885. Spiral Matrix III

Medium Array Matrix Simulation

Leetcode Link

Problem Description

In this LeetCode problem, you are given the dimensions of a grid (`rows x cols`) and a starting cell within that grid, defined by its row (`rStart`) and column (`cStart`). Your task is to simulate a walk from the starting cell in a clockwise spiral pattern until you have visited every cell in the grid. This spiral pattern means you move east, then south, then west, and finally north, increasing the distance you move in a specific direction before turning by one cell each time you complete a full cycle (east, south, west, and north). As you perform this walk, even if you move outside the grid's boundaries, you keep the pattern, and you may return to the grid's boundary later. The goal is to visit all cells on the grid and return a list of their coordinates in the order they were visited.

Intuition

The problem can be solved by simulating the walk in a spiral pattern as described. The intuition behind this solution is to track movement direction and counter changes in direction while traversing the grid. We should keep in mind that it is necessary to handle cases where the walk takes us outside the grid boundary.

We initialize the solution by adding the starting position to our result set. If the grid is of size 1×1, the problem is already solved with this step. For larger grids, we have to begin walking in a spiral. We know that for a spiral walk, the number of steps we take in the east or west directions (horizontal steps) will always be one more than the number of steps we took in the previous vertical movement (north or south). Conversely, the number of steps we take in the north or south direction will be equal to the number of steps we will take in the next horizontal movement.

To implement this pattern, we maintain a variable `k` that represents how many steps we should move in the current direction. We start with `k = 1`, as the first move after the starting position is just one step to the right (east). We then increase `k` by 2 after a full cycle of directions (east > south > west > north) to account for the changing step count in the spiral path.

For each direction, we move `k` steps, updating our current position and conditionally add the new position to the result set if it lies within the grid's boundaries. We keep doing so until we have visited all `rows * cols` cells.

Solution Approach

The implementation of the solution follows a pattern-based approach by simulating movements on the grid. Here's how it's done step by step:

1. We define a list `ans` to keep track of the cells visited, starting with the initial cell `[rStart, cStart]`.
2. We check if the grid size is `1x1`. If so, we immediately return `ans` because the single starting cell is the only cell to visit.
3. We then set `k = 1`, which will determine the number of steps we take in a given direction before turning. This `k` will be increased as necessary to simulate the spiral pattern.
4. A `while True` loop begins, which will run until we've visited all `rows * cols` cells of the grid.
5. Inside the loop, we iterate over a list of direction increments and the corresponding step counts. The list has tuples in the format `[dr, dc, dk]`, where `dr` is the row increment (0 for east or west, 1 for south, -1 for north), `dc` is the column increment (1 for east, -1 for west, 0 for north or south), and `dk` being the number of steps to move in that direction.
6. For each direction, we run another loop for `dk` steps:
   - For each step, we update `rStart` and `cStart` with `dr` and `dc` respectively.
   - We then check if the new position is within the grid's boundaries by verifying `0 <= rStart < rows` and `0 <= cStart < cols`. If yes, we append the position to `ans`.
   - We check if we have visited `rows * cols` cells. If we have, we return `ans` as the complete list of visited cells.
7. After completing the movement in all four directions, we increment `k` by 2 to maintain the spiral pattern for the next cycle of directions.

The algorithm utilizes simple iteration and directional increments to traverse the grid in a predictable, spiral pattern. No complex data structures are necessary beyond a list to hold the visited cell coordinates. The pattern's consistency allows us to increment `k` strategically, ensuring we expand our traversal in a spiraling outward fashion. This method provides full coverage of the grid while recording our path.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Consider a grid of size `3x3`, where our starting cell is at `rStart = 1` and `cStart = 1` (the center of the grid).

1. We initialize `ans` with the starting cell, so `ans = [[1, 1]]`.
2. Since the grid size is not `1x1`, we proceed with the solution.
3. We set `k = 1` as our initial step length for the eastward move.
4. Since the grid has more than one cell, we enter the `while True` loop to start the spiral traversal.
5. The first direction from our starting cell is east. We only need to move one step east (`dc = 1, dr = 0`). We move to cell `[1, 2]`, which is within the grid. We add this to `ans`, resulting in `ans = [[1, 1], [1, 2]]`.
6. Next is to move one step south (`dc = 0, dr = 1`). We move to `[2, 2]`. Again, it's within the grid, so `ans` becomes `ans = [[1, 1], [1, 2], [2, 2]]`.
7. Following is westward movement, but since `k` is `1`, we'd move only one step west (`dc = -1, dr = 0`) to `[2, 1]`. We add this to `ans`, making it `ans = [[1, 1], [1, 2], [2, 2], [2, 1]]`.
8. The last direction in this cycle is north (`dc = 0, dr = -1). We move one step north to `[1, 1]`, but since this cell has already been visited, we don't add it again to `ans`.
9. We've completed a full cycle (east > south > west > north), so we increment `k` by 2, making `k = 3`.
10. For the next eastward movement, we will move three steps, but after one step, we're already at the edge. The next position would be `[1, 3]`, within the grid, so it's added to `ans` - `ans = [[1, 1], [1, 2], [2, 2], [2, 1], [1, 3]]`.
11. Continuing the cycle, we go south three steps, adding positions `[2, 3]` and `[3, 3]` to `ans`.
12. Going west for three steps, we add `[3, 2]` and `[3, 1]` to `ans`.
13. Finally, moving north, we add `[2, 1]` and the last remaining cell `[1, 1]` is ignored as it's already visited.
14. All cells of the `3x3` grid have been added to `ans`, and the traversal is complete.

The final `ans`, with the cells visited in the order of the spiral from the center, would be:

```
[[1, 1], [1, 2], [2, 2], [2, 1], [1, 3], [2, 3], [3, 3], [3, 2], [3, 1], [2, 1]]
```

This example demonstrated the simulated spiral movement through a grid following the solution approach outlined.

Python Solution

```python
class Solution:
    def spiralMatrixIII(self, rows: int, cols: int, r_start: int, c_start: int) -> List[List[int]]:
        # Initialize the answer list with the starting cell
        result = [[r_start, c_start]]

        # If there's only one cell, return it immediately
        if rows * cols == 1:
            return result

        # 'k' represents the number of steps we take in a given direction before turning
        # It starts at 1 and gets incremented after finishing an east and north pass
        k = 1

        # Continue generating the spiral pattern until we've filled the result with all cells
        while True:
            # Each iteration goes in the pattern: East, South, West, North
            # 'dr' and 'dc' represent the change to rows and cols respectively
            # 'dk' represents how many steps we take in each direction before turning
            for dr, dc, dk in [[0, 1, k], [1, 0, k], [0, -1, k + 1], [-1, 0, k + 1]]:
                # Repeat the movement 'dk' times for the current orientation
                for _ in range(dk):
                    # Update the current position
                    r_start += dr
                    c_start += dc
                    # If we're still within the bounds of the matrix, add to result
                    if 0 <= r_start < rows and 0 <= c_start < cols:
                        result.append([r_start, c_start])
                        # If the result is now filled with all matrix cells, return it
                        if len(result) == rows * cols:
                            return result
            # Increment 'k' for the next spiral arm to have the correct step count
            k += 2

# The return type should be List[List[int]], but this cannot be used directly in a code snippet
# without importing List from typing. Be sure to include that in your actual implementation.
```

Make sure to include the import statement for `List` from the `typing` module at the beginning of your code if you're running this outside of LeetCode's environment where the import might be implicit:

```python
from typing import List

```

Java Solution

```java
class Solution {
    public int[][] spiralMatrixIII(int rows, int cols, int rStart, int cStart) {
        // Determine the total number of elements in the matrix
        int totalElements = rows * cols;
        // Initialize the answer array with a size equal to the number of elements
        int[][] result = new int[totalElements][2];
        // Starting position is the first element in the result array
        result[0] = new int[] {rStart, cStart};

        // If there's only one element, return the result immediately
        if (totalElements == 1) {
            return result;
        }

        int index = 1; // Start from the second element in the result array

        // Loop indefinitely; the exit condition is when all matrix elements have been added to result
        for (int k = 1; ; k += 2) {
            // Directions and step increment: right, down, left, and up
            int[][] directions = new int[][] {
                {0, 1, k},   // Move right k steps
                {1, 0, k},   // Move down k steps
                {0, -1, k + 1}, // Move left (k+1) steps
                {-1, 0, k + 1}  // Move up (k+1) steps
            };

            // Iterate through each direction
            for (int[] dir : directions) {
                int rowStep = dir[0], colStep = dir[1], steps = dir[2];

                // Move within the current direction for 'steps' times
                while (steps-- > 0) {
                    // Move to the next cell in the current direction
                    rStart += rowStep;
                    cStart += colStep;

                    // Check if the current cell is within the boundaries of the matrix
                    if (rStart >= 0 && rStart < rows && cStart >= 0 && cStart < cols) {
                        // Add the current cell to the result
                        result[index++] = new int[] {rStart, cStart};
                        // If we've added all matrix elements to the result, return the result
                        if (index == totalElements) {
                            return result;
                        }
                    }
                }
            }
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<int>> spiralMatrixIII(int rows, int cols, int startRow, int startCol) {
        int totalCells = rows * cols; // Total number of cells in the matrix
        vector<vector<int>> result; // This will store the path of the spiral
        result.push_back({startRow, startCol}); // Starting position

        // If there is only one cell in the matrix, return the result now
        if (totalCells == 1) {
            return result;
        }

        // Spiral can be formed by increasing the steps in the east and north directions,
        // and then increasing by an additional step when moving west and south.
        for (int stepIncrease = 1; ; stepIncrease += 2) {
            // Directions are East, South, West, North.
            // The third value in each vector holds the number of steps to take in that direction.
            vector<vector<int>> directions = {{0, 1, stepIncrease},   // Move right (East)
                                              {1, 0, stepIncrease},   // Move down (South)
                                              {0, -1, stepIncrease + 1}, // Move left (West)
                                              {-1, 0, stepIncrease + 1}}; // Move up (North)

            // Go through each of the four directions
            for (auto& dir : directions) {
                // 'dir' is vector<int> that contains the row increment, column increment, and number of steps.
                int rowIncrement = dir[0];
                int colIncrement = dir[1];
                int steps = dir[2];

                // Move the number of steps in the current direction
                while (steps-- > 0) {
                    // Update the starting position
                    startRow += rowIncrement;
                    startCol += colIncrement;

                    // Check if the new position is within the matrix bounds
                    if (startRow >= 0 && startRow < rows && startCol >= 0 && startCol < cols) {
                        result.push_back({startRow, startCol}); // Add to result
                        // If we've added all cells, return the result
                        if (result.size() == totalCells) {
                            return result;
                        }
                    }
                }
            }
        }
    }
};

```

Typescript Solution

```typescript
function spiralMatrixIII(rows: number, cols: number, startRow: number, startCol: number): number[][] {
    // Total number of cells in the matrix
    const totalCells: number = rows * cols;
    // This will store the path of the spiral
    const path: number[][] = [[startRow, startCol]];

    // If there is only one cell in the matrix, return the path now
    if (totalCells === 1) {
        return path;
    }

    // Loop wherein each iteration potentially adds two sides of the spiral.
    for (let stepIncrease = 1; ; stepIncrease += 2) {
        // Directions are East, South, West, North.
        // Each tuple contains the row and column increments, and the number of steps to take.
        const directions: [number, number, number][] = [
            [0, 1, stepIncrease],     // Move right (East)
            [1, 0, stepIncrease],     // Move down (South)
            [0, -1, stepIncrease + 1], // Move left (West)
            [-1, 0, stepIncrease + 1] // Move up (North)
        ];

        for (const [rowIncrement, colIncrement, steps] of directions) {
            // Initialize a variable for the number of steps to be taken in the current direction
            let stepsRemaining = steps;

            // Move the number of steps in the current direction
            while (stepsRemaining-- > 0) {
                // Update the starting position
                startRow += rowIncrement;
                startCol += colIncrement;

                // Check if the new position is within the matrix bounds
                if (startRow >= 0 && startRow < rows && startCol >= 0 && startCol < cols) {
                    // Add the new position to the path
                    path.push([startRow, startCol]);
                    // If we've added all cells, return the path
                    if (path.length === totalCells) {
                        return path;
                    }
                }
            }
        }
    }
}

// Example use:
// const result = spiralMatrixIII(5, 6, 1, 4);
// console.log(result); // This would log the spiral matrix to the console.

```

Time and Space Complexity

The given Python code generates all the coordinates in a matrix in a spiral order starting from a given cell `(rStart, cStart)`. Considering each coordinate in a `rows x cols` grid needs to be visited exactly once, let's analyze both the time complexity and space complexity of the code.

Time Complexity:

The time complexity of this function can be analyzed by understanding the number of steps taken to complete the spiral. The number of steps increases as the spiral grows. The spiral growth is implemented by incrementing the step size `k` after every two directions (`right-up` and `left-down`). Initially, `k` starts at 1, and after each full cycle (right → down → left → up), `k` is incremented by 2. This means that for a matrix that requires `n` cycles to fill, the total number of steps `S` would be the sum of an arithmetic series:

```
S = 1 + 2 + 3 + 4 + ... + (2n - 1) + 2n
```

This sum is equal to `n(2n + 1)`, which is `O(n^2)`.

Since `n` is proportional to the larger dimension of the matrix (`rows` or `cols`), and the number of iterations required to cover all cells of the matrix will be bound by the total number of cells, we can conclude that the time complexity is `O(max(rows, cols)^2)`.

Space Complexity:

The space complexity of the algorithm is determined by the amount of space needed to store the output. Since the output `ans` is a list of all coordinates in the matrix, it needs to store exactly `rows * cols` elements.

Thus, the space complexity is `O(rows * cols)`.