2319. Check if Matrix Is X-Matrix

Easy Array Matrix

Problem Description

The problem requires us to determine if a given square matrix can be classified as an X-Matrix. A square matrix can be considered an X-Matrix if it meets two conditions:

1. All elements on both its main diagonal (from top-left to bottom-right) and its anti-diagonal (from top-right to bottom-left) must be non-zero.
2. All other elements, which are not part of the diagonals, must be zero.

Given an input matrix named `grid`, which is represented as a 2D integer array of size `n x n`, our task is to return `true` if `grid` is an X-Matrix, and `false` otherwise.

Intuition

To solve this problem, we can iterate over each element of the matrix and verify it against the two conditions provided for an X-Matrix. We can follow these steps:

1. Loop through each element in the matrix using a nested loop, where `i` is the index for rows, and `j` is the index for columns.
2. Check if the current element (`grid[i][j]`) belongs to either the main diagonal or the anti-diagonal. This is true when `i == j` (main diagonal) or `i + j == len(grid) - 1` (anti-diagonal).
3. If the current element belongs to one of the two diagonals, check if it is non-zero. If it is zero, we can immediately return `false`, as it violates the first condition for an X-Matrix.
4. If the current element does not belong to a diagonal, check if it is zero. If it isn't, return `false` since this violates the second condition.
5. If none of these violations occur during the traversal, return `true`, as the matrix satisfies both conditions for an X-Matrix.

Solution Approach

The solution leverages a straightforward approach without using any additional data structures or complex patterns. It is purely based on element-wise inspection of the matrix. Here is how the code implements the strategy to check if the given matrix is an X-Matrix:

1. We use two nested loops to traverse each element of `grid`, where the outer loop uses the variable `i` to iterate over rows, and the inner loop uses `j` to iterate over columns. This allows us to check each element (denoted as `v`).
2. The main diagonal is where the row index is equal to the column index (`i == j`). The anti-diagonal can be identified in a square matrix of size `n` by the condition where the sum of the row index and column index equals `n - 1` (`i + j == len(grid) - 1`).
3. Inside the loop, we check if the element belongs to either the main or anti-diagonal by evaluating the above two conditions. If the current element is part of a diagonal, we verify if it's non-zero. If a zero is found on any diagonal, the function immediately returns `false`, as it contradicts the first rule of an X-Matrix.
4. If the current element does not lie on a diagonal, it must be zero to fulfill the second condition of an X-Matrix. Thus, any non-zero value encountered in this case leads to a return value of `false`.
5. If the loop concludes without finding any violations of the X-Matrix rules, it means all diagonals contain non-zero values and all other elements are zero. Thus the function returns `true`, confirming that the `grid` is an X-Matrix.

The implementation is efficient, with a time complexity of O(n^2), which is required to check every element, and space complexity of O(1), as no additional space is required beyond input and variables for iteration.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Consider a small `3x3` matrix (`grid`):

```
[
   [2, 0, 3],
   [0, 4, 0],
   [1, 0, 5]
]
```

We need to determine if this matrix is an X-Matrix according to the rules provided.

1. To check if each diagonal element is non-zero, we start with the first element on the main diagonal, `grid[0][0]`, which is `2`. It's non-zero, so we proceed.
2. Checking the next main diagonal element, `grid[1][1]`, we find a `4`. It's also non-zero, so we continue.
3. Checking the last element on the main diagonal, `grid[2][2]`, we see a `5`. It's non-zero as well, hence the main diagonal condition is satisfied.
4. Next, we move to the anti-diagonal. We start with `grid[0][2]`, which is `3`, and then `grid[1][1]` (which we've already checked), and finally `grid[2][0]`, which is `1`. All these elements are also non-zero, satisfying the anti-diagonal condition.
5. None of the diagonal elements are zero, so the first condition for an X-Matrix is met.
6. Now we check all other elements, which should all be zero. We review `grid[0][1]`, `grid[1][0]`, `grid[1][2]`, and `grid[2][1]`. They are all zero, as required.
7. Since we have verified that all diagonal elements are non-zero and all other elements are zero, we confirm that the matrix is an X-Matrix. Therefore, the function would return `true`.

In this example, we've walked through the matrix and checked both conditions specified for an X-Matrix. The matrix given fulfills both conditions, so it is indeed an X-Matrix.

Solution Implementation

Python

```python
class Solution:
    def checkXMatrix(self, grid: List[List[int]]) -> bool:
        n = len(grid)  # The dimension of the square grid
        # Iterate through each element of the grid
        for i in range(n):
            for j in range(n):
                # Check if we are on the main or secondary diagonal
                if i == j or i + j == n - 1:
                    # If the value on the diagonal is 0, the condition fails
                    if grid[i][j] == 0:
                        return False
                else:
                    # If the value is not on the diagonal and is not 0, the condition fails
                    if grid[i][j] != 0:
                        return False
        # If all conditions are satisfied, return True
        return True
```

Java

```java
class Solution {

    // Function to check if the given grid forms an X-Matrix
    public boolean checkXMatrix(int[][] grid) {
        // Get the length of the grid (since it's an N x N matrix)
        int n = grid.length;

        // Loop through each element of the grid
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                // Check the diagonal and anti-diagonal
                if (i == j || i + j == n - 1) {
                    // On the diagonals, all elements should be non-zero
                    // If a zero is found, the grid is not an X-Matrix
                    if (grid[i][j] == 0) {
                        return false;
                    }
                } else {
                    // For all other positions (off the diagonals), elements should be zero
                    // If a non-zero number is found, the grid is not an X-Matrix
                    if (grid[i][j] != 0) {
                        return false;
                    }
                }
            }
        }

        // If all conditions are met, then it's an X-Matrix
        return true;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to check if a given grid forms an X-Matrix
    bool checkXMatrix(vector<vector<int>>& grid) {
        // Get the size of the grid
        int n = grid.size();

        // Iterate over each element in the grid
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                // Check the diagonals: primary (i == j) and secondary (i + j == n - 1)
                if (i == j || i + j == n - 1) {
                    // If an element on the diagonal is zero, the grid does not form an X-Matrix
                    if (grid[i][j] == 0) {
                        return false;
                    }
                }
                // Check the elements that are not on the diagonals
                else {
                    // If an off-diagonal element is not zero, the grid does not form an X-Matrix
                    if (grid[i][j] != 0) {
                        return false;
                    }
                }
            }
        }
        // Return true if all diagonal elements are non-zero and all off-diagonal elements are zero
        return true;
    }
};
```

TypeScript

```typescript
// Function to check if a given 2D grid forms an X-Matrix.
// An X-Matrix has non-zero integers on both its diagonals,
// and zeros on all other positions.
function checkXMatrix(grid: number[][]): boolean {
    // Get the size of the grid.
    const size = grid.length;

    // Loop over each element in the grid.
    for (let row = 0; row < size; ++row) {
        for (let col = 0; col < size; ++col) {
            // Check the main diagonal and anti-diagonal elements
            if (row === col || row + col === size - 1) {
                // If any diagonal element is 0, return false.
                if (grid[row][col] === 0) {
                    return false;
                }
            } else {
                // If any off-diagonal element is non-zero, return false.
                if (grid[row][col] !== 0) {
                    return false;
                }
            }
        }
    }

    // If no rule is violated, return true.
    return true;
}
```


```python
class Solution:
    def checkXMatrix(self, grid: List[List[int]]) -> bool:
        n = len(grid)  # The dimension of the square grid
        # Iterate through each element of the grid
        for i in range(n):
            for j in range(n):
                # Check if we are on the main or secondary diagonal
                if i == j or i + j == n - 1:
                    # If the value on the diagonal is 0, the condition fails
                    if grid[i][j] == 0:
                        return False
                else:
                    # If the value is not on the diagonal and is not 0, the condition fails
                    if grid[i][j] != 0:
                        return False
        # If all conditions are satisfied, return True
        return True
```


Time and Space Complexity

The given code snippet is designed to check whether a given square 2D grid is an 'X-Matrix'. An 'X-Matrix' has non-zero elements on its diagonals and zero elements elsewhere.

Time Complexity

To determine the time complexity, we look at the number of operations relative to the input size. The code iterates over every element in the `N` x `N` grid exactly once, performing a constant amount of work for each element by checking if it's on the diagonal or anti-diagonal and then validating the value. Therefore, the time complexity is `O(N^2)`, where `N` is the length of the grid's side.

Space Complexity

The solution only uses a fixed number of variables (`i`, `j`, `v`) and does not allocate any additional space that grows with the input size. Hence, the space complexity is `O(1)` as it does not depend on the size of the input grid.