universität innsbruck

Functional Programming WS 2021 LVA 703025

Exercise Sheet 9, 10 points Deadline: Wednesday, December 15, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_09.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *The Caesar cipher* **6 p.**

The well known Caesar cipher[1] encodes a string by shifting each character n times (for some n).

1. Write a function `shift :: Int -> Char -> Char` that applies a shift factor to a lower-case letter (letters `'a'` – `'z'`). Other characters should be ignored by your function.

   Hint: `fromEnum` and `toEnum` can be used to convert between characters and `Int` values (corresponding to Unicode).

   Examples: `shift 5 'a' = 'f'`, `shift 21 'f' = 'a'`, `shift 5 '.' = '.'`

   Using `shift` and list-comprehensions define a function `encode :: Int -> String -> String` shifting each lower-case letter in a given string.

   Example: `encode 5 "here is an example." = "mjwj nx fs jcfruqj."` (1 point)

2. The key to having a program crack the Caesar cipher is the observation that some letters appear more frequently than others in English text. Below is a list of approximate frequencies (in percent) of the 26 letters of our alphabet (source: https://en.wikipedia.org/wiki/Letter_frequency):

   ```
   freqList = [8.2, 1.5, 2.8, 4.3, 13, 2.2, 2, 6.1, 7, 0.15, 0.77, 4, 2.4, 6.7,
               7.5, 1.9, 0.095, 6, 6.3, 9.1, 2.8, 0.98, 2.4, 0.15, 2, 0.074]
   ```

   If we measure how well a given frequency distribution matches up with the expected distribution, e.g. by using the chi-squared statistic, we can choose the shift factor that produces the best match for our decoding. Implement the following functions to help you achieve this task in the next item.

   (a) `count :: Char -> String -> Int` which returns the number of occurrences of a particular character in a string and `percent :: Int -> Int -> Float` that calculates the percentage of one integer with respect to another. (1 point)

   Examples: `count 'e' "example" = 2`, `percent 1 3 = 33.333336`

   (b) `freqs :: String -> [Float]` which computes the list of frequencies for a given string.

   Hint: `['a'..'z']` produces a list of the 26 lower-case letters. (1 point)

   Example: `freqs "abbcccdddd" = [10.0,20.0,30.000002,40.0,0.0,...,0.0]`

   (c) `chisqr :: [Float] -> [Float] -> Float` which, given a list of observed frequencies os and expected frequencies es, computes the *chi-square statistic*: (1 point)

   $$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

   Note that smaller results of the chi-square statistic indicate better matches between observed and expected frequencies.

[1] https://en.wikipedia.org/wiki/Caesar_cipher

(d) `rotate :: Int -> [a] -> [a]` which rotates the elements of a list n places to the left, wrapping around the end of the list, and assuming that $0 \leqslant n \leqslant$ length of the list and the function `positions :: Eq a => a -> [a] -> [Int]` which returns the list of all positions at which a value occurs in a list. (1 point)

Hint: For `positions` first pair all elements in the list with their position using `zip`.

Examples: `rotate 3 [1,2,3,4,5] = [4,5,1,2,3]`, `positions 3 [3,1,3,3] = [0,2,3]`

3. Write a function `crack :: String -> String` which attempts to decode a Caesar cipher-encoded string by first computing the frequency list of the string, then calculating the chi-square statistic of each possible rotation of the frequency list with respect to the frequencies given in `freqList`, and finally taking the position of the minimum chi-square value as the shift factor for decoding. (In the unlikely case that there are multiple minimum positions, simply pick one.) (1 point)

   Use your cracking function to decode the following text:

   rkcuovv sc pex

Solution 1

```
shift :: Int -> Char -> Char
shift n c
  | c >= 'a' && c <= 'z' = toEnum $ (fromEnum c - fromEnum 'a' + n) `mod` 26 +
(fromEnum 'a')
  | otherwise = c

encode :: Int -> String -> String
encode n xs = [shift n x | x <- xs]

freqList = [8.2, 1.5, 2.8, 4.3, 13, 2.2, 2, 6.1, 7, 0.15, 0.77, 4, 2.4, 6.7,
            7.5, 1.9, 0.095, 6, 6.3, 9.1, 2.8, 0.98, 2.4, 0.15, 2, 0.074]

count :: Char -> String -> Int
count c xs = length $ filter ((==) c) xs

percent :: Int -> Int -> Float
percent n m = fromIntegral n / fromIntegral m * 100

freqs :: String -> [Float]
freqs xs = [percent (count c xs) n | c <- ['a'..'z']]
  where n = length (filter (\c -> c >= 'a' && c <= 'z') xs)

chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [(o - e)^2 / e | (o, e) <- zip os es]

rotate :: Int -> [a] -> [a]
rotate n xs = drop n xs ++ take n xs

positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x', i) <- zip xs [0..], x' == x]

crack :: String -> String
-- factor is assumed to be the shift factor used in the encoding,
-- so for decoding we use it as negative shift
crack xs = encode (-factor) xs
  where
    factor = head $ positions (minimum chis) chis
    chis = [chisqr (rotate n table) freqList | n <- [0..25]]
    table = freqs xs

encString = "rkcuovv sc pex"
-- crack encString == "haskell is fun"
```

Exercise 2 *Bernoulli numbers* **4 p.**

The Bernoulli numbers are a sequence of rational numbers defined like this:

$$B_0 = 1 \qquad\qquad B_n = \sum_{k=0}^{n-1} \binom{n}{k} \frac{B_k}{k-n-1} \text{ if } n > 0$$

Here, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ denotes the binomial coefficient, where $n! = 1 \cdot \ldots \cdot n = \prod_{i=1}^{n} i$ denotes the factorial. The following table lists the first few values of B_n:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	...
B_n	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$	0	$-\frac{1}{30}$	0	$\frac{5}{66}$	0	$-\frac{691}{2730}$	...

Note that `Rational` is the type of rational numbers from the Haskell standard library. It has all the type class instances you would expect from it, including most notably a division operator `(/)` and a conversion from integers to rationals `fromInteger :: Integer -> Rational`. There is also the operator

```
(%) :: Integer -> Integer -> Rational
```

that turns a numerator a and denominator b into the rational number $\frac{a}{b}$, i.e. `1 % 2` corresponds to $\frac{1}{2}$.

1. Write functions `fact :: Integer -> Integer` and `binom :: Integer -> Integer -> Integer` that compute the factorial (resp. binomial coefficients) for non-negative inputs. (1 point)

2. Write a function `bernoulli :: Integer -> Rational` such that `bernoulli n` $= B_n$. What is the largest n for which your function still finishes in a reasonable amount of time? (1 point)

   Example: `map bernoulli [0..6] == [1%1, (-1)%2, 1%6, 0%1, (-1)%30, 0%1, 1%42]`

3. Write a function `bernoullis :: Integer -> [Rational]` that, given an integer $n \geq 0$, computes the list of the Bernoulli numbers B_0 to B_n, i.e. `bernoullis n == map bernoulli [0..n]`. Implement it in a more efficient way than just calling `bernoulli` n times! Avoid recomputing results that you have already computed! (1 point)

4. Looking at the sequence of Bernoulli numbers, it seems that starting with B_3, every B_i with i odd is 0. It also seems that in the sequence of the remaining ones, the sign keeps alternating in every step (i.e. B_2, B_6, B_{10}, etc. are positive, B_4, B_8, B_{12}, etc. are negative).

   Write two Haskell functions `check1 :: Integer -> Bool` and `check2 :: Integer -> Bool` that check whether these conjectures are true for all B_n up to a given number n! (1 point)

Trivia: Ada Lovelace is widely credited with having written the first non-trivial computer program in 1842 – and it was an algorithm for computing Bernoulli numbers!

Solution 2

```haskell
fact :: Integer -> Integer
fact n = product [1..n]

binom :: Integer -> Integer -> Integer
binom n k = fact n `div` (fact k * fact (n - k))
-- alternatively: binom n k = product [n-k+1..n] `div` fact k

bernoulli :: Integer -> Rational
bernoulli 0 = 1
bernoulli n = sum [bernoulli k * (binom n k % (k - n - 1)) | k <- [0..n-1]]
-- "bernoulli 22" already takes over 10 seconds to compute, and the time seems to roughly double with e

bernoullis :: Integer -> [Rational]
bernoullis 0 = [1]
bernoullis n = bs ++ [b]
  where bs = bernoullis (n - 1)
        b  = sum [bk * (binom n k % (k - n - 1)) | (k, bk) <- zip [0..n] bs]
-- with this, we can compute e.g. "last (bernoulli 500)" within about 5 seconds

-- One can also define the infinite list of all Bernoulli numbers fairly elegantly:
bernoullis' :: [Rational]
bernoullis' = 1 : map f [1..]
  where f n = sum [bk * (binom n k % (k - n - 1)) | (k, bk) <- zip [0..n-1] bernoullis']

-- Transforms a list [x0, x1, x2, ...] into the list [x0, x2, x4, ...]
dropEveryOther :: [a] -> [a]
dropEveryOther (x : y : xs) = x : dropEveryOther xs
dropEveryOther xs = xs

check1 :: Integer -> Bool
check1 n = all (== 0) bs
  where bs = tail (dropEveryOther (tail (bernoullis n)))

check2 :: Integer -> Bool
check2 n = all (\(x, y) -> signum x == -signum y) (zip bs (tail bs))
  where bs = tail (dropEveryOther (bernoullis n))
```