1020. Number of Enclaves

Depth-First Search Breadth-First Search Union Find

Problem Description

Medium

You have a matrix where each cell can either be land (1) or sea (0). A move is defined as walking from one land cell to an adjacent land cell. Adjacent means above, below, to the left, or to the right (but not diagonally). The edge of the matrix is considered as a boundary that one can walk off from. The problem asks you to find out how many land cells are there in the matrix such that it's not possible to walk off the edge of the matrix starting from any of these cells; in other words, these land cells are enclosed by other land cells or by the matrix boundaries. This can be visualized as counting the number of land cells from which you cannot reach the sea by moving only horizontally or vertically on the land cells.

Array

Matrix

To solve this problem, we can make use of a <u>Depth-First Search</u> (DFS) approach. The key intuition is to identify and eliminate the

Intuition

We start by iterating over the boundary rows and columns of the grid. If there's a land cell on the boundary, it means that this land cell can directly lead to the sea. But we also need to consider the land cells connected to it. Thus, we use DFS starting from

land cells that can lead to the boundary (and hence, to the sea), so that what remains will be our "enclosed" cells.

this land cell on the boundary to traverse and "sink" all connected land cells, marking them as sea cells (by setting them to 0). "Sinking" in this context means we are marking the cell as visited and ensuring we won't count it later as an enclosed land cell. DFS is continued until all cells that can reach the boundary are marked as sea cells. After marking these cells, the remaining land cells in the grid are guaranteed to be enclosed as they cannot reach the boundary.

Lastly, we count and return the number of cells left with a 1, which represents the number of enclosed land cells. For this, we simply iterate through the entire grid and sum up the values.

Solution Approach The solution utilizes a Depth-First Search (DFS) algorithm to explore and manipulate the grid. Here's a step-by-step explanation

Define the dfs function, which will be used to mark the land cells that can lead to the boundary as sea cells. This function takes the current coordinates (i, j) of a land cell as arguments.

of how it works:

Inside the dfs function, the current land cell is first "sunk" by setting grid[i][j] to 0. The function then iterates over the adjacent cells in the 4 cardinal directions using a tuple dirs that contains the relative

coordinates to move up, down, left, and right. The pairwise pattern (a common programming idiom) is used to retrieve directions in pairs. However, in this particular solution code, it seems they have made an error by referring to pairwise (dirs),

which should have been a traversal through the directions one by one; instead, this should be replaced with code that

iterates over each pair of directions in dirs, something akin to (dirs[i], dirs[i + 1]) for i in range(len(dirs) - 1).

- It checks if the new coordinates (x, y) are within the grid and if the cell is a land cell. If these conditions are met, DFS is recursively called on that cell.
- Before starting the DFS, the main part of the function initializes the dimensions of the grid m and n. The function then iterates over the cells on the boundaries of the grid - this is done by iterating through the rows and columns at the edges.
- For each boundary cell that is land (has value 1), the DFS is called to "sink" the cell and any connected land cells. After DFS traversal, all the cells that could walk off the boundary will be marked as 0.
- Finally, the function counts and returns the number of cells with a value of 1, which now represent only the enclosed cells, by summing up all the values in the grid.

by other land cells or the borders of the matrix and thus cannot reach the edge.

edge and mark it as 0. We do the same with other boundary 1s.

1. We enumerate the boundary cells. These would be cells in the first and last rows and columns.

doesn't use any additional complex data structures. The DFS pattern is crucial for this solution, as it enables systematically visiting and modifying cells related to the boundary cells.

library's context and is not suitable for the way directions are used. It would need to be corrected for the code to function as

Please note that the actual code given has a mistake in the usage of pairwise, which isn't actually defined in the standard Python

The data structure used here is the grid itself, which is a 2D list (a list of lists in Python), which is modified in place. The solution

Example Walkthrough Let's consider a 5×5 matrix as a small example to illustrate the solution approach:

1 0 1 0 0 1 1 1 0 1 1 0 0 1 1 In this matrix, 1 indicates land, and 0 represents sea. We want to find out the number of land cells that are completely surrounded

2. As we check these cells, we find 1s at (0,0), (0,4), (4,3), and (4,4) on the corners and some other on the edges. We know these cannot be enclosed as they are on the matrix edge.

Following the solution approach:

intended.

1 1 1 0 1

The matrix after sinking the boundary land cells is:

3. We begin our DFS with each of these boundary 1s to mark the connected land cells as 0 (sea). Starting with (0,0), we find it's already on the

All reachable cells have been "sunk". The cand cells at (1,0), (1,1), (1,2), (2,0), (2,2), (3,0), (3,1), and (3,2) remain as 1s. Now, we only count the 1s not on the boundary.

We iterate over the remaining cells to count the number of land cells that are enclosed. From our matrix above, the count is 8.

```
Through this DFS approach, we have effectively identified and eliminated any land cell connected to the boundary. The remaining
  land cells are the enclosed cells that we are interested in counting.
Solution Implementation
```

Python

0 0 0 0

1 1 1 0 0

1 0 1 0 0

1 1 1 0 0

1 0 0 0 0

class Solution: def numEnclaves(self, grid: List[List[int]]) -> int: # Helper function to perform depth-first search and mark visited land cells as water def dfs(row, col):

grid[row][col] = 0 # Mark the current cell as visited by setting it to 0 (water)

Check if the new cell is within the bounds of the grid and is land

num_rows, num_cols = len(grid), len(grid[0]) # Get the dimension of the grid

Start DFS for the boundary cells that are land (i.e., have a value of 1)

dfs(new_row, new_col) # Recursively apply DFS to the new cell

if 0 <= new_row < num_rows and 0 <= new_col < num_cols and grid[new_row][new_col]:</pre>

if grid[row][col] and (row == 0 or row == num_rows - 1 or col == 0 or col == num_cols - 1):

Loop through the four directions (up, right, down, left) for direction in range(4): new_row = row + d_row[direction] new_col = col + d_col[direction]

Directions for upward, rightward, downward, and leftward movement $d_{row} = [-1, 0, 1, 0]$ $d_{col} = [0, 1, 0, -1]$

for row in range(num_rows):

for col in range(num_cols):

grid[i][j] = 0; // Mark the current cell as water

int x = i + directions[k]; // Compute the next row number

dfs(x, y); // Continue DFS from the next cell

int y = j + directions[k + 1]; // Compute the next column number

if $(x >= 0 \&\& x < rows \&\& y >= 0 \&\& y < columns \&\& grid[x][y] == 1) {$

// Check if the next cell is within the grid and is a land cell

// Explore all 4 possible directions

#include <functional> // Needed for std::function

int directions $[5] = \{-1, 0, 1, 0, -1\};$

for (int i = 0; i < numRows; ++i) {</pre>

int enclaveCount = 0;

for (auto& row : grid) {

// Explore all four directions.

const nextRow = row + directions[k];

const nextCol = col + directions[k + 1];

for (let k = 0; k < 4; ++k) {

for (int j = 0; j < numCols; ++j) {</pre>

depthFirstSearch(i, j);

int numEnclaves(std::vector<std::vector<int>>& grid) {

// Direction vectors for exploring adjacent cells.

int numRows = grid.size(); // Number of rows in grid

int numCols = grid[0].size(); // Number of columns in grid

grid[i][j] = 0; // Flip the current cell to water

// DFS function to explore and flip land cells (1's) into water cells (0's)

for (int k = 0; k < 4; ++k) { // Explore all four adjacent cells

if $(x >= 0 \&\& x < numRows \&\& y >= 0 \&\& y < numCols && grid[x][y]) {$

depthFirstSearch(x, y); // Continue DFS if adjacent cell is land

// Check if the current cell is land and at the border, then start DFS

// After removing enclaves touching the border, count remaining enclaved lands

if (grid[i][j] && (i == 0 || i == numRows - 1 || j == 0 || j == numCols - 1)) {

std::function<void(int, int)> depthFirstSearch = [&](int i, int j) {

int x = i + directions[k], y = j + directions[k + 1];

// First pass to remove all land regions touching the grid boundaries

// Check for bounds of grid and if the cell is land

for (int k = 0; k < 4; ++k) {

```
dfs(row, col) # Remove enclaves touching the boundary by DFS
       # Count the remaining land cells that are enclaves (not touching the boundaries)
       return sum(value for row in grid for value in row)
Java
class Solution {
    private int rows; // Total number of rows in the grid
   private int columns; // Total number of columns in the grid
   private int[][] grid; // The grid representing the land and water
   // This method calculates the number of enclaves.
   public int numEnclaves(int[][] grid) {
       this.grid = grid; // Assigns the input grid to the instance variable grid
       rows = grid.length; // Sets the number of rows
       columns = grid[0].length; // Sets the number of columns
       // Iterate over the boundary cells of the grid
       for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < columns; ++j) {</pre>
               // On finding a land cell on the boundary, initiate DFS
               if (grid[i][j] == 1 && (i == 0 || i == rows - 1 || j == 0 || j == columns - 1)) {
                    dfs(i, j); // Call DFS to mark the connected land as water (0)
       // Count the land cells (value 1) that are not connected to the boundary
       int enclaveCount = 0;
       for (int[] row : grid) {
            for (int value : row) {
                enclaveCount += value;
       return enclaveCount; // Return the number of cells in enclaves
   // DFS method to convert connected land cells to water cells
   private void dfs(int i, int j) {
```

int[] directions = $\{-1, 0, 1, 0, -1\}$; // Array representing the 4 directions (up, right, down, left)

C++

public:

#include <vector>

class Solution {

};

```
for (auto& cell : row) {
                enclaveCount += cell; // Cell will be 1 if it's land not touching the border
        return enclaveCount; // Return the final count of enclaved lands
};
TypeScript
function numEnclaves(grid: number[][]): number {
    // Get the number of rows and columns in the grid.
    const rows = grid.length;
    const cols = grid[0].length;
    // Directions array represents the four possible movements (up, right, down, left).
    const directions = [-1, 0, 1, 0, -1];
   // Depth-first search function to mark land (1's) connected to the borders as water (0's).
    const dfs = (row: number, col: number) => {
       // Mark the current land piece as visited by setting it to 0.
       grid[row][col] = 0;
```

```
// Check if the new coordinates are in bounds and if there is land to visit.
              if (nextRow >= 0 && nextRow < rows && nextCol >= 0 && nextCol < cols && grid[nextRow][nextCol] === 1) {</pre>
                  dfs(nextRow, nextCol);
      };
      // Run the DFS for lands connected to the borders to eliminate those.
      for (let row = 0; row < rows; ++row) {</pre>
          for (let col = 0; col < cols; ++col) {</pre>
              // If the cell is land and it's on the border, start the DFS.
              if (grid[row][col] === 1 && (row === 0 || row === rows - 1 || col === 0 || col === cols - 1)) {
                  dfs(row, col);
      // Count the number of enclaved land pieces remaining in the grid.
      let enclaveCount = 0;
      for (const row of grid) {
          for (const cell of row) {
              enclaveCount += cell;
      // Return the count of enclaves, which are land pieces not connected to the border.
      return enclaveCount;
class Solution:
    def numEnclaves(self, grid: List[List[int]]) -> int:
        # Helper function to perform depth-first search and mark visited land cells as water
        def dfs(row, col):
            grid[row][col] = 0 # Mark the current cell as visited by setting it to 0 (water)
            # Loop through the four directions (up, right, down, left)
            for direction in range(4):
                new_row = row + d_row[direction]
                new_col = col + d_col[direction]
                # Check if the new cell is within the bounds of the grid and is land
                if 0 <= new_row < num_rows and 0 <= new_col < num_cols and grid[new_row][new_col]:</pre>
                    dfs(new_row, new_col) # Recursively apply DFS to the new cell
        num_rows, num_cols = len(grid), len(grid[0]) # Get the dimension of the grid
        # Directions for upward, rightward, downward, and leftward movement
        d_{row} = [-1, 0, 1, 0]
        d_{col} = [0, 1, 0, -1]
        # Start DFS for the boundary cells that are land (i.e., have a value of 1)
        for row in range(num_rows):
            for col in range(num_cols):
                if grid[row][col] and (row == 0 or row == num_rows - 1 or col == 0 or col == num_cols - 1):
                    dfs(row, col) # Remove enclaves touching the boundary by DFS
        # Count the remaining land cells that are enclaves (not touching the boundaries)
        return sum(value for row in grid for value in row)
```

of '1's not connected to the grid's border.

Time and Space Complexity

Time Complexity: The time complexity of the code is 0(m * n) where m is the number of rows and n is the number of columns in the grid. Each cell

in the grid is processed at most once. The DFS is started from each border cell that contains a '1' and marks all reachable '1's as

The given code performs a Depth-First Search (DFS) on a two-dimensional grid to find the number of enclaves, which are regions

'0's (meaning they're visited). Since each cell can be part of only one DFS call (once it has been visited, it's turned to '0' and won't be visited again), each cell contributes only a constant amount of time. Therefore, the time complexity is proportional to the total

number of cells in the grid.

Space Complexity: The space complexity of the DFS is 0(m * n) in the worst case, which happens when the grid is filled with '1's, and hence the call stack can grow to the size of the entire grid in a worst-case scenario of having a single large enclave. However, the function modifies the grid in-place and does not use any additional space proportional to the grid size, except for the recursion call stack. Thus the space complexity is determined by the depth of the recursion, which is 0(m * n) in this worst case.