CMSC 201 Fall 2016
Lab 12 – Recursion

Assignment: Lab 12 – Recursion
Due Date: During discussion, November 28th through December 1st
Value: 10 points

Part 1A: What is Recursion?

So far this semester, we've learned many different ways to control the flow of a program: selection statements, loops (both `for` and `while`), and functions. One specialized type of function makes use of ***recursion***, and so we call it a ***recursive function***.

Some problems can be solved by breaking a problem down into smaller pieces of the same problem. A real world example would be Matryoshka dolls, also known as Russian nesting dolls. These are sets of hollow wooden dolls that "nest" inside each other, with each doll getting progressively smaller, with the smallest doll being solid wood.

(Image from Wikimedia: http://bit.ly/2fDQstN)

If our overall goal is to open all of the dolls until we reached the smallest doll, we can break the problem down into smaller pieces of itself.

1. Open the doll
2. If there's another hollow doll inside, go back to step 1
3. If the doll is solid, stop

This is a very simple example of a recursive solution to a problem. A key component of a recursive function is that it must call itself in order to solve the problem. In our Matryoshka example, opening the doll is the "function," and we continue to "call" that function until we've reached the solid doll at the center.

Part 1B: Recursion vs Iteration

You could have also solved the previous Matryoshka problem with a **`while`** loop, or even a **`for`** loop if we knew ahead of time how many dolls there were. Both recursion and iteration break a large problem down into smaller pieces. The main difference between recursion and iteration can be found if we look at their underlying purpose.

- With iteration, the purpose is to repeat an action until a task is done. This is true for **`while`** loops (stop when the conditional evaluates to **`False`**) and **`for`** loops (stop when it reaches the end of the list).
- With recursion the purpose is to break a problem down into smaller and smaller pieces of itself. When you combine all of those solved smaller pieces of the problem, the problem as a whole is solved.

Part 1C: "Parts" of a Recursive Function

A successful recursive function must have two parts: at least one ***base case*** and at least one ***recursive case***. The base case is similar to the conditional in a **`while`** loop, in that it tells the program when to stop. In a recursive function, it stops calling itself, and typically returns something (a value, a message, or even **`None`**). A recursive function may have more than one base case, just like a **`while`** loop may have more than one comparison in its conditional.

The recursive case is the more interesting part, since this is where the function makes its ***recursive calls*** to itself. A recursive call is the most important part of a recursive function, and has a few key features:

- It must call the function again with new inputs.
- These new inputs must approach at least one of the base cases.
- If needed, the call must also include the **`return`** keyword, in order to be able to return the final result from the original function call.

Part 1D: Recursive Examples

You've seen a number of recursive examples in class already, but let's look at a few more. A very simple one is a "countdown" function – as a reminder, this is a ***toy example***. We could easily do this with a loop, but we want to instead examine how recursion works.

Here is the code for the recursive **`countdown`** function:

```
def countDown(currNum):

    # base case
    if currNum == 0:
        print("The end!")
    # recursive case
    else:
        print("Counting down from", currNum, "...")
        countDown(currNum - 1)       # <----RECURSIVE CALL
```

Take a look at this code and see if you can figure out exactly how it works. Once you have, here is a sample run, using the full code (including a simple **`main()`** to get the number and make the initial call to the recursive function):

```
Please enter a number to count down from: 4
Counting down from 4 ...
Counting down from 3 ...
Counting down from 2 ...
Counting down from 1 ...
The end!
```

The base case, when the function ends, is when the number reaches zero. The function doesn't print anything out or return anything, it simply doesn't call itself (the recursive function) again.

Here is a slightly less "toy" example: something to compute factorials. Factorials were discussed during lecture, but as a reminder, they are the product of all the numbers between the selected number and 1:

$$6! = * 6 * 5 * 4 * 3 * 2 * 1$$

Here is the code for the recursive function for factorial. It has a few extra **`print()`** statements to help us trace our way through the function when it is run.

```
def fact(num):
    print("Calculating factorial for", num)

    # base cases (0! and 1! both equal 1)
    if num == 0:
        return 1
    if num == 1:
        return 1
    # recursive case
    else:
        print("\tIt is " + str(num) + " * " \
              + str(num-1) + "!")
        return num * fact(num - 1)   # <---RECURSIVE CALL
```

Again, take a look at this code and see if you can figure out exactly how it works. Here is a sample run:

```
Please enter a number to compute factorial for: 6
Calculating factorial for 6
        It is 6 * 5!
Calculating factorial for 5
        It is 5 * 4!
Calculating factorial for 4
        It is 4 * 3!
Calculating factorial for 3
        It is 3 * 2!
Calculating factorial for 2
        It is 2 * 1!
Calculating factorial for 1
The factorial of 6 is 720
```

Part 2: Hailstone Problem

After logging into GL, navigate to the **Labs** folder inside your **201** folder. Create a folder there called **lab12**, and go inside the newly created **lab12** directory.

```
linux2[1]% cd 201
linux2[2]% cd Labs
linux2[3]% pwd
/afs/umbc.edu/users/k/k/k38/home/201/Labs
linux2[4]% mkdir lab12
linux2[5]% cd lab12
linux2[6]% pwd
/afs/umbc.edu/users/k/k/k38/home/201/Labs/lab12
linux2[7]%
```

Once you're in the folder, you will need to copy the starter file from my public directory. Type (all on one line – don't forget the rest of the command!):
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/given_hailstone.py hailstone.py

To open the file for editing, type
emacs hailstone.py
and hit enter.

The first thing you should do in your file is complete the comment header block, filling in your name, section number, email, and the date.

Then you can start completing the code, following the comments in the file and the instructions on the following page.

For Lab 12, you will be implementing the path of a hailstone. This should be familiar, since it was part of your Homework 4 from earlier in the semester. The "rules" of the hailstone are contained in your starter file, but they are also here for your reference:

Based on the current value of the height, you will repeatedly do the following:

- If the current height is 1, quit the program
- If the current height is even, cut it in half (divide by 2)
- If the current height is odd, multiply it by 3, then add 1

However, you will be implementing this not with a **`while`** loop, but with recursion! You will also need to count the number of "steps" that it takes for your hailstone to reach a height of 1 (see the sample output for an example). *(HINT: This means you will need to use return when making your recursive function calls!)*

Here are the tasks you need to accomplish to complete this lab:

- ☐ Handle both of the base cases
- ☐ Correctly call the recursive function for both of the recursive cases
- ☐ Make sure the recursive function will return the number of "steps" taken for the hailstone to reach a height of 1
- ☐ Update main() to include an initial call to the recursive function

You can find sample output from the program on the next two pages.

Here is some sample output of the program, with the user input in blue. The "steps" output is displayed in orange.

```
bash-4.1$ python hailstone.py
Welcome to the Hailstone Simulator!
Please enter a height for the hailstone to start at: -4
Invalid height of -4

It took 0 steps to hit the ground.
Thank you for using the Hailstone Simulator!



bash-4.1$ python hailstone.py
Welcome to the Hailstone Simulator!
Please enter a height for the hailstone to start at: 1
Height of 1

It took 0 steps to hit the ground.
Thank you for using the Hailstone Simulator!



bash-4.1$ python hailstone.py
Welcome to the Hailstone Simulator!
Please enter a height for the hailstone to start at: 8
Height of 8
Height of 4
Height of 2
Height of 1

It took 3 steps to hit the ground.
Thank you for using the Hailstone Simulator!
```

(A longer run is available on the next page.)

Here is a sample run of the program, with the user input in blue.

```
Welcome to the Hailstone Simulator!
Please enter a height for the hailstone to start at: 9
Height of 9
Height of 28
Height of 14
Height of 7
Height of 22
Height of 11
Height of 34
Height of 17
Height of 52
Height of 26
Height of 13
Height of 40
Height of 20
Height of 10
Height of 5
Height of 16
Height of 8
Height of 4
Height of 2
Height of 1

It took 19 steps to hit the ground.
Thank you for using the Hailstone Simulator!!
```

Part 3: Completing Your Lab

To test your program, first enable Python 3, then run **`hailstone.py`**. Start off by testing the base cases, before moving onto using heights that will require actual recursion.

Since this is an in-person lab, you do not need to use the **`submit`** command to complete your lab. Instead, raise your hand to let your TA know that you are finished.

They will come over and check your work – they may ask you to run your program for them, and they may also want to see your code. Once they've checked your work, they'll give you a score for the lab, and you are free to leave.

IMPORTANT: If you leave the lab without the TA checking your work, you will receive a **zero** for this week's lab. Make sure you have been given a grade before you leave!