CSCE 314: Programming Languages

Homework Assignment 4

Objective

This assignment involves use of functions in Haskell.

Submission

- Like before, this homework is will be graded as a 0, 1, or 2 for each question. (The questions are: "Chinese Remainder Theorem", "k-composite Numbers and Anagram Codes", "Jugs Problem"). With 0 = inadequate or no attempt; 1 = solid effort showing a good attempt at all questions; 2 = faultless/exemplary submission. You may find it helpful to discuss your homework with the peer teachers, TAs, or instructors (say, at their office hours).
- Prepare a **single** Haskell file with a `.hs` extension.
- *Recommended*: Include as a comment at the top of your code: your name and UIN.
- The file should contain functions named precisely as described below. Those functions must have proper type signatures and should follow exactly the ones that have been explicitly given. In addition, you may include any number of additional helper functions you like; you may not include any extra modules, packages, or external libraries.
- Additional documentation in the form of comments throughout your code is strongly encouraged. If you have used outside resources, document these directly in your source file.
- You are encouraged to use the class slack to communicate about/solicit help/ask questions clarifying aspects of the assignment.
- Submission will be via Canvas, due: 1 November 2021.

Instructions

- This assignment asks you to implement a set of Haskell functions. Each of those functions can be defined with just a few lines of code. If your implementations start to get much longer, it may be a sign that there are less complicated solutions that you may be missing.
- Also, you shouldn't need any functionality beyond items that are in the standard prelude. Appendix B in the textbook for a list of those functions.

Note:
There's a lot of description in this assignment, and it may take you multiple re-reads to get through it all. Don't get discouraged! The actual core of the assignment is really pretty short. It does contain some basic number theory, which you can read bit-by-bit at your own pace. And also remember to feel free to ask questions on `#hw-4` if you feel confused.

Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT) is a theorem with many uses, including as part of most RSA encryption implementations. The basic idea is as follows:

Given a number **x**, and a set of numbers $\mathbf{n_i}$, we can compute a set of remainders $\mathbf{a_i}$ modulo the $\mathbf{n_i}$. That is:

$\mathbf{a_0}$ = **x** mod $\mathbf{n_0}$
$\mathbf{a_1}$ = **x** mod $\mathbf{n_1}$
etc.

The Chinese Remainder Theorem states that for a given set of $\mathbf{a_i}$ and $\mathbf{n_i}$ (assuming the $\mathbf{n_i}$ are pairwise coprime), there is a unique number **x** less than the product of the $\mathbf{n_i}$, which satisfies $\mathbf{a_i}$ = **x** mod $\mathbf{n_i}$ for all **i**. For example, if

2 = **x** mod **7**
1 = **x** mod **5**
0 = **x** mod **3**

Then there is exactly one number less than 7×5×3=105 that meets all these criteria. There are a variety of algorithms for finding this number, but we only focus on the brute force algorithm in this question.

The brute force algorithm notes that for each ($\mathbf{a_i}$, $\mathbf{n_i}$) pair, there is an infinite sequence of numbers that satisfy that value. If you form a sequence of these numbers for each pair in ascending order, the smallest number that appears in all sequences is your answer. For example, given the pairs (2,7), (1, 5), and (0, 3):

2 = **x** mod **7** means **x** is one of: {2, 9, 16, 23, 30, 37, 44, 51, 58, 65, 72, 79, 86, 93, 100, ...}
1 = **x** mod **5** means **x** is one of: {1, 6, 11, 16, 21, 26, 31, 36, 41, 46, 51, 56, 61, 66, 71, 76, 81, 86, ...}
0 = **x** mod **3** means **x** is one of: {3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, ...}

Examining those lists will show that **51** is the smallest number and also the only number less than **105** that is so. (Note that for **x** greater than 105, we would still be able to say that **51** = **x** mod **105**).

An alternate way is that you could take those lists in pairs: by combining the first two lists, you would find that **16** = **x** mod **35**. Then, that would form a sequence {16, 51, 86, 121, ...}, which, when combined with the sequence corresponding to **0** = **x** mod **3**, would yield that **51** = **x** mod **105**.

Write a function, `crt`, that takes as input a list of tuples, ($\mathbf{a_i}$, $\mathbf{n_i}$), each with a remainder $\mathbf{a_i}$ and a number $\mathbf{n_i}$. You can assume all the $\mathbf{n_i}$ are pairwise coprime. From them, you should produce a new tuple, (**a**, **n**), where **n** is the product of the $\mathbf{n_i}$, and **a** is **x** mod **n**, or the smallest number that meets the criteria.

```
crt :: [(Integer, Integer)] -> (Integer, Integer)
```

As an example:

```
> crt [(2, 7), (0, 3), (1, 5)]

(51, 105)
```

Note: numbers are coprime if they do not have any common prime factors. So 10 = 2×5 and 21 =3×7 are coprime. The Chinese Remainder Theorem also has a less restrictive formulation in which the numbers are not required to be coprime, but you can assume these will all be coprime in this question.

Assumptions: You can assume that the input list will be non-empty, so you may treat the empty list how ever you choose. In the list, the first element for each tuple will be non-negative and the second can safely be assumed to be positive.

Aside: CRT Application: Secret Sharing System

In the film "Once Upon a Time in America", part of the plot hangs around a scenario in which the hero, Noodles, and his friends have a third party, Mo, keep a key. Noodles tells Mo that he is only to release the key when all of them appear.

A generalization of this, and perhaps one that is more useful in daily life, might have **m** people sharing a secret **S** that can only be unlocked when at least **k** are together. The CRT can be used to design this sort of secret sharing system. The basic idea is: 1) choose **m** coprime integers $n_1 < n_2 < \ldots < n_m$, such that $n_{m-k+2} \times \ldots \times n_m < S < n_1 \times n_2 \times \ldots \times n_k$, i.e., **S** is smaller than the product of any **k** of these integers, but at the same time is greater than any **k-1** of them. Then each person keeps a part, $a_1, a_2, \ldots, a_n$, where a_i is computed as $a_i = S \bmod n_i$. Using this one may determine **S**, uniquely, from any set of **k** or more shares, but not from fewer than **k**. For more details, please see here.

k-composite Numbers and Anagram Codes

Suppose a natural number **n** has the set of factors $\{1, f_1, f_2, \ldots, f_k, n\}$ where its factors have been written in increasing order so that the trivial factors, **1** and **n**, sandwich the rest. Since we have **k** non-trivial factors, we say that **n** is *k*-composite. And, thus, prime numbers are the special case of 0-composite numbers. (And, following standard convention 1 is not considered prime, so not a 0-composite number.)

k-composites

Write a function using list comprehension that, given some non-negative *k* produces the ordered (infinite) list of positive *k*-composite numbers.

```
kcomposite ::  Integer -> [Integer]
```

Here is an example of the function being called:

```
*Main> take 5 $ kcomposite 2
[6,8,10,14,15]
```

A simple anagram code

Hint: For this problem, you may want to use a previous solution.

The following describes a simple code that children sometimes use to exchange secret messages based on shuffling letters. The anagrams of a word are made by rearranging the letters that make up the word. Here you'll be shuffling whole sentences including the spaces between the words. Given a piece of text, called *plaintext*, the encoding procedure involves the following steps:

1. If the number of letters in the plaintext sentence is 2-composite — fine.
2. If not, take it up to the next 2-composite length by adding *X*'s to the end. This is called padding.
   - For example: "*LEND ME FIVE BUCKS*" has 18 characters, and 18 isn't 2-composite. The next 2-composite number is 21, so we add three *X*'s: *LEND ME FIVE BUCKSXXX*
3. The next step is to arrange the letters in a block. Since there are 21 letters, and 21 = 3 × 7, we write the letters in 3 rows, with 7 letters in each row. (The people communicating agree beforehand to have the larger number across.)

L	*E*	*N*	*D*		*M*	*E*
	F	*I*	*V*	*E*		*B*
U	*C*	*K*	*S*	*X*	*X*	*X*

4. The message you send comes from reading down the columns:
   L UEFCNIKDVS EXM XEBX

Implement a function to encrypt messages in the anagram code:

```
anagramEncode :: [Char] -> [Char]
```

Here's an example:

```
> anagramEncode "Chig-gar-roo-gar-rem/Chig-gar-roo-gar-rem/Rough Tough! Real Stuff!"

"Cihhg i-Tggo-augrga-hrr!-o roRo-eoga-algr a-Srrt-eurmfe/fmR!/oXCuXhgX"
```

Now, implement a function to decrypt messages in the anagram code. (You can assume that the pre-padded plaintext does not end with any X's, so you should strip the trailing padding off the text; the input to the decode has length that is 2-composite.)

```
anagramDecode :: [Char] -> [Char]
```

Here's one to test on:

```
"Tt hroeovrneegr  tithshr aontwo n i ctba ysc tamlnoenn oestyo X bXseX"
```

Die Hard 3: Jugs Problem

In the film "Die Hard 3", to stop a bomb exploding in a park, the heros John McClain and Zeus have to solve a "Jugs Problem" proposed by the evil Peter Krieg: given two jugs with capacities `3` and `5` gallons, make exactly `4` gallons from these two jugs.

In fact, the "Jugs Problem" can be generalized as follows: given two jugs with non-negative integer capacities `x` and `y`, determine whether it is possible to make exactly `z` units of water using those two jugs. The `z` units of water must be contained within one or both jugs in the end.

Note: We assume there is an supply of water available. You can fill any of the jugs completely, empty any of the jugs, or transfer water from one jug into another till the other jug is full or the first jug itself is empty. The jugs are oddly shaped, so filling up exactly "half" (or some fractional bit) of the jug is impossible.

Implement a Haskell function that takes `x`, `y` and `z` as inputs and outputs "True" if `z` is measurable using `x` and `y`; otherwise, output "False".

```
 measureWater :: Int -> Int -> Int -> Bool
```

Here is an example of the function being called:

```
*Main> measureWater 3 5 2
True
```

Acknowledgement

Chinese reminder theorem: https://en.wikipedia.org/wiki/Chinese_remainder_theorem
Secret sharing: https://en.wikipedia.org/wiki Secret_sharing_using_the_Chinese_remainder_theorem
The idea for anagram codes comes from a pamphlet entitled "codes" written by Ray Hemmings.
Jugs Problem is adapted from Leetcode Problem 365. You can watch the plot in Die Hard 3 here.