lab session 1: Functional Programming

The first lab session of the course *Functional Programming* consists of 9 (small) programming exercises. Each exercise is worth 1 grade point each (you get one grade point for free). No partial points are given for passing some but not all test cases of an exercise. You submit your solutions (per exercise a single ASCII file containing Haskell functions) to Themis (`https://themis.housing.rug.nl`). Note that it is essential that the types of your solutions matches exactly the types that are given in the exercises, otherwise judgment by Themis will fail.

For some of the exercises, solutions can easily be found on the internet. Be warned, that copying those is considered plagiarism (and will be forwarded to the board of examiners)! Moreover, many of these published solutions are programmed in an imperative style.

Exercise 1: Fusc

The function *fusc* takes a non-negative integer and produces a non-negative integer. The function is defined by the following recurrence:

$$\begin{aligned} \mathit{fusc}(0) &= 0 \\ \mathit{fusc}(1) &= 1 \\ \mathit{fusc}(2n) &= \mathit{fusc}(n) \\ \mathit{fusc}(2n+1) &= \mathit{fusc}(n) + \mathit{fusc}(n+1) \end{aligned}$$

Make a function `fusc :: Integer -> Integer` such that the call `fusc n` returns the number *fusc*(`n`). Your program should be able to compute `fusc 10^10000` within 1 second. You can time code in `ghci` by typing `:set +s` at the command prompt. From that point on, the computation time is reported after each computation. However, be warned for caching! Often, doing a caluclation after you did it before yields a cached answer with a computation time close to 0.0 seconds.

Exercise 2: Power Digits

Write a Haskell function `powDigits :: Integer -> Integer -> Int -> Integer` such that the call `powDigits n e d` prints the last `d` digits of `n^e` (i.e. n^e). You should not print leading zeroes. For example, $20^5 = 3200000$, so `powDigits 20 5 6` should return `200000`, while `powDigits 20 5 5` should return `0` (and not `00000`).
The time to compute `powDigits 12345 (10^5000) 100` using `ghci` should not exceed 1 second.

Exercise 3: Poulet Numbers

A *composite number* n is called a *Poulet number* if and only if $2^{n-1} \textbf{ mod } n = 1$. For example, the first Poulet number is 341, because $341 = 11 \times 31$ and $2^{340} \textbf{mod } 341 = 1$.
Write a Haskell function `poulet :: Int -> Int` such that the call `poulet n` returns the `n`th Poulet number. So, `poulet 1` should return `341`.
You may assume that the argument `n` of `poulet` is at most 300. Note that the 300th Poulet number is 1472353. The time to compute `poulet 300` should not exceed 4 seconds.

Exercise 4: Summing Intervals

Write a Haskell function `intervalSums :: Int -> Int` such that the call `intervalSums n` returns the number of pairs (a, b) (with $1 <= a < b$) such that $\sum_{i=a}^{b} i = \mathtt{n}$.

For example, `intervalSums 55` should return `3` because

$$55 = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 9 + 10 + 11 + 12 + 13 = 27 + 28$$

while there are no other consecutive integer sequences that add up to 55.

Your solution should be able to compute `intervalSums 5000000` within three seconds.

Exercise 5: Collatz Records

The famous Collatz problem goes as follows: start with any positive integer number n. If n is even, then divide it by 2, otherwise multiply it by 3 and add 1. If the result is 1, then we stop, otherwise we repeat the process with the newly obtained number. For example, if we start with $n = 6$, we find the following 8-step calculation:

$$6 \rightarrow 3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

The question is, do we always reach 1? This is a famous unsolved problem. It is conjectured that the answer is yes. Nobody has ever found a number n that did not eventually reach 1. The following table show for the first ten integers the number of steps that are required to reach 1:

n	steps
1	0
2	1
3	7
4	2
5	5

n	steps
6	8
7	16
8	3
9	19
10	6

Looking at this table, we see that $n = 2$ is the first number than needs more steps than all numbers $m < n$ (which is trivial, since the only possible value for m is 1). Therefore we call $n = 2$ the *1st Collatz record*. The 2nd Collatz record is 3, the 3rd Collatz record is 6, the 4th is 7, and the 5th is 9. Write a Haskell function `nthCollatzRecord :: Int -> Integer` such that `nthCollatzRecord n` returns the `n`th Collatz record. So, `nthCollatzRecord 5` should return `9`.
The calculation of `nthCollatzRecord 30` should not exceed 4 seconds.

Exercise 6: Palindromic Primes

A *palindromic number* is a number that remains the same when its digits are reversed. A *palindromic prime* is a prime number that is also a palindromic number. The first 20 palindromic numbers are:

$$2, 3, 5, 7, 11, 101, 131, 151, 181, 191, 313, 353, 373, 383, 727, 757, 787, 797, 919, 929$$

The number 11 is the only palindromic number with an even number of digits, since every palindromic number with an even number of digits is divisible by 11.
Write a Haskell function `nthPalPrime :: Int -> Integer` such that `nthPalPrime n` returns the `n`th palindromic prime. So, `nthPalPrime 1` should return `2`, and `nthPalPrime 20` should return `929`.
The time to compute `nthPalPrime 1000` using `ghci` should not exceed 5 seconds.

Exercise 7: Modular Equations

The input for this problem is a series of integer pairs (a_i, m_i), where $0 \leq a_i < m_i$. The output of your program must be the smallest x such that

$$\forall i : \quad x \textbf{ mod } m_i = a_i$$

For example, for the pairs $(42, 534)$, $(312, 6546)$, and $(324, 657)$ the smallest solution is $x = 88161840$ since $88161840 \textbf{ mod } 534 = 42$, $88161840 \textbf{ mod } 6546 = 312$, and $88161840 \textbf{ mod } 657 = 324$.
Write a Haskell function `solveModularEq :: [(Integer,Integer)] -> Integer` such that the call `solveModularEq [(42,534),(312,6546),(324,657)]` returns `88161840`.
Of course, a solution does not always exist. For example, for the pairs $(1, 2)$ and $(2, 4)$ there exists no solution. In that case the output should be the product of the moduli (so, in this case 8). Note that if a solution exists, it is automatically smaller than this product. You may assume that the product of the moduli is at most 10^{10}. Your program should be able to compute an answer within 10 seconds.

Exercise 8: Super Primes

A prime number p is called a *super prime number* if it remains prime by dropping any digit (leading zeros are allowed). For example, the number 4019 is a superprime because 4019 is prime, and so are 019=19, 419, 409, and 401. The first twenty superprimes are:

$$23, 37, 53, 73, 113, 131, 137, 173, 179, 197, 311, 317, 431, 617, 719, 1013, 1031, 1097, 1499, 1997$$

Write a Haskell function `nthSuperPrime :: Int -> Integer` such that the call `nthSuperPrime n` returns the `n`th super prime. So, `nthSuperPrime 1` should return `23`, and `nthSuperPrime 20` should return `1997`. The time to compute `nthSuperPrime 50` using `ghci` should not exceed 10 seconds.

Exercise 9: Semiperfect Numbers

In number theory, a *semiperfect number* number is a natural number n that is equal to the sum of all or some of its *proper divisors*. Note that the number n itself is not considered a proper divisor. For example, 24 is a semiperfect number, because its proper divisors are 1, 2, 3, 4, 6, 8, and 12. The sum 4+8+12=24 shows that 24 is a semiperfect number.
The first few semiperfect numbers are: 6, 12, 18, 20, 24, 28, 30, 36, 40, 42, ...
Write a Haskell function `nthSemiPerfectNumber:: Int -> Int` such that `nthSemiPerfectNumber n` returns the `n`th semiperfect number.
So, `nthSemiPerfectNumber 1` should return `6`, and `nthSemiPerfectNumber 10` should return `42`.
You may assume that the argument `n` of `nthSemiPerfectNumber` is at most 2500. The time to compute `nthSemiPerfectNumber 2500` should not exceed 2 seconds.