Max Area of Island

Problem Description

You are given an `m x n` binary matrix `grid`. An island is a group of `1`'s (representing land) connected **4-directionally** (horizontal or vertical). You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value `1` in the island.

Return the maximum area of an island in `grid`. If there is no island, return `0`.

Example Explanation

Input

```
8 13
0 0 1 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 0 0 0
0 1 1 0 1 0 0 0 0 0 0 0 0
0 1 0 0 1 1 0 0 1 0 1 0 0
0 1 0 0 1 1 0 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0
```

Explanation

The grid contains several islands (groups of connected 1s).

- One island has area 1.
- Another has area 4.
- The largest island (located towards the right-middle) has an area of **6**.

Output

```
6
```

Input

```
1 8
0 0 0 0 0 0 0 0
```

Explanation There are no 1s in the grid.

Output

```
0
```

Constraints & Key Observations

- `m == grid.length`, `n == grid[i].length`
- `1 <= m, n <= 50`
- `grid[i][j]` is either `0` or `1`.
- **Grid Size:** The maximum grid size is cells. This is small enough for linear graph traversal algorithms like DFS or BFS.
- **Time Complexity:** An solution is required.

Intuition

This is a classic **Connected Components** problem on a graph.

- Each cell `(r, c)` with value `1` is a node.
- An edge exists between two nodes if they are adjacent (up, down, left, right).
- The "Area" is simply the number of nodes in a connected component.

We need to iterate through every cell in the grid. If we find a `1` that hasn't been visited yet, it implies we found a new island. We then traverse the entire island to count its size.

Approaches

Approach 1: Depth First Search (DFS)

Explanation

1. Iterate through every cell `(r, c)` in the grid.
2. If `grid[r][c] == 1`, it marks the start of a new island.
3. Start a DFS traversal from `(r, c)`:

- Mark the current cell as visited (e.g., set it to `0` or use a visited set).
- Initialize `area = 1`.
- Recursively visit all valid 4-directional neighbors that are also `1`.
- Add the returned areas from neighbors to the current `area`.

4. Keep track of the `max_area` found so far.

Why It Works DFS naturally explores as deep as possible, visiting every connected node in a component before backtracking. This ensures we count every single `1` belonging to the current island exactly once.

Code

```python
import sys

# Increase recursion depth for deep islands (max 2500 cells)
sys.setrecursionlimit(3000)

def solve():
    try:
        # Reading all input at once for easier parsing
        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(int(next(iterator)))
            grid.append(row)

    except StopIteration:
        return

    max_area = 0

    def dfs(r, c):
        # Base cases: out of bounds or water
        if r < 0 or r >= m or c < 0 or c >= n or grid[r][c] == 0:
            return 0

        # Mark as visited by sinking the island (changing 1 to 0)
        grid[r][c] = 0
        area = 1

        # Explore neighbors
        area += dfs(r + 1, c)
        area += dfs(r - 1, c)
        area += dfs(r, c + 1)
        area += dfs(r, c - 1)

        return area

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

```
            if grid[i][j] == 1:
                # Found an unvisited island
                max_area = max(max_area, dfs(i, j))

    print(max_area)

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

Time Complexity

- **O(M * N)**: We visit every cell at most twice (once via the loop, and once via DFS).

Space Complexity

- **O(M * N)**: In the worst case (the entire grid is one snake-like island), the recursion stack can go up to deep.

Approach 2: Breadth First Search (BFS)

Explanation Similar logic to DFS, but uses a **Queue** instead of recursion.

1. When a `1` is found, add it to a queue and set it to `0`.
2. While the queue is not empty:

- Pop a cell.
- Increment area.
- Check all 4 neighbors. If a neighbor is `1`, add to queue and set to `0`.

Why It Works BFS explores the island layer by layer. It avoids recursion depth limits and is generally safer for very large connected components in languages with strict stack limits.

Code

```
import collections

def solve_bfs(grid, m, n):
    max_area = 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:
                # Found new island
```

```
                area = 0
                queue = collections.deque([(i, j)])
                grid[i][j] = 0 # Mark visited immediately

                while queue:
                    r, c = queue.popleft()
                    area += 1

                    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 # Mark visited to prevent re-queueing

                max_area = max(max_area, area)
    return max_area
```

Edge Cases & Common Pitfalls

- **No Islands:** Grid contains only `0`s. Result should be 0.
- **Grid Modification:** The solutions above **modify the input grid** (changing 1s to 0s) to mark visited cells. If the input must be preserved, use a separate `visited` set (costs space).
- **Recursion Limit:** For a grid, the path can be length 2500. Python's default recursion limit is usually 1000. Always `sys.setrecursionlimit` for DFS on grids.
- **Single Cell:** A grid of `1 1` with `[[1]]` should return 1.

When Not to Use DFS

- If the graph is extremely deep (millions of nodes), DFS will cause a Stack Overflow. Use BFS or Union-Find (Disjoint Set Union) instead.

‘‘‘