Andrew Taylor

atayl136

Homework 3 – Recursion

1. Write a recursive algorithm to compute a+b, where a and b are nonnegative integers.

```
Def addInt(a, b)
    If a = 0
        Return b
    Elif b = 0
        Return a
    Elif b > 0
        Return addint(a+1, b-1)
```

I think 0+0 is not a problem because a=0 will be immediately executed.

2. Let A be an array of integers. Write a recursive algorithm to compute the average of the elements of the array.

```
Def averageArray(Array, index=-1)
    If index = 0
        Return Total/length of Array +1 (if 0 indexed)
    Elif index != 0
        Index = length of Array
        Total = (Array[index]) + (Array[index] - 1)
        averageArray(Array, Index-1)
```

This solution gets broken if someone changes my default argument on the first call.

3. If an array contains n elements, what is the maximum number of recursive calls made by the binary search algorithm?

Binary search is done in two steps: Sort the data and compare the midpoint entry or entries for a match, then based upon that division and comparison continue the process in the relevant half. This continues recursively. The worst cases are when you are searching for the first, last (or even worse) an element that does not exist in the array. The Time Complexity is T(N) = O(1) + T(N/2). The maximum number of recursive calls is only O(lg n). I'm going to say it's only O(lg n) because the algorithm hasn't reached the recursive call yet at first, and that's why the first part of the Time Complexity is given in constant time.

4. The expression m % n yields the remainder of m upon (integer) division by n. Define the greatest common divisor (GCD) of two integers x and y by:

gcd(x, y) = y	**if (y ≤ x and x%y == 0)**
gcd(x, y) = gcd(y, x)	**if (x < y)**
gcd(x, y) = gcd(y, x%y)	**otherwise**

Write a recursive method to compute gcd(x,y).

```
Def gcd(x, y)
    If ((y<x) or (y=x)) and x%y == 0
        Return y
    Elif x<y
        Return gcd(y,x)
    Else
        Return gcd(y, x%y)
```

5. A generalized Fibonacci function is like the standard Fibonacci function,, except that the starting points are passed in as parameters. Define the generalized Fibonacci sequence of f0 and f1 as the sequence gfib(f0, f1, 0), gfib(f0, f1, 1), gfib(f0, f1, 2), ..., where

gfib(f0, f1, 0) = f0
gfib(f0, f1, 1) = f1
gfib(f0, f1, n) = gfib(f0, f1, n-1) + gfib(f0, f1, n-2) if n> 1

Write a recursive method to compute gfib(f0,f1,n).

```
Def gfib(f0, f1, n)
    If n == 0
        Return f0
    Elif n ==1
        Return f1
    Else
        Return gfib(f0,f1,n-1)+gfib(f0,f1,n-2)
```

This is very simple and it took me a long time to figure out in an IDE. But, if you pass it two starting points f0 and f1 with an n it will indeed pass you back the appropriate Fibonacci number.

6. Ackerman's function is defined recursively on the nonnegative integers as follows:

$a(m, n) = n + 1$	if $m = 0$
$a(m, n) = a(m-1, 1)$	if $m \neq 0$, $n = 0$
$a(m, n) = a(m-1, a(m, n-1))$	if $m \neq 0$, $n \neq 0$

Using the above definition, show that a(2,2) equals 7.

$$
\begin{aligned}
A(2, 2) &= A(1, A(2, 1)) \\
&= A(1, A(1, A(2, 0))) \\
&= A(1, A(1, A(1, 1))) \\
&= A(1, A(1, A(0, A(1, 0)))) \\
&= A(1, A(1, A(0, A(0, 1)))) \\
&= A(1, A(1, A(0, 2))) \\
&= A(1, A(1, 3)) \\
&= A(1, A(0, A(1, 2))) \\
&= A(1, A(0, A(0, A(1, 1)))) \\
&= A(1, A(0, A(0, A(0, A(1, 0))))) \\
&= A(1, A(0, A(0, A(0, A(0, 1))))) \\
&= A(1, A(0, A(0, A(0, 2)))) \\
&= A(1, A(0, A(0, 3))) \\
&= A(1, A(0, 4)) \\
&= A(1, 5) \\
&= A(0, A(1, 4)) \\
&= A(0, A(0, A(1, 3))) \\
&= A(0, A(0, A(0, A(1, 2)))) \\
&= A(0, A(0, A(0, A(0, A(1, 1))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(1, 0)))))) \\
&= A(0, A(0, A(0, A(0, A(0, A(0, 1)))))) \\
&= A(0, A(0, A(0, A(0, A(0, 2))))) \\
&= A(0, A(0, A(0, A(0, 3)))) \\
&= A(0, A(0, A(0, 4))) \\
&= A(0, A(0, 5)) \\
&= A(0, 6) \\
&= 7
\end{aligned}
$$

7. Convert the following recursive program scheme into an iterative version that does not use a stack. *f(n)* is a method that returns TRUE or FALSE based on the value of n, and *g(n)* is a method that returns a value of the same type as *n* (without modifying *n* itself).

```
int rec(int n)
{
   if ( f(n) == FALSE ) {
      /* any group of statements that  do not change the value of n */
      return (rec(g(n)));
   }//end if
   return (0);
}//end rec
```

```
Def rec(n)

    While *any group of statements that do not change the value of n*

        If f(n) == FALSE

                g(n)

                continue

        else

                break
```

A good example would be even and odd. If f(n) detects for even and g(n) provides odds this would continue providing odd numbers in an infinite loop when provided any even number n. f(n) and g(n) functions were given.