1314. Matrix Block Sum Medium Array Matrix **Prefix Sum Leetcode Link** 

## **Problem Description** Given a matrix mat with dimensions m x n and an integer k, the task is to compute a new matrix answer of the same dimensions. For

each cell answer[i][j] in the new matrix, we must calculate the sum of all elements within a square block with side length 2k + 1, centered at mat[i][j]. The boundaries of the square are constrained by the limits of the array, meaning we cannot go outside the matrix when calculating the sum. The result for answer[i][j] should only include the sum of valid elements from mat that lie within the specified range and within the matrix.

Intuition The brute-force approach to solving this problem would involve iterating over each cell of the matrix, and for each cell, iteratively summing all the elements in the K-radius square. This would require a large number of redundant computations and would not be

efficient, particularly for large matrices or values of K. To arrive at a more efficient solution, we can use dynamic programming, specifically the concept of 2D prefix sums. The idea here is to preprocess the input matrix to generate a prefix sum matrix where each element pre[i][j] stores the sum of all elements in the

rectangle defined by the upper left corner (0,0) and the lower right corner (i-1, j-1) of the original matrix. With this prefix sum

matrix, we can compute the sum of any submatrix in constant time, using the inclusion-exclusion principle.

The steps to implement this are as follows: 1. Build the prefix sum matrix (pre) where the sum up to each point (i, j) is calculated by adding the current element to the sum of the elements above and to the left, subtracting the sum of the overlapping corner (already counted twice).

2. Define a helper function get(i, j) to safely retrieve the value of pre[i][j], handling boundary cases to stay within the matrix

dimensions.

- 3. Loop through each cell (i, j) in the original matrix, and for each cell, calculate the sum of the K-radius square around it using the prefix sum matrix and the helper function. This involves adding and subtracting values from the four corners of the block surrounding (i, j), constrained by k.
- The result is an efficient algorithm that computes the desired answer matrix with much fewer operations compared to the brute-force method, thus making it scalable to larger matrices and values of k. The overall time complexity is 0(m \* n) for the sum matrix
- construction plus 0(m \* n) for calculating the block sums, and the space complexity is 0(m \* n) for storing the prefix sums. **Solution Approach**

The solution to the problem uses dynamic programming with a 2D prefix sum matrix to optimize the process of computing the sums of submatrices. Let's break down the steps taken in the provided solution code: 1. Initial Prefix Sum Matrix Generation: A 2D prefix sum matrix pre is generated where pre[i][j] will contain the sum of all

(n + 1) to handle the prefix sum calculation conveniently, even for the cells on the topmost row and leftmost column.

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

elements from the top-left corner (0, 0) to the position (i-1, j-1) in the original matrix mat. This matrix pre is of size  $(m + 1) \times (m + 1)$ 

1 ans[i][j] = (

j + k + 1).

**Example Walkthrough** 

Original matrix mat:

The prefix sum is calculated as follows:

j = max(min(n, j), 0)

get(i + k + 1, j + k + 1)

the solution is clean, efficient, and robust for the given problem.

- get(i + k + 1, j - k)

- get(i - k, j + k + 1)

+ get(i - k, j - k)

return pre[i][j]

This equation accounts for the current element and the sums above and to the left, removing the overlapped corner that was added twice.

the boundaries, and ensuring that indices outside the matrix bounds map to the nearest valid index. This function ensures that when we perform the sum calculation, we do not go outside the matrix. 1 def get(i, j): i = max(min(m, i), 0)

3. Computing the Block Sums: We then iterate over each cell (i, j) in the matrix mat. The sum for each answer[i][j] is computed

using the get function to obtain values from the prefix sum matrix corresponding to the corners of the K-radius block:

2. Safe Prefix Sum Retrieval Function (get): The get function is defined to safely retrieve values from the pre matrix, taking care of

The matrixBlockSum function ultimately returns the computed ans matrix, which represents the sum of elements within k distance from each element in the original mat.

The code structure provided makes use of a 2D prefix sum matrix and carefully handles edge cases with the get function, ensuring

This step applies the inclusion-exclusion principle to subtract the sums outside the K-radius block by considering the bottom

right corner (i + k + 1, j + k + 1), the top right (i + k + 1, j - k), top left (i - k, j - k), and bottom left corners (i - k, j - k)

```
1 | 1 | 2 | 3 |
2 | 4 | 5 | 6 |
3 | 7 | 8 | 9 |
 1. Initial Prefix Sum Matrix Generation: We first build an enlarged prefix sum matrix pre of size 4×4 to include a buffer for handling
```

Let's use a simple  $3\times3$  matrix as an example to illustrate how the solution approach works, with k=1:

the edges. The construction would look like this after using the described formula: Initial pre matrix filled with zeros (for illustration):

# 4 | 0 | 12 | 27 | 45 |

1 ans[1][1] = (

9 ans[1][1] = 28

1 ans[0][0] = (

9 ans[0][0] = 12

24 | 39 | 28

**Python Solution** 

class Solution:

8

9

10

11

12

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

2

3

6

8

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

42

43

44

45

46

Java Solution

1 class Solution {

/\*\*

\*/

private int numRows;

private int numCols;

from typing import List

get(1 + 1 + 1, 1 + 1 + 1)

get(0 + 1 + 1, 0 + 1 + 1)

- get(0 + 1 + 1, 0 - 1)

- get(0 - 1, 0 + 1 + 1)

+ get(0 - 1, 0 - 1)

8 ans[0][0] = 12 - 0 - 0 + 0

- get(1 + 1 + 1, 1 - 1)

+ get(1 - 1, 1 - 1)

8 ans [1][1] = 45 - 12 - 6 + 1

- get(1 - 1, 1 + 1 + 1)

After calculating the prefix sums:

do not access indices outside the bounds of the matrix.

7 ans[1][1] = pre[3][3] - pre[3][1] - pre[1][3] + pre[1][1]

7 ans[0][0] = pre[2][2] - pre[2][0] - pre[0][2] + pre[0][0]

areas around each cell, constrained by k and the bounds of the matrix.

# Get the dimensions of the matrix

# Populate the prefix sum matrix

def get\_cumulative\_sum(i, j):

return prefix\_sum[i][j]

for row in range(num\_rows):

i = max(min(num\_rows, i), 0)

j = max(min(num\_cols, j), 0)

# Initialize the answer matrix with zeros

\* @param mat The original matrix of integers

// Populating the prefix sum matrix

for (int i = 1; i <= numRows; ++i) {</pre>

// Create a matrix to hold answers

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

public int[][] matrixBlockSum(int[][] mat, int k) {

prefixSum = new int[numRows + 1][numCols + 1];

for (int j = 1; j <= numCols; ++j) {</pre>

int[][] answer = new int[numRows][numCols];

// Calculating the block sum for each cell

for (int j = 0; j < numCols; ++j) {</pre>

\* @param i The row index in the prefix sum matrix

\* @param j The column index in the prefix sum matrix

\* @return The prefix sum value of the cell (i, j) bounded properly

answer = [[0] \* num\_cols for \_ in range(num\_rows)]

# Compute the block sum for each element in the matrix

private int[][] prefixSum; // 2D array to hold the prefix sum of matrix

\* @param k The given distance around the element to calculate sum

numRows = mat.length; // number of rows in the input matrix

numCols = mat[0].length; // number of columns in the input matrix

// Each cell is the sum of values above it, to its left,

// Calculate the sum of the block using prefix sum

\* Get the prefix sum of the cell (i, j) ensuring the indices are within bounds.

\* @return The matrix representing the block sum of each element

\* Calculates the block sum of each element in the matrix surrounded by 'k' distance.

// Define a new matrix with additional row and column to easily compute prefix sum

for row in range(1, num\_rows + 1):

num\_rows, num\_cols = len(mat), len(mat[0])

mat[row - 1][col - 1]

corresponding to mat [1] [1] which is 5, the calculation using the corners would be:

Which is the sum of the 2×2 block that is within k distance from the top-left element 1.

0

You can see that pre[3][3] has a value of 45, which is the sum of all elements of mat. 2. Safe Prefix Sum Retrieval Function (get): The get function is used to safely access elements in the pre matrix, ensuring that we

3. Computing the Block Sums: We compute the sums for each cell in ans considering k=1. For the central element ans [1] [1]

The get function ensures we don't index into pre at negative indices or beyond its size by clamping the values appropriately.

The complete ans matrix would be computed similarly, for example, the top-left corner ans [0] [0] would be computed as:

```
Final matrix ans after computing sums for all other cells:
     12 | 21 | 16
    27 | 45 | 33
```

This walk-through demonstrates the computational steps of the algorithm using the example matrix. Notice how the sums reflect the

13 for col in range(1, num\_cols + 1): 14 prefix\_sum[row][col] = ( 15 prefix\_sum[row - 1][col] + 16 prefix\_sum[row][col - 1] -17  $prefix_sum[row - 1][col - 1] +$ 

# Helper function to get the cumulative sum up to (i, j) in the prefix sum matrix

# Boundary check to ensure indices fall within the valid range

def matrixBlockSum(self, mat: List[List[int]], k: int) -> List[List[int]]:

# Initialize a prefix sum matrix with one extra row and column

 $prefix_sum = [[0] * (num_cols + 1) for _ in range(num_rows + 1)]$ 

```
33
                for col in range(num_cols):
34
                    answer[row][col] = (
35
                        get_cumulative_sum(row + k + 1, col + k + 1) -
36
                        get_cumulative_sum(row + k + 1, col - k) -
37
                        get_cumulative_sum(row - k, col + k + 1) +
38
                        get_cumulative_sum(row - k, col - k)
39
40
            # Return the answer matrix
41
42
            return answer
43
```

// minus the cell diagonally up and left to remove the double count, plus the current cell

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

answer[i][j] = getSum(i + k + 1, j + k + 1) - getSum(i + k + 1, j - k) - getSum(i - k, j + k + 1) + getSum(i - k, j

### 36 37 38 return answer; 39 40 41 /\*\*

```
47
         private int getSum(int i, int j) {
             // Bound the indices within the limits of the prefixSum matrix
 48
 49
             i = Math.max(Math.min(numRows, i), 0);
 50
             j = Math.max(Math.min(numCols, j), 0);
 51
             return prefixSum[i][j];
 52
 53
 54
C++ Solution
  1 class Solution {
  2 public:
         vector<vector<int>> matrixBlockSum(vector<vector<int>>& mat, int k) {
             int rows = mat.size(), cols = mat[0].size();
             vector<vector<int>> prefixSum(rows + 1, vector<int>(cols + 1));
  6
             // Compute 2D prefix sum
             for (int i = 1; i <= rows; ++i) {
                 for (int j = 1; j <= cols; ++j) {
  9
                     prefixSum[i][j] = prefixSum[i - 1][j]
 10
 11
                                     + prefixSum[i][j - 1]
 12
                                     - prefixSum[i - 1][j - 1]
                                     + mat[i - 1][j - 1];
 13
 14
 15
 16
 17
             vector<vector<int>> answer(rows, vector<int>(cols));
 18
 19
             // Iterate through all the cells of the matrix
             for (int i = 0; i < rows; ++i) {
 20
                 for (int j = 0; j < cols; ++j) {
 21
 22
                     // Calculate block sum using prefixSum
 23
                     answer[i][j] = getBlockSum(i + k + 1, j + k + 1, rows, cols, prefixSum)
 24
                                  - getBlockSum(i + k + 1, j - k, rows, cols, prefixSum)
 25
                                  - getBlockSum(i - k, j + k + 1, rows, cols, prefixSum)
                                  + getBlockSum(i - k, j - k, rows, cols, prefixSum);
 26
 27
 28
 29
 30
             return answer;
 31
 32
```

int getBlockSum(int row, int col, int maxRow, int maxCol, vector<vector<int>>& prefixSum) {

const prefixSum: Matrix = Array.from({ length: rows + 1 }, () => Array(cols + 1).fill(0));

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

// Clamp row and column indices to the valid range

// Return the value at the bounded prefix sum location

row = max(min(maxRow, row), 0);

col = max(min(maxCol, col), 0);

return prefixSum[row][col];

// Compute the 2D prefix sum

for (let i = 1; i <= rows; ++i) {

for (let j = 1; j <= cols; ++j) {

### // Define the matrix type as an array of arrays of numbers 2 type Matrix = number[][]; // Calculate the 2D prefix sum matrix function buildPrefixSumMatrix(mat: Matrix): Matrix { const rows = mat.length, cols = mat[0].length;

8

10

11

12

Typescript Solution

33

34

35

36

37

38

39

40

42

41 };

```
- prefixSum[i - 1][j - 1] + mat[i - 1][j - 1];
 13
 14
 15
 16
 17
         return prefixSum;
 18 }
 19
    // Given the row and column indices, return the prefix sum within the bounded indices
     function getBlockSum(row: number, col: number, maxRow: number, maxCol: number, prefixSum: Matrix): number {
 22
         // Clamp row and column indices to the valid range
 23
         row = Math.max(Math.min(maxRow, row), 0);
 24
         col = Math.max(Math.min(maxCol, col), 0);
 25
 26
         // Return the value at the bounded prefix sum location
 27
         return prefixSum[row][col];
 28 }
 29
    // Computes the matrix block sum given the original matrix and the value k
     function matrixBlockSum(mat: Matrix, k: number): Matrix {
         const rows = mat.length, cols = mat[0].length;
 32
 33
         const prefixSum = buildPrefixSumMatrix(mat);
 34
         const answer: Matrix = Array.from({ length: rows }, () => Array(cols).fill(0));
 35
 36
         // Iterate through all the cells of the matrix
 37
         for (let i = 0; i < rows; ++i) {
 38
             for (let j = 0; j < cols; ++j) {
 39
                 // Calculate block sum using prefixSum
 40
                 answer[i][j] = getBlockSum(i + k + 1, j + k + 1, rows, cols, prefixSum)
 41
                             - getBlockSum(i + k + 1, j - k, rows, cols, prefixSum)
 42
                             - getBlockSum(i - k, j + k + 1, rows, cols, prefixSum)
 43
                             + getBlockSum(i - k, j - k, rows, cols, prefixSum);
 44
 45
 46
 47
         return answer;
 48 }
 49
Time and Space Complexity
Time Complexity:
The time complexity of the given code can be analyzed in two major parts: the construction of the prefix sum matrix pre and the
computation of the answer matrix ans.
 1. Prefix Sum Matrix Computation: Constructing the prefix sum matrix requires iterating through each element of the original m x
```

# n matrix mat, and for each element, a constant amount of work is performed (adding and subtracting values). Therefore, the time

complexity for this part is 0(m \* n).

The space complexity can be analyzed as follows:

### 2. Answer Matrix Computation: The computation of the answer matrix also involves iterating through all elements in the m x n matrix and performing a constant amount of work for each to calculate the sum of the block using the prefix sum matrix. Hence, the time complexity for this part is also 0(m \* n).

- Combining both parts, the overall time complexity remains 0(m \* n) because both parts are sequentially executed and have the same order of complexity. **Space Complexity:**
- 1. Prefix Sum Matrix: An additional m+1 x n+1 matrix pre is used for the prefix sums, which gives us a space complexity of 0((m + 1) \* (n + 1). 2. Answer Matrix: The answer matrix ans is of the same size as the input matrix mat, i.e., m x n. Since it is used to store the final
- consider it and, therefore, 0(m \* n). When combined, the term 0((m + 1) \* (n + 1)) is dominant. However, since the +1 is a constant and does not change the order of growth, the overall space complexity simplifies to 0(m \* n).

output, it might not be counted towards additional space in some analysis contexts. But for a comprehensive analysis, we