COSC122 (2021) Lab 4

Recursion

Goals

This lab will give you some practice with several problems that can be solved using *recursion*, and a number of recursive algorithms. In this lab you will:

- compute a number of simple problems (factorials, exponentiation, *etc.*) using recursive functions;
- have more fun with linked lists (or at least improve your understanding of linked lists);
- compare iterative and recursive solutions and think about where recursive methods are (in)appropriate;
- experiment with, and draw *fractals*.

You should be familiar with the material in Chapter 5 of the online textbook (online link here) before attempting this lab.

The `recursion.py` module has a number of functions and function stubs for solving simple problems recursively.

Fibonacci Numbers

A *Fibonacci number* is a number from the Fibonacci sequence of numbers:

`1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...`

The first two numbers are 1 and 1 (the *1st* and *2nd* numbers, respectively), and all other numbers in the sequence are calculated as the sum of the previous two numbers. For example, the *6th* Fibonacci number is the *4th* + the *5th* ($3+5=8$).

Here's a function that calculates the nth Fibonacci number (it's also provided in `recursion.py`):

```
def fib_iterative(n):
    if n == 1 or n == 2:
        return 1
    else:
        # n >= 3 so start with fib of 1 an fib of 2
        # as the last two terms
        # that is, calculate fib 3 first
        fib_n_minus2 = 1
        fib_n_minus1 = 1
        for i in range(3, n+1):
            curr_fib = fib_n_minus1 + fib_n_minus2
            fib_n_minus2 = fib_n_minus1
            fib_n_minus1 = curr_fib
        return curr_fib
```

Your initial tasks are:

- Try calculating various Fibonacci numbers with `fib_iterative`, eg, 5, 10, 100, 1000, 2000, 10000.

  You will see that the Fibonacci numbers become very large. Luckily Python can handle such large integers but we might need to think about how much space such numbers will take up.

  It is far more *elegant* to solve this problem recursively. Think about solving $fib(4)$ as '$fib(3) + fib(2)$', and solving $fib(3)$ as '$fib(2) + fib(1)$', and so on...—that is, solving $fib(n)$ is the same as '$fib(n-1) + fib(n-2)$'.

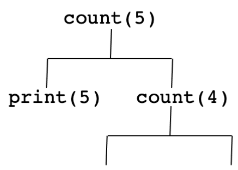
- Load the recursive version, which has been implemented for you in the `fib_recursive` function in the `recursion.py` module.

- While running the method `fib_recursive`, try to inspect how the Python stack behaves (as explained in the lectures); pythontutor.com can be useful for this.
- Try calculating the following Fibonacci numbers using `fib_recursive`: 5, 10, 20, 30, 40. You will find that although recursion is elegant, in this case it isn't so efficient. [1]
- To see why the recursive method gets out of hand, try drawing the start of the tree of calls from `fib_recursive(10)` - don't go all the way down to `fib_recursive(0)` though. For example, `fib_recursive(10)` calls `fib_recursive(9)` and `fib_recursive(8)`, then `fib_recursive(9)` calls `fib_recursive(8)` and `fib_recursive(7)`, etc...

Tracing recursive functions

To get used to understanding how recursion works, we'll start by tracing a simple recursive function. With these examples, you are asked to predict what the code will do before running it to reinforce your understanding of recursion. *Hint*: you won't have Wing available in the test, so you'll need to be good at doing this!

- Load the file `tracing_recursion.py`.
- Add `count(5)` at the end of the program, and run it.
- To trace what happened, put the whole program into `pythontutor.com` and go through it step by step. You'll see the stack frames being created for each recursive call, each with its own independent `val` variable.
- Draw a tree of function calls. The first two levels of the tree are shown here:



- Now look at the function `recount` — it's almost the same as `count`, but with two lines swapped. Try to predict what it will do by drawing a tree for it.
- Trace `recount(5)` in `pythontutor.com` to check it behaves as you expect.
- The third function, `boggle`, is a combination of the previous two. Try tracing it with `boggle(1)`, by hand if you can, then at `pythontutor.com`, to understand what it does.
- See if you can predict what `boggle(2)` will output by drawing the tree for it, then check you've got it right.

An important use of recursion: Exponents

The Fibonacci calculations above turn out to be inefficient using the recursive algorithm given, but in contrast, the problem of working out x^n using a recursive function is very efficient, and turns out to be essential in cryptography.

The recursive function definition for `quick_power` is given below:

$$x^n = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ (x^{n/2})^2 & \text{if } n \text{ is even} \\ x.x^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

[1] In fact, to calculate the n^{th} Fibonacci number, the number additions will be roughly equal to the Fibonacci number being calculated - and this grows very fast, as the function is exponential — think about why. The iterative version can be used to see how many additions the recursive version would be doing for large values of n.

- You should complete the `quick_power` function in the `recursion.py` module.

> *Complete the* **Hand cranked recursion** *and* **Tracing Recursion** *questions in Lab Quiz 4.*

Recursive Implementations

To gain experience with recursive functions:

- You should now open `recursion.py` module and complete the any functions that are missing their implementation. Refer back here for information.

  Hint: When thinking about how to write these functions, you first need to think of a *base case* where the method does not recurse any further (that is, doesn't call itself). Then think about the recursive case and make sure that when the function calls itself that it moves toward the base case.

Linked lists are recursive

Using recursion to traverse linked listed can be more natural than iteration (eg, the while loops that were used in the linked list lab). This is because you can think of a linked list as a recursive structure—the whole list is simply the current node followed by the list starting at the current node's next node. For example, to print a list starting at *node1* we simply need to print the data in *node1* and then print the rest of the list, which is simply the list starting from the next node in the list (ie, starting at `node1.next_node`).

To do:

- Once you have your recursive print function working compare it with the `__str__` methods for generating printable versions of stacks and queues in the linked list lab.

Good compilers/interpreters will usually turn such simple tail recursion in to iteration for efficiency so we get the elegance of recursion with the efficiency of iteration. Unfortunately Python doesn't do well in this area, so be careful to only use recursion where it isn't likely to cause stack blow-outs.

> *Complete the* **Recursion with linked lists** *and* **Recursion with Python lists and strings** *questions in Lab Quiz 4.*

Recurrence Relations

Analysing recursive functions uses recurrence relations. We'll start with a simple example.

- Go back to the `boggle` function we used at the start of this lab—it's in `tracing_recursion.py`. Count the number of lines of output that are produced by `boggle(0)`, `boggle(1)`, `boggle(2)`, and `boggle(3)`. Can you see a pattern?
- Let's analyse the number of lines printed by `boggle(n)`. We'll call it $P(n)$. There are two cases because of the `if` statement: $n <= 0$ and $n > 0$. Write down the a formula for $P(n)$ in each of these cases, in terms of n. (The base case is trivial; the other case will have one element for each of the three statements in the `else` statement.)
- Now try putting $n = 2$ into the formula, that is, work out $P(2)$. You'll end up with it including $P(1)$ in the result, so substitute in $P(1)$ and so on, until you get a number you can calculate. Is $P(2)$ the same as the number of lines you got when you ran `boggle(2)`?
- Now try calculating $P(5)$. A pattern should develop that makes it easy to calculate. Does it match the number of lines printed when calling `boggle(5)`?
- Can you predict $P(10)$? (It's a big number). Does it match the number of lines from `boggle(10)`?
- Can you come up with a closed formula for $P(n)$? that is, a formula that doesn't include `P(...)` on the right hand side and is simply a function of `n`.

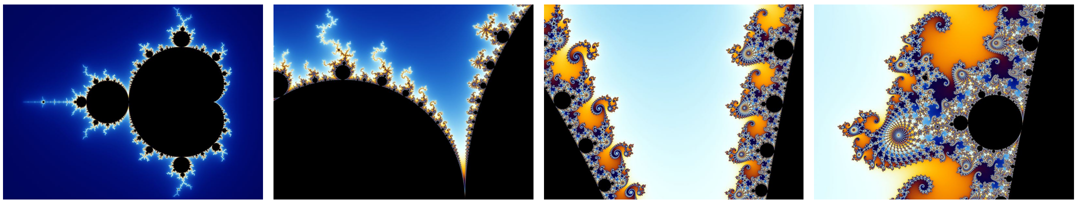
> *Complete the* **Recurrence relations** *question(s) in Lab quiz 4.*

Fractals

The remainder of this lab is optional, but explores some very powerful ways of using a simple recursive definition to draw remarkably intricate images.

A *fractal* is a geometrical shape that possesses an infinite level of detail. They are generated through a repeating pattern, such that zooming into them reveals smaller versions of the larger whole. One of the most popular examples of this is the *Mandelbrot set*, shown in Figure **??**; zooming into the Mandelbrot set reveals an incredible level of geometric complexity and eventually, smaller versions of the entire set again.

While fractals may *seem* very complicated, their final form after many iterations simply masks the simple drawing instructions that they are built on.



(a) **(b)** **(c)** **(d)**

Figure 1: Zooming into the Mandelbrot set; from http://en.wikipedia.org/wiki/Mandelbrot_set.

Turtle Graphics

The fractals we're going to be looking at can be drawn using a technique known as *turtle graphics*. In turtle graphics, you move a robotic pen across a canvas by issuing it simple commands—such as 'move forwards', 'turn left', 'lift up', *etc*. Think about strapping a pen underneath a turtle and walking the turtle around—going forwards, turning left or right, etc. Python comes with the `turtle` module that allows you to easily use a turtle to draw. Try this out in the shell:

```
from turtle import *
shape("turtle")    # Shapes the pen like a turtle
forward(50)
left(90)           # The parameter is the number of degrees to turn
forward(50)
left(90)
forward(50)
right(-90)        # This is the same as left(90)
forward(50)
```

As soon as you call shape, a *Python Turtle Graphics* window will appear. You can resize this window and keep typing commands into the shell (use `reset()` to re-centre the turtle in the window)—the turtle in the graphics window will move in response to your commands, leaving a trail behind it. You should now have a small square in the graphics window. You can do anything you want with the turtle. Check out the spiral below:

```
from turtle import *
for i in range(1,60):
    width(i/4)
    forward(i)
    right(30)
```

NOTE: If you close the turtle window manually (by clicking on the close button) then it may crash. In this case you will need to restart the shell in Wing. To do this click on the options button (top right of Shell window in Wing) and select *restart shell*.

The commands should be simple enough for you to follow with a pen and paper if you find the code a bit confusing. See `help(RawTurtle)` or the Python Manual[2] for a complete list of commands you can issue to the turtle.

A simple introductory fractal is a tree (this code is included in `recursion.py`):

```
from turtle import *
from random import *
def random_tree(size, level):
    if level > 0:
        forward(random() * size)
        x = pos()
        angle = random() * 20
        right(angle)
        random_tree(size * 0.8, level - 1)
        setpos(x)
        left(angle)
        angle = random() * -20
        right(angle)
        random_tree(size * 0.8, level - 1)
        left(angle)
        setpos(x)
    # else do nothing...

random_tree(100,5) # draw a random tree
```

The `fractal_canvas.py` module contains the `FractalCanvas` class that sets up a simple user interface for drawing fractals, including a turtle. The `fractal_drawing` module is where you will be implementing sub-classes of the `Fractal` class that actually do some drawing.

> *Complete the* **Bring out the Turtle** ***question(s) in Lab quiz 4. (Worth zero marks)***

The Sierpinski Triangle

The `fractal_drawing` module contains a complete `SierpinskiTriangle` class, and some code at the very bottom for setting up Tkinter. Running the module as-is should produce a window like that shown in Figure **??**.

This fractal is known as the *Sierpinski triangle*—built from a repeating set of triangles within other triangles. If you click on the *Draw Fractal* button (at the top of the window), it will show you the construction of the triangle. You can also adjust the parameter to the `SierpinskiTriangle` constructor at the bottom of the module to change how many levels of triangles are drawn—but be warned: drawing becomes very slow after around 7 levels; think about how many triangles are being drawn!

The 'level', or *degree* of a fractal is a count of how many times it recurses into itself. In this case, it is drawing five levels of triangles stacked inside each-other.

The `draw` method in `SierpinskiTriangle` sets up some basic parameters for the drawing—the co-ordinates of the outer triangle, the colours to use, and resetting the turtle. `draw_triangle` is where the recursive drawing happens:

1. It draws a triangle for the points it has been given (for the first level, these are the points for the outer triangle).

2. If there are more levels to draw…

3. … calculate the points for the three inner triangles, and call `draw_triangle` to draw them.

Take a moment to experiment with the Sierpinski triangle to understand its drawing code before continuing.

> *Complete the* **Sierpinski Triangles** ***question(s) in Lab quiz 4. (Worth zero marks, but is a good check of your understanding of recursion).***

[2] http://docs.python.org/library/turtle.html

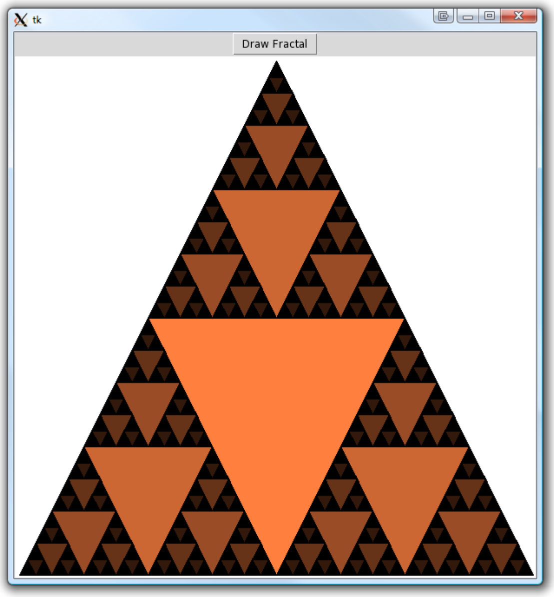
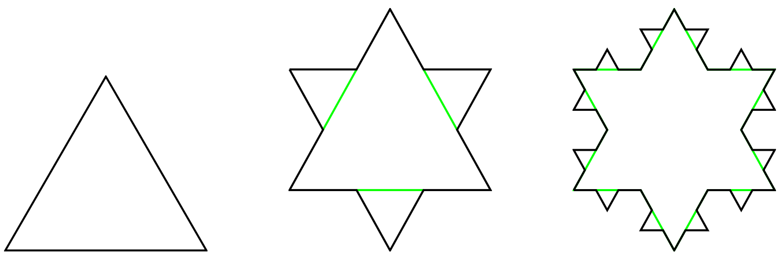


Figure 2: The Sierpinski triangle drawn to 5 levels.

The Koch Snowflake

The *Koch snowflake* is one of the oldest fractal curves; constructed by repeatedly dividing the sides of an equilateral triangle into thirds, and drawing triangles on the middle segment. Figure **??** shows the first few iterations of the snowflake. A snowflake is actually made up of three *Koch curves* drawn at 120° angles; for this exercise we will only draw a single curve (one side of the triangle), but you can implement a snowflake if you wish (see the *Extras* section).



(a) (b) (c)

Figure 3: Constructing the Koch snowflake from level 0 to 2.

The Koch Curve

The `fractal_drawing` module contains an incomplete `KochCurve` class that you will complete, based on the procedure described below.

The `draw` method in `KochCurve` is completed for you, and positions the turtle on the screen at the vertical centre of the left edge; it then calls `draw_koch` with the number of levels to draw, and the length of the curve.

The drawing algorithm for the curve is:

- If we're at the lowest level of the curve (level 0), move forward by the given length (and call `self.update_screen()`).

else

- Divide the length by 3.
- Draw the curve for the next level (ie, level - 1).
- Turn left 60°.
- Draw the curve for the next level.
- Turn right 120°.
- Draw the curve for the next level.
- Turn left 60°.
- Draw the curve for the next level.

Try tracing these instructions out on a piece of paper for the first couple of levels to get a better feeling for what your turtle will be doing.

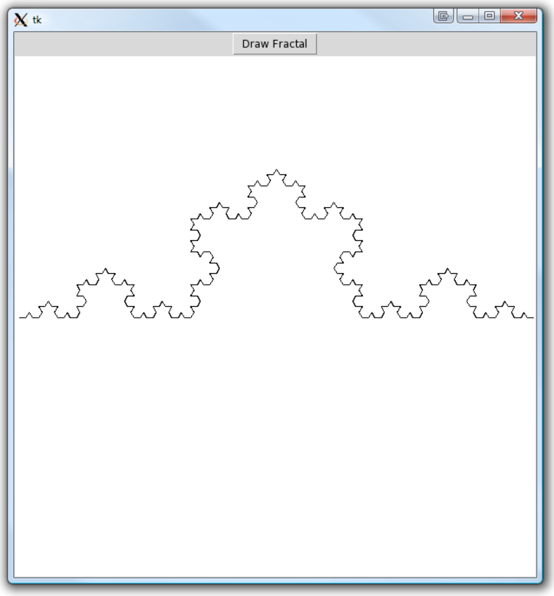


Figure 4: The Koch curve drawn to four levels.

Implement these drawing instructions in the `draw_koch` method, using the `self.turtle` object as your turtle. Remember that the levels decrease as you decend into deeper levels of the curve, and you should stop when you hit the *base case.*

Change the code at the bottom of the module to create a new instance of `KochCurve` instead of `SierpinskiTriangle` and run the code. If you implemented everything correctly, you should see something similar to Figure **??** draw on the screen.

> ***Complete the* Koch Curves *question(s) in Lab quiz 4. (Worth zero marks, but is a good check of your understanding of recursion).***

Even more fun exercises

- Write a recursive function that appends an item to the end of a linked list.

- A *Koch snowflake* is three Koch curves drawn at 120° angles to each other (a triangle). Implement a `KochSnowflake` class that draws this fractal.

- A *Koch starflake* is just like the snowflake except with the Kochcuve angle changed to 80 degrees. Implement a `KochStarflake` class that draws this fractal. We have provided some extra code in the method and you should be able to copy in your code from the snowflake and update it. Have a play with the number of turns that the starflake makes.

- Add some colour to your fractals to indicate some parameter such as the level or age of the line being drawn (for example, `SierpinskiTriangle` draws deeper levels of the triangle in darker shades). You can set the colour of the turtle by calling its `pencolor()` method with the RGB values (0–255), for example: `self.turtle.pencolor(0, 0, 255)` is blue.

- Implement a class for drawing a *Hilbert curve* (see below).

- Implement a class for drawing a *Dragon curve* (see below).

The Hilbert Curve

The *Hilbert curve* is a type of fractal known as a *space-filling curve*—given a square grid of points, it will draw a single line that runs through every point exactly once.

The construction of a Hilbert curve is shown in Figure **??**. The first iteration in Figure **??** covers all points in a 2×2 grid. If we double the size of the grid to 4×4 we take the first iteration, and place it in the corners of the 4×4 grid—the lower two are placed as-is, the upper two are rotated 90° left and right, respectively—with connecting lines added between the shapes (Figure **??**). If we want to double the size of the grid again, we repeat this procedure using the last iteration (Figure **??**).

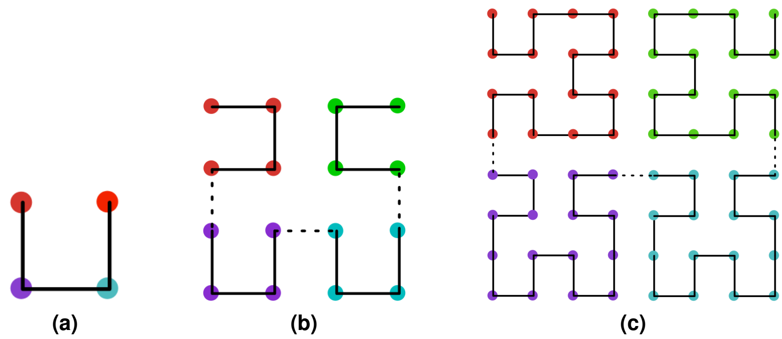


Figure 5: Constructing the Hilbert curve to 3 levels.

The drawing algorithm for this curve is:

- If level is 0 then do nothing. That was easy...

else

1. Turn to the right by the given angle.
2. Draw the curve for the next level (ie, level - 1).
3. Move forward to create a connecting line.
4. Turn to the left by the given angle.
5. Draw the curve for the next level.
6. Move forward to create a connecting line.
7. Draw the curve for the next level.
8. Turn to the left by the given angle.
9. Move forward to create a connecting line.
10. Draw the curve for the next level.
11. Turn to the right by the given angle.

As you decend into deeper levels, the angle you will need to turn will alternate between 90 and -90 to get the inner curve orientated properly. That is, make the recursive call with `draw_hilbert(level-1, -angle)` for the first and last call to `draw_hilbert` from within `draw_hilbert` and simply use `draw_hilbert(level-1, angle)` for the other two calls... phew, recursion can be hard to explain! Don't worry it will be fun playing around with the method and seeing the weird and wonderful shapes you can generate.

The Dragon Curve

The *Dragon curve* is another type of fractal that draws its name from its similarity at high levels to the mythical creature. It is drawn by taking a line and replacing it with two lines at 45° angles to the original line; this process is repeated on the new line segments, alternating the side at which the new lines are drawn on. Figure **??** shows the first few iterations of this process.

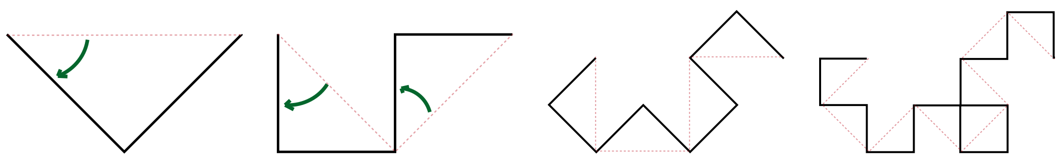
The `fractal_drawing` module contains an incomplete `DragonCurve` class that you will complete, based on the procedure described below.

The drawing algorithm for the curve is:

- If we're at the lowest level of the curve (level 0), move forward by the given length (and call `self.update_screen()`).

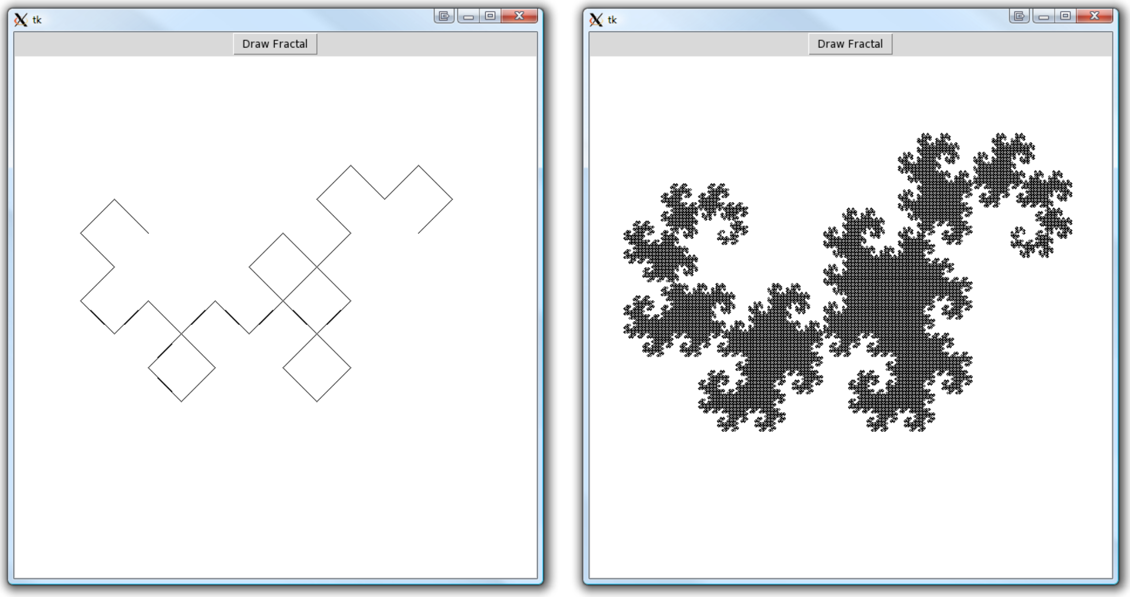
else

- Divide the length by 1.4 (to fit the curve on the screen).



(a) (b) (c) (d)

Figure 6: Levels 1–4 of the Dragon curve (level 0 is a straight line).



(a) (b)

Figure 7: The Dragon curve drawn to (a) 5 and (b) 15 levels.

- Turn right, using the current value of angle.
- Draw the next dragon level with angle set to 45° (ie, level - 1)
- Turn left by 2 times the angle.
- Draw the next dragon level with angle set to −45°.
- Turn right by the value of angle.

The `draw_dragon` method takes in three parameters: the level of the curve, the length to draw the current segment at, and the angle to turn; this angle determines whether we are dividing current line segment with new lines on either its left or right.

Using the instructions above, implement the `draw_dragon` method using the `self.turtle` obejct as your turtle. Remember that the levels decrease as you descend deeper into the curve, and to check for the base case.

Change the code at the bottom of the module to create a new instance of `DragonCurve` and run the code. If you implemented everything correctly, you should see something similar to Figure **??** draw on the screen. Experiment with the number of levels; the curve looks best at high levels (around 15), as shown in Figure **??**.

> ***Complete the* Bonus *question(s) in Lab quiz 4 — relax, these bonus question(s) are not marked.***