The Beauty and Joy of Computing[1]

Lab Exercise 9: Problem self-similarity and recursion - Python version

Objectives

By completing this lab exercise, you should learn to

- Recognize simple self-similar problems which are amenable to recursive solutions;
- Identify components of recursion: base case and recursive case;
- Build simple recursive functions in guided settings; and
- Give rough arguments as to why recursive solutions work.

Background (Pre-Lab Reading)

You should read this section before coming to the lab. It describes several important concepts, and provides example Python code similar to what you will be expected to work with and write during the lab. You only need to read this material to get familiar with it. There is no need to actually do the actions described in this section, but you certainly *can* do them if it would help you follow along with the examples.

A **recursive algorithm** is an algorithm that implements a function by *calling itself* in a controlled way. The general process of a function calling itself is called **recursion**. Recursion is an extremely powerful computing technique, but can be confusing for beginning programmers (and sometimes even for experienced programmers!). However, once you learn to "think recursively" there are a huge number of problems that have natural recursive algorithms or recursive ways of expressing a solution. In computer science there are problems for which recursion leads to highly efficient algorithms; there are also problems for which recursion is the *only* readily apparent way to solve the problem; there are important data structures that are recursively built; and there are powerful algorithmic techniques like dynamic programming in which the first step is to recursively characterize the solution to the problem. All this is just to point out how important and central recursion is to computer science – while you may not be particularly comfortable with recursion after the light coverage in this class, if you take more computer science classes this concept will keep coming up over and over. You will eventually (hopefully!) gain a deeper understanding of how to use recursion, and in this lab you will start your journey to understanding recursion by doing some basic and carefully guided examples.

We start by demonstrating the concepts of recursion and the techniques for designing a recursive algorithm with a simple example: computing the factorial function. As you hopefully remember from math classes, n factorial (written n!) is the product of the first n positive integers:

$$n! = 1 * 2 * 3 * 4 * \ldots * (n-1) * n$$

So looking at a few examples, 2! is 2, 3! is 6, and 5! is 120. The factorial function exhibits a property that we call self-similarity: the solution for n! "contains" solutions for smaller versions of

[1] *Lab Exercises for "The Beauty and Joy of Computing"*

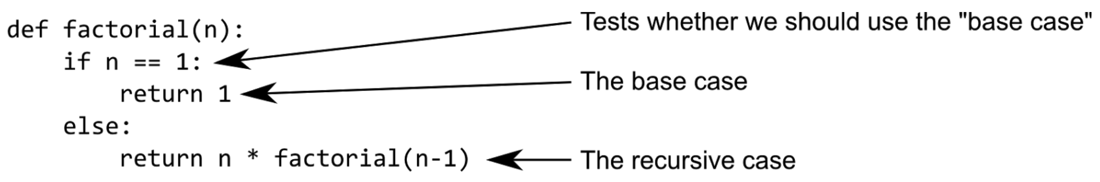
the same problem. Look at the formula above – if we simply leave out the last term (the last factor of *n*), then we have (*n*-1)!. In fact, this shows us that for large enough *n* ("large enough" so that there are enough terms in the formula to split it like this), we see that *n*! = (*n*-1)! * *n*.

So for "large enough" *n*, we can write the solution for *n*! using the solution for (*n*-1)! and a small amount of additional work (one additional multiplication). What does "large enough" mean? In this case, it just means that *n* is at least 2 (so there is something to the left of the last term). If *n* is not "large enough" to apply the recursive solutoin, then we can just write down the answer. In this case, if *n*=1, we can just say that *n*! is 1.

This is how recursion always works: we consider two cases, depending on whether the input is large enough to express the answer in terms of solutions to smaller problems, or whether it is so small that we can't break it down but can write the solution out directly. We call these the **recursive case** and the **base case**, and identifying these two cases is the first step in designing a recursive algorithm. Writing this out explicitly, for the factorial problem we might start like this:

- *Recursive case*: When *n* ≥ 2, we know that *n*! = (*n*-1)! * *n*
- *Base case*: When *n* = 1, we know that *n*! = 1

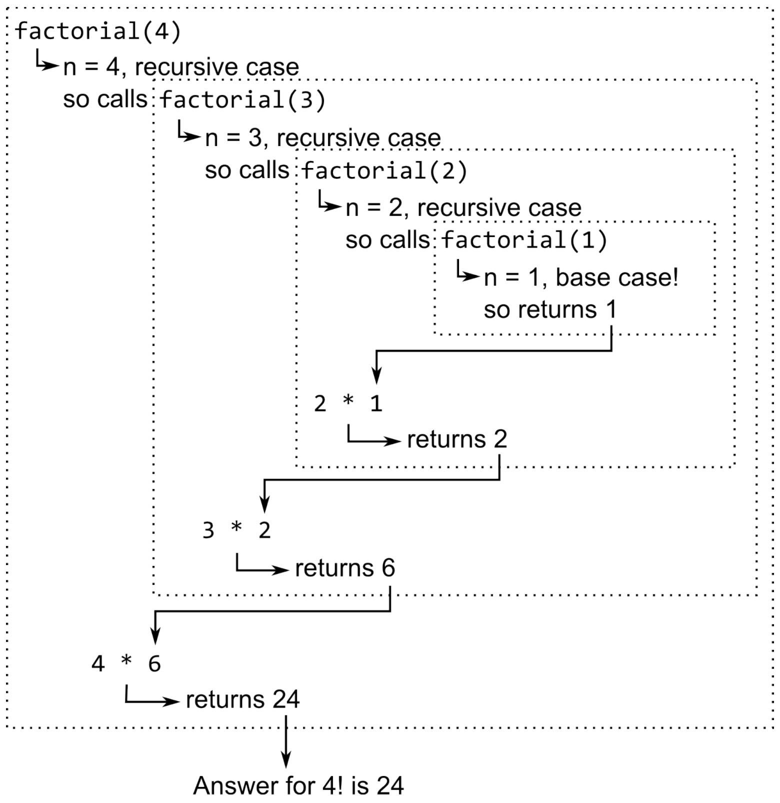
Now comes the magic part: we can write out code based on these observations. First, we use an if/else statement to test whether we are in the base case or in the recursive case, and then put in code for each case. While we described the recursive case first above, typical practice in programming is to test for and handle the base case first, and so the result is the following Python function definition to compute the factorial function:



```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Look carefully at the formula in the last return statement (in the "else" part of the if/else). We are calling the factorial function (the function we are defining) with an argument of `n-1`. Assuming our recursive call magically works (and it does), it gives (*n*-1)! which we then multiply by *n* and return that value.

It's worthwhile to trace out how this works, but it should be stressed that tracing out recursive functions is *not the right way to think about recursion*! We'll do this once, for this one example, and once you are reassured that this actually works then never do it again! The right way to think about recursion is just building the base case and the recursive case. Thinking about the recursive case, just think about one step – don't keep tracing beyond a single step to a smaller problem, because if you think across multiple steps then you're really missing the whole point of recursion. That said, here's how the factorial function defined above works – first, calling factorial for argument 4, this is the recursive case so we end up calculating n-1 (which is 3) and calling factorial with argument 3. This calls factorial of 2, and finally that calls factorial of 1. We stop making recursive calls at this point since we're in the base case. Our functions now return, and at each step multiplying the result of the recursive call with the value of n at that level. We can visualize this as follows:



Let's reason about why this works. We start off by asking, if I call "`factorial`" with an argument of 4, does it give the right answer? Well, we call `factorial(3)` and multiply that result by 4, so as long as factorial of 3 is computed correctly then we get the right answer. Do we compute factorial of 3 correctly? Well, to compute factorial of 3 we compute factorial of 2 and multiply that by 3, so as long as we compute factorial of 2 correctly then factorial of 3 is correct (which will make factorial of 4 correct!). Do we compute factorial of 2 correctly? Well, to compute factorial of 2 we compute factorial of 1 and multiply that by 2, so as long as we compute factorial of 1 correctly then factorial of 2 is correct (which will make factorial of 3 and hence factorial of 4 both correct). Finally, do we compute factorial of 1 correctly? Yes! That's our base case, so the correct value is simply filled in – so all intermediate results are correct, giving us the correct value for `factorial(4)`. That was a somewhat tedious argument, but if you think about how the correctness builds from smaller cases to bigger cases it should make sense. In later computer science (or math) classes this kind of argument is formalized in what is called a **proof by induction** – a powerful way of reasoning about the correctness of statements and algorithms.

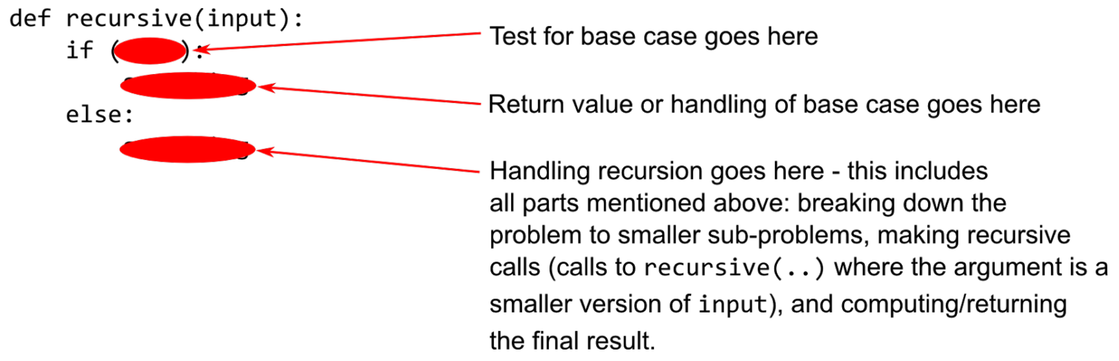
Let's go back and look at this specific example and think about the design process for coming up with this recursive algorithm. In designing a simple recursive function, we need to design the two specific parts of the recursion, which requires answering the following questions:

1. *What is the recursive step?* The recursive step will do three things: break down the problem to one or more smaller versions of the same problem, called the subproblem(s)

(we've only seen a single subproblem in our example, but there might be more, like in Activity 3 below); make the appropriate recursive call(s); and process the results of the recursive calls to produce our answer. In the factorial case, the only computation that had to be done to "break down" the problem of computing *n* factorial to a smaller sub-problem was to compute *n*-1; then we made the recursive call; then we did the post-recursion processing, which in this case was just taking the result of the recursive call and multiplying by *n*. In deciding on the recursive step, it is best to picture in your mind a value of *n* that is not very small – for example, don't think about recursion when *n* is 2 – think about how the recursion works when *n*=10 or *n*=100.

2. *What is the base case?* The recursive step should result in a recursive call with a smaller argument than the original argument – for example, in the factorial problem we make a recursive call with argument *n*-1. Each recursive step makes the input argument smaller and smaller, and so at some point you reach the smallest possible problem, which you must solve directly. This is the base case, and there are two parts to the base case. First, what defines the base case and how can you test if your input falls into the base case rather than the recursive case? In the factorial example, we are writing a factorial function that handles inputs *n* for all $n \geq 1$, so the smallest possible input is $n = 1$, which defines the base case (and gives the test for being in the base case). The second thing we must figure out is what the value of the function is in the base case. For the factorial problem, when $n = 1$, we know that *n*! is 1, so that's our answer.

Just like iteration over lists, there is a pattern we can use for basic recursion, which looks like this:



```
def recursive(input):
    if (    ):

    else:

```

Test for base case goes here

Return value or handling of base case goes here

Handling recursion goes here - this includes all parts mentioned above: breaking down the problem to smaller sub-problems, making recursive calls (calls to `recursive(..)` where the argument is a smaller version of `input`), and computing/returning the final result.

Lets do one more numerical example of recursion, which will be very similar to what you will do in activities 1 and 2 below. Let's say you'd like to define a function that can perform powering: we'd like to be able to raise numbers to powers that are non-negative integers. In other words, we are interested in being able to compute things like 2^3, 3^2, 2^9, or even $(2.5)^3$ or $(1.25)^8$, but not something with non-integer exponents like $5^{1.5}$. Expressed as a general formula, we want to compute x^y, where *y* is an integer. Let's ask and answer the questions given above, and then we can fill in our code pattern to get the final recursive algorithm.

What is the recursive step? One thing that makes this problem a little more difficult is that we have two numbers that are in the input, so we have to decide whether to make one of them smaller or both of them smaller in the recursion (some recursive solutions are even more complicated than this, but you would never need anything complex for this class). In this case, there is a clear answer if you think about powering as repeated multiplication (so x^4=x*x*x*x): we will do one less multiplication for the sub-problem, so we reduce the exponent. Now we need to write out x^y in terms of $x^{\text{something smaller than } y}$. There's actually more than one way to do this, but for now we'll just think of "peeling off" one multiplication so that we use $x^y = x * x^{y-1}$. This has defined our recursive step! The pre-recursion processing is computing y-1, the recursive call is computing x^{y-1}, and the post-recursion processing is multiplying x by the result of the recursive call.

What is the base case? In this situation, we are computing powers in which the exponent y gets smaller and smaller, so we ask "What is the smallest value of y for which this problem makes sense?" Since we said above that the power will be a non-negative integer, the answer in this case is $y = 0$ – we might ask what x^0 is for some x, and we can in fact easily answer that since x^0 is 1 for all values of x (let's ignore the slightly strange case of 0^0). So now we have answered this question: we can tell if we're in the base case by testing if $y = 0$, and we can handle the base case by returning 1.

Now we plug our answers into the pattern to get the following final recursive algorithm definition:

```
def power(x, y):
    if y == 0:
        return 1
    else:
        return x * power(x, y-1)
```

This is our final recursive function to compute powers. Testing it out with some simple values, we see that it works quite well!

Self-Assessment Questions

Use these questions to test your understanding of the Background Reading. Answers can be found at the end of the lab exercise.

1. Back in Lab 4, one of the samples we looked at was creating a loop to add numbers in a particular range. Consider a simplified version of this problem (converted to Python!), where we always start at 1, and so we create a function named "`sum_of_first_ints(n)`" to add up 1+2+3+...+n and return the answer (defined for n ≥ 1). If we wanted to compute this recursively, what is the base case, and what is the value to return in the base case?
2. What is the recursive case, and what is the computation we perform in the recursive case?
3. Put the answers to questions 1 and 2 together to write a recursive function definition for this problem.
4. If we execute this script with an argument of 15 (in other words, we call "`sum_of_first_ints(15)`"), what is the total number of recursive calls that are made?

5. Consider the following function definition (this is a little different from the normal recursion pattern):

```
def mystery(n):
    if (n % 2) == 1:
        return False
    if n == 2:
        return True
    else:
        return mystery(n / 2)
```

   This is a Python predicate (so always returns True or False): What will this predicate return when called with "`mystery(3)`"? What about "`mystery(6)`"? What about "`mystery(8)`"?

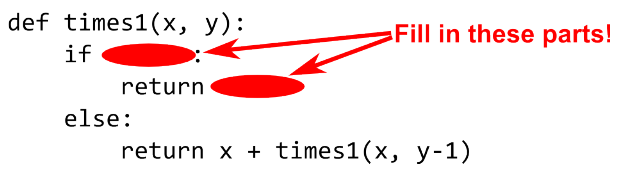
6. (Warning: This question requires some good mathematical intuition – don't let that scare you away from trying to answer it, but don't feel too bad if you don't get this one.) Describe in simple terms what the predicate in question 5 is testing.

Activities (In-Lab Work and Post-Lab Bonus Activity)

All of the following activities (Activities 1 through 3) can and should be saved as functions within a single file, named "Lab9.py" – just add new functions for each activity.

Activity 1: Early microprocessors in 1970's and early 1980's, like the Intel 8080, Motorola 6800, and early Sun SPARC processors, did not include hardware or instructions to multiply integers, so programmers supplied functions to do multiplication using an algorithm made out of simpler operations. While things are much simpler today since microprocessors are more powerful, this problem illustrates some important aspects of recursion, so we'll look at it in this activity. Practically identical issues arise in modern code for doing large modular exponentiations in cryptography, but we'll focus on the simpler setting of multiplication for this lab.

In the pre-lab reading we treated powering as repeated multiplication, and in this activity we will do something very similar but treating multiplication as repeated addition. For example, if we want to compute *x**4, we can write this as *x*+*x*+*x*+*x*, accomplishing the multiplication with four additions. In general, if we are interested in computing *x***y*, where *y* is a non-negative integer, we can write *x***y* = *x**(*y*-1) + *x*. This defines our recursive step, and the outside the recursive call to compute x*(y-1) all we need are simple addition and subtraction. What you are to do in this activity is write a "`times1`" function that implements this recursive algorithm (note the 1 in the name – we'll make a "`times2`" in Activity 2, so naming the functions like this will keep them separate). The following picture shows the function definition with the recursive part filled in:


```
def times1(x, y):
    if      :
        return
    else:
        return x + times1(x, y-1)
```

You need to think about the base case: what is the appropriate base case (what value of y), and what should you return in the base case? Once you have written this code, test it on some simple values and convince yourself that it works. Also double-check and make sure you're not "cheating" and doing multiplications. There should be no '*' operations in your code! Make sure you've got all this code saved in Lab9.py.

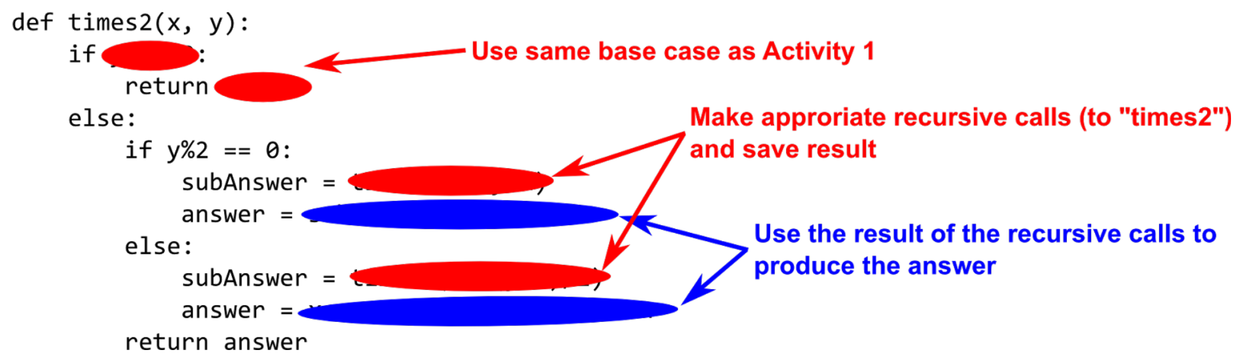
Activity 2: In this activity, we'll consider a different way to handle the recursive case. The time complexity of a recursive algorithm depends strongly on how much we reduce the problem at each recursive step, which determines how fast we can reach the base case, and just decreasing it by one is not a particularly fast way to get to the base case. For multiplication, it turns out that we can do a recursive case that roughly halves the parameter *y* at each step, rather than just reducing it by 1. So, for example, if we were to call this with *y* = 10,000, then our solution in activity 1 will make a recursive call with *y*=9,999 and then 9,998, and then 9,997 and so on; however, this new technique will make a recursive call with *y*=5,000 and then 2,500, and then 1,250 and so on. Note that it looks like we're multiplying and dividing in the following algorithm, even though we said we couldn't multiply – isn't that a violation of our restrictions? Not in this case: all multiplications are doing multiplication by 2 and divisions are dividing by 2 – these are really simple operations on binary numbers, just like multiplying and dividing by 10 are trivial in decimal. Multiplying a binary number by two just shifts all the bits left one position, and dividing by two shifts all the bits to the right, so these operations are allowed.

The key to this is to think about what you can do if *y* is even. In that case, we notice that $x*y = 2*(x * (y/2))$. So if we made a recursive call to compute $x * (y/2)$ and saved that result in a variable, we could multiply that by two (or add that variable to itself) to get $2 * (x * (y/2)) = x*y$. Note that if you add the recursive call to itself, it is extremely important to use a variable – if you were to make two recursive calls to add together, rather than using a variable, this would be slower. Really, really, *<u>really</u>* slower. Take my word for it.

What if *y* is odd? In that case, we can reduce *y* by 1 so that it's even, pull the same trick to get $2*(x * (y-1)/2) = x*(y-1) = x*y - x$, where $x * (y-1)/2$ is a recursive call, and add *x* to the result to get the answer that we need. In other words, when *y* is odd, we can compute $2*(x * (y-1)/2) + x = x*y$. If we write this multiplication algorithm out as mult(*x*,*y*), then the way to express this as a mathematical expression is as follows:

$$\begin{array}{ll} \text{mult}(x,y) = 2 * \text{mult}(x, y/2) & \text{if } y \text{ is even;} \\ \text{mult}(x,y) = 2 * \text{mult}(x, (y-1)/2) + x & \text{if } y \text{ is odd.} \end{array}$$

Your job is to take that description and formula, and fill in the missing pieces in this code for "`times2`", as shown at the top of the next page.

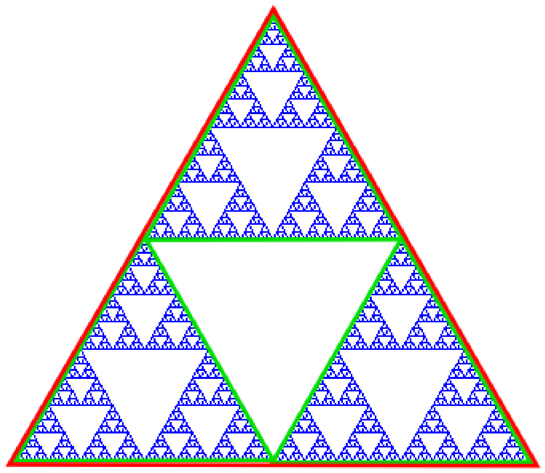


```
def times2(x, y):
    if [redacted]:
        return [redacted]
    else:
        if y%2 == 0:
            subAnswer = [redacted]
            answer = [redacted]
        else:
            subAnswer = [redacted]
            answer = [redacted]
        return answer
```

Use same base case as Activity 1
Make approriate recursive calls (to "times2") and save result
Use the result of the recursive calls to produce the answer

Make sure you test this thoroughly to make sure it works correctly for a variety of *y* values (at the very least, make sure you try some even and some odd values). For this activity, just get everything working and make sure it's saved in Lab9.py. If you finish the next activity, you'll get back to one more activity (Activity 4) that will compare the speed of these two algorithms (`times1` and `times2`), but first we'll do a completely different recursion exercise.

Activity 3: There are several cool looking self-similar shapes that people have come up with over the years, and recursion is a great way to draw these figures. This is more complex than the previous activities, but if you take it a small step at a time you should be able to do this. The shape you're going to draw is a Sierpinski triangle, which looks like this (the red and green outlines are not part of the drawing, but are marking parts of it for this discussion):



So what exactly is a Sierpinski triangle? A Sierpinski triangle is a triangle made by drawing three smaller Sierpinski triangles and placing them next to each other as shown: two side by side and one on top of those two. In the diagram, the three smaller Sierpinski triangles are outlined in green, and after combining they make a larger Sierpinski triangle, outlined in red. Think how strange this definition is: To draw a Sierpinski triangle, you draw three Sierpinski triangles. That's a recursive definition! To do this in Python, we need to be able to draw lines. Fortunately, this is much easier than the general animation tasks that we did in the last lab, and we can use the "`turtle`" module that is part of the standard Python distribution. Make sure you include

```
import turtle
```

at the top of your program file, and we'll look at the few functions you need next.

The `turtle` module is a Python implementation of "Turtle Graphics," a simple but powerful idea that dates back to the educational software language Logo in the late 1960's. The ideas should be familiar to you, because this is exactly what the "pen" blocks in BYOB are patterned after! The idea is that there is a (virtual, of course) turtle on the screen that has a current position and heading (direction), and has a pen associated with it (the pen can be up or down, and has a settable color and width). When you drew "flying Alonzo"'s trajectory in Lab 2 you were actually using turtle graphics commands, where Alonzo was the "turtle."

In the Python turtle graphics module, the turtle looks like an arrow by default. If you import the turtle module and then type the command "`turtle.reset()`" into the Python Shell, you'll see a graphics window pop up and the "turtle" will be in the middle, looking like this:

Try it! The basic drawing functions that you'll need in this lab are given below, so take a little time to experiment with these and see what you can draw:

<u>Basic Python "turtle" functions:</u>

```
turtle.clear()             - clear the screen
turtle.goto(x, y)          - go to a specific location
turtle.forward(distance)   - move forward a specified distance (like the "move" BYOB block)
turtle.setheading(angle)   - sets the heading to a specific angle
turtle.right(degrees)      - turns to the right (clockwise) by a certain number of degrees
turtle.left(degrees)       - turns to the left (counterclockwise) a certain number of degrees
turtle.pendown()           - put the pen down so all movements are traced out
turtle.penup()             - pick the pen up so that traces are not left behind the turtle
turtle.pencolor(color)     - change the pen color (takes a string, like 'red' or 'blue')
turtle.speed(speed)        - how fast the turtle moves - use 0 for fastest drawing
```

For example, see if you can draw a square (use `turtle.forward(100)` and `turtle.left(90)`). Try drawing a triangle – how many degrees to you need to turn? Being able to draw a triangle is vital for what follows! One warning: Headings are standard polar coordinate headings from mathematics, not the BYOB-style angles (Python is much more sensible in this regard, but just be careful because it's different from BYOB). Experiment with `turtle.setheading()` to see what directions angles refer to. While the functions above are all you need for this lab, if you are curious you can find the full reference manual for the turtle module, describing all available functions, at http://docs.python.org/3.4/library/turtle.html .

Back to task at hand, your goal is to create a Python function that draws Sierpinski triangles. If you wanted to draw a Sierpinski triangle, you will be used something like this (these are in fact the exact parameters that created the drawing above):

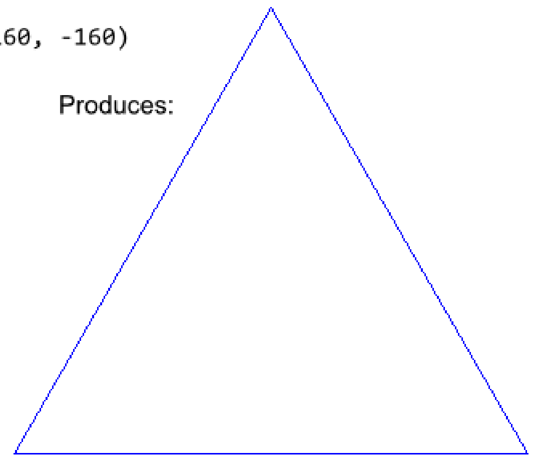
```
draw_sierpinski(320, 7, -160, -160)
```

The parameters give the length of a side of the Sierpinski triangle (320 in this case) and the coordinates of the lower left corner of the triangle ((-160,-160) here). The remaining “levels” parameter says how many recursive calls to make drawing Sierpinski triangles – this one makes 7 recursive calls, making 7 levels of Sierpinski triangles.

The recursion will use the “levels” parameter, with the base case being a 1 level Sierpinski triangle, which looks like this:

```
draw_sierpinski(320, 1, -160, -160)
```

Produces:



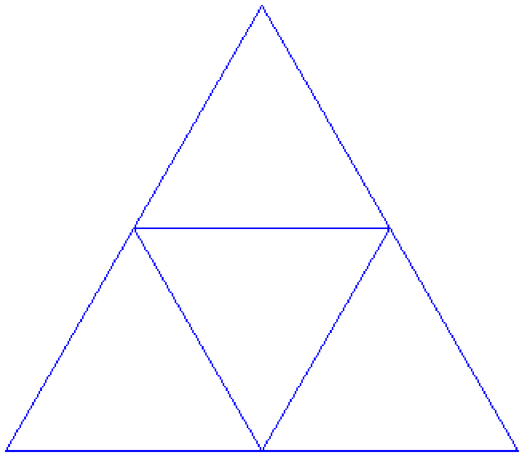
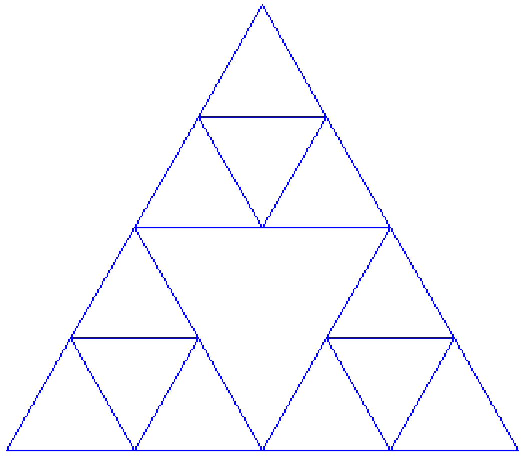
That’s right, a 1-level Sierpinski triangle is just a regular equilateral triangle, “len” units per side. Your first task is to implement this simple base case. Here’s what your code needs to do, written out in text (your job is to turn this into Python code): raise the pen up, go to the specified (x,y) coordinates, put the pen down, draw by moving up and to the right (figure out the angle!) for “len” steps, turn right by 120 degrees and move another “len” steps, turn 120 degrees again and move another “len” steps. Get this code working reliably before moving on: make sure you can draw at different sizes and different locations. This script is the base case of the “`draw_sierpinski()`” function, and the condition for the base case is “levels == 1” – make sure you put this in the basic recursion pattern that we have been using for all of our examples and activities.

Next, do the recursive case. The recursive case is exactly three recursive calls that each draw a Sierpinski triangle with side length “len/2” and with “levels-1” levels. You have to figure out the coordinates for each triangle, but here’s a hint: the one on the lower right is drawn by this call:

```
draw_sierpinski(len/2, levels-1, x+len/2, y)
```

The only tricky coordinates are the one on top. You have to use some basic trigonometry to figure out this location – try to figure it out yourself, but if you can't figure it out look at the solution at the end of this handout.

Once you think you've got the three recursive calls correct, test it! The two pictures below show a 2-level Sierpinski triangle (on the left) and a 3-level triangle (on the right):

If you have any bugs in the recursive case, they'll show up in the 2-level triangle, which should just stack three simple triangles together in the proper locations.

Once you've got this working, test it out with a 6-level triangle – you'll probably want to call `turtle.speed(0)` first though! Make sure everything is saved in Lab9.py.

Activity 4: For this activity, you don't need to write any new code – you'll just be timing the code you wrote in Activities 1 and 2. Use the Python `timeit` module, which you used in Lab 7 (refer back to that lab if you need a refresher on how it works!), to see how long these algorithms take to multiply 900 by 900, and record these times so you can report them in the lab quiz questions. For example, to time the `times1()` algorithm, you'd use something like this:

```
timeit.timeit('times1(900,900)', 'from __main__ import times1', number=XXXXX)
```

You'll need to figure out the right "`number`" of repetitions to get an accurate time, and remember to divide the total time by the number of repetitions before recording and reporting your times in the quiz!

There's nothing to save for this activity, since the times are just taken from interaction in the Python shell. However, remember to record your times for both algorithms!

Bonus Activity (after the lab): The following extra credit challenge does not need to be turned in during lab time. It can be turned in at any time before next week's lab, and is worth up to 40 points extra credit. It's not easy, but if you take it a step at a time you should be able to do it! Here's the problem: given a list of coin values, and an amount you need to make change for, how many different ways are there to make change for this amount? For example, the list of coin denominations for US coins up to a dollar would be the following list:

```
us_coins = [ 1, 5, 10, 25, 50, 100 ]
```

Your goal is to make a Python function than can, for example, count the number of different ways you could make change for one dollar. Using this function could look something like this:

```
change_possibilities(100, us_coins, 6)
```

The first argument is the amount you want to make change for (here 100 cents, or one dollar), and the second parameter is the list of coin denominations. The last parameter is what controls the recursion, and essentially says that you are allowed to use any of the first 6 coin types (since there are exactly six coin types overall, you can use any coin type in this example). If you were to call this with a 1 for the last argument instead of a 6, then it would be the number of different ways you could make change for a dollar using only pennies (and there's only one way to do that: you have to use 100 pennies!). If you called this with the last parameter being 2, then you can use pennies and/or nickels, and there are 21 different ways to do this (you can use 0 nickels, 1 nickel, or any number of nickels up to 20, making up the remainder of the dollar with pennies).

Hints: The last parameter controls the recursion, so concentrate on this. Make a base case where you directly handle the case when the last parameter is 1. Then figure out the recursive step: if you can use the first n coin types, you want to loop through every possibility for the number of coin type n can be used, and recursively call to make up the remainder of the amount with the first n-1 coin types. This is quite challenging for someone new to recursion, but is the kind of thing that is very natural once you gain some experience with this!

If you get this working, save it as Lab9-Bonus.py, and submit separately from the main lab assignment. In addition to writing the code, use your script to figure out how many ways there actually are to make change for a dollar using the US coin denominations. Then create a second list with European coin denominations (1, 2, 5, 10, 20, 50, and 100) and compute how many ways there are to make change for 1 Euro – you might be surprised by the answers!

Submission

In this lab, you should have saved all of your work in Lab9.py. Turn this file in using whatever submission mechanism your school has set up for you. If you do the bonus activity then you will submit Lab9-Bonus.py separately.

Discussion (Post-Lab Follow-up)

Time Complexity of Recursive Algorithms: How do you figure out the time complexity of a recursive algorithm? In our earlier discussion of time complexity, it was primarily about recognizing loops and looking for loops inside of loops. With recursion, operations are repeating not by loops and iteration, but through recursive calls. If the code for a recursive algorithm includes multiple recursive calls (like the Sierpinski triangle activity which had 3 recursive calls), then the analysis uses a technique called "recurrence equations" – these are covered in more advanced computer science classes, and you don't need to worry about that here. We'll just think about the simplest case here, where there is a single recursive call and a constant amount of additional work, like in Activities 1 and 2. In those cases, the time complexity is determined entirely by the total number of recursive calls that are made when executing the program. If you are processing a list with *n* items, and you make a recursive call for a list of size *n*-1, then over the course of execution there will be a call for size *n*, one for size *n*-1, one for size *n*-2, etc., for a total of *n* recursive calls. In this case, the time will be *c***n* for some constant *c* (depending on the "constant amount of work" that's done in addition to the recursive calls), so it is linear time. In other algorithms, the recursion might half the size of the list with each recursive call, so there's a recursive call for size *n*, one for size *n*/2, one for size *n*/4, etc. If you work out the math for this, you'll see that the total number of recursive calls is $\log_2 n$ (that's the base 2 logarithm of *n*), so in this case the time complexity is logarithmic time (much faster than linear!).

Self-Similar Shapes: (Note – you're not responsible for this, but it's cool stuff nonetheless!) There are lots of self-similar diagrams that have been defined over the years, with the two most famous being the Sierpinski diagram that you drew for Activity 3 and the Koch curve (sometimes called the Koch snowflake). Look this up on Wikipedia and you'll see a cool animation that zooms in repeatedly on a portion of the snowflake and you see an infinitely detailed snowflake emerge in this animation. The famous diagrams are all very regular and structured, but you can in fact make diagrams that have some irregularity but use the same rules on different scales, producing self-similar but irregular diagrams with infinite resolution. A lot of game engines use this kind of technique to generate things like game maps, where coastlines (or mountains) are curves that can be generated algorithmically, at whatever "zoom" level you want, and you always have as much resolution as you need.

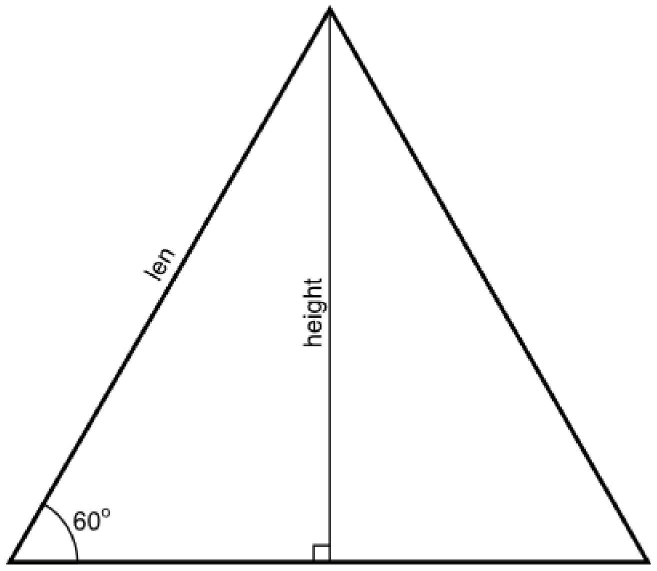
Terminology

The following new words and phrases were used in this lab:

- *base case*: the simplest and smallest case in a recursive algorithm, that is handled directly rather than making calls to smaller versions of the problem
- *proof by induction*: a way of reasoning that builds up the truth of a statement (such as an algorithm working correctly) based on the truth of smaller statements that it depends on
- *recursion*: the process of defining something in terms of itself
- *recursive algorithm*: an algorithm that implements a function by calling itself
- *recursive case*: the part of a recursive algorithm that calls itself on smaller inputs

Solution To Sierpinski Coordinates

In Activity 3, you had to figure out (x,y) coordinate so that you could place the three recursive Sierpinski triangles. The first step is to figure out the height of the triangle, given the “len” parameter – consider this picture:



Figuring out the height is a basic trigonometry problem, using a sine: height = len * sin(60). Using a calculator to find the sine, we find that the height is approximately 0.866*len. Now that you know the height of the overall triangle, the small “sub-triangle” at the top should have a y coordinate that is half the way up this height, so add half the computed height to the y coordinate of the big triangle, and you get the y coordinate of the top small triangle. With this, you should be able to place all three recursive triangles.

Answers to Pre-Lab Self-Assessment Questions

1. The base case always corresponds to the smallest input value, and since this problem is defined for n ≥ 1, the base case is when n=1. In this base case the function should return 1 as the sum (the sum of the first 1 positive integers is 1).
2. The recursive case is when n > 1, and in this case we will recursively solve the problem for a slightly smaller value of n (in other words, for n-1). The recursive call will give the sum of 1+2+...+(n-1), and so we need to add n to this to bring the sum up to the first n positive integers. In other words, in the base case we want to compute “`sum_of_first_ints(n-1) + n`”.
3. We put the pieces from the past two answers together to make the final recursive Python function definition:

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

4. Let's consider some small values of n and look for a pattern. If we call this with an argument of 1, we are in the base case and so no recursive calls are made. If we call it with an argument of 2, then we make one recursive call – in that recursive call the parameter is 1, which is the base case, so we only make the one recursive call when called with an argument of 2. When called with an argument of 3, we make a recursive call with argument 2, which makes a recursive call with argument 1, and then we stop after those two recursive calls. So with argument 1 we make 0 recursive calls; with argument 2 we make 1 recursive calls; and with argument 3 we make 2 recursive calls. This pattern repeats, and so with parameter n we make n-1 recursive calls. Therefore when calling "sum of first (15) positive integers" the program makes 14 recursive calls.
5. If we call "`mystery(3)`", the first "if" test will see of the parameter n is odd – since it is odd, this will immediately return False. If we call with "`mystery(6)`", then the first test fails (since 6 is not odd), and we fall down to the second "if" test – since n is not 2, we go into the recursive case and call "`mystery(3)`" – we just saw that this returns False, so that means that "`mystery(6)`" will also return False. Finally, if this is called with "`mystery(8)`" we will have a recursive call for "`mystery(4)`", which will in turn have a recursive call for "`mystery(2)`" – since this last recursive call returns True, and this is passed on directly through each of the higher-level recursive calls, the result of "`mystery(8)`" will be true. So to summarize:

   "`mystery(3)`" returns False
   "`mystery(6)`" returns False
   "`mystery(8)`" returns True
6. This function returns True when the input is a power of two that is greater than or equal to 2 (in other words, when the number n can be written as $n=2^k$ for some integer $k \geq 1$). Why is this? Looking at the code, it returns False for all odd values, and for all values that can reach an odd value > 2 when you repeatedly divide by 2. The only values that can be repeatedly divided by 2 without reaching an odd number before reaching 2 are the powers of 2.