CSci 1133, Spring 2015
Lab Exercise 7

Recursion

This Lab exercise introduces you to a powerful problem solving method using functional *recursion.* Recursion refers to an abstraction which is defined in terms of *itself*. Examples include mathematical abstractions such as Fibonacci series, Factorials, and Exponentials, and other abstractions such as the Towers of Hanoi puzzle, the Eight Queens problem and Fractal Geometry. Many interesting and challenging computational problems that are difficult to solve in an iterative fashion are very simple and elegant to solve using recursive function calls. Following the methods discussed in class, you should begin to get a feel for "thinking recursively".

Warm-up

1) Consider the following simple *recursive* function:

```
def foo(n):
    if n<1:
        print('')
    else:
        print('*',end='')
        foo( n-1 )
```

Discuss the following questions with your lab partner. Do not attempt the remainder of this lab exercise until you are able to answer them and are comfortable with the concepts:

a. Describe what this function does, given an integer value for *n*.

b. Every recursive function must contain at least one base case. What is the base case for the function `foo()`?

c. Why is this considered a "base case"?

d. Every recursive function includes a "reduction" step that always moves closer to one of the base cases. Will the reduction step in the `foo()` function move closer to the base case under all circumstances? Explain why or why not.

e. What distinguishes a "base case" from a "reduction step" in a recursive function definition?

2) Consider the following: "construct a *recursive* function to display the digits of an integer in reverse order on the console display". This is a non-value-returning function that takes a single integer argument:

```
def reverseNum(n):
```

and, given an integer value such as:

```
reverseNum(5786);
```

will output the following to the console display:

```
6875
```

Construct the `reverseNum` function:

a. Start by identifying a base case that is trivial to solve. For this problem, perhaps the easiest integer to "reverse" would be any value with a single digit. In this case, you would simply display the input value and be done. Write the Python statement or statements to implement the base case and return.

b. Next, determine if there is some way the original problem can be represented as a simple sub-case *combined* with a recursive call to the function using a reduced version of the original input. For this particular problem, note that if the argument contains more than one digit, reversing it entails displaying the rightmost digit first, followed by the rest of the digits in reverse order. The "*rest of the digits in reverse order*" is simply a reduced version of the original problem! First, write the Python statement or statements that will display the "rightmost" digit.

c. Now, write the *recursive* reduction step that calls the function for the "*rest of the digits*".

d. Combining these two steps forms the complete reduction case. If this is the only reduction step, it is simply performed for any but the base case(s). Complete the else clause from a) to form the entire reduction step.

e. Before you complete and test the function, take a minute to ensure that a) the base case represents a correct solution, and b) the reduction step is guaranteed to get you "closer" to the base case. Now combine the base and reduction steps and write the complete function definition.

3) Write a *recursive* function to determine the maximum value in a list of integers. The function declaration is:

```
def maxValue( vals ):
```

where `vals` is a list of integers.

a. First, describe the base case: [hint: in what size list would it be easiest to find the maximum value? and which value would it be?]

b. Next, describe the reduction step: [hint: describe the original problem using a simple case in combination with some reduced form of the problem. Also consider how you would obtain a "smaller" list]

c. Describe why the reduction step will 1) solve the problem and 2) move closer to the base case:

d. Now, write and test the complete function definition for `maxValue`

e. Is this function tail-recursive? Why or why not?

Stretch

1) **Fibonacci Numbers**

Fibonacci numbers are integers derived from the *Fibonacci Series*:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, . . .

named for Leonardo Fibonacci who described it formally in the early 13th century. However, this series is traced to the mathematics of ancient India where it was used to describe patterns of Sanskrit syllables. More than a mathematical curiosity, Fibonacci numbers appear often in many biological patterns such as the spiral of seeds in sunflowers, the arrangement of pinecone and pineapple florets, the branching of trees and the number of petals on flowers to name just a few. Each of these biological patterns have their basis in the Fibonacci Sequence.

Mathematically, the Fibonacci sequence is described recursively:

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0$$
$$F_1 = 1$$

Construct a *recursive* function named `fibonacci` that will take an integer value *n* as an argument and return F_n , and then write a short program that will call your function to print out the first 20 Fibonacci numbers.

2) **Fractal Trees**

Consider the following function that uses recursion to draw a "tree" using Turtle graphics:

```
def tree(t, trunkLength):
    if trunkLength < 5:
        return
     else:
        t.forward(trunkLength)
        t.right(30)
        tree(t, trunkLength-15)
        t.left(60)
        tree(t, trunkLength-15)
        t.right(30)
        t.backward(trunkLength)
```

Implement this function and try it out with a branch length of 100 (recall that you can speed up the drawing by using the `.speed` turtle method). It generates a symmetric drawing resembling a "tree". However no *real* trees actually grow this way! When a tree creates a new branch, there is some bounded randomness to the direction and length of each branch.

Modify the tree function so it will choose new branching angles between 15 and 45 degrees, and generate branches that are between 12 and 18 units shorter than their "parent" branches. [you will need to use the random module]

Now the output should look a bit more like a "real" tree. Go ahead and try different random ranges for the branches and angles. See if you can create more "lifelike" trees. It's fun!

Workout

1) **Binomial Coefficient**

In probability and statistics applications, you often need to know the total possible number of certain outcome combinations. For example, you may want to know how many ways a 2-card BlackJack hand can be dealt from a 52 card deck, or you may need to know the number of possible committees of 3 people that can be formed from a 12-person department, etc. The *binomial coefficient* (often referred to as "*n* choose *k*") will provide the number of combinations of *k* things that can be formed from a set of *n* things. The binomial coefficient is represented mathematically as:

$$\binom{n}{k}$$

which we refer to as "*n* choose *k*". The naïve approach to computing the binomial coefficient is to use factorials:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

However, this approach is computationally infeasible for all but the simplest problems because factorials quickly become too large to be represented by any numerical variables. For example if you wanted to know the number of BlackJack hands that can be dealt from 3 decks of cards shuffled together, the factorial (52*3)! is not computable using the naïve approach! Fortunately, the definition for the binomial coefficient can be expressed recursively:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \quad \binom{n}{0} = \binom{n}{n} = 1$$

Write a *recursive* function named `choose(n,k)` that will compute and return the value of the binomial coefficient given the integer values *n* and *k*.

Write a short Python program that will input values for *n* and *k*, call your recursive function and then output the results.

Use your program to determine the number of 5-card poker hands that can be dealt from a 52 card deck.

Use your program to determine the number of 2-card blackjack hands that can be dealt from three 52 card decks all shuffled together.

Challenge

Here are two interesting challenge problems. Try them if you have extra time or would like additional practice outside of lab.

1) **Euclidean GCD**

The *Greatest Common Divisor* of two integers *a* and *b*, GCD(*a,b*), not both of which are zero, is the largest positive integer that divides both *a* and *b*. The Greek philosopher and mathematician Euclid devised an incredibly clever and efficient algorithm to determine the GCD of two integers using a minimum number of steps.

The Euclidean algorithm for finding the Greatest Common Divisor of *a* and *b* is as follows:

> If *b*=0, then GCD(*a*,*b*) = *a*.
>
> Otherwise, divide *a* by *b* to obtain an integer quotient *q* and remainder *r*. By definition, $a = bq + r$.
>
> Since GCD(*a*,*b*) = GCD(*b*,*r*), simply replace *a* with *b* and *b* with *r* and repeat the procedure.

Note carefully that the remainders are decreasing at each iteration, therefore a remainder of 0 will eventually result. The last nonzero remainder is GCD(*a,b*).

Example: GCD(1260,198): *a*=1260, *b*=198

GCD(1260,198)	1260 = 198 x 6 + 72	*b*=198, *r*=7	= GCD(198,72)
GCD(198,72)	198 = 72 x 2 + 54	*b*=72, *r*=54	= GCD(72,54)
GCD(72,54)	72 = 54 x 1 + 18	*b*=54, *r*=18	= GCD(54,18)
GCD(54,18)	54 = 18 x 3 + 0	*b*=18, *r*=0	= GCD(18,0)
GCD(18,0)			= 18 (by rule 1)

Write a *recursive* implementation of the Euclidean GCD function: `gcd(a, b)`
Write a program that reads two integers and then calls a *recursive* implementation of the Euclidean GCD function: `gcd(a, b)` that calculates and returns the greatest common divisor of `a` and `b`. Display the two integers and the greatest common divisor on the terminal screen.

Note: *a* must be greater than or equal to *b*. If $a < b$, simply return GCD(*b*,*a*). If either *a* or *b* is negative, replace it with its absolute value.

2) **Sierpinski Carpet**

Review the discussion of fractals in your textbook and examine the figure in problem M4 (p489). The "Sierpinski Carpet" is a fractal form using squares that has infinite area in a bounded space. It is formed by the repeated pattern of a square surrounded by eight smaller "self-similar" squares.

Modify your abstract squares program from Lab6 to recursively construct and display a Sierpinski Carpet.