1738. Find Kth Largest XOR Coordinate Value

Problem Description

In this problem, we are given a matrix of non-negative integers with `n` rows and `n` columns. We need to calculate the value of 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 compute the XOR of all the elements within that rectangle.

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

Intuition

Arriving at the solution for this problem involves understanding how XOR operates and using properties of XOR to build a dynamic solution. The XOR operation has a key property of reversibility, which means that if `a ^ b = c`, then `a ^ c = b` and `b ^ 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, which corresponds to the k-th largest value.

Solution Approach

In the given Python code, the solution follows these steps using dynamic programming and a priority queue (heap):

1. Initialize a 2D list `s` of size `(n+1) x (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)`.
2. Create an empty list `ans` which will store the XOR of all coordinates of the given `matrix`.
3. Iterate through each cell `(i, j)` of the given 2D `matrix` starting from the top-left corner. For each cell, calculate the cumulative XOR using the formula:

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

   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]`).
4. After calculating the cumulative XOR for the cell `(i, j)`, append the result to the `ans` list.
5. 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 matrix.

The time complexity for computing the cumulative XOR is `O(mn)` because we iterate through each cell once, and the time complexity for finding the k-th largest element using `nlargest` is `O(n*log(k))`. Hence, the total time complexity of this approach is dominated by the larger of the two, which is typically `O(mn)` assuming k is relatively small compared to `mn`.

Example Walkthrough

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:

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

Now let's walk through the solution approach:

1. We initialize a 2D list `s` with dimensions `(m+1) x (n+1)`, which translates to a 3×4 list filled with zeros. This will be used to store cumulative XOR values:

   ```
   s = [
     [0, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 0]
   ]
   ```

2. We create an empty list `ans` to store the XOR values of all coordinates of the given `matrix`.
3. 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] = s[1][0] ^ s[0][1] ^ s[0][0] ^ matrix[0][0]
   s[1][1] = 0 ^ 0 ^ 0 ^ 1
   s[1][1] = 1
   ```

   We append the result to the `ans` list, which now looks like: `ans = [1]`.
4. We continue the process for the rest of the cells. After processing all cells, the `s` matrix is filled with cumulative XOR values up to each cell `(i, j)`:

   ```
   s = [
     [0, 0, 0, 0],
     [0, 1, 3, 0],
     [0, 5, 7, 6]
   ]
   ```

   And the `ans` list filled with the XOR values of each coordinate is: `ans = [1, 3, 0, 5, 7, 6]`.
5. 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 we return the last element, which is `6`. Thus, the 2nd largest XOR value is `6`.

Python Solution

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

class Solution:
    def kthLargestValue(self, matrix: List[List[int]], k: int) -> int:
        # Calculate the number of rows and columns
        num_rows, num_columns = len(matrix), len(matrix[0])

        # Initialize a 2D list for storing exclusive or (XOR) prefix sums
        prefix_xor = [[0] * (num_columns + 1) for _ in range(num_rows + 1)]
        # This list will hold all the XOR values in the matrix
        xor_values = []

        # Compute the XOR value for each cell and store it in prefix_xor
        for row in range(num_rows):
            for col in range(num_columns):
                # 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] ^
                    prefix_xor[row][col] ^
                    matrix[row][col]
                )
                # Add the result to the list of XOR values
                xor_values.append(prefix_xor[row + 1][col + 1])

        # Get the kth largest XOR value by using the nlargest function
        # and returning the last element in the resulting list
        return nlargest(k, xor_values)[-1]
```

Java Solution

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

        // Initialize the prefix XOR matrix with one extra row and column
        int[][] prefixXor = new int[rows + 1][cols + 1];

        // This list will store all the unique XOR values from the matrix
        List<Integer> xorValues = new ArrayList<>();

        // Calculating prefix XOR matrix and storing XOR values of submatrices
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                // Calculate the prefix XOR value for the current submatrix
                prefixXor[i + 1][j + 1] = prefixXor[i][j + 1] ^ prefixXor[i + 1][j] ^ prefixXor[i][j] ^ matrix[i][j];

                // Add the current XOR value to the list
                xorValues.add(prefixXor[i + 1][j + 1]);
            }
        }

        // Sort the XOR values in ascending order
        Collections.sort(xorValues);

        // Return the kth largest value by indexing from the end of the sorted list
        return xorValues.get(xorValues.size() - k);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int kthLargestValue(vector<vector<int>>& matrix, int k) {
        // Get the number of rows and columns in the matrix
        int rows = matrix.size(), cols = matrix[0].size();
        // Create a 2D vector to store the xor values
        vector<vector<int>> prefixXor(rows + 1, vector<int>(cols + 1));
        // 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 j = 0; j < cols; ++j) {
                // Compute the xor value for the current cell by using the previously calculated values
                prefixXor[i + 1][j + 1] = prefixXor[i + 1][j] ^ 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 Solution

```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 j = 0; j < cols; ++j) {
            // Compute the XOR value for the current cell using the previously calculated values
            prefixXor[i + 1][j + 1] =
                prefixXor[i + 1][j] ^ prefixXor[i][j + 1] ^ prefixXor[i][j] ^ matrix[i][j];
            // Add the computed XOR value to the list of XOR values
            xorValues.push(prefixXor[i + 1][j + 1]);
        }
    }

    // Sort the XOR values in ascending order
    xorValues.sort((a, b) => a - b);
    // The k-th largest value is at the index of (total number of values - k) after sorting
    return xorValues[xorValues.length - k];
}
```

Time and Space Complexity

Time Complexity

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

1. **Initialization of the Prefix XOR Matrix (`s`):**
   - The code initializes an auxiliary matrix `s` with dimensions `m + 1` by `n + 1`.
2. **Calculation of Prefix XOR values:**
   - There are two nested loops that iterate over each cell of the `matrix` which runs `m * n` times.
   - 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 each of the `m * n` cells.
   - After calculating the XOR for a cell, the result is appended to the `ans` list. This operation takes constant time.

So we can express this part of the time complexity as `O(m * n)`.

3. **Finding the `k`th largest value with `heapq.nlargest` method:**
   - The function `nlargest(k, ans)` is used to find the `k`th largest element and operates on the `ans` list of size `m * n`.
   - 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 `nlargest`. Hence, in our case, it becomes `O(m * n * log(k))`.

So when combined, the overall time complexity is `O(m * n)` from the nested loops plus `O(m * n * log(k))` from finding the `k`th largest value. Since `O(m * n)` is subsumed by `O(m * n * log(k))`, the overall time complexity simplifies to:

`O(m * n * log(k))`

Space Complexity

For space complexity analysis, we consider the additional space used by the algorithm excluding input and output storage:

1. **Space for the `s` Matrix:**
   - The code creates an auxiliary matrix `s` of size `(m + 1) * (n + 1)`, which takes `O((m + 1) * (n + 1))` or simplifying it `O(m * n)` space.
2. **Space for the List `ans`:**
   - A list `ans` is used to store XOR results which will have at most `m * n` elements.
   - Hence the space taken by `ans` is `O(m * n)`.

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

`O(m * n)`

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.