304. Range Sum Query 2D - Immutable

Medium Design Array Matrix Prefix Sum

Problem Description

We are given a 2D matrix `matrix`, and our task is to design a data structure that can efficiently calculate the sum of the elements within a rectangle defined by its upper-left and lower-right corners. The corner coordinates are given to us as `(row1, col1)` for the upper-left corner and `(row2, col2)` for the lower-right corner. This should be done for multiple queries and the data structure has to be efficient enough to provide the sum in constant time (`O(1)`).

Intuition

The key to solving this problem efficiently lies in preprocessing the matrix in such a way that it allows us to perform queries in `O(1)` time. This is achieved by computing a prefix sum matrix which is essentially a preprocessing step that facilitates quick sum retrieval.

The intuition behind using a prefix sum is that we can calculate the sum of a certain rectangle by combining the sums of different regions which are precomputed. The sum of elements inside the rectangle can be determined by adding and subtracting certain sums at key points that represent the corners of the rectangle.

To compute the prefix sum, we follow these steps:

1. Initialize an auxiliary matrix `s` with dimensions `(m+1)x(n+1)`, where `m` is the number of rows in the input matrix and `n` is the number of columns. We use an extra row and column to handle zero scenarios without special cases.
2. We iterate through each element `(i, j)` of the original matrix and calculate `s[i + 1][j + 1]` as the sum of elements above and to the left including `matrix[i][j]`. This sum is computed by adding the sum up to the previous row (`s[i][j + 1]`), the sum up to the previous column (`s[i + 1][j]`), and then subtracting the sum that is double-counted (`s[i][j]`) because it was included in both previous sums, and finally adding the current element (`matrix[i][j]`).
3. Once the prefix sum matrix is created in the initialization process, the sum region can be computed by using the formula derived from the properties of the prefix sum matrix. The sum of a rectangle from `(row1, col1)` to `(row2, col2)` is computed by summing the cumulative sum at `(row2 + 1, col2 + 1)` and subtracting the areas that extend beyond the target rectangle (cumulative sum at `(row2 + 1, col1)`, `(row1, col2 + 1)`), and adding back the area that was subtracted twice, which is at `(row1, col1)`.

This technique harnesses the power of cumulative sums to facilitate rapid sum queries of rectangular submatrices and is particularly useful when there are multiple queries on the same initial matrix, greatly reducing the time complexity from `O(m*n)` to `O(1)` for each query after an `O(m*n)` preprocessing time.

Solution Approach

The implementation of the solution is straightforward once we understand the concept of a prefix sum matrix. Let's break down the approach and the pattern used in the given solution:

1. Construction of the Prefix Sum Matrix:
   - We start by creating a 2D array `self.s` which will store the prefix sums. Notice that this array has an extra row and column (`m + 1` and `n + 1`), this is done to accommodate sums of submatrices that may include the first row or column without having to handle special cases.
   - We iterate through the original matrix, and for each element at `(i, j)` in `matrix`, we compute the prefix sum using:

     ```
     self.s[i + 1][j + 1] = self.s[i][j + 1] + self.s[i + 1][j] - self.s[i][j] + matrix[i][j]
     ```

     In this formula:
     - `self.s[i][j + 1]` represents the sum of elements above the current element (up to the previous row).
     - `self.s[i + 1][j]` represents the sum of elements to the left of the current element (up to the previous column).
     - `self.s[i][j]` is the area that was added twice (once in above sum and once in left sum) and needs to be subtracted out.
     - `matrix[i][j]` is the value of the current element which needs to be added to complete the sum of the submatrix that ends at `(i, j)`.
2. Sum Region Query:
   - Once the prefix sum matrix is constructed, we can answer any sum region query in `O(1)` time. The function `sumRegion(row1, col1, row2, col2)` computes the sum of the elements inside the rectangle using:

     ```
     return (self.s[row2 + 1][col2 + 1] - self.s[row2 + 1][col1] - self.s[row1][col2 + 1] + self.s[row1][col1])
     ```

     Here's how the terms of the formula correspond to the regions:
     - `self.s[row2 + 1][col2 + 1]` gives the sum of all elements up to `(row2, col2)`.
     - `self.s[row2 + 1][col1]` subtracts out the area to the left of `col1` since it's not part of the desired rectangle.
     - `self.s[row1][col2 + 1]` subtracts out the area above `row1`.
     - `self.s[row1][col1]` adds back the area that was subtracted out twice (once for left and once for top).

Utilizing the prefix sum method provides a powerful way to carry out cumulative sum operations both efficiently and swiftly, thus making it a preferred approach for this type of problem.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach to the described problem. Assume we have the following 2D matrix:

```
matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]
```

1. Construction of the Prefix Sum Matrix:
   - We initiate an auxiliary array `self.s` with an extra row and column. Initially, it will look like this (we fill the extra row and column with 0s):

```
self.s = [
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0]
]
```

- We proceed to calculate the prefix sums for each cell based on the elements of the original matrix. For the first element (at 0, 0 in the original matrix), the sum will just be the element itself:

```
self.s[1][1] = matrix[0][0] = 3

// The matrix self.s would look like this after this step:
self.s = [
  [0, 0, 0, 0, 0, 0],
  [0, 3, 0, 0, 0, 0],
  ...
]
```

- We fill in the rest of self.s using the formula detailed earlier. The fully filled prefix sum matrix would look like this:

```
self.s = [
  [0, 0,  0,  0,  0, 0],
  [0, 3,  3,  4,  8, 10],
  [0, 8, 14, 18, 24, 27],
  [0, 9, 17, 21, 28, 36],
  [0, 13,22, 26, 34, 49],
  [0, 14,23, 30, 38, 58]
]
```

2. Sum Region Query:
   - Let's say we wish to get the sum of elements within the rectangle defined by the corners `(row1, col1) = (2, 1)` and `(row2, col2) = (4, 3)`. Following the formula provided, we calculate this as follows:

```
sum = self.s[4 + 1][3 + 1] - self.s[4 + 1][1] - self.s[2][3 + 1] + self.s[2][1]
    = self.s[5][4] - self.s[5][1] - self.s[2][4] + self.s[2][1]
    = 38 - 14 - 24 + 8
    = 8
```

This gives us the sum of elements in the rectangle defined from `(2, 1)` to `(4, 3)` which includes the values `[2, 0, 1], [1, 0, 1], [0, 3, 0]` from the original matrix.

This example demonstrates the efficiency of the prefix sum matrix in answering sum region queries in constant time after the preprocessing step has been completed.

Python Solution

```python
class NumMatrix:
    def __init__(self, matrix):
        # First check if the matrix is empty to prevent IndexError
        if not matrix or not matrix[0]:
            raise ValueError("Matrix should not be empty")

        # Get the dimensions of the matrix
        self.num_rows, self.num_cols = len(matrix), len(matrix[0])

        # Create a 2D prefix sum array with an extra row and column (for easy calculation)
        self.prefix_sum = [[0] * (self.num_cols + 1) for _ in range(self.num_rows + 1)]

        # Calculate cumulative sum for the matrix and store it in prefix_sum array
        for row in range(self.num_rows):
            for col in range(self.num_cols):
                self.prefix_sum[row + 1][col + 1] = (
                    self.prefix_sum[row][col + 1] +       # current row prefix sum
                    self.prefix_sum[row + 1][col] -       # current column prefix sum
                    self.prefix_sum[row][col] +           # subtract overlapping area
                    matrix[row][col]                      # add current matrix value
                )

    def sumRegion(self, row1, col1, row2, col2):
        # Retrieve the sum of the desired region using the inclusion-exclusion principle
        return (
            self.prefix_sum[row2 + 1][col2 + 1] -       # sum of entire rectangle from (0,0) to (row2, col2)
            self.prefix_sum[row2 + 1][col1] -           # subtract sum above the intended row range
            self.prefix_sum[row1][col2 + 1] +           # subtract sum before the intended column range
            self.prefix_sum[row1][col1]                 # add back the sum of the overlapping rectangle
        )                                               # that was subtracted twice


# How to use the NumMatrix class:
# obj = NumMatrix(matrix)
# sum_of_region = obj.sumRegion(row1, col1, row2, col2)

```

Java Solution

```java
class NumMatrix {
    private int[][] prefixSumMatrix;

    // Constructor initializes the NumMatrix object
    public NumMatrix(int[][] matrix) {
        if (matrix.length == 0 || matrix[0].length == 0) return;
        int rows = matrix.length, cols = matrix[0].length;

        // plus one to handle the border cases without extra condition checks
        prefixSumMatrix = new int[rows + 1][cols + 1];

        // Construct prefix sums matrix
        for (int i = 1; i <= rows; ++i) {
            for (int j = 1; j <= cols; ++j) {
                // Computing the prefix sum for position (i, j)
                prefixSumMatrix[i][j] = prefixSumMatrix[i - 1][j] + prefixSumMatrix[i][j - 1]
                                        - prefixSumMatrix[i - 1][j - 1] + matrix[i - 1][j - 1];
            }
        }
    }

    // Return the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right cc
    public int sumRegion(int row1, int col1, int row2, int col2) {
        // Apply the inclusion-exclusion principle to find the sum of the region
        return prefixSumMatrix[row2 + 1][col2 + 1] - prefixSumMatrix[row2 + 1][col1]
               - prefixSumMatrix[row1][col2 + 1] + prefixSumMatrix[row1][col1];
    }
}

// This is an example usage provided for reference; actual usage may vary
/*
NumMatrix numMatrix = new NumMatrix(matrix);
int param1 = numMatrix.sumRegion(row1, col1, row2, col2);
*/

```

C++ Solution

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

class NumMatrix {
private:
    vector<vector<int>> prefixSumMatrix; // 2D vector to store the prefix sum matrx

public:
    // Constructor that pre-calculates the prefix sum for the input matrix.
    NumMatrix(vector<vector<int>>& matrix) {
        int rowCount = matrix.size(); // Number of rows in the matrix
        int colCount = matrix[0].size(); // Number of columns in the matrix

        // Resize the prefixSumMatrix to accommodate the extra row and column for easier calculations.
        prefixSumMatrix.resize(rowCount + 1, vector<int>(colCount + 1, 0));

        // Calculate the prefix sum for each cell.
        for (int row = 0; row < rowCount; ++row) {
            for (int col = 0; col < colCount; ++col) {
                // Current cell's prefix sum is equal to the sum of:
                // - Prefix sum of cell above
                // - Prefix sum of cell to the left
                // - The current cell's value
                // + Prefix sum of cell diagonally up-left (to avoid double counting)
                prefixSumMatrix[row + 1][col + 1] = prefixSumMatrix[row + 1][col] +
                                                    prefixSumMatrix[row][col + 1] -
                                                    prefixSumMatrix[row][col] +
                                                    matrix[row][col];
            }
        }
    }

    // Function to calculate the sum of elements in the given rectangular region.
    int sumRegion(int row1, int col1, int row2, int col2) {
        // Calculate the region sum using the inclusion-exclusion principle.
        return prefixSumMatrix[row2 + 1][col2 + 1] -  // Entire rectangle sum including the region
               prefixSumMatrix[row2 + 1][col1] -      // Exclude left strip outside of the region
               prefixSumMatrix[row1][col2 + 1] +      // Exclude top strip outside of the region
               prefixSumMatrix[row1][col1];           // Add back the top-left rectangle, as it was excluded twice
    }
};

// Usage example (not part of the class definition):
// NumMatrix* obj = new NumMatrix(matrix);
// int sum = obj->sumRegion(row1, col1, row2, col2);

```

Typescript Solution

```typescript
// Define the prefix sum matrix as a global variable
let prefixSumMatrix: number[][];

// Initialize the prefix sum matrix using the given matrix
function initialize(matrix: number[][]): void {
    const rows = matrix.length;
    const cols = matrix[0].length;

    // Create the prefix sum matrix with an extra row and column (for easier calculations)
    prefixSumMatrix = new Array(rows + 1).fill(0).map(() => new Array(cols + 1).fill(0));

    // Populate the prefix sum matrix using the input matrix
    for (let i = 0; i < rows; ++i) {
        for (let j = 0; j < cols; ++j) {
            // Calculate the current cell's prefix sum value
            prefixSumMatrix[i + 1][j + 1] =
                prefixSumMatrix[i + 1][j] + prefixSumMatrix[i][j + 1] -
                prefixSumMatrix[i][j] + matrix[i][j];
        }
    }
}

// Calculate the sum of the elements in the given rectangular region
function sumRegion(row1: number, col1: number, row2: number, col2: number): number {
    // Return the sum of elements in the region using the inclusion-exclusion principle
    return prefixSumMatrix[row2 + 1][col2 + 1] -
           prefixSumMatrix[row2 + 1][col1] -
           prefixSumMatrix[row1][col2 + 1] +
           prefixSumMatrix[row1][col1];
}

// Example usage:
// Initialize with a given matrix
initialize([
   [3, 0, 1, 4, 2],
   [5, 6, 3, 2, 1],
   [1, 2, 0, 1, 5],
   [4, 1, 0, 1, 7],
   [1, 0, 3, 0, 5]
]);

// Calculate the sum of a region between two coordinates
const result = sumRegion(2, 1, 4, 3); // Result for sum of the region (2,1) to (4,3)

```

Time and Space Complexity

The time complexity for initializing the `NumMatrix` object is `O(m * n)` where `m` is the number of rows and `n` is the number of columns in the input matrix. This is because we have two nested loops each going through `m` rows and `n` columns to compute the sum of each submatrix ending at `(i, j)`.

The time complexity for the `sumRegion` query operation is `O(1)` because it involves a constant number of mathematical operations irrespective of the size of the input matrix or the selected region. It directly uses the precomputed sums from the `self.s` matrix to calculate the sum of the given region.

The space complexity of the code is also `O(m * n)` due to the additional space used to store the precomputed sums in the `self.s` matrix, which has the dimensions of `(m + 1) * (n + 1)` to allow for easier calculation of sums for the `sumRegion` method.