Problem Description

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

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

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 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

The objective is to return the completed matrix.

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).

Intuition

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 -1s 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 -1s 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,

using the index of the dirs list.

whichever comes first.

Example Walkthrough

7 -> 8 -> 9.

Let's walk through a small example:

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.

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.

3. Traversal Control: i and j denote the current row and column indices in the matrix, and p keeps track of the current direction

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.

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

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

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

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.

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 ->

1. Matrix Initialization: We first initialize a 3 x 3 matrix filled with -1. 1 ans = [[-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.

[1, 2, 3],

[9, 8, 7]

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

[-1, -1, -1], [-1, -1, -1]

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

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.

We pop: 7 at ans[2][2], 8 at ans[2][1], and 9 at ans[2][0].

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

○ Next, we attempt to move up from ans [2] [0], but the above cell is also -1; we've finished the matrix spiral. 1 ans = [

5. Boundary and Direction Check: The next move is down, but since we're at the end of a column, we turn left.

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

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.

def spiralMatrix(self, rows: int, cols: int, head: Optional[ListNode]) -> List[List[int]]:

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

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

class ListNode:

class Solution:

9

10

11

12

13

14

15

16

17

18

24

25

26

27

28

29

30

31

38

39

40

42

10

11

12

14

17

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

22

23

24

25

26

27

28

29

30

31

32

33

34

35 36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

6

13

14

15

16

17

18

19

20

21

};

while (head) {

head = head->next;

while (head) {

} else {

if (!head) {

return result;

const directionVectors = [

[0, 1], // right

[1, 0], // down

break;

// Return the filled matrix.

break;

13 }

15 /**

*/

def __init__(self, val=0, next=None):

Initialize the matrix with -1s

 $matrix = [[-1] * cols for _ in range(rows)]$

directions = [[0, 1], [1, 0], [0, -1], [-1, 0]]

Move to the next node in the linked list

next_row >= rows or next_col >= cols or

if (next_row < 0 or next_col < 0 or

row, col = next_row, next_col

Return the filled matrix

1 // Import required for Arrays.fill() method

* Definition for singly-linked list.

ListNode(int val) { this.val = val; }

* @param m The number of rows of the matrix.

* @param head The head of the linked list.

int[][] resultMatrix = new int[m][n];

for (int[] row : resultMatrix) {

Arrays.fill(row, -1);

* @param n The number of columns of the matrix.

return matrix

import java.util.Arrays;

class ListNode {

int val;

class Solution {

/**

*/

ListNode next;

ListNode() {}

Direction indexes represent right, down, left, up movements

Calculate the next position based on the current direction

Check if the next position is invalid or already filled

ListNode(int val, ListNode next) { this.val = val; this.next = next; }

* @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

// Fill the matrix with -1 to indicate unfilled cells

* Solution class that contains the method to convert a linked list to a spiral matrix.

* Fills a matrix of size m * n with the values from a linked list in spiral order.

next_row, next_col = row + directions[direction][0], col + directions[direction][1]

The starting position (top-left corner)

self.val = val

self.next = next

row, col = 0, 0

direction = 0

head = head.next

1 ans = [

As long as the linked list has nodes while head: # Place the current node's value in the matrix 20 matrix[row][col] = head.val 21 22 23

```
32
                    matrix[next_row][next_col] != -1):
33
                    # Change direction (right -> down -> left -> up -> right...)
34
                    direction = (direction + 1) % 4
35
                    next_row, next_col = row + directions[direction][0], col + directions[direction][1]
36
               # Update the current position to the next position
37
```

Java Solution

```
34
35
36
           // Initialize row and column indices to start from the top left corner
37
           int row = 0, column = 0;
38
39
           // Initialize the direction index
40
           int directionIndex = 0;
```

```
41
 42
             // Define the directions for right, down, left, and up in a 2D array
 43
             int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
 44
 45
             // Continue to fill the matrix until the linked list is exhausted
 46
             while (head != null) {
                 // Fill the current cell with the linked list's node value
 47
                 resultMatrix[row][column] = head.val;
 48
 49
 50
                 // Move to the next node in the linked list
 51
                 head = head.next;
 52
 53
                 // Determine the new coordinates based on the current direction
 54
                 int nextRow = row + directions[directionIndex][0];
 55
                 int nextColumn = column + directions[directionIndex][1];
 56
 57
                 // Check for boundary conditions and if the next cell is already filled
 58
                 if (nextRow < 0 || nextColumn < 0 || nextRow >= m ||
                     nextColumn >= n || resultMatrix[nextRow][nextColumn] >= 0) {
 59
                     // Change the direction if we hit a boundary or a filled cell
 60
 61
                     directionIndex = (directionIndex + 1) % 4;
 62
                 } else {
 63
                     // Update the row and column indices if the next cell is valid
 64
                     row = nextRow;
 65
                     column = nextColumn;
 66
 67
 68
 69
             // Return the filled spiral matrix
 70
             return resultMatrix;
 71
 72
 73
C++ Solution
  1 #include <vector>
  2 using namespace std;
    // Definition for singly-linked list.
  5 struct ListNode {
         int val;
         ListNode *next;
        ListNode(): val(0), next(nullptr) {}
  8
        ListNode(int x) : val(x), next(nullptr) {}
  9
         ListNode(int x, ListNode *next) : val(x), next(next) {}
 10
 11 };
 12
 13 class Solution {
 14 public:
 15
         vector<vector<int>> spiralMatrix(int rows, int cols, ListNode* head) {
             // Initialize the matrix with -ls, which indicates unvisited positions.
 16
             vector<vector<int>> result(rows, vector<int>(cols, -1));
 17
 18
 19
             // Starting at the top-left corner of the matrix.
 20
             int currentRow = 0, currentCol = 0, directionIndex = 0;
 21
```

// Directions we will move: right, down, left, up.

result[currentRow][currentCol] = head->val;

// Move to the next node in the linked list.

// Place the current node's value into the matrix.

// Try to move to the next position in the spiral.

// change direction by rotating clockwise.

currentRow = nextRow;

currentCol = nextCol;

directionIndex = (directionIndex + 1) % 4;

// If the next position is valid, move to it.

// If we have reached the end of the list, break the outer loop.

// 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[][] {

dirIndex = 0; // Direction index to index into directionVectors

// Direction vectors for moving right, down, left, and up

// Variables to track the current position and direction

// Iterate through the linked list until we reach the end

// Place the current value into the matrix

again, as the outer loop breaks as soon as the linked list ends.

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.

int nextRow = currentRow + directions[directionIndex][0];

int nextCol = currentCol + directions[directionIndex][1];

// If the next position is out of bounds or already visited,

if (nextRow < 0 || nextCol < 0 || nextRow >= rows || nextCol >= cols || result[nextRow][nextCol] != -1) {

```
[0, -1], // left
            [-1, 0], // up
       1;
 9
10
11
       // Initialize matrix with -1, indicating unfilled positions
12
        let matrix = Array.from({ length: m }, () => new Array(n).fill(-1));
```

let row = 0,

col = 0,

while (head !== null) {

Typescript Solution

```
22
            matrix[row][col] = head.val;
 23
            // Move to the next node in the list
 24
            head = head.next;
 25
 26
             // Calculate the next position using the current direction
 27
             let nextRow = row + directionVectors[dirIndex][0];
 28
             let nextCol = col + directionVectors[dirIndex][1];
 29
 30
            // Check bounds and if the next position has already been visited
 31
             if (nextRow < 0 || nextRow >= m || nextCol < 0 || nextCol >= n || matrix[nextRow][nextCol] !== -1) {
 32
                // Change direction if out of bounds or if position is filled
 33
                dirIndex = (dirIndex + 1) % 4;
 34
 35
 36
             // Update current position to the new position based on updated direction
 37
             row = row + directionVectors[dirIndex][0];
 38
             col = col + directionVectors[dirIndex][1];
 39
 40
 41
         // Return the filled matrix
 42
         return matrix;
 43 }
 44
Time and Space Complexity
Time Complexity
The time complexity of the given code is 0(m * n), where m is the number of rows and n is the number of columns in the matrix. This
```

Space Complexity

The space complexity of the code is 0(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.

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