Homework 3: More Haskell

1 Submission

- For the first part of this homework, create a file RLE.hs, which will hold your module for doing data compression. To get started, begin your file as follows:

  ```
  module RLE where
  ```

- For the second part of this homework, create a file Vector.hs, which will hold your module for vector operations. To get started, begin your file as follows:

  ```
  module Vector where
  ```

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
   Main> rle "aaabbbbbc" = "3a 5b 1c"
   Main> rle "banana" = "1b 1a 1n 1a 1n 1a"
   ```

   Note: You can assume all repeated patterns will be less than nine characters in length.

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

   ```
   rleInverse :: String -> String
   Main> rleInverse "3a 5b 1c" = "aaabbbbbc"
   Main> rleInverse "1b 1a 1n 1a 1n 1a" = "banana"
   ```

2. Consider the following data declaration of a vector (as in a math vector in Euclidean space) in Haskell:

```
type Vector = [Double]
```

Note that this definition allows vectors of any length- e.g. 2D vectors, 3D vectors, and arbitrary N dimensional vectors.

For the second part of this homework, create a file Vector.hs, which will hold a module for Vectors. You will be writing several functions to do mathematical calculations over a vector:

(a) The function `scale :: Double -> Vector -> Vector` takes a Double y, and a vector v, and returns a new Vector that is just like v, except that each coordinate is y times the corresponding coordinate in v.

```
Main> scale 3 [5, 10] = [15, 30]
Main> scale 10 [1, 2, 3] = [10, 20, 30]
```

(b) The function `add :: Vector -> Vector -> Vector` takes two vectors and adds them together, so that each coordinate of the result is the sum of the corresponding coordinates of the argument Vectors. Your code should assume that the two vector arguments have the same length.

```
Main> add [1, 2] [3, 4] = [4, 6]
Main> add [1, 2, 3] [11, 22, 33] = [12, 24, 36]
```

(c) The function `dot :: Vector -> Vector -> Double` takes two vectors and computes their dot product (or inner product), which is the sum of the products of the corresponding elements. Your code should assume that the two vector arguments have the same length.

$$v \cdot u = \sum_i v_i u_i \tag{1}$$

```
Main> dot [1, 2] [3, 4] = 11.0
Main> dot [3, 1, 4] [1, 5, 9] = 44.0
```

(d) The function `norm :: Int -> Vector -> Double` which takes an integer p and a Vector $v = \{v_1, ..., v_n\}$ and computes the p^{th} norm of the vector, defined as:

$$|v| = \left(\sum_i v_i^p \right)^{\frac{1}{p}} \tag{2}$$

You can assume p is a positive integer.

```
Main> norm 2 [4] = 4.0
Main> norm 1 [3,4] = 7.0
Main> norm 2 [3,4] = 5.0
Main> norm 2 [5, 10, 20] = 22.913
Main> norm 3 [5, 10, 20] = 20.897
```

Extra Credit Problems

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"
```