498. Diagonal Traverse

Matrix **Simulation** Medium

Problem Description

The problem provides us with an m x n matrix, named mat, and asks us to return an array of all the elements in the matrix arranged in a specific order. This order follows the diagonal patterns of the matrix. We start from the top-left corner, move diagonally up and to the right, and upon reaching the top or right boundary, we move to the next diagonal, which starts from the leftmost or bottommost element of the current boundary. The process continues until all elements are traversed. It is important to note that when moving along the diagonals, the order of traversal alternates. Diagonals that have a bottom-left to top-right orientation are traversed from top to bottom, while diagonals with a top-right to bottom-left orientation are traversed from bottom to top.

Intuition

then going rightward along the bottom side. The total number of such diagonals would correspond to m + n - 1, considering every element of the first and last row and column except the bottom-right corner element, which is accounted for twice if we simply sum m and n. **Solution Approach**

The solution uses a single-pass algorithm that iterates over the potential starting points for each diagonal in the matrix. A crucial

observation is that if you list the starting points, they follow the border of the matrix, first going downward along the left side and

variable called k which runs from 0 to m + n − 2 inclusive. Here are the steps taken by the algorithm:

A loop begins with k starting from 0 and running up to m + n - 1. Each iteration of this loop corresponds to a diagonal in the matrix.

The solution approach involves iterating through all the possible diagonals in the matrix. These diagonals are indexed by a

- For each k, we determine the starting position (i, j) of the current diagonal. If k is less than n (number of columns), then i starts at 0 and j starts at k. Otherwise, when k is at least n, i starts at k - n + 1 and j starts at n - 1.
- diagonal and collect elements from mat[i][j] as long as i < m and j >= 0, incrementing i and decrementing j. The direction of iteration along the diagonal is important. After collecting elements in the temporary list, if k is even, the order

A temporary list t is created to store the elements of the current diagonal. We use while loop to navigate through the

- of the elements is reversed (using t[::-1]), to adhere to the proper diagonal traversal sequence as specified by the problem. The elements in t (in the correct order) are then added to ans, which is the list that will ultimately be returned.
- After all iterations, ans contains all matrix elements in the desired diagonal order, and the list is returned. This approach uses straightforward indexing to access the matrix elements and employs a list to collect the output. The
- conditional reversal of the temporary list accounts for the zigzag pattern of the traversal, alternating between diagonals. Such an approach is space-efficient since it does not require additional space proportional to the size of the matrix, besides the output

list. It also traverses each element exactly once, making it time-efficient.

Example Walkthrough Let's walk through an example to illustrate the solution approach with a 3×4 matrix.

9 10 11 12

Given matrix mat:

Move to k = 1, representing the second diagonal [2, 5]. k is odd, so we reverse the order when appending to ans. The answer list becomes [1, 5, 2].

For k = 2, the diagonal is [3, 6, 9]. k is even, so append in original order: ans = [1, 5, 2, 3, 6, 9].

We need to collect the elements in the matrix following a specific diagonal pattern:

from top to bottom. Our answer list ans begins with [1].

k = 3 gives diagonal [4, 7, 10]. k is odd, so reverse it: ans = [1, 5, 2, 3, 6, 9, 10, 7, 4]. For k = 4, the diagonal is [8, 11]. k is even: ans = [1, 5, 2, 3, 6, 9, 10, 7, 4, 8, 11].

Finally, k = 5 corresponds to the last diagonal which is simply [12]. Since k is odd, it remains as is: ans = [1, 5, 2, 3, 6, 9,

Start with k = 0, which corresponds to the first diagonal, which is just the element 1. Since k is even, we collect elements

- The final output is [1, 5, 2, 3, 6, 9, 10, 7, 4, 8, 11, 12], the matrix elements arranged in the desired diagonal order.
- Through these steps, the solution iterates over each possible starting point for the diagonals, collects the elements in either forward or reverse order depending on the index of the diagonal, and constructs the output list efficiently. The k loop guarantees

matrix traversal into a 1-dimensional list that preserves the diagonal ordering constraint.

Determine the number of rows and columns in the matrix.

row = 0 if k < num_cols else k - num_cols + 1

diagonal.add(matrix[row][col]);

Collections.reverse(diagonal);

for (int element : diagonal) {

result[index++] = element;

// add the diagonal elements to the result array

// clear the temporary diagonal list for the next iteration

std::vector<int> findDiagonalOrder(std::vector<std::vector<int>>& matrix) {

// Initialize row index (i) and column index (j) for the start of the diagonal

int rows = matrix.size(), columns = matrix[0].size();

// Iterate over all possible diagonals in the matrix

int j = diag < columns ? diag : columns - 1;</pre>

for (int diag = 0; diag < rows + columns - 1; ++diag) {</pre>

int i = diag < columns ? 0 : diag - columns + 1;</pre>

// Collect all elements in the current diagonal

++row;

--col;

if (diag % 2 == 0) {

diagonal.clear();

std::vector<int> result;

std::vector<int> diagonalElements;

if (row < 0 && col < numCols) {</pre>

while (row < numRows && col >= 0) {

result.push(matrix[row][col]);

def findDiagonalOrder(self, matrix: List[List[int]]) -> List[int]:

This is the list which will hold the elements in diagonal order.

There will be (num_rows + num_cols - 1) diagonals to cover in the matrix.

Extend the main result list with the elements of the current diagonal.

Determine the number of rows and columns in the matrix.

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

// If we've exceeded the right-most column, move down to the next row.

// If we've exceeded the bottom row, reset to start from the next column.

// If we've exceeded the left-most column, move up to the next row.

// Move diagonally down-left until the boundary is reached.

row = 0;

row += 2;

col--;

row++;

col--;

row--;

if (col < 0) {

col = 0;

col += 2;

} else {

if (col === numCols) {

if (row === numRows) {

return result;

otherwise we start at 'num_cols - 1' and go down.

This is the list which will hold the elements in diagonal order.

There will be $(num_rows + num_cols - 1)$ diagonals to cover in the matrix.

otherwise we start at an index which goes down from 'num_rows - 1'.

Calculate the starting row index. It is 0 for the first 'num_cols' diagonals,

Calculate the starting column index. It is 'k' for the first 'num_cols' diagonals,

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

for k in range(num_rows + num_cols - 1):

Solution Implementation

that we cover all possible diagonals in the matrix. The given solution is a simple yet effective means to convert 2-dimensional

from typing import List class Solution: def findDiagonalOrder(self, matrix: List[List[int]]) -> List[int]:

Temp list to store the elements of the current diagonal. temp = []

diagonal_order = []

10, 7, 4, 8, 11, 12].

Python

```
col = k if k < num_cols else num_cols - 1</pre>
           # Fetch the elements along the current diagonal.
           # Continue while 'row' is within the matrix row range and 'col' is non-negative.
           while row < num_rows and col >= 0:
                temp.append(matrix[row][col])
                row += 1 # Move down to the next row.
                col -= 1 # Move left to the next column.
           # Reverse every other diagonal's elements before appending it to the result list
           # to get the right order.
            if k % 2 == 0:
                temp = temp[::-1]
           # Extend the main result list with the elements of the current diagonal.
            diagonal_order.extend(temp)
       # Return the final result list.
       return diagonal_order
Java
class Solution {
    public int[] findDiagonalOrder(int[][] matrix) {
       // matrix dimensions
       int m = matrix.length;
       int n = matrix[0].length;
       // result array
       int[] result = new int[m * n];
       // index for the result array
       int index = 0;
       // temporary list to store diagonal elements
       List<Integer> diagonal = new ArrayList<>();
       // loop through each diagonal starting from the top—left corner moving towards the right—bottom corner
        for (int diag = 0; diag < m + n - 1; ++diag) {
           // determine the starting row index for the current diagonal
           int row = diag < n ? 0 : diag - n + 1;
           // determine the starting column index for the current diagonal
           int col = diag < n ? diag : n - 1;
            // collect all the elements from the current diagonal
            while (row < m && col >= 0) {
```

// reverse the diagonal elements if we are in an even diagonal (starting counting from 0)

C++

public:

#include <vector>

class Solution {

#include <algorithm>

```
while (i < rows && j >= 0) {
                diagonalElements.push_back(matrix[i++][j--]);
            // If the diagonal index is even, we need to reverse the diagonal elements
            // to maintain the "zigzag" diagonal order
            if (diag % 2 == 0) {
                std::reverse(diagonalElements.begin(), diagonalElements.end());
            // Append the current diagonal's elements to the result vector
            for (int value : diagonalElements) {
                result.push_back(value);
            // Clear the diagonal elements vector for the next diagonal
            diagonalElements.clear();
        // Return the final vector with all elements in diagonal order
        return result;
};
TypeScript
function findDiagonalOrder(matrix: number[][]): number[] {
    // Initialize the result array where diagonal elements will be stored.
    const result: number[] = [];
    // Determine the number of rows (m) and columns (n) in the matrix.
    const numRows = matrix.length;
   const numCols = matrix[0].length;
    // Define the starting indices for matrix traversal.
    let row = 0;
    let col = 0;
    // A boolean flag to determine the direction of traversal.
    let isUpward = true;
    // Continue traversing until the result array is filled with all elements.
    while (result.length < numRows * numCols) {</pre>
        if (isUpward) {
            // Move diagonally up-right until the boundary is reached.
            while (row >= 0 && col < numCols) {</pre>
                result.push(matrix[row][col]);
                row--;
                col++;
           // If we've exceeded the top row, reset to start from the next column.
```

```
// Flip the direction for the next iteration.
          isUpward = !isUpward;
      // Return the array containing elements in diagonal order.
      return result;
from typing import List
```

diagonal_order = []

class Solution:

```
for k in range(num_rows + num_cols - 1):
   # Temp list to store the elements of the current diagonal.
   temp = []
   # Calculate the starting row index. It is 0 for the first 'num_cols' diagonals,
   # otherwise we start at an index which goes down from 'num_rows - 1'.
    row = 0 if k < num_cols else k - num_cols + 1
   # Calculate the starting column index. It is 'k' for the first 'num_cols' diagonals,
   # otherwise we start at 'num_cols - 1' and go down.
   col = k if k < num_cols else num_cols - 1</pre>
   # Fetch the elements along the current diagonal.
   # Continue while 'row' is within the matrix row range and 'col' is non-negative.
   while row < num_rows and col >= 0:
       temp.append(matrix[row][col])
       row += 1 # Move down to the next row.
       col -= 1 # Move left to the next column.
   # Reverse every other diagonal's elements before appending it to the result list
   # to get the right order.
   if k % 2 == 0:
       temp = temp[::-1]
```

The given code snippet traverses through all the elements of a two-dimensional matrix mat of size m x n diagonally, collecting the elements of each diagonal and appending them to the final result in a specified order.

diagonal_order.extend(temp)

Return the final result list.

return diagonal_order

Time and Space Complexity

Time Complexity: O(m * n)

Since the matrix has m rows and n columns, there are m * n elements in total. The outer loop runs for m + n - 1 iterations, where each diagonal is processed. In each iteration of this loop, a while-loop runs, collecting elements along the current diagonal. Each element in the matrix is processed exactly once inside the nested while-loops. Thus, the total number of operations is proportional to the number of elements in the matrix, leading to a time complexity of 0(m * n). Space Complexity: O(min(m, n))

The time complexity of the code is determined by the number of iterations needed to traverse all the elements of the matrix.

The space complexity of the code is primarily due to the auxiliary data structure t used for storing each diagonal before appending it to the result list ans. The length of a diagonal is at most min(m, n), because the diagonals are limited either by the number of rows or the number of columns. The reversing operation, t[::-1], creates a new list of the same size as t, and thus, the temporary space required is also in the order of min(m, n). Since the additional space needed is not dependent on the total number of elements but rather on the longer dimension of the input matrix, the space complexity is 0(min(m, n)). The result list ans grows to hold all m * n elements, but this does not contribute to space complexity in the analysis, as it is

required to hold the output of the function. Hence, we only consider the additional space used by the algorithm beyond the output space.