463. Island Perimeter

Easy Depth-First Search Breadth-First Search Array Matrix

Leetcode Link

Problem Description

In this problem, you are presented with a 2D grid that represents a map, where the value `1` indicates land and the value `0` represents water. The key points about the map are:

- There's exactly one island, and it is made up of land cells (`1`'s) that are connected horizontally or vertically.
- The surrounding cells outside the grid are water (`0`'s).
- There are no "lakes," meaning there are no enclosed areas of water within the island.
- Each cell of the grid is a square with a side length of 1.
- The dimensions of the grid will not be larger than 100×100.

Your task is to determine and return the perimeter of the island in this grid. Remember that the perimeter counts the boundary that separates land cells from water cells.

Intuition

To solve this problem, we need to calculate the total perimeter contributed by each land cell in the grid. Since we're working with a grid that has connected cells horizontally and vertically, each land cell that does not touch another land cell contributes `4` units to the perimeter (as it has four sides).

Here's the intuitive step-by-step approach:

1. Initialize a perimeter count to 0.
2. Iterate through each cell in the grid.
3. If a cell is land (`1`), increment the perimeter count by `4` (all possible sides of a single cell).
4. Then, check the adjacent cells:
   - If there is a land cell to the right (horizontally adjacent), the shared edge does not contribute to the perimeter, so we subtract `2` from the perimeter count (as it removes one edge from each of the two adjacent land cells).
   - Similarly, if there is a land cell below (vertically adjacent), subtract `2` for the shared edge.
5. Continue this process for all land cells in the grid.
6. Return the total perimeter count.

This approach works because it dynamically adjusts the perimeter count based on the land cell's adjacency with other land cells, ensuring that shared edges are only counted once.

Solution Approach

The solution approach for determining the perimeter of the island adheres to the following algorithmic steps, aligning with the explained intuition:

1. **Initiate a Counter for Perimeter:** Start with a variable `ans`, initialized to 0, which will keep track of the island's perimeter.
2. **Iterate over Grid Cells:** Use a nested loop to go through every cell in the grid. Let `m` be the number of rows and `n` be the number of columns of the grid:

   ```
   1  for i in range(m):
   2      for j in range(n):
   ```

3. **Check for Land Cells:** If the current cell `grid[i][j]` is a land cell (`1`), increment the perimeter counter by 4:

   ```
   1  if grid[i][j] == 1:
   2      ans += 4
   ```

   This is because a land cell has 4 potential edges contributing to the perimeter.
4. **Check for Adjacent Land:** Determine if the land cell has adjacent land cells that would reduce the perimeter:
   - To check the cell below the current cell (if it exists and is a land cell), we perform:

     ```
     1  if i < m - 1 and grid[i + 1][j] == 1:
     2      ans -= 2
     ```

     We use the condition `i < m - 1` to ensure we're not on the bottom most row before checking the cell below.
   - To check the cell to the right of the current cell (if it exists and is a land cell), we perform:

     ```
     1  if j < n - 1 and grid[i][j + 1] == 1:
     2      ans -= 2
     ```

     We use the condition `j < n - 1` to ensure we're not on the right most column before checking the cell to the right.
5. **Subtract Shared Edges:** The subtraction of `2` from the perimeter `ans` in the case of adjacent land cells accounts for the fact that each shared edge is part of the perimeter of two adjacent cells. Since this edge cannot be counted twice, we subtract `2` from our total perimeter count — 1 for each of the two cells sharing the edge.
6. **Return Perimeter Count:** After the entire grid has been processed, return the calculated perimeter `ans`.

The algorithm makes use of nested loops to process a 2D matrix, while the main data structure utilized is the 2D list given as input. This approach is straightforward with a linear runtime that corresponds to the size of the grid ($O(mn)$), as each cell is visited exactly once.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach. Consider a 3×3 grid, where `1` represents land and `0` represents water:

```
1  Grid:
2  1 0 1
3  1 1 0
4  0 1 0
```

Following the steps outlined in the algorithm:

1. **Initiate a Counter for Perimeter:** Start with `ans = 0`.
2. **Iterate over Grid Cells:** We will examine each cell to determine if it contributes to the perimeter.

First row:

- The first cell (`0,0`) is a `1` (land), so `ans += 4` → `ans = 4`.
- The second cell (`0,1`) is a `0` (water), so no change to `ans`.
- The third cell (`0,2`) is a `1` (land), so `ans += 4` → `ans = 8`.

Second row:

- The first cell (`1,0`) is a `1` (land). We check the cell above (`0,0`) which is also a `1`. This means we have adjacent land cells, so `ans += 4` and `ans -= 2` for the shared edge → `ans = 10`.
- The second cell (`1,1`) is a `1` (land). It's surrounded by land on two sides (above and to the left), so `ans += 4` and `ans -= 4` (2 for each shared edge) → `ans = 10`.
- The third cell (`1,2`) is a `0` (water), so no change to `ans`.

Third row:

- The first cell (`2,0`) is a `0` (water), so no change to `ans`.
- The second cell (`2,1`) is a `1` (land). It's surrounded by land above only, so `ans += 4` and `ans -= 2` for the shared edge → `ans = 12`.
- The third cell (`2,2`) is a `0` (water), so no change to `ans`.

3. **Check for Land Cells:** We have done this part during our iteration and added `4` to `ans` for each land cell.
4. **Check for Adjacent Land:** We have also done this part during our iteration and subtracted `2` from `ans` for every shared edge with adjacent land.
5. **Subtract Shared Edges:** Subtractions are accounted for when checking for adjacent land.
6. **Return Perimeter Count:** After processing the entire grid, the total perimeter `ans` is `12`.

Thus, the perimeter of the island in the given grid is 12.

Python Solution

```python
class Solution:
    def islandPerimeter(self, grid: List[List[int]]) -> int:
        # Get the number of rows and columns of the grid
        rows, cols = len(grid), len(grid[0])

        # Initialize perimeter count
        perimeter = 0

        # Go through each cell in the grid
        for row in range(rows):
            for col in range(cols):

                # If we encounter a land cell
                if grid[row][col] == 1:
                    # Add 4 sides to the perimeter
                    perimeter += 4

                    # If there is a land cell below the current one,
                    # subtract 2 from the perimeter (common side with the bottom cell)
                    if row < rows - 1 and grid[row + 1][col] == 1:
                        perimeter -= 2

                    # If there is a land cell to the right of the current one,
                    # subtract 2 from the perimeter (common side with the right cell)
                    if col < cols - 1 and grid[row][col + 1] == 1:
                        perimeter -= 2

        # Return the total perimeter of the island
        return perimeter
```

Java Solution

```java
class Solution {

    // Function to calculate the perimeter of the island.
    public int islandPerimeter(int[][] grid) {
        // Initialize perimeter sum to 0.
        int perimeter = 0;
        // Get the number of rows in the grid.
        int rows = grid.length;
        // Get the number of columns in the grid.
        int cols = grid[0].length;

        // Iterate through the grid using nested loops.
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {

                // Check if the current cell is land (1 indicates land).
                if (grid[i][j] == 1) {
                    // Add 4 for each land cell as it could potentially contribute 4 sides to the perimeter.
                    perimeter += 4;

                    // If there is land directly below the current land, subtract 2 from perimeter count
                    // (one for the current cell's bottom side and one for the bottom cell's top side).
                    if (i < rows - 1 && grid[i + 1][j] == 1) {
                        perimeter -= 2;
                    }

                    // If there is land directly to the right of the current land, subtract 2 from perimeter count
                    // (one for the current cell's right side and one for the right cell's left side).
                    if (j < cols - 1 && grid[i][j + 1] == 1) {
                        perimeter -= 2;
                    }
                }
            }
        }

        // Return the total perimeter of the island.
        return perimeter;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the perimeter of islands in a grid.
    int islandPerimeter(vector<vector<int>>& grid) {
        // m is the number of rows in the grid.
        int rowCount = grid.size();
        // n is the number of columns in the grid.
        int columnCount = grid[0].size();

        // Initialize the perimeter result to 0.
        int perimeter = 0;

        // Iterate over each cell in the grid.
        for (int row = 0; row < rowCount; ++row) {
            for (int column = 0; column < columnCount; ++column) {
                // Check if the current cell is part of an island.
                if (grid[row][column] == 1) {
                    // Each island cell contributes 4 to the perimeter.
                    perimeter += 4;

                    // If the cell below the current one is also part of the island,
                    // reduce the perimeter by 2 (since two sides are internal and do not contribute to the perimeter).
                    if (row + 1 < rowCount && grid[row + 1][column] == 1) perimeter -= 2;

                    // If the cell to the right of the current one is also part of the island,
                    // reduce the perimeter by 2 for the same reason.
                    if (column + 1 < columnCount && grid[row][column + 1] == 1) perimeter -= 2;
                }
            }
        }

        // Return the total perimeter calculated.
        return perimeter;
    }
};
```

Typescript Solution

```typescript
// Function to calculate the perimeter of islands.
// The grid is represented by a 2D array where 1 indicates land and 0 indicates water.
function islandPerimeter(grid: number[][]): number {
    let heightOfGrid = grid.length;  // The height of the grid
    let widthOfGrid = grid[0].length; // The width of the grid

    let perimeter = 0; // Initialize perimeter counter

    // Iterate over each cell in the grid
    for (let row = 0; row < heightOfGrid; row++) {
        for (let col = 0; col < widthOfGrid; col++) {
            let topNeighbor = 0; // Variable to track the top neighbor's value
            let leftNeighbor = 0; // Variable to track the left neighbor's value

            // Check if the top neighbor exists, and if so, get its value
            if (row > 0) {
                topNeighbor = grid[row - 1][col];
            }

            // Check if the left neighbor exists, and if so, get its value
            if (col > 0) {
                leftNeighbor = grid[row][col - 1];
            }

            let currentCell = grid[row][col]; // Current cell value

            // Compare current cell with the top and left cells; increment perimeter accordingly
            if (currentCell !== topNeighbor) ++perimeter;
            if (currentCell !== leftNeighbor) ++perimeter;
        }
    }

    // Account for the last row and last column edges
    for (let i = 0; i < height; ++i) {
        if (grid[i][width - 1] === 1) ++perimeter; // Increment if last column cell is land
    }
    for (let j = 0; j < width; ++j) {
        if (grid[height - 1][j] === 1) ++perimeter; // Increment if last row cell is land
    }

    return perimeter; // Return the total perimeter of the islands
}
```

Time and Space Complexity

The time complexity of the given code is $O(m * n)$ where `m` is the number of rows and `n` is the number of columns in the `grid`. This is because there is a nested loop which iterates over each cell in the grid exactly once.

The space complexity of the code is $O(1)$ since it only uses a constant amount of additional space. The variable `ans` is updated in place and no additional space that scales with the size of the input is allocated.