2435. Paths in Matrix Whose Sum Is Divisible by K

Hard | Array | Dynamic Programming | Matrix | Leetcode Link

Problem Description

In this LeetCode problem, we have a two-dimensional grid representing a matrix with `m` rows and `n` columns, and we are tasked with finding paths from the top left corner `(0, 0)` to the bottom right corner `(m - 1, n - 1)`. The only permitted movements are either right or down. Each cell in the grid contains an integer, and we want to consider only those paths for which the sum of the integers along the path is divisible by a given integer `k`.

The problem statement asks us to return the total number of such paths modulo `10^9 + 7`. This large number is used to prevent overflow issues due to very large result values, which is a common practice in computational problems.

Intuition

To arrive at the solution, we have to think in terms of dynamic programming, which is a method for solving problems by breaking them down into simpler subproblems. The main idea behind the approach is to create a 3-dimensional array `dp` where each element `dp[i][j][s]` represents the number of ways to reach cell `(i, j)` such that the sum of all elements in the path modulo `k` is `s`.

Here's how we can think through it:

1. Initialize a 3-dimensional array `dp` of size `m x n x k` with zeroes, which will store the count of paths that lead to a certain remainder when the sum of path elements is divided by `k`. The third dimension `s` represents all possible remainders `[0, k-1]`.
2. Set `dp[0][0][grid[0][0] % k]` to 1 as a base case since there's one way to be at the starting cell with the sum equal to the element of that cell modulo `k`.
3. Start iterating over the grid, cell by cell. At each cell, we want to update the `dp` array for all possible sums modulo `k`. We consider two possibilities to arrive at a cell `(i, j)`: from the cell above `(i - 1, j)` and from the cell to the left `(i, j - 1)`. For each of these cells, we add to the path count for the current remainder `s`.
4. We calculate the new remainder `t` after including the current cell's value using the formula `t = ((s - grid[i][j] % k) + k) % k`. This gives us the remainder from the previous cell that would lead to a current remainder `s` after adding `grid[i][j]`.
5. If the cell above `(i - 1, j)` is valid, we add the number of ways to reach it with the remainder `t` to `dp[i][j][s]`. If the cell to the left `(i, j - 1)` is valid, we do the same.
6. Since we only care about the counts modulo `10^9 + 7`, we take the modulo at each update step to keep numbers in the range.
7. Finally, we're interested in the number of paths that have a sum divisible by `k` when we've reached the bottom-right cell. This corresponds to `dp[m - 1][n - 1][0]`, the number of paths with a remainder of `0`, which we return as the answer.

By following these steps, we can fill up our `dp` table and compute the required value efficiently, avoiding the need to explicitly enumerate all possible paths, which would be impractical for large grids.

Solution Approach

The implementation utilizes dynamic programming to efficiently compute the number of paths that lead to the bottom-right corner of the grid with sums divisible by `k`:

1. **Initialization**: We initialize a 3D list, `dp`, of size `m * n * k`, where `m` is the number of rows and `n` is the number of columns in the grid. This array will store the count of paths for each possible sum modulo `k` (`s` ranges from 0 to `k-1`) for each cell. Initially, all elements are set to 0.
2. **Base Case**: The path count of the starting position `(0, 0)` is set such that `dp[0][0][grid[0][0] % k]` is 1, because there is only one way to be at the starting cell and the sum equals the element in the starting cell modulo `k`.

   ```
   dp[0][0][grid[0][0] % k] = 1
   ```

3. **Main Logic**: We iterate through each cell `(i, j)` in the grid, and for each cell, we iterate through all possible remainders `s` (from 0 to `k-1`). We update `dp[i][j][s]` by adding the number of ways to reach either the cell above `(i - 1, j)` or the cell to the left `(i, j - 1)` that would result in a remainder `s` after adding the value of the current cell.

   The transition formula used is:

   ```
   t = ((s - grid[i][j] % k) + k) % k
   if i:
       dp[i][j][s] += dp[i - 1][j][t]
   if j:
       dp[i][j][s] += dp[i][j - 1][t]
   ```

   For each `dp` update, we take the modulo operation to ensure the result stays within the required range:

   ```
   dp[i][j][s] %= mod
   ```

4. **Final Result**: The number of paths where the sum of the elements is divisible by `k` is the last cell's value `dp[m - 1][n - 1][0]`, since we are interested in paths with a sum that has a remainder of 0 after modulo division by `k`.

   ```
   return dp[-1][-1][0]
   ```

By following this approach, we can compute the number of valid paths without visiting all possible paths, which would be computationally expensive especially on larger grids. This solution leverages the property of modulo operation and the principle of dynamic programming, specifically memoization, to store intermediate results and avoid repetitive work. The modular arithmetic ensures that we handle large numbers efficiently and prevent arithmetic overflow, which is a common issue in problems involving counting and combinatorics on a large scale.

Example Walkthrough

Let's illustrate the dynamic programming approach with a small example. Consider a `2x3` grid with the following values:

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

And let `k = 3`. We want to find all the paths from the top left corner to the bottom right with sums divisible by `k`.

1. **Initialization**: We first set up a 3D array `dp` of size `2x3x3` (since `m = 2`, `n = 3`, and `k = 3`). We initialize all elements to 0.
2. **Base case**: For the starting cell `(0, 0)` with the value `1`, we set `dp[0][0][1 % 3] = dp[0][0][1] = 1`. There is one way to be at `(0, 0)` with a sum that modulo `k` is `1`.
3. **Main logic**:
   - Starting with cell `(0, 1)` with the value `1`, the remainder `s` will be `((1 + 1) % 3) = 2`. Since we can only arrive from the left, we will update `dp[0][1][2]` to 1.
   - Now for cell `(0, 2)` with the value `2`, `s` will be `((2 + 2) % 3) = 1`. Update `dp[0][2][1]` to 1 as we can only arrive from the left.
   - For cell `(1, 0)` with the value `2`, `s` will be `((1 + 2) % 3) = 0`. Update `dp[1][0][0]` to 1 since we can only arrive from above.
   - At cell `(1, 1)` with the value `3`, we check from above `(0, 1)` and left `(1, 0)`. From above, `t = ((s - 3 % 3) + 3) % 3 = s`, we add `dp[0][1][t] = dp[0][1][s]` to `dp[1][1][s]`. We will do the same for the left cell `(1, 0)`. Hence, `dp[1][1][0]` will be updated to 2 (1 from above, 1 from the left).
   - Lastly, for cell `(1, 2)` with the value `4`, the remainder `s` will be `((0 + 4) % 3) = 1`. Update `dp[1][2][1]` by adding counts from above `(1, 1)` with value `dp[1][1][t]` where `t = ((1 - 4 % 3) + 3) % 3 = 0`, and from the left `(1, 2)` with value `dp[1][1][1]`. Thus, `dp[1][2][1]` will increase by 2 (all from above, nothing from the left, as `dp[1][1][1]` is 0).
4. **Final result**: We look at `dp[1][2][0]`, but in our case, the number of paths that end with a sum divisible by `k` is stored in `dp[1][2][1]`. Since the bottom right cell (`(1, 2)`) has a sum remainder `s = 1`, not `0`, there are no paths that sum up to a number divisible by `k`, and the method would return `0`.

Using this approach, we managed to calculate the required paths without exhaustively iterating through all paths. We used a combination of iterative updates based on previous states and modular arithmetic to maintain efficiency and correctness.

Python Solution

```python
class Solution:
    def numberOfPaths(self, grid, k):
        # Obtain the dimensions of the grid
        num_rows, num_cols = len(grid), len(grid[0])

        # Initialize a dynamic programming table to hold counts of paths
        # with different remainders modulo k at each cell
        # dp[row][col][remainder] will store the number of ways to reach
        # cell (row, col) such that the path sum has a remainder of `remainder` when divided by k
        dp = [[[0] * k for _ in range(num_cols)] for _ in range(num_rows)]
        # Set the initial case for the starting cell (top-left corner)
        dp[0][0][grid[0][0] % k] = 1

        # Define the modulo operation base
        mod_base = 10**9 + 7

        # Iterate through each cell in the grid
        for row in range(num_rows):
            for col in range(num_cols):
                # For each cell, iterate through all possible remainders
                for remainder in range(k):
                    # Compute the adjusted remainder to update the paths count
                    adjusted_remainder = ((remainder - grid[row][col] % k) + k) % k
                    # If there is a row above the current cell, add the number of paths from the cell above
                    if row:
                        dp[row][col][remainder] += dp[row - 1][col][adjusted_remainder]
                    # If there is a column to the left of the current cell, add the number of paths from the cell to the left
                    if col:
                        dp[row][col][remainder] += dp[row][col - 1][adjusted_remainder]
                    # Apply modulo operation to avoid large integers
                    dp[row][col][remainder] %= mod_base

        # Return the result, which is the number of paths that lead to the bottom-right corner of the grid
        # with a path sum divisible by k (remainder 0)
        return dp[-1][-1][0] # dp[num_rows - 1][num_cols - 1][0] in non-Pythonic indexing

```

Java Solution

```java
class Solution {
    // Define the modulus constant for preventing integer overflow
    private static final int MOD = (int) 1e9 + 7;

    public int numberOfPaths(int[][] grid, int k) {
        // m and n represent the dimensions of the grid
        int numRows = grid.length;
        int numCols = grid[0].length;

        // 3D dp array to store the number of ways to reach a cell (i, j)
        // such that the path sum modulo k is s
        int[][][] dp = new int[numRows][numCols][k];

        // Base case: start at the top-left corner of the grid
        dp[0][0][grid[0][0] % k] = 1;

        // Iterate over all cells of the grid
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                // Try all possible sums modulo k
                for (int sumModK = 0; sumModK < k; ++sumModK) {
                    // Calculate the modulo to identify how the current value of grid contributes to the new sum
                    int remainder = ((sumModK - grid[i][j] % k) + k) % k;

                    // If not in the first row, add paths from the cell above
                    if (i > 0) {
                        dp[i][j][sumModK] += dp[i - 1][j][remainder];
                    }
                    // If not in the first column, add paths from the cell on the left
                    if (j > 0) {
                        dp[i][j][sumModK] += dp[i][j - 1][remainder];
                    }

                    // Use modulus operation to prevent integer overflow
                    dp[i][j][sumModK] %= MOD;
                }
            }
        }

        // The result is the number of ways to reach the bottom-right corner such that path sum modulo k is 0
        return dp[numRows - 1][numCols - 1][0];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>
using namespace std;

class Solution {
public:
    int numberOfPaths(vector<vector<int>>& grid, int k) {
        // Dimensions of the grid
        int m = grid.size(), n = grid[0].size();
        // Modulo value to prevent overflow
        const int MOD = 1e9 + 7;

        // A 3D vector to store the number of paths, with the third dimension representing the sum modulo k
        vector<vector<vector<int>>> memo(m, vector<vector<int>>(n, vector<int>(k, -1)));

        // Define the depth-first search function using std::function for recursion
        function<int(int, int, int)> dfs = [&](int row, int col, int sum) {
            // Base case: outside of the grid bounds, return 0
            if (row < 0 || row >= m || col < 0 || col >= n) return 0;

            // Add the current cell's value to the running sum and apply modulo k
            sum = (sum + grid[row][col]) % k;

            // If we reached the bottom-right cell, return 1 if sum modulo k is 0, otherwise return 0
            if (row == m - 1 && col == n - 1) {
                return sum == 0 ? 1 : 0;
            }

            // Check if this state has already been computed
            if (memo[row][col][sum] != -1) {
                return memo[row][col][sum];
            }

            // Recurse to the right cell and the bottom cell and sum their path counts
            int pathCount = dfs(row + 1, col, sum) + dfs(row, col + 1, sum);
            // Apply modulo operations to prevent overflow
            pathCount %= MOD;

            // Cache the result in the memoization table
            memo[row][col][sum] = pathCount;

            // Return the total number of paths from this cell
            return pathCount;
        };

        // Call the DFS function starting from the top-left cell of the grid with an initial sum of 0
        return dfs(0, 0, 0);
    }
};

```

Typescript Solution

```typescript
function numberOfPaths(grid: number[][], k: number): number {
    // Define the modulo constant for the final answer
    const MOD = 10 ** 9 + 7;

    // Get the dimensions of the grid
    const numRows = grid.length;
    const numCols = grid[0].length;

    // Initialize a 3D array to store the number of ways to reach a cell
    // such that the sum of values mod k is a certain remainder
    let paths = Array.from({ length: numRows + 1 }, () =>
        Array.from({ length: numCols + 1 }, () => new Array(k).fill(0)),
    );

    // There is one way to reach the starting position (0,0) with a sum of 0 (mod k)
    paths[0][1][0] = 1;

    // Iterate over all cells in the grid
    for (let row = 0; row < numRows; row++) {
        for (let col = 0; col < numCols; col++) {
            // Iterate over all possible remainders
            for (let remainder = 0; remainder < k; remainder++) {
                // Compute the next key as the sum of the current cell's value and the previous remainder, mod k
                let newRemainder = (grid[row][col] + remainder) % k;

                // Update the number of ways to reach the current cell such that the sum of values mod k
                // is the new remainder. Take into account paths from the top and from the left.
                // Ensure the sum is within the MOD range
                paths[row + 1][col + 1][newRemainder] =
                    (paths[row][col + 1][remainder] + paths[row + 1][col][remainder] + paths[row + 1][col + 1][newRemainder]) % MOD
            }
        }
    }

    // The answer is the number of ways to reach the bottom-right corner
    // such that the total sum mod k is 0
    return paths[numRows][numCols][0];
}

```

Time and Space Complexity

The provided Python code defines a method to calculate the number of paths on a 2D grid where the sum of the values along the path is divisible by `k`. It uses dynamic programming to store the counts for intermediate paths where the sum of the values modulo `k` is a specific remainder.

Time Complexity:

The time complexity of the given code can be analyzed by considering the three nested loops:

1. The outermost loop runs for `m` iterations, where `m` is the number of rows in the grid.
2. The middle loop runs for `n` iterations for each `i`, where `n` is the number of columns in the grid.
3. The innermost loop runs for `k` iterations for each combination of `i` and `j`.

Combining these, we get `m * n * k` iterations in total. Within the innermost loop, all operations are constant time. Hence, the time complexity is `O(m * n * k)`.

Space Complexity:

The space complexity is determined by the size of the `dp` array, which stores intermediate counts for each cell and each possible remainder modulo `k`:

- The `dp` array is a 3-dimensional array with dimensions `m`, `n`, and `k`.
- This results in a space requirement for `m * n * k` integers.

Hence, the space complexity of the code is also `O(m * n * k)`.