Math 15 - Spring 2017 - Homework 4.6 Solutions

1. (5.1 # 18) The following is known as the Ackermann function.
   `function Ack`$(m, n \in \{0, 1, 2, 3, \dots\})$
   `if` $m = 0$ `then`
   `return` $n + 1$
   `else`
   `if` $n = 0$ `then`
   `return Ack`$(m - 1, 1)$
   `else`
   `return Ack`$(m - 1,$`Ack`$(m, n - 1))$
   Compute the values returned by the following function calls.
   (a) Ack (0,7)=8
   (b) Ack (1,0)=Ack(0,1)=2
   (c) Ack (1,1)=Ack(0,Ack(1,0))=Ack(0,2)=3
   (d) Ack (2,1)=Ack(1,Ack(2,0))=Ack(1,Ack(1,Ack(1,1))=Ack!
   This gets kind of dicey to thread, so I wrote Python program to do this:

```
def Ack(m,n):
    if m==0:
        return n+1
    elif n==0:
        print('Ack(',m-1,',',1,')')
        return Ack(m-1,1)
    else:
        print('Ack(',m-1,',Ack(',m,',',n-1,'))')
        return Ack(m-1,Ack(m,n-1))

print(Ack(2,1))
```

   The program prints out the recursive calls and shows this:

```
Ack( 1 ,Ack( 2 , 0 ))
Ack( 1 , 1 )
Ack( 0 ,Ack( 1 , 0 ))
Ack( 0 , 1 )
Ack( 0 ,Ack( 1 , 2 ))
Ack( 0 ,Ack( 1 , 1 ))
Ack( 0 ,Ack( 1 , 0 ))
Ack( 0 , 1 )
5
```

   Try running this with some larger input...the notariety of the function is that it grows really fast...but it also taxes resources: the message, ": maximum recursion depth exceeded in comparison" comes up pretty soon!

2. (5.1 # 22) Write a recursive function in pseudocode that computes the value of the following recurrence relation:

$$H(n) = \begin{cases} 1 & : n = 1 \\ H(n-1) + 6n - 6 & : n > 1 \end{cases}$$

   Give descriptive preconditions and postconditions. (Hint: See Example 3.10.) ANS:
   `precondition:` $n \in \{1, 2, 3, \dots\}$
   `postcondition:` The n^{th} hexagonal number, $3n^2 - 3n + 1$

```
def Hex(n):
    if n==1:
        return 1
    else:
        return Hex(n-1)+6*n-6

for i in range(1,6):
    print(Hex(i))
```

The for-loop produces correct results:

```
1
7
19
37
61
```

3. (5.1 # 24) Write an iterative algorithm to compute $F(n)$, the nth Fibonacci number.
We did this earlier in the term. The naive algorithm is

```
function fibonacci(n)
   if n < 2
      return 1
   else
      return fibonacci(n-1)+fibonacci(n-2)
```

However, this is typically highly inefficient when implemented on a machine that duplicates the computation of $f(n-2)$ many, many times. The fix for this is called "memoization" or "tail recursion" and is implemented in the Python program below:

```
def fibo(n, memoizer={}):
    if n in memoizer:
        return memoizer[n]
    if n < 2:
        memoizer[n] = 1
    else:
        memoizer[n] =  fibo(n-1) + fibo(n-2)
    return memoizer[n]

fibo(411)
```

The call on my machine (Intel Core i7-6700 CPU @ 3.4GHz) to `fibo(411)` almost instantaneously returns `566790785050287471088226051660549846871789428987517093799713295408147807911158803928 19`

However, it's worth mentioning that many compilers are optimized to deal with this automatically, and Python's iterative assignment method implemented below is equivalently efficient, though, counting this way, the above is the 412th Fibonacci number:

```
def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a+b
```

```
        n -= 1
    return a
fib(412)
```