Recursion

To begin:

- In Eclipse create a Scala project called RecursionLab.
- Add a worksheet to this project called recursionSession
- Add a Scala interpreter to this project

In the session worksheet define and test the following functions. Your implementations should be recursive.

To end:

- Export session.sc to your file system and send it to the grader by the deadline.

Enter the following definitions into the beginning of your session:

```
def inc(n: Int) = n + 1
def dec(n: Int) = n - 1
def isZero(n: Int) = n == 0
```

Note: In the following problems your solutions may assume that all inputs are non-negative integers (these are sometimes called the natural numbers).

1. Problem

Re-define the function:

```
add(n: Int, m: Int) = n + m
```

Your definition should only use recursion, inc, dec, and isZero.

2. Problem

Re-define the function:

```
mul(n: Int, m: Int) = n * m
```

Your definition should only use recursion, add, inc, dec, and isZero.

3. Problem

Re-define the function:

```
exp2(m: Int) = pow(2, m) // = 2^m
```

Your definition should only use recursion, add, mul, inc, dec, and isZero.

4. Problem

Define the hyper-exponentiation function:

```
hyperExp(n: Int) = exp(exp(... (exp(0)) ...)) // n-times
```

Your definition should only use recursion, exp2, add, mul, inc, dec, and isZero.

Notes:

- This will probably cause a stack overflow each time it's called.
- hyperExp(0) = 1

- hyperExp(1) = exp(0) = 2
- hyperExp(2) = exp(exp(0)) = exp(2) = 4
- hyperExp(3) = exp(exp(exp(0))) = exp(4) = 16
- etc.

5. Problem

If the original implementations of the above functions were not tail recursive, re-implement them using tail recursion. Does that improve the stack overflow problem? What about the computation time, is that improved?

6. Problem

At age six Friedrich Gauss discovered an algorithm for tri that uses O(1) space and time! What is it?

7. Problem

Reimplement and test the repl procedure in Calculator.scala without using iteration. Your solution should avoid stack overflow errors.

8. Problem

In earlier problems we saw that addition is iterated increment, multiplication is iterated addition, exponentiation is iterated multiplication, and hyper-exponentiation is iterated exponentiation. We also noted that the growth rate of each function in the sequence is substantially larger than its predecessors. But why stop iterating? Why not define hyper-hyper-exponentiation as iterated hyper-exponentiation, and hyper-hyper-hyper-exponentiation as iterated hyper-hyper-exponentiation:

```
hyper2Exp(m: Int) = hyperExp(hyperExp(... hyperExp(0)...)) // m-times
hyper3Exp(m: Int) = hyper2Exp(hyper2Exp(... hyper2Exp(0)...)) // m-times
```

This infinite sequence of functions—each growing faster than its predecessors-- is called the hyper-exponential hierarchy.

In calculus we learn that some infinite sequences have limits—a point that the sequence approaches but never quite reaches. The limit of the hyper-exponential hierarchy is called Ackermann's function:

```
def ack(n: Int) = hyper(n, n)
```

Where the hyper function combines the entire hierarchy into a single function:

```
hyper(0, n) = inc(n)
hyper(m, 0) = hyper(m - 1, 1)
hyper(m, n) = the result of iterating hyper(m - 1, x) n-times
```

Implement Ackermann's function. Can it be tested?

9. Problem

Find a recursive and tail recursive implementations of the Fibonacci function.

10. Problem

Find a recursive implementation of:

```
choose(n, m) = # of ways to choose m things from n
```

Note: n and m are non-negative integers.

Hint: Pick a special item from the set. Call it x. Then add the number of choices containing x and the number that don't.

11. Problem

Assume two teams, A and B are playing a tournament. The first team that wins k games wins. Assume the probability that either team wins one game is 0.5. Find a recursive implementation of:

```
probability(n, m) = the probability A wins the tournament assuming A must win n more
games and B must win m more games.
```