1886. Determine Whether Matrix Can Be Obtained By Rotation

Easy Array Matrix

Leetcode Link

Problem Description

The problem provides two `n x n` binary matrices: `mat` and `target`. A binary matrix is a matrix where each element is either `0` or `1`. The goal is to determine whether it is possible to make the matrix `mat` identical to the matrix `target` by rotating `mat` in 90-degree increments. The task is to check all possible rotations of `mat` and see if any rotation matches the `target`. If at least one rotation results in `mat` being the same as `target`, the function should return `true`. Otherwise, if none of the rotations yield the target matrix, the function should return `false`. Note that it is possible to rotate `mat` up to three times to achieve this since a fourth rotation would bring the matrix back to its original orientation.

Intuition

To solve this problem, we need to simulate the rotation of the matrix and compare the result with the target matrix after each rotation. A 90-degree rotation of a matrix can be achieved by reversing the matrix along its horizontal axis (i.e., flipping it upside down) and then taking the transpose. The transpose of a matrix is obtained by switching the rows with the columns, which means element `[i][j]` becomes `[j][i]`.

The intuition behind the solution is to apply this transformation to `mat` up to four times (since rotating four times would return the matrix to its original state), each time checking if the resulting matrix is equal to `target`. In Python, this rotation can be conveniently performed using a combination of list comprehensions and the `zip` function, which groups the elements of the rows (after reversing `mat`) into columns, effectively transposing them.

The steps for a 90-degree rotation are:

1. Flip the matrix upside down (`mat[::-1]`)
2. Transpose the matrix (achieved by `zip(*mat)` after the flip)
3. Convert the zipped elements back into lists (`[list(col) for col in zipped]`)
4. Compare with `target` (`if mat == target`)

The function `findRotation` continuously applies this rotation method up to four times or until `mat` matches `target`. If a match is found before the fourth rotation, it returns `true`. If no match is found after all possible rotations, it returns `false`.

Solution Approach

The solution approach can be explained as follows:

1. **Create a Rotation Function**: The solution defines an inline function within the loop that performs a 90-degree clockwise rotation on `mat`. This is done by flipping the matrix vertically first (`mat[::-1]`) and then transposing it, which in Python can be effortlessly implemented with the `zip` function. The `zip(*mat[::-1])` statement pairs row elements with column indices, effectively rotating the matrix.

2. **Loop Through Rotations**: It initiates a loop that will run four times, each iteration representing a 90-degree rotation. The loop is used because four rotations will bring the matrix back to its initial position. Hence, beyond this, additional rotations would only repeat previous states.

3. **Transform and Compare**: Inside the loop, the matrix `mat` is rotated using the rotation function from step 1. After each rotation, `mat` is compared to `target` (`if mat == target`). If at any point the matrices match, the function immediately returns `true` because we have found a valid rotation that turns `mat` into `target`.

4. **Handling Non-Matching Cases**: If the loop completes without finding a matching rotation (i.e., all four rotations do not result in a matrix equal to `target`), the function returns `false`. This implies that there is no sequence of 90-degree rotations that can transform `mat` into `target`.

The solution elegantly leverages Python's advanced list comprehension and the `zip` function to perform matrix rotations cleanly and concisely. The decision to check for equality only four times is based on the mathematical fact that any square matrix will return to its original orientation after four 90-degree rotations. The process is highly efficient, avoiding unnecessary calculations or rotations.

This solution has a time complexity of O(n^2) for each rotation due to the matrix traversal, where `n` is the number of rows/columns in the matrix. Since a fixed number of rotations (at most 4) are performed, the overall time complexity remains O(n^2). The space complexity is O(n^2) as well, which arises from the storage needed for the rotated matrix at each step.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following `3x3` matrices `mat` and `target`:

`mat`:

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

`target`:

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

First Rotation (0-degree, the original `mat`):

We compare `mat` with `target`. Clearly, they are not identical, so we proceed to rotate `mat`.

Second Rotation (90-degree clockwise rotation):

1. Flip `mat` upside down:

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

2. Transpose the flipped matrix:

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

3. After rotation, we compare the matrix with `target`. Now, `mat` and `target` are identical.

Since `mat` matches `target` after the second rotation, the function would return `true` at this point, indicating that it is possible to make the matrix `mat` identical to `target` by rotating `mat` in 90-degree increments.

If `mat` had not matched `target`, we would have continued to the third and fourth rotations to check all possibilities before determining that `mat` cannot be matched with `target` through rotations.

Python Solution

```python
class Solution:
    def findRotation(self, matrix, target):
        # Try each of the four rotations
        for _ in range(4):
            # Rotate the matrix by 90 degrees clockwise
            matrix = [list(row) for row in zip(*matrix[::-1])]

            # Check if the rotated matrix matches the target matrix
            if matrix == target:
                # If a match is found, return True
                return True
        # If none of the rotations match the target, return False
        return False

```

Java Solution

```java
class Solution {

    // Checks if the matrix "mat" can be rotated to match the matrix "target".
    public boolean findRotation(int[][] mat, int[][] target) {
        // Determine the size of the matrix
        int n = mat.length;

        // Try rotating the matrix 0, 90, 180, and 270 degrees
        for (int k = 0; k < 4; ++k) {
            // Rotate the matrix by 90 degrees
            int[][] rotated = new int[n][n];
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < n; ++j) {
                    // When rotating 90 degrees, the new row is the old column,
                    // and the new column is n - 1 minus the old row.
                    rotated[i][j] = mat[j][n - i - 1];
                }
            }

            // Check if the rotated matrix matches the target matrix
            if (areMatricesEqual(rotated, target)) {
                return true;
            }

            // Update mat to be the rotated matrix for the next comparison
            mat = rotated;
        }

        // If none of the rotations match, return false
        return false;
    }

    // Helper method to check if two matrices are equal
    private boolean areMatricesEqual(int[][] a, int[][] b) {
        int n = a.length;

        // Compare each element of the matrices
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (a[i][j] != b[i][j]) {
                    // If any element does not match, the matrices are not equal
                    return false;
                }
            }
        }

        // All elements match, the matrices are equal
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This function checks if the matrix 'mat' can be rotated to match the 'target' matrix.
    bool findRotation(vector<vector<int>>& mat, vector<vector<int>>& target) {
        int size = mat.size(); // 'size' holds the dimension of the matrix.
        // We will attempt to rotate the matrix up to 4 times (0, 90, 180, 270 degrees).
        for (int rotation = 0; rotation < 4; ++rotation) {
            vector<vector<int>> rotatedMatrix(size, vector<int>(size));
            // Rotate the matrix by 90 degrees clockwise.
            for (int i = 0; i < size; ++i) {
                for (int j = 0; j < size; ++j) {
                    rotatedMatrix[i][j] = mat[j][size - i - 1];
                }
            }
            // After the rotation, check if the rotatedMatrix matches the target.
            if (rotatedMatrix == target) {
                return true; // If match found, return true.
            }
            // Update 'mat' to be the newly rotated matrix for the next iteration.
            mat = rotatedMatrix;
        }
        // If no matching rotation found, return false.
        return false;
    }
};

```

Typescript Solution

```typescript
// Function to check if a matrix can be rotated to match a target matrix.
function findRotation(mat: number[][], target: number[][]): boolean {
    // Try four rotations.
    for (let k = 0; k < 4; k++) {
        rotate(mat); // Rotate the matrix.
        if (isEqual(mat, target)) {
            // If the matrix after rotation equals the target matrix, return true.
            return true;
        }
    }
    // If none of the rotations match the target, return false.
    return false;
}

// Function to check if two matrices are equal.
function isEqual(matrixA: number[][], matrixB: number[][]): boolean {
    const size = matrixA.length;
    for (let i = 0; i < size; i++) {
        for (let j = 0; j < size; j++) {
            // If any corresponding elements differ, return false.
            if (matrixA[i][j] !== matrixB[i][j]) {
                return false;
            }
        }
    }
    // If all elements are equivalent, the matrices are equal.
    return true;
}

// Function to rotate a matrix 90 degrees clockwise.
function rotate(matrix: number[][]): void {
    const size = matrix.length;
    // Only iterate over the first half of rows and first half of columns for a square matrix.
    for (let i = 0; i < size >> 1; i++) {
        for (let j = 0; j < (size + 1) >> 1; j++) {
            // Perform a four-way swap of elements in clockwise direction.
            [
                matrix[i][j],
                matrix[size - 1 - j][i],
                matrix[size - 1 - i][size - 1 - j],
                matrix[j][size - 1 - i],
            ] = [
                matrix[size - 1 - j][i],
                matrix[size - 1 - i][size - 1 - j],
                matrix[j][size - 1 - i],
                matrix[i][j],
            ];
        }
    }
}

```

Time and Space Complexity

The given Python function `findRotation` checks whether one matrix is a rotation of another. It does this by rotating `mat` up to four times (0 degrees, 90 degrees, 180 degrees, and 270 degrees rotations) and comparing it to `target` after each rotation.

The time complexity of the function is determined by these major factors:

1. The number of rotations - which is constant, at `4`.
2. The cost of each rotation - which includes reversing the rows of `mat` (`O(n)`) and then zipping and list conversion (`O(n^2)`).
3. The comparison of `mat` and `target` - which is `O(n^2)` where `n` is the dimension size of the matrix `mat`.

So for each rotation, the total time cost is `O(n) + O(n^2) = O(n^2)`, since `zip(*mat[::-1])` essentially involves looking at all `n^2` elements of the matrix. And since we rotate up to 4 times, the total time complexity is `4 * O(n^2)`, which simplifies to `O(n^2)`.

The space complexity is determined by the extra space needed to store the rotated matrix:

1. The reversed matrix `mat[::-1]` does not use extra space as it is a shallow copy that references the same rows of `mat`.
2. However, `[list(col) for col in zip(*mat[::-1])]` creates a new list of lists for each rotation. This list of lists contains `n` lists of `n` integers, so it uses `O(n^2)` space.

Therefore, the space complexity of the function is `O(n^2)` as it needs space to store a copy of the matrix each time it is rotated.

In summary:

- Time complexity: `O(n^2)`
- Space complexity: `O(n^2)`