1. Define a function `secs :: Time -> Int`
   ```haskell```
   data Time = Seconds Int | Minutes Int
   ```haskell```

   ```haskell```
   secs :: Time -> Int
   secs (Minutes m) = 60 * m
   secs (Seconds s) = s
   ```haskell```

2. Define an `Eq` instance for `Time` using `secs`
   ```haskell```
   instance Eq Time where
     Seconds x == Seconds y = x == y
     Minutes x == Minutes y = x == y
     Seconds x == Minutes y = x == y * 60
     Minutes x == Seconds y = x * 60 == y
   ```haskell```

   ```haskell```
   instance Eq Time where
     secs (Minutes m) == secs (Minutes m') = secs m == secs m'
     secs (Seconds s) == secs (Seconds s') = secs s == secs s'
     secs (Minutes m) == secs (Seconds s) = secs (Minutes m) * 60 == secs (Seconds s)
     secs (Seconds s) == secs (Minutes m) = secs (Seconds s) * 60 == secs (Minutes m)
   ```haskell```
3. Discuss different approaches to defining an `Eq` instance for `IntSet`.

4. Define a function `member :: Int -> IntSet -> Bool`

5. Define a function `subset :: IntSet -> IntSet -> Bool`

```haskell
member :: Int -> IntSet -> Bool
member i (Ins j s) = i == j || member i s
member _ ESet      = False

subset :: IntSet -> IntSet -> Bool
subset ESet      _ = True
subset (Ins i s) t = member i s && subset s t
```
Exercises

6. Define an `Eq` instance for `IntSet` using `subset`

```haskell
instance Eq IntSet where
    s == s' = subset s s' && subset s' s
```
7. Define an `Ord` instance for `Time`.

8. Define an `Ord` instance for `IntSet`.

instance `Ord` `Time` where
   \[ t < t' = \text{secs} t < \text{secs} t' \]

instance `Ord` `IntSet` where
   \[ s < s' = \text{subset} s s' \land \neg (s == s') \]
Exercises

9. Define a `Show` instance for `Time` that produces output like `0:13` or `10:06`. (Just minutes and seconds)

class Show a where
    show :: a -> String

instance Show Time where
    show t = show min ++ "":"" ++ show0 sec
    where min = s `div` 60
            sec = s `mod` 60
            s = secs t

    show0 s = (if s <= 9 then "0" else ")" ++ show s
10. Define an `Eq` instance for `List a`.

```haskell
instance Eq a => Eq (List a) where
  Nil       == Nil       = True
  Cons x xs == Cons y ys = x==y && xs==ys
_         == _         =   False
```

```haskell
data List a = Nil | Cons a (List a)
```

```haskell
instance Show a => Show (Maybe a) where
  show (Just a) = show a
  show Nothing = "?"
```

((==) function defined on type `a`

((==) function defined on type `List a`
11. Define a \texttt{Show} instance for \texttt{List} \texttt{a} that produces output such as [], [3], [3,4].

\begin{verbatim}
data List a = Nil | Cons a (List a)

instance Show a => Show (List a) where
    show xs = "["++showElems xs++"]"

showElems :: Show a => List a -> String
showElems Nil          = ""
showElems (Cons x Nil) = show x
showElems (Cons x xs)  = show x++","++showElems xs
\end{verbatim}
Multiple Class Constraints

12. What is the type of \texttt{isqrt}?

\texttt{isqrt} \texttt{x} = \texttt{head} \{ y \mid y \leftarrow [x, x-1 \ldots 1], y^2 \leq x \}

\texttt{isqrt} :: (\texttt{Num} \ a, \texttt{Enum} \ a, \texttt{Ord} \ a) \Rightarrow a \to a

\texttt{isqrt} \texttt{x} = \texttt{head} \{ y \mid y \leftarrow [x, x-1 \ldots 1], y^2 \leq x \}
13. Define a type class `BiFunctor` for binary type constructors that has a member function `bmap`.

```
class BiFunctor f where
    bmap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

14. Define the pair type constructor `(,)` as an instance of `BiFunctor`.

```
instance BiFunctor (,) where
    bmap f g (x,y) = (f x,g y)
```
More on BiFunctors

**class** BiFunctor f **where**

\[ \text{bmap} :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow f \ a \ b \rightarrow f \ c \ d \]

15. Define three functions `mapL`, `mapR`, and `mapLR` to map a function along the left, right, and both arguments of a type given by a binary type constructor.

\[ \text{mapL} :: \text{BiFunctor} \ f \Rightarrow (a \rightarrow c) \rightarrow f \ a \ b \rightarrow f \ c \ b \]
\[ \text{mapL} \ f = \text{bmap} \ f \ \text{id} \]

\[ \text{mapR} :: \text{BiFunctor} \ f \Rightarrow (b \rightarrow c) \rightarrow f \ a \ b \rightarrow f \ a \ c \]
\[ \text{mapR} \ f = \text{bmap} \ \text{id} \ f \]

\[ \text{mapLR} :: \text{BiFunctor} \ f \Rightarrow (a \rightarrow b) \rightarrow f \ a \ a \rightarrow f \ b \ b \]
\[ \text{mapLR} \ f = \text{bmap} \ f \ f \]
Using BiFunctors

16. Define **Graph** as an instance of **BiFunctor**.

```haskell
instance BiFunctor Graph where
  bmap f g (G ns es) = G (map f ns) (map \(v,w,l\)->(v,w,g l)) es
```

17. What meaning do the functions **mapL** and **mapR** have for elements of type **Graph a b**?

```haskell
mapNodeLabel :: (a -> c) -> Graph a b -> Graph c b
mapNodeLabel = mapL

mapEdgeLabel :: (b -> c) -> Graph a b -> Graph a c
mapEdgeLabel = mapR
```