1329. Sort the Matrix Diagonally

Medium Array Matrix Sorting Leetcode Link

Problem Description

The task is to sort the integers in each diagonal of a given `m x n` matrix `mat`. A matrix diagonal is defined as a line of cells that starts from a cell in either the topmost row or leftmost column and goes diagonally in the bottom-right direction until it reaches the end of the matrix. The sorting of each diagonal should be done in ascending order, and the goal is to return the matrix with all diagonals sorted.

For instance, if a diagonal starts from `mat[2][0]` in a `6 x 3` matrix, it would include the cells `mat[2][0]`, `mat[3][1]`, and `mat[4][2]`. This diagonal, like all others in the matrix, needs to be sorted so that the smallest number is at `mat[2][0]` and the largest at `mat[4][2]`.

Intuition

To solve the problem, we understand that each diagonal can be sorted independently of the others, since the sorting of one diagonal does not affect the elements in another diagonal.

A brute force approach to sort each diagonal is to use a "bubble sort"-like method where for each diagonal, we perform a series of comparisons and swaps until each element on the diagonal is in ascending order. The solution code provided is based on this approach.

Here's how we approach it:

1. Iterate over each possible starting cell of a diagonal. This would usually mean iterating over cells on the top row and the leftmost column, but the code provided is only iterating through `min(m, n)` elements. This part of the approach seems to be limiting the number of iterations and may not cover all diagonals in a non-square matrix (where `m` is not equal to `n`).

2. Perform a double loop, iterating through cells within the matrix boundaries, excluding the last row and column, to compare the current cell `mat[i][j]` with the next cell in the diagonal `mat[i+1][j+1]`.

3. If the current cell is larger than the next cell in the diagonal, we swap them. The swap ensures that the smaller value moves towards the start of the diagonal.

4. Repeat the comparison and swapping process until all cells in the matrix are checked. Note that this requires multiple passes through the matrix to get all elements in the correct order, given the nature of the "bubble sort"-like method.

This solution ensures each diagonal is sorted, but it's not the most efficient approach due to its high time complexity, and it's not optimized to handle non-square matrices correctly in all cases.

Solution Approach

The code provided uses a simplified in-place sorting algorithm similar to bubble sort to sort each diagonal of the matrix.

Let's walk through the implementation step by step:

1. The first step is to retrieve the dimensions of the matrix using `len(mat)` for the number of rows (`m`) and `len(mat[0])` for the number of columns (`n`).

2. It then uses a nested loop to iterate over the elements of the matrix, excluding the last row and column to prevent out-of-bounds access. The loop variables `i` and `j` represent the current cell being considered.

3. The code compares each element `mat[i][j]` with the next diagonal element `mat[i + 1][j + 1]`. When it finds that `mat[i][j]` is greater than `mat[i + 1][j + 1]`, it performs a swap. This operation is intended to move the larger numbers down and right along the diagonals and the smaller numbers up and left.

4. However, the provided solution approach only iterates through the diagonals that start in the cells `[0, k]` for `k < min(m, n)`. This is a simplification and does not sort all diagonals properly in the case of non-square matrices, since it doesn't address all possible starting cells of the diagonals that could be on the leftmost column for a matrix where `m > n`.

5. The code will perform the comparison and swapping process repeatedly in a brute force manner, iterating over the matrix cells `m - 1` times to ensure all cells are eventually sorted. Since bubble sort has a time complexity of O(N^2), the time complexity of this code will also reflect the inefficiencies of bubble sort, especially as the size of the matrix grows.

To sum up, the solution attempts to use a bubble sort-like algorithm over the matrix's diagonals to sort each one in ascending order. The problem with this implementation is that it may not sort all diagonals as desired, especially for non-square matrices, and it's not efficient due to the bubble sort approach resulting in a high time complexity.

Note that a more efficient solution would sort each diagonal individually by extracting the diagonal elements, sorting them using a more efficient sorting algorithm (e.g., quicksort or timsort), and then placing them back into the matrix. This approach would significantly reduce the overall time complexity to O(DlogD) for each diagonal of average length D.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach described. Consider the following `3x3` matrix `mat`:

```
mat = [
  [3, 3, 1],
  [2, 2, 2],
  [1, 1, 1]
]
```

The goal is to sort the integers in each diagonal. In this matrix, there are five diagonals:

1. The first diagonal starting from `mat[0][0]` includes the elements `[3]` (already sorted).
2. The second diagonal starting from `mat[0][1]` includes the elements `[3, 2]`.
3. The third diagonal starting from `mat[0][2]` includes the elements `[1, 2, 1]`.
4. The fourth diagonal starting from `mat[1][0]` includes the elements `[2, 1]`.
5. The fifth diagonal starting from `mat[2][0]` includes the elements `[1, 2, 3]` (already sorted).

The sorting algorithm provided will iterate through these diagonals and sort them. Let's focus on how this algorithm would sort the third diagonal `[1, 2, 1]` as an example, as it's the longest and requires actual sorting:

1. We retrieve the dimensions of the matrix, which are `m = 3` for rows and `n = 3` for columns.

2. We iterate starting from `mat[0][2]`, excluding the last row and column to make sure we won't access `mat[3][3]` which is out of bounds.

3. Now, we compare each element on this diagonal with the subsequent diagonal element:

   - Compare `mat[0][2]` (which is `1`) with `mat[1][3]` (which doesn't exist, so we skip this step).
   - Compare `mat[1][2]` (which is `2`) with `mat[2][3]` (which doesn't exist, so we skip this step).

   Following the above approach, it seems we don't make any swaps. However, since we have to iterate `m - 1` times, we will re-iterate and compare the elements within the matrix boundaries again.

4. Since we are not going out of bounds, we re-iterate over the matrix:

   - Compare `mat[0][2]` (which is `1`) with `mat[1][2]` (which is `2`). No swap needed since `1` is less than `2`.
   - Compare `mat[1][2]` (which is `2`) with `mat[2][2]` (which is `1`). Swap needed since `2` is greater than `1`.
5. After the necessary swap, our diagonal now looks like `[1, 1, 2]`, and it is sorted in ascending order.

If we repeat this process for all diagonals, the final sorted matrix will be:

```
mat = [
  [3, 2, 1],
  [2, 1, 2],
  [1, 1, 2]
]
```

Note that each diagonal is individually sorted, even if the matrix as a whole isn't.

However, the described algorithm is not efficient as it might require many iterations for larger matrices due to its similarity to bubble sort, which has a high time complexity. Also, this example is for a square matrix – the algorithm may not perform correctly for non-square matrices because it does not iterate through all possible starting cells of the diagonals. A more efficient approach would be to extract each diagonal, sort it, and then place it back into the matrix.

Python Solution

```python
class Solution:
    def diagonalSort(self, mat: List[List[int]]) -> List[List[int]]:
        # Get the number of rows and columns in the matrix.
        num_rows, num_cols = len(mat), len(mat[0])

        # Only need to iterate up to the smallest dimension minus one
        # since we start the comparison from the second-to-last diagonal.
        for _ in range(min(num_rows, num_cols) - 1):
            # Iterate through each element in the matrix except for the last row and column.
            for row in range(num_rows - 1):
                for col in range(num_cols - 1):
                    # Compare the current element with the element in the next diagonal position.
                    if mat[row][col] > mat[row + 1][col + 1]:
                        # Swap the elements if they are out of order.
                        mat[row][col], mat[row + 1][col + 1] = mat[row + 1][col + 1], mat[row][col]

        # Return the sorted matrix.
        return mat

```

Java Solution

```java
class Solution {

    /**
     * Sorts each diagonal of the given matrix independently, where a diagonal
     * is defined from the top-left to the bottom-right corners.
     *
     * @param matrix The matrix to be sorted diagonally.
     * @return The diagonally sorted matrix.
     */
    public int[][] diagonalSort(int[][] matrix) {
        // Determine the number of rows and columns in the matrix
        int numRows = matrix.length;
        int numCols = matrix[0].length;

        // Iterate over each element of the matrix except the last row and column
        for (int k = 0; k < Math.min(numRows, numCols) - 1; ++k) {
            for (int i = 0; i < numRows - 1; ++i) {
                for (int j = 0; j < numCols - 1; ++j) {

                    // Compare and swap elements if current element is greater
                    // than the element in the next row and next column
                    if (matrix[i][j] > matrix[i + 1][j + 1]) {
                        // Swap elements using a temporary variable
                        int temp = matrix[i][j];
                        matrix[i][j] = matrix[i + 1][j + 1];
                        matrix[i + 1][j + 1] = temp;
                    }
                }
            }
        }

        // Return the sorted matrix
        return matrix;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For std::sort function

class Solution {
public:
    std::vector<std::vector<int>> diagonalSort(std::vector<std::vector<int>>& matrix) {
        int rowCount = matrix.size();    // Number of rows in the matrix
        int colCount = matrix[0].size(); // Number of columns in the matrix

        // Sort each diagonal that starts from the first column
        for (int row = 0; row < rowCount; ++row) {
            sortDiagonal(matrix, row, 0, rowCount, colCount);
        }

        // Sort each diagonal that starts from the first row, except the first diagonal which is already sorted
        for (int col = 1; col < colCount; ++col) {
            sortDiagonal(matrix, 0, col, rowCount, colCount);
        }

        return matrix;
    }

private:
    // Helper function to sort a single diagonal starting at (startRow, startCol)
    void sortDiagonal(std::vector<std::vector<int>>& matrix, int startRow, int startCol, int rowCount, int colCount) {
        int row = startRow;
        int col = startCol;
        std::vector<int> diagonalElements;

        // Collect all elements of the diagonal
        while (row < rowCount && col < colCount) {
            diagonalElements.push_back(matrix[row][col]);
            ++row;
            ++col;
        }

        // Sort the collected diagonal elements
        std::sort(diagonalElements.begin(), diagonalElements.end());

        // Put the sorted elements back into their places on the diagonal
        row = startRow;
        col = startCol;
        for (int element : diagonalElements) {
            matrix[row][col] = element;
            ++row;
            ++col;
        }
    }
};

```

Typescript Solution

```typescript
type Matrix = number[][];

// This function sorts the matrix diagonally
function diagonalSort(matrix: Matrix): Matrix {
    const rowCount = matrix.length;    // Number of rows in the matrix
    const colCount = matrix[0].length; // Number of columns in the matrix

    // Sort each diagonal that starts from the first column
    for (let row = 0; row < rowCount; ++row) {
        sortDiagonal(matrix, row, 0, rowCount, colCount);
    }

    // Sort each diagonal that starts from the first row,
    // except for the first diagonal which is already sorted
    for (let col = 1; col < colCount; ++col) {
        sortDiagonal(matrix, 0, col, rowCount, colCount);
    }

    return matrix;
}

// Helper function that sorts a single diagonal
// starting at (startRow, startCol)
function sortDiagonal(matrix: Matrix, startRow: number, startCol: number, rowCount: number, colCount: number): void {
    let row = startRow;
    let col = startCol;
    const diagonalElements: number[] = [];

    // Collect all elements of the diagonal
    while (row < rowCount && col < colCount) {
        diagonalElements.push(matrix[row][col]);
        ++row;
        ++col;
    }

    // Sort the collected diagonal elements
    diagonalElements.sort((a, b) => a - b);

    // Put the sorted elements back into their places on the diagonal
    row = startRow;
    col = startCol;
    for (const element of diagonalElements) {
        matrix[row][col] = element;
        ++row;
        ++col;
    }
}

```

Time and Space Complexity

The provided code has a nested loop structure that iterates over the elements in the matrix in a specific pattern to sort the diagonals. Let's analyze both the time and space complexity:

Time Complexity

The outermost loop is controlled by `min(m, n) - 1`, where `m` is the number of rows and `n` is the number of columns in the matrix. This loop will run for the minimum dimension - 1.

Inside the outer loop, there are two nested loops that each run `m - 1` and `n - 1` times respectively, iterating over the elements of the matrix.

The combined time complexity is the product of these factors, resulting in `O((min(m, n) - 1) * (m - 1) * (n - 1))`.

Simplifying, we drop constants and lower-order terms to focus on the growth rates, which gives us a time complexity of `O(min(m, n) * m * n)`.

Space Complexity

The space complexity of the code is `O(1)` because there are no data structures being used that grow with the size of the input. The only extra space used here is for the loop variables and a couple of temporary variables for the swapping process.