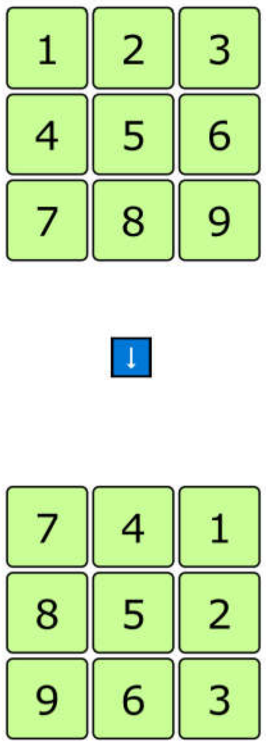
Rotate Matrix

We'll learn how to work with a matrix in JavaScript and how to efficiently manipulate it. We'll cover a couple of clever tricks that can make our lives a lot easier by allowing us to reuse code.

We'll cover the following

Rotate Matrix#



Instructions#

A matrix in JavaScript can be represented as an array of arrays. For example, here is a 3 x 3 matrix:

```
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

Write a function that takes a matrix as input and returns a new matrix. This new matrix should represent the original matrix rotated 90 degrees clockwise. The original matrix should be unchanged. This function should work for both square and rectangular matrixes.

Input: Array of Arrays of Numbers

Output: Array of Arrays of Numbers

Example#

When given the matrix above as input, the output should be:

```
[[7, 4, 1],
 [8, 5, 2],
 [9, 6, 3]]
```

Hints#

- We'll need to start by creating a new, empty matrix.
- The new matrix dimensions will need to be switched, e.g. 2 x 3 becomes 3 x 2.
- We'll need to process every item.

```
function rotateClockwise(matrix) {
    // Your code here
    const newMatrix = matrix[0].map(() => []);

}
```

Solution#

```
function rotateClockwise(matrix) {
    const newMatrix = matrix[0].map(() => []);

    for (let i = 0; i < matrix.length; i++) {
        for (let j = 0; j < matrix[0].length; j++) {
            newMatrix[j][matrix.length - 1 - i] = matrix[i][j];
        }
    }

    return newMatrix;
}
```

Show Results Show Console

0.92s

5 of 5 Public Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	[[1]]	[[1]]	[[1]]	Succeeded
✓	[[1, 2], [3, 4]]	[[3, 1], [4, 2]]	[[3,1],[4,2]]	Succeeded
✓	[[1, 2, 3], [4, 5, 6], [7, 8, 9]]	[[7, 4, 1], [8, 5, 2], [9, 6, 3]]	[[7,4,1],[8,5,2],[9,6,3]]	Succeeded
✓	[[1, 2, 3]]	[[1], [2], [3]]	[[1],[2],[3]]	Succeeded
✓	[[1, 2, 3], [4, 5, 6]]	[[4, 1], [5, 2], [6, 3]]	[[4,1],[5,2],[6,3]]	Succeeded

How it Works#

Line 2 above creates a new matrix. Notice that we're mapping over `matrix[0]` to create our new matrix. This ensures that *the number of rows in our new matrix will equal the number of columns in the old matrix.*

When we're rotating a matrix, a 4 x 2 turns into a 2 x 4. So the number of rows and the number of columns swaps. That's what line 2 is doing.

The double for-loop goes through each item in our matrix and inserts it into the correct position. Try taking a small matrix and testing out the algorithm by hand. You'll see how it works and how it places items in the correct place.

Time#

The double for-loop here does not signify an `O(n^2)` time complexity. The outer for-loop goes through the rows one-by-one and the inner loop goes through each column in that row. Each item ends up being processed only once, so we have a time complexity of:

O(n).#

Space#

Space complexity is:

O(n).#

Our new matrix takes up the same space as our last one.

Other Methods#

Instead of a double for-loop, we could have used a while statement. In the while statement, we would still need to use two index variables, `i` and `j`, for the row and column numbers. The while loop would end once the row and column numbers together reached the end of the matrix. This would have the same time and space complexities.

I think the double for-loop is a little easier to work with.

Bonus#

Instructions#

1. Write another function that performs the same task as above, but rotates an input matrix counter-clockwise instead.
2. Write another function that rotates an input matrix 180 degrees.

Hints#

- We can use use the function that we've already written above
- A 180 degree rotation is a 90 degree rotation applied twice

```javascript
function rotateClockwise(matrix) {
    const newMatrix = matrix[0].map(() => []);

    for (let i = 0; i < matrix.length; i++) {
        for (let j = 0; j < matrix[0].length; j++) {
            newMatrix[j][matrix.length - 1 - i] = matrix[i][j];
        }
    }

    return newMatrix;
}

function rotate180(matrix) {
    return rotateClockwise(rotateClockwise(matrix));
}

function rotateCounterClockwise(matrix) {
    return rotateClockwise(rotate180(matrix));
}
```

We could have written more complex functions and dealt with more nasty double for-loops. This short-cut allows us to avoid all of that.

Time and Space#

The time and space complexities of these new functions are the same as the original function. The original was `O(n)` for both time and space. These new functions simply run that function either twice or three times, multiplying that complexity by either 2 or 3.

This means the new functions are `O(2n)` and `O(3n)`. Since we drop constants, they are both reduced to

O(n).#

← Back

All Anagrams

Next →

Rotate Matrix, Advanced

✓ Mark as Completed

Report an Issue