CIS 194: Homework 7  
Due Friday, 24 October

It’s all about being lazy.

Start off your homework in HW07.hs with the usual module header of module HW07 where.

Fibonacci numbers

The Fibonacci numbers $F_n$ are defined as the sequence of integers, beginning with 0 and 1, where every integer in the sequence is the sum of the previous two. That is,

\[
F_0 = 0 \\
F_1 = 1 \\
F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)
\]

For example, the first fifteen Fibonacci numbers are

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …

It’s quite likely that you’ve heard of the Fibonacci numbers before. The reason they’re so famous probably has something to do with the simplicity of their definition combined with the astounding variety of
ways that they show up in various areas of mathematics as well as art
and nature.

Exercise 1
Translate the above definition of Fibonacci numbers directly into a
recursive function definition of type

```
fib :: Integer -> Integer
```
so that `fib n` computes the `n`th Fibonacci number `F_n`.

Now use `fib` to define the _infinite list_ of all Fibonacci numbers,

```
fibs1 :: [Integer]
```

(*Hint:* You can write the list of all positive integers as `[0..]`.)

Try evaluating `fibs1` at the `ghci` prompt. You will probably get
bored watching it after the first 30 or so Fibonacci numbers, because
`fib` is ridiculously slow. Although it is a good way to _define_ the Fi-
bonacci numbers, it is not a very good way to _compute_ them—in order
to compute `F_n`, it essentially ends up adding 1 to itself `F_n` times! For
example, shown at right is the tree of recursive calls made by evaluat-
ing `fib 5`.

As you can see, it does a lot of repeated work. In the end, `fib` has running time
`O(F_n)`, which (it turns out) is equivalent to `O(\phi^n)`, where
`\phi = \frac{1+\sqrt{5}}{2}` is the “golden ratio”. That’s right, the running time
is _exponential_ in `n`. What’s more, all this work is also repeated from
each element of the list `fibs1` to the next. Surely we can do better.

Exercise 2
When I said “we” in the previous sentence I actually meant “you”.
Your task for this exercise is to come up with more efficient imple-
mentation. Specifically, define the infinite list

```
fibs2 :: [Integer]
```
so that it has the same elements as `fibs1`, but computing the first `n`
elements of `fibs2` requires only (roughly) `n` addition operations.

(*Hint:* You know that the list of Fibonacci numbers starts with 0
and 1, so `fibs2 = [0,1] ++` is a great start. The thing after the `++`
will have to mention `fibs2`, of course, because subsequent Fibonacci
numbers are built using previous ones. Oh, and `zipWith` and `tail`
will be helpful, too. (Why is `tail` here OK?)

Streams
We can be more explicit about infinite lists by defining a type `Stream`
representing lists that _must be_ infinite. (The usual list type represents

Of course there are several billion
Haskell implementations of the Fi-
bonacci numbers on the web, and I have
no way to prevent you from looking
at them; but you’ll probably learn a
lot more if you try to come up with
something yourself first.
lists that may be infinite but may also have some finite length.)

In particular, streams are like lists but with only a “cons” constructor—
whereas the list type has two constructors, [] (the empty list) and (: ) (cons), there is no such thing as an empty stream. So a stream is
simply defined as an element followed by a stream:

\[
\text{data } \text{Stream } a = \text{Cons } a \ (\text{Stream } a)
\]

**Exercise 3**

Write a function to convert a Stream to an infinite list,

\[
\text{streamToList :: Stream } a \rightarrow [a]
\]

**Exercise 4**

To test your Stream functions in the succeeding exercises, it will be useful to have an instance of Show for Streams. However, if you put deriving Show after your definition of Stream, as one usually does, the resulting instance will try to print an entire Stream—which, of course, will never finish. Instead, make your own instance of Show for Stream,

\[
\text{instance Show } a \Rightarrow \text{Show } (\text{Stream } a) \text{ where}
\]

\[
\text{show } \ldots
\]

which works by showing only some prefix of a stream (say, the first 20 elements).

**Exercise 5**

Let’s create some simple tools for working with Streams.

a) Write a function

\[
\text{streamRepeat :: } a \rightarrow \text{Stream } a
\]

which generates a stream containing infinitely many copies of the given element.

b) Write a function

\[
\text{streamMap :: } (a \rightarrow b) \rightarrow \text{Stream } a \rightarrow \text{Stream } b
\]

which applies a function to every element of a Stream.

c) Write a function

\[
\text{streamFromSeed :: } (a \rightarrow a) \rightarrow a \rightarrow \text{Stream } a
\]
which generates a Stream from a "seed" of type a, which is the
first element of the stream, and an "unfolding rule" of type
a -> a which specifies how to transform the seed into a new
seed, to be used for generating the rest of the stream.

Example:

streamToList (streamFromSeed ('x' :) "o") == ["o", "xo", "xxo", "xxxo", "xxxxo", ... ]

Exercise 6

Now that we have some tools for working with streams, let's cre-
ate a few:

a) Define the stream

   nats :: Stream Integer

   which contains the infinite list of natural numbers 0, 1, 2, ...

b) Define the stream

   ruler :: Stream Integer

   which corresponds to the ruler function

   0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, ...

   where the nth element in the stream (assuming the first element
   corresponds to n = 1) is the largest power of 2 which evenly
divides n.

Random numbers

The next section will require a pseudo-random list of numbers. The
exercises in this section will help you generate them.

This section is based on the System.Random module. If you don’t
have this module available, just cabal install random.¹

Computers are deterministic machines. That is, a computer will
blindly follow the sequence of instructions given to it, and there is no
way a computer does anything without a sequence of instructions.
Yet, sometimes, we humans like spontaneity. We want our computers
to produce random numbers—except that determinism tells us this is
impossible.

Of course, there is no such thing as a random number. For ex-
ample, is 33 random? No, it’s the sum of the birthday dates of my

¹It seems that the documentation generation bot on hackage.haskell.org
is having a rough time of things. Go to this version to see the (old, but
still useful) documentation: http://hackage.haskell.org/package/
random-1.0.1.3
wife and me. But, there can be such a thing as a random sequence of numbers, which is a sequence such that the next number can not be predicted by knowing what numbers have come before.

Computers can only approximate generating random sequences. They do so by following a hard-to-predict, yet completely deterministic process. That’s why we say computers produce pseudo-random sequences. (Pseudo- is a Greek prefix meaning “fake”.)

Further complicating matters from an implementation standpoint (but rather clarifying them from a theoretical one), Haskell’s purity means that we cannot have a function `rand :: Int` that produces numbers from a random sequence. Instead, we need a notion of a random number generator, which is some data structure that stores enough information to produce a pseudo-random sequence. According to the System.Random module, such a random number generator is a member of the RandomGen type class.

System.Random also gives us the Random type class, which includes a variety of types for which random generation is possible. Happily for us, Int is in the Random class.

**Exercise 7**

Write a function

```haskell
randomList :: (Random a, RandomGen g) => g -> [a]
```

that produces an infinite pseudo-random sequence of as given a generator of type `g`. The `random` function will be helpful.

**Exercise 8**

Write a function

```haskell
randomInts :: Int -> [Int]
```

such that `randomInts n` is a pseudo-random sequence of Ints, with length `n`. The members of this sequence can range over the full range of Ints. (You can say `minBound :: Int` and `maxBound :: Int` to see the limits of this range, but these functions aren’t necessary in your `randomInts` implementation.)

Use `mkStdGen` and choose your favorite number to be the seed. The choice of seed is irrelevant, but the fact that it’s the same between runs means that your pseudo-random sequence will be the same between runs, which is generally helpful.

**Profiling**

It’s wonderful to be lazy, but laziness occasionally gets in the way of productive work.
Say I want to calculate both the maximum and minimum values of a list of Ints:

```haskell
minMax :: [Int] -> Maybe (Int, Int)
minMax [] = Nothing  -- no min or max if there are no elements
minMax xs = Just (minimum xs, maximum xs)
```

**Exercise 9** Use \( \text{minMax} \) to find the minimum and maximum of a pseudo-random sequence of 1,000,000 Ints. Then, print out these values from a main action. Now, compile your program, enabling RTS options (\texttt{ghc HW07.hs -rtsopts -main-is HW07}),\(^2\) and run your program to see how much memory it takes. (\texttt{./HW07 +RTS -s} or \texttt{HW07.exe +RTS -s} on Windows) It should be a lot. Record the “total memory in use” figure in a comment in your source file.

Then, run your program to see its heap profile, like this:

```bash
> ./HW07 +RTS -h -i0.001
> hp2ps -c HW07.hp
```

(or, for Windows users running at the Windows command prompt \texttt{cmd.exe}):

```bash
> HW07.exe +RTS -h -i0.001
> hp2ps -c HW07.hp
```

) This will create a \texttt{HW07.ps} file, which can be viewed by most modern PDF readers. Check it out. Include this \texttt{HW07.ps} file with your submission.

**Exercise 10** As written, \( \text{minMax} \) does not take advantage of Haskell’s laziness, because it calculates the maximum of \( xs \) and the minimum of \( xs \) separately. The running program must remember all of \( xs \) between these calculations. But, with a rewrite, \( \text{minMax} \) can calculate both the minimum and maximum on the fly, and your program will never need to store the whole list. Implement this better version, run with +RTS -s, and include the improved memory footprint (the “total memory in use” is the one that matters!) in a comment.

*Fibonacci numbers via matrices (extra credit)*

It turns out that it is possible to compute the \( n \)th Fibonacci number with only \( O(\log n) \) (arbitrary-precision) arithmetic operations. This section explains one way to do it.
Consider the $2 \times 2$ matrix $F$ defined by

$$F = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$ 

Notice what happens when we take successive powers of $F$ (see http://en.wikipedia.org/wiki/Matrix_multiplication if you forget how matrix multiplication works):

$$F^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 1 \cdot 1 & 1 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + 0 \cdot 1 & 1 \cdot 1 + 0 \cdot 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$F^3 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix}$$

$$F^4 = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix}$$

$$F^5 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 5 & 3 \end{bmatrix}$$

Curious! At this point we might well conjecture that Fibonacci numbers are involved, namely, that

$$F^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

for all $n \geq 1$. Indeed, this is not hard to prove by induction on $n$.

The point is that exponentiation can be implemented in logarithmic time using a binary exponentiation algorithm. The idea is that to compute $x^n$, instead of iteratively doing $n$ multiplications of $x$, we compute

$$x^n = \begin{cases} (x^{n/2})^2 & \text{n even} \\ x \cdot (x^{(n-1)/2})^2 & \text{n odd} \end{cases}$$

where $x^{n/2}$ and $x^{(n-1)/2}$ are recursively computed by the same method. Since we approximately divide $n$ in half at every iteration, this method requires only $O(\log n)$ multiplications.

The punchline is that Haskell’s exponentiation operator (^) already uses this algorithm, so we don’t even have to code it ourselves!
Exercise 11  (Optional)

- Create a type `Matrix` which represents $2 \times 2$ matrices of `Integers`.

- Make an instance of the `Num` type class for `Matrix`. In fact, you only have to implement the `(*)` method, since that is the only one we will use. (If you want to play around with matrix operations a bit more, you can implement `fromInteger`, `negate`, and `(+)` as well.)

- We now get fast (logarithmic time) matrix exponentiation for free, since `(^)` is implemented using a binary exponentiation algorithm in terms of `(*)`. Write a function

```haskell
fib4 :: Integer -> Integer
```

which computes the $n$th Fibonacci number by raising $F$ to the $n$th power and projecting out $F_n$ (you will also need a special case for zero). Try computing the one millionth or even ten millionth Fibonacci number.

Don’t worry about the warnings telling you that you have not implemented the other methods. (If you want to disable the warnings you can add

```haskell
{-# OPTIONS_GHC -fno-warn-missing-methods #-}
```
to the top of your file.)

On my computer the millionth Fibonacci number takes only 0.32 seconds to compute but more than four seconds to print on the screen—after all, it has just over two hundred thousand digits.