867. Transpose Matrix

Easy Array Matrix Simulation

Problem Description

The problem requires us to transpose a given 2D integer array, `matrix`. Transposing a matrix involves flipping the matrix over its main diagonal. This process converts rows to columns and vice versa, which leads to the interchange of the matrix's row and column indexes.

For example, let's consider a matrix as follows:

```
1 2 3
4 5 6
```

The main diagonal of this matrix is the set of elements that extend from the top left to the bottom right (elements `1` and `5` in this case).

When we transpose the matrix, the rows become columns, and the columns become rows. The transposed matrix will look like this:

```
1 4
2 5
3 6
```

The element that was originally at the second row, first column (`4`), is now at the first row, second column. The solution requires us to perform this operation on any given matrix and return the new transposed matrix.

Intuition

For the given solution, Python's built-in functions simplify the process of transposing a matrix. Here is the intuition behind the used approach:

- The `*` operator, when used in the context of function argument unpacking, will unpack the argument list. For a 2D matrix, this effectively unpacks the rows of the matrix, making them available as individual arguments.
- The `zip` function takes iterables (can be zero or more), aggregates them in a tuple, and returns it. When used with a 2D matrix unpacked into rows, `zip` essentially combines the elements of the rows that have the same index, thus forming the columns of the transposed matrix.
- Finally, the `list` function converts the resulting tuples back into lists, as required for the solution. In Python, the `zip` function returns an iterator of tuples. To match the expected format of the solution, we convert each tuple into a list.

Solution Approach

Implementing the solution for transposing a matrix in Python is quite straightforward thanks to Python's powerful syntax and built-in functions. The provided reference solution uses almost no explicit algorithms because high-level function calls handle the necessary operations. Nevertheless, it's beneficial to break down the solution to understand the underlying patterns and behavior.

Here's the provided solution for reference:

```python
class Solution:
    def transpose(self, matrix: List[List[int]]) -> List[List[int]]:
        return list(zip(*matrix))
```

Let's walk through how it works, step by step:

1. Function Argument Unpacking (`*` Operator):
   - The first step involves an advanced Python pattern called argument unpacking. In the expression `zip(*matrix)`, the `*` operator is used to unpack the 2D `matrix` list.
   - Essentially, if the `matrix` is a list of lists like `[ [a1, a2], [b1, b2], [c1, c2] ]`, calling `zip(*matrix)` is the same as calling `zip([a1, a2], [b1, b2], [c1, c2])`. Every individual list in `matrix` is passed as a separate argument to `zip`.
2. `zip` Function:
   - The `zip` function takes any number of iterables and returns an iterator of tuples, where each tuple contains the i-th element from each of the input iterables.
   - When applied to rows of a matrix, `zip` effectively groups together the elements of the matrix by their column indices. For instance, `a1` will be paired with `b1` and `c1`, forming the first tuple of the new row in the transposed matrix.
3. `list` Conversion:
   - The output from `zip` is an iterator of tuples. The `list()` function is used to convert these tuples into lists, as the problem expects a list of lists structure for the transposed matrix.

Since there is no nested loop or manual iteration, the entire operation is quite efficient. The actual transposition — the core algorithmic task — is completely delegated to the `zip` function, which is a built-in, highly optimized component of Python. The clever use of argument unpacking with `*` allows us to avoid manual index handling or iterating through rows and columns of the matrix, which would be needed in a more traditional, lower-level language solution.

Ultimately, this solution showcases the power of Python in terms of writing concise and readable code that leverages high-level functions and language features to perform complex operations with minimal code.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Consider the following 2D integer array, `matrix`:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6]
]
```

We want to transpose this matrix, which will result in flipping the matrix over its main diagonal. To do this, we'll apply the steps in the solution approach.

1. **Function Argument Unpacking (`*` Operator)**:
   - We use the `*` operator to unpack the matrix's rows as arguments to the `zip` function. We can visualize this step as taking the two rows `[1, 2, 3]` and `[4, 5, 6]` and unpacking them such that they're passed to `zip` like `zip([1, 2, 3], [4, 5, 6])`.
2. **`zip` Function**:
   - The `zip` function then takes these two lists and pairs elements at the same positions together, resulting in tuples. The output of `zip` given our two rows would be an iterator that generates the following tuples one by one: `(1, 4)`, `(2, 5)`, `(3, 6)`.
   - These tuples represent the rows of the new, transposed matrix. The first tuple `(1, 4)` will be the first row, the second tuple `(2, 5)` will be the second row, and so on.
3. **`list` Conversion**:
   - Next, we convert each of these tuples back into lists using the `list` function since the expected output format is a list of lists.
   - Applying the `list` function to the iterator of tuples from `zip`, we obtain the transposed matrix as a list of lists: `[[1, 4], [2, 5], [3, 6]]`.

After going through the steps with our example `matrix`, the final transposed matrix is:

```
[
    [1, 4],
    [2, 5],
    [3, 6]
]
```

This walkthrough demonstrates how the provided Python solution transposes a matrix efficiently by utilizing function argument unpacking, the `zip` function, and list conversion to transform the matrix's rows into the transposed matrix's columns. The elegance of the solution lies in its simplicity and effective use of Python's built-in functionality to accomplish the task with a single line of code.

Solution Implementation

Python

```python
# Import typing module to use type hints
from typing import List

class Solution:
    def transpose(self, matrix: List[List[int]]) -> List[List[int]]:
        # Transpose the input matrix.
        # This is done by unpacking the rows of the matrix as arguments to the zip function.
        # zip(*matrix) couples elements with the same index from each row together, effectively transposing the elements.
        # Then, the zip object is converted into a list of lists, which is the transposed matrix.
        transposed_matrix = [list(row) for row in zip(*matrix)]

        # Return the transposed matrix.
        return transposed_matrix
```

Java

```java
class Solution {

    // Function to transpose a given matrix
    public int[][] transpose(int[][] matrix) {
        // 'rows' is the number of rows in the input matrix
        int rows = matrix.length;
        // 'cols' is the number of columns in the input matrix which is derived from the first row
        int cols = matrix[0].length;

        // 'transposedMatrix' is the transposed matrix where the rows and columns are swapped
        int[][] transposedMatrix = new int[cols][rows];

        // Iterate over each column of the transposed matrix
        for (int i = 0; i < cols; i++) {
            // Iterate over each row of the transposed matrix
            for (int j = 0; j < rows; j++) {
                // Assign the value from the original matrix to the correct position in the transposed matrix
                transposedMatrix[i][j] = matrix[j][i];
            }
        }
        // Return the transposed matrix
        return transposedMatrix;
    }
}
```

C++

```cpp
#include <vector> // Include vector from Standard Template Library (STL)

// Solution class
class Solution {
public:
    // Function to transpose a given matrix
    // @param originalMatrix: the original matrix to be transposed
    // @return: a new matrix which is the transpose of the original matrix
    vector<vector<int>> transpose(vector<vector<int>>& originalMatrix) {
        int rowCount = originalMatrix.size();      // Number of rows in the matrix
        int columnCount = originalMatrix[0].size(); // Number of columns in the matrix

        // Create a new matrix with dimensions swapped (columns x rows)
        vector<vector<int>> transposedMatrix(columnCount, vector<int>(rowCount));

        // Iterate over each element in the new matrix
        for (int i = 0; i < columnCount; ++i) {
            for (int j = 0; j < rowCount; ++j) {
                // Assign the value from the original matrix to the new position
                // in the transposed matrix by swapping indices
                transposedMatrix[i][j] = originalMatrix[j][i];
            }
        }
        return transposedMatrix; // Return the transposed matrix
    }
};
```

TypeScript

```typescript
/**
 * Transposes a given matrix (converts rows to columns and vice versa).
 * @param {number[][]} matrix The matrix to be transposed.
 * @return {number[][]} The transposed matrix.
 */
function transpose(matrix: number[][]): number[][] {
    // Get the number of rows in the matrix.
    const rowCount: number = matrix.length;
    // Get the number of columns in the matrix.
    const columnCount: number = matrix[0].length;

    // Initialize a new matrix with dimensions swapped (rows become columns and vice versa).
    const transposedMatrix: number[][] = new Array(columnCount)
        .fill(0)
        .map(() => new Array(rowCount).fill(0));

    // Iterate over each column of the new transposed matrix.
    for (let i: number = 0; i < columnCount; ++i) {
        // Iterate over each row of the new transposed matrix.
        for (let j: number = 0; j < rowCount; ++j) {
            // Assign the transposed value from the original matrix to the new matrix.
            transposedMatrix[i][j] = matrix[j][i];
        }
    }

    // Return the newly formed transposed matrix.
    return transposedMatrix;
}
```

```python
# Import typing module to use type hints
from typing import List

class Solution:
    def transpose(self, matrix: List[List[int]]) -> List[List[int]]:
        # Transpose the input matrix.
        # This is done by unpacking the rows of the matrix as arguments to the zip function.
        # zip(*matrix) couples elements with the same index from each row together, effectively transposing the elements.
        # Then, the zip object is converted into a list of lists, which is the transposed matrix.
        transposed_matrix = [list(row) for row in zip(*matrix)]

        # Return the transposed matrix.
        return transposed_matrix
```

Time and Space Complexity

The provided code receives a matrix and transposes it using Python's built-in `zip` function combined with argument unpacking (`*`).

Time Complexity:

The time complexity for transposing a matrix involves iterating over each element exactly once. In this code, `zip` takes `m` sequences (rows), where `m` is the number of rows of the matrix, and combines them into `n` tuples, where `n` is the number of columns. Since each element is touched once during the operation, the time complexity is `O(m*n)` where `m` is the number of rows and `n` is the number of columns in the original matrix.

Space Complexity:

The `zip` function creates `n` tuples (where `n` is the number of columns of the input matrix), each containing `m` elements (where `m` is the number of rows of the input matrix), and `list()` then converts these tuples into lists. This operation creates a new list of lists with the same number of elements as the original matrix. Therefore, the space complexity is also `O(m*n)`, as it requires additional space proportional to the size of the input matrix.