1424. Diagonal Traverse II

Sorting Heap (Priority Queue)

## **Problem Description**

Medium

column indices) should be grouped together. Additionally, within each diagonal group, elements should be sorted based on their column indices. This creates a zigzag pattern if we visualize it, as we traverse diagonals starting from the top row, moving downright on the grid, and then up to the next diagonal start point. The challenge is to implement this in a way that respects the diagonal grouping and within-group ordering, and to do so as efficiently as possible.

Given a 2-dimensional grid nums, where each cell contains an integer, our task is to return a list of all the elements of nums but in

a specific order—diagonal order. In this ordering, elements from the same diagonal (elements with the same sum of their row and

The intuition behind the solution lies in recognizing that elements that are on the same diagonal share a certain property: the sum

Intuition

the sum of its indices, and use this sum as a way to group elements into their respective diagonals.

### of their row and column indices is constant. With this realization, we can iterate through every element in the nums grid, calculate

To maintain the within-diagonal order (where elements with lower column indices come first), we keep track of the column index alongside the diagonal index sum. This means that for each element, we store a triple containing the sum (which represents the diagonal), the column index, and the value itself.

Once we have gone through all the elements and collected this information, we sort this list of triples. The sort will naturally group elements from the same diagonal together (because of the sums), and within each group, it will order elements by their column indices. Finally, we extract the third element of each triple (which is the value from the original grid) to get our result in the correct

diagonal order. The fact that the Python sort function sorts tuples lexicographically, first by the first element, then by the second, etc., makes it particularly well-suited for this task. Solution Approach

The implementation of the solution makes efficient use of Python's list and sorting capabilities. Here's a step-by-step explanation of the approach:

Initialize an empty list arr: This list is used to store the triples (sum of indices, column index, value) for each element in the

# Iterate over the grid: We use a nested loop to go through each element of the nums grid. The outer loop (for i, row in

nums array.

enumerate(nums)) iterates over the rows, and the inner loop (for j, v in enumerate(row)) goes over each element in a row.

operation arranges the elements in exactly the order required for the final result.

• Enumerate Function: Provides a neat way to get both the index and the value when iterating over a list.

• Triple (Tuple): An immutable data structure to store the related elements - sum of indices, column index, and value.

Here,  $\mathbf{i}$  is the row index,  $\mathbf{j}$  is the column index, and  $\mathbf{v}$  is the value at the nums  $[\mathbf{i}]$   $[\mathbf{j}]$  position. Compute the diagonal index and store triples: For each grid element nums[i][j], we compute the sum of the indices (i +

j) which uniquely identifies the diagonal to which the element belongs. We then create a triple (i + j, j, v) and append it to our arr list. The triple contains the diagonal index, the column index, and the value itself, respectively.

Sort the list of triples: We call arr.sort() to sort the list of triples. The sort method uses the natural lexicographical order

for tuples, so it will first sort them by the diagonal index, and then by their column index within the same diagonal. This

- Extract the results: The last line is a list comprehension [v[2] for v in arr] that goes through the sorted list of triples and extracts the third element from each triple (the value). This creates a new list of integers which is our final, diagonally ordered list of elements.
- Sorting: A built-in Python method to sort lists in ascending order based on the first element of the tuple, and if they are the same, based on the second element, and so on. This approach has a time complexity of O(N log N), where N is the total number of elements in the nums array, because the

most expensive operation is sorting the list of triples. The spatial complexity is O(N) since we need to create a list arr that holds

as many triples as there are elements in the original array.

The key algorithms and data structures used in this approach are:

**Example Walkthrough** 

• **Nested Loops**: To iterate over a 2D array.

• List Comprehension: A concise way to construct lists.

- Let's use a small 3×3 grid example to illustrate the solution approach:
- nums = [[1, 2, 3],

Following the steps in the solution approach: **Initialize an empty list arr:** 

#### Now we start the nested loop over the grid: $\circ$ For i = 0, row = [1, 2, 3]:

(4, 2, 9)

**Sort the list of triples:** 

(3, 2, 6), (4, 2, 9)]

Solution Implementation

 $diagonal_elements = []$ 

# Iterate through the matrix rows

# within the same diagonal line.

diagonal\_elements.sort()

for row index, row in enumerate(matrix):

# Iterate through the elements in the current row

for column index, value in enumerate(row):

return [element[2] for element in diagonal\_elements]

\* @param nums 2D List of integers representing a matrix.

public int[] findDiagonalOrder(List<List<Integer>> nums) {

List<int[]> diagonalElements = new ArrayList<>();

for (int i = 0; i < nums.size(); i++) {</pre>

// Iterate over the 2D List to process each element

for (int j = 0; j < nums.get(i).size(); j++) {</pre>

**Python** 

Iterate over the grid:

 $\circ$  For i = 1, row = [4, 5, 6]:

[4, 5, 6],

[7, 8, 9]

```
■ For j = 1, v = 2: Compute diagonal index i + j = 1, append (1, 1, 2) to arr.
■ For j = 2, v = 3: Compute diagonal index i + j = 2, append (2, 2, 3) to arr.
```

■ For j = 0, v = 1: Compute diagonal index i + j = 0, append (0, 0, 1) to arr.

■ For j = 0, v = 4: Compute diagonal index i + j = 1, append (1, 0, 4) to arr.

At the beginning, arr is an empty list: arr = []

```
■ For j = 1, v = 5: Compute diagonal index i + j = 2, append (2, 1, 5) to arr.
   ■ For j = 2, v = 6: Compute diagonal index i + j = 3, append (3, 2, 6) to arr.
\circ For i = 2, row = [7, 8, 9]:
   ■ For j = 0, v = 7: Compute diagonal index i + j = 2, append (2, 0, 7) to arr.
   ■ For j = 1, v = 8: Compute diagonal index i + j = 3, append (3, 1, 8) to arr.
   ■ For j = 2, v = 9: Compute diagonal index i + j = 4, append (4, 2, 9) to arr.
arr now looks like this: [(0, 0, 1), (1, 1, 2), (2, 2, 3), (1, 0, 4), (2, 1, 5), (3, 2, 6), (2, 0, 7), (3, 1, 8),
```

After sorting, arr should look like this: [(0, 0, 1), (1, 0, 4), (1, 1, 2), (2, 0, 7), (2, 1, 5), (2, 2, 3), (3, 1, 8),

Notice the elements are grouped by the sum of their indices (the diagonals) and sorted by their column indices within each

By extracting the third element of each tuple, we get the diagonally ordered list of elements: [1, 4, 2, 7, 5, 3, 8, 6, 9]

group. **Extract the results:** 

That's the diagonal order traversal of the 2D nums grid using the proposed solution approach.

from typing import List class Solution: def findDiagonalOrder(self, matrix: List[List[int]]) -> List[int]: # Initialize an empty list to store the values along with their diagonal indices

Java import java.util.List;

\* Function to find the diagonal order of a given list of lists of integers.

// Calculate the diagonal index and store it with the element

diagonalElements.add(new int[] {i + j, j, nums.get(i).get(j)});

// The format of the stored array is [diagonal index, column index, value]

\* Diagonals are considered in a "zig-zag"/top-right to bottom-left manner.

\* @return An array of integers representing the diagonal traversal.

// Temporary list to store elements and their diagonal indices

# The sum of row and column indices gives the diagonal index.

# Append a tuple containing the diagonal index, column index and the value

diagonal\_elements.append((row\_index + column\_index, column\_index, value))

# Extract the values from the sorted list of tuples and return them in the correct order

# Sort the list of tuples based on diagonal index, then by column index (as secondary)

# This will order the elements first by diagonal, then from top-right to bottom-left

#### import java.util.ArrayList; import java.util.Collections; class Solution {

/\*\*

```
// Sort the diagonalElements based on their computed diagonal indices
        // If two elements are on the same diagonal, sort them by their column index
        Collections.sort(diagonalElements, (a, b) -> {
            if (a[0] == b[0]) {
                return a[1] - b[1];
            } else {
                return a[0] - b[0];
        });
        // Initialize the answer array with the correct size
        int[] ans = new int[diagonalElements.size()];
        // Extract the values from the sorted diagonalElements and populate the answer array
        for (int i = 0; i < diagonalElements.size(); i++) {</pre>
            ans[i] = diagonalElements.get(i)[2];
        // Return the final diagonal order traversal as an array
        return ans;
C++
#include <vector>
#include <tuple>
#include <algorithm>
class Solution {
public:
    // Function to find and return the elements of the 2D vector in diagonal order.
    vector<int> findDiagonalOrder(vector<vector<int>>& nums) {
        // Create a vector to store tuples containing the diagonal number, column number, and value.
        vector<tuple<int, int, int>> diagonalElements;
        // Iterate over the 2D vector to populate diagonalElements.
        for (int row = 0; row < nums.size(); ++row) {</pre>
            for (int col = 0; col < nums[row].size(); ++col) {</pre>
                // The sum of row and column indices determines the diagonal number.
                // Store diagonal number, column number, and element.
                diagonalElements.push_back({row + col, col, nums[row][col]});
        // Sort the elements of diagonalElements. By default, tuples are sorted by their
        // first element, then by their second element, so this will sort by the diagonal number,
        // then by the column index.
        sort(diagonalElements.begin(), diagonalElements.end());
        // Create a vector to store the answer in the diagonal order.
        vector<int> ans:
        // Extract the elements from the tuples in sorted order and append them to ans.
        for (auto& element : diagonalElements) {
            ans.push_back(get<2>(element));
```

```
for (let row = 0; row < nums.length; ++row) {</pre>
```

**}**;

**TypeScript** 

// Return the answer.

type DiagonalElement = [number, number, number];

function findDiagonalOrder(nums: number[][]): number[] {

// Iterate over the 2D array to populate diagonalElements.

for (let col = 0; col < nums[row].length; ++col) {</pre>

let diagonalElements: DiagonalElement[] = [];

// Function to find and return the elements of the 2D array in diagonal order.

// Store diagonal number, column number, and element.

diagonalElements.push([row + col, col, nums[row][col]]);

// Create an array to store tuples containing the diagonal number, column number, and value.

// The sum of row and column indices determines the diagonal number.

return ans;

```
// Sort the elements of diagonalElements. Since TypeScript requires a comparator function,
    // it's provided here. This function sorts by the diagonal number, then by the column index.
    diagonalElements.sort((a, b) => {
        if (a[0] === b[0]) return a[1] - b[1]; // If diagonal numbers match, sort by column.
        return a[0] - b[0]; // Otherwise, sort by diagonal number.
   });
    // Create an array to store the answer in the diagonal order.
    let answer: number[] = [];
    // Extract the elements from the tuples in sorted order and append them to answer.
    for (let element of diagonalElements) {
        answer.push(element[2]);
    // Return the answer.
    return answer;
// Example of how to use the function.
// const matrix: number[][] = [
       [1, 2, 3],
      [4, 5, 61,
       [7, 8, 9]
// 1:
// const diagonalOrder: number[] = findDiagonalOrder(matrix);
from typing import List
class Solution:
    def findDiagonalOrder(self. matrix: List[List[int]]) -> List[int]:
        # Initialize an empty list to store the values along with their diagonal indices
       diagonal_elements = []
       # Iterate through the matrix rows
        for row index, row in enumerate(matrix):
            # Iterate through the elements in the current row
            for column index, value in enumerate(row):
                # The sum of row and column indices gives the diagonal index.
                # Append a tuple containing the diagonal index, column index and the value
                diagonal_elements.append((row_index + column_index, column_index, value))
       # Sort the list of tuples based on diagonal index, then by column index (as secondary)
       # This will order the elements first by diagonal, then from top-right to bottom-left
```

# within the same diagonal line.

diagonal\_elements.sort()

# Extract the values from the sorted list of tuples and return them in the correct order return [element[2] for element in diagonal\_elements] Time and Space Complexity The time complexity of the code is determined by the number of operations it performs. The given Python code iterates over all the elements in the list of lists to create a flat list of tuples (arr). The iteration has a complexity of O(N), where N is the total number of elements in the nums list of lists. Additionally, the code sorts the arr list, which has O(N log N) complexity for the Timsort algorithm used in Python's sort() method. Since the sort is the most expensive operation here, the overall time complexity is O(N log N).

main extra space used is for the arr list of tuples. Since each element in the input is transformed into a tuple and stored in arr, the space complexity is O(N), where N is again the total number of elements in nums.

The space complexity of the code considers the additional space used by the algorithm outside the input and output data. The