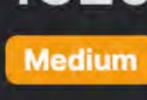
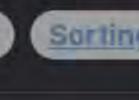
1329. Sort the Matrix Diagonally





Problem Description





diagonal does not affect the elements in another diagonal.

Sorting

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]

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

To solve the problem, we understand that each diagonal can be sorted independently of the others, since the sorting of one

Intuition

[2].

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
- Solution Approach

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

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

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

code will also reflect the inefficiencies of bubble sort, especially as the size of the matrix grows.

significantly reduce the overall time complexity to O(DlogD) for each diagonal of average length D.

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

optimized to handle non-square matrices correctly in all cases.

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

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

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.

Let's walk through a simple example to illustrate the solution approach described. Consider the following 3x3 matrix mat: 1 mat = [[3, 3, 1], [2, 2, 2], [1, 1, 1]

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].

Example Walkthrough

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:

4. The fourth diagonal starting from mat [1] [0] includes the elements [2, 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).

5. The fifth diagonal starting from mat [2] [0] includes the elements [1, 2, 3] (already sorted).

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).

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

- Following the above approach, it seems we don't make any swaps. However, since we have to iterate m 1 times, we will reiterate and compare the elements within the matrix boundaries again.
- 5. After the necessary swap, our diagonal now looks like [1, 1, 2], and it is sorted in ascending order.

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

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

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

def diagonalSort(self, mat: List[List[int]]) -> List[List[int]]:

Only need to iterate up to the smallest dimension minus one

since we start the comparison from the second-to-last diagonal.

if mat[row][col] > mat[row + 1][col + 1]:

* Sorts each diagonal of the given matrix independently, where a diagonal

// Iterate over each element of the matrix except the last row and column

int colCount = matrix[0].size(); // Number of columns in the matrix

// Helper function to sort a single diagonal starting at (startRow, startCol)

// Sort each diagonal that starts from the first column

sortDiagonal(matrix, row, 0, rowCount, colCount);

sortDiagonal(matrix, 0, col, rowCount, colCount);

for (int row = 0; row < rowCount; ++row) {</pre>

for (int col = 1; col < colCount; ++col) {</pre>

return matrix;

int row = startRow;

int col = startCol;

++row;

++col;

row = startRow;

col = startCol;

std::vector<int> diagonalElements;

// Collect all elements of the diagonal

// Sort the collected diagonal elements

for (int element : diagonalElements) {

while (row < rowCount && col < colCount) {

diagonalElements.push_back(matrix[row][col]);

std::sort(diagonalElements.begin(), diagonalElements.end());

// Put the sorted elements back into their places on the diagonal

* is defined from the top-left to the bottom-right corners.

// Determine the number of rows and columns in the matrix

* @param matrix The matrix to be sorted diagonally.

* @return The diagonally sorted matrix.

int numRows = matrix.length;

int numCols = matrix[0].length;

public int[][] diagonalSort(int[][] matrix) {

Swap the elements if they are out of order.

Get the number of rows and columns in the matrix.

num_rows, num_cols = len(mat), len(mat[0])

for _ in range(min(num_rows, num_cols) - 1):

for col in range(num_cols - 1):

for row in range(num_rows - 1):

[3, 2, 1], [2, 1, 2], [1, 1, 2]

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-

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.

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.

Iterate through each element in the matrix except for the last row and column.

Compare the current element with the element in the next diagonal position.

mat[row][col], mat[row + 1][col + 1] = mat[row + 1][col + 1], mat[row][col]

16 17 # Return the sorted matrix. 18 return mat 19

Java Solution class Solution {

Python Solution

class Solution:

10

11

12

13

14

15

6

9

10

11

12

13

14

15

9

10

11

12

13

14

16

17

18

19

20

21

22

23

24

25

26

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

49 };

50

private:

1 mat = [

```
for (int k = 0; k < Math.min(numRows, numCols) - 1; ++k) {</pre>
16
                for (int i = 0; i < numRows - 1; ++i) {
17
                   for (int j = 0; j < numCols - 1; ++j) {
18
19
20
                       // Compare and swap elements if current element is greater
21
                       // than the element in the next row and next column
22
                       if (matrix[i][j] > matrix[i + 1][j + 1]) {
23
                           // Swap elements using a temporary variable
24
                           int temp = matrix[i][j];
25
                           matrix[i][j] = matrix[i + 1][j + 1];
26
                           matrix[i + 1][j + 1] = temp;
27
28
29
30
31
           // Return the sorted matrix
32
33
           return matrix;
34
35 }
36
C++ Solution
1 #include <vector>
2 #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
```

// Sort each diagonal that starts from the first row, except the first diagonal which is already sorted

void sortDiagonal(std::vector<std::vector<int>>& matrix, int startRow, int startCol, int rowCount, int colCount) {

matrix[row][col] = element; 44 ++row; ++col; 47 48

```
Typescript Solution
   type Matrix = number[][];
   // This function sorts the matrix diagonally
    function diagonalSort(matrix: Matrix): Matrix {
                                          // Number of rows in the matrix
        const rowCount = matrix.length;
       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) {</pre>
            sortDiagonal(matrix, row, 0, rowCount, colCount);
10
11
12
13
       // Sort each diagonal that starts from the first row,
       // except for the first diagonal which is already sorted
14
       for (let col = 1; col < colCount; ++col) {</pre>
15
            sortDiagonal(matrix, 0, col, rowCount, colCount);
16
18
        return matrix;
19
20 }
21
   // Helper function that sorts a single diagonal
   // starting at (startRow, startCol)
   function sortDiagonal(matrix: Matrix, startRow: number, startCol: number, rowCount: number, colCount: number): void {
25
       let row = startRow;
26
       let col = startCol;
       const diagonalElements: number[] = [];
28
       // Collect all elements of the diagonal
       while (row < rowCount && col < colCount) {
           diagonalElements.push(matrix[row][col]);
32
           ++row;
33
           ++col;
34
35
       // 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;
45
46
```

37 38 39

Time and Space Complexity

diagonals. Let's analyze both the time and space complexity:

36

Time Complexity

matrix.

*m*n).

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

The provided code has a nested loop structure that iterates over the elements in the matrix in a specific pattern to sort the

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)

Space Complexity

The space complexity of the code is 0(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.