

EXPERIMENT 9

Name: Vatsala Singh

UID: 22BCS10028

Section/Group: 605-A

Ques- Number of Islands

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

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

```
class Solution { public:

void dfs(vector<vector<char>>& grid, int i, int j) {

// Check for out-of-bound or water

if (i < 0 || i >= grid.size() || j < 0 || j >= grid[0].size() || grid[i][j] == '0')

return;

// Mark the cell as visited (turn land to water)

grid[i][j] = '0';

// Visit all 4 adjacent cells (up, down, left, right)

dfs(grid, i + 1, j);    dfs(grid, i - 1, j);

dfs(grid, i, j + 1);    dfs(grid, i, j - 1);

}

int numIslands(vector<vector<char>>& grid) {

int count = 0;

for (int i = 0; i < grid.size(); i++) {

for (int j = 0; j < grid[0].size(); j++) {

if (grid[i][j] == '1') {
```

```

count++;          // Found an island

dfs(grid, i, j);  // Mark all connected land

}

}

}

return count;

}

};

```

The screenshot displays the LeetCode submission page for the "Number of Islands" problem. The left sidebar shows the submission status as "Accepted" with 49/49 test cases passed. The runtime performance is highlighted as 23 ms, beating 83.82% of other submissions. A memory usage of 16.36 MB is also shown. The main area contains the C++ code for the solution, which implements a DFS algorithm to explore each island in the grid. The code defines a `Solution` class with a `dfs` method to mark visited cells and a `numIslands` method to count the total number of islands. The bottom of the screen shows a test case input grid.

```

class Solution {
public:
    void dfs(vector<vector<char>>& grid, int i, int j) {
        // Check for out-of-bound or water
        if (i < 0 || i >= grid.size() || j < 0 || j >= grid[0].size() || grid[i][j] == '0')
            return;

        // Mark the cell as visited (turn land to water)
        grid[i][j] = '0';

        // Visit all 4 adjacent cells (up, down, left, right)
        dfs(grid, i + 1, j);
        dfs(grid, i - 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i, j - 1);
    }

    int numIslands(vector<vector<char>>& grid) {
        int count = 0;
    }
};

```

Ques- Course Schedule Problem:

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1.

Some courses may have prerequisites, given as a list of prerequisites where prerequisites[i] = [a, b] means you must take course b before course a.

Return true if you can finish all courses. Otherwise, return false.

```
#include <iostream>

#include <vector>

#include <queue>

using namespace std;

bool canFinish(int numCourses, vector<vector<int>>&
prerequisites) {    vector<vector<int>> adj(numCourses);
vector<int> indegree(numCourses, 0);

for (auto& p : prerequisites) {
adj[p[1]].push_back(p[0]);
indegree[p[0]]++;
}

queue<int> q;    for (int i = 0; i <
numCourses; i++) {        if
(indegree[i] == 0) q.push(i);
}

int count = 0;

while (!q.empty()) {

int curr = q.front();
```

```

q.pop();

count++;

for (int neighbor : adj[curr]) {
    indegree[neighbor]--;      if
    (indegree[neighbor] == 0)
    q.push(neighbor);
}

}

return count == numCourses;

}

int main() {    int numCourses = 2;

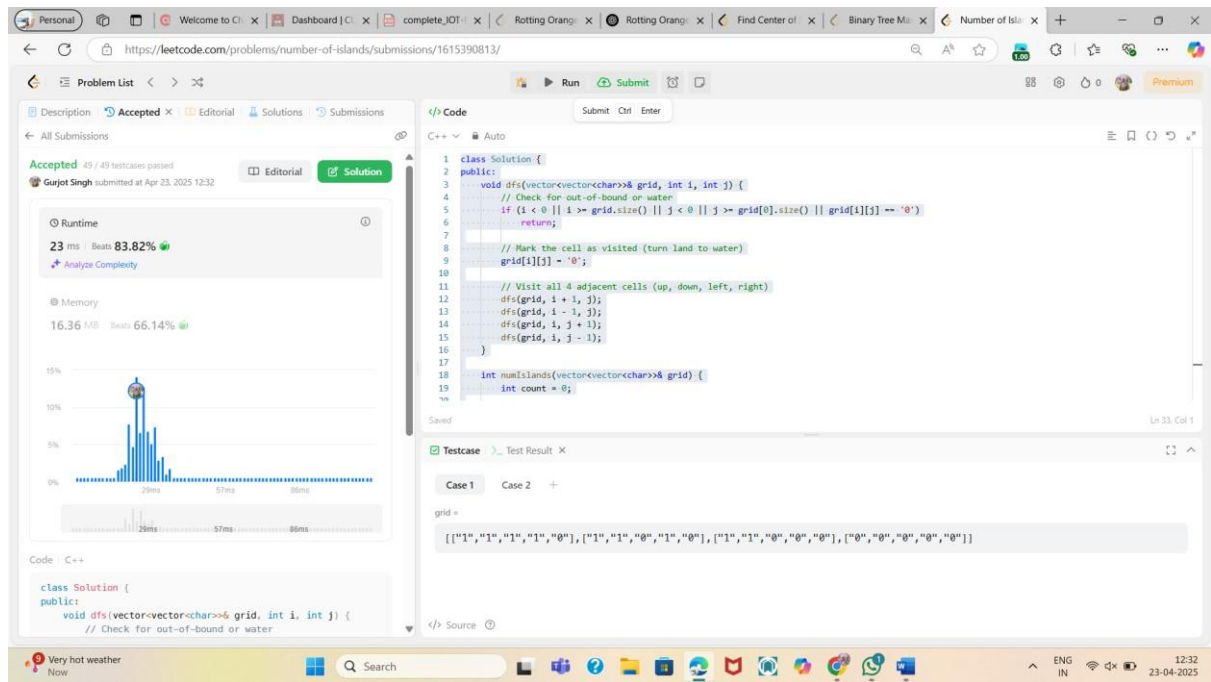
    vector<vector<int>> prerequisites = {{1, 0}};

    cout << (canFinish(numCourses, prerequisites) ? "true" : "false") << endl;

    return 0;

}

```



Ques- Longest Increasing Path in a Matrix Problem:

Given an $m \times n$ integers matrix, return the length of the longest increasing path in the matrix.

From each cell, you can move in 4 directions (up, down, left, right). You may not move diagonally or move outside the boundary.

cpp

CopyEdit

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace
```

```
std;
```

```
class Solution { public:    int
```

```
longestIncreasingPath(vector<vector<int>>& matrix) {
```

```
int m = matrix.size();    int n = matrix[0].size();
```

```
vector<vector<int>> dp(m, vector<int>(n, 0));    int
maxLen = 0;
```

```
for (int i = 0; i < m; i++) {        for (int j = 0; j <
n; j++) {        maxLen = max(maxLen,
dfs(matrix, dp, i, j));
}
}
return maxLen;
}
```

```
private:    vector<int> dir = {-1,
0, 1, 0, -1};
```

```
int dfs(vector<vector<int>>& matrix, vector<vector<int>>& dp, int i, int j) {
if (dp[i][j] != 0) return dp[i][j];
```

```
int maxPath = 1;    for
(int d = 0; d < 4; d++) {
int x = i + dir[d];        int
y = j + dir[d + 1];
```

```
if (x >= 0 && x < matrix.size() &&        y >= 0 &&
y < matrix[0].size() &&        matrix[x][y] >
matrix[i][j]) {        maxPath = max(maxPath, 1 +
dfs(matrix, dp, x, y));
}
```

```
}
```

```
dp[i][j] = maxPath;
```

```
return maxPath;
```

```
}
```

```
};
```

```
int main() {    Solution sol;
```

```
vector<vector<int>> matrix =
```

```
{
```

```
{9, 9, 4},
```

```
{6, 6, 8},
```

```
{2, 1, 1}
```

```
};
```

```
cout << sol.longestIncreasingPath(matrix) << endl;
```

```
return 0;
```

```
}
```

The screenshot displays a LeetCode submission for the "Number of Islands" problem. The interface includes a navigation bar at the top with tabs for "Description", "Accepted", "Editorial", "Solutions", and "Submissions". The "Accepted" tab is active, showing a green checkmark and the text "Accepted 49 / 49 testcases passed". Below this, a runtime graph shows the performance of the solution, with a peak at 23ms and a memory usage of 16.36 MB. The C++ code is displayed in a text editor, showing a class `Solution` with a method `dfs` that recursively visits all 4 adjacent cells (up, down, left, right) and marks the cell as visited. The main function `main` initializes a matrix and calls the `dfs` method to count the number of islands.

```
1 class Solution {
2 public:
3     void dfs(vector<vector<char>>& grid, int i, int j) {
4         // Check for out-of-bound or water
5         if (i < 0 || i >= grid.size() || j < 0 || j >= grid[0].size() || grid[i][j] == '0')
6             return;
7
8         // Mark the cell as visited (turn land to water)
9         grid[i][j] = '0';
10
11        // Visit all 4 adjacent cells (up, down, left, right)
12        dfs(grid, i + 1, j);
13        dfs(grid, i - 1, j);
14        dfs(grid, i, j + 1);
15        dfs(grid, i, j - 1);
16    }
17
18    int numIslands(vector<vector<char>>& grid) {
19        int count = 0;
20        for (int i = 0; i < grid.size(); i++)
21            for (int j = 0; j < grid[0].size(); j++)
22                if (grid[i][j] == '1')
23                    count++;
24        return count;
25    }
26 }
```