1886. Determine Whether Matrix Can Be Obtained By Rotation

Matrix Easy <u>Array</u>

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

Problem Description

1. The goal is to determine whether it is possible to make the matrix mat identical to the matrix target by rotating mat in 90degree 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

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

down) and then taking the transpose. The transpose of a matrix is obtained by switching the rows with the columns, which

(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])

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,

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

Handling Non-Matching Cases: If the loop completes without finding a matching rotation (i.e., all four rotations do not result

The solution approach can be explained as follows:

only repeat previous states.

Solution Approach

4. Compare with target (if mat == target)

effectively rotating the matrix.

- 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.
- 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

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.

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

7 8 9 target:

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

1. Flip mat upside down:

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

Second Rotation (90-degree clockwise rotation):

or rotations.

mat:

1 2 3

4 5 6

7 4 1

9 6 3

7 4 1

8 5 2

9 6 3

Python

Java

class Solution {

class Solution:

Example Walkthrough

1 2 3 2. Transpose the flipped matrix:

Since mat matches target after the second rotation, the function would return true at this point, indicating that it is possible to

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.

def findRotation(self, matrix, target):

Try each of the four rotations

Rotate the matrix by 90 degrees clockwise

public boolean findRotation(int[][] mat, int[][] target) {

// Determine the size of the matrix

for (int i = 0; i < n; ++i) {

return true;

mat = rotated;

return false;

for (int j = 0; j < n; ++j) {

if (areMatricesEqual(rotated, target)) {

// If none of the rotations match, return false

for (int rotation = 0; rotation < 4; ++rotation) {</pre>

for (int j = 0; j < size; ++j) {

// If no matching rotation found, return false.

for (int i = 0; i < size; ++i) {

if (rotatedMatrix == target) {

mat = rotatedMatrix;

return false;

// Try four rotations.

return false;

for (let k = 0; k < 4; k++) {

return true;

const size = matrixA.length;

for (let i = 0; i < size; i++) {

if (isEqual(mat, target)) {

rotate(mat): // Rotate the matrix.

// Function to check if two matrices are equal.

for (let j = 0; j < size; j++) {

// Function to rotate a matrix 90 degrees clockwise.

for (let i = 0; i < (size + 1) >> 1; i++) {

matrix[size - 1 - i][size - 1 - j],

matrix[size - 1 - i][size - 1 - j],

matrix[size - 1 - j][i],

matrix[j][size - 1 - i],

matrix[size - 1 - j][i],

matrix[i][size - 1 - i],

function rotate(matrix: number[][]): void {

for (let i = 0; i < size >> 1; i++) {

matrix[i][i].

matrix[i][j],

const size = matrix.length;

] = [

};

TypeScript

// Rotate the matrix by 90 degrees clockwise.

rotated[i][j] = mat[j][n - i - 1];

// Check if the rotated matrix matches the target matrix

// Update mat to be the rotated matrix for the next comparison

// We will attempt to rotate the matrix up to 4 times (0, 90, 180, 270 degrees).

// After the rotation, check if the rotatedMatrix matches the target.

// Update 'mat' to be the newly rotated matrix for the next iteration.

// If the matrix after rotation equals the target matrix, return true.

vector<vector<int>> rotatedMatrix(size, vector<int>(size));

rotatedMatrix[i][j] = mat[j][size - i - 1];

return true; // If match found, return true.

// Function to check if a matrix can be rotated to match a target matrix.

function findRotation(mat: number[][], target: number[][]): boolean {

// If none of the rotations match the target, return false.

function isEqual(matrixA: number[][], matrixB: number[][]): boolean {

// If any corresponding elements differ, return false.

matrix = [list(row) for row in zip(*matrix[::-1])]

If none of the rotations match the target, return False

// Checks if the matrix "mat" can be rotated to match the matrix "target".

// When rotating 90 degrees, the new row is the old column,

// and the new column is n-1 minus the old row.

Solution Implementation

for in range(4):

return False

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

make the matrix mat identical to target by rotating mat in 90-degree increments.

Check if the rotated matrix matches the target matrix if matrix == target: # If a match is found, return True return True

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];

```
// 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][i] != b[i][i]) {
                    // If any element does not match, the matrices are not equal
                    return false;
        // All elements match, the matrices are equal
        return true;
C++
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.
```

```
if (matrixA[i][j] !== matrixB[i][j]) {
            return false;
// If all elements are equivalent, the matrices are equal.
return true;
```

// Perform a four-way swap of elements in clockwise direction.

// Only iterate over the first half of rows and first half of columns for a square matrix.

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

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.

Time and Space Complexity

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 $4 * 0(n^2)$, which simplifies to $0(n^2)$.

2. The cost of each rotation - which includes reversing the rows of mat(0(n)) and then zipping and list conversion $(0(n^2))$.

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,

3. The comparison of mat and target - which is $O(n^2)$ where n is the dimension size of the matrix mat.

so it uses $O(n^2)$ space. Therefore, the space complexity of the function is $0(n^2)$ as it needs space to store a copy of the matrix each time it is rotated.

• Time complexity: 0(n^2) Space complexity: 0(n^2)

In summary: