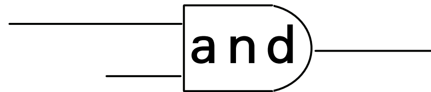
Mathematics
for the Digital Age



Programming
in Python

>>> Second Edition:
with Python 3

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts

Skylight Publishing
9 Bartlet Street, Suite 70
Andover, MA 01810

web: http://www.skylit.com
e-mail: sales@skylit.com
support@skylit.com

Copyright © 2010 by Maria Litvin, Gary Litvin, and Skylight Publishing

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the authors and Skylight Publishing.

Library of Congress Control Number: 2009913596

ISBN 978-0-9824775-8-8 (soft cover)
ISBN 978-0-9824775-4-0 (hard cover)

The names of commercially available software and products mentioned in this book are used for identification purposes only and may be trademarks or registered trademarks owned by corporations and other commercial entities. Skylight Publishing and the authors have no affiliation with and disclaim any sponsorship or endorsement by any of these products' manufacturers or trademarks' owners.

1 2 3 4 5 6 7 8 9 10 15 14 13 12 11 10

Printed in the United States of America

11 Recurrence Relations and Recursion

11.1 Prologue

In Chapter 1 we talked about different ways to define a function: in words, with a table, with a chart, or with a formula. But we left out one very important method: we can also describe a function, especially a function on positive (or non-negative) integers, *recursively*. Here is an example. Suppose you state that a function f with the domain of all positive integers is defined as follows:

$$f(n) = \begin{cases} 1, \text{ if } n = 1 \\ n \cdot f(n-1), \text{ if } n > 1 \end{cases}$$

This definition does not tell us right away how to find the value of $f(n)$. $f(n)$ is described in terms of $f(n-1)$ (except for one simple *base case*, $n=1$). And yet, with some work, we can find the value of $f(n)$ for any positive n. Let's see: we know that $f(1)=1$. Next: $f(2)=2\cdot f(1)=2\cdot 1=2$. Next: $f(3)=3\cdot f(2)=3\cdot 2=6$. Next: $f(4)=4\cdot f(3)=4\cdot 6=24$. And so on. There is only one function that satisfies the above definition. In this case you can easily find a formula to describe this function: it is our old friend $f(n)=n!$ (n-factorial).

Recursion is a powerful tool for defining functions. The designers of computer hardware and programming languages have made sure that recursion is also supported in programming languages.

Copyright © 2008-2010 by Skylight Publishing

11.2 Recurrence Relations

Recall that a function defined on the set of all positive integers can be expressed as a sequence of the function's values: $a_1, a_2, ..., a_n, ...$. To define such a function recursively, we define each term of the sequence, except the first (or, perhaps, the first few), through its relation to the previous term(s). The relation that connects the

n-th term to one or more of the previous terms is often the same for each n. It is called a *recurrence relation*. For example:

$$\begin{cases} a_1 = 1 \\ a_n = na_{n-1}, \text{ for any } n \geq 2 \end{cases}$$

We get, again, $a_n = n!$. This is the same as our recursive definition of $n!$, just rewritten in a slightly different way.

It is easy to calculate on a computer the values of a sequence defined through a recurrence relation. Recall, for example, our calculation of $s(n) = 1 + 2 + ... + n$ in Chapter 4. We noticed that $s(0) = 0$ and $s(n) = s(n-1) + n$, for any $n \geq 2$. This led to the following Python code:

```
n = 1
sum1n = 0
while n <= nMax:
    sum1n += n
    print('{0:3d} {1:6d}'.format(n, sum1n))
    n += 1
```

Here `sum1n` stands for $s(n)$. Iterations through the `while` loop are equivalent to calculating successive values of $s(n)$.

One of the famous sequences defined with a recurrence relation is called *Fibonacci numbers*:

$$f_1 = 1; \ f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}, \text{ if } n > 2.$$

This sequence is named after Leonardo Fibonacci (the name means son of Bonaccio) who lived in Pisa in the 13th century, but the numbers were known in India over 2000 years ago. Fibonacci came upon the numbers while he was modeling a population of rabbits under simplistic assumptions: we start with one adult pair; an adult pair produces a pair of baby rabbits each month; a baby rabbit becomes an adult after one month; the rabbits never die. This model results in the Fibonacci recurrence relation for the number of adult pairs after n months (Figure 11-1). The sequence goes like this: 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on. f_n represents the number of adult pairs of rabbits at month n. The total number of pairs of rabbits — adults and

Copyright © 2008-2010 by Skylight Publishing

babies — is also a Fibonacci sequence, but it starts at $p_1 = 1$; $p_2 = 2$, $p_n = p_{n-1} + p_{n-2}$, if $n > 2$.

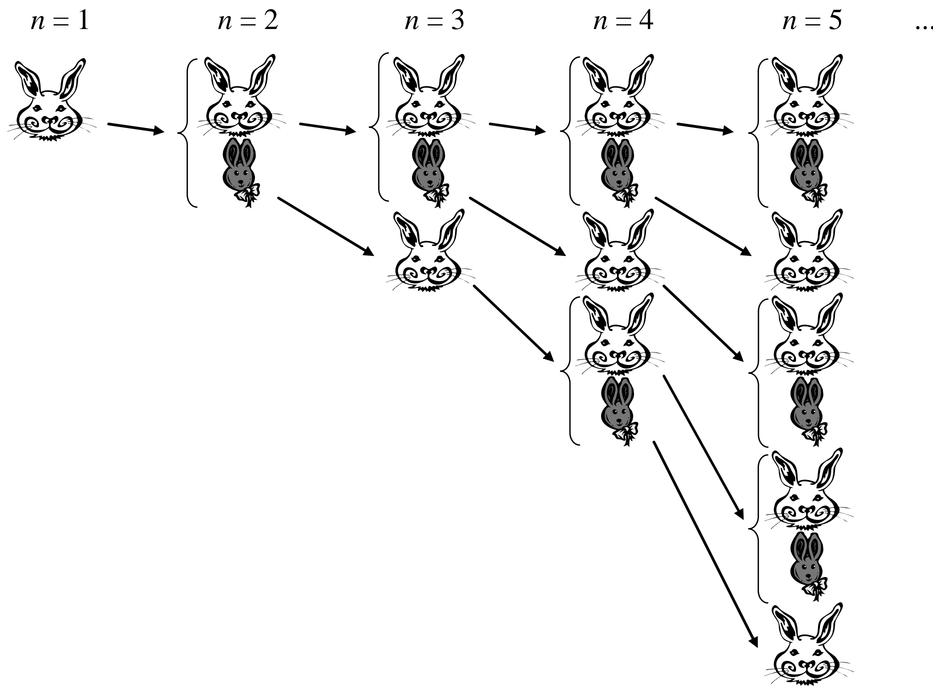


Figure 11-1. Rabbits' population growth in the Fibonacci model.

Fibonacci numbers have many interesting properties. For example, the sequence of ratios of consecutive Fibonacci numbers $r_n = \frac{f_{n+1}}{f_n}$ converges to the *golden ratio* $\tau = \frac{1+\sqrt{5}}{2} \approx 1.62$. Fibonacci numbers pop up in various branches of mathematics and in the natural world (Figure 11-2; search the Internet for "fibonacci + nature" for other examples).

Copyright © 2008-2010 by Skylight Publishing

Figure 11-2. Broccoli Romanesco. The numbers of clusters in the spirals form Fibonacci sequences.

Photograph by Madelaine Zadik, courtesy Botanic Garden of Smith College.

Copyright © 2008-2010 by Skylight Publishing

Exercises

1. Consider a function

$$f(n) = \begin{cases} 1, \text{ if } n = 1 \\ 2 \cdot f(n-1) + 1, \text{ if } n > 1 \end{cases}$$

What is $f(4)$? ✓

2. A function on positive integers is described by

$$f(n) = \begin{cases} 1, \text{ if } n = 1 \\ f(n-1) + 2, \text{ if } n > 1 \end{cases}$$

Redefine it by a simple non-recursive formula.

3. Define $f(n) = 2^n$ recursively, in terms of $f(n-1)$, for all integers $n \geq 1$. ✓

4.■ Write a recursive definition of a function $f(n)$ that has the following properties: $f(1)=10,\ f(2)=30$, and the values $f(1),\ f(2),\ ...,\ f(n),\ ...$ form an arithmetic sequence. ✓

5.■ The same as Question 4, but for a geometric sequence.

6.■ Describe the function from Question 1 with a simple formula, without recursion.

7.■ Define recursively, without factorials, the function $f(n)=\dfrac{n!}{(n-3)!}$ for integers $n \geq 3$. ✓

8. Given $a_1=1,\ a_n=a_{n-1}+n$ for any $n \geq 2$, describe a_n with a simple non-recursive formula in terms of *n*.

9.■ Write a Python program that prompts the user to enter a positive integer *n* and prints the first *n* Fibonacci numbers.

10.◆ Find a sequence that satisfies the Fibonacci equation $a_n=a_{n-1}+a_{n-2}$, if $n>2$ and, at the same time, is a geometric sequence with $a_1=1$. How many such sequences are there?

11.3 Recursion in Programs

Now back to our favorite, $s(n)=1+2+...+n$. As we have seen, the recursive definition of $s(n)$ is $s(n)=\begin{cases}1, \text{ if } n=1 \\ s(n-1)+n, \text{ if } n>1\end{cases}$. Now suppose, just out of curiosity, we implement this definition directly in a Python function:

```
def sum1toN(n):
    if n == 1:
        return 1
    else:
        return sum1toN(n-1) + n
```

Copyright © 2008-2010 by Skylight Publishing

Will this work? What will `sum1toN(5)` return? As you know, in a program, a function translates into a piece of code that is callable from different places in the program. The caller passes to the function some argument values and a return address — the address of the instruction to which to return control after the function finishes its work. So how can this work when a piece of code passes control to itself? Won't the program get confused? Won't Python choke trying to swallow its own tail?

And yet it works! Try it:

```
>>> sum1toN(5)
15
```

Because recursion is such a useful tool in programming, the developers of Python (and other programming languages) have made special efforts to ensure it works. There are also CPU instructions that facilitate implementing recursive calls.

Here is an allegory for what happens. Imagine a function as a mail-order bakery. To place an order you take an empty box, put your written order into it and also put in a label with your address, and ship it to the bakery. The baker, call him Frank, receives the box, reads the order (the arguments you pass to the function), bakes the cake you ordered, puts it in the box, attaches your label, and ships the box back to you.

Now suppose Frank has received an order and is working on it. Meanwhile a new order arrives. Frank immediately drops what he is doing, saves the unfinished cake (the values of the local variables) in the box the order came in, then puts the new box that has just arrived <u>on top</u> of the one he was working on, opens the new box, and starts working on that. Meanwhile a new order arrives. Frank immediately drops what he is doing, saves the unfinished cake (the values of the local variables) in the box the order came in, then puts the new box that has just arrived <u>on top</u> of the one he was working on, opens the new box, and starts working on that. Meanwhile... You get the picture. The orders keep coming and the boxes keep piling up: now there is a whole *stack* of them.

Finally, at some point, Frank gets a break: the orders stop coming. Frank finishes the top order and ships it to the customer. He then finishes the one that was just below it and ships that to the customer. And so on, until he gets to the bottom of the stack, to the last remaining order, finishes that order, and goes home.

You might be wondering: Why is Frank processing orders in this funny sequence? The order that came in last is completed first. Wouldn't it be more fair to do it in a first-in-first-out manner? In a moment you will see why Frank does it his way.

Copyright © 2008-2010 by Skylight Publishing

One day, Frank received an order for a multi-layered wedding cake with four layers. The layers in such a cake are basically the same, only each layer is smaller in diameter than the one beneath it. Frank started working on the bottom layer, but began to worry that he'd get confused in all the layers. Luckily, a friend of his (a computer programmer) came up with this fabulous idea: why doesn't Frank send an order to himself for a multi-layered cake that consists of the three top layers, and then, when the three-layer cake is ready, just put it on top of the layer he is working on? Frank doesn't have to ship his order very far, of course, but he still follows his usual procedure: takes an empty box, writes out an order for a three-layer cake, writes a label with the return address (his own) and sends in the order (by putting the box on the top of the stack). Then as usual, he opens the box and sees: aha, an order for a three-layer cake! So he writes himself an order for a two-layer cake. You get the picture. Finally Frank receives (his own) order for a single-layer cake. He finishes it quickly and ships it back (to himself, of course). He receives it, uses it to complete the two-layer cake, and ships that back (to himself). Receives it, completes the three-layer cake, ships back, receives that, completes the four-layer cake, and finally ships it to the customer. Due to his LIFO (last-in-first-out) method, Frank untangles the whole layered business without any confusion.

This seems like a lot of shipping back and forth, but it is easy for a computer. Python allocates a special chunk of memory, called a *stack*, for recursive function calls. The CPU has the `push` instruction that puts a data item onto the top of the stack. The complementary `pop` instruction removes and returns the top item.

In Python, Frank's activity may be represented as something like this (assuming the + operator is properly defined for cakes):

```
def bakeSingleLayeredCake(d):
    'Bakes a simple cake of diameter d'
    return Cake(d)

def bakeMultiLayeredCake(n):
    'Bakes a cake with n layers'
    d = 3 * n  # the diameter of the bottom layer in inches
    cake = bakeSingleLayeredCake(d)
    # Base case: n == 1 -- do nothing
    # Recursive case:
    if n > 1:
        cake += bakeMultiLayeredCake(n - 1)
    return cake
```

For a recursive function to work, it must have a simple *base case*, where recursive calls are not needed.

Copyright © 2008-2010 by Skylight Publishing

In the above code, `n == 1` is the base case.

> **When a function is called recursively, it must be called for smaller and smaller tasks, which eventually converge to the base case (or one of the base cases).**

Each time we call the function `bakeMultiLayeredCake(n-1)` recursively, we call it with an argument that is smaller by 1 than the previous time.

Recursion has its cost: first, you need to have enough space on the stack and, second, the code spends extra time pushing and popping data and calling the same function. Iterations are often more efficient. But some applications, especially those dealing with nested structures or branching processes, require a stack anyway, and recursion allows you to write very concise and clear code.

Example 1

Write a recursive function that calculates 10^n for a non-negative integer n.

Solution

```
def pow10(n):
    if n == 0:  # base case
        return 1
    else:
        return pow10(n-1) * 10
```

Example 2

Write a recursive function that returns the sum of the digits in a non-negative integer.

Solution

```
def sumDigits(n):
    s = n % 10
    # Base case: n < 10 -- do nothing
    # Recursive case:
    if n >= 10:
        s += sumDigits(n // 10)
    return s
```

Copyright © 2008-2010 by Skylight Publishing

Example 3

Write a recursive function that takes a string and returns a reversed string.

Solution

```
def reverse(s):
    if len(s) > 1:
        s = reverse(s[1:]) + s[0]
    return s
```

The base case here is implicit: when the string is empty or consists of only one character, there is nothing to do.

A very popular example of recursion in computer programming books and tutorials is the "Towers of Hanoi" puzzle. We have three pegs and n disks of increasing size. Initially all the disks are stacked in a tower on one of the pegs (Figure 11-3). The objective is to move the tower to another peg, one disk at a time, without ever placing a bigger disk on top of a smaller one.

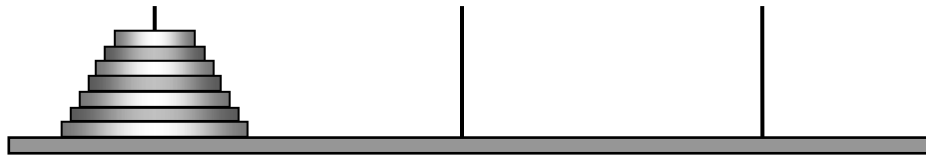


Figure 11-3. The Towers of Hanoi puzzle

The key to the solution is to notice that in order to move the whole tower, we first need to move a smaller tower of $n-1$ disks to the spare peg, then move the bottom disk to the destination peg, then move the tower of $n-1$ disks from the spare peg to the destination peg (Figure 11-4).

Copyright © 2008-2010 by Skylight Publishing

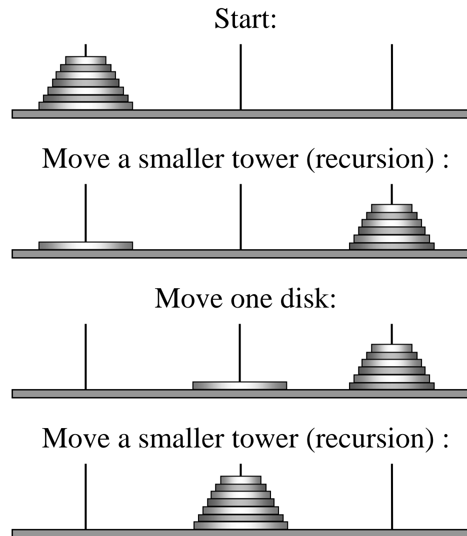


Figure 11-4. A recursive solution to Towers of Hanoi

Copyright © 2008-2010 by Skylight Publishing

Example 4

Write a recursive function that solves the Towers of Hanoi puzzle for *n* disks.

Solution

```
def moveTower(nDisks, fromPeg, toPeg):
    sparePeg = 6 - fromPeg - toPeg
    if nDisks > 1:
        moveTower(nDisks - 1, fromPeg, sparePeg)

    print('From ' + str(fromPeg) + ' to ' + str(toPeg))
    # This code does not model disk movements
    # -- it just prints out each move

    if nDisks > 1:
        moveTower(nDisks - 1, sparePeg, toPeg)
```

If we want to move a tower of, say, seven disks from peg 1 to peg 2, an initial call to `moveTower` would be `moveTower(7, 1, 2)`.

Note that in the above example the function does not return a value (more precisely, it returns `None`), so it is a *recursive procedure*. Programming this task without recursion would be rather difficult.

Exercises

1. Write and test a recursive function that returns $n!$. Do not use any loops. ✓

2. In Section 5.3 Question 7, you created a function `intToBin(n)` that takes a non-negative integer and returns a string of its binary digits. For example, `intToBin(5)` returns `'101'`. Rewrite this function recursively, without any loops. ⦃ Hint: implement two base cases: `n == 0` and `n == 1`: they cover the situations when `n` has only one binary digit. ⦄

3. Write and test a function

```
def printDigits(n):
```

that displays a triangle with *n* rows, made of digits. For example, `printDigits(5)` should display

```
55555
4444
333
22
1
```

The function's code under the `def` line should be three lines. ⦃ Hint: `print n*str(n)` prints '*n*' *n* times. ⦄ ✓

4. The same as Question 3, but invert the triangle, so that `printDigits(5)` prints

```
1
22
333
4444
55555
```

5. Explain the statement `sparePeg = 6 - fromPeg - toPeg` in Example 4. ✓

Copyright © 2008-2010 by Skylight Publishing

6. Identify the base case in the `moveTower` function in Example 4. ✓

7. Without running the code in Example 4, determine the output when `moveTower(3, 1, 2)` is called.

8. The Towers of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. The "legend" that accompanied the puzzle stated that in Benares, India, there was a temple with a dome that marked the center of the world. The Hindu priests in the temple moved golden disks between three diamond needles. God placed 64 gold disks on one needle at the time of creation and the universe will come to an end when the priests have moved all 64 disks to another needle. What is the total number of moves needed to move a tower of 2 disks? 3 disks? 4 disks? *n* disks? Assuming one move per second, estimate the lifespan of the universe. ✓

11.4 Mathematical Induction

Let d_n be the number of moves required to move a tower of *n* disks in the Towers of Hanoi puzzle. It is pretty obvious that, for $n > 1$, $d_n = d_{n-1} + 1 + d_{n-1} = 2d_{n-1} + 1$, because to move a tower of *n* disks, we need to first move a tower of $n-1$ disks, then one disk, then again a tower of $n-1$ disks. If you finished the last exercise, you probably guessed correctly that $d_n = 2^n - 1$. How can we prove this fact more rigorously? In questions of this kind we can use the method of proof called *mathematical induction*.

Suppose you have two sequences, $\{a_n\}$ and $\{b_n\}$. Suppose that:

1. $a_1 = b_1$ (base case).
2. For <u>any</u> $n > 1$ you have managed to establish the following fact: <u>if</u> $a_{n-1} = b_{n-1}$ <u>then</u> $a_n = b_n$.

Is it true then that $a_n = b_n$ for all $n \geq 1$?

Let's see: we know that $a_1 = b_1$. We know that if $a_1 = b_1$ then $a_2 = b_2$. So we must have $a_2 = b_2$. Next step: we know that $a_2 = b_2$. We know that if $a_2 = b_2$ then $a_3 = b_3$. So we must have $a_3 = b_3$. Next step: ... and so on. We have to conclude that $a_n = b_n$ for all $n \geq 1$. In the future, we do not have to repeat this chain of

Copyright © 2008-2010 by Skylight Publishing

reasoning steps every time — we just say our conclusion is true "by mathematical induction."

Example 1

Prove that the number of moves in the Towers of Hanoi puzzle for n disks $d_n = 2^n - 1$.

Solution

d_n satisfies the relationship

$$d_1 = 1,$$
$$d_n = 2d_{n-1} + 1 \text{ for } n > 1$$

We want to show that $d_n = 2^n - 1$ for all $n \geq 1$.

1. First let's establish the base case, $n = 1$. $d_1 = 1$ and $2^1 - 1 = 1$, so $d_1 = 2^1 - 1$. OK.

2. Now let's proceed with the *induction step*. Assume that $d_{n-1} = 2^{n-1} - 1$. Then $d_n = 2d_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$. So, for any $n > 1$, we were able to show that if $d_{n-1} = 2^{n-1} - 1$ then $d_n = 2^n - 1$.

By mathematical induction, $d_n = 2^n - 1$ for all $n \geq 1$, Q.E.D.

The method of mathematical induction applies to any sequence of propositions $\{P_n\}$. Suppose we know that P_1 is true; suppose we can show for any $n > 1$ that if P_{n-1} is true then P_n is true. Then, by mathematical induction, P_n is true for all $n \geq 1$. (In the above example, P_n is the proposition that $d_n = 2^n - 1$.)

Copyright © 2008-2010 by Skylight Publishing

Sometimes it is necessary to use a more general version of mathematical induction, with several base cases and a more general induction step. Suppose that:

1. $P_1, ..., P_k$ are true (base case(s)).
2. For any $n > k$ we can prove that if $P_1, P_2, ..., P_{n-1}$ are all true, then P_n is true.

Then, by mathematical induction, P_n is true for all $n \geq 1$.

Example 2

Prove that the *n*-th Fibonacci number $f_n \geq \left(\frac{3}{2}\right)^{n-2}$.

Solution

1. We can establish two base cases:

$$f_1 = 1 \geq \frac{2}{3} = \left(\frac{3}{2}\right)^{-1} = \left(\frac{3}{2}\right)^{1-2}$$

$$f_2 = 1 \geq \left(\frac{3}{2}\right)^{0} = \left(\frac{3}{2}\right)^{2-2}.$$

2. Now the induction step. Assume that for any $m < n$ $f_m \geq \left(\frac{3}{2}\right)^{m-2}$. In particular, for $n > 2$, $f_{n-1} \geq \left(\frac{3}{2}\right)^{n-3}$ and $f_{n-2} \geq \left(\frac{3}{2}\right)^{n-4}$. Then

$$f_n = f_{n-1} + f_{n-2} \geq \left(\frac{3}{2}\right)^{n-3} + \left(\frac{3}{2}\right)^{n-4} = \left(\frac{3}{2}\right)^{n-4}\left(\frac{3}{2}+1\right) =$$

$$\left(\frac{3}{2}\right)^{n-4} \cdot \frac{5}{2} > \left(\frac{3}{2}\right)^{n-4} \cdot \frac{9}{4} = \left(\frac{3}{2}\right)^{n-2}$$

So the proposition is true for the two base cases, and if it is true for $n-1$ and $n-2$ then it is also true for *n*. By math induction, it is true for all $n \geq 1$, Q.E.D.

Copyright © 2008-2010 by Skylight Publishing

If you finished Question 9 from Section 11.2, you probably tried to print the first 100 Fibonacci numbers and got $f_{100} = 354224848179261915075$. The above result explains why Fibonacci numbers grow so fast.

Exercises

1. Prove using mathematical induction that $1+3+5+...+(2n-1)=n^2$. ✓

2.■ Consider the following function:

```
def tangle(s):
    n = len(s)
    if n < 2:
        return ''  # empty string
    else:
        return tangle(s[1:n]) + s[1:n-1] + tangle(s[0:n-1])
```

Show that `tangle('abcde')` returns a string that contains neither `'a'` nor `'e'`.

3.◆ Show that for a string `s` of length *n*, `tangle(s)`, defined in Question 2, returns a string of length $L_n = 2^{n-1} - n$. { Hint: first obtain a recurrence relation for L_n, then prove it using mathematical induction. }

4.◆ Suppose you have *n* straight lines in a plane, such that no two lines are parallel to each other and no three lines go through the same point. Show that the number of regions into which these lines cut the plane depends only on *n*, but not on a particular configuration of the lines. Find a formula for that number and prove it using mathematical induction. { Hint: when you add a line, the existing lines cut it into pieces. Each of these pieces cuts a region into two, adding a new region. } ✓

5.◆ Consider the sequence $a_0 = 2,\ a_1 = 2,\ a_n = 2a_{n-1} + 3a_{n-2}$ for $n \geq 2$. Write a Python program that prints out the first *n* terms of this sequence. Examine the pattern and come up with a non-recursive formula for a_n, then prove that your guess is correct using mathematical induction. { Hint: compare a_n to 3^n. }

Copyright © 2008-2010 by Skylight Publishing

6.◆ Consider a recursive version of the function that returns the *n*-th Fibonacci number:

```
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Test it with $n = 6$ and $n = 20$. Now test it with $n = 100$. The computer won't return the result right away. Perhaps the recursive version takes a little longer... Prove by mathematical induction that the total number of calls to `fibonacci`, including the original call and all the recursive calls combined, is not less than the *n*-th Fibonacci number f_n. We have shown above that $f_n \geq \left(\frac{3}{2}\right)^{n-2}$. Assuming that your computer can execute one trillion calls per second, estimate how long you will have to wait for `fibonacci(100)` (in years). Press Ctrl-C to abort the program. As we said earlier, sometimes recursion can be costly.

Copyright © 2008-2010 by Skylight Publishing

11.5 Review

Terms introduced in this chapter:

Recurrence relation
Fibonacci numbers
Golden ratio
Recursion
Recursive function
Recursive procedure
Base case
Mathematical induction

Python feature introduced in this chapter:

Recursive functions