Problem Sheet 2.1: Recursion in Haskell

Recursion on Strings

We will implement some simple functions on strings. Remember, strings in Haskell are written between double quotation marks, like `"this"` whereas single characters are written between single quotation marks like `'t' 'h' 'i' 's'`. In the first exercise, we will adapt some standard functions from the `Data.Char` module already loaded at the top of your Haskell file.

a) The function `toUpper :: Char -> Char` changes lower case letters to upper case letters, leaving all other characters the same. Using `toUpper`, complete the recursive definition of `toUpperSt`, which changes all the letters in a string to upper case.

```
*Main> toUpperSt "Hello World"
"HELLO WORLD"
```

b) The function `isDigit :: Char -> Bool` returns `True` for recursive function `deleteDigits` which deletes all digits from a given string.

```
*Main> deleteDigits "Hello World 2020"
"Hello World "
```

c) Write the function `leetSpeak`, which converts strings to a basic form of "leetspeak". It should:

- replace all 'e's with '7's,
- replace all 'o's with '0's,
- replace all 's's with 'z's,
- add an '!' to the end of every string.

Type and implement the function `leetSpeak`. If you would like, you can add extra features to your `leetSpeak` function.

```
*Main> leetSpeak "Hello Worlds"
"H7ll0 W0rldz!"
```

Recursion on Numbers

In these exercises, we will build up to defining a function `factors` that, given an integer, outputs a list of its factors.

a) Every number can be expressed as $n = 2^k a$, where a is an odd number. The function `factors2` takes an integer `n` as input, outputting a list of k `2`s, followed by `a`.

```
*Main> factors2 72
[2,2,2,9]
```

In fact, for any number m, we can express n as $m^k a$, where a is not divisible by m. Adapt `factors2` to a function `factorsm` of type `Int -> Int -> [Int]`, so that given two integers `m` and `n`, `factorsm` outputs a list of k `m`s followed by `a`.

```
*Main> factorsm 3 72
[3,3,8]
```

b) The function `factorsFrom :: Int -> Int -> Int` expands on `factorsm`. Given two input integers `m` and `n` with $n = m^k a$, as before, it should output a list of k `m`s, but then, instead of `a`, if $a = (m+1)^{k_1} a_1$, with a not divisible by $m+1$, the next elements of the list should be k_1 copies of $m+1$, and so on until $m+i$ is greater than or equal to a_i. Complete the definition of `factorsFrom`.

```
*Main> factorsFrom 3 120
[3,4,5,2]
```

c) We can use the function `factorsFrom` to implement a function `primeFactors` that given an integer, outputs a list of its prime factors. Type and implement primeFactors. If you can, try and implement this function using partial application of `factorsFrom`.

```
*Main> primeFactors 120
[2,2,2,3,5]
```