Recursive Functions

Introduction

Many functions can naturally be defined in terms of other functions.

```
factorial  :: Int → Int
factorial n = product [1..n]
```

factorial maps any integer n to the product of the integers between 1 and n.

Expressions are <u>evaluated</u> by a stepwise process of applying functions to their arguments.

For example

```
  factorial 3
=
  product [1..3]
=
  product [1,2,3]
=
  1*2*3
=
  6
```

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called <u>recursive</u>.

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

factorial maps 0 to 1, and any other integer to the product of itself with the factorial of its predecessor.

For example:

```
    factorial 3
=
    3 * factorial 2
=
    3 * (2 * factorial 1)
=
    3 * (2 * (1 * factorial 0))
=
    3 * (2 * (1 * 1))
=
    3 * (2 * 1)
=
    3 * 2
=
    6
```

Why is Recursion Useful?

- Some functions, such as factorial, are simpler to define in terms of other functions;

- In practice, however, most functions can naturally be defined in terms of themselves;

- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

Recursion is not restricted to numbers, but can also be used to define functions on <u>lists</u>.

```
product :: [Int] → Int
product [] = 1
product (x:xs) = x * product xs
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

For example:

```
   product [1,2,3]
=
   product (1:(2:(3:[])))
=
   1 * product (2:(3:[]))
=
   1 * (2 * product (3:[]))
=
   1 * (2 * (3 * product []))
=
   1 * (2 * (3 * 1))
=
   6
```

Quicksort

The quicksort algorithm for sorting a list of integers can be specified by the following two rules:

- The empty list is already sorted;
- Non-empty lists can be sorted by sorting the tail values ≤ the head, sorting the tail values > the head, and then appending the resulting lists on either side of the head value.

Using recursion, this specification can be translated directly into an implementation:

```
qsort :: [Int] → [Int]
qsort []     = []
qsort (x:xs) = qsort [a | a ← xs, a ≤ x]
                     ++ [x] ++
               qsort [b | b ← xs, b > x]
```

Note:

- This is probably the simplest implementation of quicksort in any programming language!

For example (abbreviating qsort as q):

```
q [3,2,4,1,5]

q [2,1] ++ [3] ++ q [4,5]

q [1] ++ [2] ++ q []      q [] ++ [4] ++ q [5]

[1]             []        []               [5]
```

Exercise

Define a recursive function

```
insert :: Int → [Int] → [Int]
```

that inserts an integer into the correct position in a sorted list of integers. For example:

```
> insert 3 [1,2,4,5]

     [1,2,3,4,5]
```