Programming I: Functional Programming in Haskell
Solutions to Unassessed Exercises

Set 3: Lists

1 Basics

1. (a) Error: `"H"` should be `'H'`
   (b) `"ongoing"`
   (c) `"Lugworm"`
   (d) `""`
   (e) `1`
   (f) Error: `(:)` used on two strings
   (g) `"gasket"`
   (h) `[('1',1),('2',2)]`
   (i) Error: argument is not a list
   (j) Error: list contains objects of different types
   (k) `2`
   (l) `False`
   (m) `[(5,(True,False,True))]`
   (n) `("bad","dog")`
   (o) `True`
   (p) `9`
   (q) `('a', 2)`
   (r) Error: `zip` call should be in brackets
   (s) `["not","with","standing"]`

2. Note that the second rule is missing a case, but the top-to-bottom matching rule means that we fall through to the third rule if `c > c'`.

```
precedes :: String -> String -> Bool
precedes [] s'
  = True
precedes (c : cs) (c' : cs')
  | c < c'  = True
  | c == c' = precedes cs cs'
precedes s s'
  = False
```

3. ```
pos :: Char -> String -> Int
pos c (c' : cs)
  | c == c'   = 0
  | otherwise = 1 + pos c cs
```

4. ```
twoSame :: [Int] -> Bool
twoSame []       = False
twoSame (x : xs) = elem x xs || twoSame xs
```

   The complexity is $O(n^2)$ since `twoSame` will be called $O(n)$ times on average and each call to elem is $O(n)$ on average.

5. ```
replace :: Int -> a -> [a] -> [a]
replace 0 p (c : cs)
  = p : cs
replace _ p []
  = []
replace n p (c : cs)
  = c : replace (n - 1) p cs
```

6. ```
rev :: [a] -> [a]
rev []
  = []
rev (c : cs)
  = rev cs ++ [c]
```

   The cost is $O(n^2)$ where n is the length of the list being revd.

   ```
rev xs
  = rev' xs []
  where
    rev' [] a       = a
    rev' (x : xs) a = rev' xs (x : a)
```

   The cost now is $O(n)$ where n is the length of the list being reversed.

7. We could implement `isPrefix` using `take` and `==`:

   ```
isSubstring' s s'
  = isSubstring' s'
  where
    n = length s
    isSubstring' ""
      = False
    isSubstring' s'@(c : cs)
      = take n s' == s || isSubstring' cs
```

   Note that we compute `n` *once* by using a helper function. This is a common trick, so you should practice it. In this solution both `take` and `==` pass over the string `s'`. We can do this in a single pass, although it's not clear that it will be any more efficient in practice:

   ```
isSubstring s []
  = False
isSubstring s s'@(c : cs)
  = isPrefix s s' || isSubstring s cs
  where
```

```
isPrefix [] cs
  = True
isPrefix (c : cs) (c' : cs')
  | c == c' = isPrefix cs cs'
isPrefix s s'
  = False
```

Note that the second rule has a single guard; if this fails we drop down to the third (default) rule.

8.
```
rearrange :: String -> String -> String -> String
rearrange s s' []
  = []
rearrange s s' (c : cs)
  = s !! pos c s' : rearrange s s' cs
```

9.
```
transpose :: [[a]] -> [[a]]
transpose ([] : rs)
  = []
transpose a
  = heads a : transpose (tails a)
  where
    heads [] = []
    heads (r : rs) = head r : heads rs
    tails [] = []
    tails (r : rs) = tail r : tails rs
```

10. `diags` is quite messy. When you've learnt about higher-order functions, try using `map` (twice!).

```
rows :: ([a], Int) -> [[a]]
rows (xs , n)
  = rows' xs
  where
    rows' []
      = []
    rows' xs
      = r : rows' rs
      where
        -- Can also use take n and drop n...
        (r, rs) = splitAt n xs

cols :: ([a], Int) -> [[a]]
cols xs
  = transpose (rows xs)

diags :: ([a], Int) -> [[a]]
diags (xs, n)
  = [traverseElements [k * (n + 1) | k <- [0 .. n - 1]],
     traverseElements [k * (n - 1) | k <- [1 .. n]]]
  where
    traverseElements []
      = []
    traverseElements (i : is)
      = xs !! i : traverseElements is
```

11. ```
removeWhitespace :: String -> String
removeWhitespace ""
  = ""
removeWhitespace s@(c : cs)
  | isSpace c = removeWhitespace cs
  | otherwise = s
```
    (A better way to do this is to use higher-order function like `dropWhile` – see the notes.)

12. ```
nextWord :: String -> (String, String)
--Pre: The first character is non-whitespace
nextWord ""
  = ("", "")
nextWord (c : cs)
  | isSpace c = ("", cs)
  | otherwise = (c : w, s)
  where
    (w, s) = nextWord cs
```

13. ```
splitUp :: String -> [String]
splitUp "" = []
splitUp s
  = w : splitUp ws
  where
    (w, ws) = nextWord (removeWhitespace s)
```

14. ```
primeFactors :: Integer -> [Integer]
-- Pre: n > 0
primeFactors n
  = factors 2 n
  where
    factors p 1
      = []
    factors p m
      | m `mod` p == 0 = p : factors p (m `div` p)
      | otherwise      = factors (p + 1) m
```

15. ```
hcf :: Int -> Int -> Int
hcf a b
  = product (fs \\ (fs \\ fs'))
  where
    fs = primeFactors a
    fs' = primeFactors b
```

16. ```
lcm :: Int -> Int -> Int
lcm a b
  = product (fs' \\ fs) * min a b
  where
    fs  = primeFactors (min a b)
    fs' = primeFactors (max a b)
```

2 List Comprehensions

1. You get `[(1,2),(1,3),(4,3)]` instead of `[(1,2),(1,3)]`. The problem is that the `x` in `(x, y)` is a new variable, so the comprehension picks out all elements of the table unconditionally. Here's a fix:

```
findAll x t = [y | (x', y) <- t, x' == x]
```

2. ```
   isSubstring :: String -> String -> Bool
   isSubstring xs ys
     = or [take (length xs) t == xs | t <- tails ys]
   ```

   Note that GHC ensures that `length xs` is computed only once, so there is no need to name it in a `where` clause.

3. ```
   remove :: Eq a => a -> [(a, b)] -> [(a, b)]
   remove x table
     = [p | p@(x', _) <- table, x /= x']
   ```

   Using a filter, the predicate you need is `((/=x).fst)` where `x` is the item you are removing.

4. ```
   qsort :: [Int] -> [Int]
   qsort []
     = []
   qsort (x : xs)
     = qsort [y | y <- xs, y <= x] ++ [x] ++
       qsort [y | y <- xs, y > x]
   ```

5. ```
   allSplits :: [a] -> [([a], [a])]
   allSplits xs
     = [splitAt n xs | n <- [1..length xs - 1]]
   ```

6. ```
   prefixes :: [t] -> [[t]]
   prefixes []
     = []
   prefixes (c : cs)
     = [c] : [c : ps | ps <- prefixes cs]
   ```

7. There are lots of ways to do this, e.g.

   ```
   substrings :: String -> [String]
   substrings []
     = []
   substrings s@(c : cs)
     = substrings' s ++ substrings cs
     where
       substrings' []       = []
       substrings' (c : cs) = [c] : [c : s | s <- substrings' cs]
   ```

8. ```
   substrings :: String -> [String]
   substrings s
     = [i | t <- tails s, i <- tail (inits t)]
   ```

9. ```
   perms :: [Int] -> [[Int]]
   perms []
     = [[]]
   perms xs
     = [x : ps | x <- xs, ps <- perms (xs \\ [x])]
   ```

10. ```
    routes :: Int -> Int -> [(Int, Int)] -> [[Int]]
    routes m n g
      | m == n    = [[m]]
      | otherwise = [m : r | m'' <- [n' | (m', n') <- g, m' == m],
                             r <- routes m'' n g]
    ```

    This version does cycle detection:

```
routes m n g
  = routes' m []
  where
    routes' m seen
      | elem m seen = []
      | m == n      = [[m]]
      | otherwise   = [m : r | m'' <- [n' | (m', n') <- g, m' == m],
                               r <- routes' m'' (m : seen)]
```

`seen` is the list of nodes that have been visited so far.