48. Rotate Image

Medium Array Math Matrix

Problem Description

You're given a square matrix, which is a 2D array, and your task is to rotate this image by 90 degrees clockwise. To clarify, rotating an image by 90 degrees clockwise means that the image is turned in such a way that if you start from the top left corner of the original image, it would become the top right corner of the rotated image. However, the challenge is to perform this rotation in-place, that is, without using any extra space for another array. You'll need to transform the matrix within the given data structure itself.

Intuition

To rotate the matrix by 90 degrees clockwise, you can think of a two-step process. First, imagine if you flip the image upside down, which can be visualized as folding the bottom row up to where the top row is, and vice versa. This can be done by swapping the elements in the first row with the corresponding elements in the last row, then the second row with the second-to-last row, and so on until you reach the middle of the matrix.

Once the matrix is flipped upside down, the next step is to flip it along its main diagonal. The main diagonal of a matrix is a line of entries starting from the top-left corner to the bottom-right corner. Flipping a matrix along the main diagonal means that for each element `matrix[i][j]`, you swap it with the element `matrix[j][i]`.

By following these two steps - flipping the matrix upside down and then swapping elements along the main diagonal - each element moves to its correct position as it would be after a 90-degree clockwise rotation.

Solution Approach

The solution provided is a two-step process that involves:

1. Flipping the matrix upside down.
2. Then, flipping the matrix along its main diagonal.

For the first step, the code iterates through the first half of the rows (hence the `range(n >> 1)` which is a bitwise shift operation equivalent to `range(n // 2)`) and swaps all elements in row `i` with the corresponding elements in row `n-i-1`. The swapping is done column by column for each row (`for j in range(n):`). This effectively turns the matrix upside down without using any additional space.

Once the matrix is flipped upside down, we proceed with the second step, which involves swapping elements across the main diagonal. We iterate row by row, and within each row, we only go up to the main diagonal (`for j in range(i)`) to avoid re-swapping the already swapped elements. At each iteration, we swap the element at position `[i][j]` with the element at position `[j][i]`. This flips the matrix along the main diagonal and completes the 90-degree rotation.

The patterns used in this algorithm are in-place replacement to avoid additional memory usage, and careful indexing to walk through the 2D matrix in the required order. The operations of this method are designed to achieve the rotation with only O(1) additional space, by performing swaps directly on the given `matrix`.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we have the following 3×3 matrix (a square matrix with 3 rows and 3 columns):

```
1 2 3
4 5 6
7 8 9
```

We want to rotate this matrix by 90 degrees clockwise.

Step 1: Flipping the matrix upside down.

We start by flipping the entire matrix upside down, which involves swapping rows first with last, and so on until reaching the middle of the matrix. We swap the 1st row with the 3rd row, and because we have an odd number of rows (3), the middle row (2nd row) stays in place.

After flipping upside down, the matrix will look like this:

```
7 8 9
4 5 6
1 2 3
```

We can see that the 1st row has become the last row and the last row has become the 1st row.

Step 2: Flipping the matrix along its main diagonal.

Next, we need to flip the matrix along its main diagonal to achieve the 90-degree rotation. We swap the elements symmetrically across the diagonal. The main diagonal (1 to 9) doesn't change place.

For our example, this means swapping element (2,1) with (1,2), then (3,1) with (1,3), and finally (3,2) with (2,3) — note how we don't swap elements on the main diagonal itself or elements that have already been swapped.

Here's how we perform each swap:

- Swap element (2,1) with (1,2) — swaps 4 with 8 which gives us:

```
7 4 9
8 5 6
1 2 3
```

- Swap element (3,1) with (1,3) — swaps 1 with 9 which gives us:

```
7 4 1
8 5 6
9 2 3
```

- Swap element (3,2) with (2,3) — swaps 2 with 6 which gives us:

```
7 4 1
8 5 2
9 6 3
```

After performing these swaps, the final matrix after a 90-degree clockwise rotation is:

```
7 4 1
8 5 2
9 6 3
```

If you visualize turning the original matrix by 90 degrees clockwise, you will see that this final matrix is indeed the original matrix rotated, and just as importantly, the transformation has been performed in place, using no additional storage space.

Solution Implementation

Python

```python
class Solution:
    def rotate(self, matrix):
        # Determine the size of the matrix
        size = len(matrix)

        # Perform a vertical flip of the matrix
        for i in range(size >> 1):  # size >> 1 is a faster way to do size // 2
            for j in range(size):
                # Swap the top and bottom elements in the column
                matrix[i][j], matrix[size - i - 1][j] = matrix[size - i - 1][j], matrix[i][j]

        # Perform a diagonal flip of the matrix, which is equivalent
        # to transposing the matrix
        for i in range(size):
            for j in range(i):
                # Swap the elements across the diagonal
                matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
```

Java

```java
class Solution {
    public void rotate(int[][] matrix) {
        // Obtain the length of the matrix which is a square (same width and height)
        int n = matrix.length;

        // Step 1: Perform a vertical flip of the matrix
        // Loop over the first half of the rows (vertically)
        for (int row = 0; row < (n >> 1); ++row) {
            // Loop over all columns
            for (int col = 0; col < n; ++col) {
                // Swap the element at the current position with the element at the mirrored row
                // across the horizontal axis
                int temp = matrix[row][col];
                matrix[row][col] = matrix[n - row - 1][col];
                matrix[n - row - 1][col] = temp;
            }
        }

        // Step 2: Transpose the matrix by flipping it along its diagonal
        // Loop over all rows
        for (int row = 0; row < n; ++row) {
            // Loop over the columns up to the current row (to avoid re-flipping)
            for (int col = 0; col < row; ++col) {
                // Swap the element at (row, col) with the element at (col, row)
                int temp = matrix[row][col];
                matrix[row][col] = matrix[col][row];
                matrix[col][row] = temp;
            }
        }
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // For std::swap
using std::vector;
using std::swap;

class Solution {
public:
    // Function to rotate the matrix by 90 degrees clockwise.
    void rotate(vector<vector<int>>& matrix) {
        int size = matrix.size();

        // First step: Reverse every row of the matrix.
        // (Equivalent to a vertical reflection of the matrix)
        for (int row = 0; row < size / 2; ++row) {
            for (int col = 0; col < size; ++col) {
                // Swap elements symmetrically along
                // the middle row of the matrix
                swap(matrix[row][col], matrix[size - row - 1][col]);
            }
        }

        // Second step: Swap the symmetric elements across the diagonal
        // (top-left to bottom-right).
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < i; ++j) {
                // Swap elements to achieve 90-degree rotation
                swap(matrix[i][j], matrix[j][i]);
            }
        }

        // At the end of these two steps, the matrix is rotated 90 degrees clockwise.
    }
};
```

TypeScript

```typescript
/**
 * Rotates an N x N matrix by 90 degrees clockwise.
 * @param matrix The N x N matrix to rotate (will be modified in place).
 */
function rotate(matrix: number[][]): void {
    // First, reverse the rows of the matrix to position elements for the rotation.
    matrix.reverse();

    // Then, swap the elements along the diagonal.
    // Only iterate through the first row up to row before the last,
    // since after that elements would have already been swapped.
    for (let row = 0; row < matrix.length; ++row) {
        // Swap elements across the diagonal, hence j starts from row + 1 to prevent swapping elements twice.
        for (let col = 0; col < row; ++col) {
            // Temporary variable to hold the current element to be swapped.
            const temp = matrix[row][col];

            // Swap the element with its transposed position element.
            matrix[row][col] = matrix[col][row];
            matrix[col][row] = temp;
        }
    }
}
```

```python
class Solution:
    def rotate(self, matrix):
        # Determine the size of the matrix
        size = len(matrix)

        # Perform a vertical flip of the matrix
        for i in range(size >> 1):  # size >> 1 is a faster way to do size // 2
            for j in range(size):
                # Swap the top and bottom elements in the column
                matrix[i][j], matrix[size - i - 1][j] = matrix[size - i - 1][j], matrix[i][j]

        # Perform a diagonal flip of the matrix, which is equivalent
        # to transposing the matrix
        for i in range(size):
            for j in range(i):
                # Swap the elements across the diagonal
                matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
```

Time and Space Complexity

The time complexity of the given code is `O(n^2)`, where `n` is the length of the matrix. This is because there are two nested loops that iterate over the elements of the matrix. The first set of nested loops is responsible for the vertical flipping of the matrix (first part of the rotation), and it iterates over half of the matrix rows (hence `n/2`) for all columns, but since we drop constants when expressing time complexity, it simplifies to `O(n^2)`.

The second part of the rotation is swapping elements across the diagonal, again involving a nested loop, but this time it only processes elements in the upper triangle (excluding the diagonal) of the matrix, which still leads to a total of `n*(n-1)/2` swaps. Despite the fact that only roughly half of the matrix elements are being swapped (upper triangle), this still results in a time complexity of `O(n^2)` because the leading term `n^2` dominates as `n` grows large.

As for the space complexity, the reference answer correctly states it is `O(1)`. The algorithm only uses a constant amount of extra space for variable storage, regardless of the size of the input matrix. The rotation is done in place, therefore no additional space proportional to the input size is required.