3393. Count Paths With the Given XOR Value

How is memoization helpful here?

The concept of **memoization** might seem a little abstract at first, but in the context of your DFS solution, it plays a crucial role in optimizing the algorithm. Let's explore **how memoization helps** and why it's necessary here.

What is Memoization?

Memoization is a technique to optimize recursive algorithms by storing the results of expensive function calls and reusing them when the same inputs occur again, instead of recalculating them.

In the context of your problem, memoization is used to store the results of recursive calls that have already been computed. When the algorithm encounters the same state (the same position `(i, j)` and the same XOR value `xor_sum`), it can return the cached result instead of recalculating it, which **saves time** and avoids redundant computations.

Why Memoization is Helpful in this DFS Approach?

In the code, you're using DFS to explore all possible paths from the top-left corner `(0, 0)` to the bottom-right corner `(m-1, n-1)` of the grid. As you explore, you're calculating the XOR sum along each path.

Without Memoization (naive approach):

Without memoization, the DFS would perform redundant calculations every time it visits the same grid cell `(i, j)` with the same XOR value. In the worst case, if there are many paths that pass through the same cell with the same XOR sum, you would calculate the same values repeatedly, leading to an **exponential time complexity**.

For example, if you have a state `(i, j, xor_sum)` and you encounter it from two different directions (say, coming from the left or from above), you'd compute the same result twice — which is inefficient.

With Memoization:

Memoization solves this by **remembering** the results for each unique state `(i, j, xor_sum)` so that whenever the same state is encountered again, you can **immediately return the cached result** instead of re-exploring it.

How does Memoization work in this code?

1. **Memoization check**:
   Before the DFS function starts exploring a given cell `(i, j)` with a specific XOR sum, it first checks if the result for this state has already been computed:

   ```
   if (i, j, xor_sum) in memo:
       return memo[(i, j, xor_sum)]
   ```

   If the result is in the `memo` dictionary, it returns the cached value.

2. **Storing the result**:
   After computing the result for a state `(i, j, xor_sum)`, the algorithm stores it in the `memo` dictionary so that it can reuse it if the same state is encountered again:

   ```
   memo[(i, j, xor_sum)] = (right_path_sum + down_path_sum) %
   MOD
   ```

Example to Illustrate the Benefit of Memoization

Let's say you are currently at cell `(0, 0)` and have a `xor_sum = 2`. The DFS will recursively call itself to explore paths going right and down. But consider the situation when:

- You move right to `(0, 1)` with `xor_sum = 2 ^ 1 = 3`
- Then you move right again to `(0, 2)` with `xor_sum = 3 ^ 5 = 6`
- You continue the recursion, and eventually you backtrack and explore other paths.

Now imagine that **on a different recursive branch**, you visit the same cell `(0, 2)` again with `xor_sum = 6`. Without memoization, you'd have to recalculate the result of the entire DFS from this point again, even though you've already done this calculation before.

With memoization, the second time you visit `(0, 2)` with `xor_sum = 6`, the algorithm **remembers** that the result for this state was already computed, and it simply **returns the stored result** instead of recalculating it.

Why This Saves Time?

- **Avoid Redundant Computation**: In many cases, multiple paths will visit the same cell `(i, j)` with the same XOR value. Without memoization, you'd compute the result from this cell repeatedly. With memoization, you compute it once and reuse the result.
- **Reducing Time Complexity**: Memoization significantly reduces the number of recursive calls. Each unique state `(i, j, xor_sum)` is computed only once, and future visits to that state will return the cached result. This transforms an exponential time complexity (without memoization) into a much more manageable one.

Time Complexity Analysis with and without Memoization

1. **Without Memoization**:

   Each DFS call explores two possible paths (right and down), and in the worst case, the number of recursive calls could grow exponentially with the number of cells. There are $O(2^{m+n})$ possible paths, leading to an exponential time complexity.

2. **With Memoization**:$O(m \times n \times 2^B)$

   Each state `(i, j, xor_sum)` is computed only once. There are at most m×n possible grid cells, and for each cell, you store results for each possible XOR sum. Since XOR values can range from 0 to $2^{bit\ length\ of\ the\ grid\ values}$−1, the number of distinct XOR values is at most O(2^B), where B is the number of bits needed to represent the largest grid value.

The overall time complexity with memoization is reduced to:

$O(m \times n \times 2^B)$

where BBB is the number of bits required to represent the grid values. This is a huge improvement over the exponential complexity.

Conclusion

- **Memoization** ensures that the algorithm doesn't waste time recalculating the result for the same state multiple times.
- By caching results for each unique state `(i, j, xor_sum)`, memoization **significantly reduces the number of recursive calls** and leads to much better performance, especially for large grids.

Why store xor_sum?

You're right to question whether storing `xor_sum` is necessary, and your intuition is pointing to an important detail. Let's clarify **why we need to store the `xor_sum`** along with `(i, j)` in the memoization cache, and why just storing `(i, j)` would not be enough.

Why is `xor_sum` necessary?

The key reason for storing `xor_sum` in addition to `(i, j)` is that the **same grid cell `(i, j)`** can be visited with **different XOR values** along different paths.

Here's why it matters:

1. **Different Paths, Same Cell**:

   The grid itself is the same, so the coordinates `(i, j)` are the same. However, the XOR value (`xor_sum`) accumulates differently depending on the path taken to reach that cell. Since you can move right or down and accumulate XOR values in different ways, the result depends on **how you arrived at the cell** (i.e., the XOR sum along the path).

Example:

Suppose you're at `(i, j)` and the XOR sum is `xor_sum`. If you go **down** from `(i, j)`, you might end up with a different XOR sum compared to going **right** from the same cell. Therefore, **the same position `(i, j)` can lead to different outcomes based on the `xor_sum`**.

2. **A single cell is visited with many possible XOR sums**:

   Consider the following grid:

   ```
   Grid:  [[1, 2],
           [3, 4]]
   ```

   Let's say the starting point is `(0, 0)` and the target XOR sum is `k = 4`.

   - From `(0, 0)`, you can either:
     - Move **right** to `(0, 1)` with XOR sum = `1 ^ 2 = 3`.
     - Move **down** to `(1, 0)` with XOR sum = `1 ^ 3 = 2`.
   - From `(0, 1)` with XOR sum `3`, you can:
     - Move **down** to `(1, 1)` with XOR sum `3 ^ 4 = 7`.
   - From `(1, 0)` with XOR sum `2`, you can:
     - Move **right** to `(1, 1)` with XOR sum `2 ^ 4 = 6`.

   The **same cell** `(1, 1)` is reached multiple times but with different XOR sums (either `7` or `6`), and each path may yield different results. In this case, you cannot use the position `(i, j)` alone to distinguish between these different XOR sums.

Example with Memoization

Consider the example where we use memoization:

1. **State Tracking**:
   In your memoization dictionary, you store a

key that uniquely identifies a state. The state is represented by **both** the coordinates `(i, j)` and the accumulated `xor_sum` from the start to that point.

Without storing `xor_sum`, you wouldn't be able to distinguish between two different paths that reach the same cell `(i, j)` but with different XOR sums, leading to incorrect results.

Here's how the memoization works with `xor_sum`:

```
python
Copy code
memo[(i, j, xor_sum)] = number_of_paths_from_this_state
```

2. **Why just `(i, j)` won't work**:
   If you only store
   `(i, j)`, you would lose the information about the XOR sum and end up with incorrect results. Specifically, two different paths might reach the same cell `(i, j)` with different XOR sums, and you wouldn't be able to distinguish between them.

   - For example, consider cell `(i, j) = (0, 0)` with XOR sum `0`. From here, you might go either right or down, and the `xor_sum` will change accordingly. If you only stored `(i, j)`, you'd be unable to track which path you're on, and this would result in incorrect or incomplete results for that cell.

Visualizing the Importance of `xor_sum`

Let's visualize a small part of the grid to show why `xor_sum` is necessary:

```
plaintext
Copy code
Grid:  [[2, 1, 5],
        [7, 10, 0],
        [12, 6, 4]]
```

Starting from `(0, 0)`:

- **Path 1 (Right → Right → Down)**:
  - From `(0, 0)`, XOR sum = `0 ^ 2 = 2`.
  - Move right to `(0, 1)` with XOR sum `2 ^ 1 = 3`.
  - Move right to `(0, 2)` with XOR sum `3 ^ 5 = 6`.
  - Move down to `(1, 2)` with XOR sum `6 ^ 0 = 6`.
  - Move down to `(2, 2)` with XOR sum `6 ^ 4 = 2` (not equal to `k = 11`).
- **Path 2 (Down → Right → Down)**:
  - From `(0, 0)`, XOR sum = `0 ^ 2 = 2`.
  - Move down to `(1, 0)` with XOR sum `2 ^ 7 = 5`.
  - Move right to `(1, 1)` with XOR sum `5 ^ 10 = 15`.
  - Move down to `(2, 1)` with XOR sum `15 ^ 6 = 9`.
  - Move right to `(2, 2)` with XOR sum `9 ^ 4 = 13` (not equal to `k = 11`).
- **Path 3 (Down → Down → Right)**:
  - From `(0, 0)`, XOR sum = `0 ^ 2 = 2`.
  - Move down to `(1, 0)` with XOR sum `2 ^ 7 = 5`.
  - Move down to `(2, 0)` with XOR sum `5 ^ 12 = 9`.
  - Move right to `(2, 1)` with XOR sum `9 ^ 6 = 15`.
  - Move right to `(2, 2)` with XOR sum `15 ^ 4 = 11` (this path results in a valid solution).

Even though all paths eventually visit the same cell `(2, 2)`, each path has a **different XOR sum** due to the path taken. If you **only stored** **`(i, j)`**, you wouldn't be able to differentiate between the different XOR sums at `(2, 2)` and would miss valid paths (like Path 3).

Conclusion

- **Storing `xor_sum` is essential** because different paths can lead to the same cell `(i, j)` with different XOR sums.

- If you only store `(i, j)`, the algorithm would not be able to differentiate between these different XOR sums and could incorrectly count paths or miss valid ones.
- Memoizing the state `(i, j, xor_sum)` ensures that each unique combination of position and XOR sum is computed only once, and subsequent visits to the same state can quickly return the cached result.

This is why the memoization key is `(i, j, xor_sum)` and not just `(i, j)`.