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.

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]
```

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)])
     ```

     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))`.

[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.