Csci 1523
Spring 2016
Lab 10 - Recursion
April 18, 2016

Lab Partner 1(Print): _____

Lab Partner 2(Print): _____

This lab contains 6 pages (including this cover page) and 2 problems. Check to see if any pages are missing.

It is our expectation that students will collaborate and share equally in the conduct of this exercise. However, we understand that often times students will not allocate sufficient time to the exercise thereby transferring responsibility for the completion of the exercise to their partner.

In the event a laboratory team is facing a situation in which one partner feels that an undue amount of the work in completion of the laboratory has been transferred on to them. This partner may elect to submit this work as a solo effort.

Your work in this case will be graded for you individually. No credit will be deducted in the event that partners decide to submit individual weekly efforts.

The laboratories typically consist of some short answer questions followed by a brief programming exercise. in answering your questions please PRINT your answers. In the event we cannot read your answers credit will be taken from your effort.

Also please PRINT LEGIBLY your full name in the space provided on this cover sheet. In the event we cannot read your name we will not award credit for the laboratory. All names should contain both your first name and last name.

Solving problems using recursive function calls
Programming languages such as *Python* support *re-entrant* function calls. A language that supports emphre-entrant functions calls allows a function defined in a particular program to call itself from within its own function body.

Recursive programming is a technique which exploits this capability by using the the program stack to hold intermediate results in stack frames until a terminating condition is reached.

Problems which can divided into sub parts can be solved using recursion. Many times these problem solving solutions require very few programming statements when compared to iterative approaches.

It is important to understand that any problem which can be solved using recursion can also be solved using iteration. Most problems that can be solved iteratively may also be solved using a recursive approach. Generally due to the computational overhead of using many function calls most problems which require successive operations to be solved can be computed much more quickly than using a recursive approach. We use recursion when it is easier to express an algorithm using recursive techniques than when employing an iterative approach. The *Towers of Hanoi* problem is a classical example of recursion solutions being far more easily implemented that iterative solutions.

A good general rule of thumb to use when deciding whether or not to implement recursion is to first evaluate the complexity of the algorithm being implemented. Algorithms of the order, $O(n) = log(n)$ are good candidate for recursive approaches. Algorithms with higher orders of complexity are generally best solved using iteration unless the implementation of the solution by recursion justifies its use.

Recursion has four basic phases:

1. *Initiation* in which the state of the set of variables and-or data structures is set.
2. *General Case* in which a calculation(s) or operation(s) are used to reduce the size of the problem.
3. *Base Case* or *terminating criteria* conditions which halt successive function calls are reached.
4. *Generation of the Solution* once a *base case* is reached the solution to the problem is determined.

Example - Computing the factorial of a number
A classic example of using recursion is to use recursion to calculate the factorial of a number, n. Below is an equation showing the factorial of a number, n:

$$f(n) = \begin{cases} \prod_{x=1}^{n-1} n - x, & \text{if } n \geq 2 \\ 1, & \text{if} \quad n = 0 \quad \text{or} \quad 1 \end{cases} \tag{1}$$

We can see that $n!$ is 1 if $n = 0$ or $n = 1$ otherwise $n!$ is defined by successive multiplications each number in the sequence of numbers leading up to n.

So $n!$ for $n = 3$ is:

$$n = 3 \tag{2}$$

$$f(3) = 3 \times 2 \times 1 \tag{3}$$

$$f(3) = 6 \tag{4}$$

We can see that this problem can be thought about recursively by simply thinking that the number n needs to be broken down to each successive integer from n to 1 and then these resulting numbers need to be multiplied by each other to develop, $n!$.

Using our basic phases described above:

1. *Initiation:* Equation 2, above initializes n, $n = 3$.
2. *General Case:* Equation 3, shows the general case which has identified 3, 2, 1 as the successive integers leading up to n.
3. *Base Case:* In this particular case was to stop when th *terminating criteria* criteria was reached, $(n - x) = 1$.
4. *Generation of the Solution:* Equation 4, is the value returned after the successive integers leading up to n are multiplied by one another.

   Below you will find a code listing of this recursion in Python. Take note of the function definition, the initialization, general case, base case and how the solution is generated.

Listing 1: Using recursion to determine a factorial

```
1  # Python implementation of a recursive solution to the
2  # problem of determining the factorial of a number, n.
3
4  def factorial(n):
5      if (n == 0 or n ==0):
6          return 1;
7      else:
8          return n * factorial(n-1)
9
10 def main():
11     n = eval(input("Number to find factorial: "))
12     print("The value of n input: ",n)
13     n_fac = factorial(n)
14     print("The factorial of n: ",n_fac)
15
16 main()
```

1. **Recursion - Stack Trace**
   This lab exercise provides practice tracing out *recursive* function calls used in a Python script, identifying initiation, general case, base case and generation.

   You will work with a partner on this exercise during your lab session. Two people should work at one computer. Occasionally switch the person who is typing. Talk to each other about what you are doing and why so that both of you understand each step.

   (a) Using Listing 1, enter the program into a file on your computer.

   (b) Use your local program to determine 6!, record it below.

   (c) On which line number of the listing above is the *initiation* done?

   (d) On which line number of the listing above is the *general case* implemented?

   (e) On which line number of the listing above is the *base case* implemented?

   (f) Using the diagram of our segmented memory model sketch out the function calls on the stack at the time the *base case* is reached. Using sketched in arrows show how the *generation step* is accomplished.

2. **Implement a Simple Recursion**
   Give a string, *s*, *s = 'spots'* develop a recursion to print each letter in the string out to the terminal. Before writing any code:

   (a) Sketch out in box below how your recursion may work

(b) Write out your code below(s) giving lines numbers to each line of code.

(c) On which line number of the listing above is the *initiation* done?

(d) On which line number of the listing above is the *general case* implemented?

(e) On which line number of the listing above is the *base case* implemented?

(f) Using the diagram of our segmented memory model sketch out the function calls on the stack at the time the *base case* is reached. Using sketched in arrows show how the *generation step* is accomplished.

(g) Implement your solution in Python. Provide a listing.