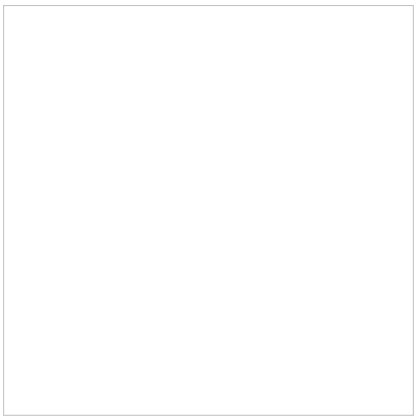
Knight Probability in Chessboard

On an `N` x `N` chessboard, a knight starts at the `r`-th row and `c`-th column and attempts to make exactly `K` moves. The rows and columns are 0 indexed, so the top-left square is `(0, 0)`, and the bottom-right square is `(N-1, N-1)`.

A chess knight has 8 possible moves it can make, as illustrated below. Each move is two squares in a cardinal direction, then one square in an orthogonal direction.



Each time the knight is to move, it chooses one of eight possible moves uniformly at random (even if the piece would go off the chessboard) and moves there.

The knight continues moving until it has made exactly `K` moves or has moved off the chessboard. Return the probability that the knight remains on the board after it has stopped moving.

Example:

```
Input: 3, 2, 0, 0
Output: 0.0625
Explanation: There are two moves (to (1,2), (2,1)) that will keep the knight on the b
oard.
From each of those positions, there are also two moves that will keep the knight on t
he board.
The total probability the knight stays on the board is 0.0625.
```

Note:

- `N` will be between 1 and 25.
- `K` will be between 0 and 100.
- The knight always initially starts on the board.

Solution 1

```
int[][] moves = {{1, 2}, {1, -2}, {2, 1}, {2, -1}, {-1, 2}, {-1, -2}, {-2, 1}, {-2, -1}};
public double knightProbability(int N, int K, int r, int c) {
    int len = N;
    double dp0[][] = new double[len][len];
    for(double[] row : dp0) Arrays.fill(row, 1);
    for(int l = 0; l < K; l++) {
        double[][] dp1 = new double[len][len];
        for(int i = 0; i < len; i++) {
            for(int j = 0; j < len; j++) {
                for(int[] move : moves) {
                    int row = i + move[0];
                    int col = j + move[1];
                    if(isLegal(row, col, len)) dp1[i][j] += dp0[row][col];
                }
            }
        }
        dp0 = dp1;
    }
    return dp0[r][c] / Math.pow(8, K);
}
private boolean isLegal(int r, int c, int len) {
    return r >= 0 && r < len && c >= 0 && c < len;
}
```

written by szlghl1 original link here

Solution 2

For this problem, I think memoization is more optimal than direct DP. The reason is that memoization can avoid a lot of unnecessary subproblems.
The runtime is O(KN^2).

C++

```
class Solution {
public:
    double knightProbability(int N, int K, int r, int c) {
        vector<vector<vector<double>>> dp(K+1, vector<vector<double>>(N, vector<dou
ble>(N, -1.0)));
        return helper(dp, N, K, r, c)/pow(8, K);
    }
private:
    double helper(vector<vector<vector<double>>>& dp, int N, int k, int r, int c) {
        // if out of board, return 0.0
        if (r < 0 || r >= N || c < 0 || c >= N) return 0.0;
        // when k = 0, no more move, so it's 100% safe
        if (k == 0) return 1.0;
        if (dp[k][r][c] != -1.0) return dp[k][r][c];
        dp[k][r][c] = 0.0;
        for (int i = -2; i <= 2; i++) {
            if (i == 0) continue;
            dp[k][r][c] += helper(dp, N, k-1, r+i, c+3-abs(i)) + helper(dp, N, k-1,
r+i, c-(3-abs(i)));
        }
        return dp[k][r][c];
    }
};
```

Java

```
class Solution {
    int[][] moves = {{1,2},{1,-2},{-1,2},{-1,-2},{2,-1},{2,1},{-2,-1},{-2,1}};
    public double knightProbability(int N, int K, int r, int c) {
        double[][][] dp = new double[K+1][N][N];
        return helper(dp, N, K, r, c)/Math.pow(8.0, K);
    }
    private double helper(double[][][] dp, int N, int k, int r, int c) {
        if (r < 0 || r >= N || c < 0 || c >= N) return 0.0;
        if (k == 0) return 1.0;
        if (dp[k][r][c] != 0.0) return dp[k][r][c];
        for (int i = 0; i < 8; i++)
            dp[k][r][c] += helper(dp, N, k-1, r+moves[i][0], c+moves[i][1]);
        return dp[k][r][c];
    }
}
```

written by zestypanda original link here

Solution 3

At the start, the knight is at (r, c) with probability 1 (and anywhere else with probability 0). Then update those probabilities over K moves.

```
def knightProbability(self, N, K, r, c):
    p = {(r, c): 1}
    for _ in range(K):
        p = {(r, c): sum(p.get((r+i, c+j), 0) + p.get((r+j, c+i), 0) for i in (1, -
1) for j in (2, -2)) / 8
             for r in range(N) for c in range(N)}
    return sum(p.values())
```

Shorter and maybe nicer version, influenced a bit by @flamesofmoon's solution:

```
def knightProbability(self, N, K, r, c):
    p = {(r, c): 1}
    for _ in range(K):
        p = {(r, c): sum(p.get((r+i, c+j), 0) for x in (1, 2) for i in (x, -x) for
j in (3-x, x-3)) / 8
             for r in range(N) for c in range(N)}
    return sum(p.values())
```

written by StefanPochmann original link here