MEDIUM

Number of Enclaves

Intuition

0 - Sea Cell
1 - Land Cell

Allowed to move in 4 direction

0	0	0	**1**
0	1	1	0
0	1	1	0
0	0	0	**1**
0	**1**	**1**	0

1 - Able to move out of the boundary
1 - Not able to move out of the boundary

Question asks us to count number of lands/1s where we cannot move out of the boundary

Eg.

0	0	**1**	**1**
0	**1**	**1**	0
0	**1**	**1**	0
0	0	0	**1**
0	**1**	**1**	0

It will return 0 as all the 1 are able to move out of the boundary using the 4 moves

If a 1 is connected to a boundary it will never be our answer and we will again be able to solve it the way we were able to solve the **G-14** problem.

We will traverse the boundary and mark all the 1s which are connected to the boundary then the remaining unmarked are our answer

Eg.

0	0	0	**1**	**1**
0	0	**1**	**1**	0
0	1	0	0	0
0	1	1	0	0

0 0 0 **1** **1**

Output : 3

DFS intuition is same as the problem G-14

BFS intuition

Create a corresponding visited array and perform all functions over it
Mark the nodes which are 1 at the boundary and add all those in the initially empty queue

Original Matrix

0	1	2	3	4
0	0	0	1	1
0	0	1	1	0
0	1	0	0	0
0	1	1	0	0
0	0	0	1	1

Visited Matrix

0	0	0	1	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	1	1

Queue = [[1,3],[1,4],[4,3],[4,4]] //initial configuration

[[1,4],[4,3],[4,4],[2,3]]

[[4,3],[4,4],[2,3].........] will continue until the queue becomes empty

Approach

We can use the BFS as well as the DFS algorithm

DFS :

NumberOfEnclaves()

- Declare :
    - Visited array having all elements as 0 of size n*m
    - Traverse through the first row and last row for all columns :
        - Make a dfs call as dfs(row,col,visited,grid)
    - Traverse through the first row and last row for all columns :
        - Make a dfs call as dfs(row,col,visited,grid)
    - Initialize a count variable counter with 0
    - Traverse for all the elements of grid and check if the element is unvisited and is 1 then increment the counter for such a variable

DFS()

- Mark the source node as 1
- Traverse in the 4 possible movement direction :
    - Calculate the rows and columns of the neighboring elements
    - Check for the validity of the neighboring elements
        - Check if they are unvisited and actually 1
            - Make a dfs call as dfs(nrow,ncol,visited,grid)

BFS :

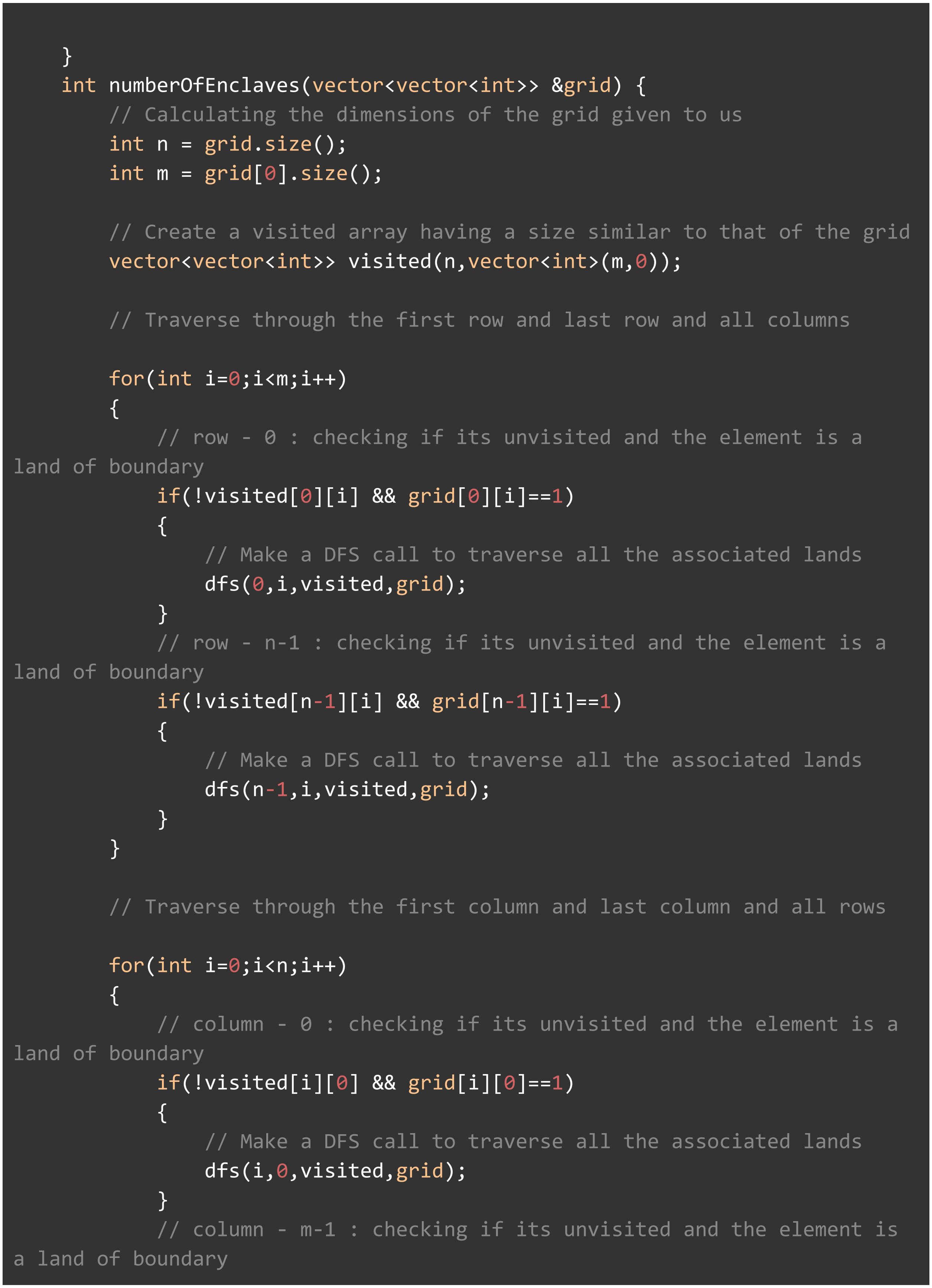
- Declare :
    - Visited array of size m*n having initially all elements as 0
    - An empty queue
- Traverse through the first row and the last row and all columns :
    - If the elements are not visited and are 1
        - Push them to queue
- Traverse through the first row and the last row and all columns :
    - If the elements are not visited and are 1
        - Push them to queue
- Traverse until the queue becomes empty :
    - Extract the first node in the queue
    - Mark the node as visited
    - Pop the node
    - Traverse in the 4 possible directions given from that node :
        - Calculate the rows and columns of neighboring elements
        - Check for the validity of coordinates
            - Check if they are visited or not and they are actually 1
                - Push them to the queue
- Initialize a count variable counter with 0

- Traverse for all the elements of grid and check if the element is unvisited and is 1 then increment the counter for such a variable

Function Code :

DFS :

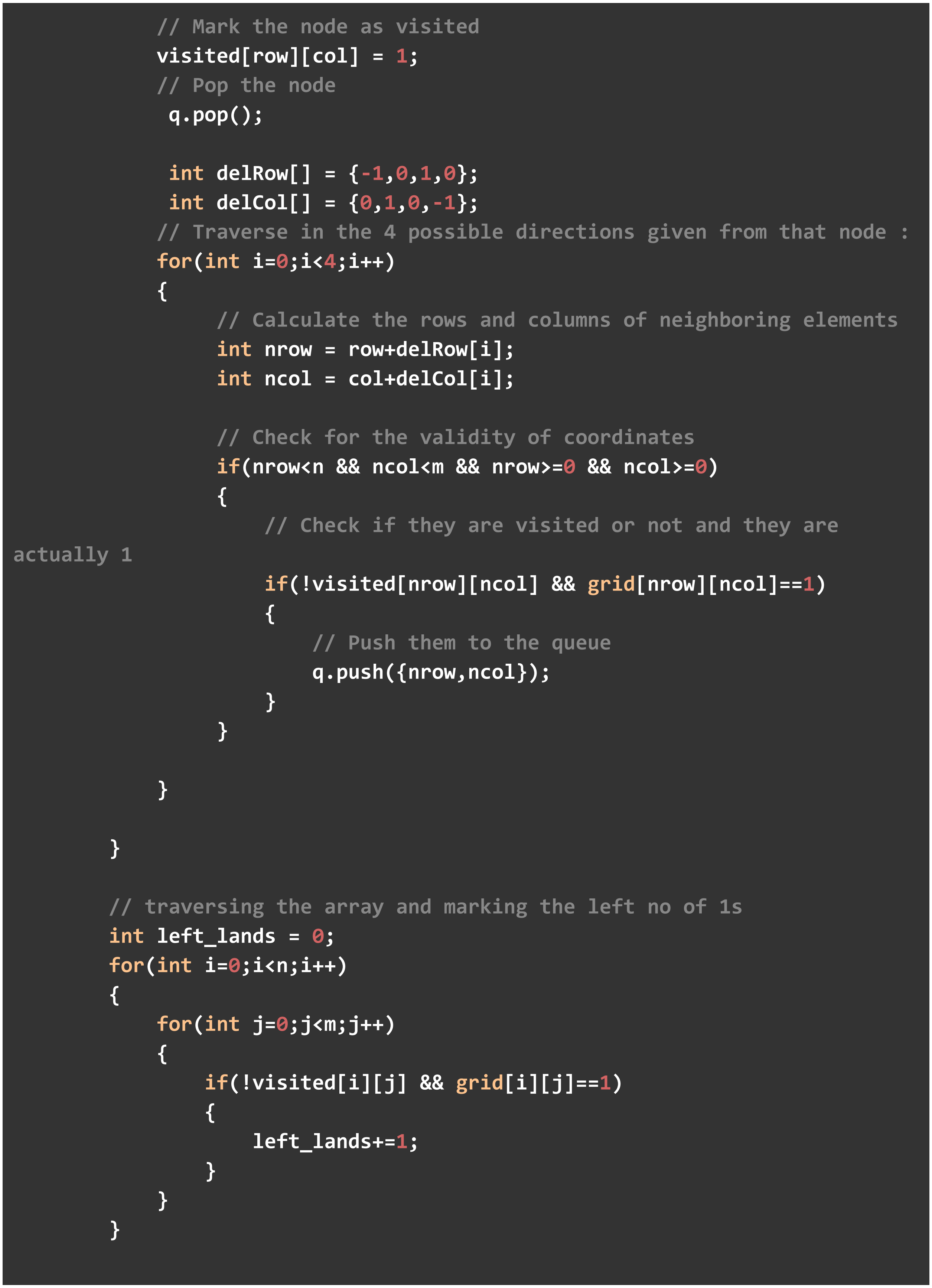
```
void dfs(int row,int col,vector<vector<int>> &visited,vector<vector<int>>
&grid)
    {
        //calculating the dimensions of the grid

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

        //marking the source element as visited
        visited[row][col]=1;

        //Able to move in the 4 directions
        int delRow[] = {-1,0,1,0};
        int delCol[] = {0,1,0,-1};

        for(int i=0;i<4;i++)
        {
            //calculating the neighboring rows and columns
            int nrow = row+delRow[i];
            int ncol = col+delCol[i];

            // checking for the validity of the neighbor row and column
            if(nrow<n && ncol<m && nrow>=0 && ncol>=0)
            {
                //checking if the land is not already traversed and if this
is a land actually
                if(!visited[nrow][ncol] && grid[nrow][ncol]==1)
                {
                    //make a dfs call for marking the connected neighbor
nodes
                    dfs(nrow,ncol,visited,grid);
                }
            }
        }
```



```
    }
    int numberOfEnclaves(vector<vector<int>> &grid) {
        // Calculating the dimensions of the grid given to us
        int n = grid.size();
        int m = grid[0].size();

        // Create a visited array having a size similar to that of the grid
        vector<vector<int>> visited(n,vector<int>(m,0));

        // Traverse through the first row and last row and all columns

        for(int i=0;i<m;i++)
        {
            // row - 0 : checking if its unvisited and the element is a land of boundary
            if(!visited[0][i] && grid[0][i]==1)
            {
                // Make a DFS call to traverse all the associated lands
                dfs(0,i,visited,grid);
            }
            // row - n-1 : checking if its unvisited and the element is a land of boundary
            if(!visited[n-1][i] && grid[n-1][i]==1)
            {
                // Make a DFS call to traverse all the associated lands
                dfs(n-1,i,visited,grid);
            }
        }

        // Traverse through the first column and last column and all rows

        for(int i=0;i<n;i++)
        {
            // column - 0 : checking if its unvisited and the element is a land of boundary
            if(!visited[i][0] && grid[i][0]==1)
            {
                // Make a DFS call to traverse all the associated lands
                dfs(i,0,visited,grid);
            }
            // column - m-1 : checking if its unvisited and the element is a land of boundary
            if(!visited[i][m-1] && grid[i][m-1]==1)
            {
                // Make a DFS call to traverse all the associated lands
                dfs(i,m-1,visited,grid);
            }
        }

        // traversing the array and marking the left no of 1s
        int left_lands = 0;
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<m;j++)
            {
                if(!visited[i][j] && grid[i][j]==1)
                {
                    left_lands+=1;
                }
            }
        }

        return left_lands;

    }
```

BFS :

```
int numberOfEnclaves(vector<vector<int>> &grid) {

        //calculating the dimensions of the grid
        int n = grid.size();
        int m = grid[0].size();
        //          Declare :
        // Visited array of size m*n having initially all elements as 0
        vector<vector<int>> visited(n,vector<int>(m,0));
        // An empty queue
        queue<pair<int, int>> q;
        // Traverse through the first row and the last row and all columns
:
        // Traverse through the first row and last row and all columns

        for(int i=0;i<m;i++)
        {
            // row - 0 : checking if its unvisited and the element is a 
land of boundary
            if(!visited[0][i] && grid[0][i]==1)
            {
                // push the elements to the queue
                q.push({0,i});
            }
            // row - n-1 : checking if its unvisited and the element is a 
land of boundary
            if(!visited[n-1][i] && grid[n-1][i]==1)
            {
                // Make a DFS call to traverse all the associated lands
                q.push({n-1,i});
            }
        }

        // Traverse through the first column and last column and all rows

        for(int i=0;i<n;i++)
        {
            // column - 0 : checking if its unvisited and the element is a 
land of boundary
            if(!visited[i][0] && grid[i][0]==1)
            {
                // Make a DFS call to traverse all the associated lands
               q.push({i,0});
            }
            // column - m-1 : checking if its unvisited and the element is 
a land of boundary
            if(!visited[i][m-1] && grid[i][m-1]==1)
              {
                // Make a DFS call to traverse all the associated lands
               q.push({i,m-1});
            }
        }

        // Traverse until the queue becomes empty :
        while(!q.empty())
        {
            // Extract the first node in the queue
            int row = q.front().first;
            int col = q.front().second;
```



```
            // Mark the node as visited
            visited[row][col] = 1;
            // Pop the node
             q.pop();

             int delRow[] = {-1,0,1,0};
             int delCol[] = {0,1,0,-1};
            // Traverse in the 4 possible directions given from that node :
            for(int i=0;i<4;i++)
            {
                 // Calculate the rows and columns of neighboring elements
                 int nrow = row+delRow[i];
                 int ncol = col+delCol[i];

                 // Check for the validity of coordinates
                 if(nrow<n && ncol<m && nrow>=0 && ncol>=0)
                 {
                     // Check if they are visited or not and they are
actually 1
                     if(!visited[nrow][ncol] && grid[nrow][ncol]==1)
                     {
                         // Push them to the queue
                         q.push({nrow,ncol});
                     }
                 }

            }

        }

        // traversing the array and marking the left no of 1s
        int left_lands = 0;
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<m;j++)
            {
                if(!visited[i][j] && grid[i][j]==1)
                {
                    left_lands+=1;
                }
            }
        }
        return left_lands;
    }
```

Time Complexity

DFS :

O(n*m)

BFS :

O(n*m)