Homework Three: Recursion

Question One

> Write a recursive algorithm to compute a+b, where a and b are non-negative integers

To do recursive addition, we need to break a larger problem down into very small and repeatable steps. For the case such as a+b, this is difficult to imagine a proper method for a relatively simple problem. However, as you can think of addition of two numbers as (((1 + 1) + 1)...), where the total number of additions of one are a + b,

Thus, we can have some function:

```
function sum(a, b)
    if a == 0 and b == 0 and :
        return 0
    else if a != and b == 0 :
        return 1 + sum(a-1, b)
    else :
        return 1 + (a, b-1)
end function: sum
```

To show how this would work, we can call

sum(1,2)

|b = 2 | 1 | 0, a = 1 | a = 0|

|return 1 + | +1 | + 1 | = 3|

Question Two

> Let A be an array of integers. Write a recursive algorithm to compute the average of the elements of the array. Solutions calculating the sum recursively instead of the average are worth fewer points.

If A is an array of integers, the process of taking the average can be split up recursively by calculating what the multiplier factor is on a per iteration basis.

eg, if for an array [a,b,c] the average is (a+b+c)/3, then to get from a+b/2 to a+b+c/3, we just need to multiply by two(i), add the new element, and divide by n.

It is simpler, because the base case is at position 0, to simply iterate down from the top of the array than to iterate up.

// n is the length of the array, hence accessing the elements needs to be offset by 1 function average(A,n) : if n == 1 - this is the base case where we've either iterated down far enough or we have a 1 element array return A[n-1] return (average(A, n-1) * (n-1) + A[n-1])/n

Question Three

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

If we have an array of n elements, the case in which we would have the maximum number of calls made would be the worst case. For binary search, which functions by dividing a sorted array and then continuing to subdivide based on if the value is less or greater than the median, the worst case would be if the searched for value is either at the maximum or the minimum location of the array. This would thus be represented as the number of divisions it would take for the value N by two to get to one. Thus, we can represent n as some power of 2, such that 2^(x-1) = N, where x is the number of iterations it would take to return. To isolate, we can apply the log to both sides to find that x - 1 = log(n), thus we'll need to make log(n) + 1 comparisons.

Question Four

> The expression m % n yields the remainder of m upon (integer) division by n. Define the greatest common divisor 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) else write a recursive method to do this

```
function gcd(x,y) begin
    if ( y <= x && !x%y)
        return y
    else if (x < y)
        return gcd(y, x)
    else
        return gcd(y, x%y)
endfunction
```

Question Five

> 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)

In order to compute fgib, we will want a single function api gfib(f0, f1, n) which implements the algo

```
function gfib(f0,f1,n) begin
    if(n == 0) return f0
    else if(n == 1) return f1
    else
        return gfib(f0, f1, n-1) + gfib(f0, f1, n-2)
endfunction
```

Question Six

> 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 != 0, n = 0 a(m, n) = a(m-1, a(m, n-1)) if m != 0, n != 0 Using the above definition, show that a(2,2) equals 7.

In order to show this, we have to work through the calls and trace their corresponding sub calls. 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(1,1))) -> a(1,a(0,a(0,a(1,0)))) -> a(1,a(0,a(0,a(0,1)))) a(1,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(03)))) -> a(0,a(0,a(0,4))) -> a(0,a(0,5)) -> a(0,6) -> 7. This can be simplified once you realize that a(1,1) always returns a constant value, thus simplifying some of the stacks with a rapid evaluation

Question Seven

> 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

In order to flatten this into an iterative method, we'll need to analyze the call stack.

we'll want to use a while loop that will run with some non-n condition which will break the loop, equating to a true return from f(n)

```
int rec(int n){
    int m = n;
    bool test_val = f(m)
    while(test_val == FALSE) {
        /* any group of statements that don't change the value of n*/
        m = g(m);
        test_val = f(m); // This updates the test val, meaning we'll break
if test_val is true
    }
    return 0;
}
```