Exercise 1: Searching patterns inside DNA sequences

In this exercise, we will represent a sequence of DNA bases as a Haskell string consisting of the four letters G C A T denoting one of the four possible nucleotides:

- G = Guanine
- C = Cytosine
- A = Adenine
- T = Thymine

Typically, we will use the string "GAACAT" to represent the sequence of DNA bases appearing on the left strand of the DNA helix below.

Given such a sequence "GAACAT" of DNA bases on the left strand, we know that the sequence of DNA bases on the right strand is "CTTGTA". The reason is that every DNA base on the left strand comes with a DNA base on the right strand to form a base pair A-T or G-C, as explained and depicted in the figure above.

§1.a. Write a Haskell function `dualStrandSeq` which takes a sequence of DNA bases and produces the sequence of DNA bases on the opposite strand. The code should typically behave in the following way:

```
ghci> dualStrandSeq "GAACAT"
"CTTGTA"
```

§1.b. Write a Haskell function `isPrefix` which takes two strings `str1` and `str2` and checks whether the first string is a prefix of the second one. The `isPrefix` function should typically behave in the following way:

```
ghci> isPrefix "GAA" "GAACAT"
True
ghci> isPrefix "GAC" "GAACAT"
False
```
§1.c. A substring is a contiguous sequence of characters within a string. Write a Haskell function isSubString which takes two strings str1 and str2 and checks whether the first string is a substring of the second string. The isSubString function should typically behave in the following way:

ghci> isSubString "CAT" "GAACATA"
True
ghci> isSubString "GAT" "GAACATA"
False
ghci> isSubString "GATT" "GAACATA"
False

§1.d. A subsequence is a not necessarily contiguous sequence of characters within a string. Write a Haskell function isSubSeq which takes two strings str1 and str2 and checks whether the first string is a subsequence of the second string. The isSubSeq function should typically behave in the following way:

ghci> isSubSeq "GAAATA" "GAACATA"
True
ghci> isSubSeq "GATAA" "GAACATA"
False

§1.e. Write a Haskell function listOfSubString which takes a string str and returns the list of its substrings.

§1.f. Write a Haskell function listOfSubSeq which takes a string str and returns the list of its subsequences.

§1.g. Suppose given a pair of strings

\[ u = a_1 \ldots a_m \quad \text{and} \quad v = b_1 \ldots b_n \]

consisting of m letters \(a_1, \ldots, a_m\) and of n letters \(b_1, \ldots, b_n\), respectively. An embedding of the string \(u = a_1 \ldots a_m\) in the string \(v = b_1 \ldots b_n\) is defined as a strictly increasing function

\[ \varphi : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\} \]

such that \(a_i = b_{\varphi(i)}\) for every index \(i \in \{1, \ldots, m\}\) of a letter \(a_i\) in the string \(u\). Observe that the string \(u = a_1 \ldots a_m\) is a subsequence of the string \(v = b_1 \ldots b_n\) in the sense of §1.d. precisely when there exists an embedding \(\varphi\) of \(u\) in \(v\). Note also that every strictly increasing function \(\varphi : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}\) can be represented as a strictly increasing list

\[ \varphi = [\varphi(1), \ldots, \varphi(m)] \]

of \(m\) integers between 1 and \(n\). Write a Haskell function listOfEmb which takes two strings str1 and str2 and returns the list (which can be empty) of the embeddings of str1 in str2. The function should behave in that way:
**Exercise 2 : Sieve of Eratosthenes**

The sieve of Eratosthenes is a well-known algorithm to compute the list of prime numbers. You will find a detailed description of the algorithm here:

**Sieve of Eratosthenes**

§2.a. Define a Haskell function `eratoSieve` which takes as argument an integer `m` and a list `intList` of integers, and returns the list `intList` where all the multiples of the integer `m` have been removed.

§2.b. Define a Haskell function `listOfPrimes` which takes an integer `n` as argument and returns the list of all the prime numbers smaller than `n`. The function should behave in that way:

```haskell
ghci> listOfPrimes 80
```

§2.c. Define a Haskell function `listOfPrimesPowerOfTwoMinusOne` which takes an integer `n` as argument and returns the list of all the prime numbers `p` smaller than `n` and of the form `p = 2^k - 1` for a specific integer `k`. The function should behave in that way:

```haskell
ghci> listOfPrimesPowerOfTwoMinusOne 1000000
[3,7,31,127,8191,131071,524287]
```

§2.d. Using the function `listOfPrimes`, define a Haskell function `decompInPrimes` which takes an integer `n` as input and factors it into a product of prime numbers:

\[ n = p_1 \times \cdots \times p_k \]

where each number `p_i` is a prime number (in particular `p_i \geq 2`) and the sequence is increasing in the sense that `p_i \leq p_j` whenever `i \leq j`. In order to express this finite sequence of prime numbers, the function `decompInPrimes` will be designed to return the list `[p_1, \ldots, p_k]` of integers. The function will typically behave as follows:

```haskell
ghci> decompInPrimes 21
[3,7]
ghci> decompInPrimes 24
[2,2,2,3]
```

§2.e. [optional] A number is called perfect when its is equal to the sum of its proper divisors. Typically, 6 is a perfect number because its proper divisors are 1, 2, 3 and 6 = 12 + 3. You will find more information and examples on that website: Perfect numbers. Write a function `listOfPerfectNumbers` which takes an integer `n` as argument and returns the list of all perfect numbers smaller than `n`.\]