2536. Increment Submatrices by One

Medium Array Matrix Prefix Sum [Leetcode Link]

Problem Description

We start with an `n x n` matrix `mat` filled with zeroes, and we are tasked to perform a series of operations based on a list of queries, each represented by `[row1, col1, row2, col2]`. For each query, we increment by `1` every element in the submatrix with its top left corner at `(row1, col1)` and its bottom right corner at `(row2, col2)`. After performing all queries, we need to return the final resulting matrix.

The main challenge is to perform these updates efficiently since applying each query individually could lead to a time complexity that is too high if the number of queries or the size of the matrix is large.

Intuition

The intuition behind the solution lies in using the concept of `prefix sum` in a two-dimensional space, which is an extension of the prefix sum algorithm used for one-dimensional arrays.

In one-dimensional space, the prefix sum at any point `i` represents the sum of all elements from index `0` to `i`. In two-dimensional space, we extend this concept to both rows and columns. So at any point `(i, j)` in the `mat`, we intend to store the sum of all elements in the submatrix whose top left corner is `(0, 0)` and the bottom right corner is `(i, j)`.

We can achieve this efficiently using a technique called `difference array`. Instead of incrementing all elements within the submatrix specified by the query, we only increment the top left corner element by `1`, and then decrement the elements just outside the lower and right bounds by `-1`. This marks the boundaries of the region that should be increased.

Once the boundary updates for all queries are made, we re-construct the actual matrix by computing prefix sums row-wise and column-wise. These prefix sums account for the boundaries we've marked earlier, and effectively simulate the addition of `1` across the entire submatrix for each query.

By processing the updates this way, we make the operation for each query independent of the size of the submatrix, therefore, improving the efficiency of the solution significantly.

Solution Approach

The solution approach uses the following key steps which are reflected in the provided code:

1. Initialize a matrix `mat` of size `n x n` filled with zeroes.
2. Process each query `[x1, y1, x2, y2]` to update the matrix boundaries:
   - Increment the element at the top left corner `(x1, y1)` of the submatrix by `1` to indicate the start of a region where every element should be incremented.
   - Decrement the element just outside the bottom boundary `(x2 + 1, y1)` and right boundary `(x1, y2 + 1)` to fix the overestimated addition if it's within matrix bounds. This is because when we later calculate the prefix sum, this decrement will cancel out the increments for elements beyond the submatrix' boundaries.
   - If the bottom right corner `(x2 + 1, y2 + 1)` is inside the bounds of the matrix, increment it to counter the extra decrement that occurs when both the right boundary and bottom boundary adjustments overlap.

```
for x1, y1, x2, y2 in queries:
    mat[x1][y1] += 1
    if x2 + 1 < n:
        mat[x2 + 1][y1] -= 1
    if y2 + 1 < n:
        mat[x1][y2 + 1] -= 1
    if x2 + 1 < n and y2 + 1 < n:
        mat[x2 + 1][y2 + 1] += 1
```

3. Construct the final matrix by computing the prefix sum:
   - Accumulate the sums in the `mat` matrix row-wise and column-wise. The current element `mat[i][j]` is updated by accumulating the value from the top `mat[i-1][j]` if it exists, from the left `mat[i][j-1]` if it exists, and then subtracting the top-left diagonal `mat[i-1][j-1]` if it exists to avoid double-counting.

```
for i in range(n):
    for j in range(n):
        if i:
            mat[i][j] += mat[i - 1][j]
        if j:
            mat[i][j] += mat[i][j - 1]
        if i and j:
            mat[i][j] -= mat[i - 1][j - 1]
```

The final `mat` after this step contains the answer.

By following these steps, the code employs a clever bookkeeping strategy to efficiently update the matrix for each query, transforming what would be a potentially quadratic per-query operation into a linear one (with respect to the size of the matrix). The use of boundary marking and prefix sums is the crux of this efficient solution.

Example Walkthrough

Let's consider a `3 x 3` matrix and a list of queries `[[0, 0, 1, 1], [1, 1, 2, 2]]` to illustrate the solution approach.

Initially, the matrix `mat` is filled with zeros:

```
0 0 0
0 0 0
0 0 0
```

Now let's apply the queries:

1. For the first query `[0, 0, 1, 1]`, we increment `mat[0][0]` by 1, decrement `mat[2][0]` and `mat[0][2]` by 1 (which are outside of our matrix bounds, so we ignore in this case), and since `mat[2][2]` is outside our matrix bounds, there are no operations. After this query, `mat` looks like:

```
1 0 0
0 0 0
0 0 0
```

2. The second query `[1, 1, 2, 2]` instructs us to increment `mat[1][1]` by 1, decrement `mat[3][1]` and `mat[1][3]` by 1 (again outside of our matrix bounds), and increment `mat[3][3]` (which is also outside of our bounds). After this query, `mat` looks like:

```
1 0 0
0 1 0
0 0 0
```

Now, we need to construct the final matrix by computing the prefix sum row-wise and column-wise:

1. Starting at `mat[0][0]`, we move right. There are no previous elements, so we keep moving across the first row:

```
1 1 1
0 1 0
0 0 0
```

2. Now we move down to the next row, `mat[1][0]`, adding the value above, so we get `1` in `mat[1][0]`. We do the same across the second row:

```
1 1 1
1 2 1
0 0 0
```

3. Finally, we process the last row in the same way. Start at `mat[2][0]`, add `mat[1][0]` to `mat[2][0]` to get `1` and continue the process for the row:

```
1 1 1
1 2 1
1 1 1
```

The final matrix now accurately reflects the result after applying the given list of queries, according to the described solution approach:

```
1 1 1
1 2 1
1 1 1
```

The example demonstrates how the solution approach efficiently updates the entire matrix using boundary marking and prefix sums, avoiding incrementing every single element within each target submatrix for the queries given.

Python Solution

```python
from typing import List

class Solution:
    def rangeAddQueries(self, n: int, queries: List[List[int]]) -> List[List[int]]:
        # Create a matrix initialized with zeros
        matrix = [[0] * n for _ in range(n)]

        # Apply the query operations using a prefix sum approach
        for x1, y1, x2, y2 in queries:
            # Increment the top-left corner of the range by 1
            matrix[x1][y1] += 1

            # Decrement the positions just outside the range
            if x2 + 1 < n:
                matrix[x2 + 1][y1] -= 1
            if y2 + 1 < n:
                matrix[x1][y2 + 1] -= 1
            if x2 + 1 < n and y2 + 1 < n:
                # This position was decremented twice, so we increment it once
                matrix[x2 + 1][y2 + 1] += 1

        # Convert the operations to actual values by propagating the increments
        for i in range(n):
            for j in range(n):
                # If not in the first row, add value from the cell directly above
                if i > 0:
                    matrix[i][j] += matrix[i - 1][j]
                # If not in the first column, add value from the cell directly to the left
                if j > 0:
                    matrix[i][j] += matrix[i][j - 1]
                # If not in the first row or column, subtract the value from
                # the cell to the top-left to counteract double counting
                if i > 0 and j > 0:
                    matrix[i][j] -= matrix[i - 1][j - 1]

        # The resultant matrix contains the final values after all queries
        return matrix

```

Java Solution

```java
class Solution {
    public int[][] rangeAddQueries(int n, int[][] queries) {
        // Initialize the matrix with size n x n and all values set to 0
        int[][] matrix = new int[n][n];
        // Process each query to increment the corners of the submatrix
        for (int[] query : queries) {
            int startX = query[0], startY = query[1], endX = query[2], endY = query[3];
            matrix[startX][startY]++; // Top-left corner

            // Prevent out of bounds and mark just outside the bottom boundary if within limits
            if (endX + 1 < n) {
                matrix[endX + 1][startY]--;
            }

            // Prevent out of bounds and mark just outside the right boundary if within limits
            if (endY + 1 < n) {
                matrix[startX][endY + 1]--;
            }

            // Adjust for overlap by incrementing the bottom-right corner outside of the submatrix
            if (endX + 1 < n && endY + 1 < n) {
                matrix[endX + 1][endY + 1]++;
            }
        }

        // Cumulative sum logic to update the matrix values to reflect all range queries
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i > 0) {
                    // Add the value from the previous row to accumulate the vertical sums
                    matrix[i][j] += matrix[i - 1][j];
                }
                if (j > 0) {
                    // Add the value from the previous column to accumulate the horizontal sums
                    matrix[i][j] += matrix[i][j - 1];
                }
                if (i > 0 && j > 0) {
                    // Subtract the overlapping value that has been added twice
                    matrix[i][j] -= matrix[i - 1][j - 1];
                }
            }
        }
        // Return the updated matrix after processing all the range add queries
        return matrix;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<vector<int>> rangeAddQueries(int size, vector<vector<int>>& queries) {
        // Initialize the matrix 'resultMatrix' with 'size' rows and columns filled with zeros
        vector<vector<int>> resultMatrix(size, vector<int>(size, 0));

        // Process each range add query
        for (auto& query : queries) {
            // Retrieve the range's top-left and bottom-right coordinates
            int topLeftRow = query[0], topLeftCol = query[1],
                bottomRightRow = query[2], bottomRightCol = query[3];

            // Increment value at the top-left corner of the range
            resultMatrix[topLeftRow][topLeftCol]++;

            // If the range extends beyond the bottom border of the matrix,
            // decrement the value just outside the bottom border of the range
            if (bottomRightRow + 1 < size) {
                resultMatrix[bottomRightRow + 1][topLeftCol]--;
            }

            // If the range extends beyond the right border of the matrix,
            // decrement the value just outside the right border of the range
            if (bottomRightCol + 1 < size) {
                resultMatrix[topLeftRow][bottomRightCol + 1]--;
            }

            // If the range extends beyond both the bottom and right borders of the matrix,
            // increment the value at the coordinate just outside the bottom-right corner of the range (to cancel out the double su
            if (bottomRightRow + 1 < size && bottomRightCol + 1 < size) {
                resultMatrix[bottomRightRow + 1][bottomRightCol + 1]++;
            }
        }

        // Propagate the range additions through the entire matrix
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                // If not at the first row, add value from the cell directly above
                if (i > 0) {
                    resultMatrix[i][j] += resultMatrix[i - 1][j];
                }

                // If not at the first column, add value from the cell directly left
                if (j > 0) {
                    resultMatrix[i][j] += resultMatrix[i][j - 1];
                }

                // If not at the first row or column, subtract the value from the top-left diagonal to avoid double counting
                if (i > 0 && j > 0) {
                    resultMatrix[i][j] -= resultMatrix[i - 1][j - 1];
                }
            }
        }

        // Return the resulting matrix after all range additions have been applied
        return resultMatrix;
    }
};

```

Typescript Solution

```typescript
function rangeAddQueries(size: number, queries: number[][]): number[][] {
    // Initialize the resultMatrix with 'size' rows and columns filled with zeros
    const resultMatrix: number[][] = Array.from({ length: size }, () => new Array(size).fill(0));

    // Process each range add query
    for (const query of queries) {
        // Retrieve the range's top-left and bottom-right coordinates
        const topLeftRow = query[0],
              topLeftCol = query[1],
              bottomRightRow = query[2],
              bottomRightCol = query[3];

        // Increment value at the top-left corner of the range
        resultMatrix[topLeftRow][topLeftCol]++;

        // If the range extends beyond the bottom border of the matrix,
        // decrement the value just outside the bottom border of the range
        if (bottomRightRow + 1 < size) {
            resultMatrix[bottomRightRow + 1][topLeftCol]--;
        }

        // If the range extends beyond the right border of the matrix,
        // decrement the value just outside the right border of the range
        if (bottomRightCol + 1 < size) {
            resultMatrix[topLeftRow][bottomRightCol + 1]--;
        }

        // If the range extends beyond both the bottom and right borders of the matrix,
        // increment the value at the coordinate just outside the bottom-right corner of the range
        if (bottomRightRow + 1 < size && bottomRightCol + 1 < size) {
            resultMatrix[bottomRightRow + 1][bottomRightCol + 1]++;
        }
    }

    // Propagate the range additions through the entire matrix
    for (let i = 0; i < size; i++) {
        for (let j = 0; j < size; j++) {
            // If not at the first row, add value from the cell directly above
            if (i > 0) {
                resultMatrix[i][j] += resultMatrix[i - 1][j];
            }

            // If not at the first column, add value from the cell directly to the left
            if (j > 0) {
                resultMatrix[i][j] += resultMatrix[i][j - 1];
            }

            // If not at the first row or column, subtract the value from the top-left diagonal to avoid double counting
            if (i > 0 && j > 0) {
                resultMatrix[i][j] -= resultMatrix[i - 1][j - 1];
            }
        }
    }

    // Return the resulting matrix after all range additions have been applied
    return resultMatrix;
}

```

Time and Space Complexity

The provided code aims to perform range additions for a matrix of size `n x n` based on a list of queries. Each query consists of coordinates defining the top-left `(x1, y1)` and bottom-right `(x2, y2)` corners of a submatrix where the values should be incremented by 1.

Time Complexity

The time complexity of the code can be analyzed by breaking it down into two parts:

1. Processing the queries:
   - The queries are processed one by one in a loop. For each query, there are constant time updates being made to the matrix `mat`. Consequently, this part has a time complexity of `O(Q)`, where `Q` is the number of queries.
2. Prefix Sum Calculation:
   - After processing the queries, two nested loops are used to calculate the prefix sums over the matrix. Each element in the matrix is visited and updated based on the values of its neighbors. Therefore, this part has a time complexity of `O(n^2)`, since we iterate over all elements of the `n x n` matrix.

Combining both parts, the overall time complexity is `O(Q + n^2)`.

Space Complexity

The space complexity is determined by the memory required for the input and the internal data structures used by the algorithm.

1. The matrix `mat` is the main data structure with a size of `n x n`, so it has a space complexity of `O(n^2)`.
2. No additional significant space is used, so the space complexity remains `O(n^2)` overall.

Hence, the time complexity of the code is `O(Q + n^2)` and the space complexity is `O(n^2)`.