This is the last lab of the semester! And it only has two checkpoints. You will explore the fundamentals 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 (because it tends to be slower for larger inputs). 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 — Euclid

Euclid’s algorithm for finding the greatest common divisor (GCD) is one of the world’s oldest known algorithms. A brief algorithm description is as follows: If \( a \) and \( b \) are positive integers, with \( a \geq b \), then let \( r \) be the remainder of dividing \( a \) by \( b \). If \( r == 0 \), then \( b \) is the GCD of the two integers. Otherwise, the GCD of \( a \) and \( b \) equals the GCD of \( b \) and \( r \). Here is the Python code:

```python
def gcd( a, b ):
    if a < b:
        a, b = b, a
    r = a % b
    if r == 0:
        return b
    else:
        return gcd( b, r )
```

1. Show the sequence of recursive calls made for the statements below by determining the values of \( a \) and \( b \) for each call. Don’t hesitate to implement this code and add `print` statements to make sure you understand what’s happening here.

   ```
   print gcd( 36, 24 )
   print gcd( 73, 84 )
   print gcd( 84, 66 )
   ```

2. Write and test a non-recursive version of `gcd()` and test with the above values.

To complete Checkpoint 1: Show the calls described above and a working non-recursive version of the code.

Checkpoint 2

In this section, you will write a recursive function to draw a self repeating plus sign. You are given the code to draw the first plus sign in the middle of the canvas. Remember that (0,0) is the upper left corner and (800,800) is therefore the lower right corner.

At each iteration, your code will draw the same pattern at the four end points of the current plus sign, then reduce the length of the sign to half its current value. The expected figures at iterations 0, 1, 2, and a much higher level are shown below.
When you start, your origin is at location \((400,400)\) and the original length is given as 150 on each side. You first draw a plus sign by drawing two lines, between:

\((250,400)\)-(\(550,400\)) (a horizontal line with total length 300)

and

\((400,250)-(400,550)\) (a vertical line with total length 300)

Next, you must start from the end points of this plus sign and draw four new plus signs (recursively) of length 75 on each side; the centers of these four new plus signs will be \((250,400)\), \((400,550)\), \((400,250)\), and \((400,550)\).

Every time you call your function recursively, increase the depth variable. Your function must stop executing when depth reaches the maximum level, which is given as input to the function.

Follow the same pattern we used for the Sierpinski triangle in class. The code is on the course website. First, examine this program carefully and ask your TAs and mentors questions about how it works. Then, adapt the same solution to draw the plus sign patterns.

To complete Checkpoint 2: Show your working program to your TA or a mentor.

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

Try some different variations now to see different effects. Instead of decreasing the length by half, try changing the length as follows:

\[
\text{length} = 2 \times \text{length} / 3
\]

or

\[
\text{length} = 3 \times \text{length} / 5
\]

and other variations. You will see that as patterns overlap, you start seeing some very interesting Moiré patterns:

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

Finally, 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.