2326. Spiral Matrix IV

Medium Array Linked List Matrix Simulation Leetcode Link

Problem Description

You are tasked with creating an `m x n` matrix that represents the elements of a given linked list in a spiral format. A linked list is a linear collection of elements, where each element points to the next one. The number of rows (`m`) and columns (`n`) are provided, and these will be the dimensions of the matrix you need to fill.

The filling should start from the top-left corner of the matrix and proceed in a clockwise spiral pattern. In a spiral pattern, you move right until you hit the edge or an already filled cell, then move down, move left, and move up, repeating this process until all elements from the linked list are placed within the matrix. If the matrix is larger than what is needed for the linked list, the remaining cells should be filled with `-1`.

The objective is to return the completed matrix.

Intuition

The solution to this problem comes down to simulating the process of filling the matrix in a spiral. One way to achieve this is to keep track of the direction we are moving in and change it whenever we hit the edge of the matrix or an already filled cell (indicated by `-1`).

We start from the top-left corner, initially moving to the right. This is followed by moving downwards once we can't move right anymore, then to the left, and finally, moving upwards — cycling through these directions (right, down, left, up) as needed.

To implement this efficiently:

1. We initialize a matrix full of `-1`s to ensure we know when a cell is filled already and to handle the case when the linked list ends before filling the entire matrix.
2. We create a direction tracking list `dirs` composed of pairs, where each pair represents a change on the `i` and `j` indexes of the matrix (row and column movements).
3. A pointer `p` within the `dirs` is utilized to change the direction correctly when necessary.
4. We iterate over the linked list and the matrix simultaneously, moving the current index based on the direction from `dirs` and checking boundaries to know when to change direction.

This approach allows us to fill in the matrix in spiral order by updating the indexes to represent the movement on each iteration, only altering direction when we hit a boundary or previously filled cell. The process stops when we reach the end of the linked list.

Solution Approach

The provided solution in Python is quite straightforward and efficiently implements the spiral matrix population from a linked list. Here's a step-by-step explanation of the approach using algorithms and patterns:

1. **Matrix Initialization**: We initiate an `m x n` matrix `ans` filled with `-1`s to signify unfilled cells, which later helps to check whether we can continue in our current direction or need to turn.

2. **Direction Control**: A `dirs` list contains the directional changes as pairs of row and column increments for moving right, down, left, and up, respectively `[[0, 1], [1, 0], [0, -1], [-1, 0]]`. This is equivalent to following an (x, y) axis movement where (0, 1) moves right, (1, 0) moves down, (0, -1) moves left, and (-1, 0) moves up.

3. **Traversal Control**: `i` and `j` denote the current row and column indices in the matrix, and `p` keeps track of the current direction using the index of the `dirs` list.

4. **List Iteration and Matrix Population**: Starting from the top-left corner, we iterate over the linked list and populate the matrix elements with the current list node's value. After embedding the value from the current list node into the matrix, we move the `head` reference forward to the next node.

5. **Boundary and Direction Check**: After placing each value in the matrix, we check if the next move would go out of bounds or into an already filled cell by tentatively applying the current direction to `i` and `j`. If the tentative move is invalid, we rotate the direction by moving to the next element in the `dirs` list, accomplished by the `p = (p + 1) % 4`.

6. **Ending Condition**: The loop continues until the `head` is `None`, meaning we've reached the end of the linked list.

7. **Final Result**: The algorithm returns the populated `ans` matrix once the entire linked list is traversed or once all cells are filled, whichever comes first.

The code uses common data structures like lists and simple control flow constructs such as while-loops and if-else statements, giving us an elegant solution to build a matrix in spiral order from a predefined linked list.

Example Walkthrough

Let's walk through a small example:

Suppose we're given the dimensions of the matrix `m x n` as `3 x 3` and a linked list with the elements `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9`.

1. **Matrix Initialization**: We first initialize a `3 x 3` matrix filled with `-1`.

```
ans = [
  [-1, -1, -1],
  [-1, -1, -1],
  [-1, -1, -1]
]
```

2. **Direction Control Initialization**: We set up our direction array `dirs` to `[0, 1], [1, 0], [0, -1], [-1, 0]`, which represents right, down, left, and up movements, respectively. We also set our current direction index `p` to `0`.

3. **Assign Variables for Traversal**: We initialize our current position indices as `i = 0`, `j = 0` for the top-left start.

4. **Traverse and Populate Matrix**: Starting with the head of the linked list, we start populating the matrix.

   - Place `1` at `ans[0][0]` and move right to `ans[0][1]`.
   - Place `2`, `3` turning when needed until we've reached `ans[1][2]` with `6`.

```
ans = [
  [ 1, 2, 3],
  [-1, -1, 4],
  [-1, -1, 5]
]
```

5. **Boundary and Direction Check**: The next move is down, but since we're at the end of a column, we turn left.
   - We pop: `7 at ans[2][2]`, `8 at ans[2][1]`, and `9 at ans[2][0]`.
   - Next, we attempt to move up from `ans[2][0]`, but the above cell is also `-1`; we've finished the matrix spiral.

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

6. **Finishing Touches**: We've reached the end of the linked list; therefore, we fill any remaining `-1` with `-1`, which are already in place.

7. **Final Result**: We have successfully filled the `3 x 3` matrix in a spiral with the linked list values. The final matrix is:

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

This matrix is our output. We iterated through the linked list, populating the matrix in a spiral pattern, adhering to our directional rules and boundary checks, and terminating when we processed all list nodes.

Python Solution

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def spiralMatrix(self, rows: int, cols: int, head: Optional[ListNode]) -> List[List[int]]:
        # Initialize the matrix with -1s
        matrix = [[-1] * cols for _ in range(rows)]

        # The starting position (top-left corner)
        row, col = 0, 0

        # Direction indexes represent right, down, left, up movements
        direction = 0
        directions = [[0, 1], [1, 0], [0, -1], [-1, 0]]

        # As long as the linked list has nodes
        while head:
            # Place the current node's value in the matrix
            matrix[row][col] = head.val

            # Move to the next node in the linked list
            head = head.next

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

            # Check if the next position is invalid or already filled
            if (next_row < 0 or next_col < 0 or
                next_row >= rows or next_col >= cols or
                matrix[next_row][next_col] != -1):
                # Change direction (right -> down -> left -> up -> right...)
                direction = (direction + 1) % 4
                next_row, next_col = row + directions[direction][0], col + directions[direction][1]

            # Update the current position to the next position
            row, col = next_row, next_col

        # Return the filled matrix
        return matrix

```

Java Solution

```java
// Import required for Arrays.fill() method
import java.util.Arrays;

/**
 * Definition for singly-linked list.
 */
class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

/**
 * Solution class that contains the method to convert a linked list to a spiral matrix.
 */
class Solution {
    /**
     * Fills a matrix of size m * n with the values from a linked list in spiral order.
     *
     * @param m The number of rows of the matrix.
     * @param n The number of columns of the matrix.
     * @param head The head of the linked list.
     * @return A 2D integer array representing the filled spiral matrix.
     */
    public int[][] spiralMatrix(int m, int n, ListNode head) {
        // Initialize the matrix with the desired dimensions
        int[][] resultMatrix = new int[m][n];

        // Fill the matrix with -1 to indicate unfilled cells
        for (int[] row : resultMatrix) {
            Arrays.fill(row, -1);
        }

        // Initialize row and column indices to start from the top left corner
        int row = 0, column = 0;

        // Initialize the direction index
        int directionIndex = 0;

        // Define the directions for right, down, left, and up in a 2D array
        int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

        // Continue to fill the matrix until the linked list is exhausted
        while (head != null) {
            // Fill the current cell with the linked list's node value
            resultMatrix[row][column] = head.val;

            // Move to the next node in the linked list
            head = head.next;

            // Determine the new coordinates based on the current direction
            int nextRow = row + directions[directionIndex][0];
            int nextColumn = column + directions[directionIndex][1];

            // Check for boundary conditions and if the next cell is already filled
            if (nextRow < 0 || nextColumn < 0 || nextRow >= m ||
                nextColumn >= n || resultMatrix[nextRow][nextColumn] >= 0) {
                // Change the direction if we hit a boundary or a filled cell
                directionIndex = (directionIndex + 1) % 4;
            } else {
                // Update the row and column indices if the next cell is valid
                row = nextRow;
                column = nextColumn;
            }
        }

        // Return the filled spiral matrix
        return resultMatrix;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

class Solution {
public:
    vector<vector<int>> spiralMatrix(int rows, int cols, ListNode* head) {
        // Initialize the matrix with -1s, which indicates unvisited positions.
        vector<vector<int>> result(rows, vector<int>(cols, -1));

        // Starting at the top-left corner of the matrix.
        int currentRow = 0, currentCol = 0, directionIndex = 0;

        // Directions we will move: right, down, left, up.
        vector<vector<int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

        // Continue to fill the matrix until we reach the end of the linked list.
        while (head) {
            // Place the current node's value into the matrix.
            result[currentRow][currentCol] = head->val;
            // Move to the next node in the linked list.
            head = head->next;

            // Try to move to the next position in the spiral.
            while (head) {
                int nextRow = currentRow + directions[directionIndex][0];
                int nextCol = currentCol + directions[directionIndex][1];

                // If the next position is out of bounds or already visited,
                // change direction by rotating clockwise.
                if (nextRow < 0 || nextCol < 0 || nextRow >= rows || nextCol >= cols || result[nextRow][nextCol] != -1) {
                    directionIndex = (directionIndex + 1) % 4;
                } else {
                    // If the next position is valid, move to it.
                    currentRow = nextRow;
                    currentCol = nextCol;
                    break;
                }
            }

            // If we have reached the end of the list, break the outer loop.
            if (!head) {
                break;
            }
        }

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

```

Typescript Solution

```typescript
// Function to populate an m x n matrix with the values from a given linked list in spiral order.
function spiralMatrix(m: number, n: number, head: ListNode | null): number[][] {
    // Direction vectors for moving right, down, left, and up
    const directionVectors = [
        [0, 1], // right
        [1, 0], // down
        [0, -1], // left
        [-1, 0], // up
    ];

    // Initialize matrix with -1, indicating unfilled positions
    let matrix = Array.from({ length: m }, () => new Array(n).fill(-1));

    // Variables to track the current position and direction
    let row = 0,
        col = 0,
        dirIndex = 0; // Direction index to index into directionVectors

    // Iterate through the linked list until we reach the end
    while (head !== null) {
        // Place the current value into the matrix
        matrix[row][col] = head.val;
        // Move to the next node in the list
        head = head.next;

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

        // Check bounds and if the next position has already been visited
        if (nextRow < 0 || nextRow >= m || nextCol < 0 || nextCol >= n || matrix[nextRow][nextCol] !== -1) {
            // Change direction if out of bounds or if position is filled
            dirIndex = (dirIndex + 1) % 4;
        }

        // Update current position to the new position based on updated direction
        row = row + directionVectors[dirIndex][0];
        col = col + directionVectors[dirIndex][1];
    }

    // Return the filled matrix
    return matrix;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(m * n)`, where `m` is the number of rows and `n` is the number of columns in the matrix. This is because the code iterates over each cell of the `m x n` matrix exactly once to fill it with the values from the linked list. Note that while there are nested loops, the inner loop only serves to change the direction when necessary and doesn't iterate over the matrix again, as the outer loop breaks as soon as the linked list ends.

Space Complexity

The space complexity of the code is `O(m * n)` due to the `ans` matrix that is created with `m` rows and `n` columns, each cell initialized to `-1`. No other additional significant space is used, except for constant extra space for variables `i`, `j`, `x`, `y`, and `p`. The linked list itself is not counted towards space complexity as it's given as part of the input. Therefore, the space consumed by the output matrix is the dominant factor.