Number of Islands

Problem Description

Given an `m x 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.

Example Explanation

Input

```
4 5
1 1 1 1 0
1 1 0 1 0
1 1 0 0 0
0 0 0 0 0
```

Explanation

- All the `'1'`s are connected to each other either horizontally or vertically.
- There is only **1** distinct group of land.

Output

```
1
```

Input

```
4 5
1 1 0 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
```

Explanation

- Top-left group forms one island.
- Middle isolated `'1'` forms a second island.
- Bottom-right group forms a third island.
- Total islands: **3**.

Output

```
3
```

Constraints & Key Observations

- `m == grid.length`, `n == grid[i].length`
- `1 <= m, n <= 300`
- `grid[i][j]` is `'0'` or `'1'`.
- **Grid Size:** Max dimensions are cells.
- **Algorithm Choice:** An approach is required. Both DFS and BFS fit comfortably within the 1000ms time limit.
- **Connectivity:** Diagonals do **not** count as connections.

Intuition

This is a classic **Connected Components** problem. We treat the grid as a graph where:

- Each cell with value `'1'` is a **node**.
- An edge exists between adjacent `'1'`s (up, down, left, right).

Our goal is to count the number of disconnected components (islands).

Strategy:

1. Iterate through every cell in the grid.
2. If we encounter a `'1'`, it means we have found a **new** island.
3. Increment our island counter.
4. Trigger a traversal (DFS or BFS) to visit **all** land connected to this cell.
5. Mark every visited cell as `'0'` (or "visited") so we don't count it again later.

Approaches

Approach 1: Depth First Search (DFS)

Explanation We scan the grid linearly. When we find a `'1'`, we call a recursive `dfs` function. This function "sinks" the island by turning the current `'1'` into a `'0'`, and then recursively calls itself on all 4 neighbors.

Why It Works Recursion naturally follows the path of the land as deep as possible. By marking cells as `'0'` immediately upon visiting, we ensure that the outer loop skips them, guaranteeing each island is counted exactly once.

Code

```
import sys

# Increase recursion depth for large spiral islands
```

```python
sys.setrecursionlimit(100000)

def solve():
    try:
        # Read all input at once
        input_data = sys.stdin.read().split()
        if not input_data: return

        iterator = iter(input_data)
        m = int(next(iterator))
        n = int(next(iterator))

        grid = []
        for _ in range(m):
            row = []
            for _ in range(n):
                row.append(next(iterator)) # Note: Input is string '1'/'0'
            grid.append(row)

    except StopIteration:
        return

    count = 0

    def dfs(r, c):
        # Base case: Check bounds and if it is water ('0')
        if r < 0 or r >= m or c < 0 or c >= n or grid[r][c] == '0':
            return

        # Mark as visited (sink the island)
        grid[r][c] = '0'

        # Visit neighbors
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    for i in range(m):
        for j in range(n):
            if grid[i][j] == '1':
                # Found a new island
                count += 1
                dfs(i, j)

    print(count)
```

```python
if __name__ == "__main__":
    solve()
```

Time Complexity

- **O(M * N)**: Each cell is visited at most twice (once by the loop, once by DFS).

Space Complexity

- **O(M * N)**: Worst-case recursion stack depth if the entire grid is one island.

Approach 2: Breadth First Search (BFS)

Explanation Instead of recursion, we use a **Queue**. When we find a `'1'`, we add it to the queue and mark it visited. Then we process the queue, adding any unvisited land neighbors until the queue is empty.

Why It Works BFS expands outwards layer-by-layer. It avoids recursion depth limits, making it safer for very large grids in environments with strict stack limits.

Code

```python
import collections

def bfs_solve(grid, m, n):
    count = 0
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for i in range(m):
        for j in range(n):
            if grid[i][j] == '1':
                count += 1

                # Start BFS
                queue = collections.deque([(i, j)])
                grid[i][j] = '0' # Mark visited immediately

                while queue:
                    r, c = queue.popleft()

                    for dr, dc in directions:
```

```
                    nr, nc = r + dr, c + dc
                    if 0 <= nr < m and 0 <= nc < n and grid[nr][nc] == '1':
                        queue.append((nr, nc))
                        grid[nr][nc] = '0'

print(count)
```

Edge Cases & Common Pitfalls

- **Input Format:** Be careful if the input is given as strings `"1"` vs integers `1`.
- **Diagonal Connections:** Remember that islands connect **only** horizontally and vertically. `1` at `(0,0)` and `1` at `(1,1)` are **two** islands if `(0,1)` and `(1,0)` are `0`.
- **Grid Mutation:** The standard approach destroys the input grid. If the grid must remain read-only, use a separate `visited` set (costs extra memory).

When Not to Use DFS

- If constraints were significantly larger (e.g.,), DFS might hit stack overflow limits. In those cases, BFS or Union-Find (DSU) is preferred.

“‘