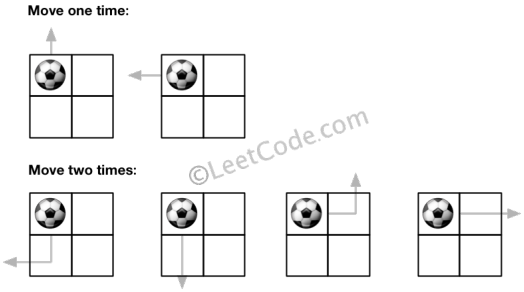
Out of Boundary Paths

There is an **m** by **n** grid with a ball. Given the start coordinate **(i,j)** of the ball, you can move the ball to **adjacent** cell or cross the grid boundary in four directions (up, down, left, right). However, you can **at most** move **N** times. Find out the number of paths to move the ball out of grid boundary. The answer may be very large, return it after mod $10^9 + 7$.

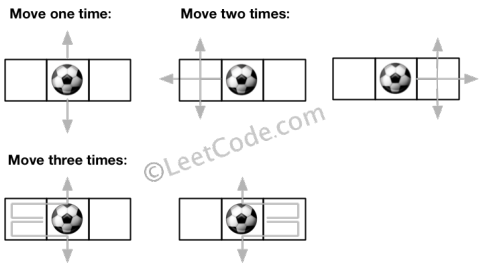
Example 1:

Input: `m = 2, n = 2, N = 2, i = 0, j = 0`
Output: `6`
Explanation:



Example 2:

Input: `m = 1, n = 3, N = 3, i = 0, j = 1`
Output: `12`
Explanation:



Note:

1. Once you move the ball out of boundary, you cannot move it back.
2. The length and height of the grid is in range [1,50].
3. N is in range [0,50].

Solution 1

The number of paths for N moves is the sum of paths for N - 1 moves from the adjacent cells. If an adjacent cell is out of the border, the number of paths is 1.

```
int findPaths(int m, int n, int N, int i, int j) {
  uint dp[51][50][50] = {};
  for (auto Ni = 1; Ni <= N; ++Ni)
    for (auto mi = 0; mi < m; ++mi)
      for (auto ni = 0; ni < n; ++ni)
        dp[Ni][mi][ni] = ((mi == 0 ? 1 : dp[Ni - 1][mi - 1][ni]) + (mi == m - 1? 1
: dp[Ni - 1][mi + 1][ni])
           + (ni == 0 ? 1 : dp[Ni - 1][mi][ni - 1]) + (ni == n - 1 ? 1 : dp[Ni - 1
][mi][ni + 1])) % 1000000007;
  return dp[N][i][j];
}
```

We can also reduce the memory usage by using two grids instead of N, as we only need to look one step back. We can use N % 2 and (N + 1) % 2 to alternate grids so we do not have to copy.

```
int findPaths(int m, int n, int N, int i, int j) {
    unsigned int g[2][50][50] = {};
    while (N-- > 0)
        for (auto k = 0; k < m; ++k)
            for (auto l = 0, nc = (N + 1) % 2, np = N % 2; l < n; ++l)
                g[nc][k][l] = ((k == 0 ? 1 : g[np][k - 1][l]) + (k == m - 1 ? 1 : g
[np][k + 1][l])
                    + (l == 0 ? 1 : g[np][k][l - 1]) + (l == n - 1 ? 1 : g[np][k][l
+ 1])) % 1000000007;
    return g[1][i][j];
}
```

As suggested by @mingthor, we can further decrease the memory usage (2 * m * n >> m * (n + 1)) as we only looking one row up. We will store new values for the current row in an array, and write these values back to the matrix as we process cells in the next row. This approach, however, impacts the runtime as we need extra copying for each step.

I experimented with different n and m (50 - 500), and N (5,000 - 50,000), and the second solution is approximately 10% faster than this one.

```
int findPaths(int m, int n, int N, int i, int j) {
    unsigned int g[50][50] = {}, r[50];
    while (N-- > 0)
        for (auto k = 0; k <= m; ++k)
            for (auto l = 0; l < n; ++l) {
                auto tmp = r[l];
                r[l] = (k == m ? 0 : ((k == 0 ? 1 : g[k - 1][l]) + (k == m - 1 ? 1
: g[k + 1][l])
                    + (l == 0 ? 1 : g[k][l - 1]) + (l == n - 1 ? 1 : g[k][l + 1]))
% 1000000007);
                if (k > 0) g[k - 1][l] = tmp;
            }
    return g[i][j];
}
```

written by votrubac original link here

Solution 2

`DP[i][j][k]` stands for how many possible ways to walk into cell `j,k` in step `i`, `DP[i][j][k]` only depends on `DP[i - 1][j][k]`, so we can compress 3 dimensional dp array to 2 dimensional.

```
public class Solution {
    public int findPaths(int m, int n, int N, int i, int j) {
        if (N <= 0) return 0;

        final int MOD = 1000000007;
        int[][] count = new int[m][n];
        count[i][j] = 1;
        int result = 0;

        int[][] dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        for (int step = 0; step < N; step++) {
            int[][] temp = new int[m][n];
            for (int r = 0; r < m; r++) {
                for (int c = 0; c < n; c++) {
                    for (int[] d : dirs) {
                        int nr = r + d[0];
                        int nc = c + d[1];
                        if (nr < 0 || nr >= m || nc < 0 || nc >= n) {
                            result = (result + count[r][c]) % MOD;
                        }
                        else {
                            temp[nr][nc] = (temp[nr][nc] + count[r][c]) % MOD;
                        }
                    }
                }
            }
            count = temp;
        }

        return result;
    }
}
```

written by shawngao original link here

Solution 3

Fastest posted Python solution so far. Takes about 120 ms, of which about 80 ms are for judge overhead and importing NumPy. The other three Python solutions posted so far take about 350-850 ms, of which about 40 ms are for judge overhead.

My two-dimensional `paths` array tells the number of paths ending at each cell with the moves made so far. So initially all zeros except for the starting position, which has one path (the empty path). The for each move, I spread each path number in all four directions.

This only keeps track of the paths staying **inside** the boundary. To compute how many paths went **outside** in the latest move, I take all previous paths, multiply them by 4 (for the four directions) and subtract the new number of inside-paths after the move.

```
import numpy as np

class Solution(object):
    def findPaths(self, m, n, N, i, j):
        paths = np.zeros((m, n), dtype=np.int64)
        paths[i][j] = 1
        out = 0
        mod = 10**9 + 7
        for _ in range(N):
            prev = paths % mod
            paths = prev - prev
            paths[1:] += prev[:-1]
            paths[:-1] += prev[1:]
            paths[:,1:] += prev[:,:-1]
            paths[:,:-1] += prev[:,1:]
            out += 4 * prev.sum() - paths.sum()
        return int(out % mod)
```

A slightly shorter version using Python `int`s (which can grow arbitrarily large) and doing mod 10^9-7 only at the end:

```
import numpy as np

class Solution(object):
    def findPaths(self, m, n, N, i, j):
        paths = np.zeros((m, n), dtype=object)
        paths[i][j] = 1
        out = 0
        for _ in range(N):
            prev = paths
            paths = prev * 0
            paths[1:] += prev[:-1]
            paths[:-1] += prev[1:]
            paths[:,1:] += prev[:,:-1]
            paths[:,:-1] += prev[:,1:]
            out += 4 * prev.sum() - paths.sum()
        return out % (10**9 + 7)
```

Still fairly fast, about 190 ms.

written by StefanPochmann original link here