3033. Modify the Matrix

Easy Array Matrix

Problem Description

Given a two-dimensional grid, 'matrix', with m rows and n columns, we are asked to create an identical grid, 'answer'. The task involves identifying elements in 'matrix' with a value of -1 and updating them to the maximum value found in their respective columns. It's important to keep in mind that our grids are 0-indexed, meaning that row and column indices start from 0. The goal is to process 'matrix' and transform it into 'answer' following this rule, ultimately returning 'answer'.

Intuition

The solution revolves around handling the grid column by column. For each column, we look for the highest number (disregarding any -1 values, since we're only interested in positive numbers for replacement). After finding this maximum value, we proceed down the same column, substituting any instance of -1 with the column's maximum value we previously found.

The process can be split into two main steps:

1. Traverse each column of the matrix to determine the maximum non-negative value present in that column.
2. Go through the matrix again, this time replacing every -1 with the maximum value found in step 1 for the respective column.

Solution Approach

The algorithm implemented in the provided solution is straightforward and efficient. It employs a nested loop structure to process the matrix column by column.

Here's the step-by-step breakdown of the algorithm:

1. Determine the size of the matrix using the `*len*` function to find `m` (number of rows) and `n` (number of columns).
2. Use an outer loop to iterate through each column index from `0` to `n - 1`.
3. Inside the outer loop, perform a list comprehension to determine the maximum value `mx` in the current column. This is done by iterating over each row `i` for the current column `j` and collecting each element's value, except for `-1`. The `max` function then gives the highest value in that column.
   - This is achieved with the expression: `max(matrix[i][j] for i in range(m))`
4. After finding the maximum value for the column, we use another inner loop to go through each row for the same column.
5. Here we check if any element is `-1` and replace it with the maximum value `mx` found earlier.
   - The replacement is done only when the condition `if matrix[i][j] == -1:` is true.
6. Once all the columns have been processed, the original `matrix` is now modified to the `answer` matrix with all `-1`s replaced by their respective column maximums.
7. Lastly, return the modified `matrix` as the result.

No additional data structures or complex patterns are used in this implementation; it utilizes only basic loops and list comprehension techniques. The space complexity is constant, as we're simply updating the original matrix in place, without using any extra space proportional to the size of the input (beyond the fixed number of variables needed for iteration).

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have the following grid `matrix`, consisting of 3 rows and 4 columns:

```
matrix = [
  [0, -1, 3, 4],
  [3,  2,  1, -1],
  [-1, 2, -1, 0]
]
```

Following the steps of the solution approach:

1. Determine the size of the matrix. We can see that `m` is 3 and `n` is 4.
2. Start with the first column (`j=0`). We iterate through each row and find that the maximum value for this column is 3 (ignoring the -1).
3. Replace all `-1` values in this column with the maximum value found. There is one `-1` value in the third row, so we get:

```
matrix = [
  [0, -1, 3, 4],
  [3,  2,  1, -1],
  [3,  2, -1, 0]
]
```

4. Move to the second column (`j=1`). The maximum value for this column is 2.
5. Replace `-1` values in this column. There is one `-1` in the first row, resulting in:

```
matrix = [
  [0, 2, 3, 4],
  [3, 2, 1, -1],
  [3, 2, -1, 0]
]
```

6. Continue to the third column (`j=2`). The maximum value, ignoring -1, is 3.
7. Replace `-1` values in the third column. There is one `-1` in the third row, leading to:

```
matrix = [
  [0, 2, 3, 4],
  [3, 2, 1, -1],
  [3, 2, 3, 0]
]
```

8. Finally, proceed to the last column (`j=3`). The maximum value is 4.
9. There's one `-1` in the second row that needs to be replaced:

```
matrix = [
  [0, 2, 3, 4],
  [3, 2, 1, 4],
  [3, 2, 3, 0]
]
```

After completing these steps, `matrix` is now transformed into `answer` by replacing all `-1` values with the maximum values in their respective columns. The resulting matrix is:

```
answer = [
  [0, 2, 3, 4],
  [3, 2, 1, 4],
  [3, 2, 3, 0]
]
```

This matrix is then returned as the final output of the algorithm. The approach ensures that each -1 in the original matrix is replaced by the maximum positive value present in its respective column.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def modifiedMatrix(self, matrix: List[List[int]]) -> List[List[int]]:
        # Get the number of rows (m) and columns (n) in the matrix
        num_rows, num_columns = len(matrix), len(matrix[0])

        # Iterate over each column
        for col in range(num_columns):
            # Find the maximum value in the current column
            max_value = max(matrix[row][col] for row in range(num_rows))

            # Replace all occurrences of -1 with the maximum value in the current column
            for row in range(num_rows):
                if matrix[row][col] == -1:
                    matrix[row][col] = max_value

        # Return the modified matrix
        return matrix
```

Java

```java
class Solution {

    /**
     * Modifies a given matrix such that every -1 entry in a column is replaced with
     * the maximum value in that column.
     *
     * @param matrix The original matrix that will be modified.
     * @return The modified matrix.
     */
    public int[][] modifiedMatrix(int[][] matrix) {
        // Get the number of rows m and the number of columns n in the matrix
        int rowCount = matrix.length;
        int columnCount = matrix[0].length;

        // Iterate over columns
        for (int column = 0; column < columnCount; ++column) {
            // Initialize the maximum value for the current column, starting with the smallest possible integer value
            int maxInColumn = Integer.MIN_VALUE;
            // Find the maximum value in the current column
            for (int row = 0; row < rowCount; ++row) {
                maxInColumn = Math.max(maxInColumn, matrix[row][column]);
            }
            // Replace all -1 entries in the current column with the maximum value found
            for (int row = 0; row < rowCount; ++row) {
                if (matrix[row][column] == -1) {
                    matrix[row][column] = maxInColumn;
                }
            }
        }
        // Return the modified matrix
        return matrix;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // The function modifiedMatrix takes a 2D vector of integers, representing a matrix,
    // and modifies it according to certain rules.
    std::vector<std::vector<int>> modifiedMatrix(std::vector<std::vector<int>>& matrix) {
        // Get the number of rows and columns in the matrix.
        int rowCount = matrix.size();
        int colCount = matrix[0].size();

        // Iterate over columns
        for (int col = 0; col < colCount; ++col) {
            // Initialize 'maxValue' to the smallest possible integer to find the maximum later.
            int maxValue = std::numeric_limits<int>::min();

            // Find the maximum value in the current column.
            for (int row = 0; row < rowCount; ++row) {
                maxValue = std::max(maxValue, matrix[row][col]);
            }

            // Replace all -1s in the current column with the maximum value found.
            for (int row = 0; row < rowCount; ++row) {
                if (matrix[row][col] == -1) {
                    matrix[row][col] = maxValue;
                }
            }
        }

        // Return the modified matrix.
        return matrix;
    }
};
```

TypeScript

```typescript
function modifiedMatrix(matrix: number[][]): number[][] {
    // Get the number of rows (m) and columns (n) from the matrix.
    const numberOfRows = matrix.length;
    const numberOfColumns = matrix[0].length;

    // Iterate over each column.
    for (let columnIndex = 0; columnIndex < numberOfColumns; ++columnIndex) {
        // Initialize maximum value in the column as the smallest possible number.
        let maximumInColumn = -1;

        // Find the maximum value in the current column.
        for (let rowIndex = 0; rowIndex < numberOfRows; ++rowIndex) {
            maximumInColumn = Math.max(maximumInColumn, matrix[rowIndex][columnIndex]);
        }

        // Replace all -1s in the current column with the maximum value found.
        for (let rowIndex = 0; rowIndex < numberOfRows; ++rowIndex) {
            if (matrix[rowIndex][columnIndex] === -1) {
                matrix[rowIndex][columnIndex] = maximumInColumn;
            }
        }
    }

    // Return the modified matrix.
    return matrix;
}
```

```python
from typing import List

class Solution:
    def modifiedMatrix(self, matrix: List[List[int]]) -> List[List[int]]:
        # Get the number of rows (m) and columns (n) in the matrix
        num_rows, num_columns = len(matrix), len(matrix[0])

        # Iterate over each column
        for col in range(num_columns):
            # Find the maximum value in the current column
            max_value = max(matrix[row][col] for row in range(num_rows))

            # Replace all occurrences of -1 with the maximum value in the current column
            for row in range(num_rows):
                if matrix[row][col] == -1:
                    matrix[row][col] = max_value

        # Return the modified matrix
        return matrix
```

Time and Space Complexity

The time complexity of the given code is `O(m * n)`, where `m` is the number of rows and `n` is the number of columns in the matrix. This is because the code contains two nested loops, the outer loop iterating over the columns, and the inner loop iterating over the rows. For each column, the maximum value is found (taking `O(m)` time), and then each row in that column is updated if necessary (another `O(m)` time). Therefore, for each of the `n` columns, the algorithm performs work proportional to the number of rows `m`, leading to a total time complexity of `O(m * n)`.

The space complexity of the algorithm is `O(1)`. This reflects that the amount of extra space needed by the algorithm does not depend on the size of the input matrix. The only additional memory used is for the loop variables and the temporary variable `mx`, and this does not scale with the size of the input matrix, hence the constant space complexity.