Computer Science 1 — CSci 1100
Lab 12 — Recursion
Spring Semester 2017

Lab Overview

This is the last lab of the semester! It only has two checkpoints. You will explore the basic mechanisms of recursion, the design of simple recursive functions, and conversion between recursion and iteration.

As discussed during class, recursion is most often used as a formal way of modeling algorithms and data structures and less often used as a practical programming technique. There are some problems, however, that are almost impossible to solve using a non-recursive algorithm. Many of the search algorithms for tree-like data structures — which you will see if you take Data Structures — fit into this category. Unfortunately, we will not see any algorithms here that fit this category. Still, building an understanding of recursion is an important step in your development as a programmer and as a computer scientist.

Checkpoint 1

Attempts to define the formal foundations of mathematics at the start of the 20th century depended heavily on the mathematical notion of recursion. While ultimately completing this formalization was shown to be impossible, the theory that was developed contributed heavily to the formal basis of computer science. The idea is to start with primitive, axiomatic definitions and build from there.

We will build basic arithmetic operations starting with just one value — 0 — and three operations:

1. adding 1 to a value (the *successor* operation),
2. subtracting 1 from a value (the *predecessor* operation), and
3. testing a value to see if it is 0.

In this case, we can think of all values as successor of 0: the value 1 is the successor of 0, the value 2 is the successor of 1, etc.

Using this logic, we can do represent addition `m+n` as adding 1 to `m` as many times as we can subtract 1 from `n` (until we reach 0). We can write this recursively as follows:

```
def add(m,n):
    if n == 0:
        return m
    else:
        return add(m,n-1) + 1
```

To make sure you understand this, show the sequence of calls made by

```
print(add(5,3))
```

You may add print statements to show the results.

Building on this idea, now write a recursive function to multiply two non-negative integers using only the `add` function we've just defined, together with +1, -1 and the equality with 0 test. Call this function `mult`. Demonstrate the result by multiplying 8 and 3. As a recursive function, `mult` can call itself of course.

Now, define the integer power function, power$(x, n) = x^n$, in terms of the `mult` function you just wrote, together with +1, -1, and equality. Demonstrate the result by computing `power(6,4)`.

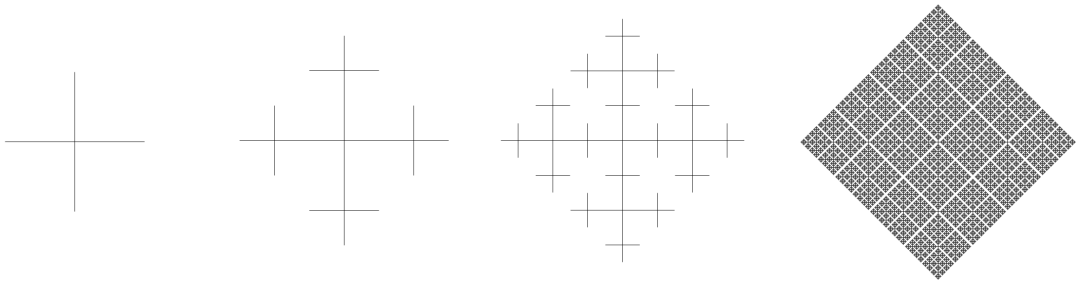
To complete Checkpoint 1: Show:

1. the calls from `add(5,3)`,
2. a working version of `mult`, and
3. a working version of `power`.

Checkpoint 2

In this section, you will write a recursive function to draw a self repeating plus sign. You are given the code — `lab12_check2_start.py` — to draw the first plus sign in the middle of the canvas. First, remember that `(0,0)` is the upper left corner, and `(600,600)` is the lower right corner.

At each iteration, your code will draw the same pattern at the four end points of the current plus sign and then reduce the length of the sign to half of its current value. The expected figure at iterations 0, 1, 2 and a much higher level is shown below.



When you start, your origin is at location `(300,300)` and original length is given as `150` on each side. You first draw a plus sign by drawing two lines, between:

`(150,300)-(450,300)` (horizontal line, total length `300`)

and

`(300,150)-(300,450)` (vertical line, total length `300`)

Now, call the same function recursive 4 times, each time starting from one the end points of this plus sign to draw four new plus signs, but half the length of the current plus sign.

The first time you call this function recursively, the new length of the plus sign will be `75` and the center of these four new plus signs will be: `(150,300)`, `(450,300)`, `(300,150)`, and `(300,450)`.)

Remember each time you call the function recursively, the length will decrease by half and the starting point will be the end point.

Think of a good stopping condition for your recursion. May be a lower limit on the length would be a good choice.

To solve this, you can review the code we have used for the Sierpinski triangle in class.

To complete Checkpoint 2: Show your working program.

Congratulations! You have completed all the labs in the course!

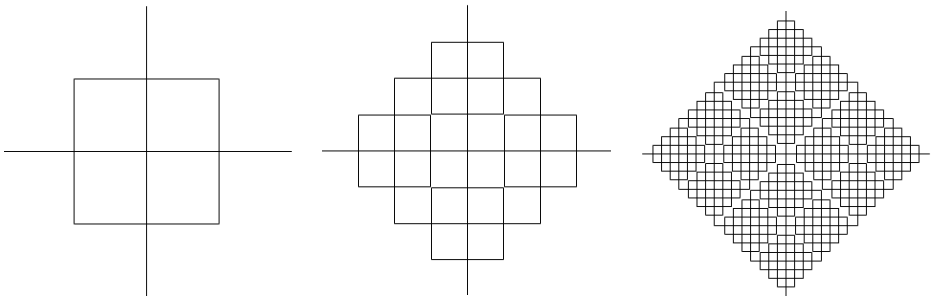
Some extra things to try for fun!

Think about how you can change the order in which the shape is being drawn. Think about adding buttons to increase and decrease the depth of the recursion. Add a Boolean like we did for other tkinter programs that check whether the interface has been destroyed before drawing, so that the program exits cleanly.

One of the interesting things you can try is getting different designs. First, try drawing the lines not at the end of the plus sign, but 1/4 way in from the end points. For example, the second lines will have center locations at follows:

`(300,375)`, `(375,300)`, `(225,300)`, `(300,225)`

(basically length/2 away from the center in all four dimensions.) If you repeat this process, you will get a different pattern, as you can see below:



You can use the same code for the original checkpoint, but simply change the center locations. You may want to keep two versions of the code or include an if statement to toggle between the two. We will call this new version the rectangle version.

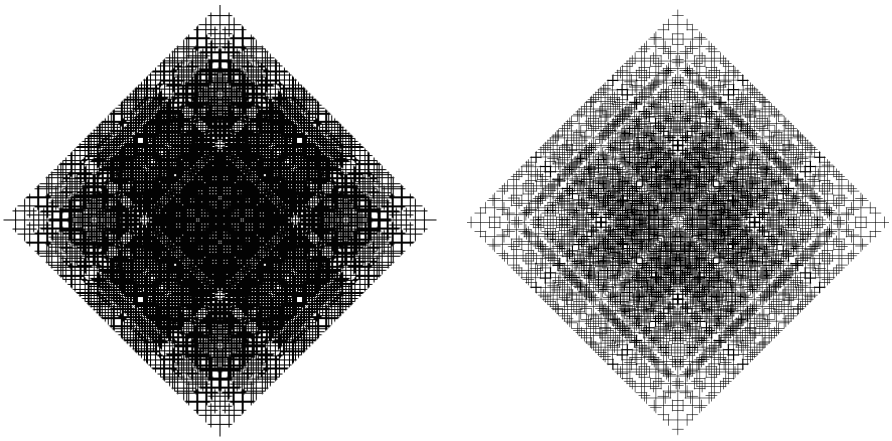
For both the rectangle and the plus version, the new lines don't overlap with the old ones because we keep dividing the length by half and moving the center by the same amount at least. You can change this as well and change the length by a different fraction at each step. Try setting the length at each iteration to:

```
length = 2*length/3
length = 3*length/5
```

instead of length/2. You will see that as patterns overlap, you start seeing some very interesting Moiré patterns:

`http://en.wikipedia.org/wiki/Moire_pattern`

Here are some example patterns you might get (for 2*length/3 for plus and rectangle versions):



Now, try changing the length to `3/2` of its current value every third iteration and halving it at other iterations. Try shifting the starting values a little every time and see new patterns emerge.

Here is a final challenge. We can speed this up considerably by not drawing on top of the same area again and again. Can you change the iterations to produce the same result but only draw in the areas that have a new line?