Topic Introduction

Today, let's dive into a classic interview concept: **Grid Traversal and Connected Components**. If you've ever had to count clusters, find regions, or process islands in a 2D matrix, you've used this idea—even if you didn't realize it!

What is Grid Traversal?

Grid traversal is about systematically visiting each cell in a 2D grid (or matrix), often to identify and process groups of connected cells (like islands in water). It's a foundational technique in many interview problems, especially those that involve images, maps, or puzzles.

How Does It Work?

The core idea is to explore the grid using **Depth-First Search (DFS)** or **Breadth-First Search (BFS)**. In grid problems, each cell can usually move in four directions: up, down, left, right—sometimes diagonally too. We often use recursion (for DFS) or a queue (for BFS) to explore all cells connected to a given starting point.

Why Is This Useful in Interviews?

Grid traversal problems test your grasp of recursion, iterative algorithms, and how you handle edge cases—literally! They're also a great way to show you can reason about spatial data and apply search algorithms in non-linear structures.

Example (Not from the target problems):

Suppose you're given a grid of 0s and 1s. You need to flip all 1s that are connected to the edge to 0. This is a classic traversal problem: for every 1 on the edge, use DFS/BFS to flip all connected 1s. This demonstrates why traversal is powerful: you can "spread out" from a starting cell to affect an entire region.

Our Trio: The Island Problems

Today’s problems are:

- **Number of Islands** (LeetCode 200)
- **Max Area of Island** (LeetCode 695)
- **Island Perimeter** (LeetCode 463)

Why are these grouped together? All three are about finding and exploring *islands* in a grid, but each asks for a different property: how many islands, the size of the largest one, and the total perimeter. Solving them involves using DFS/BFS to traverse and mark connected components.

Let’s start with the classic: counting islands!

Problem 1: Number of Islands

Problem Link: [LeetCode 200: Number of Islands]

Problem Statement (Reworded):

Given a grid (matrix) of '1's (land) and '0's (water), count how many distinct islands there are. An island is a group of adjacent '1's connected horizontally or vertically. The grid's edges are surrounded by water.

Example Input and Output:

```
Input:
11110
11010
11000
00000

Output: 1
```

Here, there's only *one* island: all the 1s are connected.

Thought Process:

Think of the grid as a map of land and sea. Each time you find a piece of land ('1') that hasn't been visited, you want to "explore" the whole island it belongs to—marking all its land cells as visited—so you don't double-count. Each new exploration means you've found a new island.

Try This Manually:

Draw out this grid:

```
11000
11000
00100
00011
```

How many islands do you see? (Answer: 3)

Brute-Force Approach:

- For every cell, if it's '1', do a DFS/BFS to mark all connected '1's (visited).
- Time Complexity: O(m*n), since each cell is visited once.
- Space Complexity: O(m*n) for visited status (or O(*mn) call stack for recursion).

Optimal Approach:

We use DFS to traverse and mark every land cell in an island. For each unvisited '1', increment the island count, and then recursively mark all connected '1's as '0' (or visited).

Step-by-Step Logic

- Loop through each cell in the grid.
- When you find a '1', start a DFS from it:
    - Mark the current cell as '0' to indicate it's visited.
    - Recursively visit its up/down/left/right neighbors if they are also '1'.
- Each time you start a DFS, increment your island counter.

Encouragement:

Try drawing the grid and simulating the process! It really helps solidify the concept.

Python Solution

```python
def numIslands(grid):
    """
    Given a 2D grid, returns the number of islands.
    """

    if not grid or not grid[0]:
        return 0

    rows, cols = len(grid), len(grid[0])
    islands = 0

    def dfs(r, c):
        # Base case: out of bounds or water
        if r < 0 or r >= rows or c < 0 or c >= cols or grid[r][c] != '1':
            return
        # Mark this cell as visited
        grid[r][c] = '0'
        # Explore all four directions
        dfs(r+1, c)
        dfs(r-1, c)
        dfs(r, c+1)
        dfs(r, c-1)

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1':
                islands += 1    # Found a new island!
                dfs(r, c)       # Mark all its land as visited

    return islands
```

Time Complexity: O(m*n), since every cell is visited at most once.

Space Complexity: O(m*n) in the worst case due to recursion stack (all land).

Explaining the Solution

- The outer loops scan the grid.
- When a '1' is found, we increment the count and run DFS.
- DFS marks every cell in the island as '0', so we never double-count.
- Each direction (up/down/left/right) is explored recursively.

Trace Example:

For grid:

```
11000
11000
00100
00011
```

- Start at (0,0): DFS marks (0,0), (0,1), (1,0), (1,1)
- Next unvisited '1': (2,2): DFS marks (2,2)
- Next unvisited '1': (3,3): DFS marks (3,3), (3,4)

Try This Case Yourself:

```
10110
10110
00000
11000
```

How many islands? (Try dry-running!)

Take a moment:
Try to write this logic yourself before peeking at the code. Drawing out grids and simulating is a great way to build intuition.

Problem 2: Max Area of Island

Problem Link: LeetCode 695: Max Area of Island

How This Problem Differs

- Instead of counting islands, now you want the *size* (number of cells) of the largest island.
- Still, an island is connected '1's in the grid.

Example Input/Output:

```
Input:
11000
11000
00100
00011

Output: 4
```

Largest island has 4 cells (top left).

Approach:

- Still use DFS to traverse each island.
- This time, for each island, count the number of cells as you traverse.
- Track and update the maximum area found.

Brute-Force:

- For each cell, if it's '1', do a DFS to count the area.
- Time Complexity: O(m*n).
- Space Complexity: O(m*n) due to recursion stack.

Optimal Approach (Step-by-Step):

- Loop through all cells.
- For every '1', start a DFS: each time you visit a '1', increment the area.
- After finishing an island, update your max area if this island is larger.
- Mark all visited '1's as '0' to avoid revisiting.

Pseudocode

```
max_area = 0
for each cell (r, c) in grid:
    if grid[r][c] == '1':
        area = dfs(r, c)
        max_area = max(max_area, area)

function dfs(r, c):
    if out of bounds or grid[r][c] != '1':
        return 0
    grid[r][c] = '0'
    area = 1
    for each direction (up, down, left, right):
        area += dfs(next_r, next_c)
    return area
```

Example Walkthrough:

For the previous grid, the DFS from (0,0) counts 4 cells before all '1's are marked as '0'. Other islands have areas of 1 or 2.

Try This Test Case:

```
11011
10001
01110
```

What is the largest island's area? (Hint: 7)

Trace:

- Start at (0,0): DFS marks 3 cells.
- Start at (0,3): DFS marks 1 cell.
- At (2,2): DFS spreads and marks 7 connected cells—this is the max.

Time Complexity: O(m*n)
Space Complexity: O(m*n) (call stack for DFS)

Problem 3: Island Perimeter

Problem Link: LeetCode 463: Island Perimeter

What's Different Here?

- Instead of counting islands or their area, you need to calculate the *perimeter* of the island(s).
- Assume the grid has only one island (or all land forms a single connected component).

Example Input/Output:

```
Input:
0100
1110
0100

Output: 12
```

Approach:

- For each land cell, check its four sides.
- If a side faces water or the edge of the grid, it contributes to the perimeter.

Brute-Force:

- For every cell, look at all four neighbors.
- Time Complexity: O(m*n).

- Space Complexity: O(1).

Optimal Approach (Pseudocode):

```
perimeter = 0
for each cell (r, c) in grid:
    if grid[r][c] == 1:
        for each direction (up, down, left, right):
            if neighbor is water or out of bounds:
                perimeter += 1
return perimeter
```

- No need for DFS here, unless you want to traverse just the island cells.
- Alternatively, you could use DFS to visit all land, but in practice, the four-direction check is sufficient and more direct.

Try This Test Case:

```
Input:
111
101
111

Output: 16
```

- Each interior land cell shares edges with others, so they contribute less to the perimeter.

Challenge:

Try implementing this yourself! As you code, think about:

- What happens at the grid edge?
- How do you avoid double-counting shared edges?

Time Complexity: O(m*n), every cell is checked once.
Space Complexity: O(1) (no recursion needed unless using DFS).

Summary and Next Steps

Today, we tackled **grid traversal problems** where the core pattern is exploring connected components in a 2D grid using DFS or BFS. You learned how this pattern helps you:

- Count distinct islands (connected regions).
- Calculate the size of the largest island.
- Compute the perimeter of island shapes.

Key Patterns and Insights:

- Use DFS/BFS to traverse and mark visited cells.

- Always handle grid edges and avoid index errors.
- Marking visited cells is crucial to avoid infinite loops.
- Carefully track what you're counting: number of islands, area, perimeter.

Common Mistakes:

- Not marking visited cells, leading to double-counting or infinite recursion.
- Forgetting to handle out-of-bounds in recursion.
- Mixing up row and column indices.

Action List

- Solve all three problems on your own, even the one with code above.
- For Problems 2 and 3, try implementing both DFS and BFS approaches.
- Explore related problems like Surrounded Regions, Flood Fill, or Rotting Oranges.
- Compare your solutions with others for edge cases and code style.
- Try drawing out test cases and simulating your algorithms step by step.
- Remember: It's normal to get stuck—keep practicing, and these patterns will become second nature!

Happy coding, and see you for tomorrow’s PrepLetter!