2536. Increment Submatrices by One

Medium Array Matrix Prefix Sum

that is too high if the number of queries or the size of the matrix is large.

prefix sum algorithm used for one-dimensional arrays.

improving the efficiency of the solution significantly.

#### Leetcode Link

We start with an  $n \times n$  matrix mat filled with zeroes, and we are tasked to perform a series of operations based on a list of queries,

**Problem Description** 

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

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

### 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,

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

1. Initialize a matrix mat of size  $n \times n$  filled with zeroes.

o Increment the element at the top left corner (x1, y1) of the submatrix by 1 to indicate the start of a region where every

overestimated addition if it's within matrix bounds. This is because when we later calculate the prefix sum, this decrement

### element should be incremented. $\circ$ Decrement the element just outside the bottom boundary (x2 + 1, y1) and right boundary (x1, y2 + 1) to fix the

**Solution Approach** 

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

2. Process each query [x1, y1, x2, y2] to update the matrix boundaries:

that occurs when both the right boundary and bottom boundary adjustments overlap.

1 for x1, y1, x2, y2 in queries:

3. Construct the final matrix by computing the prefix sum:

mat[i][j] += mat[i - 1][j]

mat[i][j] += mat[i][j - 1]

mat[i][j] = mat[i - 1][j - 1]

use of boundary marking and prefix sums is the crux of this efficient solution.

mat[x2 + 1][y1] = 1

mat[x1][y1] += 1

if  $x^2 + 1 < n$ :

if j:

**Example Walkthrough** 

Now let's apply the queries:

this query, mat looks like:

3 0 0 0

if i and j:

The final mat after this step contains the answer.

if y2 + 1 < n:
mat[x1][y2 + 1] -= 1
if x2 + 1 < n and y2 + 1 < n:
mat[x2 + 1][y2 + 1] += 1</pre>

By following these steps, the code employs a clever bookkeeping strategy to efficiently update the matrix for each query,

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

transforming what would be a potentially quadratic per-query operation into a linear one (with respect to the size of the matrix). The

- 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 j:
- Initially, the matrix mat is filled with zeros:

1 100 2 000 3 000

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:

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

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

1 1 0 0 2 0 1 0 3 0 0 0

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

second row:

1 1 1 1

3 0 0 0

1 1 1 1

3 0 0 0

2 1 2 1

3 1 1 1

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:

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

approach:

1 1 1 1
2 1 2 1
3 1 1 1

avoiding incrementing every single element within each target submatrix for the queries given.

def rangeAddQueries(self, n: int, queries: List[List[int]]) -> List[List[int]]:

# This position was decremented twice, so we increment it once

# If not in the first row or column, subtract the value from

int startX = query[0], startY = query[1], endX = query[2], endY = query[3];

// Cumulative sum logic to update the matrix values to reflect all range queries

// Subtract the overlapping value that has been added twice

// Prevent out of bounds and mark just outside the bottom boundary if within limits

// Prevent out of bounds and mark just outside the right boundary if within limits

// Adjust for overlap by incrementing the bottom-right corner outside of the submatrix

// Add the value from the previous row to accumulate the vertical sums

// Add the value from the previous column to accumulate the horizontal sums

# the cell to the top-left to counteract double counting

# If not in the first column, add value from the cell directly to the left

# Convert the operations to actual values by propagating the increments

# Apply the query operations using a prefix sum approach

# Increment the top-left corner of the range by 1

# Decrement the positions just outside the range

matrix[i][j] += matrix[i][j - 1]

matrix[startX][startY]++; // Top-left corner

matrix[endX + 1][startY]--;

matrix[startX][endY + 1]--;

**if**  $(endX + 1 < n \&\& endY + 1 < n) {$ 

matrix[endX + 1][endY + 1]++;

matrix[i][j] += matrix[i - 1][j];

matrix[i][j] += matrix[i][j - 1];

matrix[i][j] -= matrix[i - 1][j - 1];

// Return the updated matrix after processing all the range add queries

matrix[i][j] -= matrix[i - 1][j - 1]

# The resultant matrix contains the final values after all queries

# Create a matrix initialized with zeros

matrix[x2 + 1][y1] = 1

matrix[x1][y2 + 1] -= 1

matrix[x2 + 1][y2 + 1] += 1

if  $x^2 + 1 < n$  and  $y^2 + 1 < n$ :

 $matrix = [[0] * n for _ in range(n)]$ 

for x1, y1, x2, y2 in queries:

matrix[x1][y1] += 1

if  $x^2 + 1 < n$ :

if  $y^2 + 1 < n$ :

if j > 0:

if (endX + 1 < n) {

if (endY + 1 < n) {

for (int i = 0; i < n; ++i) {

if (i > 0) {

if (j > 0) {

return matrix;

for (int j = 0; j < n; ++j) {

if (i > 0 && j > 0) {

return matrix

if i > 0 and j > 0:

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

The example demonstrates how the solution approach efficiently updates the entire matrix using boundary marking and prefix sums,

Python Solution

1 from typing import List

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

28

29

30

31

32

33

34

35

36

37

38

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

class Solution:

Java Solution

```
46
 47
C++ Solution
  1 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
  9
 10
                 int topLeftRow = query[0], topLeftCol = query[1],
                     bottomRightRow = query[2], bottomRightCol = query[3];
 12
 13
                 // Increment value at the top-left corner of the range
                 resultMatrix[topLeftRow][topLeftCol]++;
 14
 15
 16
                 // If the range extends beyond the bottom border of the matrix,
 17
                 // decrement the value just outside the bottom border of the range
 18
                 if (bottomRightRow + 1 < size) {</pre>
                     resultMatrix[bottomRightRow + 1][topLeftCol]--;
 19
 20
 21
 22
                 // If the range extends beyond the right border of the matrix,
 23
                 // decrement the value just outside the right border of the range
                 if (bottomRightCol + 1 < size) {</pre>
 24
 25
                     resultMatrix[topLeftRow][bottomRightCol + 1]--;
 26
 27
 28
                 // If the range extends beyond both the bottom and right borders of the matrix,
 29
                 // 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) {</pre>
 30
                     resultMatrix[bottomRightRow + 1][bottomRightCol + 1]++;
 31
 32
 33
 34
 35
             // Propagate the range additions through the entire matrix
 36
             for (int i = 0; i < size; ++i) {
 37
                 for (int j = 0; j < size; ++j) {
                     // If not at the first row, add value from the cell directly above
 38
                     if (i > 0) {
 39
                         resultMatrix[i][j] += resultMatrix[i - 1][j];
 40
```

// If not at the first column, add value from the cell directly left

// If not at the first row or column, subtract the value from the top-left diagonal to avoid double counting

resultMatrix[i][j] += resultMatrix[i][j - 1];

resultMatrix[i][j] -= resultMatrix[i - 1][j - 1];

// Return the resulting matrix after all range additions have been applied

// Initialize the resultMatrix with 'size' rows and columns filled with zeros

// Retrieve the range's top-left and bottom-right coordinates

// If the range extends beyond the bottom border of the matrix,

// If the range extends beyond the right border of the matrix,

// decrement the value just outside the right border of the range

// decrement the value just outside the bottom border of the range

// Increment value at the top-left corner of the range

resultMatrix[bottomRightRow + 1][topLeftCol]--;

resultMatrix[topLeftRow][bottomRightCol + 1]--;

const resultMatrix: number[][] = Array.from({ length: size }, () => new Array(size).fill(0));

# Typescript Solution 1 function rangeAddQueries(size: number, queries: number[][]): number[][] {

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

4

5

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

};

if (j > 0) {

return resultMatrix;

// Process each range add query

for (const query of queries) {

const topLeftRow = query[0],

topLeftCol = query[1],

if (bottomRightRow + 1 < size) {</pre>

if (bottomRightCol + 1 < size) {</pre>

bottomRightRow = query[2],

bottomRightCol = query[3];

resultMatrix[topLeftRow][topLeftCol]++;

if (i > 0 && j > 0) {

```
26
 28
             // If the range extends beyond both the bottom and right borders of the matrix,
 29
             // increment the value at the coordinate just outside the bottom-right corner of the range
             if (bottomRightRow + 1 < size && bottomRightCol + 1 < size) {</pre>
 30
                 resultMatrix[bottomRightRow + 1][bottomRightCol + 1]++;
 31
 32
 33
 34
 35
         // Propagate the range additions through the entire matrix
 36
         for (let i = 0; i < size; i++) {
             for (let j = 0; j < size; j++) {</pre>
 37
                 // If not at the first row, add value from the cell directly above
 38
 39
                     resultMatrix[i][j] += resultMatrix[i - 1][j];
 40
 41
 42
 43
                 // If not at the first column, add value from the cell directly to the left
                 if (j > 0) {
 44
                     resultMatrix[i][j] += resultMatrix[i][j - 1];
 45
 46
 47
 48
                 // 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) {
 49
                     resultMatrix[i][j] -= resultMatrix[i - 1][j - 1];
 50
 51
 52
 53
 54
 55
         // Return the resulting matrix after all range additions have been applied
 56
         return resultMatrix;
 57 }
 58
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
```

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

1. Processing the queries:

incremented by 1.

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.
 Prefix Sum Calculation:

- since we iterate over all elements of the  $n \times n$  matrix. Combining both parts, the overall time complexity is  $0(0 + n^2)$ .
- Space Complexity

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

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 0(n^2),

2. No additional significant space is used, so the space complexity remains  $0(n^2)$  overall. Hence, the time complexity of the code is  $0(0 + n^2)$  and the space complexity is  $0(n^2)$ .

1. The matrix mat is the main data structure with a size of  $n \times n$ , so it has a space complexity of  $O(n^2)$ .

Tierroe, the time complexity of the code is oft in 27 and the spa