CSC 401 - AQ2016 – Assignment 7
Due 2 November 2016 by 5.30pm

Week 8 Reading Assignment

We spent our time on a new computational problem solving technique: recursion. We programmed various examples and used the stack to trace the recursive functions. We learned about the properties of recursion (base case(s), recursive step(s)). We saw that with recursion, the stack continues to grow UNTIL we reach the base case. Once we reach the base case, the return value of each recursive call is used by the previous recursive call, until the stack is empty.

Week 9 Reading Assignment

We will continue next week with additional examples of recursion, including applications to search folder structures (10.2). We will revisit Fibonacci numbers, with a very basic discussion on run time analysis (10.3). We will cover linear and binary search (10.4).

Homework Submission

- Complete the problems as specified within the document, using either Microsoft Word or Mac Pages.
- The extra credit problem should contain a function definition, docstring and a commented out section of test runs for that function using multiline comment.
- Save document as hw7.pdf to submit.
- To receive full credit, the problems must be completed as instructed and file must be saved as hw7.pdf to be submitted.
- Submit hw7.pdf through d2l in Dropbox for HW 7 – I will not accept late homework.

Programming Problems – Read each problem carefully

**Remember >>> indicates interactive shell.

Problem 1 recursion call stack (14 pts)

Trace the function call `f(3)`, found after the example. In column one of the table (build stack), show how frames are added to the stack until the base case is reached. In column two of the table (unwind stack), show how return values are updated on the stack as you unwind recursive calls. Please review example.

Example:

```
2  def factorialRecursive( n ):
3      'return factorial of number >= 0'
4
5      if n < 0:
6          print( 'n must be a number >= 0' )
7          return
8
9      if n == 0: # base case 0! = 1
10         return 1
11
12     else:
13         # recursive case: n! = n * (n-1)!
14         return n * factorialRecursive( n - 1 )
```

```
>>> factorialRecursive(3)
```

n = 0	unwind stack ↓
factorial(0) 14	
return: None	1
n = 1	
factorial(1) 14	
return: None	1
n = 2	
factorial(2) 14	
return: None	2
n=3	
factorial(3)	
return: None	6
↑ build stack	

Trace the function call `f(3)` and complete the table

```
2  def f( n ):
3      if n <= 1:
4          return 0
5
6      elif n%2 == 0:
7          return f( n-1 ) + n
8
9      else:
10         return f( n-1 ) - n
```

```
>>> f(3)
```

n = 1	unwind stack ↓
f(1)	
Return: None	0
n=2	
f(2)	
Return: None	2
n = 3	
f(3)	
Return: None	-1
↑ build stack	

Problem 2 recursion pattern and properties (15 pts)

We have a function `powersOfTwo()` that will calculate 2**`n` using recursion. To get full credit for this problem, complete the table below (found after the example) and identify the base and recursive cases.

Example:

`factorial()`

n!	calculation	pattern
0!	1	
1!	1	1 * 0!
2!	2 * 1	2 * 1!
3!	3 * 2 * 1	3 * 2!
4!	4 * 3 * 2 * 1	4 * 3!

Pattern gives us an algorithm for solving the problem:

base case: `factorial(n) = 1, if n==0`

recursive case: `factorial(n) = n * factorial(n-1), if n > 0`

Take a look at what the function `powersOfTwo()` computes for the different values of `n`, to identify the pattern . Complete the table.

2^n	calculation	pattern
2^0	1	
2^1	2	2 * 2^0
2^2	2 * 2	2 * 2^1
2^3	2 * 2 * 2	2 * 2^2
2^4	2 * 2 * 2 * 2	2 * 2^3

Identify base case:

powersOfTwo(n) = 1, if n == 0

Identify recursive case:

powersOfTwo(n) = 2 * powersOfTwo(n-1), if n > 0

Problem 3 more practice with scope (8 pts)

In the code below, identify the local and global variables. For local variables, indicate the in which namespace that the variable has local scope. Write inline comments, as shown in D2L week7 content global_scope.py, to indicate scope.

You are only required to add comments inline to indicate scope, no output or docstring is required. Feel free to run the code to get help with the scope, as I have the added vars() function as discussed during week 7 lecture.

```
def f( y ):  #y is local to f(y)
    print( 'f() local vars:', vars() )
    print()
    return x + y #x is global, y is local, so 5 + 2

def g( y ): #y is local to g(y)
    global x
    x = 10 #because it was specified, this x is redefining the global x
    print( 'g() local vars:', vars() )
    print()
    return x*y #redefined global x, local y, so 10 * 3

def h( y ): #y is local to h(y)
    x = 5 #because it was not specified, this x is local and does not change
global x
    print( 'h() local vars:', vars() )
    print()
    return y - 1 #y is local, so 4 - 1


x = 5 #x is global
res = f(2) + g(3) + h(4) #res is global
print( 'res = {0}'.format( res ) ) #res is global
```

There are additional examples of loop versus recursion.

Extra credit recursion program (10 pts)
DO NOT work on this if you have not completed the rest of the homework yet. You will not receive credit for this problem as a substitution for not doing the assigned problems.

Write a function ***juggle()*** that has one parameter: `num`.

The function generates a sequence of numbers as follows:

- It starts with `num`
- If `num` is even, it replaces `num` with the integer value of `num` **0.5
- If `num` is odd, it replaces `num` with the integer value of `num` **1.5
- This repeats until it hits `num` == 1, printing all the values of `num` along the way

The function **prints** the numbers, each one on a line by itself, including the starting value of `num` and the ending value of 1. This time do it with recursion. DO NOT use any type of loop (while or for). Make sure to include a meaningful docstring. I included in documents for this assignment, a file named halves_recursive.py. Your solution will be similar except you will have two recursive steps.

Test calling function with `num` = 8, 17

```
def juggle(num):
    '''this function calculates a sequence of numbers as follows: it starts with num
    and replaces num with the integer value of num**.5 if num is even, if num is odd
    it replaces num with the integer value of num**1.5; this repeats until it hits
    num == 1, and it prints all of the num along the way.
    '''
    print(num)
    if num == 1: #base case
        return
    elif num%2 == 0:
        return juggle(int(num**0.5))
    else:
        return juggle(int(num**1.5))

'''test runs
>>> juggle(8)
8
2
1
>>> juggle(17)
17
70
8
2
1
>>>
'''
```

Copy and paste a **screenshot** of your final program, including output into your Word or Mac Pages document. If you do not include the screenshot, you will not receive credit. Word has Snipping Tool, unsure of equivalent on a Mac. Here is an example of screenshot for halves_recursive.py:

```
def printHalves( n ):
    '''
    this function calculates of sequence of numbers as follows:
    it starts with n and replaces n with the integer halve. what
    does that mean? if n is odd number, say 9, the integer halve
    is 4, not 4.5.  the function keeps halving until it hits
    n == 1, printing all the values of n it gets along the way.

    instead of while or for loop, we use recursion
    '''

    print( n ) # print all values of n, including last value of 1

    if n == 1: # base case
        return

    else: # recursive step
        return printHalves( n // 2 )

'''test runs
>>> printHalves(8)
8
4
2
1
>>> printHalves( 5 )
5
2
1
>>>
'''
```