Experiment- 8A

Student Name: ANMOLPREET SINGH **UID: 22BCS15759**

Branch: BE-CSE **Section/Group: NTPP 602-A**

Semester:6TH **Date of Performance:16/03/25**

Subject Name: AP Lab-2 **Subject Code: 22CSH-352**

1. TITLE:

Number of islands.

2. AIM:

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

3. Algorithm

- **Iterate** through the grid, and when a land cell (`'1'`) is found, start a DFS/BFS traversal.
- **Mark all connected land cells as visited** (`'0'`) to avoid recounting.
- **Increase the island count** for each new DFS/BFS call and return the total count.

4. Implemetation/Code

```
class Solution {
public:
int numIslands(vector<vector<char>>& grid) {
constexpr int kDirs[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
const int m = grid.size();
const int n = grid[0].size();
int ans = 0;

auto bfs = [&](int r, int c) {
queue<pair<int, int>> q{{{r, c}}};
grid[r][c] = '2';  // Mark '2' as visited.
while (!q.empty()) {
const auto [i, j] = q.front();
q.pop();
for (const auto& [dx, dy] : kDirs) {
const int x = i + dx;
const int y = j + dy;
```

```
if (x < 0 || x == m || y < 0 || y == n)
continue;
if (grid[x][y] != '1')
continue;
q.emplace(x, y);
grid[x][y] = '2';  // Mark '2' as visited.
}
}
};

for (int i = 0; i < m; ++i)
for (int j = 0; j < n; ++j)
if (grid[i][j] == '1') {
bfs(i, j);
++ans;
}

return ans;
}
};
```

5. Output:

Time Complexity : O(M×N)

Space Complexity : O(M×N)

Learning Outcomes:-

- Understanding how DFS/BFS can be used to explore connected components in a grid.
- Marking visited cells ensures each island is counted only once.

Experiment - 8B

1. TITLE:

Word Ladder.

2. AIM:

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s_1 -> s_2 -> ... -> s_k.

3. Algorithm

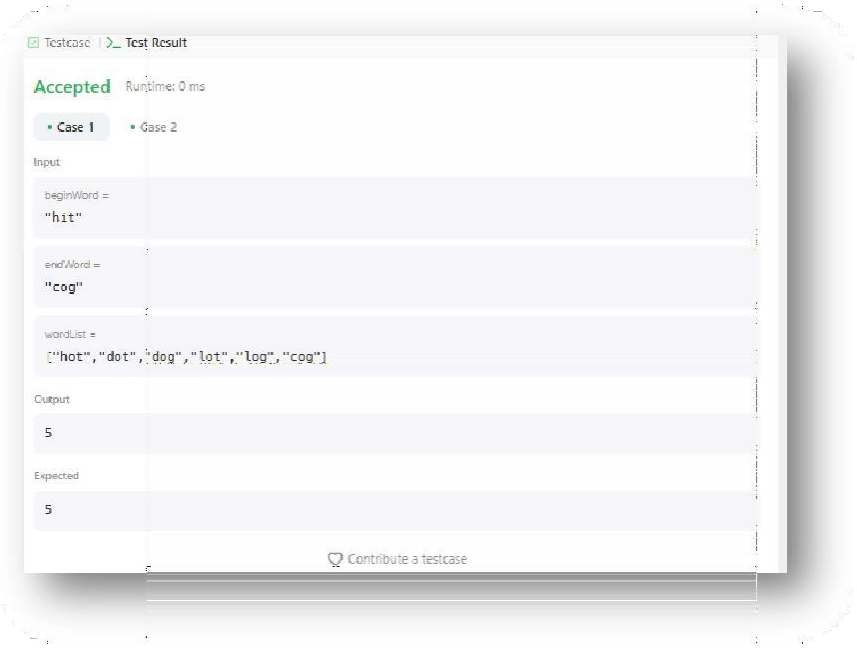
- BFS ensures the shortest path is found first.
- Try replacing each letter in `beginWord` with `'a'` to `'z'` and check if the new word exists in `wordList`.
- Use a queue to store `(word, transformation_steps)`.

4. Implemetation/Code:

```
class Solution {
public:
int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
unordered_set<string> wordSet(wordList.begin(), wordList.end());
if (!wordSet.contains(endWord))
return 0;

queue<string> q{{beginWord}};

for (int step = 1; !q.empty(); ++step)
for (int sz = q.size(); sz > 0; --sz) {
string word = q.front();
q.pop();
for (int i = 0; i < word.length(); ++i) {
const char cache = word[i];
for (char c = 'a'; c <= 'z'; ++c) {
word[i] = c;
```

```
if (word == endWord)
return step+1;
if (wordSet.contains(word)) {
q.push(word);
wordSet.erase(word);
}
}
word[i] = cache;
}
}

return 0;
}
};
```

5. Output:



6. Time Complexity : O(M×N)

7. Space Complexity : O(N)

8. Learning Outcomes:-

- Breadth-First Search efficiently finds the shortest transformation sequence.
- Generating all possible words by changing one letter at a time.