2352. Equal Row and Column Pairs

Medium Array Hash Table Matrix Simulation

Problem Description

The problem presents a square integer matrix (`grid`) that has a size of `n x n`. We are tasked with finding the number of pairs (`r_i, c_j`), where `r_i` represents the rows and `c_j` represents the columns of the matrix. For a pair of row and column to be counted, they must have the same elements in the exact same order - essentially, they should form identical arrays.

To better understand the problem, let's consider a simple example. Suppose we have the following matrix:

```
grid = [
  [1, 2],
  [2, 1]
]
```

Here, we can compare each row with each column. We can see that the first row `[1, 2]` does not match either column (first column is `[1, 2]`, second column is `[2, 1]`). However, the second row `[2, 1]` matches the second column `[2, 1]`. Thus, there is only one equal pair in this matrix: the pair consisting of the second row and the second column.

Intuition

When approaching this solution, we have to consider that we can't simply compare each row directly to each column since they are not stored in the same format within the matrix. Rows are sublists of the matrix, while columns can be viewed as tuples formed by taking the same element index from each row.

The solution employs two main steps:

1. Transpose the matrix. This is achieved using the `zip` function along with unpacking the original `grid` matrix. By doing this, we obtain a transposed version of the original grid, where rows are now columns and vice versa. For the given example, the transposed `grid` would look like this:

```
g = [
  [1, 2],
  [2, 1]
]
```

2. Count the equal pairs: We can now iterate over each row of the original `grid` and for each row, iterate over each row of the transposed `grid` (`g`). Whenever we find a row in the original `grid` matches a row in `g`, it means that the original row and the corresponding column of the original grid form an equal pair. We sum up all such instances to get the final result.

Solution Approach

The implementation of the solution can be broken down as follows:

- The `zip` function is a built-in Python function used to aggregate elements from two or more iterables (lists, tuples, etc.). It takes `n` iterables and returns a list (or in Python 3.x, an iterator) of tuples, where the `i`-th tuple contains the `i`-th element from each of the iterable arguments. Hence, applying `zip` to the original `grid` matrix, while using the unpacking operator `*`, effectively transposes the matrix. This new matrix `g` holds the columns of the original `grid` as its rows.

  The best way to visualize this is by comparison:

  Original `grid`:

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

  Transposed `g` using `zip`:

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

- The next step is using list comprehension to iterate through each row of the original `grid` and simultaneously through each row of the transposed grid `g`. At every iteration, the row from the original grid is compared to the current row in `g` using the `==` operator, which checks if both lists contain the same elements in the same order.
- The condition `row == col` returns a `True` (which is interpreted as `1`) if the row and column are identical, and `False` (interpreted as `0`) if they aren't. The `sum` function is then used to add up these truth values to get the total count of matching pairs.
- The technique of using a nested for loop within the list comprehension is a common algorithm pattern that allows for checking each combination of elements between two sequences or structures. This pattern is known as "Cartesian product" and is essential in different domains for creating every possible pairing of sets of items.

Here is how the Python code uses these steps with the algorithmic patterns:

```
class Solution:
    def equalPairs(self, grid: List[List[int]]) -> int:
        g = [list(col) for col in zip(*grid)]  # Transpose the grid by converting the zipped tuples to lists
        return sum(row == col for row in grid for col in g)  # Count equal row-column pairs
```

As we can see, the solution is concise and leverages powerful built-in Python functions and list comprehensions to solve the problem efficiently. The time complexity of this solution is `O(n^2)`, where `n` is the size of the grid's row or column since every row is compared with every column.

Example Walkthrough

Consider a 3×3 square matrix as our `grid`:

```
grid = [
  [8, 2, 9],
  [5, 1, 4],
  [7, 6, 3]
]
```

To find the number of equal pairs of rows and columns, let's apply our solution approach step by step:

1. **Transpose the matrix `grid`:**

By using the `zip` function with unpacking, we transpose the `grid`. The transposed matrix `g` would be the following:

```
g = [
  [8, 5, 7],
  [2, 1, 6],
  [9, 4, 3]
]
```

This is the visual representation of how the `zip` function has taken the first element of each row and created the first row of `g`, and so on.

2. **Count the equal pairs:**

Now we compare each row of the original `grid` with each row of the transposed `g` and count the instances where they are identical:

- Compare `grid` row `[8, 2, 9]` with `g`:
  - It does not match any row in `g`.
- Compare `grid` row `[5, 1, 4]` with `g`:
  - It matches the first row of `g` (`[8, 5, 7]`) at the second position, but each row has different elements.
- Compare `grid` row `[7, 6, 3]` with `g`:
  - The third row of `grid` (`[7, 6, 3]`) is identical to the third row of `g` (`[7, 6, 3]`), so we have a match.

In this case, we find that there is exactly *one* equal pair consisting of the third row of `grid` and the third row (which corresponds to the third column of the original `grid`) of `g`.

Using the code provided in the solution approach, we would have:

```
class Solution:
    def equalPairs(self, grid: List[List[int]]) -> int:
        g = [list(col) for col in zip(*grid)] # This makes `g` as shown above
        return sum(row == col for row in grid for col in g) # This counts and returns 1 for our example
```

When run with our example `grid`, `equalPairs` would return `1`, indicating that there is one pair of a row and a column that has the same elements in the exact same order.

Solution Implementation

Python

```
class Solution:
    def equal_pairs(self, grid: List[List[int]]) -> int:
        # Transpose the input grid to get columns as rows
        transposed_grid = [list(column) for column in zip(*grid)]

        # Initialize a counter for equal pairs
        equal_pairs_count = 0

        # Iterate through each row in the original grid
        for row_in_grid in grid:
            # For each row, compare it with each row in the transposed grid (which are originally columns)
            for transposed_row_in_transposed_grid:
                # If the original row and the transposed row (column) match, increment the counter
                if row == transposed_row:
                    equal_pairs_count += 1

        # Return the total count of equal row-column pairs
        return equal_pairs_count
```

```
Note: The `List` import from typing is assumed as per the original code snippet. If you need to run this code outside of a typing-aware environment, you should include the import statement.
```

```
```python
from typing import List
```

Java

```
class Solution {
    public int equalPairs(int[][] grid) {
        // Get the length of the grid, which is also the number of rows and columns (n by n)
        int n = grid.length;

        // Create a new grid 'transposedGrid' which will store the transposed version of 'grid'
        int[][] transposedGrid = new int[n][n];

        // Transpose the grid: turn rows of 'grid' into columns of 'transposedGrid'
        for (int j = 0; j < n; ++j) {
            for (int i = 0; i < n; ++i) {
                transposedGrid[i][j] = grid[j][i];
            }
        }

        // Initialize a counter for the number of equal pairs
        int equalPairsCount = 0;

        // Iterate through each row of the original grid
        for (var row : grid) {
            // Iterate through each column in the transposed grid
            for (var column : transposedGrid) {
                // Initialize a flag to check if the current row and column are equal
                int areEqual = 1;
                // Compare corresponding elements of the current row and column
                for (int k = 0; k < n; ++k) {
                    if (row[k] != column[k]) {
                        // If there's a mismatch, set the flag to zero and break the loop
                        areEqual = 0;
                        break;
                    }
                }
                // If the row and column are equal, increment the count
                equalPairsCount += areEqual;
            }
        }
        // Return the total count of equal pairs
        return equalPairsCount;
    }
}
```

C++

```
#include <vector>
using namespace std;

class Solution {
public:
    int equalPairs(vector<vector<int>>& grid) {
        // Get the size of the grid.
        int size = grid.size();

        // Create a new grid to hold the transposed matrix.
        vector<vector<int>> transposed(size, vector<int>(size));

        // Transpose the original grid and fill the transposed grid.
        for (int col = 0; col < size; ++col) {
            for (int row = 0; row < size; ++row) {
                transposed[row][col] = grid[col][row];
            }
        }

        // Initialize a counter for the number of equal pairs.
        int equalPairCount = 0;

        // Compare each row of the original grid with each row of the transposed grid.
        for (auto& rowOriginal : grid) {
            for (auto& rowTransposed : transposed) {
                // If a pair is identical, increment the counter.
                equalPairCount += (rowOriginal == rowTransposed); // Implicit conversion of bool to int (true -> 1, false -> 0)
            }
        }

        // Return the total count of equal pairs.
        return equalPairCount;
    }
};
```

TypeScript

```
// Function to count the number of equal pairs in a grid
// A pair is equal if one row of the grid is identical to one column of the grid
function equalPairs(grid: number[][]): number {
    // Determine the size of the grid
    const gridSize: number = grid.length;

    // Create a new grid to store the transposed version of the original grid
    let transposedGrid: number[][] = Array.from({ length: gridSize }, () => Array(n).fill(0));

    // Transpose the original grid and fill the transposedGrid
    for (let i = 0; i < gridSize; ++i) {
        for (let j = 0; j < gridSize; ++j) {
            transposedGrid[i][j] = grid[j][i];
        }
    }

    // Initialize a counter to keep track of equal pairs
    let equalPairsCount: number = 0;

    // Compare each row of the original grid with each column of the transposed grid
    for (const row of grid) {
        for (const col of transposedGrid) {
            // Increment the count if the row and column are identical
            equalPairsCount += Number(row.toString() === col.toString());
        }
    }

    // Return the total number of equal pairs
    return equalPairsCount;
}
```

```
class Solution:
    def equal_pairs(self, grid: List[List[int]]) -> int:
        # Transpose the input grid to get columns as rows
        transposed_grid = [list(column) for column in zip(*grid)]

        # Initialize a counter for equal pairs
        equal_pairs_count = 0

        # Iterate through each row in the original grid
        for row in grid:
            # For each row, compare it with each row in the transposed grid (which are originally columns)
            for transposed_row in transposed_grid:
                # If the original row and the transposed row (column) match, increment the counter
                if row == transposed_row:
                    equal_pairs_count += 1

        # Return the total count of equal row-column pairs
        return equal_pairs_count
```

```
Note: The `List` import from typing is assumed as per the original code snippet. If you need to run this code outside of a typing-aware environment, you should include the import statement.
```

```
```python
from typing import List
```

Time and Space Complexity

The time complexity of the given code is `O(n^2)` where `n` is the size of one dimension of the square matrix `grid`. This is because the code includes two nested loops each iterating through `n` elements (rows and columns). The comparison `row == col` inside the loops is executed `n * n` times. Each comparison operation takes `O(n)`, thus overall, the time complexity is `O(n^3)`.

The space complexity of the code is `O(n^2)`. The list `g` is created by taking a transpose of the original grid, which requires additional space proportional to the size of the grid, hence `n * n`.