universität innsbruck

Functional Programming	WS 2021	LVA 703025
Exercise Sheet 6, 10 points		Deadline: Wednesday, November 17, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_06.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Functions on Numbers* **4 p.**

1. Implement a Haskell function `dividesRange :: Integer -> Integer -> Integer -> Bool` that checks whether there is any divisor of a number within a given range: `dividesRange n l u` should be true iff there is some `x` such that `l` $\leq$ `x` $\leq$ `u` and `x` divides `n`. (1 point)

   Example: `dividesRange 629 15 25 == True` since $15 \leq 17 \leq 25$ and 17 divides 629.

   Hint: You can use the built-in functions `div` or `mod` for checking divisibility of two numbers: `div x y` and `mod x y` compute the quotient and the remainder of the integral division of `x` by `y`, respectively. E.g., `div 25 4 = 6` and `mod 25 4 = 1`, since $25 = 6 \cdot 4 + 1$.

2. Implement a Haskell function `prime :: Integer -> Bool` to determine whether a number is prime. Recall: n is a prime number if $n \geq 2$ and n has exactly two divisors. (1 point)

   Example: `prime 7793 == True` and `prime 7797 == False`.

3. Implement a Haskell function `generatePrime :: Integer -> Integer` which takes a number `d` as input and computes a prime number with at least `d` digits. (1 point)

   Valid examples: `generatePrime 4 == 1009` and `generatePrime 8 == 10000019`.

4. How far can you increase `d` such that `generatePrime d` is computed within 1 minute? If this value is below 10, then improve your algorithm `prime`. (1 point)

   Hint: Implement a square root function directly on integers, i.e., without using `sqrt :: Double -> Double`. It does not matter if you implement $\lfloor\sqrt{n}\rfloor$ or $\lceil\sqrt{n}\rceil$. For instance, the integer square root of 27 can either be 5 or 6.

Solution 1

```
divides :: Integer -> Integer -> Bool
divides x y = mod y x == 0

dividesRange :: Integer -> Integer -> Integer -> Bool
dividesRange n l u
  | l > u = False
  | otherwise  = divides l n || dividesRange n (l + 1) u

prime :: Integer -> Bool
prime n = n >= 2 && not (dividesRange n 2 (sqrtIntBisect n))

generatePrime :: Integer -> Integer
generatePrime d = gen (10^(d-1) + 1) where
  gen n
     | prime n = n
     | otherwise = gen (n+2)

-- sqrtIntVariant x computes the floor of sqrt(x)

-- simple version just counts x upwards until x^2 > n
sqrtIntSlow :: Integer -> Integer
sqrtIntSlow n = main 0 where
  main x
    | x * x > n = x - 1
    | otherwise = main (x + 1)

-- bisection version finds correct value by
-- iteratively dividing a search interval [l,r] in two halves
sqrtIntBisect :: Integer -> Integer
sqrtIntBisect x = s 0 x where
  s l u -- we have lower and bounds l and u satisfying l^2 <= x <= u^2
    | u - l > 1 =
        let m = div (u + l) 2;
            m2 = m * m
         in if m2 > x then s l m
            else if m2 < x then s m u
            else m
    | u == l = l
    | otherwise = if u^2 <= x then u else l
```

If `prime` is defined as `prime n = n >= 2 && not (dividesRange n 2 (n - 1))`, then `generatePrime` cannot compute `generatePrime 10` within one minute (using a computer with a 3.1 GHz Intel Core i7). With the current solution `generatePrime 14` is computed in around 1 minute.

Exercise 2 *Heron's method* **2 p.**

Heron's method is an ancient (but very efficient) method to approximate the square root of a given non-negative real number x. It works like this: We recursively define the following sequence of numbers:

$$y_0 = x \qquad\qquad y_{n+1} = \begin{cases} 0 & \text{if } y_n = 0 \\ \frac{1}{2}(y_n + \frac{x}{y_n}) & \text{otherwise} \end{cases}$$

Mathematically, this sequence converges monotonically to $\sqrt{x}$ but never actually reaches it (unless $x = 0$ or $x = 1$), giving successively better and better approximations to $\sqrt{x}$.

However, due to the finite precision of the `Double` type, doing this computation in Haskell, you will always find that at some point $y_{n+1} == y_n$.

Your task is to write a function `heron :: Double -> [Double]` that outputs the sequence of numbers $[y_0, \ldots, y_n]$, where n is the smallest number such that $y_{n+1} == y_n$. (2 points)

Examples:

```
heron 0 == [0.0]
heron 1 == [1.0]
heron 2 == [2.0, 1.5, 1.4166666666666665, 1.4142156862745097,
            1.4142135623746899, 1.414213562373095]
```

Solution 2

```
heron :: Double -> [Double]
heron x = aux x
  where aux y = let y' = if y == 0 then 0 else 0.5 * (y + x / y)
                in  if y' == y then [y] else y : aux y'
```

Exercise 3 *Fibonacci numbers* **4 p.**

The Fibonacci numbers $(a_n)_{n\in\mathbb{N}} = 0, 1, 1, 2, 3, 5, 8, \ldots$ are defined by the recurrence

$$a_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ a_{n-1} + a_{n-2} & \text{otherwise} \end{cases}$$

They can also be computed more efficiently by the following recurrence:

$$a_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \text{ or } n = 2 \\ a^2_{\lfloor \frac{n}{2} \rfloor} + a^2_{\lfloor \frac{n}{2} \rfloor + 1} & \text{if } n \text{ is odd} \\ (2a_{\lfloor \frac{n}{2} \rfloor + 1} - a_{\lfloor \frac{n}{2} \rfloor}) a_{\lfloor \frac{n}{2} \rfloor} & \text{if } n \text{ is even} \end{cases}$$

You will implement the computation of the a_n given a non-negative integer n as an input in three different ways:

1. Write a function `fib :: Integer -> Integer` that computes a_n using the naïve recurrence $a_{n+2} = a_{n+1} + a_n$ and a function `fib' :: Integer -> Integer` that does the same using the more complicated recurrence given above.

   Test your functions on increasingly large values and see how much time they take.
   Explain the results. (2 points)

   Hint:

   - If you run the command `:set +s` in GHCi, it will print how long each evaluation took.[1]
   - If an evaluation takes too long, you can abort it using the key combination `Ctrl + C`.
   - Recall that in Haskell, $\lfloor \frac{n}{2} \rfloor$ is written as `div n 2`. Also note that the following pre-defined functions exist: `even :: Integer -> Bool` and `odd :: Integer -> Bool`

2. Write a function `fibFast :: Integer -> Integer` that does the same as `fib'` but internally remembers all values that have already been computed in a lookup table. Again check how long it takes on increasingly large inputs. (2 points)

   Hint:

   - You will need an auxiliary function

     ```
     fibFastAux :: Integer -> [(Integer, Integer)] -> (Integer, [(Integer, Integer)])
     ```

[1] Note that benchmarking functions in GHCi like this is not particularly accurate for a number of reasons: the code is not compiled but only interpreted, and many optimisations that GHC normally does are not performed. But for the scope of this exercise, this is fine.

that takes a number n and a lookup table (consisting of pairs (i, a_i)) and returns both the result a_n and a (possibly bigger) lookup table. If the pair for n is already in the table, the stored value of a_n should be returned – otherwise the recurrence should be used to recursively compute the value of a_n, and the new pair (n, a_n) is then stored in the table.

- The pre-defined function `lookup :: Eq a => a -> [(a, b)] -> Maybe b` will be useful to lookup values in the table.
- The numbers involved grow *very* fast, so even the printing takes a lot of time. For more consistent results, try showing the number of digits of the result instead of the actual result, e.g. `length (show (fib' 10000))` or `length (show (fibFast 10000))`.

Solution 3

1. For `fib`, every function call spawns two new recursive calls, each decreasing the argument by a constant amount. Thus, we get exponentially many calls (roughly $2^{\texttt{n}}$).

   Consequently, `fib'` is very inefficient and takes fairly long even on small examples like `fib 35`.

   In `fib'`, every function call again spawns two recursive calls, but we (roughly) halve the argument with every call. Thus, we get roughly $2^{log_2\ \texttt{n}} = \texttt{n}$ calls, i.e. linearly many.

   This function also starts to hit performance problems around $\texttt{n} = 2 \cdot 10^6$. But the important observation is that it is much faster than `fib`.

   ```
   fib :: Integer -> Integer
   fib 0 = 0
   fib 1 = 1
   fib n = fib (n - 1) + fib (n - 2)


   fib' :: Integer -> Integer
   fib' n
     | n == 0            = 0
     | n == 1 || n == 2  = 1
     | odd n             = a ^ 2 + b ^ 2
     | even n            = (2 * b - a) * a
     where a = fib' (div n 2)
           b = fib' (div n 2 + 1)
   ```

2. The number of actual recursive calls is hard to estimate here, but some experimentation shows that it is roughly logarithmic in `n`. This is much better than what we had before, and consequently, the function is much faster.

   Note however that this does not mean that the overall running time of `fibFast` is logarithmic - the numbers grow exponentially, so in the end, the running time is still super-linear.

```haskell
fibFast :: Integer -> Integer
fibFast n = fst (fibFastAux n [])
  where
   fibFastAux :: Integer -> [(Integer,Integer)] -> (Integer, [(Integer,Integer)])
   fibFastAux n table | n <= 2 = (if n == 0 then 0 else 1, table)
   fibFastAux n table =
        case lookup n table of
            Just res -> (res, table)
            Nothing ->
              let (a, table1) = fibFastAux (div n 2) table
                  (b, table2) = fibFastAux (div n 2 + 1) table1
                  res = if odd n then a ^ 2 + b ^ 2 else (2 * b - a) * a
              in  (res, (n, res) : table2)

{-
  As a bonus, the following function computes the number of recursive calls
  of "fibFast".
-}
fibFastCalls :: Integer -> Integer
fibFastCalls n = fst (fibFastAux n [])
  where fibFastAux n table | n <= 2 = (1, table)
        fibFastAux n table =
          if elem n table then
            (1, table)
          else
            let (m1, table1) = fibFastAux (div n 2) table
                (m2, table2) = fibFastAux (div n 2 + 1) table1
            in  (m1 + m2 + 1, n : table2)
```