CSCI 490 problem set 1

When solving the homework, strive to create not just code that works, but code that is stylish and concise. See the style guide on the website for some general guidelines. Try to write small functions which perform just a single task, and then combine those smaller pieces to create more complex functions. Don't repeat yourself: write one function for each logical task, and reuse functions as necessary.

Validating Credit Card Numbers[1]

Most credit card providers rely on a checksum formula, the *Luhn algorithm* (`http://en.wikipedia.org/wiki/Luhn_algorithm`), as a straightforward line of defense against simple typos. (It does not protect against *fraud*.)

In this section, you will implement the validation algorithm for credit cards. It follows these steps:

- Double the value of every second digit beginning from the right. That is, the last digit is unchanged; the second-to-last digit is doubled; the third-to-last digit is unchanged; and so on. For example, `[1,3,8,6]` becomes `[2,3,16,6]`.
- Add the digits of the doubled values and the undoubled digits from the original number. For example, `[2,3,16,6]` becomes `2+3+1+6+6 = 18`.
- Calculate the remainder when the sum is divided by 10. For the above example, the remainder would be 8.

If the result equals 0, then the number is valid.

[1] Adapted from the first practicum assigned in the University of Utrecht functional programming course taught by Doaitse Swierstra, 2008-2009.

Exercise 1 We need to first find the digits of a number. Define the functions

```
toDigits    :: Integer -> [Integer]
toDigitsRev :: Integer -> [Integer]
```

`toDigits` should convert positive `Integers` to a list of digits. (For `0` or negative inputs, `toDigits` should return the empty list.) `toDigitsRev` should do the same, but with the digits reversed.

Example: `toDigits 1234 == [1,2,3,4]`

Example: `toDigitsRev 1234 == [4,3,2,1]`

Example: `toDigits 0 == []`

Example: `toDigits (-17) == []`

Exercise 2 Once we have the digits in the proper order, we need to double every other one. Define a function

```
doubleEveryOther :: [Integer] -> [Integer]
```

Remember that `doubleEveryOther` should double every other number *beginning from the right*, that is, the second-to-last, fourth-to-last, ... numbers are doubled.

Example: `doubleEveryOther [8,7,6,5] == [16,7,12,5]`

Example: `doubleEveryOther [1,2,3] == [1,4,3]`

Exercise 3 The output of `doubleEveryOther` has a mix of one-digit and two-digit numbers. Define the function

```
sumDigits :: [Integer] -> Integer
```

to calculate the sum of all digits.

Example: `sumDigits [16,7,12,5] = 1 + 6 + 7 + 1 + 2 + 5 = 22`

Exercise 4 Define the function

```
validate :: Integer -> Bool
```

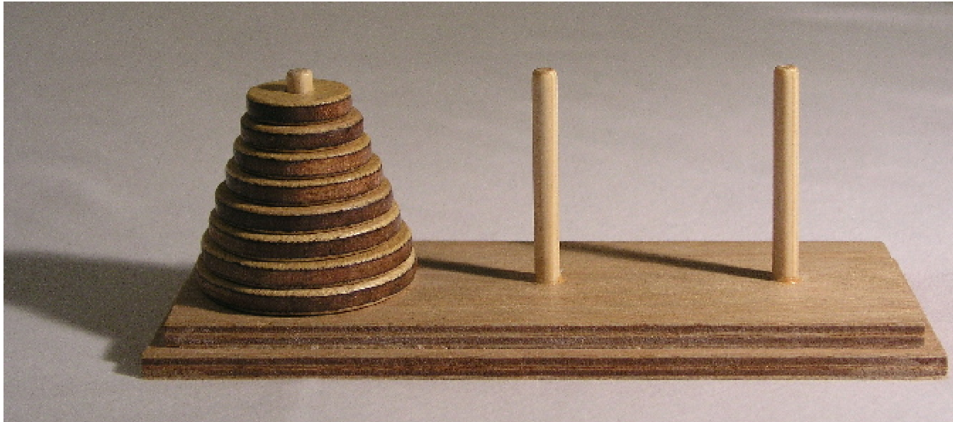
that indicates whether an `Integer` could be a valid credit card number. This will use all functions defined in the previous exercises.

Example: `validate 79927398713 = True`

Example: `validate 79927398714 = False`

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

The Towers of Hanoi[2]



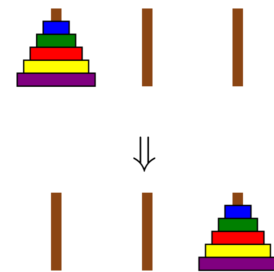


Figure 1: The Towers of Hanoi

Exercise 5 The *Towers of Hanoi* is a classic puzzle with a solution that can be described recursively. Disks of different sizes are stacked on three pegs; the goal is to get from a starting configuration with all disks stacked on the first peg to an ending configuration with all disks stacked on the last peg, as shown in Figure 1.

The only rules are

- you may only move one disk at a time, and
- a larger disk may never be stacked on top of a smaller one.

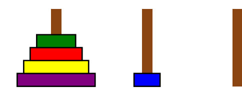


Figure 2: A valid first move.

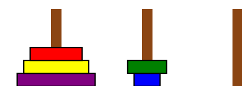


Figure 3: An illegal configuration.

For example, as the first move all you can do is move the topmost, smallest disk onto a different peg, since only one disk may be moved at a time.

From this point, it is *illegal* to move to the configuration shown in Figure 3, because you are not allowed to put the green disk on top of the smaller blue one.

For this exercise, define a function `hanoi` with the following type:

```
type Peg = String
type Move = (Peg, Peg)
hanoi :: Integer -> Peg -> Peg -> Peg -> [Move]
```

Given the number of discs and names for the three pegs, `hanoi` should return a list of moves to be performed to move the stack of discs from the first peg to the last.

Note that a `type` declaration, like `type Peg = String` above, makes a *type synonym*. In this case `Peg` is declared as a synonym

[2] Adapted from an assignment given in UPenn CIS 552, taught by Benjamin Pierce

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.

for `String`, and the two names `Peg` and `String` can now be used interchangeably. Giving more descriptive names to types in this way can be used to give shorter names to complicated types, or (as here) simply to help with documentation.

Example: `hanoi 2 "a" "b" "c" == [("a","b"), ("a","c"), ("b","c")]`

Exercise 6 What if there are four pegs instead of three? That is, the goal is still to move a stack of discs from the first peg to the last peg, without ever placing a larger disc on top of a smaller one, but now there are two extra pegs that can be used as "temporary" storage instead of only one. Write a function similar to `hanoi` which solves this problem in as few moves as possible.

It should be possible to do it in far fewer moves than with three pegs. For example, with three pegs it takes $2^{15} - 1 = 32767$ moves to transfer 15 discs. With four pegs it can be done in 129 moves.

See Exercise 1.17 in Graham, Knuth, and Patashnik, *Concrete Mathematics*, second ed., Addison-Wesley, 1994.

If you are stuck, feel free to search for more information on the Internet; be sure to cite any sources you use.

Exercise 7 Figure 4 shows several circles, cut by chords into one, two, four, and six regions, respectively.

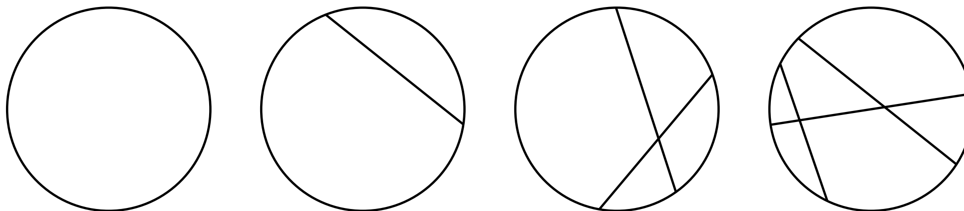


Figure 4: Circles cut by zero, one, two, and three chords.

The pictures with zero, one, and two chords show the maximum possible number of regions (one, two, and four, respectively) which can be created with that many chords. However, using three chords it is possible to create more regions than shown.

In general, what is the maximum number of regions that can be created using n chords? Write a paragraph explaining your answer, along with a Haskell function

```
maxRegions :: Integer -> Integer
```

which computes this number. (Your explanation can be included above the code as a comment.)

© 2016 Brent A. Yorgey. This work is licensed under a Creative Commons Attribution 4.0 International License.