

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment-8(A)

Student Name: Pranav Saini

UID: 22BCS15994

Branch: CSE

Section/Group: NTPP_602-A

Semester: 6

Date of Performance: 15-03-25

Subject Name: Advanced Programming Lab-2

Subject Code: 22CSH-359

1. **Title:** Graphs (Number of Islands)
2. **Objective:** To count the number of islands in a given $m \times n$ 2D grid where '1' represents land and '0' represents water.
3. **Algorithm:**
 - a) **Input:** A 2D grid representing land ('1') and water ('0').
 - b) **Initialization:**
 - a. Define `count = 0` to track the number of islands.
 - c) **DFS Traversal:**
 - a. For each cell (i, j) in the grid:
 - i. If `grid[i][j] == '1'`:
 1. Increment `count` by 1.
 2. Call the **DFS** function to mark all connected '1's as visited.
 - d) **DFS Function:**
 - a. If cell (i, j) is out of grid boundaries or is '0', return.
 - b. Otherwise, mark `grid[i][j] = '0'`.
 - c. Recursively call DFS for its 4 adjacent cells (up, down, left, right).
 - e) **Output:** Return the `count` as the total number of islands.

4. **Implementation/Code:**

```
class Solution {
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;

        int count = 0;
        int rows = grid.length;
        int cols = grid[0].length;
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

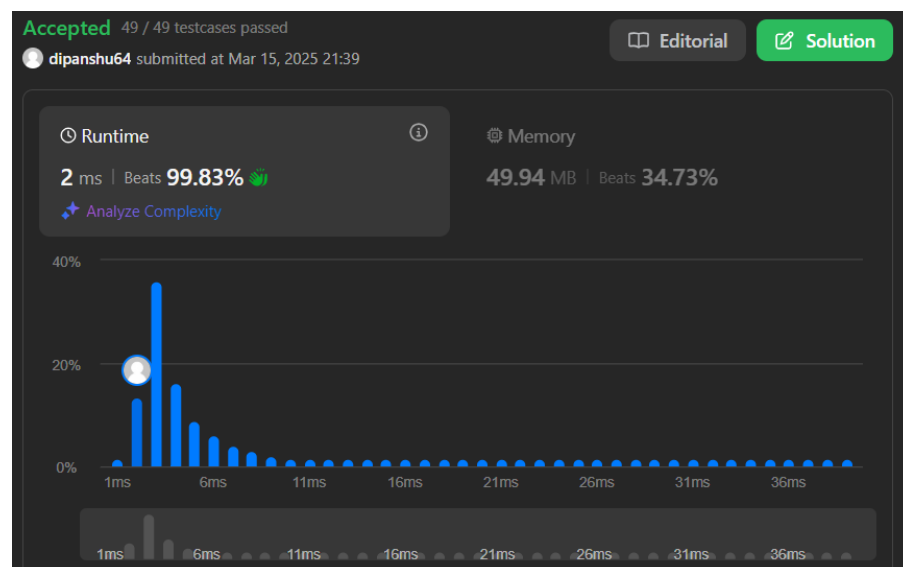
Discover. Learn. Empower.

```
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (grid[i][j] == '1') {
                    count++;
                    dfs(grid, i, j); // Mark the entire island as
visited
                }
            }
        }
        return count;
    }

    // DFS function to mark connected '1's as visited
    private void dfs(char[][] grid, int i, int j) {
        if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length ||
grid[i][j] == '0') {
            return;
        }

        grid[i][j] = '0'; // Mark the cell as visited
        dfs(grid, i + 1, j); // Down
        dfs(grid, i - 1, j); // Up
        dfs(grid, i, j + 1); // Right
        dfs(grid, i, j - 1); // Left
    }
}
```

5. Output:



6. Time Complexity: $O(m * n)$

7. Space Complexity: $O(m * n)$



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment 8(B)

1. **Title:** Word Ladder
2. **Objective:** To find the shortest transformation sequence from beginWord to endWord such that:
 - Each transformed word must exist in the given word list.
 - Each transformation changes only one letter at a time.
3. **Algorithm:**
 - **Input:** beginWord, endWord, and wordList.
 - **Check Condition:** If endWord is not in wordList, return 0.
 - **BFS Initialization:**
 - Use a queue for BFS traversal.
 - Add beginWord to the queue with steps = 1.
 - **BFS Traversal:**
 - While the queue is not empty:
 - Dequeue the front element.
 - For each letter position in the word:
 - Replace that letter with 'a' to 'z'.
 - If the new word is endWord, return steps + 1.
 - If the new word exists in wordList, add it to the queue.
 - **If no transformation found:** Return 0.

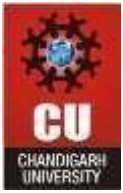
4. **Implementation/Code:**

```
import java.util.*;

class Solution {
    public int ladderLength(String beginWord, String endWord,
List<String> wordList) {
        Set<String> wordSet = new HashSet<>(wordList);
        if (!wordSet.contains(endWord)) return 0;

        Queue<String> queue = new LinkedList<>();
        queue.offer(beginWord);
        int steps = 1;

        while (!queue.isEmpty()) {
```



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

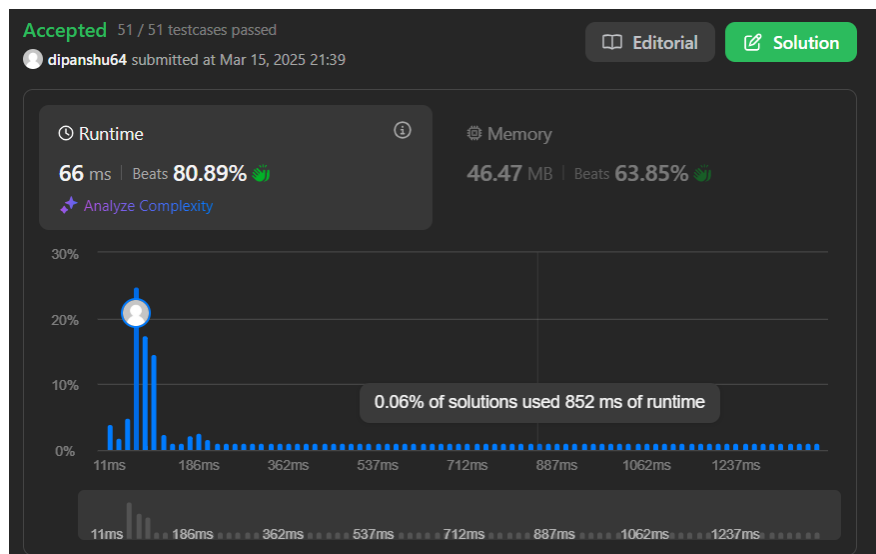
Discover. Learn. Empower.

```
int size = queue.size();
for (int i = 0; i < size; i++) {
    String word = queue.poll();
    char[] wordArray = word.toCharArray();

    for (int j = 0; j < wordArray.length; j++) {
        char originalChar = wordArray[j];
        for (char c = 'a'; c <= 'z'; c++) {
            wordArray[j] = c;
            String newWord = new String(wordArray);

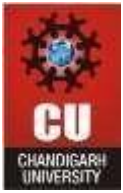
            if (newWord.equals(endWord)) return steps + 1;
            if (wordSet.contains(newWord)) {
                queue.offer(newWord);
                wordSet.remove(newWord);
            }
        }
        wordArray[j] = originalChar;
    }
    steps++;
}
return 0;
}
```

5. Output:



6. Time Complexity: $O(n * m * 26)$

7. Space Complexity: $O(n)$



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment 8(C)

1. **Title:** Surrounded Regions

2. **Objective:** To modify the given board such that all regions surrounded by 'X' are captured.

3.Algorithm:

- **Input:** A 2D character array board.
- **DFS Traversal:**
 - Perform DFS on all boundary 'O's and mark them as safe.
- **Conversion Step:**
 - Iterate over the board:
 - Change remaining 'O' to 'X'.
 - Change safe-marked 'S' back to 'O'.
- **Output:** Return the modified board.

5. Implementation/Code:

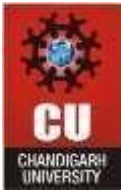
```
class Solution {
    public void solve(char[][] board) {
        int m = board.length, n = board[0].length;

        for (int i = 0; i < m; i++) {
            dfs(board, i, 0);
            dfs(board, i, n - 1);
        }

        for (int j = 0; j < n; j++) {
            dfs(board, 0, j);
            dfs(board, m - 1, j);
        }

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == 'O') board[i][j] = 'X';
                if (board[i][j] == 'S') board[i][j] = 'O';
            }
        }

        private void dfs(char[][] board, int i, int j) {
            if (i < 0 || j < 0 || i >= board.length || j >= board[0].length ||
                board[i][j] != 'O')
                return;
            board[i][j] = 'S';
```

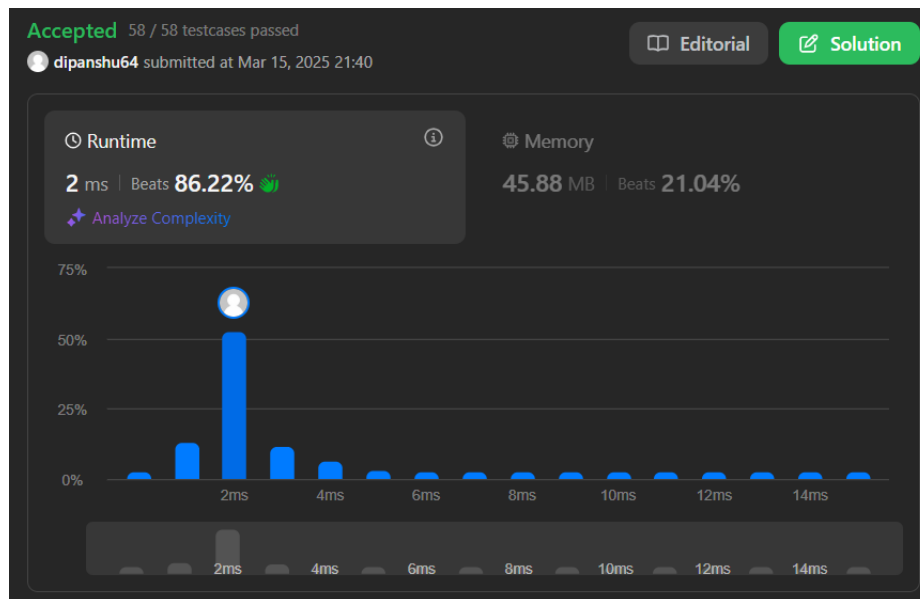


DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

```
        dfs(board, i + 1, j);  
        dfs(board, i - 1, j);  
        dfs(board, i, j + 1);  
        dfs(board, i, j - 1);  
    }  
}
```

6. Output:



8. Time Complexity: $O(m * n)$

9. Space Complexity: $O(m * n)$

10. Learning Outcomes:

- Learned effective strategies for marking visited nodes in a 2D matrix.
- Improved problem-solving skills using recursion for complex data structures.
- Learned BFS traversal for shortest path problems.
- Understood the efficient use of HashSet for quick lookups.
- Mastered DFS for connected component problems.
- Improved understanding of marking techniques in grid-based problems.