# EXTRA : Caesar Cipher

**Note:** For this assignment you may find some functions from the `Data.Char` library useful.

Julius Caesar before sending his messages often encoded them, by replacing each letter by the letter three places further down in the alphabet (wrapping around at the end of the alphabet). For example, the string

```
"haskell is fun"
```

would be encoded as

```
"kdvnhoo lv ixq"
```

In general, we can do even more than Caesar did, and encode our strings using any integer between 1 and 25 (since the alphabet has 26 letters), having 25 different ways of encoding a string. For example, with a shift factor of 10, the original string would be encoded as:

```
"rkcuovv sc pex"
```

## Encoding and Decoding

For simplicity, in this exercise we will only encode lowercase letters, leaving all other letters unchanged. Function `let2int` converts a lowercase letter between `'a'` and `'z'` to an integer from `0` to `25`, and function `int2let` does the inverse:

```
let2int :: Char -> Int
let2int c = ord c - ord 'a'

int2let :: Int -> Char
int2let n = chr (ord 'a' + n)
```

(The library functions `ord :: Char -> Int` and `chr :: Int -> Char` convert a character to its unicode representation and vice-versa) For example:

```
Main> let2int 'a'
0

Main> int2let 0
'a'
```

- Define function `shift :: Int -> Char -> Char`, which applies a shift factor to a lowercase letter and leaves any other character unchanged (**Hint:** Use the above functions, as well as function `mod` to ensure that the resulting integer representation does not exceed 26).
- Define function `encode :: Int -> String -> String` by means of function `shift`, which, given a shift factor, encodes a whole string.

**Examples**

```
Main> shift 3 'a'
'd'

Main> shift 3 'z'
'c'

Main> shift (-3) 'c'
'z'

Main> shift 3 ' '
' '

Main> encode 3 "haskell is fun"
"kdvnhoo lv ixq"

Main> encode (-3) "kdvnhoo lv ixq"
"haskell is fun"
```

Note that there is no need for a "decode" function, since if a string is encoded using a shift factor `n`, we can always take it back be re-encoding it using `(-n)` as a shift factor.

## Frequency Tables

The key to crack the Caesar cipher is the observation that some letters of the English alphabet appear more often than others. In fact, by analyzing a large volume of text, one can derive the following table of approximate percentage frequencies of the 26 letters of the alphabet:

```
table :: [Float]
table = [ 8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0, 6.1, 7.0, 0.2, 0.8, 4.0, 2.4
        , 6.7, 7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1 ]
```

For example, letter `'e'` occurs most often, with a frequency of 12.7%, while `'q'` and `'z'` appear least often, with a frequency of 0.1% each.

- Define function `percent :: Int -> Int -> Float` which computes the percentage of an integer with respect to another (**Hint:** Use library function `fromIntegral :: (Integral a, Num b) => a -> b` to convert the arguments to `Float` before dividing them). For example:

  ```
  Main> percent 6 12
  50.0

  Main> percent 3 15
  20.0
  ```

- Define function `freqs :: String -> [Float]` which computes the frequencies of the 26 letters of the alphabet for a given string. Assume that the given string will contain at least one lowercase letter. For example:

```
Main> freqs "abbcccdddeedeee"
[ 6.7, 13.3, 20.0, 26.7, 33.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 ]
```

  That is, letter `'a'` appears with frequency 6.7, letter `'b'` with frequency 13.3 and so on.

## Cracking The Cipher

Now that we have laid the foundations, it is time to crack Caesar's Cipher.

A standard method for comparing a list of observed frequencies `o` with a list of expected frequencies `e` is the *chi-square* statistic, defined as follows:

$$\texttt{chisqr o e} = \sum_{i=0}^{n-1} \frac{(\texttt{o}_i - \texttt{e}_i)^2}{\texttt{e}_i}$$

The details of the chi-square method are not important to us, only the fact that the smaller the result of `chisqr o e`, the better the match between frequency tables `o` and `e`.

- Implement function `chisqr :: [Float] -> [Float] -> Float`. **Hint:** this exercise can be easily solved using the `zip` function and list comprehensions.
- Implement function `rotate :: Int -> [a] -> [a]`, which rotates the elements of a list a given number of times to the left. For example:

```
Main> rotate 3 [1,2,3,4,5]
[4,5,1,2,3]
```

  You can assume that the integer argument is always between 0 and the length of the list. **Hint:** Use functions `take` and `drop` to implement this exercise.

Now, if we are given an encoded string but not the shift factor used for the encoding, we can find the shift factor as follows:

1. We produce the frequency table of the encoded string
2. We calculate the chi-square statistic for each possible rotation of this table with respect to the expected frequencies (value `table`)
3. The position of the minimum chi-square value is the most probable shift-factor used to encode the string.

For example, if `table' = freqs "kdvnhoo lv ixq"`, then

```
[ chisqr (rotate n table') table | n <- [0..25] ]
```

gives the result

$$[1408.8, 640.3, 612.4, 202.6, 1439.8, 4247.2, 651.3, \cdots, 626.7]$$

, in which the minimum value is 202.6, appearing in position 3 in this list (counting from 0). Hence, we can conclude that the shift factor used to encode the string was 3, and to retrieve the original string, we just need to encode it again using -3.

- Define function `crack :: String -> String` which takes an encoded string and, using the above method, computes the original string. **Hint:** In addition to all the functions you have already defined, you will also find functions `minimum :: Ord a => [a] -> a` and `elemIndex :: Eq a => a -> [a] -> Maybe Int` useful for solving this exercise (function `elemIndex` is defined in the `Data.List` library).

**Examples**

```
Main> crack "kdvnhoo lv ixq"
"haskell is fun"

Main> crack "vscd mywzboroxcsyxc kbo ecopev"
"list comprehensions are useful"
```

Note that cracking is not always accurate, especially in cases where the encoded word is too short, or has an unusual distribution of letters.

```
Main> crack (encode 3 "haskell")
"piasmtt"

Main> crack (encode 3 "boxing wizards jump quickly")
"wjsdib rduvmyn ephk lpdxfgt"
```