1738. Find Kth Largest XOR Coordinate Value Bit Manipulation Array Divide and Conquer **Quickselect** Heap (Priority Queue) Medium Prefix Sum

```
Problem Description
```

certain coordinates, with the value being defined as the XOR (exclusive OR) of all the elements of the submatrix defined by the corner (0, 0) and the coordinate (a, b). To clarify, for each coordinate (a, b), we consider the rectangle from the top-left corner (0, 0) to the coordinate (a, b) and

In this problem, we are given a matrix of non-negative integers with m rows and n columns. We need to calculate the value of

Matrix

Our goal is to find the k-th largest such XOR value from all possible coordinates.

Arriving at the solution for this problem involves understanding how XOR operates and using properties of XOR to build a dynamic

Intuition

compute the XOR of all the elements within that rectangle.

which corresponds to the k-th largest value.

cumulative XOR using the formula:

solution. The XOR operation has a key property of reversibility, which means that if $a \land b = c$, then $a \land c = b$ and $b \land c = a$. Knowing this, we can compute the cumulative XOR in a dynamic fashion as we traverse the matrix. For each cell (i, j), we can

determine its XOR value based on previously computed values in the matrix: the XOR of the rectangle from (0, 0) to (i, j) is the XOR of the rectangle from (0, 0) to (i-1, j), the rectangle from (0, 0) to (i, j-1), the overlapping rectangle ending at (i-1, j-1) (since it's included twice, it cancels out using the XOR reversibility), and the current cell value matrix[i][j].

Therefore, for any cell (i, j), we can calculate its cumulative XOR as s[i][j] = s[i-1][j] ^ s[i][j-1] ^ s[i-1][j-1] ^ matrix[i][j]. This formula helps us determine the cumulative XOR efficiently. After computing the XOR value for all possible coordinates, we add them to a list. Once we have the XOR values for all coordinates, we want the k-th largest value. Python's heapq.nlargest() function can be extremely helpful here. It allows us to quickly obtain the k largest elements from a list, and we return the last of these elements,

In the given Python code, the solution follows these steps using dynamic programming and a priority queue (heap): Initialize a 2D list s of size $(m+1) \times (n+1)$ with zeros. This list will store the cumulative XOR values where s[i][j]corresponds to the XOR value from the top-left corner (0, 0) to the coordinate (i-1, j-1). Create an empty list ans which will store the XOR of all coordinates of the given matrix.

Iterate through each cell (i, j) of the given 2D matrix starting from the top-left corner. For each cell, calculate the

matrix.

matrix = [

[1, 2, 3],

[4, 5, 6]

Example Walkthrough

Solution Approach

 $s[i + 1][j + 1] = s[i + 1][j] ^ s[i][j + 1] ^ s[i][j] ^ matrix[i][j]$

After calculating the cumulative XOR for the cell (i, j), append the result to the ans list.

This formula uses the concept of inclusion-exclusion to avoid double-counting the XOR of any region. Here, s[i + 1][j + 1] includes the value of the cell itself (matrix[i][j]), the XOR of the rectangle above it (s[i + 1][j]), the XOR of the rectangle

to the left (s[i][j+1]), and excludes the XOR of the overlapping rectangle from the top-left to (i-1, j-1) (s[i][j]).

- Once all cells have been processed, we have a complete list of XOR values for all coordinates. Now, we need to find the k-th largest value. The nlargest method from Python's heapq library can efficiently accomplish this by return a list of the k
- largest elements from ans. Here's the code line that employs it: return nlargest(k, ans)[-1]

This code snippets returns the last element from the list returned by nlargest, which is the k-th largest XOR value from the

The time complexity for computing the cumulative XOR is 0(m*n) because we iterate through each cell once, and the time

complexity for finding the k-th largest element using nlargest is 0(n*log(k)). Hence, the total time complexity of this approach is dominated by the larger of the two, which is typically 0(m*n) assuming k is relatively small compared to m*n.

Now let's walk through the solution approach: We initialize a 2D list s with dimensions $(m+1) \times (n+1)$, which translates to a 3×4 list filled with zeros. This will be used to store cumulative XOR values:

Let's consider a matrix with m = 2 rows and n = 3 columns, and let's find the 2nd largest XOR value. The matrix looks like this:

We create an empty list ans to store the XOR values of all coordinates of the given matrix. We iterate through each cell (i, j) of the matrix. On the first iteration (i, j) = (0, 0), we calculate the cumulative XOR

as follows:

s[1][1] = 1

s = [

 $s[1][1] = 0 ^ 0 ^ 0 ^ 1$

up to each cell (i, j):

[0, 0, 0, 0],

[0, 1, 3, 0],

Solution Implementation

xor_values = []

for row in range(num rows):

for col in range(num columns):

matrix[row][col]

return nlargest(k, xor_values)[-1]

prefix xor[row][col] ^

Python

Java

C++

public:

class Solution {

class Solution:

[0, 5, 7, 6]

 $s[1][1] = s[1][0] ^ s[0][1] ^ s[0][0] ^ matrix[0][0]$

we return the last element, which is 6. Thus, the 2nd largest XOR value is 6.

from heapq import nlargest # We'll use nlargest function from the heapq module

Initialize a 2D list for storing exclusive or (XOR) prefix sums

Compute the XOR value for each cell and store it in prefix_xor

Add the result to the list of XOR values

and returning the last element in the resulting list

int[][] prefixXor = new int[rows + 1][cols + 1];

List<Integer> xorValues = new ArrayList<>();

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

// Sort the XOR values in ascending order

return xorValues.get(xorValues.size() - k);

int kthLargestValue(vector<vector<int>>& matrix, int k) {

int rows = matrix.size(), cols = matrix[0].size();

// Create a 2D vector to store the xor values

// Get the number of rows and columns in the matrix

vector<vector<int>> prefixXor(rows + 1, vector<int>(cols + 1));

// Add the computed XOR value to the list of XOR values

from heapq import nlargest # We'll use nlargest function from the heapq module

Initialize a 2D list for storing exclusive or (XOR) prefix sums

Compute the XOR value for each cell and store it in prefix_xor

XOR of the current value with its prefix sums

Add the result to the list of XOR values

and returning the last element in the resulting list

xor_values.append(prefix_xor[row + 1][col + 1])

Get the kth largest XOR value by using the nlargest function

 \circ The code initializes an auxiliary matrix s with dimensions m + 1 by n + 1.

prefix xor = [[0] * (num columns + 1) for in range(num_rows + 1)]

def kthLargestValue(self, matrix: List[List[int]], k: int) -> int:

num_rows, num_columns = len(matrix), len(matrix[0])

prefix xor[row + 1][col + 1] = (

prefix xor[row][col] ^

prefix xor[row + 1][col] ^

prefix xor[row][col + 1] ^

This list will hold all the XOR values in the matrix

Calculate the number of rows and columns

// The k-th largest value is at the index of (total number of values - k) after sorting

xorValues.push(prefixXor[i + 1][j + 1]);

// Sort the XOR values in ascending order

return xorValues[xorValues.length - k];

xorValues.sort((a, b) => a - b);

xor_values = []

for row in range(num rows):

for col in range(num columns):

matrix[row][col]

return nlargest(k, xor_values)[-1]

Initialization of the Prefix XOR Matrix (s):

Calculation of Prefix XOR values:

for each of the m * n cells.

class Solution:

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

Collections.sort(xorValues);

xor_values.append(prefix_xor[row + 1][col + 1])

Get the kth largest XOR value by using the nlargest function

// Initialize the prefix XOR matrix with one extra row and column

// This list will store all the unique XOR values from the matrix

// Add the current XOR value to the list

xorValues.add(prefixXor[i + 1][j + 1]);

// Calculating prefix XOR matrix and storing XOR values of submatrices

// Calculate the prefix XOR value for the current submatrix

// Return the kth largest value by indexing from the end of the sorted list

prefix xor = [[0] * (num columns + 1) for in range(num_rows + 1)]

def kthLargestValue(self, matrix: List[List[int]], k: int) -> int:

num_rows, num_columns = len(matrix), len(matrix[0])

This list will hold all the XOR values in the matrix

Calculate the number of rows and columns

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]

We append the result to the ans list, which now looks like: ans = [1].

We continue the process for the rest of the cells. After processing all cells, the s matrix is filled with cumulative XOR values

```
And the ans list filled with the XOR values of each coordinate is: ans = [1, 3, 0, 5, 7, 6].
 Finally, to find the 2nd largest value, we use the nlargest method from Python's heapq library and return the second item of
 the list:
return nlargest(2, ans)[-1]
```

When we apply the final step, nlargest yields the list [7, 6] (since 7 and 6 are the two largest numbers from the list ans), and

XOR of the current value with its prefix sums prefix xor[row + 1][col + 1] = (prefix xor[row + 1][col] ^ prefix xor[row][col + 1] ^

```
class Solution {
   public int kthLargestValue(int[][] matrix, int k) {
       // Obtain the dimensions of the input matrix
       int rows = matrix.length, cols = matrix[0].length;
```

 $prefixXor[i + 1][j + 1] = prefixXor[i][j + 1] ^ prefixXor[i + 1][j] ^ prefixXor[i][j] ^ matrix[i][j];$

```
// Vector to store the xor of all elements in the matrix
        vector<int> xorValues;
        // Calculate the prefix xor values for each cell in the matrix
        for (int i = 0; i < rows; ++i) {
            for (int i = 0; i < cols; ++i) {
                // Compute the xor value for the current cell by using the previously calculated values
                prefixXor[i + 1][i + 1] = prefixXor[i + 1][i] ^ prefixXor[i][j + 1] ^ prefixXor[i][j] ^ matrix[i][j];
                // Add the computed xor value to the list of xor values
                xorValues.push_back(prefixXor[i + 1][j + 1]);
        // Sort the xor values in ascending order
        sort(xorValues.begin(), xorValues.end());
        // The kth largest value is at index (size - k) after sorting
        return xorValues[xorValues.size() - k];
};
TypeScript
function kthLargestValue(matrix: number[][], k: number): number {
    // Get the number of rows and columns in the matrix
    const rows = matrix.length;
    const cols = matrix[0].length;
    // Create a 2D array to store the prefix XOR values for each cell
    const prefixXor: number[][] = Array.from({ length: rows + 1 }, () => Array(cols + 1).fill(0));
    // Array to store the XOR of all elements in the matrix
    const xorValues: number[] = [];
    // Calculate the prefix XOR values for each cell in the matrix
    for (let i = 0; i < rows; i++) {
        for (let i = 0; i < cols; i++) {
            // Compute the XOR value for the current cell using the previously calculated values
            prefixXor[i + 1][i + 1] =
                prefixXor[i + 1][i] ^ prefixXor[i][i + 1] ^ prefixXor[i][j] ^ matrix[i][j];
```

```
Time and Space Complexity
Time Complexity
```

The time complexity of the given code can be evaluated by looking at each operation performed:

 \circ There are two nested loops that iterate over each cell of the matrix which runs m * n times.

• After calculating the XOR for a cell, the result is appended to the ans list. This operation takes constant time.

 \circ The function nlargest(k, ans) is used to find the kth largest element and operates on the ans list of size m * n.

So we can express this part of the time complexity as 0(m * n). 3. Finding the kth largest value with heapq.nlargest method:

nlargest. Hence, in our case, it becomes 0(m * n * log(k)).

largest value. Since 0(m * n) is subsumed by 0(m * n * log(k)), the overall time complexity simplifies to: 0(m * n * log(k))

 \circ The nlargest function has a time complexity of O(N * log(k)) where N is the number of elements in the list and k is the argument to

So when combined, the overall time complexity is 0(m * n) from the nested loops plus $0(m * n * \log(k))$ from finding the kth

Within the inner loop, there is a calculation that takes constant time, which performs the XOR operations to fill in the s matrix. This is done

For space complexity analysis, we consider the additional space used by the algorithm excluding input and output storage: **Space for the s Matrix:** ○ The code creates an auxiliary matrix s of size (m + 1) * (n + 1), which takes 0((m + 1) * (n + 1)) or simplifying it 0(m * n) space.

```
    A list ans is used to store XOR results which will have at most m * n elements.

    Hence the space taken by ans is 0(m * n).
```

Space for the List ans:

Space Complexity

0(m * n)

So combining these, the space complexity of the algorithm is the sum of the space needed for s and ans, which is 0(m * n) + 0(m * n), which simplifies to:

Both time and space complexities are proposed considering the list List and integer int types are imported from Python's typing module as is customary in type-hinted Python code.