54. Spiral Matrix

Medium Array Matrix Simulation Leetcode Link

Problem Description

In this problem, we are tasked with traversing a 2D array (or matrix) in a spiral pattern. Imagine starting at the top-left corner of the matrix and going right, then down, then left, and then up, turning inward in a spiral shape, until we traverse every element in the matrix exactly once. The function should return a list of the elements of the matrix in the order they were visited during this spiral traversal.

Intuition

To solve this problem, we need to simulate the process of traveling around the matrix spiral. Essentially, we keep moving in the same direction (right initially) until we meet a boundary of the matrix or a previously visited cell. When we encounter such a boundary, we make a clockwise turn and continue the process.

There are two insightful approaches to this problem:

1. **Simulation Approach**: We initiate four direction vectors (or in our case, a direction array `dirs`) that represent right, down, left, and up. We iterate over all elements in the matrix by taking steps in the initial direction until we can no longer move forward. At this point, we turn 90 degrees and continue. To avoid revisiting cells, we mark each visited cell by adding it to a set `vis` so that we know when to turn. The time complexity here is `O(m x n)` because we visit each element once, and the space complexity is also `O(m x n)` due to the extra space used to store visited cells.

2. **Layer-by-layer Simulation**: Instead of marking visited cells, we can visualize the matrix as a series of concentric rectangular layers. We traverse these layers from the outermost to the innermost, peeling them away as we go. This approach requires careful handling of the indices to ensure we stay within the bounds of the current layer. This way of traversal helps to potentially reduce extra space usage because we don't explicitly need to keep track of visited cells.

In the provided solution code, we follow the simulation approach. We appropriately update our direction of movement based on the bounds of the matrix and whether we've visited a cell. The direction change is done by iterating over the `dirs` array and updating our row and column pointers `i` and `j` respectively. The result list `ans` stores the elements as we traverse the matrix.

Solution Approach

The given Python solution follows the **Simulation Approach** described in the intuition section.

1. **Initialization**:
   - We define `m` and `n` which are the dimensions of the matrix (number of rows and number of columns respectively).
   - The `dirs` array `dirs = (0, 1, 0, -1, 0)` encodes the direction vectors. `dirs[k]` and `dirs[k+1]` together represent the direction we move in, with `k` starting at 0 and cycling through values 0 to 3 to represent right, down, left, and up in that order.
   - The `i` and `j` variables represent the current row and column positions in the matrix.
   - `ans` list is where we collect the elements of the matrix as we visit them.
2. **Visiting Elements**:
   - We loop exactly `m * n` times, once for each element of the matrix.
   - Each time through the loop, we append the current element to the `ans` list and mark its position `(i, j)` as visited by adding it to the `vis` set.
3. **Moving Through the Matrix**:
   - We calculate the next position `(x, y)` based on the current direction we are moving. This is done using the current values of `i`, `j`, and `k`.
   - Before we move to the next position, we check if the position is valid - it must be within bounds and not already visited. This is the `if not 0 <= x < m or not 0 <= y < n or (x, y) in vis:` check.
   - If the move is invalid, we change the direction by increasing `k` modulo 4. This works because if `k` is 3 and we add 1, `(k + 1) % 4` will reset `k` back to 0.
   - Once the direction is confirmed as valid, we update `i` and `j` to move to the next position in the matrix.
4. **Handling Edge Cases**:
   - Because we update the direction whenever we hit the edge of the matrix or a visited cell, the algorithm naturally handles non-square matrices and any edge cases where the spiral must turn inward.
5. **Completing the Spiral**:
   - The process continues, circling around the matrix and moving inward until all elements have been added to `ans`.
6. **Space Optimization**:
   - In the reference solution, it's suggested that instead of using a `vis` set to keep track of visited cells (contributing to space complexity of `O(m x n)`), we could modify the matrix itself to mark cells as visited. This could be done by adding a constant value which is outside the range of the matrix values, effectively using the input matrix as the `vis` state. This reduces the space complexity to `O(1)`, provided the matrix can be modified and that the added constant is chosen such that it doesn't cause integer overflow.

The executed code follows this approach rigorously, and through simulation, delivers the correct spiral traversal of the input matrix. The attention to directional changes and boundary conditions ensures that all cases are handled smoothly.

Example Walkthrough

Let's illustrate the solution approach using a small example where our input is a 2D matrix:

```
matrix = [
  [ 1,  2,  3],
  [ 4,  5,  6],
  [ 7,  8,  9]
]
```

Now we'll walk through the solution step by step:

1. **Initialization**:
   - `m = 3` (3 rows), `n = 3` (3 columns)
   - Direction array `dirs = (0, 1, 0, -1, 0)` indicates the x and y offsets for right, down, left, and up movements respectively.
   - We start at the top-left corner, so initial indices `i = 0` and `j = 0`.
   - `ans = []` will collect the elements in spiral order.
   - `vis = set()` to store visited positions.
2. **Visiting Elements**:
   - We begin by appending `matrix[0][0]` to `ans`, so `ans = [1]`.
   - We add `(0, 0)` to `vis` to mark it as visited.
3. **Moving Through the Matrix**:
   - We calculate the next position using the current direction (right, `k = 0`), so `x = i + dirs[k] = 0` and `y = j + dirs[k+1] = 1`.
   - This position is within bounds and not visited, so we move there and append `matrix[0][1]` to `ans`, making it `[1, 2]`.
   - We repeat this process and append `[3, 6, 9]` to `ans`.
4. **Direction Change and Boundary Check**:
   - After reaching the last column, we check the next right move and find it's out of bounds.
   - We then change direction to down (`k = 1`), and append `[8, 7]` to `ans`.
5. **Avoiding Visited Cells**:
   - Next, we attempt to move left but the cell `(2, 0)` is visited.
   - We turn again, moving up and find the top center cell `(0, 1)` is also visited.
   - We turn right and add `[4, 5]` to the spiral traversal.
6. **Completing the Spiral**:
   - We've now visited all cells, and `ans` is `[1, 2, 3, 6, 9, 8, 7, 4, 5]`.
7. **Space Optimization (Optional)**:
   - A potentially more space-efficient approach might entail modifying the given `matrix` to mark visited elements if allowed.

By following the simulation approach, the function would return the traversal in spiral order as `[1, 2, 3, 6, 9, 8, 7, 4, 5]`.

Python Solution

```python
class Solution:
    def spiralOrder(self, matrix):
        """
        This function takes a matrix and returns a list of elements in spiral order.
        """

        # Define matrix dimensions.
        rows, cols = len(matrix), len(matrix[0])

        # Define directions for spiral movement (right, down, left, up).
        directions = ((0, 1), (1, 0), (0, -1), (-1, 0))

        # Initialize row and column indices and the direction index.
        row = col = direction_index = 0

        # Initialize the answer list and a set to keep track of visited cells.
        result = []
        visited = set()

        # Iterate over the cells of the matrix.
        for _ in range(rows * cols):
            # Append the current element to the result list.
            result.append(matrix[row][col])
            # Mark the current cell as visited.
            visited.add((row, col))

            # Calculate the next cell's position based on the current direction.
            next_row, next_col = row + directions[direction_index][0], col + directions[direction_index][1]

            # Check if the next cell is within bounds and not visited.
            if not (0 <= next_row < rows) or not (0 <= next_col < cols) or (next_row, next_col) in visited:
                # Change direction if out of bounds or cell is already visited.
                direction_index = (direction_index + 1) % 4

            # Update the row and column indices to the next cell's position.
            row += directions[direction_index][0]
            col += directions[direction_index][1]

        # Return the result list.
        return result
```

Java Solution

```java
import java.util.List;
import java.util.ArrayList;

class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        // Dimensions of the 2D matrix
        int rowCount = matrix.length;
        int colCount = matrix[0].length;
        // Direction vectors for right, down, left, and up
        int[] directionRow = {0, 1, 0, -1};
        int[] directionCol = {1, 0, -1, 0};
        // Starting point
        int row = 0, col = 0;
        // Index for the direction vectors
        int directionIndex = 0;
        // List to hold the spiral order
        List<Integer> result = new ArrayList<>();
        // 2D array to keep track of visited cells
        boolean[][] visited = new boolean[rowCount][colCount];

        for (int h = rowCount * colCount; h > 0; --h) {
            // Add the current element to the result
            result.add(matrix[row][col]);
            // Mark the current cell as visited
            visited[row][col] = true;
            // Compute the next cell position
            int nextRow = row + directionRow[directionIndex];
            int nextCol = col + directionCol[directionIndex];
            // Check if the next cell is out of bounds or visited
            if (nextRow < 0 || nextRow >= rowCount || nextCol < 0 || nextCol >= colCount || visited[nextRow][nextCol]) {
                // Update the direction index to turn right in the spiral order
                directionIndex = (directionIndex + 1) % 4;
                // Recompute the next cell using the new direction
                nextRow = row + directionRow[directionIndex];
                nextCol = col + directionCol[directionIndex];
            }
            // Move to the next cell
            row = nextRow;
            col = nextCol;
        }
        return result;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        if (matrix.empty()) return {}; // Return an empty vector if the matrix is empty

        int rows = matrix.size(), cols = matrix[0].size(); // rows and cols store the dimensions of the matrix
        vector<int> directions = {0, 1, 0, -1, 0}; // Row and column increments for right, down, left, up movements
        vector<int> result; // This vector will store the elements of matrix in spiral order
        vector<vector<bool>> visited(rows, vector<bool>(cols, false)); // Keep track of visited cells

        int row = 0, col = 0, dirIndex = 0; // Start from the top-left corner and use dirIndex to index into directions

        for (int remain = rows * cols; remain > 0; --remain) {
            result.push_back(matrix[row][col]); // Add the current element to result
            visited[row][col] = true; // Mark the current cell as visited

            // Calculate the next cell position
            int nextRow = row + directions[dirIndex], nextCol = col + directions[dirIndex + 1];

            // Change direction if next cell is out of bounds or already visited
            if (nextRow < 0 || nextRow >= rows || nextCol < 0 || nextCol >= cols || visited[nextRow][nextCol]) {
                dirIndex = (dirIndex + 1) % 4; // Rotate to the next direction
            }

            // Move to the next cell
            row += directions[dirIndex];
            col += directions[dirIndex + 1];
        }
        return result; // Return the result
    }
};
```

Typescript Solution

```typescript
function spiralOrder(matrix: number[][]): number[] {
    const rowCount = matrix.length; // Number of rows in the matrix
    const colCount = matrix[0].length; // Number of columns in the matrix
    const result: number[] = []; // The array that will be populated and returned
    const visited = new Array(rowCount).fill(0).map(() => new Array(colCount).fill(false)); // A 2D array to keep track of visited
    const directions = [0, 1, 0, -1, 0]; // Direction array to facilitate spiral traversal: right, down, left, up
    let remainingCells = rowCount * colCount; // Total number of cells to visit

    // Starting point coordinates and direction index
    let row = 0;
    let col = 0;
    let dirIndex = 0;

    // Iterate over each cell, decrementing the count of remaining cells
    for (; remainingCells > 0; --remainingCells) {
        result.push(matrix[row][col]); // Add the current cell's value to the result
        visited[row][col] = true; // Mark the current cell as visited

        // Calculate the indices for the next cell in the current direction
        const nextRow = row + directions[dirIndex];
        const nextCol = col + directions[dirIndex + 1];

        // Check if the next cell is out of bounds or already visited
        if (nextRow < 0 || nextRow >= rowCount || nextCol < 0 || nextCol >= colCount || visited[nextRow][nextCol]) {
            dirIndex = (dirIndex + 1) % 4; // Change direction (right -> down -> left -> up)
        }

        // Move to the next cell in the updated/current direction
        row += directions[dirIndex];
        col += directions[dirIndex + 1];
    }

    return result; // Return the array containing the spiral order traversal of the matrix
}
```

Time and Space Complexity

The time complexity of the function `spiralOrder` is `O(m * n)` where `m` is the number of rows and `n` is the number of columns in the input matrix. This is because the function iterates over every element in the matrix exactly once.

The space complexity of the function, however, is not `O(1)` as stated in the reference answer. Instead, it is `O(m * n)` because the function uses a set `vis` to track visited elements, which in the worst-case scenario, can grow to contain every element in the matrix.