Daily Coding Problem #304

Problem

This problem was asked by Two Sigma.

A knight is placed on a given square on an `8 x 8` chessboard. It is then moved randomly several times, where each move is a standard knight move. If the knight jumps off the board at any point, however, it is not allowed to jump back on.

After `k` moves, what is the probability that the knight remains on the board?

Solution

When we are faced with a problem that involves moving pieces, it is often a good idea to create a helper function that modularizes this part of the code. Here, we know that a knight always moves in an `L`-shape, two steps in one direction and then one step at a right angle. Therefore, to get all eight moves from a given square we can directly compute each possibility.

```
def get_moves(x, y):
    moves = [(1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1), (-2, 1), (-1, 2)]
    return [(x + i, y + j) for (i, j) in moves]
```

With this out of the way, we can formulate a straightforward recursive solution to our problem. There are two base cases:

- When we reach the last move, we should return 1 if the knight is still on the board, else 0.
- If at any point the knight moves off the board, the probability of that path becomes 0.

Otherwise, we should retrieve all eight jumps that can be made from the current square, apply our function to each of them, and average the results.

```
def on_board(x, y):
    return 0 <= x <= 7 and 0 <= y <= 7

def get_probability(x, y, k):
    if k == 0:
        return on_board(x, y)
    if not on_board(x, y):
        return 0

    jumps = get_moves(x, y)
    res = [get_probability(x, y, k - 1) for x, y in jumps]

    return 0.125 * sum(res)
```

While this works, it will quickly become extremely slow. With each successive value of k, there can be up to 8 new paths to explore each square, leading to an exponential runtime.

We can improve this with memoization or bottom-up dynamic programming. Either way, we want to store the probabilities associated with specific values of x, y, and k, so that we do not have to recompute them at later points.

If we choose the bottom-up approach, we could create a three-dimensional array that stored the probability of a knight ending up on a specific square for each turn, and populate each cell in order. However, this would be somewhat wasteful. Note, for example, that after the first move there will be at most eight squares that can be landed on, not 64.

In this case, it is simpler and more efficient to use memoization, caching each value of x, y, and k in a dictionary before returning the result.

```
def get_probability(x, y, k, memo={}):
    if (x, y, k) in memo:
        return memo[(x, y, k)]

    if k == 0:
        return on_board(x, y)
    if not on_board(x, y):
        return 0

    jumps = get_moves(x, y)
    probs = [get_probability(x, y, k - 1, memo) for x, y in jumps]

    memo[(x, y, k)] = 0.125 * sum(probs)

    return memo[(x, y, k)]
```

In the worst case, as `k` gets large, every square and move will be stored as a key in our dictionary, making the time and space complexity $O(N^2 * k)$, where `N` is the size of the chessboard.