CS3200: Programming Languages
Homework 9: more on Haskell

Required Problems

1. Data compression is very useful because it helps reduce resources usage, such as data storage space or transmission capacity. For this task, you can assume that the input strings consist only of letters of the English alphabet.

   (a) Run-length encoding (RLE) is a very simple form of data compression in which runs of data are stored as a single data value and a count, rather than as the original run. Define a function rle that applied RLE to a the given string.

   ```
    rle ::  String -> String
   rle "aaabbbbbc"  ⟹  "3a 5b 1c"
   rle "banana"  ⟹  "1b 1a 1n 1a 1n 1a"
   ```

   (b) Define rleInverse that applies the inverse RLE operation (RLE decoding) on a given string.

2. For this problem, you will write a module that holds sets over a type a. Our goal is to represent the set as a sorted list with NO repeated elements. Therefore, the type a will always be of the classes:

   - Eq: so that == and /= are defined for elements of type a
   - Ord: so that we can compare using <, etc.
   - Enum: So that we can make lists of the form $[x..y]$ where x and y are elements of type a.
   - Bounded: so that minBound::a and maxBound::a are the smallest and largest elements of a.

   (Note that this means we can form [minBound..maxBound]::[a], a list of all the elements of a.)

   So our declaration of the Set type is:

   ```
   data Set a = Set [a]
                deriving (Show, Eq, Ord)
   ```

   I also have 2 functions that let you go back and forth between sets and lists, primarily for testing purposes. You need to import Data.List for these to work, so I'm giving the syntax for that below:

```
import qualified Data.List as L

list2set :: Ord a => [a] -> Set a
list2set = Set . L.nub . L.sort

set2list :: Set a -> [a]
set2list (Set xs) = xs
```

The temptation in implementing the set operations below is the overrely on list2set which results in code that is simple, clear, and slow!! For example, for the union operation we could define:

```
unionS_slow :: (Ord a) => Set a -> Set a -> Set a
unionS_slow (Set xs) (Set ys) = list2set (xs ++ ys)
```

The problem is that this will result in worst case $O(n^2)$ running time (where n is the max of the length of the 2 sets) and this is much too slow. To speed things up, we need to take advantage of the fact that the lists are sorted and have no repeat elements. So a much better implementation of union is the following, which has a $O(n)$ running time:

```
unionS :: (Ord a) => Set a -> Set a -> Set a
unionS (Set xs) (Set ys) = Set $ merge xs ys
    where
     merge [] ys = ys
     merge xs [] = xs
     merge (x:xs) (y:ys)
        | x<y = x:merge xs (y:ys)
        | x>y = y:merge (x:xs) ys
        | otherwise = x:merge xs ys
```

Note that on any of these problems, I will be looking for (at worst) an $O(n)$ running time, so be careful about using list2set! In particular, you don't want to use those for intersectS or diffS.

(a) Write two functions:

```
singS :: a -> Set a

emptyS :: Set a
```

which (respectively) create a single element set of the input and an empty set.

(b) Write the function:

```
addToS :: (Ord a) => a -> Set a -> Set a
```

so that the first input will be added to the set in the appropriate location.

(c) Write the function:

```
intersectS :: (Ord a) => Set a -> Set a -> Set a
```

so that `intersectS s1 s2` returns a set representing the intersection of s1 and s2.

(d) Write the function:

```
diffS :: (Ord a) => Set a -> Set a -> Set a
```

So that `diffS s1 s2` returns a set representing the set-difference of s1 and s2, which is precisely the elements contained in s1 that are not in s2.

(e) Write the function:

```
subseteq :: (Ord a) => Set a -> Set a -> Bool
```

So that `subseteq s1 s2` returns true whenever s1 is a subset of s2.

(f) Now, put all these in a module named sets, and test your functions. I would like you to submit either a haskell script or a set of instructions you run at the command prompt after loading your module that indicate success of each of your functions.

3. Extra credit: Define a function `subsequence` that takes two lists and returns the ascending list of indices at which the first list occurs as a subsequence of the second list. If there are multiple solutions, return the one with smallest sum of all indices.

```
    subsequence ::  Eq a => [a] -> [a] -> [Int]
subsequence "abcde" "abcbcdef" ⟹ [0, 1, 2, 5, 6]
subsequence [9, 9, 7] [9, 7, 7, 9, 9, 7] ⟹ [0, 3, 5]
subsequence "abc" "caccdcbdca" ⟹ [1, 6, 8]
subsequence "312" "1212313" ⟹ error "subsequence does not exist"
```