1572. Matrix Diagonal Sum

Easy Array Matrix Leetcode Link

Problem Description

The problem provides us a square matrix `mat`, which means the number of rows and columns in the matrix are equal. Our task is to calculate the sum of the diagonal elements, which includes elements from both the primary diagonal and the secondary diagonal. The primary diagonal is the one that starts from the top left corner and ends at the bottom right corner. The secondary diagonal starts from the top right corner and ends at the bottom left corner. However, there's a catch: If any element is common between the primary and secondary diagonals (which would be the case for the central element in a matrix with odd dimensions), we must include it only once in our sum.

Intuition

To approach this problem, we consider the primary and secondary diagonals of the matrix.

- The primary diagonal elements have the same index for their row and column. In terms of indexes, these are the elements `mat[i][i]` where `i` ranges from 0 to the `n-1`, wherein `n` is the size of one dimension since it's a square matrix.
- The secondary diagonal elements have row and column indexes that sum up to `n-1`. In other words, the indexes are of the form `mat[i][j]` where `j` is `n-i-1`.

Now, the challenge is to make sure that we don't double-count the element in the case of an odd-dimension matrix where the primary and secondary diagonals intersect. To avoid this, we simply check if the index `i` is equal to the index `j`. If they are equal, this means we're looking at the central element in the case of an odd-size matrix and we shouldn't add it again.

Therefore, the sum `ans` starts at 0, and we iterate through each row with its index `i`. While iterating, we calculate `j` as `n-i-1` for each row to pinpoint the element in the secondary diagonal. We then add to `ans` the sum of elements `mat[i][i]` and `mat[i][j]`, unless `i == j`, in which case we only add the element `mat[i][i]` once.

By iterating over all rows, we calculate the sum required by the problem without having to deal with two separate loops or maintaining additional data structures, therefore making our approach both efficient and straightforward.

Solution Approach

The solution provided is written in Python and follows a straightforward approach that efficiently computes the sum of the diagonals of a square matrix.

Here's a detailed walkthrough of the implementation:

1. We start by initializing a variable `ans` to 0. This will hold the cumulative sum of the diagonal elements.
2. We need the size of the matrix, which is the length of one of its sides (since it is square), so we take the length of the matrix `n = len(mat)`.
3. The `for` loop is used to iterate over each row of the matrix. The loop variable `i` serves as the index for both rows and the primary diagonal elements. `enumerate` is used so we can have both the index and the row elements available during each iteration.
4. We then calculate the column index `j` for the secondary diagonal element corresponding to the current row. As explained earlier, `j` is given by `n - i - 1`.
5. Inside the loop, we add to `ans` the value of the primary diagonal element `row[i]`.
6. We also want to add the secondary diagonal element `row[j]` unless `i` is equal to `j` (which is the case when we are at the central element of a matrix with an odd number of rows/columns). To do this succinctly, we add `row[j]` only if `j != i`, otherwise, we add `0`. This conditional addition is achieved by the expression `(0 if j == i else row[j])`.
7. After the loop has processed all the rows, `ans` now contains the total sum of both diagonals, with no duplicates for the intersecting element (if any).
8. Finally, the method returns the computed `ans`.

There are no complex algorithms, data structures, or patterns involved here. The solution effectively uses index manipulation to address the specific elements required for the sum, making it a simple yet effective approach to solving this problem.

Here is the core logic encapsulated in Python code:

```python
class Solution:
    def diagonalSum(self, mat: List[List[int]]) -> int:
        ans = 0
        n = len(mat)
        for i, row in enumerate(mat):
            j = n - i - 1
            ans += row[i] + (0 if j == i else row[j])
        return ans
```

By carefully choosing and manipulating indexes, we maintain a clear and concise solution without additional memory usage for storing intermediate results or redundant computations.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we have the following 3×3 square matrix:

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

Here's the step-by-step process of how the solution works:

1. Initialize `ans` to 0. This will keep track of the sum of diagonal elements.
2. Determine the size `n` of the matrix. In this case, `n = len(mat) = 3`.
3. Start the `for` loop with `i` iterating from 0 to 2 (since the matrix has 3 rows and columns).
4. For each row `i`, calculate the index `j` for accessing the secondary diagonal's element, which is `n - i - 1`. This will give us `2, 1, 0` for `i = 0, 1, 2` respectively.
5. Now, add the primary diagonal element to `ans`. In the first iteration with `i = 0`, add `mat[0][0]` which is `1`.
6. Check if `i` equals `j`. If they are not the same, add the secondary diagonal element to `ans`. With `i = 0` and `j = 2`, they aren't equal, so add `mat[0][2]` which is `3`.
7. For the second iteration where `i = 1`, we add `mat[1][1]` (the middle element) to `ans`. Since `i` and `j` are equal here (both equal to 1), we don't add the secondary diagonal element because that would be double-counting.
8. In the third iteration with `i = 2`, add the primary diagonal element `mat[2][2]` which is `9` to `ans`. Index `i` and `j` are not the same (because `j = 0`), so add the secondary diagonal element `mat[2][0]` which is `7`.
9. After adding these elements through the loop, the sum `ans` becomes `1 + 3 + 5 + 9 + 7 = 25`.
10. The method returns the value of `ans`, which is `25`. This is the required sum of the elements on the primary and secondary diagonals of the matrix, without double-counting the center element.

Putting it all into the Python code, we have:

```python
class Solution:
    def diagonalSum(self, mat: List[List[int]]) -> int:
        ans = 0
        n = len(mat)
        for i, row in enumerate(mat):
            j = n - i - 1
            ans += row[i] + (0 if j == i else row[j])
        return ans
```

When we call `diagonalSum(mat)` with our example matrix, it will return `25`, which is the correct answer. This demonstrates the efficiency and simplicity of the provided solution approach.

Python Solution

```python
class Solution:
    def diagonalSum(self, matrix: List[List[int]]) -> int:
        # Initialize the sum of the diagonals
        total_sum = 0

        # Get the size of the matrix (assuming it's square)
        n = len(matrix)

        # Loop over each row and calculate the diagonal sum
        for i, row in enumerate(matrix):
            # Calculate the index for the secondary diagonal
            j = n - i - 1
            # Add the primary diagonal element
            total_sum += row[i]
            # Add the secondary diagonal element if it's not the same as the primary diagonal
            if j != i:
                total_sum += row[j]

        # Return the computed sum
        return total_sum

```

Java Solution

```java
class Solution {
    public int diagonalSum(int[][] matrix) {
        int totalSum = 0; // This will hold the sum of the diagonal elements
        int size = matrix.length; // The matrix is size x size

        // Loop through each row of the matrix
        for (int i = 0; i < size; ++i) {
            int reverseIndex = size - i - 1; // Calculate the corresponding column index for the secondary diagonal

            // Add the primary diagonal element
            totalSum += matrix[i][i];

            // If it's not the same element (which would be the case in the middle of an odd-sized matrix)
            // then add the secondary diagonal element
            if (i != reverseIndex) {
                totalSum += matrix[i][reverseIndex];
            }
        }

        // Return the sum of primary and secondary diagonals, excluding the middle element if counted twice
        return totalSum;
    }
}

```

C++ Solution

```cpp
#include<vector>

class Solution {
public:
    // Function to calculate the sum of the elements on the diagonals of a square matrix
    int diagonalSum(std::vector<std::vector<int>>& mat) {
        int total = 0; // Used to store the sum of the diagonal elements
        int size = mat.size(); // Get the size of the square matrix

        // Iterate through each row of the matrix
        for (int rowIndex = 0; rowIndex < size; ++rowIndex) {
            int colIndex = size - rowIndex - 1; // Calculate the column index for the secondary diagonal

            // Sum the primary diagonal element
            total += mat[rowIndex][rowIndex];

            // Sum the secondary diagonal element only if it's not the same as the primary diagonal element
            if (rowIndex != colIndex) {
                total += mat[rowIndex][colIndex];
            }
        }
        return total; // Return the sum of the diagonal elements
    }
};

```

Typescript Solution

```typescript
function diagonalSum(matrix: number[][]): number {
    // 'matrixSize' stores the size of the matrix (number of rows/columns)
    const matrixSize = matrix.length;

    // Initialize 'sum' to zero, which will store the final diagonal sum
    let sum = 0;

    // Iterate through each row of the matrix
    for (let row = 0; row < matrixSize; row++) {
        // Add the elements from both the primary and secondary diagonals for the current row
        sum += matrix[row][row] + matrix[row][matrixSize - 1 - row];
    }

    // If the matrix size is odd, subtract the central element once to correct the sum
    if (matrixSize % 2 === 1) {
        // The central element is at the position ['matrixSize' / 2] in both dimensions
        sum -= matrix[matrixSize >> 1][matrixSize >> 1];
    }

    // Return the calculated sum of the diagonal elements
    return sum;
}

```

Time and Space Complexity

Time Complexity

The provided code traverses each row only once, and within each row, it accesses two elements directly by their index, which is an (O(1)) operation. Since there are `n` rows in a square matrix with size (n \times n), the overall time complexity of the code is (O(n)) where (n) is the number of rows (and also the number of columns) in the matrix.

Space Complexity

The code uses a fixed number of variables (`ans`, `n`, `i`, `j`, `row`). It does not depend on the size of the input matrix, therefore the space complexity is (O(1)), that is, constant space complexity.