Spiral Matrix Problem

The spiral matrix problem involves printing the elements of a given N x M matrix in a spiral form, starting from the outer layer and moving inwards.

Problem Description

Given an N x M matrix, the task is to print the matrix elements in a spiral order. This means starting from the top-left corner, traversing right, then down, then left, then up, and continuing this pattern until all elements are visited.

Example: Starting from 1, go right, then down to 6, left to 26, and end up at 36.

Key Objectives

- **Implementation skills**: Assesses your ability to translate a problem description into code.
- **Code cleanliness**: Evaluates how cleanly and efficiently you can write the code.

Approach to Solving the Spiral Matrix Problem

The problem is approached by identifying the pattern of movement in a spiral: right, bottom, left, top. Each direction represents a layer of the spiral.

Pattern Identification

The spiral traversal follows a specific pattern:

1. Move **right**
2. Move **bottom**
3. Move **left**
4. Move **top**

This pattern repeats for each layer of the matrix until the center is reached.

Variables to Track

To implement the spiral traversal, the following variables are tracked:

- `top`: Index of the topmost row.
- `bottom`: Index of the bottommost row (n - 1).
- `left`: Index of the leftmost column.
- `right`: Index of the rightmost column (m - 1).

Traversal Steps

1. **Right**: Traverse from left to right along the top row.
2. **Bottom**: Traverse from top to bottom along the right column.
3. **Left**: Traverse from right to left along the bottom row.
4. **Top**: Traverse from bottom to top along the left column.

After each traversal, the boundaries are adjusted to move to the next inner layer.

Implementing the Traversal

The implementation involves using loops to traverse the matrix in each direction, updating the boundaries after each traversal.

Initial Setup

- Initialize `left` to 0, `right` to `M - 1`, `top` to 0, and `bottom` to `N - 1`.
- Create a list to store the spiral order of elements.

Right Traversal

Iterate from `left` to `right` along the `top` row:

```
for i in range(left, right + 1):
    result.append(matrix[top][i])
```

After the right traversal, increment `top` by 1 to move the top boundary down.

Bottom Traversal

Iterate from `top` to `bottom`:

```
for i in range(top, bottom + 1):
    result.append(matrix[i][right])
```

After the bottom traversal, decrement `right` by 1 to move the right boundary to the left.

Left Traversal

Iterate from `right` to `left`:

```
for i in range(right, left - 1, -1):
    result.append(matrix[bottom][i])
```

After the left traversal, decrement `bottom` by 1 to move the bottom boundary up.

Top Traversal

Iterate from `bottom` to `top`:

```
for i in range(bottom, top - 1, -1):
    result.append(matrix[i][left])
```

After the top traversal, increment `left` by 1 to move the left boundary to the right.

Looping the Process

The above steps are repeated until all elements are processed. The loop continues as long as `left <= right` and `top <= bottom`.

Code Implementation

```python
def spiral_matrix(matrix):
    result = []
    top = 0
    bottom = len(matrix) - 1
    left = 0
    right = len(matrix[0]) - 1

    while top <= bottom and left <= right:
        # Right
        for i in range(left, right + 1):
            result.append(matrix[top][i])
        top += 1

        # Bottom
        for i in range(top, bottom + 1):
            result.append(matrix[i][right])
        right -= 1

        # Check if there are more layers to process
        if top <= bottom and left <= right:
            # Left
            for i in range(right, left - 1, -1):
                result.append(matrix[bottom][i])
            bottom -= 1

            # Top
            for i in range(bottom, top - 1, -1):
                result.append(matrix[i][left])
            left += 1

    return result
```

Code Explanation

1. **Initialization**: The function initializes the boundaries (`top`, `bottom`, `left`, `right`) and the `result` list.
2. **Loop Condition**: The `while` loop continues as long as the `top` boundary is less than or equal to the `bottom` boundary AND the `left` boundary is less than or equal to the `right` boundary.
3. **Right Traversal**: The first `for` loop traverses from `left` to `right` along the `top` row, appending each element to the `result` list. After this, the `top` boundary is incremented to move to the next row.
4. **Bottom Traversal**: The second `for` loop traverses from `top` to `bottom` along the `right` column, appending each element to the `result` list. The `right` boundary is then decremented to move to the previous column.
5. **Conditional Check**: Before the left and top traversals, there's a check `if top <= bottom and left <= right:`. This is important because, in cases where the matrix is not square, either the `top` can become greater than the `bottom` or the `left` can become greater than the `right`, which means there are no more elements to traverse, and we should avoid the extra traversals to prevent duplicates.
6. **Left Traversal**: The third `for` loop traverses from `right` to `left` along the `bottom` row, appending each element to the `result` list. The `bottom` boundary is then decremented to move to the previous row.
7. **Top Traversal**: The fourth `for` loop traverses from `bottom` to `top` along the `left` column, appending each element to the `result` list. The `left` boundary is then incremented to move to the next column.
8. **Return Result**: After the `while` loop completes (i.e., all layers of the spiral have been traversed), the function returns the `result` list, which contains all the elements of the matrix in spiral order.

Edge Cases and Considerations

- The code includes a check (`if top <= bottom and left <= right`) before the left and top traversals to ensure that it doesn't process the same layer twice in cases of non-square matrices or when the spiral reaches the center.
- The initial determination of rows and columns accounts for the structure of the input matrix as a list of lists.

Determining Rows and Columns

The number of rows and columns is determined as follows:

- Rows: Number of lists inside the main list.
- Columns: Number of elements in the first list.

```
rows = len(matrix)
cols = len(matrix[0])
```

Updating Boundaries

After each traversal, the boundaries are updated to move inward:

- `top++` (move top boundary down)
- `right--` (move right boundary left)
- `bottom--` (move bottom boundary up)
- `left++` (move left boundary right)

Summary Table

Step	Action	Boundary Update
Right	Traverse from left to right along top	`top++`
Bottom	Traverse from top to bottom along right	`right--`
Left	Traverse from right to left along bottom	`bottom--`
Top	Traverse from bottom to top along left	`left++`

Notes

- The outer covering is printed first, followed by the second covering, and so on until the center.
- Four loops are used to handle the four directions of movement (right, bottom, left, top).

Spiral Matrix Traversal Logic

The algorithm spirals through a matrix, and after each side (top, right, bottom, left) is traversed, the boundaries are updated. The key is to ensure that after updating the boundaries, the algorithm checks whether there are any rows or columns left to traverse.

Checking for Remaining Traversal

The algorithm checks if there are remaining rows and columns using the following conditions:

- If `left` is less than or equal to `right`, there are columns to be traversed.
- If `top` is less than or equal to `bottom`, there are rows to be traversed.

```
while (top <= bottom && left <= right) {
    // Traversal logic here
}
```

Performing Rightward Traversal

For a single row matrix, the algorithm performs a rightward traversal as the first case because `top` is less than or equal to `bottom` and `left` is less than or equal to `right`.

Avoiding Duplicate Printing

After completing the top row, the `top` is incremented. This ensures that the algorithm doesn't reprint the same row when moving to the next layer of the spiral.

The bottom row is checked using a for loop condition to avoid printing the bottom row when only one row is present.

Handling Edge Cases

The algorithm handles edge cases such as single rows or columns by checking if there are still rows or columns to be printed before traversing.

- Before moving from right to left, it checks if `top` is still under `bottom` to ensure there's a row to print.
- Before moving from bottom to top, it checks if `left` is still under `right` to ensure there's a column to print.

Time Complexity

The time complexity of the algorithm is $O(n * m)$ because each element in the matrix is visited once. The space complexity is also $O(n * m)$ due to storing the answer.