Project Euler

Oliver Krischer

October 2021

Contents

1 Problem 001

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. **Find the sum of all the multiples of 3 or 5 below 1000!**

1.1 Naive solution based on list comprehension

```haskell
import Criterion.Main
sumMultiplesNaive :: Integer -> Integer
sumMultiplesNaive n =
    sum [x | x <- [1..n-1], x `rem` 3 == 0 || x `rem` 5 == 0]
```

The runtime complexity of this algorithm is linear to the input size n, thus $\mathcal{O}(n)$.

1.2 Improved solution using triangular numbers

The starting point for developing an efficient solution is the following idea: instead of checking if the target value is divisible by 3 and 5, we can check separately for division of 3 and 5 and add the results. But then we have to subtract the sum of numbers divisible by `15 (= 3 * 5)`, as we have counted them twice in the first step. When we define a function `sumDivisibleBy :: Int -> Int -> Int`, we can express the result like so:

```haskell
sumMultiplesOptim :: Integer -> Integer
sumMultiplesOptim  n = divBy3 + divBy5 - divBy15
  where divBy3  = sumDivisibleBy 3 n
        divBy5  = sumDivisibleBy 5 n
        divBy15 = sumDivisibleBy 15 n
```

If we apply our naive implementation on `sumDivisibleBy` for 3 and 5 we would then get:

$$3 + 6 + 9 + 12 + \cdots + 999 = 3 * (1 + 2 + 3 + 4 + \cdots + 333)$$
$$5 + 10 + 15 + \cdots + 995 = 5 * (1 + 2 + 3 + \cdots + 199)$$

Thus, we can apply the equation for *Triangular Numbers* (1)

$$T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n * (n + 1)}{2}$$

on our function and we get:

```haskell
sumDivisibleBy :: Integer -> Integer -> Integer
sumDivisibleBy factor limit =
  let n = (limit - 1) `div` factor
  in factor * (n*(n+1)) `div` 2
```

Since `sumDivisibleBy` represents a closed formula, the runtime complexity of this algorithm is constant, thus $\mathcal{O}(1)$.

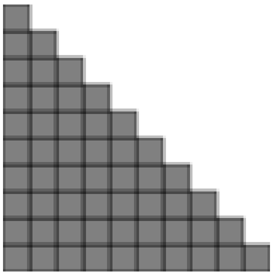
1.3 Triangular Numbers

1.3.1 Theorem

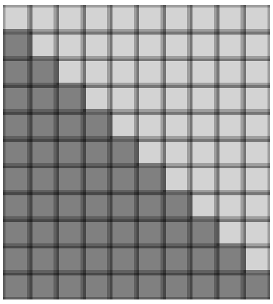
$$T_n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{1}$$

1.3.2 Proof

Triangular numbers are formed by stacking rows of the first n integers, creating a *triangular* geometric pattern, e.g. for $n = 10$:



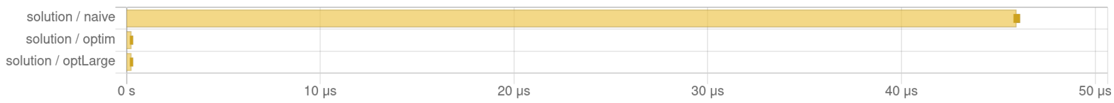
It's easy to see that the number of elements in such a *triangle* is the sum of the integers from 1 to n. If we now geometrically combine two copies of T_n, the resulting rectancle will have side lengths of n and $n + 1$:



Thus, the number of elements in this rectangle is $n(n + 1)$. Since such a rectangle has the double size of the underlying *triangular number*, the size of the *triangular number* is: $\frac{n(n+1)}{2}$ ■

1.4 Benchmarking the solutions

```
main = defaultMain [
  bgroup "solution" [ bench "naive"     $ whnf sumMultiplesNaive 1000
                    , bench "optim"     $ whnf sumMultiplesOptim 1000
                    , bench "optLarge" $ whnf sumMultiplesOptim 1000000
                    ]
  ]
```



2 Problem 002

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms!

2.1 Recursive Implementation with Memoization

```
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}
import Criterion.Main ( defaultMain, bench, bgroup, whnf )
```

This recursive implementation with memoization ist substancially faster then a naive recursive implementation, which would follow the mathematical rule: $fib(n) = fib(n-1) + fib(n-2)$.

```
fibMem :: Int -> Int
fibMem limit = sum $ filter even $ run limit [1,1]
  where
  run limit memo@(n1:n2:_)
    | next > limit = memo
    | otherwise = run limit (next:memo)
    where next = n1 + n2
```

2.2 Imperative Implementation

While the recursive implementation was based on working with lists, the following implementation mimics an imperative solution in which the values of $(a = n - 2)$ and $(b = n - 1)$ and the accumulated `sum` are passed to the next recursive call:

```
fibImp :: Int -> Int
fibImp limit = run limit (1,1) 0
  where
  run limit (a,b) sum
    | c > limit = sum
    | otherwise =
      if even c then run limit (b,c) (sum+c)
      else run limit (b,c) sum
    where c = a + b
```

2.3 Further Improving

Looking at the Fibonacci sequence

$$1, 1, \mathbf{2}, 3, 5, \mathbf{8}, 13, 21, \mathbf{34}, 55, 89, \mathbf{144}, \ldots$$

we can easily see that every third Fibonacci number is even. If this holds true for all Fibonacci numbers, we can get rid of the test for `even` like this:

```
fibOpt :: Int -> Int
fibOpt limit = run limit (1,1,2) 0
  where
  run limit (a, b, c) sum
    | c > limit = sum
    | otherwise = run limit (a', b', c') (sum + c)
    where
    a' = c  + b
    b' = a' + c
    c' = a' + b'
```

2.4 Fibonacci Numbers

2.4.1 Theorem

Every third Fibonacci number is even.

2.4.2 Proof

Following the rule for Fibonacci numbers that every next number is the sum of it's two predecessors or more rigourus

$$fib(n) = fib(n-1) + fib(n-2), where fib\{0,1\} = 1$$

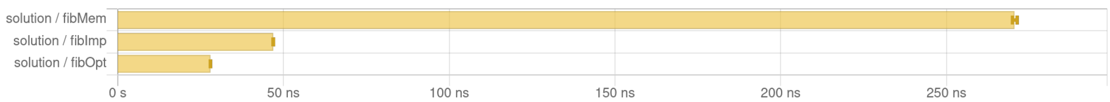
we get an *even* number if both preceeding numbers are odd, and an *odd* number if only one of the predecessors is odd. Given the starting values of $fib(n)$ with $\{1,1\}$ (both *odd*), we get 2 as the first successor, which is *even*. We now have a sequence of $\{1,1,2\}$, which is $\{odd, odd, even\}$. Given this sequence of the first three Fibonacci numbers $\{a,b,c\}$, we can show that every following sequence of three numbers $\{a',b',c'\}$ must also be $\{odd, odd, even\}$:

$$\{a,b,c\} = \{odd, odd, even\} \implies \{a',b',c'\} = \{odd, odd, even\} \tag{2a}$$

$$\begin{aligned}
\{a',b',c'\} &= \{(c+b),(a'+c),(a'+b')\} && \text{(2b)}\\
&= \{(even+odd),(a'+c),(a'+b')\} && \text{(2c)}\\
&= \{odd,(odd+even),(a'+b')\} && \text{(2d)}\\
&= \{odd,odd,(odd+odd)\} && \text{(2e)}\\
&= \{odd,odd,even\} && \text{(2f)}
\end{aligned}$$

2.5 Benchmarking the solutions

```haskell
main :: IO ()
main = defaultMain [
  bgroup "solution" [ bench "fibMem" $ whnf fibMem 4000000
                    , bench "fibImp" $ whnf fibImp 4000000
                    , bench "fibOpt" $ whnf fibOpt 4000000
                    ]
  ]
```



3 Problem 003

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

What is the largest prime factor of the number 600851475143?

3.1 Naive functional Approach

```
{-# 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.

3.1.1 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:

```
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
```

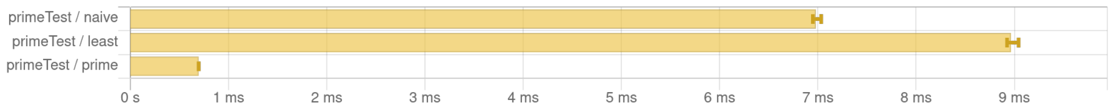
3.1.2 Making `ld` more efficient

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 3.4).

```
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..]
```



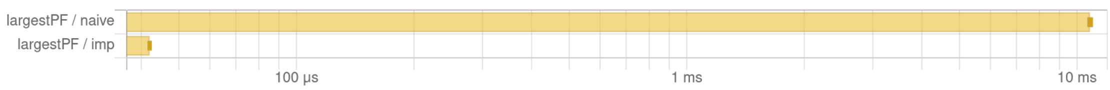
3.2 Imperative Approach

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 3.3). 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`.

3.3 Using Monads in Haskell

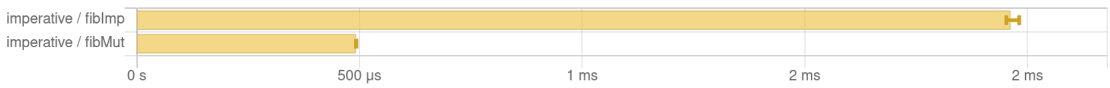
Here is an example of *real* imperative programming in Haskell using the state-thread monad `Control.Monad.ST` to calculate Fibonacci numbers. As a reference, here is an efficient functional approach:

```
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
```



3.4 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
                     ]
  ]
```