Problem 003 - Largest Prime Factor

Oliver Krischer

October 2021

Contents

1 Problem 003

The prime factors of 13195 are 5, 7, 13 and 29.

What is the largest prime factor of the number 600851475143?

1.1 Naive functional Approach

```haskell
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}
import Control.Monad ( replicateM_ )
import Control.Monad.ST ( runST, ST )
import Data.STRef ( newSTRef, readSTRef, writeSTRef )
import Criterion.Main ( defaultMain, bench, bgroup, whnf )
```

This recursive implementation has four parts:

- A function `factor` which finds the largest factor of a number
- A function `isPrime` which checks for primality of a number

- A helper function `divides`, implementing $k|n$
- A function `largestPF` which just starts the computation

First of all, our entry-point: as we need to find the largest prime factor, we start with an upper limit of $\sqrt{n}$ and iterate downwards:

```
largestPF :: Integer -> Integer
largestPF number | even limit = factor number (limit-1)
                 | otherwise  = factor number limit
  where limit = truncate $ sqrt $ fromIntegral number :: Integer
```

Next, we find the first factor of n (which is the largest) and check for primality. As even numbers cannot be prime, we start with an odd one and skip every second number:

```
factor :: Integer -> Integer -> Integer
factor n k | k `divides` n && isPrime'' k = k
           | otherwise = factor n (k-2)

divides :: Integer -> Integer -> Bool
divides d n = rem n d == 0
```

Now, to the heart of our algorithm: we check for primality by iterating through all numbers from 2 to $\sqrt{n}$, this time in ascending order, and check if k divides n ($k|n$):

```
isPrime :: Integer -> Bool
isPrime number = solve number 2 limit
  where
    limit = truncate $ sqrt $ fromIntegral number :: Integer
    solve n k l | k > l          = True
                | k `divides` n = False
                | otherwise      = solve n (k+1) l
```

The crucial part of this algorithm is testing for primality with `isPrime`, therefor we try to find a better (more efficient) algorithm for testing.

1.1.1 Using a function `ld` for finding the least divisor

The idea is to find the least divisor (except 1) of a number and check it against the number itself: if $ld(n) = n$ then n is prime. With the follwing implementation we get rid of taking the *squareroot* as upper limit and thus obtain a more readable code. This code will be slightly less performant as the naive version, but it prepares for the next step of optimizing:

```haskell
isPrime' :: Integer -> Bool
isPrime' n = ld 2 n == n

ld :: Integer -> Integer -> Integer
ld k n | k `divides` n = k
       | k^2 > n       = n
       | otherwise     = ld (k+1) n
```

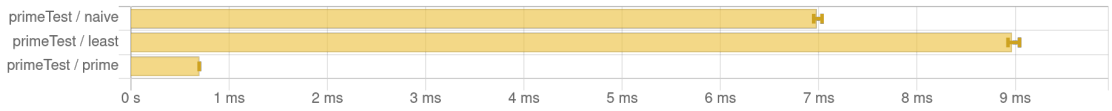
1.1.2 Making `ld` more efficient

With this invariant of *least divisor* we are checking only against prime numbers like this: check $p|n$ for *primes* p with $2 \leq p \leq \sqrt{n}$. Observe that the function `primes` generates an infinite list of prime numbers, which will be only evaluated when needed. This is possible due to *Haskell's* lazy computing model and the way we are calling it: `primes` and `isPrime` are mutually recursive functions, thus prime numbers are only generated up to n. The first call to `isPrime` using `ldp` takes much more time than subsequent calls, as the list of primes hast to be generated in advance. But every subsequent call will be about ten times faster than a call to `ld` (for generating the benchmarks see section 1.4).

```haskell
isPrime'' :: Integer -> Bool
isPrime'' n = ldp primes n == n

ldp :: [Integer] -> Integer -> Integer
ldp (p:ps) n | p `divides` n = p
             | p^2 > n       = n
             | otherwise     = ldp ps n

primes :: [Integer]
primes = 2 : filter isPrime'' [3..]
```



1.2 Imperative Approach

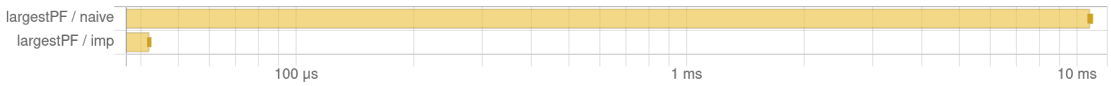
As we have already seen, it is sometimes convinient to write code inspired by imperative programming. Not only makes this the code more compact, but also in many cases more efficient. But be aware that it is not possible to write *real* imperative code in Haskell.

Haskell is a pure functional language, not allowing to re-assingn values to variables which have already been bound to a value. In order to "re-assign" values, we have to pass them as arguments to recursive function calls (unless we use monads in Haskell as in section 1.3). The resulting code is still pure functional code, but inspired by the idea of mutable values.

```haskell
largestPF' :: Integer -> Integer
largestPF' number =
  let (num, last, fact) = largest (n,l,3) in
    if num == 1 then last else num
  where (n,l,f) = factorize (number, 1, 2)

factorize :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
factorize (n,l,f) | f `divides` n = factorize (n `div` f, f, f)
                  | otherwise     = (n,l,f)

largest :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
largest (n,l,f) | n > 1 && f^2 < n = largest (num, last, fact+2)
                | otherwise        = (n,l,f)
                where (num, last, fact) = factorize (n,l,f)
```



As we can see from the benchmark report above, this imperative solution is much more efficient than our naive appoach (about 200 times faster). Without going into details of asymptotic calculation, we can easily see why: where our naive solution had to iterate through every second element in the search space, a call to `factorize` in the imperative solution reduces the search space by factor `f`. And there is even no need of explicit primality checking, because every new found factor must be prime, as all lower factors have already been removed by earlyer calls to `factorize`.

1.3 Using Monads in Haskell

Here is an example of *real* imperative programming in Haskell using the state-thread monad `Control.Monad.ST` to calculate Fibonacci numbers. As a reference, here is an efficient functional approach:

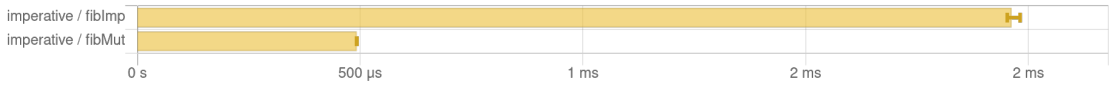
```haskell
fibImp :: Int -> Integer
fibImp n = fst (run n)
  where
```

```
    run 0 = (0,1)
    run n = (b, a+b) where (a,b) = run (n-1)
```

And here the real imperative version with mutable references:

```
fibMut :: Int -> Integer
fibMut n = runST (fibST n)

fibST :: Int -> ST s Integer
fibST n = do
  a <- newSTRef 0
  b <- newSTRef 1
  replicateM_ n (do
    x <- readSTRef a
    y <- readSTRef b
    writeSTRef a y
    writeSTRef b $! (x+y))
  readSTRef a
```



1.4 Benchmarking the solutions

```
main :: IO ()
main = defaultMain [
  bgroup "primeTest" [ bench "naive" $ whnf isPrime   44560482149
                     , bench "least" $ whnf isPrime'  44560482149
                     , bench "prime" $ whnf isPrime'' 44560482149
                     ]
  ]
```

```
main :: IO ()
main = defaultMain [
  bgroup "imperative" [ bench "fibImp" $ whnf fibImp 10000
                      , bench "fibMut" $ whnf fibMut 10000
                      ]
  ]
```

```
main :: IO ()
main = defaultMain [
  bgroup "largestPF" [ bench "naive"      $ whnf largestPF  600851475143
                     , bench "imp" $ whnf largestPF' 600851475143
                     ]
  ]
```