Daily Coding Problem #312

Problem

This problem was asked by Wayfair.

You are given a `2 x N` board, and instructed to completely cover the board with the following shapes:

- Dominoes, or `2 x 1` rectangles.
- Trominoes, or `L`-shapes.

For example, if `N = 4`, here is one possible configuration, where `A` is a domino, and `B` and `C` are trominoes.

```
A B B C
A B C C
```

Given an integer `N`, determine in how many ways this task is possible.

Solution

This is one of those problems when there is an elegant dynamic programming solution, but it can be quite hard to find.

Let's imagine we have filled the board from left to right, and we are just placing the last piece. There are four different ways this can happen, which we will illustrate below. Here, `x` marks represent squares in the board that have already been covered, and `o` marks represent the final move.

```
Option 1: vertical domino
... x x x x o
... x x x x o

Option 2: horizontal domino
... x x x o o
... x x x x x

Option 3: upper tromino
... x x x o o
... x x x x o

Option 4: lower tromino
... x x x x o
... x x x o o
```

We can examine each of these options in order to come up with a recurrence relation. To formalize this a bit, let `f(N)` denote the number of possible configurations that can make up a `2 x N` rectangle.

If we end with a vertical domino, this number will be equal to `f(N - 1)`, since we are reducing the problem to one in which the block is one unit shorter. If we end with a horizontal domino, it must have been the case that another horizontal domino was laid just before. In this case, `f(N)` will equal `f(N - 2)`, since we are reducing the width of the block by two.

Cases 3 and 4 are trickier, but fortunately, since they are symmetrical, we know they will have the same value. For now let us denote `g(N)` be the number of possible ways to form a shape that consists of a `2 x N` block, plus a square jutting out.

So far, then, we have the following recurrence relation:

```
f(N) = f(N - 1) + f(N - 2) + 2 * g(N - 2)
```

Now let us try to solve for `g(N)` in a similar way. That is, we want to figure out a recurrence

relation that describes the number of ways to form a shape consisting of a block of length `N`, plus an extra square jutting out. This shape can end in the two ways shown below:

```
Option 1: upper tromino
... x x x o o
... x x x x

Option 2: lower tromino
... x x x o o
... x x x o
```

In the former case, a domino can extend out into the jut, leaving us with `g(N - 1)`. In the latter, we use a tromino to cover the extra square and the previous column, leaving us with `f(N - 1)`.

Therefore, we find that:

```
g(N) = g(N - 1) + f(N - 1)
```

Finally, let us note the initial conditions for each of these formulas. For a rectangular block of width `1`, there is only one way to tile it: by using a vertical domino. If we want to cover a rectangular block of width `2`, there are two ways: we can lay two dominos vertically or horizontally.

For a block of width `1` with an extra square jutting out, we can only cover this with a single tromino. Lastly, for a block of width `2` with an extra square, there are two possible tilings. These are represented below, with `x` and `o` representing different pieces.

```
Tiling 1: vertical domino + tromino
x o o
x o

Tiling 2: tromino + horizontal domino
x o o
x x
```

The initial conditions, then, are as follows:

```
f(1) = 1; f(2) = 2
g(1) = 1; g(2) = 2
```

Now, since we have the initial conditions, and we know the recurrence relation, we can iterate from `3` to `N` for each of these functions, populating each value into the respective array. In the end, we can return the value of `f(N)`, which represents exactly the number of ways to tile a rectangle of shape `2 x N`.

```
def tilings(n):
    f = [0] * (n + 1)
    g = [0] * (n + 1)

    f[1] = 1; f[2] = 2
    g[1] = 1; g[2] = 2

    for i in range(3, n + 1):
        f[i] = f[i - 1] + f[i - 2] + 2 * g[i - 2]
        g[i] = g[i - 1] + f[i - 1]

    return f[n]
```

Since we only require a single pass through each array to populate them based on previous values, the time and space complexity of this algorithm will be `O(N)`.