It’s all about being lazy.

Fibonacci numbers

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

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

For example, the first fifteen Fibonacci numbers are

$$
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots
$$

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

\[ \text{fib} :: \text{Integer} \to \text{Integer} \]

so that \( \text{fib} \ n \) computes the \( n \)th Fibonacci number \( F_n \).

Now use \( \text{fib} \) to define the infinite list of all Fibonacci numbers,

\[ \text{fibs1} :: [\text{Integer}] \]

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

Try evaluating \( \text{fibs1} \) at the \texttt{ghci} prompt. You will probably get bored watching it after the first 30 or so Fibonacci numbers, because \( \text{fib} \) is ridiculously slow. Although it is a good way to define the Fibonacci 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 evaluating \( \text{fib} \ 5 \).

As you can see, it does a lot of repeated work. In the end, \( \text{fib} \) has running time \( O(F_n) \), which (it turns out) is equivalent to \( O(q^n) \), where \( q = \frac{1 + \sqrt{5}}{2} \) 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 \( \text{fibs1} \) 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 implementation. Specifically, define the infinite list

\[ \text{fibs2} :: [\text{Integer}] \]

so that it has the same elements as \( \text{fibs1} \), but computing the first \( n \) elements of \( \text{fibs2} \) requires only (roughly) \( n \) addition operations.

Hint: You know that the list of Fibonacci numbers starts with 1 and 1, so \( \text{fibs2} = 1 : 1 : \ldots \) is a great start. The thing after the second (:) will have to mention \( \text{fibs2} \), of course, because subsequent Fibonacci numbers are built using previous ones. Oh, and \texttt{zipWith} and \texttt{tail} will be helpful, too. (Why is \texttt{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 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:

data Stream a = Cons a (Stream a)

Exercise 3 Write a function to convert a Stream to an infinite list,

streamToList :: Stream a -> [a]

Exercise 4 You may have noticed that the Show instance for Streams has already been defined for you. However, there are several other type classes that we may want instances of. Complete the Functor instance by defining the fmap function:

\[ \text{fmap} :: \text{Functor } f => (a \rightarrow b) \rightarrow f a \rightarrow f b \]

Would it make sense to make a Monoid instance for Streams? Why or why not? You do not need to answer this question, but it is a good exercise to think about it.

Exercise 5 Let’s create some simple tools for working with Streams.

a) Write a function

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

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

b) Write a function

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

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:

\[ \text{sIterate } ('x') \ "o" == ["o", "xo", "xxo", "xxxxo", "xxxxxo", ...] \]
c) Write a function

\[
s\text{Interleave} :: \text{Stream } a \rightarrow \text{Stream } a \rightarrow \text{Stream } a
\]

which interleaves the elements from two streams. You will want \(s\text{Interleave}\) to be lazy in its second parameter. This means that you should not deconstruct the second stream in the function.

\text{Example:}

\[
s\text{Interleave} (\text{sRepeat } 0) (\text{sRepeat } 1) = [0, 1, 0, 1, 0, 1, \ldots]
\]

d) Write a function

\[
s\text{Take} :: \text{Int} \rightarrow \text{Stream } a \rightarrow [a]
\]

which takes in an Int \(n\) and returns a list containing the first \(n\) elements in the stream.

\text{Example:}

\[
s\text{Take } 3 (\text{sRepeat } 0) = [0, 0, 0]
\]

**Exercise 6** Now that we have some tools for working with streams, let's create a few:

a) Define the stream

\[
nats :: \text{Stream } \text{Integer}
\]

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

b) Define the stream

\[
ruler :: \text{Stream } \text{Integer}
\]

which corresponds to the \textit{ruler function}

\[
0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, \ldots
\]

where the \(n\)th 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.

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 example, is 194 random? No, it’s the course number for this class! 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 -> Stream Int that produces numbers from a random sequence. Instead, we need a notion of a random number generator.

You will implement a random number generator as a Stream of random numbers. In C, if you do not initialize the random number generator, then it is defaulted to a linear congruential generator with hard coded parameters. Given an initial seed $R_0$, the formula for the generator is:

$$R_n = (1103515245 \times R_{n-1} + 12345) \mod 2147483648 \quad (n \geq 1)$$

Exercise 7 Write a function

rand :: Int -> Stream Int

that produces a Stream of “pseudo-random” numbers using the formula above given an initial seed.

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

Exercise 8  Use minMaxSlow to find the minimum and maximum of
a pseudo-random sequence of 1,000,000 Ints. The code to do this
is provided for you in the main action. Now, compile your program,
creating RTS options (ghc HW06.hs -rtsopts -main-is HW06), and
run your program to see how much memory it takes. (.\HW06 +RTS -s
or HW06.exe +RTS -s on Windows) It should be a lot. Record the “to-
tal memory in use” figure in a comment in your source file.

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

> ./HW06 +RTS -h -i0.001
> hp2ps -c HW06.hp

(or, for Windows users running at the Windows command prompt
cmd.exe:

> HW06.exe +RTS -h -i0.001
> hp2ps -c HW06.hp

This will create a HW06.ps file, which can be viewed by most mod-
ern PDF readers. Check it out!

Exercise 9  As written, minMaxSlow 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 remem-
ber all of xs between these calculations. But, with a rewrite, minMax
can calculate both the minimum and maximum on the fly, and your
program will never need to store the whole list. Implement this bet-
ter 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 (Optional Challenge Problem)

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

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

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 10 (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
fastFib :: 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.

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.