FUNCTIONAL PROGRAMMING MT2019

Practical 1: Factoring Numbers

Deadline: Week 4

This practical requires only a small amount of actual programming. Its main purpose is to give you a chance to get to know the Haskell environment. However, we would like a brief report on it, in order to evaluate your progress. The report should consist of your well-commented Haskell script, together with your test and experiment results, and answers to each of the exercises below. Your work should be completed, and signed-off by a demonstrator, by the end of your practical session in Week 4.

Copy the file `factors.lhs` from the course web page at:
`https://www.cs.ox.ac.uk/teaching/materials19-20/fp/`
or from the course practicals directory `/usr/local/practicals/fp/prac1` into your own directory. Answer the exercises below within this script; label each answer clearly, to help the marker.

Factoring numbers

> Given any two numbers, we may by a simple and infallible process obtain their product, but it is quite another matter when a large number is given to determine its factors. Can the reader say what two numbers multiplied together will produce the number 8616460799? I think it is unlikely that anyone but myself will ever know.
>
> W. S. Jevons, The Principles of Science, Macmillan, 1874.

This practical is concerned with a simple problem whose solution appears extremely difficult: factoring numbers efficiently. In fact, many of today's measures to protect confidential data and communication depend on the belief that there are no efficient methods for factoring numbers.

Of course, there are algorithms that are simple to state, and we will be considering some here, but the problem is that none of them are guaranteed to work within a reasonable time bound. Today's fastest supercomputers and the best algorithms are unable routinely to factor numbers with a few hundred decimal digits.

Here is a simple method for finding the smallest prime factor of a positive integer:

```
> factor :: Integer -> (Integer, Integer)
> factor n = factorFrom 2 n

> factorFrom :: Integer -> Integer -> (Integer, Integer)
> factorFrom m n | r == 0    = (m,q)
>                | otherwise = factorFrom (m+1) n
>    where (q,r) = n `divMod` m
```

The expression *factorFrom* m n returns a pair of integers (a, b), where $a \times b = n$ and a is the least such number with $m \leqslant a$. The value of *divMod* n m is a pair (q, r), where $q = \lfloor n/m \rfloor$ is the quotient of n divided by m (i.e. the largest number of times that m can be subtracted from n), and $r = n - q \times m$ is the remainder. The function *divMod* is one of those provided automatically as a library function in the Haskell Standard Prelude (a library module of standard datatypes, classes, and functions which is imported by default in any Haskell module).

You can see a list of the types of names defined in the prelude by typing `:browse Prelude`
at a GHCi prompt, and can look up the definitions using *Hoogle* at `http://haskell.org/hoogle`.

Exercise 1 *Without* first trying it on the computer, determine the values of `factor 0` and `factor 1`. ♢

Exercise 2 You will find the functions given above, and others, in the script file `factors.lhs`. Run GHCi and check your answer to Exercise 1. ♢

In what follows we are only interested in *factor* n for $n \geqslant 2$.

Exercise 3 Explain why the smallest factor of n cannot be both bigger than $\sqrt{n}$ and less than n.

Code a copy of the *factor* function (call it `factor1`) which calls an auxilliary function `factorFrom1`, and add another guarded equation to the definition of `factorFrom1` with the guard `n <= m*m` to avoid searching for a factor in this range. Be careful to make sure that `factorFrom1` calls `factorFrom1` itself, rather than `factorFrom`.

Does the order of the guarded equations matter? Approximately how many recursive calls are needed to evaluate `factor1 n` in the worst case (as a function of n)? ♢

Exercise 4 Explain (informally) why replacing the test `n <= m*m` by `q < m` defines the same function, and why it is more efficient. Code `factor2` and `factorFrom2` to use this and add it to your copy of the script file. ♢

Exercise 5 The real inefficiency with these simple methods is that all numbers greater than 1 are used as trial divisors. It would be better if, for example, 2 was treated as a special case, and the odd numbers $3, 5, 7, \ldots$ were used as trial divisors. Code this version (call it `factor3`) and add it to your copy of the script file.

How much more efficient would you expect this version to be? ♢

In GHCi you can use the command:
`:set +s`
to set the system to print the approximate time and memory space used to simplify each expression.

Exercise 6 Test your function, `factor3`, and paste some of the test results into your file as part of your final report. (*This is something you should do implicitly for every exercise that involves writing code from now on.*) ♢

Exercise 7 It would be more efficient still to throw away multiples of 3 as well. After treating both 2 and 3 as special cases, the list $5, 7, 11, 13, 17, \ldots$ of odd numbers increasing alternately by 2 and 4 could be used as trial divisors. Implement this idea by coding functions `factor4` and `factorFrom4`; the latter function should take an extra argument s which alternates in value between 2 and 4. (Hint: $6 - s$ does this switching). ♢

Exercise 8 The logical extension of these ideas is to use only prime numbers as trial divisors. Fine in principle, but what is the problem with such an idea? ♢

Prime factorisation

Finding the smallest prime factor is all well and good, but we want to continue the process by finding all prime factors. Here is the algorithm:

```
> factors :: Integer -> [Integer]
> factors n = factorsFrom 2 n

> factorsFrom :: Integer -> Integer -> [Integer]
> factorsFrom m n | n == 1    = []
>                 | otherwise = p:factorsFrom p q
>    where (p,q) = factorFrom m n
```

The type [*Integer*] is the type of lists whose elements are integers. In the definition of *factorsFrom* the empty brackets [] denote the empty list, while $p : ps$ denotes the list whose first element is p and whose remaining elements are those in the list ps.

Exercise 9 The function `factors` is also provided in the script. Try the function out on some suitable examples. Rewrite the function (call it `factors2`) to make use of the improved *factorsFrom* described in Exercise 7.

Check the that `factors` and `factors2` give the same results on some argumenst, perhaps by checking that

```
map factors [2..1000] == map factors2 [2..1000]
```

evaluates to `True`. ♢

Exercise 10 Perform some experiments to compare the speed of `factors` and `factors2`. Paste a small selection of the results into a file for eventual submission as part of your report. How long does it take to find the two factors of Jevons' number (from page 1)? ♢

Optional exercises

The following questions are not compulsory, but are designed for those wanting to pursue these ideas further.

Another approach to factoring was used by Fermat in 1643. It is more suited to finding large factors than small ones.

Suppose that n is an odd number and that $n = u \times v$. It follows that $n = x^2 - y^2$, where $x = (u + v)/2$ and $y = (v - u)/2$ are both whole numbers (why?). Fermat's method consists of systematically searching for the smallest value of x for which there is a y such that

$$x^2 - y^2 = n \quad \text{and} \quad 0 \leqslant y < x.$$

Exercise 11 Suppose that at some stage the search has been narrowed to $x \geqslant p$ and $y \geqslant q$. Let $r = p^2 - q^2 - n$. If $r = 0$, then we are done. If $r < 0$, how should we change p or q? And what if $r > 0$?

Why is this method guaranteed to terminate for all odd n?

Design a function `search` so that $search\ p\ q\ n$ carries out the search.

Hence design a function `fermat` to return two factors of a given odd number. ♢

Exercise 12 What is the smallest possible value of x, that is, the value we should begin the search with? Modify your search, if necessary, so avoid searching at smaller values of x.

You might need the following function for computing integer square roots:

```
> isqrt :: Integer -> Integer
> isqrt = truncate . sqrt . fromInteger
```

which (for a reasonable range of integers n) calculates $\lfloor\sqrt{n}\rfloor$, which is the largest integer whose square is no more than n, that is $(isqrt\ n)^2 \leqslant n < ((isqrt\ n) + 1)^2$. ♢

Exercise 13 Use *fermat* to find the two factors of Jevons' number, and to find the factors of 1963272347809. ♢

Exercise 14 Calculating r on every call of *search* involves a squaring each of two potentially large numbers. A better idea would be to add an extra argument r to *search* which is always equal to $p^2 - q^2 - n$, and use the value of r in calculating the extra argument to each recursive call. Implement this as *fermat* and *search2*. ♢

The *isqrt* function only works for a limited range of arguments. (The *sqrt* function operates on floating point numbers, which are only approximate representations of real numbers, with arithmetic that is also only approximate. In fact `(isqrt (2^105))^2 - 2^105` instead of being no more than zero evaluates to 5545866846675497. Numbers as big as 2^{1024} cannot be represented at all as floating point numbers. You might want to explore what actually happens for larger numbers.)

Exercise 15 Write a function to replace *isqrt* which searches for the largest integer k whose square is no more than n, that is the smallest integer k for which $(k + 1)^2$ is bigger than n. Use *factorsFrom* as a model. ♢

One drawback of searching for an answer like this is that, assuming your answer to exercise 15 uses a linear search, it will take about $\sqrt{n}$ steps to find the answer.

A better strategy involves *binary search*. Suppose we know that $l \leqslant \sqrt{n} < r$, then if we choose an m between l and r it must be that either $l \leqslant \sqrt{n} < m$ or $m \leqslant \sqrt{n} < r$. Moreover we can tell which of these by comparing m^2 with n because $m^2 \leqslant n$ if and only if $m \leqslant \sqrt{n}$. (Why?)

If $l + 1 = r$ and $l \leqslant \sqrt{n} < r$ then $l \leqslant \sqrt{n} < l + 1$, so l is the integer square root $\lfloor \sqrt{n} \rfloor$.

If $l < r$ but $l + 1 \neq r$, then $l < (l + r)\text{`}div\text{`}2 < r$ (why?) so $m = (l + r)\text{`}div\text{`}2$ will do to split the interval more or less in half.

Exercise 16 Design a function *split* which takes a pair (l, r) with $l \leqslant \sqrt{n} < r$ and $l + 1 \neq r$, and returns a pair representing a smaller interval. ♢

Exercise 17 One way of finding an initial (l, r) is to use $1 \leqslant \sqrt{n} < n$ provided $n \geqslant 2$. Write an implementation of *isqrt* that works for all $n \geqslant 2$ by using this observation, and a binary search.

Approximately how many steps does *isqrt n* take? ♢

Exercise 18 Another possible strategy is to search for an upper bound b on $\sqrt{n}$ and start the binary search from the pair $(1, b)$. The search for an upper bound must not take too long, so a good strategy is to look for the first power of two that will do. Implement this strategy, repeatedly doubling a candidate power of two until a bound is found. (This is sometimes called an *open binary search*.)

Approximately how many steps does this implementation of *isqrt* take? Is this worth the extra effort? ♢

Geraint Jones, October 2019