59. Spiral Matrix II

Medium Array Matrix Simulation

Leetcode Link

Problem Description

The task is to create a square matrix (2-dimensional array) of a given size `n` where `n` is a positive integer. The matrix should be filled with numbers from `1` to `n^2` (1 to n squared) following a spiral pattern. A spiral pattern means we start from the top-left corner (0,0) and fill the matrix to the right, then downwards, then to the left, and finally upwards, before moving inwards in a spiral manner and repeating the directions until the entire matrix is filled.

Intuition

The intuition behind the solution is to replicate the spiral movement by using direction vectors to move the filling process right, down, left, and up. We use a variable to track the value to be filled in the next cell, which begins at `1` and ends at `n^2`. To navigate the matrix:

1. We maintain a direction vector `dirs` which contains tuples representing the direction of movement: right `(0, 1)`, down `(1, 0)`, left `(0, -1)`, and up `(-1, 0)`.
2. We initialize our position at the start of the matrix `(0,0)`
3. We iterate through the values from `1` to `n^2`, filling the cells of the matrix.
4. After inserting a value, we check if the next step will go out of bounds or into a cell that already has a value. If so, we change direction by rotating to the next direction vector.
5. This process is continued until all cells are filled accordingly maintaining the spiral order.

The solution uses modulo division to cycle through the directions, ensuring when we reach the end of the direction vector, it loops back to the beginning. This helps us maintain the spiral path without creating complex conditional statements.

Solution Approach

The solution applies a simulation approach, where we simulate the spiral movement within the matrix. Let's dissect the solution approach.

1. We start by creating an `n x n` matrix filled with `0`s to hold the values. This is achieved by the list comprehension `[[0] * n for _ in range(n)]`.
2. We define a `dirs` array which contains four tuples. Each tuple represents the direction change for each step in our spiral: right is `(0, 1)`, down is `(1, 0)`, left is `(0, -1)`, and up is `(-1, 0)`.
3. We initialize three variables `i`, `j`, and `k` which represent the current row, current column, and current direction index, respectively.
4. A `for` loop is used to iterate through the range from `1` to `n^2`, inclusive. During each iteration, we perform the following steps:
   - Place the current value `v` in the `ans` matrix at position `(i, j)`.
   - Calculate the next position `(x, y)` by adding the current direction vector `dirs[k]` to the current position `(i, j)`.
   - Check if the next position is out of bounds or if the cell has already been visited (non-zero value). If either is true, we change the direction by updating the value of `k` with `(k + 1) % 4`, which rotates to the next direction vector in `dirs`.
   - Update the current position `(i, j)` to the new position `(x, y)` based on the direction we are moving.
5. The loop stops when all values from `1` to `n^2` have been placed into the `ans` matrix.

By following this approach, we can generate the matrix with numbers from `1` to `n^2` in spiral order dynamically for any size of `n`.

The use of a direction vector is a common technique in grid traversal problems. It simplifies the process of moving in the four cardinal directions without writing multiple if-else conditions. The modulo operator `%` assists in cycling through our direction vectors to maintain the correct spiral movement.

Example Walkthrough

Let's assume `n = 3` to illustrate the solution approach. Our goal is to fill a 3×3 matrix with numbers from `1` to `3^2` (which is 9), in a spiral pattern.

1. We create an empty 3×3 matrix filled with 0's.

```
ans = [
  [0, 0, 0],
  [0, 0, 0],
  [0, 0, 0]
]
```

2. The `dirs` array contains direction vectors: `[(0, 1), (1, 0), (0, -1), (-1, 0)]`. This represents right, down, left, and up movement respectively.
3. We set `i`, `j`, and `k` to `0`. Here, `(i, j)` is the current position (initially at the top-left corner), and `k` is the index for direction vectors (starting with right movement).
4. We will fill the matrix with values from 1 to 9. For each value `v`, we do the following:
   - Place `v` at `ans[i][j]`. For the first iteration, we place `1` at `ans[0][0]`.
   - Update the next position `(x, y)` by adding the direction vector to the current position `(i, j)`.
   - If `(x, y)` is out of bounds or `ans[x][y]` is not `0`, we update `k` to `(k + 1) % 4` to change direction.
   - Move to `(x, y)` and repeat the process.
5. We continue this process until all values are filled into the matrix. After the completion, the `ans` matrix looks like:

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

The steps are as follows: Start with the top-left corner (0,0), move right and fill `1`, `2`, `3`, then move down to fill `4`, move left for `5`, move up for `6`, again move right for `7`, then go to the center and fill `8`, and finally move right to fill `9`.

This makes the `ans` matrix to have its elements in a spiral order, from `1` to `9`. By doing so for any `n`, we can generate the desired spiral matrix.

Python Solution

```python
class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        # Initialize the matrix with zeros
        matrix = [[0] * n for _ in range(n)]
        # Define directions for movement: right, down, left, and up
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        # Initialize the starting point and direction index
        row = column = direction_index = 0

        # Iterate over all values from 1 to n^2 to fill the matrix
        for value in range(1, n * n + 1):
            # Assign the current value to the matrix
            matrix[row][column] = value
            # Calculate the next position
            next_row, next_column = row + directions[direction_index][0], column + directions[direction_index][1]
            # Check if the next position is out of bounds or already filled
            if next_row < 0 or next_column < 0 or next_row >= n or next_column >= n or matrix[next_row][next_column]:
                # Change direction if the next position is invalid
                direction_index = (direction_index + 1) % 4
                # Recalculate the next position after changing the direction
                next_row, next_column = row + directions[direction_index][0], column + directions[direction_index][1]
            # Move to the next position
            row, column = next_row, next_column

        # Return the filled matrix
        return matrix
```

Java Solution

```java
public class Solution {

    public int[][] generateMatrix(int n) {
        // Initialize the matrix to be filled.
        int[][] matrix = new int[n][n];

        // Starting point for the spiral is (0,0), top-left corner of the matrix.
        int row = 0, col = 0;

        // 'dirIndex' is used to determine the current direction of the spiral.
        int dirIndex = 0;

        // Define directions for right, down, left, up movement.
        int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

        // Fill up the matrix with values from 1 to n squared.
        for (int value = 1; value <= n * n; ++value) {
            // Place the value into the matrix.
            matrix[row][col] = value;

            // Calculate the next position using the current direction.
            int nextRow = row + directions[dirIndex][0];
            int nextCol = col + directions[dirIndex][1];

            // Check boundary conditions and whether the cell is already filled.
            if (nextRow < 0 || nextCol < 0 || nextRow >= n || nextCol >= n || matrix[nextRow][nextCol] > 0) {
                // Change direction: right -> down -> left -> up -> right ...
                dirIndex = (dirIndex + 1) % 4;
                // Calculate the position again after changing direction.
                nextRow = row + directions[dirIndex][0];
                nextCol = col + directions[dirIndex][1];
            }

            // Move to the next cell.
            row = nextRow;
            col = nextCol;
        }

        // Return the filled spiral matrix.
        return matrix;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Directions array to help navigate right, down, left, and up.
    const int directions[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    // Generates a n-by-n matrix filled with elements from 1 to n^2 in spiral order.
    vector<vector<int>> generateMatrix(int n) {
        vector<vector<int>> matrix(n, vector<int>(n));
        int row = 0, col = 0, dirIndex = 0; // Initialize the starting point and direction index.

        for (int value = 1; value <= n * n; ++value) {
            // Fill in the current cell with the current value.
            matrix[row][col] = value;

            // Calculate the next cell's row and column indexes based on the current direction.
            int nextRow = row + directions[dirIndex][0];
            int nextCol = col + directions[dirIndex][1];

            // If the next cell is out of bounds or already filled, change direction.
            if (nextRow < 0 || nextCol < 0 || nextRow >= n || nextCol >= n || matrix[nextRow][nextCol] != 0) {
                dirIndex = (dirIndex + 1) % 4; // Update direction index to turn clockwise.

                // Recalculate the next cell's row and column indexes after changing direction.
                nextRow = row + directions[dirIndex][0];
                nextCol = col + directions[dirIndex][1];
            }

            // Move to the next cell.
            row = nextRow;
            col = nextCol;
        }

        return matrix; // Return the filled matrix.
    }
};
```

Typescript Solution

```typescript
function generateMatrix(n: number): number[][] {
    // Initializing the matrix with 'undefined' values
    let matrix = Array.from({ length: n }, () => new Array(n).fill(undefined));

    // Directions represent right, down, left, up movements respectively.
    let directions = [
        [0, 1],  // Move right
        [1, 0],  // Move down
        [0, -1], // Move left
        [-1, 0], // Move up
    ];

    // Starting position in the top-left corner of the matrix
    let row = 0, col = 0;

    // Filling out the matrix with values from 1 to n*n
    for (let value = 1, directionIndex = 0; value <= n * n; value++) {
        // Assign the current value to the current position
        matrix[row][col] = value;

        // Calculate the next position using current direction
        let nextRow = row + directions[directionIndex][0],
            nextCol = col + directions[directionIndex][1];

        // Check if the next position is out of bounds or already filled
        if (nextRow < 0 || nextRow === n || !nextCol < 0 || nextCol === n || matrix[nextRow][nextCol] !== undefined) {
            // Change direction if out of bounds or cell is already filled
            directionIndex = (directionIndex + 1) % 4;

            // Update next position after changing direction
            nextRow = row + directions[directionIndex][0];
            nextCol = col + directions[directionIndex][1];
        }

        // Update current position to the next position
        row = nextRow;
        col = nextCol;
    }

    // Returning the filled matrix
    return matrix;
}
```

Time and Space Complexity

The given Python code generates a spiral matrix of size `n x n`, where `n` is the input to the method `generateMatrix`. Let's analyze both the time complexity and space complexity of the code.

Time Complexity:

The time complexity of this algorithm is determined by the number of elements that need to be filled into the matrix, which corresponds to every position in the matrix being visited once. Since the matrix has `n x n` positions, the algorithm has to perform `n * n` operations, one for each element.

Thus, the time complexity is `O(n^2)`.

Space Complexity:

The space complexity includes the space taken up by the output matrix, and any additional space used by the algorithm for processing. In this case, the output matrix itself is of size `n x n`, so that takes `O(n^2)` space. The algorithm uses a small, constant amount of extra space for variables and the direction tuple `dirs`.

This means the additional space used by the algorithm does not grow with `n`, which makes it `O(1)`. However, since the output matrix size is proportional to the square of `n`, we consider it in the overall space complexity.

Thus, the overall space complexity of the algorithm is `O(n^2)`.