CSCI 365 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.

Be sure to write functions with exactly the specified name and type signature for each exercise (to help in testing your code). You may create additional helper functions with whatever names and type signatures you wish.

Getting started

Exercise 1 What is the largest possible value of type `Int` on the computer you are using? How do you know?

Exercise 2 The following code contains multiple syntax and type errors. Explain each of the errors and how to fix it. (You may want to save the code in a `.hs` file and try loading it in `ghci`.)

```
a, b, c :: Integer
a = 3
b = 99.3
c = b / a
c = 8

ints :: [Integer]
ints = [3,4] : [5,6,"seven"]

bar : Char
bar = 'xy'
```

Look and Say Sequence

The *Look and Say Sequence*[1], introduced by John Conway, begins as follows:

$$1, 11, 21, 1211, 111221, 312211, \ldots$$

[1] https://en.wikipedia.org/wiki/Look-and-say_sequence

where each sequence of digits "describes" the previous sequence; read the Wikipedia article for a full description.

Exercise 3 Write a Haskell function `getRun :: [Integer] -> ([Integer], [Integer])` which splits its input list into two pieces, a run of consecutive equal digits at the beginning, and the rest. For example,

```
getRun [1,1,1,2,3] = ([1,1,1], [2,3])
getRun [3,2,1]     = ([3], [2,1])
getRun [1]         = ([1],[])
```

If you find it helpful, you are welcome to write additional helper function(s).

Exercise 4 Now write a Haskell function `lookAndSay :: [Integer] -> [Integer]` which outputs the next digit sequence when given a digit sequence as input. For example,

```
lookAndSay [1] = [1,1]
lookAndSay [1,1,1,2,2,1] = [3,1,2,2,1,1]
```

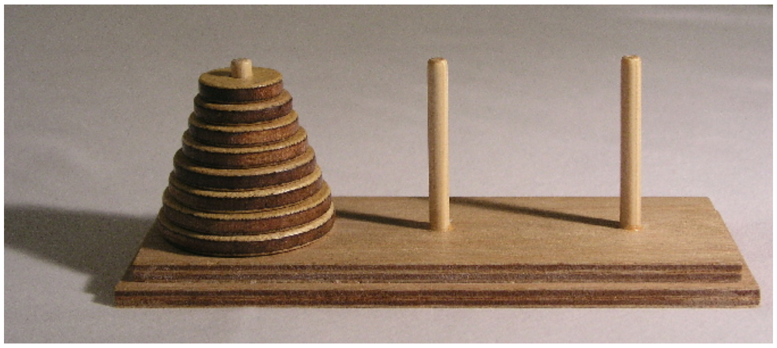
Of course, you should use your `getRun` function from the previous exercise.

Exercise 5 Finally, write a function `lookAndSaySeq :: Integer -> [[Integer]]`, where `lookAndSaySeq n` produces the first `n` terms of the look and say sequence. For example,

```
lookAndSaySeq 4 = [[1], [1,1], [2,1], [1,2,1,1]]
```

The Towers of Hanoi

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



Exercise 6 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.

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 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")]`

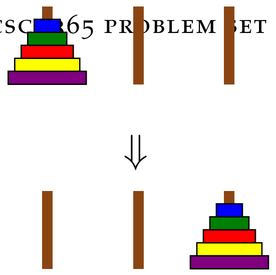


Figure 1: The Towers of Hanoi

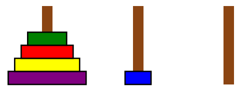


Figure 2: A valid first move.

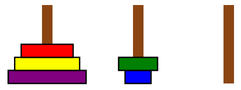


Figure 3: An illegal configuration.

Exercise 7 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.

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

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

Exercise 8 Figure 4 shows several circles, cut by chords into one, two, four, and six regions, respectively.

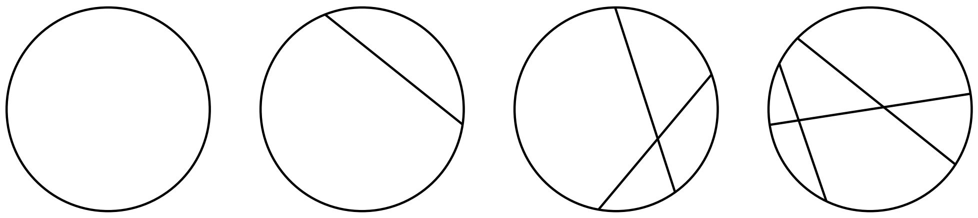


Figure 4: Circles cut by zero, one, two, and three chords.

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.