Project Euler

Solutions to the problems in Haskell and Go

Oliver Krischer

December 11, 2023

Functional approaches to the solution of a problem are implemented in *Haskell* (light blue background), while imperative approaches are implemented in *Go* (light green background).

For most of the solutions simple benchmarks are provided. Please keep in mind that those benchmarks have no absolute meaning, as no effort was taken to standarize the benchmarks results of both languages. So, you can never say that *Haskell* is faster than *Go* or vice versa.

But, within a given language, the results have a relative meaning: if `someFunction` is 2.5 times slower than `anotherFunction` in the same language, then `anotherFunction` is indeed faster.

Contents

1 Problem 001: Multiples of 3 or 5

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.

```
import Criterion.Main
import Test.QuickCheck ( (==>), quickCheck, Property )
import GHC.StgToCmm.ExtCode (code)
```

1.1 Naive solution based on list comprehension

```
filterMultiples :: Int -> Int
filterMultiples 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 $\Theta(n)$.

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 or 5, we can check separately for division of 3 and 5 and then add the results. But then we have to subtract the sum of numbers divisible by 15 ($= 3 * 5$), as we have added them twice in the first step. If we define a function `triangular :: Int -> Int -> Int`, we can express the result like so:

```
closedFormula :: Int -> Int
closedFormula n = m3 + m5 - m15
    where   m3  = triangular 3 n
            m5  = triangular 5 n
            m15 = triangular 15 n
```

If we would apply our naive implementation for 3 and 5 we'd get:

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

Thus, we can apply the equation for *Triangular Numbers*

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

on our function and we get:

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

Since `triangular` represents a closed formula, the runtime complexity of this algorithm is constant, thus $\Theta(1)$.

1.3 Testing

We will test our functions with *QuickCheck* by comparing their results for a generated range of input values.

```haskell
testProp :: Int -> Property
testProp n = n > 9 ==> filterMultiples n == closedFormula n
```

Executing the tests with `main`:

```haskell
main = quickCheck testProp
```

```haskell
-- alternative main for benchmarking
main = defaultMain [
  bgroup "multiples"  [ bench "filter"  $ whnf filterMultiples 1000
                      , bench "closed"  $ whnf closedFormula 1000
                      ]
  ]
```

If you want to get a feeling about the difference between a runtime complexity of $\Theta(1)$ and $\Theta(n)$, hava a look at the benchmark results for problem001 in the `bench` folder of the root directory.

2 Problem 002: Even Fibonacci Numbers

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.

```haskell
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}
import Criterion.Main
import Test.QuickCheck ( (==>), quickCheck, Property )
```

2.1 Recursive Implementation with Memoization

The following implementation with memoization ist substancially faster then a naive recursive implementation, which would follow the mathematical rule:

$$fib(n) = fib(n-1) + fib(n-2).$$

```haskell
fibMem :: Integer -> Integer
fibMem limit = sum $ filter even $ run limit [1,1]
    where run limit memo@(a:b:_)
            | next > limit = memo
            | otherwise = run limit (next:memo)
            where next = a + b
```

2.2 Functional approach

Using lazy list evaluation and higher order functions we can implement a more idiomatic Haskell solution:

```haskell
fibFun :: Integer -> Integer
fibFun limit = sum $ takeWhile (<= limit) $ filter even $ fibs 1 2
    where fibs a b = a : fibs b (a + b)
```

2.3 Imperative Implementation

While the recursive and functional implementations were based on working with lists in Haskell, the following implementation gives an imperative solution in Go:

```go
func fibImp(limit int) int {
  a := 1
  b := 1
  s := 0
  for b <= limit {
    if b%2 == 0 {
      s += b
    }
    a, b = b, a+b
  }
  return s
}
```

2.4 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. Thus, we can get rid of the test for `b%2 == 0` like this:

```go
func fibOpt(limit int) int {
  a := 1
  b := 1
  c := 2
  s := 0
  for c <= limit {
    s += c
    a = b + c
    b = a + c
    c = a + b
  }
  return s
}
```

2.5 Testing

```haskell
equalsMemFun :: Integer -> Property
equalsMemFun n = n > 0 ==> fibMem n == fibFun n

main = do
  quickCheck equalsMemFun
```

2.6 Benchmark

```haskell
-- alternative main for benchmarking
main = defaultMain [
  bgroup "fib" [ bench "Mem"  $ whnf fibMem 4000000
               , bench "Fun"  $ whnf fibFun 4000000
               ]
  ]
```

From this benchmark we see that `fibFun` is about twice as fast as `fibMem`.

A benchmark for Go can be generated inside a testfile like so:

```go
func BenchmarkProblem002FibImp(b *testing.B) {
  for i := 0; i < b.N; i++ {
  fibImp(4000000)
  }
}
```

From that, we see that `fibOpt` is about three times faster than `fibImp`.

3 Problem 003: Largest prime factor

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

What is the largest prime factor of the number 600851475143?

```haskell
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}
import Criterion.Main
import Test.QuickCheck ( (==>), quickCheck, Property )
import Test.QuickCheck.Text (paragraphs)
```

3.1 Checking for Prime Numbers

With this approach, we generate an infininte but lazy evaluated list of prime numbers, from which we take the potential prime factors. The prime numbers are evaluated (if they divide the given number) until their square exceeds the number.

```haskell
checkPrimes :: Integer -> Integer
checkPrimes n = maximum (getPF n primes)

getPF :: Integer -> [Integer] -> [Integer]
getPF n primes@(p:ps)
    | p*p > n = [n]
    | rem n p == 0 = p:getPF (n `div` p) primes
    | otherwise = getPF n ps

primes :: [Integer]
primes = sieve [2..]

sieve :: [Integer] -> [Integer]
sieve (x:xs) = x:sieve [y | y <- xs, rem y x /= 0]
```

3.2 Factor out all Factors

The key to this solution is a simple idea: *instead of just checking if a number divides n we actually divide n by this number.*

Repeatedly dividing n by its factors decreases n very fast, making early termination of the algorithm possible.

The algorithm works as follows: for each integer number $k \geq 2$, if k is a factor of n, divide n by k and completely divide out each k before moving to the next k. When the next k is a factor it will necessarily be prime, as all smaller factors have already been removed. After dividing out all prime factors n will equal to 1.

```
factorOut :: Integer -> Integer
factorOut number
    | num == 1   = last
    | otherwise = num
    where
        (n,l,k) = factorize (number, 1, 2)
        (num, last, fact) = largest (n,l,3)

factorize (n,l,k) | rem n k == 0 = factorize (n `div` k, k, k)
                  | otherwise = (n,l,k)

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

The problem with this imperative solution in Haskell is that it is much slower than the solution based on lazy lists (about seven times slower). So, we'll give it a try with Go:

```
func factorNaive(n int) int {
  factors := []int{}
  k := 2
  for k <= n {
    for n%k == 0 {
      factors = append(factors, k)
      n /= k
    }
    k += 1
  }
  return slices.Max(factors)
}
```

That works quite well ($\approx 10\mu s$ per call), but there's still room for improvement:

1. we don't need to create an array of factors, we just need the biggest one
2. we factor out all 2s at first and then increase k by 2, starting with k=3
3. the upper bound for k is $\sqrt{n}$, as there could be only one prime factor greater then $\sqrt{n}$; if n is greater than 1 after dividing out all factors, then n is the greatest factor.

```
func factorOpt(n int) int {
  k := 3
  for n%2 == 0 {
    n /= 2
  }
  for k*k <= n && n > 1 {
    for n%k == 0 {
      n /= k
    }
    k += 2
  }
  if n > 1 {
    return n
  } else {
    return k
  }
}
```

This solution is about 10 times faster than the naive imperative solution. Even more interesting, the solution in Haskell based on lazy lists is on par with this optimized solution ($\approx 2\mu s$).

3.3 Testing

```
factorOutDevidesN :: Integer -> Property
factorOutDevidesN n = n > 1 ==> rem n (factorOut n) == 0

checkPrimesDevidesN :: Integer -> Property
checkPrimesDevidesN n = n > 1 ==> rem n (checkPrimes n) == 0

equalResults :: Integer -> Property
equalResults n = n > 1 ==> checkPrimes n == factorOut n
```

```
main = do
    quickCheck factorOutDevidesN
    quickCheck checkPrimesDevidesN
    quickCheck equalResults
```

```
-- alternative main for benchmarking
main = defaultMain [
  bgroup "lpf" [ bench "factor" $ whnf factorOut   600851475143
               , bench "check"  $ whnf checkPrimes 600851475143
               ]
  ]
```

4 Problem 004: Largest palindrome product

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.

Find the largest palindrome made from the product of two 3-digit numbers.

4.1 Exhaustive Search

If we choose the complete interval $[100, 999]$ of 3-digit numbers for both numbers, we get $900 \cdot 900 = 810,000$ candidates, which will take some time to filter for palindromes. So we have to constrain the generated candidates from this *exhaustive search* to be *good* candidates. Good in this context means that the candidates are more likely to be palindromes in fact.

The first thing to do is to take care that candidates are not generated twice; therefore we constrain n to be greater or equal to m. This reduces candidates from 810,000 to 405,450.

Second, we check if the candidates could be a palindrome by testing if they are divisible by 11. Since 11 is a prime number, one of the factors m or n must be divisible by 11. This reduces candidates to 69,660, which is less than a tenth of the original list; so we are done here.

```haskell
candidates :: [Int]
candidates =
    [m * n | m <- [100..999], n <- [100..999],
    n >= m, m `rem` 11 == 0 || n `rem` 11 == 0]
```

```haskell
reverseNum :: Int -> Int
reverseNum n = run n 0
    where
    run 0 r = r
    run n r = run (n `div` 10) (10 * r + n `rem` 10)
```

```
exhaustiveSearch :: Int
exhaustiveSearch = maximum $ filter (\x -> x == reverseNum x) candidates
```

4.2 Constraining the Generating Function

Multiplying two 3-digits numbers gives numbers with at most six digits ($999 \cdot 999 = 998001$). Thus, we are searching for a 6-digit number, having a '9' in the first place (most significant digit).
The smallest of these numbers is 900900, beeing the product 910×990. Constraining the generating function, we only generate a list of all products from the intervals $[910, 999]$ and $[990, 999]$ and then filter out the ones that are palindromes.

```
constrains :: String
constrains = maximum $ filter (\x -> x == reverse x)
                [show (m * n) | m <- [910..999], n <- [990..999]]
```

This solution is faster than the exhaustive search: we only generated $90 \cdot 10 = 900$ possible solutions. On the other hand, we made the (possibly) wrong assumption that there will be a solution in the interval $[900900, 999999]$, which is actually hard to prove. However, having another solution from the last section, we can convince ourself that both solutions are probably correct.

4.3 Testing

```
main = if read constrains == exhaustiveSearch
    then putStrLn "+++ OK, results match."
    else putStrLn "--- ER, results differ."
```