CMSC 201 Fall 2017
Lab 13 – Recursion

Assignment: Lab 13 – Recursion
Due Date: During discussion, December 4th through December 7th
Value: 10 points (8 points during lab, 2 points for Pre Lab quiz)

This week's lab will give you even more practice with using recursion.

(Having concepts explained in a new and different way can often lead to a better understanding, so make sure to pay attention as your TA explains.)

Part 1A: Review – Introduction to 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 the doll is solid wood, stop
3. If there's another hollow doll inside, go back to step 1

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 the nesting dolls 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: Review – Recursion vs Iteration

You could have also solved the previous nesting dolls 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: Review – 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 cause the function to 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 Example

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 a 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
```

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: 7
Counting down from 7 ...
Counting down from 6 ...
Counting down from 5 ...
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. When it stops running, this function doesn't return anything, it simply doesn't call itself (the recursive function) again.

Part 2: Exercise

In this lab, you'll be downloading the start of a program that recursively scrambles a word, and prints out all of the possible permutations of that word. For example, if the string was "201", the permutations would be:

"201", "210", "021", "012", "120", and "102"

Tasks

Starting:

- ☐ Copy the **`given_ scramble.py`** file from Dr. Gibson's **`pub`** directory
  - ☐ It should have been renamed to be **`scramble.py`**

Programming:

- ☐ Open the file and examine the **`permute()`** function and **`main()`**
- ☐ Determine what the base case needs to be
  - ☐ Write code for the base case conditional
- ☐ Calculate the new values needed for the recursive call
  - ☐ Write the code to make the recursive call
- ☐ Update **`main()`** to include a call to the recursive function

General:

- ☐ Run and test your code as needed
- ☐ Show your work to your TA

If you get stuck, don't forget what you learned in Lab 09!

Remember, a "debug statement" is a **`print()`** statement that gives you more information on what exactly is going on. Placing a **`print()`** statement inside your code, can show you what is going on in the "background" of your program. Each time the code is run, the information in your debug statement will be printed to the screen, allowing you to trace what is happening with your program.

For example, you might want to see what the current scramble looks like, or what the recursive function is being given as parameters.

Part 3A: Downloading the File

First, create the **`lab13`** folder using the **`mkdir`** command – the folder needs to be inside your **`Labs`** folder as well.

Next, copy a file into your **`lab13`** folder using the **`cp`** command. (The command should be all on one line.)

```
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/given_scramble.py scramble.py
```

This will copy the file **`given_ scramble.py`** from Dr. Gibson's public folder into your current folder, and will change the file's name to **`scramble.py`** instead.

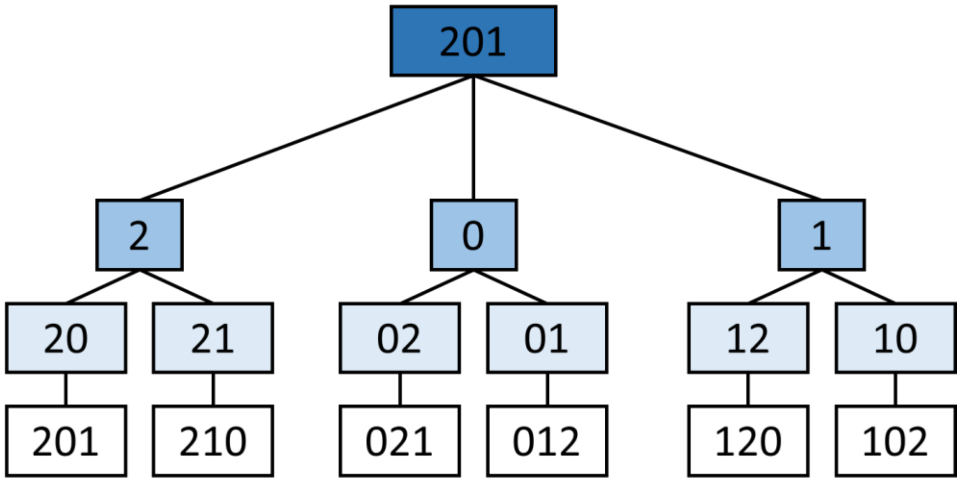
The first thing you should do in your file is complete the file header comment, filling in your name, section number, email, and the date.

Part 3B: Word Scrambler Program

For Lab 13, you will be implementing a word scrambler, using recursion. The algorithm your program should use is:

- Start with an empty string, and a string of letters (the word to scramble)
    - For each letter remaining, add it to the empty string (which is now the currently growing scrambled word)
        - Continue until no letters remain

This image gives a breakdown of each recursive call, and how the recursive calls will branch, given "201" as a starting string to scramble.



Here is some sample output of the program, with the user input in **blue**.

```
bash-4.1$ python scramble.py
Welcome to the Scrambler!
Please enter a string to scramble: one
one
oen
noe
neo
eon
eno
Thank you for using the Scrambler!
```

(More sample output, showing a longer run, is on the following page.)

Here is more sample output of the program, with the user input in **blue**.

```
bash-4.1$ python scramble.py
Welcome to the Scrambler!
Please enter a string to scramble: CMSC
CMSC
CMCS
CSMC
CSCM
CCMS
CCSM
MCSC
MCCS
MSCC
MSCC
MCSC
MCCS
SCMC
SCCM
SMCC
SMCC
SCMC
SCCM
CMSC
CMCS
CSMC
CSCM
CCMS
CCSM
Thank you for using the Scrambler!
```

(Note that the final scrambled list includes duplicates, because the original word has two “C” characters.)

Part 4: Completing Your Lab

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.

Tasks

Starting:

- ☐ Copy the **`given_ scramble.py`** file from Dr. Gibson's **`pub`** directory
    - ☐ It should have been renamed to be **`scramble.py`**

Programming:

- ☐ Open the file and examine the **`permute()`** function and **`main()`**
- ☐ Determine what the base case needs to be
    - ☐ Write code for the base case conditional
- ☐ Calculate the new values needed for the recursive call
    - ☐ Write the code to make the recursive call
- ☐ Update **`main()`** to include a call to the recursive function

General:

- ☐ Run and test your code as needed
- ☐ Show your work to your TA

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!