# Topic Introduction

Let’s talk about a core skill for coding interviews: **2D matrix manipulation**.

If you’ve solved Leetcode-style grid problems before, you may know that matrices are just 2D lists or arrays. Many interview questions will require you to read, update, and traverse these 2D grids in clever ways.

**What is a matrix in programming?**
A matrix is simply a rectangular array of numbers, arranged in rows and columns. In Python, it’s usually represented as a list of lists:
`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

**Why are matrices important?**
Matrices are used to model everything from game boards to images, graphs, and more. Interviewers love them because they test your ability to manipulate data structures, manage indices, and spot patterns.

**Key patterns in matrix manipulation:**

- Row and column traversal
- In-place updates
- Boundary conditions
- Layer-by-layer (spiral, ring) processing

**Simple example (not from today’s problems):**
Suppose you want to sum all elements in a matrix. You’d use two nested loops:

```
total = 0
for row in matrix:
    for val in row:
        total += val
```

This double loop is the backbone of many matrix problems.

## Why These 3 Problems?

The following problems are all about **manipulating 2D matrices** in different ways:

- **Set Matrix Zeroes:** Mark rows and columns as zero if any cell is zero.
- **Spiral Matrix:** Traverse and return all elements in spiral order.
- **Rotate Image:** Rotate a matrix by 90 degrees in place.

They’re often grouped because they require careful index management, attention to in-place updates, and a solid understanding of 2D traversal patterns. We’ll start with the most conceptually tricky (Set Matrix Zeroes), move to Spiral traversal, and finish with in-place rotation.

## Problem 1: Set Matrix Zeroes

Problem link: Leetcode 73 - Set Matrix Zeroes

**Problem Statement (Rephrased):**

Given an m x n matrix, if an element is 0, set its entire row and column to 0. Do this in-place.

**Example Input/Output:**

Input:

```
[
 [1,1,1],
 [1,0,1],
 [1,1,1]
]
```

Output:

```
[
 [1,0,1],
 [0,0,0],
 [1,0,1]
]
```

Explanation: The zero at position (1,1) causes the entire row 1 and column 1 to become zero.

**Thought Process:**

This is trickier than it looks! If you naively set zeros as you find them, you'll accidentally turn the whole matrix into zeros. Try drawing it on paper to see why.

**Try this test case by hand:**

Input:

```
[
 [0,1,2,0],
 [3,4,5,2],
 [1,3,1,5]
]
```

**Brute-force approach:**

- For every zero you find, mark its row and column to be zeroed.
- But: if you zero as you go, you'll "infect" the rest of the matrix.
- To avoid this, you could make a copy of the matrix and use it to track original zeros.
- Time: O(mn), Space: O(mn) (for the copy).

**Optimal approach:**

- The core pattern is **using the first row and first column as markers** to store which rows and columns need to be zeroed, minimizing extra space.
- Steps:
  - First, scan the matrix. For each zero, mark its row and column by setting the first cell in that row and column to zero.
  - Use two flags to remember if the first row or first column originally contained a zero.
  - In a second pass, use the markers to set the appropriate cells to zero.
  - Finally, zero out the first row and/or column if needed.

**Python Solution:**

```
def setZeroes(matrix):
    m, n = len(matrix), len(matrix[0])
    first_row_zero = any(matrix[0][j] == 0 for j in range(n))
    first_col_zero = any(matrix[i][0] == 0 for i in range(m))

    # Use first row and column to mark zeros
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][j] == 0:
                matrix[i][0] = 0
                matrix[0][j] = 0

    # Set cell to zero if its row or column is marked
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][0] == 0 or matrix[0][j] == 0:
                matrix[i][j] = 0

    # Zero the first row if needed
    if first_row_zero:
        for j in range(n):
            matrix[0][j] = 0

    # Zero the first column if needed
    if first_col_zero:
        for i in range(m):
            matrix[i][0] = 0
```

**Explanation:**

- `first_row_zero` and `first_col_zero` help us remember if the first row/col should be zeroed at the end.
- We mark rows and columns in the first pass, skipping the first row and column to avoid overwriting our markers.
- In the second pass, we update cells based on the markers.
- Finally, we handle the first row and column separately.

**Trace with Example:**

Input:

```
[
 [1,1,1],
 [1,0,1],
 [1,1,1]
]
```

- First pass: Matrix markers become:

`

[
[1,0,1],
[0,0,1],
[1,1,1]
]
`

- Second pass: Set cells to zero if their row or column is marked.
- Handle first row/col: Only column 1 is zeroed in the first row.

**Try this test case yourself:**

Input:

```
[
 [1,2,3],
 [4,0,6],
 [7,8,9]
]
```

Take a moment to solve this on your own before checking the code!

**Time and Space Complexity:**

- Time: O(mn), as we scan the matrix a constant number of times.
- Space: O(1), since we only use a few variables (modifying in-place).

Did you know this can also be solved using extra sets to record zero rows and columns? Try implementing that for practice!

## Problem 2: Spiral Matrix

Problem link: Leetcode 54 - Spiral Matrix

**Problem Statement (Rephrased):**

Given an m x n matrix, return all elements in spiral order (clockwise, starting from the top-left).

**How is this related?**

Like Problem 1, you need to carefully manage row and column indices. But here, you're traversing the matrix in a spiral path rather than updating values.

**Example Input/Output:**

Input:

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

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

**Another test case for you:**

Input:

```
[
 [1, 2, 3, 4],
 [5, 6, 7, 8],
 [9,10,11,12]
]
```

Expected Output: `[1,2,3,4,8,12,11,10,9,5,6,7]`

**Brute-force approach:**

- You could mark visited cells and move in the spiral pattern, but this is unnecessarily complex.

**Optimal approach:**

- The core pattern is **maintaining boundaries**: left, right, top, bottom.
- At each step, you traverse one side of the current "layer," then move the corresponding boundary inward.
- Repeat until all elements are covered.

**Step-by-step Logic:**

- Start with boundaries: `top=0`, `bottom=m-1`, `left=0`, `right=n-1`.
- Traverse:
  - From left to right along the top.
  - From top+1 to bottom along the right.
  - From right-1 to left along the bottom (if top != bottom).
  - From bottom-1 to top+1 along the left (if left != right).
- After each, move the respective boundary inward (increment/decrement).
- Continue until `left > right` or `top > bottom`.

**Python code:**

```
def spiralOrder(matrix):
    result = []
    if not matrix:
        return result
    top, bottom = 0, len(matrix)-1
    left, right = 0, len(matrix[0])-1

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

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

```
        if top <= bottom:
            # Traverse from right to left
            for j in range(right, left-1, -1):
                result.append(matrix[bottom][j])
            bottom -= 1

        if left <= right:
            # Traverse upwards
            for i in range(bottom, top-1, -1):
                result.append(matrix[i][left])
            left += 1

    return result
```

**Time and Space Complexity:**

- Time: O(mn), since we visit each cell once.
- Space: O(mn), for the output list.

**Trace with Example:**

Input:

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

Steps:

- Top row: [1,2,3]
- Right column: [6,9]
- Bottom row reversed: [8,7]
- Left column up: [4]
- Remaining: [5]

**Try this case yourself:**

Input:

```
[
 [1,2],
 [3,4],
 [5,6]
]
```

Expected Output: `[1,2,4,6,5,3]`

## Problem 3: Rotate Image

Problem link: Leetcode 48 - Rotate Image

**Problem Statement (Rephrased):**

Given an n x n matrix, rotate it by 90 degrees (clockwise) **in-place**.

**What's different or more challenging?**

- You must rotate in-place (no extra matrix).
- You need to move groups of 4 elements at a time, layer by layer.

**Example Input/Output:**

Input:

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

Output:

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

**Another test case to try:**

Input:

```
[
 [5,1,9,11],
 [2,4,8,10],
 [13,3,6,7],
 [15,14,12,16]
]
```

Expected Output:

```
[
 [15,13,2,5],
 [14,3,4,1],
 [12,6,8,9],
 [16,7,10,11]
]
```

**Brute-force approach:**

- Make a copy of the matrix and write rotated values into the copy.
- But: this uses extra space, not allowed in the problem.

**Optimal approach:**

- The core pattern is **layer-by-layer rotation using index manipulation**.
- For each "ring" in the matrix, rotate the four corresponding elements.
- Alternatively, you can transpose the matrix and then reverse each row (a neat trick!).

**Pseudocode for layer-by-layer rotation:**

```
for each layer from outermost to innermost:
    first = layer
    last = n - 1 - layer
    for i from first to last-1:
        offset = i - first
        save top
        top = matrix[first][i]
        left -> top
        bottom -> left
        right -> bottom
        top -> right
```

**Or, for the transpose + reverse approach:**

- Transpose the matrix (swap matrix[i][j] with matrix[j][i])
- Reverse each row

**Time and Space Complexity:**

- Time: O(n^2), since every element is moved once.
- Space: O(1), in-place.

**Try this by hand:**

Input:

```
[
 [1,2],
 [3,4]
]
```

Expected Output:

```
[
 [3,1],
 [4,2]
]
```

**Guidance:**

- For each layer, swap four elements at a time.
- For the transpose + reverse trick: first swap across the diagonal, then reverse rows.
- If you finish, try implementing both approaches and see which makes more sense for you!

# Summary and Next Steps

You've just worked through three classic problems that build your **matrix manipulation** skills:

- **Set Matrix Zeroes:** Mark and propagate changes carefully, minimizing space.
- **Spiral Matrix:** Traverse with moving boundaries, avoiding repeats.

- **Rotate Image:** Layered, in-place transformation, or clever use of transpose + reverse.

**Key patterns to remember:**

- Be careful when updating in-place: use markers or flags if needed.
- For traversing in patterns (spiral, rings), track your boundaries.
- For rotations, think in terms of swapping sets of elements or using properties like transposition.

**Common mistakes:**

- Accidentally updating cells before you finish reading all original values.
- Off-by-one errors with boundaries.
- Forgetting to handle the first row/column separately when using them as markers.

## Action List

- Solve all 3 problems on your own, even if you followed the solution above.
- Try solving Problem 2 and 3 with a different approach (e.g., recursion for spiral traversal, or the transpose-reverse trick for rotation).
- Explore other matrix problems: search in a 2D matrix, word search, island counting, etc.
- Compare your code with top Leetcode submissions for style and edge-case handling.
- If you get stuck, don’t worry! The key is to keep practicing and reviewing tricky cases.

Happy coding! Keep pushing your comfort zone, and matrix problems will soon feel like second nature.