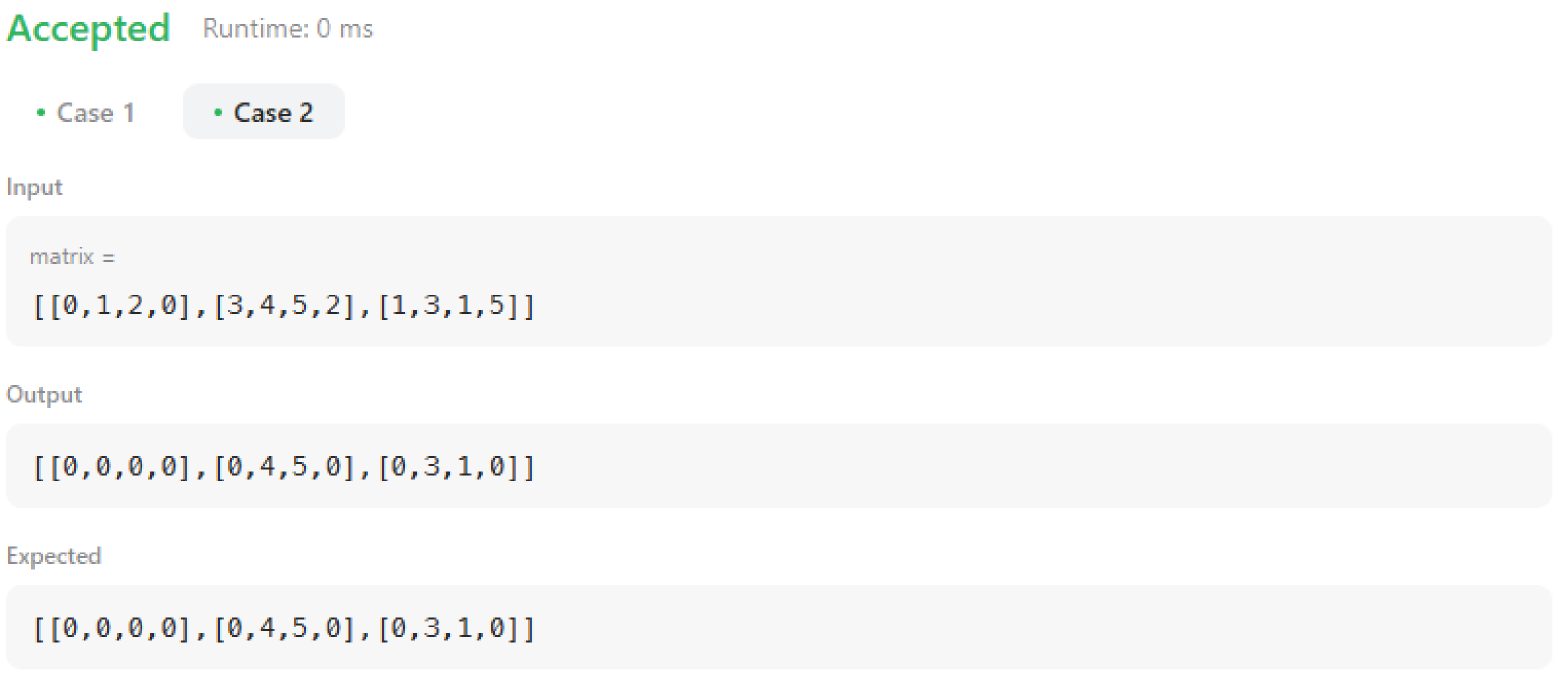
```
[[1,1,1],[1,0,1],[1,1,1]]
```

Output

```
[[1,0,1],[0,0,0],[1,0,1]]
```

Expected

```
[[1,0,1],[0,0,0],[1,0,1]]
```



Input

matrix =

```
[[0,1,2,0],[3,4,5,2],[1,3,1,5]]
```

Output

```
[[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

Expected

```
[[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

Problem-2

Aim:- You Given a string s, find the length of the longest substring that does not contain any repeating characters.
Example 1: Input: s = "abcabcbb" Output: 3 Explanation: The longest substring without repeating characters is "abc", which has a length of 3.
2: Input: s = "bbbbb" Output: 1 Explanation: All characters in the string are the same, so the longest substring with unique characters is "b", with a length of 1.

Objective- The goal of this problem is to find the length of the longest substring in a given string that contains all unique characters without any repetition. It helps to build skills in string manipulation, the sliding window technique, and hash-based tracking of characters. This problem develops an understanding of how to maintain optimal time complexity while processing continuous sequences in strings.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm : Longest Substring Without Repeating Characters

Approach Used:
Sliding Window + **HashSet** to track unique characters.

1. **Initialize Pointers and Variables**
   - left = 0: Starting index of the sliding window.
   - maxLength = 0: Keeps track of the maximum length of the substring found so far.
   - charSet = {}: An unordered set to store characters currently in the window (to check for duplicates efficiently).
2. **Iterate with Right Pointer**
   - Loop right from 0 to s.length() - 1:
     - Each iteration, the right pointer represents the end of the current substring window.
3. **Check for Duplicate Character**
   - While s[right] exists in charSet:
     - It means we have a duplicate character.
     - Remove s[left] from the charSet.
     - Move left++ to shrink the window from the left side until the duplicate is removed.
4. **Add Current Character**
   - Insert s[right] into the charSet.
5. **Update Maximum Length**
   - Compute the current window size: right - left + 1
   - Update maxLength with the maximum of current maxLength and window size.
6. **Return Result**
   - After the loop ends, return maxLength.

Example Walkthrough:

Let s = "abcabcbb"

- Start with left = 0, right = 0
- Expand window: "a", "ab", "abc" → maxLength = 3
- Encounter "a" again → shrink from left to remove duplicate
- Continue expanding... maxLength remains 3

Time Complexity:

- **O(n)** — Each character is visited at most twice (once by right, once by left).

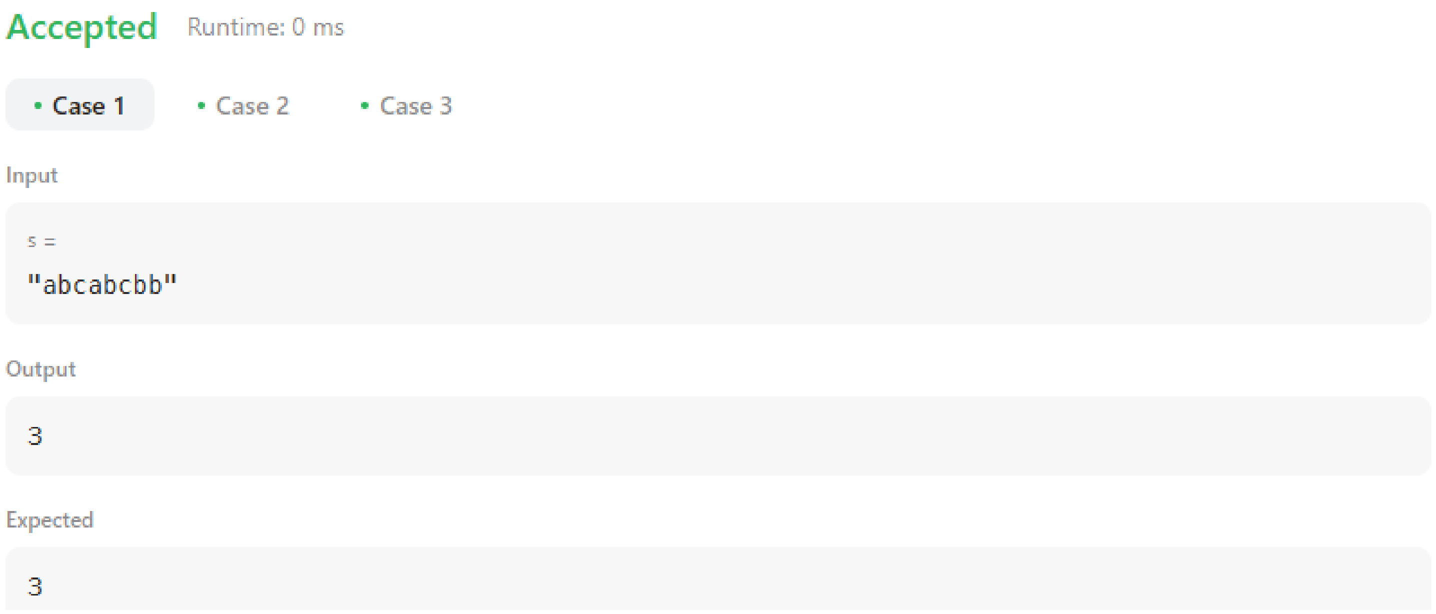
Space Complexity:

- **O(k)** — k is the size of the character set (at most 26 or 128 depending on input).

Code-

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

        for (int right = 0; right < s.length(); right++) {
            while (charSet.find(s[right]) != charSet.end()) {
                charSet.erase(s[left]);
                left++;
            }

            charSet.insert(s[right]);
            maxLength = max(maxLength, right - left + 1);
        }

        return maxLength;
       }
    }
};
```

Result-All test cases passed


Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

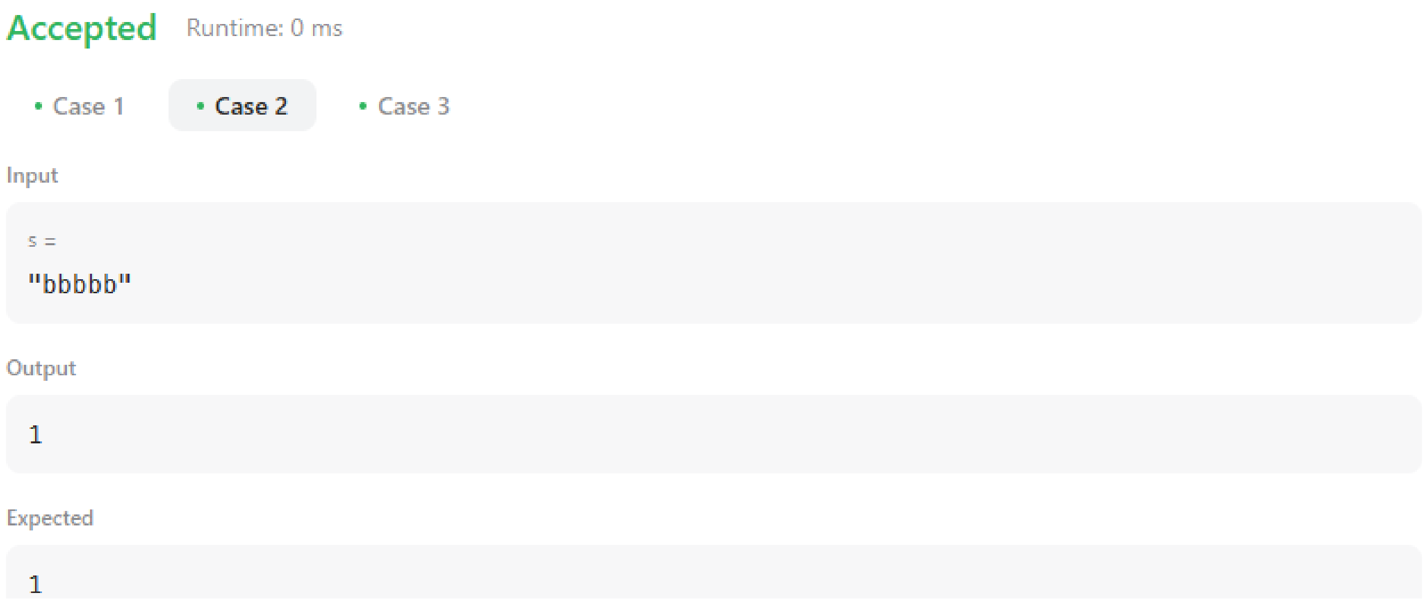
```
s =
"abcabcbb"
```

Output

```
3
```

Expected

```
3
```


Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

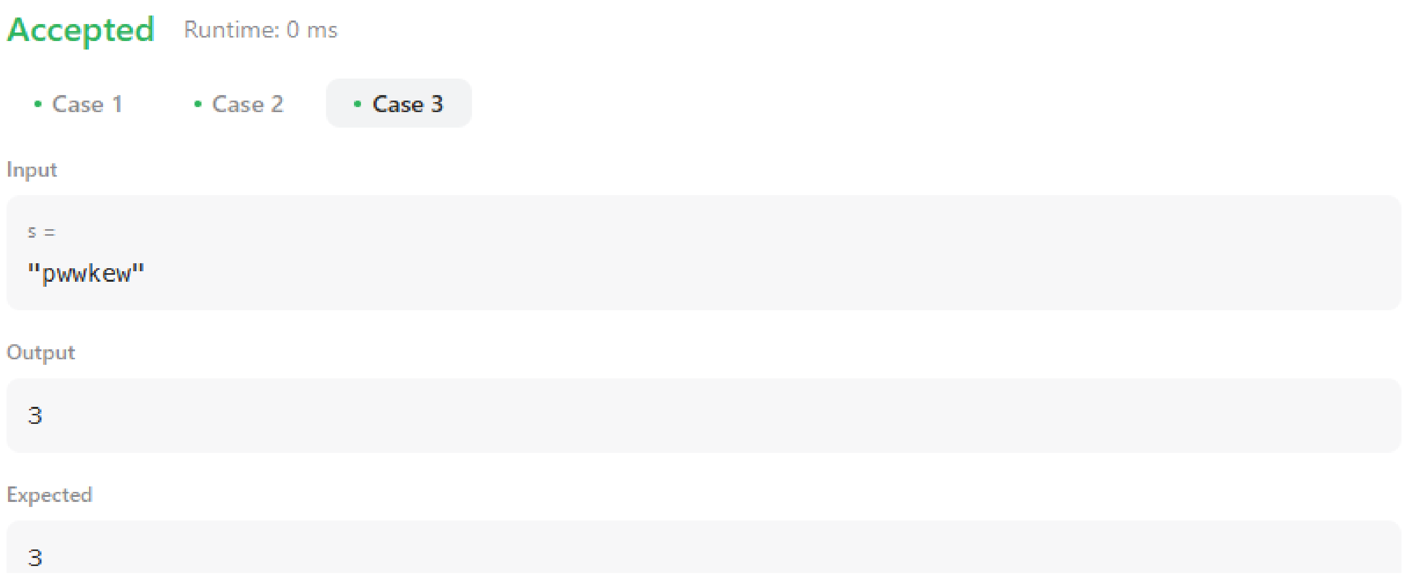
```
s =
"bbbbb"
```

Output

```
1
```

Expected

```
1
```


Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
s =
"pwwkew"
```

Output

```
3
```

Expected

```
3
```

Problem-3

Aim- 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.

Example 1: Input: Linked list: [3, 2, 0, -4] with the tail node (-4) pointing to the node with value 2. Output: true Explanation: The tail node connects back to an earlier node, forming a cycle.

Example 2: Input: Linked list: [1, 2] with no cycle (each node points to null at the end). Output: false Explanation: There is no cycle since no node points back to a previous node.

Objective- The objective is to determine whether a given singly linked list contains a cycle, i.e., whether any node in the list links back to a previous node. This problem helps in understanding pointer manipulation, Floyd's cycle detection algorithm (Tortoise and Hare), and linked list traversal techniques. It is fundamental for detecting loops in data structures which is essential in preventing infinite loops in real applications.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm: Detect a Cycle in a Linked List

Approach Used:
Floyd's Cycle Detection Algorithm (also known as the **Tortoise and Hare** algorithm).

Step-by-step Algorithm:

1. **Initialize Two Pointers**
   - slow pointer: starts at the head, moves **one step** at a time.
   - fast pointer: also starts at the head, moves **two steps** at a time.
2. **Traverse the Linked List**
   - While fast and fast->next are not nullptr:
     - Move slow by one step: slow = slow->next
     - Move fast by two steps: fast = fast->next->next
3. **Check for Cycle**
   - After each movement, check:
     if (slow == fast) → This means the two pointers met, so a **cycle exists**.
   - Return true if a cycle is detected.
4. **End of List Reached**
   - If fast reaches the end (nullptr), the list **does not** have a cycle.

- Return false.

Time Complexity:

- **O(n)** — Each pointer traverses at most n nodes.

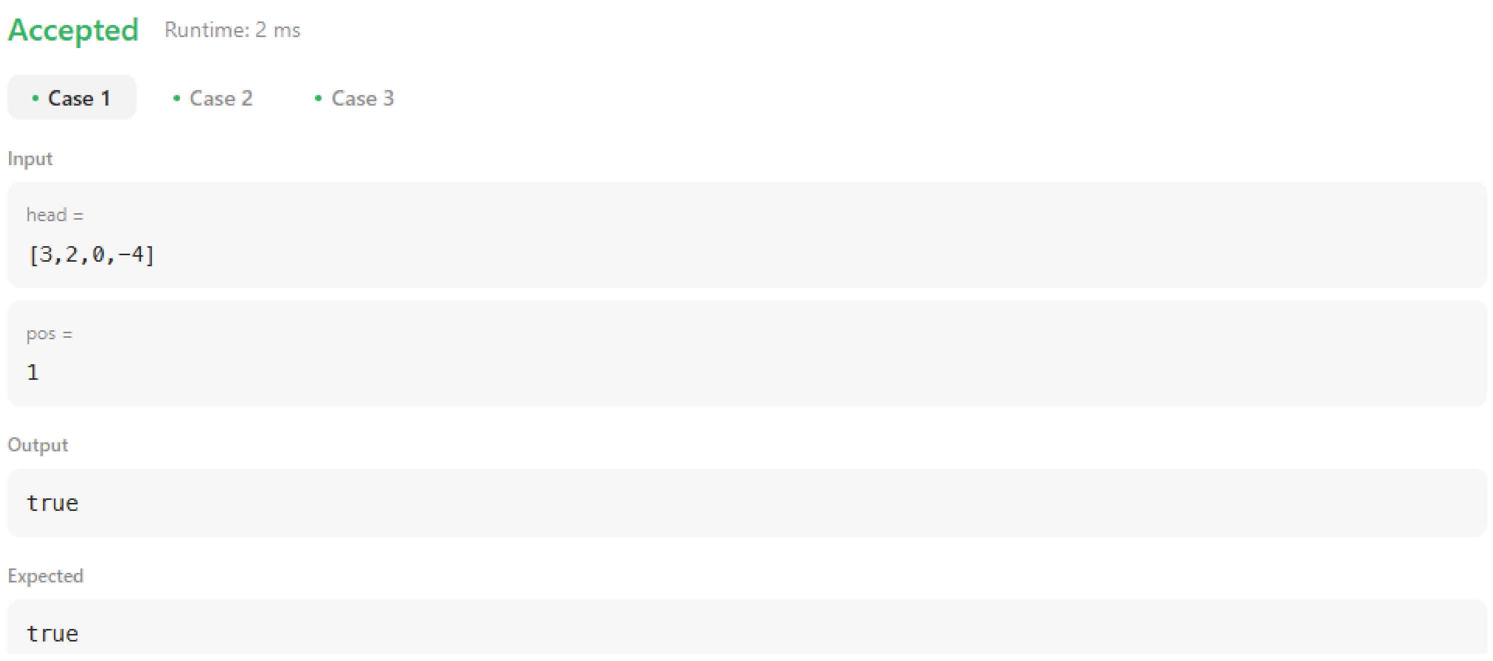
Space Complexity:

- **O(1)** — No extra memory used; only two pointers.

Code-

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* fast = head;
        ListNode* slow = head;

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

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

        return false;
    }

};
```

Result- All test cases passes



Accepted Runtime: 2 ms

• Case 1 • Case 2 • Case 3

Input

```
head =
[3,2,0,-4]
```

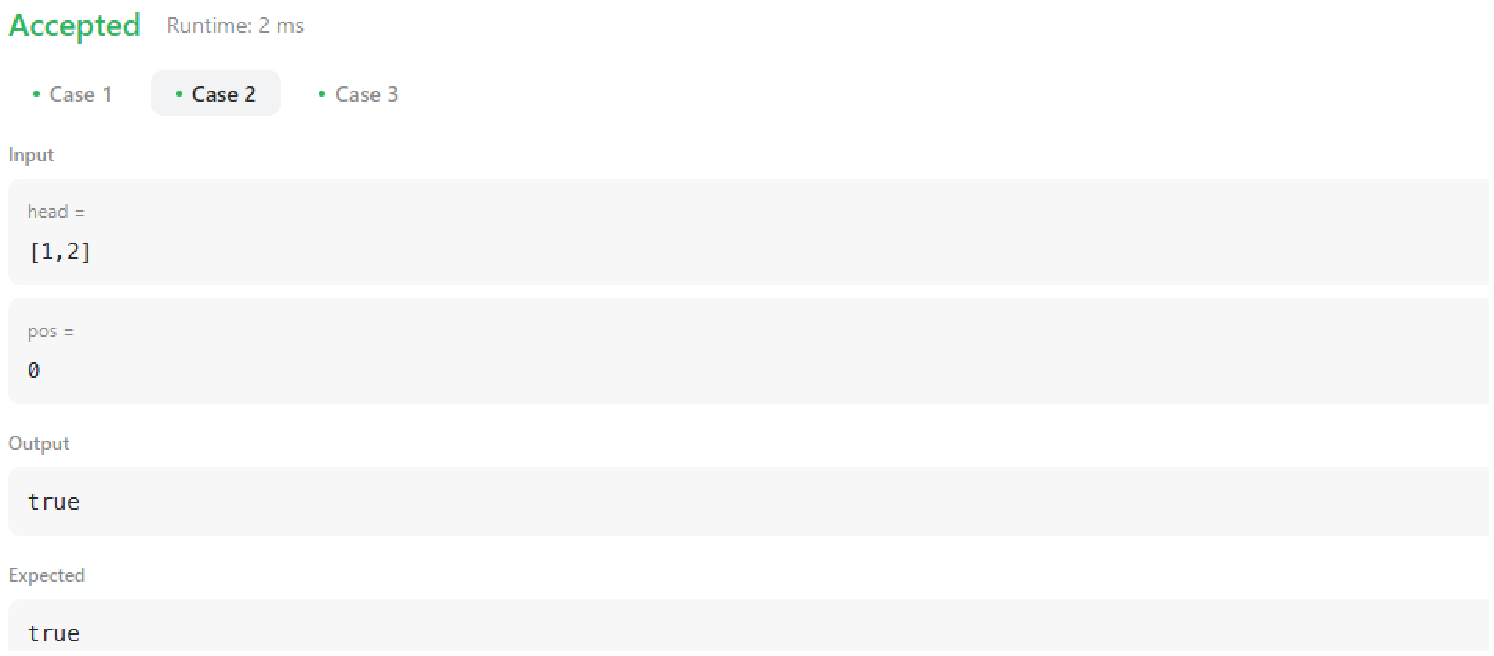
```
pos =
1
```

Output

```
true
```

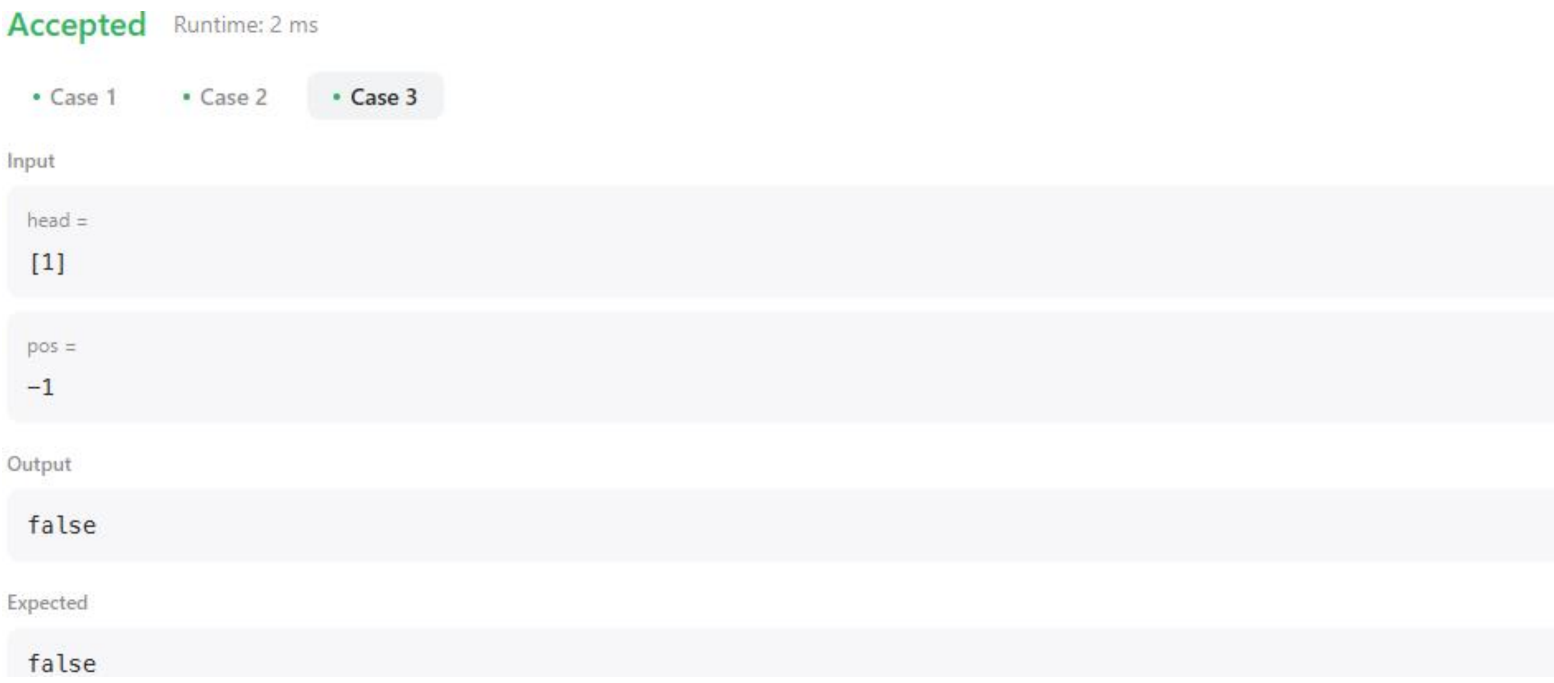
Expected

```
true
```


Accepted Runtime: 2 ms

Input

head =
```
[1,2]
```
pos =
```
0
```
Output
```
true
```
Expected
```
true
```


Accepted Runtime: 2 ms

Input

head =
```
[1]
```
pos =
```
-1
```
Output
```
false
```
Expected
```
false
```

Learning Outcomes

1. **Understand and implement efficient in-place matrix manipulation**, including the use of markers to minimize space complexity while updating entire rows and columns.
2. **Apply the sliding window technique** to solve substring-related problems, enhancing skills in handling dynamic window sizes and avoiding repeated characters efficiently.
3. **Utilize two-pointer techniques for cycle detection**, particularly Floyd's Cycle Detection Algorithm, to identify cycles in linked lists with constant space.
4. **Develop proficiency in using hash-based data structures** such as sets for quick lookups and uniqueness checks within string or array contexts.
5. **Strengthen logical thinking and algorithmic problem-solving** through practical applications of array, string, and linked list manipulation in real-time scenarios.