lab session 1: Functional Programming

Introduction

The first lab session of the course *Functional Programming* consists of 9 (small) programming exercises. Each exercise is worth 10 points. You are required to submit your solutions (per exercise a single ASCII file containing Haskell functions) to Themis (`https://themis.housing.rug.nl`).

Note that in the first lab Themis determines your grade. However, your grade for lab sessions 2 and 3 will be a combination of points awarded by Themis and points that are awarded by the teaching assistants through manual inspection for (a Haskellish) style of programming and efficiency. Hence, for these lab sessions your grade is not automatically the grade that was awarded by Themis.

For some of the exercises, solutions can easily be found on the internet. Be warned, that copying those is considered plagiarism! Moreover, many of these published solutions are programmed in imperative languages.

Exercise 1: Summing Primes

Make a function `sumPrimesTo :: Integer -> Integer` such that `sumPrimesTo n` returns the sum of all prime numbers that are less or equal to `n`.

Exercise 2: Chebyshev numbers

The *Chebyshev numbers* are defined by the following (Fibonacci-like) recurrence:

$$\begin{aligned} C_0 &= 0 \\ C_1 &= 2 \\ C_n &= 4 \cdot C_{n-1} - C_{n-2} \end{aligned}$$

Write a function `chebyshev` such that the call `chebyshev n` returns the number C_n. The type of the function must be `chebyshev :: Integer -> Integer`.

Your program should be able to compute `chebyshev 5000` 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 calculation after you did it before yields a cached answer with a computation time close to 0.0 seconds.

Exercise 3: Modular Nth Root of Unity

Let n and m be integers that are greater than 1. We call a positive integer x an nth *root of unity* modulo m if x^n **mod** $m = 1$. A mathematician would write this as $x^k \equiv 1 (\textbf{mod}\ m)$.

Write a Haskell function `isNthRootOfUnity :: Integer -> Integer -> Integer -> Bool` such that `isNthRootOfUnity x n m` returns `True` if `x` is an `n`th root of unity modulo `m`, and `False` otherwise.

Exercise 4: Fibonacci Sums

In this exercise, we shift the Fibonacci series by two positions (such that we do not have a zero, and a double occurrence of one in the series). So, we define the (shifted) Fibonacci series as:

$$F_0 = 1, \quad F_1 = 2, \quad F_n = F_{n-1} + F_{n-2}$$

It is well-known that each positive integer can be written as a sum of Fibonacci numbers (in which no Fibonacci number occurs more than once). For example, the number 64 can be written as a sum of shifted Fibonacci numbers in 5 different ways.

$$\begin{aligned} 64 &= F_8 + F_4 + F_0 = 55 + 8 + 1 \\ &= F_8 + F_3 + F_2 + F_0 = 55 + 5 + 3 + 1 \\ &= F_7 + F_6 + F_4 + F_0 = 34 + 21 + 8 + 1 \\ &= F_7 + F_6 + F_3 + +F_2 + F_0 = 34 + 21 + 5 + 3 + 1 \\ &= F_7 + F_5 + F_4 + F_3 + F_2 + F_0 = 34 + 13 + 8 + 5 + 3 + 1 \end{aligned}$$

Write a Haskell function `numberOfFiboSums :: Integer -> Integer` such that `numberOfFiboSums n` returns the number of ways that `n` can be written as a sum of Fibonacci numbers. Note that a sum like $F_0 + F_1$ is considered the same sum as $F_1 + F_0$ (so the order of terms in the sum is irrelevant).

Exercise 5: Totient Function

In number theory, *Euler's totient function* counts the positive integers up to a given integer n that are relatively prime to n. It is written using the Greek letter phi as $\varphi(n)$.

One way of computing $\varphi(n)$ is based on the factorization of n. Let $n = p_0^{e_0} \cdot p_1^{e_1} \cdots p_m^{e_m}$, where $p_0 < p_1 < ... < p_m$ are prime numbers and each $e_i \geq 1$. Then the totient function of n is given by

$$\varphi(n) = p_0^{e_0-1} \cdot (p_0-1) \cdot p_1^{e_1-1} \cdot (p_1-1) \cdots p_m^{e_m-1} \cdot (p_m-1).$$

Write a Haskell function `totient :: Integer -> Integer` such that `totient n` returns φ(`n`).

Exercise 6: Happy numbers

A *happy number* is defined by the following process:

> *S*tarting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers, while those that do not end in 1 are unhappy numbers.

Make a Haskell function `countHappyNumbers :: Integer -> Integer -> Int` such that the call `countHappyNumbers a b` returns the number of happy numbers in the interval `[a,b]`. Your program should be able to compute `countHappyNumbers 1 100000` within 5 seconds.

Exercise 7: Improved Caesar cipher

A Caesar cipher is one of the simplest and most widely known encryption techniques. The method is named after Julius Caesar, who used it in his private correspondence. It is a type of substitution cipher in which each letter in a text is replaced by a letter some fixed number of positions down the alphabet. For example, with a right rotation of 3 positions, A would be replaced by X, B by Y, C by Z, D by A, etc. Coding (and decoding) is easy if the alphabet and the rotated alphabet are placed in alignment:

```
Plain:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:   XYZABCDEFGHIJKLMNOPQRSTUVW
```

In the above example, the *key* of the coder is 3, being the number of positions over which the alphabet is shifted to the right during the coding process. This key is used in the following example.

```
Plaintext:  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
```

Using statistical knowledege about the number of occurrences of letters in a given language, the Caesar cipher is pretty easy to crack. In the *improved Caesar cipher* this is a lot harder. In this scheme the alphabet is rotated by `key` positions again after having used a letter from the alphabet. For example, using the key 3, the text `AAA` is now encoded as follows: the first `A` is encoded as `X`, next the alphabet is rotated again by 3 positions yielding the mapping of the second `A` onto `U`. After another rotation, the third `A` is mapped on `R`, yielding the encoded text `XUR`.
Write the Haskell functions `cipherEncode :: Int -> String -> String` and `cipherDecode` (with the same typoe) that accept two arguments: a small integer (the key) and a string. The functions should yield the coding (or decoding) of the second argument, given the first argument as the key. You may assume that the input consists only of the uppercase letters (A..Z), and spaces (which are not replaced).
So, `cipherEncode 3 "AAA"` should return `"XUR"`, while `cipherDecode 3 "XUR"` should return `"AAA"`.

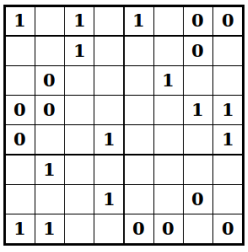
Exercise 8: Takuzu Bitstrings

A *Takuzu* is a logic-based number placement puzzle[1]. The objective is to fill a $n \times m$ grid with 1s and 0s, such that there is an equal number of 1s and 0s in each row and column (hence n and m are even numbers) and no more than two of either number adjacent to each other. Additionally, there can be no identical rows, nor can there be identical columns.

Write a Haskell function `takuzuStrings :: Int -> [String]` such that `takuzuStrings n` generates the list of all bitstrings with length `n` that satisfy the property that there are no more than two 0s or 1s are adjacent to each other. Moreover, the list should be ordered in lexicographic increasing order. The function should not check that the number of 0s equals the number of 1s (we will do that in exercise 9 though). For example, `takuzuStrings 3` should produce the output:

```
["001","010","011","100","101","110"]
```

Exercise 9: Takuzu solver [Hard]



In the previous exercise, the rules are given for a Takuzu puzzle. An example of such a puzzle is given in the figure on the right.
A *correct* Takuzu has one unique solution. An incorrect Takuzu has no solution at all, or it has at least two solutions.
Write a Haskell function `isCorrectTakuzu :: [String] -> Bool` that takes as its input a Takuzu, and produces the output `True` if the Takuzu is a correct puzzle, or `False` otherwise. For example, the call

```
isCorrectTakuzu ["1.1.1.00",
                 "..1...0.",
                 ".0...1..",
                 "00....11",
                 "0..1...1",
                 ".1......",
                 "...1..0.",
                 "11..00.0"]
```

should yield the answer `True`. Note that the input format is a list of lists, one list per row of the grid. A '.' represents an empty cell. In fact the example input corresponds with the Takuzu in the above figure (and is downloadable from Themis).

[1] In Dutch newspapers, this type of puzzles is often called a *Binaire* or *Binairo*.