Gaussian Integers

Recall the type of Gaussian integers from lecture 8:

```haskell
data GaussianInteger : Type where
  Gauss : Integer -> Integer -> GaussianInteger
```

We saw in lecture how to define a `Num` instance for this type so that we could add and multiply them. There is another numeric interface extending `Num` called `Neg`, with a single method called `negate` representing the unary minus \( x \mapsto -x \). Subtraction is implemented using this interface by defining \( x - y \) as \( x + \text{(negate } y) \).

**Task 1**
Write a `Neg` instance for the type of Gaussian integers:

```haskell
implementation Neg GaussianInteger where
```

With this you should be able to do things like:

```haskell
> -(Gauss 1 2)
Gauss -1 -2
> (Gauss 1 2) + - (Gauss 3 4)
Gauss -2 -2
```

*note:* I think that you should be able to use the binary infix subtraction operator (-) too, but it doesn’t seem to work for me.

**Task 2**
Write an `Eq` instance for Gaussian integers:

```haskell
implementation Eq GaussianInteger where
```

**Task 3**
Write a named `Ord` instance for Gaussian integers:

```haskell
implementation [lex] Ord GaussianInteger where
```

which compares them lexicographically:

```haskell
> compare @{lex} (Gauss 1 200) (Gauss 2 1) LT
> compare @{lex} (Gauss 2 1) (Gauss 2 1) EQ
> compare @{lex} (Gauss 3 1) (Gauss 2 4) GT
```
**Task 4**

Use the `Mag` instance for Gaussian integers defined in lecture to write a named `Ord` instance for Gaussian integers:

```haskell
implementation [mag] Ord GaussianInteger where

which compares them by magnitude:

\[
\begin{align*}
> \text{compare @ \{mag\} (Gauss 1 200) (Gauss 2 1)} & \quad \text{GT} \\
> \text{compare @ \{mag\} (Gauss 2 1) (Gauss 2 1)} & \quad \text{EQ} \\
> \text{compare @ \{mag\} (Gauss 3 1) (Gauss 2 4)} & \quad \text{LT}
\end{align*}
\]

**Comparing Lists**

The default `Eq` instance for `Lists` compares them *pointwise*, that is, two lists are considered equal if they have the same elements in the same order:

\[
\begin{align*}
> \text{the (List Nat) [1,2,3] == [3,2,1]} & \quad \text{False} \\
> \text{the (List Nat) [1,2,3] == [1,2,3,3]} & \quad \text{False} \\
> \text{the (List Nat) [1,2,3] == [1,2,3]} & \quad \text{True}
\end{align*}
\]

For the following tasks you will need to import `Data.List`.

**Task 5**

Write a named `Eq` instance for lists that compares them *setwise*:

```haskell
implementation [setwise] Eq a => Eq (List a) where

that is, two lists should be considered equal if each element that occurs (at least once) in one of the lists also occurs (at least once) in the other:

\[
\begin{align*}
> (==) @ \{setwise\} [1,2,3] [3,2,1] & \quad \text{True} \\
> (==) @ \{setwise\} [1,2,3] [1,2,3,3] & \quad \text{True} \\
> (==) @ \{setwise\} [1,2,3] [1,2,4] & \quad \text{False}
\end{align*}
\]

*hint:* the following functions may be useful:

- `elem : Eq a => a -> List a -> Bool`
- `all : (a -> Bool) -> List a -> Bool`

**Task 6**

Write a named `Eq` instance for lists that compares them *multisetwise*:

```haskell
implementation [multisetwise] Eq a => Eq (List a) where

that is, two lists should be considered equal if each list contains the same number of copies of each element as the other, regardless of order:
> (==) @{multisetwise} [1,2,3] [3,2,1]
True
> (==) @{multisetwise} [1,2,3] [1,2,3,3]
False
> (==) @{multisetwise} [1,2,3] [1,2,4]
False

hint: the following functions may be useful:

- elem : Eq a => a -> List a -> Bool
- delete : Eq a => a -> List a -> List a