1260. Shift 2D Grid

Easy Array Matrix Simulation

Leetcode Link

Problem Description

In this problem, we are provided with a 2D grid that represents a matrix of size `m x n` and an integer `k`. The task is to perform a 'shift' operation on the grid exactly `k` times, where each shift operation moves elements in the grid to their adjacent right position with wrapping around at the edges. This means that:

- An element at `grid[i][j]` moves to `grid[i][j + 1]`.
- An element at the end of a row (`grid[i][n - 1]`) moves to the starting of the next row (`grid[i + 1][0]`).
- The bottom-right element (`grid[m - 1][n - 1]`) moves to the very first position of the grid (`grid[0][0]`).

We need to return the grid as it looks after performing all `k` shifts.

Intuition

To solve this problem, we aim to simulate the process of shifting elements. However, directly shifting elements in the grid `k` times would be inefficient, especially for large values of `k`. Instead, we can take advantage of the observation that after `m * n` shifts, the grid will look exactly as it started (here, `m * n` represents the total number of elements in the grid).

By using this fact, we can calculate the effective number of shifts needed by taking the remainder of `k` divided by the total number of elements `m * n` (`k % (m * n)`). This way, we minimize the number of shifts to no more than `m * n - 1`.

To implement the shifts, the solution takes the following steps:

1. Flatten the grid into a one-dimensional list `t`, thus simplifying the problem to shifting elements in an array.
2. Since we are shifting to the right, we take the last `k` elements of the array `t[-k:]` and move them to the front, effectively performing a rotation. The remaining elements `t[:-k]` are then appended to the result of the previously rotated elements to form the new array after the shift.
3. Finally, we map the elements back to the original grid shape (`m x n`) by iterating through the matrix indices and assigning the values from the shifted list back into the grid.

The given solution is efficient and avoids unnecessary computations by leveraging modular arithmetic and array slicing to achieve the desired shifts before reconstructing the final matrix.

Solution Approach

The solution implementation uses a straightforward approach to solve the problem by manipulating the grid as a one-dimensional list and then reshaping it back to the 2D grid. Here's the breakdown of the implementation:

1. First, we determine the size of the grid with `m, n = len(grid), len(grid[0])`. This gives us the number of rows (`m`) and columns (`n`) in the grid.
2. Since the grid is effectively a circular data structure after `m * n` shifts, we compute the effective shifts needed: `k %= m * n`. This step uses the modulus operator to ensure we only shift the minimum required number of times as shifting `m * n` times would return the grid to its original configuration.
3. We then flatten the grid into a one-dimensional list `t`: `t = [grid[i][j] for i in range(m) for j in range(n)]`. This step uses list comprehension to traverse all elements in a row-major order.
4. The next step involves rotating the list (shifting the elements to the right by `k` places). This is accomplished with list slicing: `t = t[-k:] + t[:-k]`. Here, `t[-k:]` obtains the last `k` elements which will move to the front of the list, and `t[:-k]` gets the elements before the last `k`, which will now follow the shifted `k` elements.
5. Finally, we map the elements from the rotated list back to the 2D grid form. This is done using nested loops where we traverse the rows and columns of the grid and set each element to its corresponding value from the rotated list:

```
for i in range(m):
    for j in range(n):
        grid[i][j] = t[i * n + j]
```

   - For each index `(i, j)` in the grid, we calculate the one-dimensional index using `i * n + j`, which is then used to access the correct value from the list `t`.
6. The modified grid after all `k` shifts is then returned.

The algorithm makes use of simple list manipulations, modular arithmetic, and the efficiency of Python's list slicing operations to provide a swift resolution to the grid shifting problem. It avoids the more computationally expensive direct simulation of the shifts and elegantly manages the wrapping around behavior by treating the 2D grid as a circular list.

Example Walkthrough

Let's illustrate the solution approach using a small example grid and a number of shifts `k`. Consider the following grid `grid` with dimensions `2 x 3` and an integer `k = 4`.

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

1. **Determine Grid Size**: First, we recognize the grid has `m = 2` rows and `n = 3` columns.
2. **Compute Effective Shifts**: Since the total number of elements is `m * n = 6`, we compute the effective number of shifts `k %= 6`, which is `k = 4 % 6 = 4`. It means we only need to shift elements four positions to the right.
3. **Flatten the Grid**: Next, we flatten the grid into a one-dimensional list.

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

4. **Rotate the List**: To simulate the shift operation, we perform a rotation on the list. Given `k = 4`, we slice and re-arrange the list:

```
t[-k:] = [3, 4, 5, 6]
t[:-k] = [1, 2]
```

   After rotation, the list `t` will look like:

```
t = [3, 4, 5, 6] + [1, 2] = [3, 4, 5, 6, 1, 2]
```

5. **Map Rotated List Back to Grid**: Lastly, we convert the rotated list back into a 2D grid form using the formulas from our approach. This will restructure the list into rows and columns based on the grid's dimensions.

```
grid[0][0] = t[0 * 3 + 0] = t[0] = 3
grid[0][1] = t[0 * 3 + 1] = t[1] = 4
grid[0][2] = t[0 * 3 + 2] = t[2] = 5
grid[1][0] = t[1 * 3 + 0] = t[3] = 6
grid[1][1] = t[1 * 3 + 1] = t[4] = 1
grid[1][2] = t[1 * 3 + 2] = t[5] = 2
```

   The final grid after 4 shifts will be:

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

And that's how you apply the given solution to perform shift operations on a grid. The grid `grid` after four shifts turns out to be `[ [3, 4, 5], [6, 1, 2] ]`, which demonstrates the effectiveness of this method for moving elements to their right positions while wrapping around the edges.

Python Solution

```python
class Solution:
    def shiftGrid(self, grid: List[List[int]], k: int) -> List[List[int]]:
        # Number of rows and columns in the grid
        num_rows, num_columns = len(grid), len(grid[0])

        # Normalize k to be within the range of the number of elements in grid to avoid unnecessary full cycles
        k %= num_rows * num_columns

        # Flatten the grid into a single list 'flattened_grid'
        flattened_grid = [grid[row][column] for row in range(num_rows) for column in range(num_columns)]

        # Shift the flattened_grid by k positions to the right
        shifted_grid = flattened_grid[-k:] + flattened_grid[:-k]

        # Reconstruct the original grid structure with the new shifted positions
        for row in range(num_rows):
            for col in range(num_columns):
                grid[row][col] = shifted_grid[row * num_columns + col]

        # Return the modified grid after k shifts
        return grid

```

Java Solution

```java
class Solution {
    public List<List<Integer>> shiftGrid(int[][] grid, int k) {
        // Get the dimensions of the grid.
        int rowCount = grid.length;
        int colCount = grid[0].length;

        // Ensure the number of shifts 'k' is within the grid's boundary.
        k %= (rowCount * colCount);

        // Create a result list that will contain the shifted grid.
        List<List<Integer>> result = new ArrayList<>();

        // Initialize the result list with zeros in preparation for value assignment.
        for (int row = 0; row < rowCount; ++row) {
            List<Integer> newRow = new ArrayList<>();
            for (int col = 0; col < colCount; ++col) {
                newRow.add(0);
            }
            result.add(newRow);
        }

        // Iterate over each element in the grid to compute its new position after shifting.
        for (int row = 0; row < rowCount; ++row) {
            for (int col = 0; col < colCount; ++col) {
                // Compute the one-dimensional equivalent index of the current position.
                int currentOneDIndex = row * colCount + col;
                // Compute the new one-dimensional index after shifting 'k' times.
                int newOneDIndex = (currentOneDIndex + k) % (rowCount * colCount);

                // Convert the one-dimensional index back to two-dimensional grid coordinates.
                int newRow = newOneDIndex / colCount;
                int newCol = newOneDIndex % colCount;

                // Assign the current value to its new position in the result list.
                result.get(newRow).set(newCol, grid[row][col]);
            }
        }

        // Return the result which contains the shifted grid.
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to shift the elements of a 2D grid.
    std::vector<std::vector<int>> shiftGrid(std::vector<std::vector<int>>& grid, int k) {
        // Extracting the number of rows and columns of the grid.
        int rows = grid.size();
        int columns = grid[0].size();

        // Creating a new 2D vector to store the answer.
        std::vector<std::vector<int>> shiftedGrid(rows, std::vector<int>(columns));

        // Iterating over each element of the grid.
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < columns; ++col) {
                // Calculating the new position of each element after shifting 'k' times.
                int newPosition = (row * columns + col + k) % (rows * columns);

                // Placing the element in its new position in the result grid.
                shiftedGrid[newPosition / columns][newPosition % columns] = grid[row][col];
            }
        }

        // Returning the result grid after performing the shifts.
        return shiftedGrid;
    }
};

```

Typescript Solution

```typescript
function shiftGrid(grid: number[][], k: number): number[][] {
    // m denotes the number of rows in the grid
    const numRows = grid.length;
    // n denotes the number of columns in the grid
    const numCols = grid[0].length;
    // Total number of elements in the grid
    const totalSize = numRows * numCols;
    // Normalize k to prevent unnecessary rotations
    k %= totalSize;

    // If no shift is required or the grid is effectively 1x1, return the original grid
    if (k === 0 || totalSize <= 1) return grid;

    // Flatten the 2D grid into a 1D array
    const flatArray = grid.flat();

    // Initialize the answer array with the same structure as the original grid
    let shiftedGrid = Array.from({ length: numRows }, () => new Array(numCols));

    // Fill the new grid by shifting the elements according to 'k'
    for (let i = 0, shiftIndex = totalSize - k; i < totalSize; i++) {
        // Calculate the new position for the element
        shiftedGrid[Math.floor(i / numCols)][i % numCols] = flatArray[shiftIndex];

        // Update the shiftIndex to cycle through the flatArray
        shiftIndex = shiftIndex === totalSize - 1 ? 0 : shiftIndex + 1;
    }

    // Return the newly shifted grid
    return shiftedGrid;
}

```

Time and Space Complexity

The given Python code performs a shift operation on a 2D grid by `k` places.

Time Complexity

The time complexity of the code is determined by several factors:

1. Calculating the total number of cells in the grid, which is `m * n`. This is constant-time operation `O(1)`.
2. Flattening the grid into a 1D list, which involves iterating through all the cells. This takes `O(m * n)` time.
3. Slicing the list to perform the shift operation. The list slicing operation in Python takes `O(k)` time in the worst case, where `k` is the number of elements being sliced. In this case, it's also `O(m * n)` because `k` is at most `m * n` after the modulo operation.
4. Iterating through the grid again to place the shifted values back into the 2D grid. This is another `O(m * n)` operation.

Combining these steps, the overall time complexity is `O(m * n)` because all other operations are constant time or also `O(m * n)`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm, not including the space used by the input and output.

1. Storing the flattened grid as a 1D list `t` requires `O(m * n)` space.
2. The space needed for storing shifted list slices before concatenation is also `O(m * n)`.

Overall, the space complexity is `O(m * n)` due to the additional list created to store the grid elements.