2946. Matrix Similarity After Cyclic Shifts

Easy Array Math Matrix Simulation

Problem Description

In this problem, we are given a 2D matrix of integers where each element of the matrix is at row `i` and column `j`. You are instructed to perform a specific operation on the matrix:

- If the row index `i` is odd, you are to cyclically shift that row `k` times to the right.
- If the row index `i` is even, you are to cyclically shift that row `k` times to the left.

A cyclic shift to the right means that the last column of the matrix will move to the first position, and all other columns will move one position to the right. Conversely, a cyclic shift to the left means that the first column of the matrix will move to the last position, and all other columns will move one position to the left.

Your task is to determine if after applying these shifts `k` times to each row, the final matrix obtained is identical to the initial matrix. You need to return `true` if they are identical, and `false` otherwise.

Intuition

The solution approach involves simulating the cyclic shifts on the matrix to compare the final positions of each element with their original positions.

Since we want to avoid actually performing `k` shifts (which could be costly if `k` is large), we instead calculate the final position of each element directly using modular arithmetic:

- For an odd indexed row (shifts to the right): The new position of an element initially at index `j` will now be `(j + k) % n` where `n` is the number of columns.
- For an even indexed row (shifts to the left): The new position of an element initially at index `j` will now be `(j - k + n) % n`.

The `% n` ensures that the calculation wraps around the row correctly, simulating the cyclic nature of the shift.

As we iterate through every element of the matrix, we check if after applying either the left or right shift (based on whether the row index is even or odd), the element in the original matrix is the same as the element in its calculated new position after the shift. If for any element the values differ, we can return `false` right away as the matrices will not be identical after the shifts. If we complete the iteration without finding any differences, we return `true`.

Solution Approach

The solution is quite straightforward and does not require complex data structures or algorithms. It simply makes use of a nested loop to iterate through each element in the input matrix `mat`, and applies conditional checks and modular arithmetic to validate the matrix's shift requirements.

Here's a step-by-step explanation of the implementation:

1. We first determine the number of columns `n` in the matrix by inspecting the length of the first row (`len(mat[0])` since it's a 0-indexed `m x n` integer matrix).
2. The outer loop iterates through each row of the matrix. For each row, our goal is to determine if its elements would end up in the same position if we cyclically shifted the row `k` times, corresponding to its index (even or odd).
3. Inside this outer loop, we create an inner loop to iterate over each element (`j`) in the current row.
4. We use a conditional (`if-else`) statement to check if we are at an even (`i % 2 == 0`) or odd (`i % 2 == 1`) row, as this determines the direction of the shift (left or right).
5. For each element `j`, we calculate its expected position after the shifts, as follows:
   - If we're in an odd-indexed row (`i % 2 == 1`), we calculate the new index by right shifting, using `(j + k) % n`. We check if the current element `x` matches the element in the column `(j + k) % n` of the initial matrix.
   - If we're in an even-indexed row (`i % 2 == 0`), we calculate the new index by left shifting, using `(j - k + n) % n`. We check if the current element `x` matches the element in the column `(j - k + n) % n` of the initial matrix.
6. If at any point during the iteration we find a mismatch (i.e., `x` does not equal the value at its new position), we immediately return `false`, as the matrix will no longer be the same after the cyclic shifts.
7. If the loop completes successfully without finding any mismatches, it means all elements retain their position after the cyclic shifts, and we return `true` signifying that the initial and final matrices are exactly the same.

The code snippet for this solution makes use of efficient indexing and modular arithmetic to avoid the need for actual shifting, which is an optimization that allows the algorithm to run in $O(m * n)$ time, where `m` is the number of rows and `n` is the number of columns in the matrix `mat`. This time complexity is derived from the fact that we must check every element in the matrix exactly once.

Example Walkthrough

Let's assume we have the following small 3×3 matrix `mat` and the number of times we need to perform the shift is `k = 2`.

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

Following the steps of the solution approach to check whether a cyclic shift of `k` times would result in the same matrix:

1. Number of columns `n` is 3 as there are 3 elements in the first row of `mat`.
2. We iterate through each row:
   - Row 0 (even index, shift left): `[1, 2, 3]`
     - Element `1` (index 0) would move to index `0 - 2 + 3) % 3 = 1`. But the element at index 1 is `2`. Not a match, so we return `false`.

   Thus, without needing to continue further, we can already conclude that the matrix will not be identical after the shifts since in the very first step, the element `1` would not end up in the same position.

Here's how the matrix would look after applying the cyclic shifts considering all elements, although the mismatch in the first check precludes this:

Even row (row 0): Left shift by `k` (2 times): Original row: `[1, 2, 3]` → After shift: `[3, 1, 2]`

Odd row (row 1): Right shift by `k` (2 times): Original row: `[4, 5, 6]` → After shift: `[5, 6, 4]`

Even row (row 2): Left shift by `k` (2 times): Original row: `[7, 8, 9]` → After shift: `[9, 7, 8]`

It's evident that the shifted matrix would not match the original `mat`. Hence, our function would return `false`:

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

// This does not match the original `mat`, so the result is false.
```

This example illustrates the core intuition behind the prescribed solution approach and showcases how an early mismatch can shortcut the process, providing a quick response without having to process the entire matrix.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def are_similar(self, matrix: List[List[int]], shift_amount: int) -> bool:
        '''
        Check if the matrix has a special similarity property.

        The function checks if each row (when considered as a ring)
        can be rotated by a fixed number of steps 'shift_amount' to
        match either the next row or the previous row, depending on
        whether the index is odd or even.

        Args:
        matrix : A list of lists, where each sublist represents a row of the matrix.
        shift_amount : An integer representing the fixed number of steps of rotation.

        Returns:
        A boolean value indicating whether the matrix has the similarity as per the mentioned conditions.
        '''

        # Calculate the number of columns in the matrix
        num_columns = len(matrix[0])

        # Loop through each row and each element in the row
        for row_index, row in enumerate(matrix):
            for col_index, element in enumerate(row):
                # For odd-indexed rows, check if the current element matches the element in the shifted
                # position of the same row to the right by 'shift_amount' steps
                if row_index % 2 == 1 and element != matrix[row_index][(col_index + shift_amount) % num_columns]:
                    return False

                # For even-indexed rows, check if the current element matches the element in the shifted
                # position of the same row to the left by 'shift_amount' steps
                if row_index % 2 == 0 and element != matrix[row_index][(col_index - shift_amount + num_columns) % num_columns]:
                    return False

        # If all elements match the shifted positions, the matrix has the similarity property
        return True
```

Java

```java

class Solution {

    // Method to check if a given matrix can be made similar by rotating every row either left or right
    public boolean areSimilar(int[][] matrix, int k) {
        // m is the number of rows in the matrix
        int numRows = matrix.length;
        // n is the number of columns in the matrix
        int numCols = matrix[0].length;

        // Normalize the rotation step k to be in the range of [0, numCols)
        k %= numCols;

        // Loop through each row of the matrix
        for (int i = 0; i < numRows; ++i) {
            // Loop through each column of the current row
            for (int j = 0; j < numCols; ++j) {
                // For even rows, check if rotating to the right gives the correct entry
                if ((i % 2 == 1) && matrix[i][j] != matrix[i][(j + k) % numCols]) {
                    return false; // If not, the matrix cannot be made similar
                }
                // For even rows, check if rotating to the left gives the correct entry
                if ((i % 2 == 0) && matrix[i][j] != matrix[i][(j - k + numCols) % numCols]) {
                    return false; // If not, the matrix cannot be made similar
                }
            }
        }

        // If all rows satisfy the condition, the matrix can be made similar
        return true;
    }
}
```

C++

```cpp

#include <vector>

class Solution {
public:
    bool areSimilar(std::vector<std::vector<int>>& matrix, int rotationCount) {
        int numRows = matrix.size(); // Number of rows in the matrix
        int numCols = matrix[0].size(); // Number of columns in the matrix

        // Reduce the rotation count to its equivalent
        // within the range of the number of columns.
        rotationCount %= numCols;

        // Loop through each cell in the matrix.
        for (int row = 0; row < numRows; ++row) {
            for (int col = 0; col < numCols; ++col) {
                // If the current row is odd, check if the value in the
                // current cell is equal to the value in the cell
                // obtained by rotating the current row right by rotationCount steps.
                if (row % 2 == 1 && matrix[row][col] != matrix[row][(col + rotationCount) % numCols]) {
                    return false; // Return false if they are not equal.
                }

                // If the current row is even, check if the value in the
                // current cell is equal to the value in the cell
                // obtained by rotating the current row left by rotationCount steps.
                if (row % 2 == 0 && matrix[row][col] != matrix[row][(col - rotationCount + numCols) % numCols]) {
                    return false; // Return false if they are not equal.
                }
            }
        }

        // If all comparisons passed, the rotated matrix is similar
        // to the original matrix.
        return true;
    }
};
```

TypeScript

```typescript

function areSimilar(matrix: number[][], shiftAmount: number): boolean {
    const numRows = matrix.length;  // m represents the number of rows in the matrix.
    const colCount = matrix[0].length;  // n represents the number of columns in the matrix.
    shiftAmount %= colCount;  // Normalize the shifting amount to be within the bounds of the columns.

    // Loop through each row of the matrix.
    for (let row = 0; row < numRows; ++row) {
        // Loop through each column in the current row.
        for (let col = 0; col < colCount; ++col) {
            // If the row is odd (1-indexed), check if the current element is equal to the element at the shifted position.
            if (row % 2 === 1 && matrix[row][col] !== matrix[row][(col + shiftAmount) % colCount]) {
                return false;
            }

            // If the row is even (1-indexed), check if the current element is equal to the element at the shifted position.
            if (row % 2 === 0 && matrix[row][col] !== matrix[row][(col - shiftAmount + colCount) % colCount]) {
                return false;
            }
        }
    }
    return true; // If all elements match their respective shifted positions, return true.
}
```

```python

from typing import List

class Solution:
    def are_similar(self, matrix: List[List[int]], shift_amount: int) -> bool:
        '''
        Check if the matrix has a special similarity property.

        The function checks if each row (when considered as a ring)
        can be rotated by a fixed number of steps 'shift_amount' to
        match either the next row or the previous row, depending on
        whether the index is odd or even.

        Args:
        matrix : A list of lists, where each sublist represents a row of the matrix.
        shift_amount : An integer representing the fixed number of steps of rotation.

        Returns:
        A boolean value indicating whether the matrix has the similarity as per the mentioned conditions.
        '''

        # Calculate the number of columns in the matrix
        num_columns = len(matrix[0])

        # Loop through each row and each element in the row
        for row_index, row in enumerate(matrix):
            for col_index, element in enumerate(row):
                # For odd-indexed rows, check if the current element matches the element in the shifted
                # position of the same row to the right by 'shift_amount' steps
                if row_index % 2 == 1 and element != matrix[row_index][(col_index + shift_amount) % num_columns]:
                    return False

                # For even-indexed rows, check if the current element matches the element in the shifted
                # position of the same row to the left by 'shift_amount' steps
                if row_index % 2 == 0 and element != matrix[row_index][(col_index - shift_amount + num_columns) % num_columns]:
                    return False

        # If all elements match the shifted positions, the matrix has the similarity property
        return True
```

Time and Space Complexity

Time Complexity

The time complexity of the function is determined by the nested loops which iterate over every element of the matrix. Since `mat` is an `n x n` matrix, where `n` is the length of its rows, and there are two for-loops iterating over `n` rows and `n` columns, the time complexity is $O(n^2)$.

Space Complexity

The space complexity of the function is $O(1)$. This is because the function uses a constant amount of additional memory space, regardless of the input size. There are only a few variable assignments, and there's no use of any additional data structures that scale with input size.