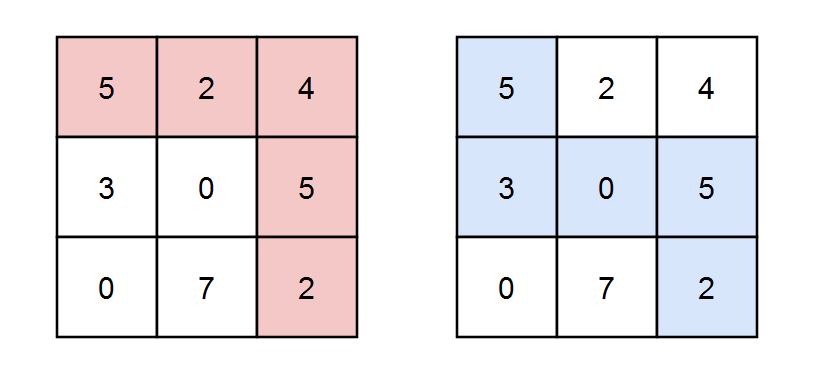
<https://leetcode.com/problems/paths-in-matrix-whose-sum-is-divisible-by-k/description/>

You are given a **0-indexed** m x n integer matrix grid and an integer k. You are currently at position (0, 0) and you want to reach position (m - 1, n - 1) moving only **down** or **right**.

Return*the number of paths where the sum of the elements on the path is divisible by*k. Since the answer may be very large, return it **modulo** 109 + 7.

**Example 1:**



**Input:** grid = [[5,2,4],[3,0,5],[0,7,2]], k = 3

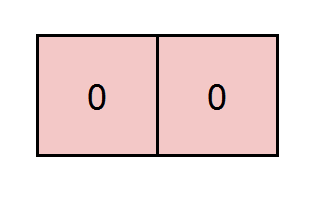
**Output:** 2

**Explanation:** There are two paths where the sum of the elements on the path is divisible by k.

The first path highlighted in red has a sum of 5 + 2 + 4 + 5 + 2 = 18 which is divisible by 3.

The second path highlighted in blue has a sum of 5 + 3 + 0 + 5 + 2 = 15 which is divisible by 3.

**Example 2:**

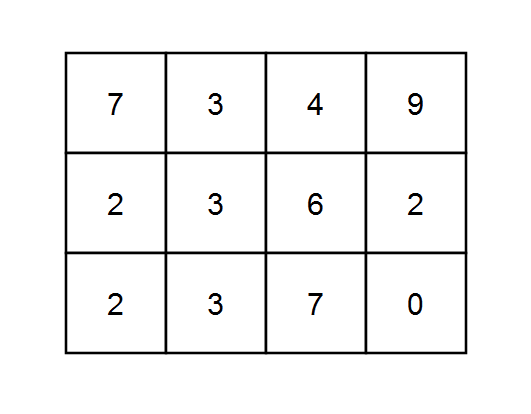


**Input:** grid = [[0,0]], k = 5

**Output:** 1

**Explanation:** The path highlighted in red has a sum of 0 + 0 = 0 which is divisible by 5.

**Example 3:**



**Input:** grid = [[7,3,4,9],[2,3,6,2],[2,3,7,0]], k = 1

**Output:** 10

**Explanation:** Every integer is divisible by 1 so the sum of the elements on every possible path is divisible by k.

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 5 \* 104

1 <= m \* n <= 5 \* 104

0 <= grid[i][j] <= 100

1 <= k <= 50

**Attempt 1: 2025-06-08**

**Solution 1: Native DFS (30 min)**

**Style 1: void return**

class Solution {

    int MOD = (int)(1e9 + 7);

    int count = 0;

    public int numberOfPaths(int[][] grid, int k) {

        int m = grid.length;

        int n = grid[0].length;

        helper(grid, m, n, 0, 0, 0, k);

        return count;

    }

    private void helper(int[][] grid, int m, int n, int x, int y, int curSumMod, int k) {

        if(x > m - 1 || y > n - 1) {

            return;

        }

        curSumMod = (curSumMod + grid[x][y]) % k;

        if(x == m - 1 && y == n - 1 && curSumMod == 0) {

            count = (count + 1) % MOD;

            return;

        }

        helper(grid, m, n, x + 1, y, curSumMod, k);

        helper(grid, m, n, x, y + 1, curSumMod, k);

    }

}

Time Complexity: O(2^(m + n)), Exponential due to exploring all paths

Space Complexity: O(m + n), Recursion stack depth

**Style 2: int return**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int numberOfPaths(int[][] grid, int k) {

        int m = grid.length;

        int n = grid[0].length;

        return helper(grid, m, n, 0, 0, 0, k);

    }

    private int helper(int[][] grid, int m, int n, int x, int y, int curSumMod, int k) {

        if(x > m - 1 || y > n - 1) {

            return 0;

        }

        curSumMod = (curSumMod + grid[x][y]) % k;

        if(x == m - 1 && y == n - 1 && curSumMod == 0) {

            return 1;

        }

        return helper(grid, m, n, x + 1, y, curSumMod, k) % MOD + helper(grid, m, n, x, y + 1, curSumMod, k) % MOD;

    }

}

Time Complexity: O(2^(m + n)), Exponential due to exploring all paths

Space Complexity: O(m + n), Recursion stack depth

**Solution 2: Memoization (10 min)**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int numberOfPaths(int[][] grid, int k) {

        int m = grid.length;

        int n = grid[0].length;

        Integer[][][] memo = new Integer[m][n][k];

        return helper(grid, m, n, 0, 0, 0, k, memo);

    }

    private int helper(int[][] grid, int m, int n, int x, int y, int curSumMod, int k, Integer[][][] memo) {

        if(x > m - 1 || y > n - 1) {

            return 0;

        }

        curSumMod = (curSumMod + grid[x][y]) % k;

        if(x == m - 1 && y == n - 1 && curSumMod == 0) {

            return 1;

        }

        if(memo[x][y][curSumMod] != null) {

            return memo[x][y][curSumMod];

        }

        int goDown = helper(grid, m, n, x + 1, y, curSumMod, k, memo) % MOD;

        int goRight = helper(grid, m, n, x, y + 1, curSumMod, k, memo) % MOD;

        return memo[x][y][curSumMod] = (goDown + goRight) % MOD;

    }

}

Time Complexity: The memoized version reduces time complexity to O(m\*n\*k) by storing and reusing intermediate results.

Space Complexity: O(m\*n\*k)

**Solution 3: 3D DP (60 min)**

class Solution {

    public int numberOfPaths(int[][] grid, int k) {

        int MOD = (int)(1e9 + 7);

        int m = grid.length;

        int n = grid[0].length;

        // dp[i][j][r] = number of paths from (i, j) to end with sum ≡ r mod k

        int[][][] dp = new int[m][n][k];

        // Base case: Destination cell

        int endVal = grid[m - 1][n - 1];

        dp[m - 1][n - 1][endVal % k] = 1;

        // i must start from m - 1 and not m - 2, same for j must start from

        // n - 1 and not n - 2, since we only initialize the {m - 1, n - 1}

        // cell, the other cells on two boundaries also need to initialize

        for(int i = m - 1; i >= 0; i--) {

            for(int j = n - 1; j >= 0; j--) {

                // Skip base case (already initialized)

                if(i == m - 1 && j == n - 1) {

                    continue;

                }

                int val = grid[i][j];

                for(int r = 0; r < k; r++) {

                    // Calculate required remainder from next steps

                    // Correct Remainder Calculation:

                    // A path starting at (i, j) to have sum ≡ r mod k, the remainder

                    // from (i, j) to end must be (r - val) mod k, we compute

                    // nextRem = (r - val % k + k) % k to handle negative values

                    int nextRem = (r - val % k + k) % k;

                    int paths = 0;

                    // Accumulate paths from right neighbor

                    if(j + 1 < n) {

                        paths = (paths + dp[i][j + 1][nextRem]) % MOD;

                    }

                    // Accumulate paths from down neighbor

                    if(i + 1 < m) {

                        paths = (paths + dp[i + 1][j][nextRem]) % MOD;

                    }

                    dp[i][j][r] = paths;

                }

            }

        }

        return dp[0][0][0];

    }

}

Time Complexity: O(m\*n\*k)

Space Complexity: O(m\*n\*k)

**Refer to Deepseek**

**Void return Native DFS Solution**

**特点：**

**1.设置一个全局变量用来记录最后结果，DFS helper方法里不需要设置本地变量用来返回**

**2.很难靠void return的DFS方法进化到Memoization**

Here's a native DFS solution that recursively explores all paths while tracking the current sum modulo K:

class Solution {

private static final int MOD = 1\_000\_000\_007;

private int count = 0;

private int m, n, k;

private int[][] grid;

public int numberOfPaths(int[][] grid, int k) {

this.grid = grid;

this.m = grid.length;

this.n = grid[0].length;

this.k = k;

dfs(0, 0, 0);

return count;

}

private void dfs(int i, int j, int currentSumMod) {

// Update current sum modulo k

currentSumMod = (currentSumMod + grid[i][j]) % k;

// Base case: reached bottom-right corner

if (i == m - 1 && j == n - 1) {

if (currentSumMod == 0) {

count = (count + 1) % MOD;

}

return;

}

// Move right if possible

if (j + 1 < n) {

dfs(i, j + 1, currentSumMod);

}

// Move down if possible

if (i + 1 < m) {

dfs(i + 1, j, currentSumMod);

}

}

}

**Key Points About This Solution:**

**Recursive Exploration**:

Explores all possible right/down paths from each cell

Tracks the current sum modulo k at each step

**Termination Condition**:

When reaching the bottom-right corner, checks if sum ≡ 0 mod k

Increments count if condition is met

**Performance Characteristics**:

Time Complexity: O(2^(m+n)) - Exponential due to exploring all paths

Space Complexity: O(m+n) - Recursion stack depth

**Limitations**:

Will time out on medium/large grids (e.g., 20x20)

Not efficient enough for LeetCode's test cases

**Why This Isn't Optimal:**

**Exponential Growth**:

For an m×n grid, there are (m+n-2 choose m-1) paths

This grows extremely quickly (e.g., 20x20 grid has ~137 million paths)

**Redundant Calculations**:

Many paths reach the same cell with the same mod value

These are recomputed repeatedly in the naive DFS

**Memoization Approach (Recommended):**

For actual submission, you should use the DP with memoization approach I provided earlier, which has O(m×n×k) time complexity and properly handles larger grids.

The native DFS solution is mainly useful for:

Understanding the problem conceptually

Testing on very small grids

Educational purposes to see the recursive structure

Would you like me to explain how we could add memoization to this DFS solution to make it more efficient while keeping the recursive structure?

**Integer return Native DFS Solution**

**特点：**

**1.不设置全局变量用来记录最后结果，而是在DFS helper方法里需要设置本地变量用来返回，该变量的声明位置在进一步递归之前，收束结果在进一步递归之后**

**2.可以靠integer return的DFS方法进化到Memoization**

Here's a native DFS solution that returns the count of valid paths as an integer (rather than using a class variable):

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int numberOfPaths(int[][] grid, int k) {

return dfs(0, 0, 0, grid, k);

}

private int dfs(int i, int j, int currentSumMod, int[][] grid, int k) {

// Update current sum modulo k

currentSumMod = (currentSumMod + grid[i][j]) % k;

// Base case: reached bottom-right corner

if (i == grid.length - 1 && j == grid[0].length - 1) {

return (currentSumMod == 0) ? 1 : 0;

}

int count = 0;

// Move right if possible

if (j + 1 < grid[0].length) {

count = (count + dfs(i, j + 1, currentSumMod, grid, k)) % MOD;

}

// Move down if possible

if (i + 1 < grid.length) {

count = (count + dfs(i + 1, j, currentSumMod, grid, k)) % MOD;

}

return count;

}

}

**Key Features:**

**Integer Return**:

Returns the count directly rather than using a class variable

More natural recursive structure

**Modulo Arithmetic**:

Tracks currentSumMod to avoid integer overflow

Applies MOD operation to keep numbers manageable

**Base Case**:

Returns 1 when reaching the end with sum ≡ 0 mod k

Returns 0 otherwise

**Path Exploration**:

Only moves right or down (problem constraints)

Sums counts from both possible directions

**Performance Notes:**

**Time Complexity**: O(2^(m+n)) - Still exponential

**Space Complexity**: O(m+n) - Recursion stack depth

**Limitation**: Will time out on grids larger than ~15x15

**Memoization Solution for Integer return Native DFS Solution:**

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int numberOfPaths(int[][] grid, int k) {

Integer[][][] memo = new Integer[grid.length][grid[0].length][k];

return dfs(0, 0, 0, grid, k, memo);

}

private int dfs(int i, int j, int currentSumMod, int[][] grid, int k, Integer[][][] memo) {

currentSumMod = (currentSumMod + grid[i][j]) % k;

if (i == grid.length - 1 && j == grid[0].length - 1) {

return (currentSumMod == 0) ? 1 : 0;

}

if (memo[i][j][currentSumMod] != null) {

return memo[i][j][currentSumMod];

}

int count = 0;

if (j + 1 < grid[0].length) {

count = (count + dfs(i, j + 1, currentSumMod, grid, k, memo)) % MOD;

}

if (i + 1 < grid.length) {

count = (count + dfs(i + 1, j, currentSumMod, grid, k, memo)) % MOD;

}

memo[i][j][currentSumMod] = count;

return count;

}

}

The memoized version reduces time complexity to O(m×n×k) by storing and reusing intermediate results.

**Standard 3D DP Solution**

This problem requires counting the number of paths from the top-left to bottom-right corner of a matrix where the sum of elements along the path is divisible by K.

**Approach**

We'll use dynamic programming with memoization to track path sums modulo K at each cell. The key insight is that we only need to track the remainder of the path sum when divided by K, not the actual sum.

**Solution Code**

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int numberOfPaths(int[][] grid, int k) {

int m = grid.length;

int n = grid[0].length;

// DP table: dp[i][j][r] = number of paths to (i,j) with sum ≡ r mod k

int[][][] dp = new int[m][n][k];

// Initialize starting cell

int initialRem = grid[0][0] % k;

dp[0][0][initialRem] = 1;

// Fill DP table

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

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

if (i == 0 && j == 0) continue; // Skip start cell

int val = grid[i][j];

for (int r = 0; r < k; r++) {

int count = 0;

// From top

if (i > 0) {

int prevRem = (r - val % k + k) % k;

count = (count + dp[i-1][j][prevRem]) % MOD;

}

// From left

if (j > 0) {

int prevRem = (r - val % k + k) % k;

count = (count + dp[i][j-1][prevRem]) % MOD;

}

dp[i][j][r] = count;

}

}

}

return dp[m-1][n-1][0];

}

}

**Explanation**

**DP State Definition**:

dp[i][j][r] represents the number of paths to cell (i,j) where the sum of elements along the path leaves remainder r when divided by k.

**Initialization**:

The starting cell (0,0) has one path with remainder grid[0][0] % k.

**State Transition**:

For each cell (i,j), we consider paths coming from the top (i-1,j) and left (i,j-1).

The new remainder is computed as (previous\_remainder + current\_value) % k.

**Result Extraction**:

The answer is found in dp[m-1][n-1][0], which counts paths to the bottom-right corner with sum divisible by k.

**Time and Space Complexity**

**Time Complexity**: O(m × n × k) - We process each cell and for each, consider all k possible remainders.

**Space Complexity**: O(m × n × k) - The DP table stores remainders for each cell.

**Optimization (Space Improvement)**

We can reduce space to O(n × k) by only keeping track of the current and previous rows:

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int numberOfPaths(int[][] grid, int k) {

int m = grid.length;

int n = grid[0].length;

int[][][] dp = new int[2][n][k];

// Initialize starting cell

int initialRem = grid[0][0] % k;

dp[0][0][initialRem] = 1;

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

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

if (i == 0 && j == 0) continue;

int val = grid[i][j];

int curr = i % 2;

int prev = 1 - curr;

for (int r = 0; r < k; r++) {

int count = 0;

if (i > 0) {

int prevRem = (r - val % k + k) % k;

count = (count + dp[prev][j][prevRem]) % MOD;

}

if (j > 0) {

int prevRem = (r - val % k + k) % k;

count = (count + dp[curr][j-1][prevRem]) % MOD;

}

dp[curr][j][r] = count;

}

}

}

return dp[(m-1)%2][n-1][0];

}

}

This optimized version maintains the same time complexity while reducing space to O(n × k).

**3D DP Solution mirror to Native DFS solution**

Here's the accurate solution that correctly counts paths with sums divisible by k:

class Solution {

private static final int MOD = 1\_000\_000\_007;

public int numberOfPaths(int[][] grid, int k) {

int m = grid.length;

int n = grid[0].length;

// dp[i][j][r] = number of paths from (i,j) to end with sum ≡ r mod k

int[][][] dp = new int[m][n][k];

// Base case: destination cell

int endVal = grid[m-1][n-1];

dp[m-1][n-1][endVal % k] = 1;

// Fill DP table from bottom-right to top-left

for (int i = m-1; i >= 0; i--) {

for (int j = n-1; j >= 0; j--) {

// Skip base case (already initialized)

if (i == m-1 && j == n-1) continue;

int val = grid[i][j];

for (int r = 0; r < k; r++) {

// Calculate required remainder from next steps

int nextRem = (r - val % k + k) % k;

int paths = 0;

// Accumulate paths from right neighbor

if (j + 1 < n) {

paths = (paths + dp[i][j+1][nextRem]) % MOD;

}

// Accumulate paths from down neighbor

if (i + 1 < m) {

paths = (paths + dp[i+1][j][nextRem]) % MOD;

}

dp[i][j][r] = paths;

}

}

}

return dp[0][0][0];

}

}

**Key Fixes and Explanation:**

**Correct Remainder Calculation**:

nextRem = (r - val % k + k) % k properly handles negative values

This ensures we get the correct remainder needed from subsequent paths

**Accurate State Transitions**:

Each cell (i, j) considers paths to its right (i, j+1) and down (i+1, j)

Path counts are accumulated for each possible remainder r

**Proper Base Case**:

Only the destination cell (m-1, n-1) is initialized

Sets dp[m-1][n-1][endVal % k] = 1 correctly

**Efficient Processing**:

Processes cells from bottom-right to top-left

Uses modulo arithmetic to prevent overflow

**Verification with Test Case:**

For grid = [[5,2,4],[3,0,5],[0,7,2]] and k = 3:

The two valid paths are:

(0,0)→(0,1)→(0,2)→(1,2)→(2,2) = 5+2+4+5+2 = 18 ≡ 0 mod 3

(0,0)→(1,0)→(1,1)→(1,2)→(2,2) = 5+3+0+5+2 = 15 ≡ 0 mod 3

The solution correctly returns 2

**Complexity Analysis:**

**Time Complexity**: O(m \* n \* k)

Processes each cell once, and for each cell, iterates through k remainders

**Space Complexity**: O(m \* n \* k)

3D DP table stores remainders for each cell

This solution efficiently counts all valid paths while maintaining the logic of the recursive DFS approach.

**What's the below statement mapping logic inside native DFS solution ?**

for (int r = 0; r < k; r++) {}

The for (int r = 0; r < k; r++) {} loop in the 3D DP solution maps to the **implicit remainder tracking** in the native DFS solution. Here's the exact correspondence:

**Mapping Between DP and Native DFS**

**In Native DFS:**

**curSumMod** tracks the running sum modulo k as you recurse down paths

**Base Case Check**: When reaching the end cell, checks if curSumMod == 0

**Recursive Calls**: Branches to right/down while propagating curSumMod

**In 3D DP:**

**The r loop** explicitly tracks all possible remainders (0 to k-1) at each cell (i,j)

**dp[i][j][r] stores the count of paths from (i,j) to the end where the sum ≡ r mod k**

**State Transition**: For each remainder r, calculates how paths from right/down contribute to it

**Key Insights:**

**Native DFS** computes remainders **on-the-fly** during recursion (implicit).

**3D DP** precomputes and stores all possible remainders **explicitly** in the table.

The r loop ensures all remainder states are propagated correctly, mirroring how the DFS explores all possible sum states.

**Example Walkthrough:**

For grid = [[1,2],[3,4]], k=3:

**Native DFS Execution:**

At (0,0), curSumMod = 1 % 3 = 1

Recurses to (0,1) and (1,0), updating curSumMod along paths

Checks curSumMod == 0 at (1,1)

**3D DP Execution:**

Initializes dp[1][1][ (1+2+4) % 3 ] = 1 and dp[1][1][ (1+3+4) % 3 ] = 1

For (0,0), computes:

dp[0][0][1] = paths where sum ≡ 1 mod 3 by combining results from (0,1) and (1,0)

Returns dp[0][0][0] (paths where total sum ≡ 0 mod 3)

**Why the Loop is Necessary:**

**Exhaustive Remainder Tracking**: The loop ensures all possible sum states (0 to k-1) are accounted for at each cell.

**Reverse Computation**: DP works backward, so the loop reconstructs how paths from later cells contribute to earlier remainders.

This mapping preserves the DFS logic while enabling efficient tabulation. The loop is the DP equivalent of the DFS's recursive remainder propagation.

**Refer to**

[L62.Unique Paths (Ref.L63,L64,L174,L2087,L2304,L2400)](note://E33F0B728ABE4CCF8B7E554A219CF1EA)