Rotate Matrix by 90 Degrees (Rotate Image)

This lecture covers the "Rotate Matrix by 90 Degrees" problem, also known as "Rotate Image." The goal is to rotate a given *n x n* square matrix in a clockwise direction by 90 degrees.

Understanding the Problem

Given a square matrix, the task is to rotate it 90 degrees clockwise. For example, a 4x4 matrix:

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

After rotation, becomes:

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

Brute Force Solution

1. Create an Answer Matrix: Initialize a new *n x n* matrix to store the rotated result.
2. Iterate and Place: Traverse the original matrix and place each element at its correct rotated position in the answer matrix.
    - The first row of the original matrix becomes the last column of the answer matrix.
    - The second row becomes the second-to-last column, and so on.

Mapping Indices

Observe how elements are moved from the original matrix to the answer matrix. If *(i, j)* represents an element in the original matrix, its position in the answer matrix can be determined through observation.

For a 4x4 matrix, the mapping is as follows:

- (0, 0) -> (0, 3)
- (0, 1) -> (1, 3)
- (0, 2) -> (2, 3)
- (0, 3) -> (3, 3)
- (1, 0) -> (0, 2)
- (1, 1) -> (1, 2)
- (1, 2) -> (2, 2)
- (1, 3) -> (3, 2)

From this, a pattern emerges:

- The column index *j* in the original matrix becomes the row index in the answer matrix.
- The row index *i* in the original matrix corresponds to *n - 1 - i* as the column index in the answer matrix.

Therefore, the element at *(i, j)* in the original matrix goes to *(j, n - 1 - i)* in the answer matrix.

Brute Force Code

```
// Assuming a square matrix of size n x n
int n = matrix.size();
vector<vector<int>> ans(n, vector<int>(n, 0));

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        ans[j][n - 1 - i] = matrix[i][j];
    }
}
// 'ans' is the rotated matrix
```

Complexity Analysis

- Time Complexity: $O(n^2)$ because we iterate through each element of the *n x n* matrix.
- Space Complexity: $O(n^2)$ due to the creation of an extra *n x n* matrix.

Optimal (In-Place) Solution

The interviewer is typically not satisfied with the $O(n^2)$ space complexity. An in-place solution is preferred, meaning we should modify the given matrix directly without using extra space.

Key Observations

1. Columns to Rows: Notice that the first column of the rotated matrix is the reverse of the first row of the original matrix.

2. Transposition and Reversal: The optimal solution involves two steps:

    - Transpose the Matrix: Swap rows with columns.
    - Reverse Each Row: Reverse the elements in each row.

Transposing a Matrix

> Transposing a Matrix: Transforming a matrix by interchanging its rows and columns, so that the row *i* becomes column *i* and vice versa.

To transpose the matrix, we swap elements across the main diagonal (from top-left to bottom-right). Elements on the diagonal remain unchanged.

For example:

```
1 2 3 4        1 5 9 13
5 6 7 8   -->  2 6 10 14
9 10 11 12       3 7 11 15
13 14 15 16      4 8 12 16
```

Reversing Each Row

After transposing, reverse the elements of each row to get the final rotated matrix.

For example:

```
1 5 9 13       13 9 5 1
2 6 10 14  -->  14 10 6 2
3 7 11 15       15 11 7 3
4 8 12 16      16 12 8 4
```

Optimal Solution Code

```cpp
int n = matrix.size();

// Transpose the matrix
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        swap(matrix[i][j], matrix[j][i]);
    }
}

// Reverse each row
for (int i = 0; i < n; i++) {
    reverse(matrix[i].begin(), matrix[i].end());
}
```

Complexity Analysis

- Time Complexity: $O(n^2)$ because we iterate through each element of the *n x n* matrix twice (once for transpose and once for reverse).
- Space Complexity: $O(1)$ because the rotation is done in-place without using any extra space.

Summary of Approaches

Approach	Time Complexity	Space Complexity	In-Place?
Brute Force	$O(n^2)$	$O(n^2)$	No
Optimal	$O(n^2)$	$O(1)$	Yes

By transposing the matrix and then reversing each row, we efficiently achieve a 90-degree clockwise rotation in place.

Transposing a Matrix

When transposing a matrix, elements are swapped across the diagonal. The element at position `(i, j)` is swapped with the element at position `(j, i)`.

Swapping Elements

Consider a matrix where elements are being swapped:

- A `2` at `(0, 1)` is swapped with a `5` at `(1, 0)`.
- A `3` at `(0, 2)` is swapped with a `9` at `(2, 0)`.
- A `7` at `(1, 2)` is swapped with a `10` at `(2, 1)`.

Algorithm

1. Iterate through the upper triangle of the matrix (excluding the diagonal).
2. Swap `matrix[i][j]` with `matrix[j][i]`.

Pseudo Code

```
for i from 0 to n-2:
    for j from i+1 to n-1:
        swap(a[i][j], a[j][i])
```

Example

For the zeroth row, you travel from index `1` to `3`. For the first row, you travel from index `2` to `3`. Thus, for `i`, you travel from `i + 1` to `n - 1`.

Rotating the Matrix

To rotate the matrix, you have to transpose the matrix and reverse every row.

Step 1: Transpose the Matrix

As described above, the first step is to transpose the matrix.

Step 2: Reverse Each Row

After transposing, reverse each row in the matrix.

Code Implementation

```
void rotate(vector<vector<int>>& matrix) {
    int n = matrix.size();

    // Transpose the matrix
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }

    // Reverse each row
    for (int i = 0; i < n; ++i) {
        reverse(matrix[i].begin(), matrix[i].end());
    }
}
```

Reversing Rows

If your language doesn't have a reverse function, a two-pointer approach can be used to reverse the array.

Time and Space Complexity

Time Complexity

The time complexity for transposing the matrix is $O(n/2 * n/2)$ because we are traversing one half of the array. Reversing each row takes $O(n * n/2)$, where `n` is for all rows and `n/2` is for reversing half of the array using a two-pointer approach.

Space Complexity

This is done in place, so no extra space is used, making the space complexity $O(1)$.