48. Rotate Matrix

<https://leetcode.com/problems/rotate-image/>

1. **Listen**

**Problem Statement:**

You are given an **n x n** **2D matrix** representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm), which means you have to modify the input 2D matrix directly. **DO NOT**allocate another 2D matrix and do the rotation.

**Input:**

n x n 2D matrix

**Goal:**

rotate the image by **90** degrees clockwise in-place (modify input 2D matrix directly).

**Return:**

return the input 2D matrix rotated by 90 degrees

1. **Examples**

**Example 1:**

A screenshot of a graph

Description automatically generated with low confidence

**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

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

Although the numbers are in sorted order in this example, there is no constraint saying that the numbers are sorted. They can be in random order.

**Example 2:**

A picture containing text, shoji, crossword puzzle, clipart

Description automatically generated

**Input:** matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]

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

**Constraints:**

* Rotate the input 2D matrix in-place
* **DO NOT** allocate another 2D matrix and do the rotation (use O(1) space)
* n == matrix.length == matrix[i].length
* 1 <= n <= 20
* -1000 <= matrix[i][j] <= 1000

**Test Cases:**

* 1x1 matrix
* 3x3 matrix

1. **Brute Force**

**Approach 1:** **Swap Rows and Columns Using Extra Space**

* Let’s take a look at a visualization of Example 1.
* With matrix problems, you will greatly benefit from drawing it out as opposed to thinking in only code.
* Can we see any **patterns** in the drawing?

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Matrix**   |  |  |  | | --- | --- | --- | | 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 | 9 | |  | **Output Matrix**   |  |  |  | | --- | --- | --- | | 7 | 4 | 1 | | 8 | 5 | 2 | | 9 | 6 | 3 | |

At a first glance, an obvious pattern is immediately visible.

* The 1st row of the input matrix becomes the last column in the output matrix
* The 2nd row of the input matrix becomes the 2nd to last column in the output matrix
* The 3rd row of the input matrix becomes the 3rd to last column in the output matrix

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Matrix**   |  |  |  | | --- | --- | --- | | 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 | 9 | |  | **Output Matrix**   |  |  |  | | --- | --- | --- | | 7 | 4 | 1 | | 8 | 5 | 2 | | 9 | 6 | 3 | |

* Going off this pattern, we could approach to solve this using extra space.
* By allocating a new empty 2D matrix that is the same size and structure as the original input matrix, we can directly copy elements over according to the required rotated order of the output matrix.

**Time Complexity:**

This algorithm would use two nested loops to traverse the n x n matrix O(n²) times.

**Space Complexity:**

The use of an extra matrix takes O(n²) space.

**Problem:**

This solution does not meet the problem constraints:

* it uses extra space
* it does not solve the problem in-place

1. **Optimize**

**Approach 2: Transpose and Reverse**

**Look for Another Pattern**

* Can we observe any other patterns in the rotation of the matrix?
* There at least, seem to be multiple ways to solve the problem.
* Let's think!
* Since the problem requires us to edit the input matrix in-place, there must be some way to reposition rows/columns in a way that allows us to accomplish this.

Let’s look back at and build off of the previous solution.

* The 1st row becomes the last column
* The 2nd row becomes the 2nd to last column
* The 3rd row becomes the 3rd to last column

cols

rows

3rd

2nd

1st

3rd

2nd

1st

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Matrix**   |  |  |  | | --- | --- | --- | | 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 | 9 | |  | **Output Matrix**   |  |  |  | | --- | --- | --- | | 7 | 4 | 1 | | 8 | 5 | 2 | | 9 | 6 | 3 | |

**Transpose**

This actually closely resembles the **transpose** of a matrix

We can **transpose** a matrix by **swapping its rows with its columns**

* The 1st row becomes the 1st column
* The 2nd row becomes the 2nd column
* The 3rd row becomes the 3rd column

cols

rows

3rd

2nd

1st

3rd

2nd

1st

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Matrix**   |  |  |  | | --- | --- | --- | | 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 | 9 | |  | **Output Matrix**   |  |  |  | | --- | --- | --- | | 1 | 4 | 7 | | 2 | 5 | 8 | | 3 | 6 | 9 | |

**Question:** What’s the difference between the **transpose** of the matrix and a matrix **rotated clockwise by 90 degrees**?

**Answer: A matrix** **rotated clockwise 90 degrees**

**is equivalent to**

**the transpose** **of a** **matrix reversed**

**Algorithm**

Therefore, we can rotate a matrix clockwise by 90 degrees in two steps:

1. Transpose the matrix
2. Reverse the matrix (horizontally)

Step 1: Transpose

cols

rows

3rd

2nd

1st

3rd

2nd

1st

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Matrix**   |  |  |  | | --- | --- | --- | | 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 | 9 | |  | **Transposed**   |  |  |  | | --- | --- | --- | | 1 | 4 | 7 | | 2 | 5 | 8 | | 3 | 6 | 9 | |

Step 2: Reverse (horizontally)

cols

rows

3rd

2nd

1st

3rd

2nd

1st

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Transposed**   |  |  |  | | --- | --- | --- | | 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 | 9 | |  | **Reversed**   |  |  |  | | --- | --- | --- | | 7 | 4 | 1 | | 8 | 5 | 2 | | 9 | 6 | 3 | |

**Pseudocode**

// transpose

int n = matrix.length;

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

for(int **j = i**; j < n; j++)

swap(matrix[i][j], matrix[j][i])

When transposing, each time we enter the nested loop, j must start from i.

This is because if we started j from 0, the loop would eventually end up undoing all of its changes and we would be left with the original matrix.

This is best shown with a larger example

Yellow = swapping

Green = swapped

**Original**

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Transpose 1**   |  |  |  |  | | --- | --- | --- | --- | | 5 | 1 | 9 | 11 | | 2 | 4 | 8 | 10 | | 13 | 3 | 6 | 7 | | 15 | 14 | 12 | 16 |   1. swap(m[0,0], m[0,0])  2. swap(m[0,1], m[1,0])  3. swap(m[0,2], m[2,0])  4. swap(m[0,3], m[3,0]) | → | **Transpose 2**   |  |  |  |  | | --- | --- | --- | --- | | 5 | 2 | 13 | 15 | | 1 | 4 | 8 | 10 | | 9 | 3 | 6 | 7 | | 11 | 14 | 12 | 16 |   1. swap(m[1,1], m[1,1])  2. swap(m[1,2], m[2,1])  3. swap(m[1,3], m[3,1]) | → | **Transpose 3**   |  |  |  |  | | --- | --- | --- | --- | | 5 | 2 | 13 | 15 | | 1 | 4 | 3 | 14 | | 9 | 8 | 6 | 7 | | 11 | 10 | 12 | 16 |   1. swap(m[2,2], m[2,2])  2. swap(m[2,3], m[3,2]) | → | **Transpose 4**   |  |  |  |  | | --- | --- | --- | --- | | 5 | 2 | 13 | 15 | | 1 | 4 | 3 | 14 | | 9 | 8 | 6 | 7 | | 11 | 10 | 12 | 16 |   1. swap(m[2,2], m[2,2]) |

**Output**

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

// reverse

int n = matrix.length;

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

for(int j = 0; **j <** **n / 2**; j++)

swap(matrix[j][i], matrix[j][n-i-1])

The matrix has n x n dimensions.

Therefore, we can simply reverse n arrays n times.

Reversing a single array is relatively simple. We traverse over half of the array, swapping the first and last value.

* swap first and last
* swap first+1 and last-1
* swap first+2 and last-2
* … n/2 times

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Transpose Output**   |  |  |  |  | | --- | --- | --- | --- | | 5 | 2 | 13 | 15 | | 1 | 4 | 3 | 14 | | 9 | 8 | 6 | 7 | | 11 | 10 | 12 | 16 | | → | **Reverse 1**   |  |  |  |  | | --- | --- | --- | --- | | 15 | 2 | 13 | 5 | | 14 | 4 | 3 | 1 | | 7 | 8 | 6 | 9 | | 16 | 10 | 12 | 11 | | → | **Reverse Output**   |  |  |  |  | | --- | --- | --- | --- | | 15 | 13 | 2 | 5 | | 10 | 3 | 4 | 2 | | 7 | 6 | 8 | 13 | | 16 | 12 | 10 | 15 | |

**Runtime**

Time Complexity: O(n2) + O(n2) = O(n2)

Space Complexity: O(1)

1. **Walkthrough**

**Approach 3: Rotate Groups of Four Cells**

* Is there yet another way to rotate the matrix?
* Let's think!

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input Matrix**   |  |  |  | | --- | --- | --- | | 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 | 9 | |  | **Output Matrix**   |  |  |  | | --- | --- | --- | | 7 | 4 | 1 | | 8 | 5 | 2 | | 9 | 6 | 3 | |

We can see a pattern of how cells move in groups when we rotate the matrix.

Looking at the corners, they all rotate together.

We can perform a swap on them to achieve this effect.

This same effect can be seen with the middle indexes.

For larger matrices, we would find that there would be a total of n/2 of these squares.

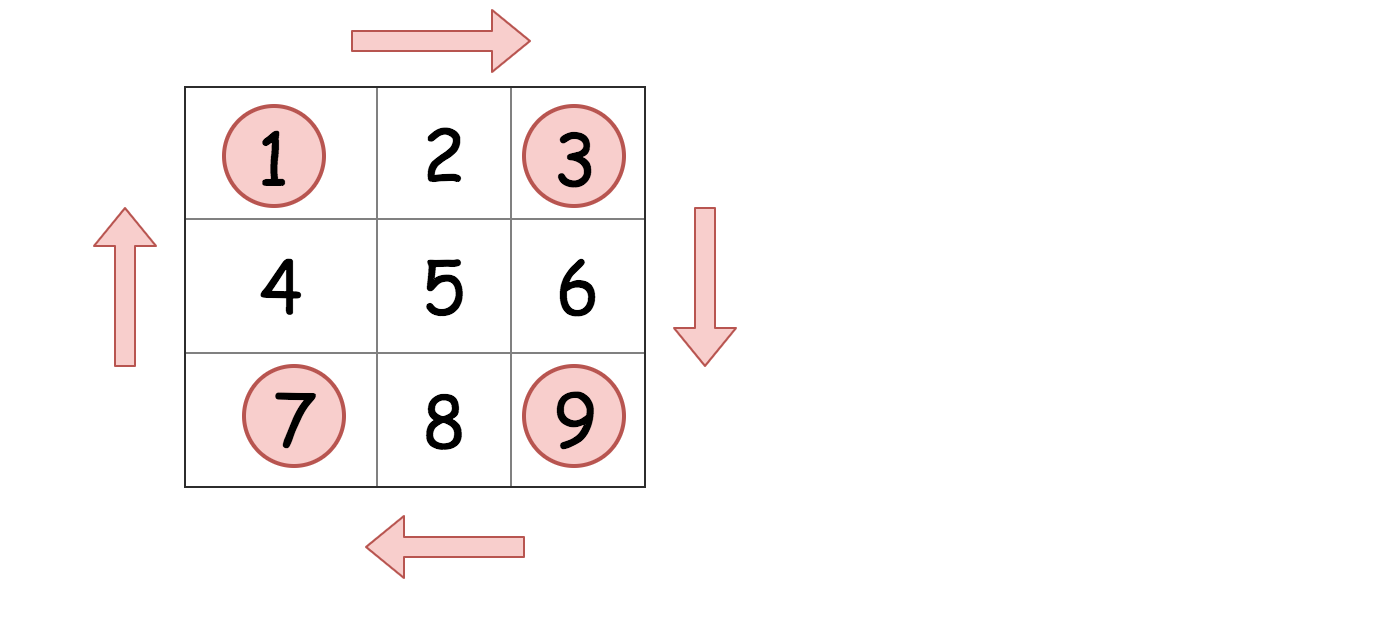
We first process the first square formed by the

top left

bottom left

top right

bottom right



Then we process the second square formed by the

top middle

left middle

right middle

bottom middle

In each square, elements are moved in a cycle of 4 elements.

1. **Implement**

int n = matrix.length;

for(int i = 0; i < n / 2; i++)

{

for(int j = i; j < n - i - 1; j++)

{

/\*

int topLeft = matrix[i][j];

int bottomLeft = matrix[n - j - 1][i];

int bottomRight = matrix[n - i - 1][n - j - 1];

int topRight = matrix[j][n - i - 1];

\*/

// save the topleft

int topLeft = matrix[i][j];

// top left becomes bottom left (topLeft = bottomLeft)

matrix[i][j] = matrix[n - j - 1][i];

// bottom left becomes bottom right (bottomLeft = bottomRight)

matrix[n - j - 1][i] = matrix[n - i - 1][n - j - 1];

// bottom right becomes top right (topRight = bottomRight)

matrix[n - i - 1][n - j - 1] = matrix[j][n - i - 1];

// top right becomes top left (topRight = topLeft)

matrix[j][n - i - 1] = topLeft;

}

}

1. **Test**