Software Specification and Testing Exam, October 26, 2016.

```
module ExamOct2016 where

import Data.List
import Test.QuickCheck
```

Problem 1

Which of the following are correct Hoare statements (assume that x ranges over integer numbers)?

(a)
$$\{\lambda x \mapsto x < 0\}\ \lambda x \mapsto x^2\ \{\lambda x \mapsto x > 0\}.$$

(b)
$$\{\lambda x \mapsto x \leq 0\}\ \lambda x \mapsto x^2\ \{\lambda x \mapsto x > 0\}.$$

(c)
$$\{\lambda x \mapsto x < 0\}\ \lambda x \mapsto x^2\ \{\lambda x \mapsto x \geq 0\}.$$

(d)
$$\{\lambda x \mapsto x \leq 0\}\ \lambda x \mapsto x^2\ \{\lambda x \mapsto x \geq 0\}.$$

Problem 2

Let R be the relation $\{(0,1),(1,2),(3,0)\}$ on $\{0,1,2,3\}$.

(a) What is $R \circ R$?

(b) What is the transitive closure of R?

Please state your answers as sets of pairs.

Problem 3

List all relations on $\{0,1,2\}$ that are equivalences, in two ways:

(a) as sets of pairs,

(b) as partitions.

(Recall that an equivalence is a relation that is reflexive, symmetric and transitive, and that every equivalence corresponds to a partition.)

Problem 4

A relation R is anti-symmetric on a domain A if for all $x, y \in A$ it holds that xRy and yRx together imply $x = y$.

Suppose relations are represented as lists of pairs:

```
type Rel a = [(a,a)]
```

Implement a function for checking whether a relation is anti-symmetric The domain is implicit: it consists of all items that are mentioned in the pairs in the relation. The type specification imposes the constraint that `a` is an equality type:

```
isAntiSymm :: Eq a => Rel a -> Bool
isAntiSymm r = error "not yet implemented"
```

How can you test whether your implementation is correct?

Problem 5

A *Huffman tree* is a binary tree with characters at the leaf nodes, and weight information (given by an integer) at every node.

```
data HTree  = Leaf Char Int
            | Fork HTree HTree Int
            deriving (Show)
```

The weight of a tree is given by:

```
weight :: HTree -> Int
weight (Leaf _ w)    = w
weight (Fork _ _ w)  = w
```

Call the following property the *Huffman property*:

```
prop_huffman :: HTree -> Bool
prop_huffman (Leaf _ _) = True
prop_huffman (Fork t1 t2 w) = prop_huffman t1 && prop_huffman t2
                    && weight t1 + weight t2 == w
```

Character frequency tables are lists of pairs `(Char,Int)` where the second element gives the frequency of the first character in some text.

Huffman trees are built from character frequency tables, as follows:

```
createTree :: [(Char,Int)] -> HTree
createTree []      = error "empty input list"
createTree [(c,i)] = Leaf c i
createTree ((c,i):t) = merge (Leaf c i) (createTree t)
```

This uses the following merge function:

```
merge :: HTree -> HTree -> HTree
merge t1 t2 = Fork t1 t2 (weight t1 + weight t2)
```

Give an induction proof to show that it holds for any non-empty input list `xs :: [(Char,Int)]` that `createTree xs` has the Huffman property.

Problem 6

Frequency tables are lists of pairs `(Char,Int)` where the second element gives the frequency of the first character in some text.

Frequency tables are built from strings with the following function:

```
freqList :: String -> [(Char,Int)]
freqList s = error "not yet implemented"
```

This should give, e.g.:

```
*ExamOct2016> freqList "Lab Exam"
[(' ',1),('E',1),('L',1),('a',2),('b',1),('m',1),('x',1)]
```

The composition of `freqList` and `createTree` gives us a map from strings to Huffman trees:

```
string2tree :: String -> HTree
string2tree = createTree . freqList
```

For more information about how Hufmann trees are used, see *Huffman coding*.

Your tasks are:

(a) to implement `freqList`,

(b) to test your implementation of `freqList` with QuickCheck for correctness, using some appropriate properties,

(c) to test with QuickCheck whether for any nonempty string `s` it is the case that `string2tree s` has the Huffman property.

Problem 7

Russian roulette is played with a revolver (a six-shooter) and a single bullet. The player inserts the bullet in the cylinder, spins the cylinder, puts the gun against his head, and pulls the trigger.

If you play this for one round, the chance that you die is $\frac{1}{6}$. If you survive a round you can play a next round by spinning the cylinder again and repeating the procedure.

If you play like this for at most 6 rounds, the chance that you die is given by:

$$1/6 + (5/6)(1/6) + (5/6)^2(1/6) + (5/6)^3(1/6) + (5/6)^4(1/6) + (5/6)^5(1/6).$$

If the example bothers you, you can also think about the probability of throwing at least one six with a fair die, in 6 rounds.

Explanation: the probability of a fatal outcome is given by: probability of fatal outcome in round 1, plus probability of surviving round 1 and fatal outcome in round 2, plus ..., plus probability of surviving rounds 1 through 5 and fatal outcome in round 6.

In general, if success (but in the example: fatal outcome) in one round happens with probability p, then failure in one round happens with probability $q = 1 - p$, and success in at most k rounds happens with probability:

$$p + qp + \cdots + q^{k-1}p.$$

You are asked to investigate the following three implementations of *probability of success in at most k rounds*, when a single round has probability p of success:

```
probSuccess1 :: Int -> Rational -> Rational
probSuccess1 k p = let q = 1 - p in
  sum [ p * q^m | m <- [1..k-1] ]
```

Here is a different implementation:

```
probSuccess2 :: Int -> Rational -> Rational
probSuccess2 k p = 1 - (1 - p)^k
```

And here is yet a different one:

```
probSuccess3 :: Int -> Rational -> Rational
probSuccess3 k p = let q = 1 - p in
  ((q^k - 1) / (q - 1)) * p
```

Which of these implementations is correct? Or are they all flawed? Or maybe they are all correct? State your answer and describe how you found out.
