Question 1: Some Numbers are Perfect

A perfect number is a positive integer greater than 1 that equals the sum of its divisors, including 1 but not itself. For example, 6 is a perfect number because its divisors are 1, 2, and 3 and \(1 + 2 + 3 = 6\).

(a) Write a function `isPerfect :: Int -> Bool` that determines for a given integer whether it is perfect or not.

\[
isPerfect \ n = (n == \text{sum} \ [v \mid v \leftarrow [1..n-1], \ \text{mod} \ n \ v == 0])
\]

(b) Write a function `listPerfectNumbers :: Int -> [Int]` so that `listPerfectNumbers \ n` returns a list of all perfect numbers less than or equal to \(n\).

\[
\text{listPerfectNumbers} \ n = \text{filter} \ \text{isPerfect} \ [1..n]
\]

(c) Write a function `initialPerfectNumbers :: Int -> [Int]` so that `initialPerfectNumbers \ n` returns a list of the \(n\) smallest perfect numbers in ascending order. You get a bonus point if you use an infinite list (such as \([1..]\)) and Frege’s laziness to solve this problem.

\[
\text{initialPerfectNumbers} \ n = \text{take} \ n \ \$ \ \text{filter} \ \text{isPerfect} \ [1..]
\]

Question 2: Prime Time

Write a function `primeFactors :: Int -> [Int]` that for a given integer greater than 1 returns the list of its prime factors in ascending order. For example, `primeFactors 300` should output `[2, 2, 3, 5, 5]`. 
primeFactors 1 = []
primeFactors n = f:(primeFactors $ div n f)
    where f = head $ filter (\x -> mod n x == 0) [2..n]

**Question 3: Lists of Lists**

As you know, list elements can be of any type, and so they can also be lists themselves. For example, \[[3, 4, 2], [1, 9, 9], [5, 8]\] is such a list of lists, and its type is \[[\text{Int}]\]. Your task is to write a function `flatten :: [[a]] -> [a]` that turns a list of lists of type `a` into a list of type `a` while preserving the order of elements. For the example above, the output should be \[3, 4, 2, 1, 9, 9, 5, 8\].

Now there is one caveat: You need to use the fold functions (foldl or foldr) to accomplish this. If possible, use the point-free style.

```hs
flatten xs = foldl (++) [] xs

Point-free version:

flatten = foldl (++) []
```

**Question 4: Growing Trees**

In Frege, it is easy and convenient to define and operate on recursive data structures. For example, we can define a data type `Tree` that implements a binary tree with an integer label at each node:

```hs
data Tree = Empty | Node Int Tree Tree
```

That is all we need to define the data structure. As before, we can use:

```hs
derive Show Tree
```

to make Tree values printable. We can now write:

```hs
awesomeTree = Node 5 (Node 4 (Node 6 Empty Empty) Empty) (Node 7 (Node 9 Empty Empty) (Node 3 Empty Empty))
```

to let `awesomeTree` represent the following tree:
We can now define recursive functions operating on such trees, for example, a function
\[ \text{treeSum :: Tree -> Int} \]
that computes the sum of all node labels in a given tree:

\[
\begin{align*}
\text{treeSum \ Empty} &= 0 \\
\text{treeSum \ (Node \ label \ left \ right)} &= \text{label} + \text{treeSum \ left} + \text{treeSum \ right}
\end{align*}
\]

For our awesomeTree, the result is 34.

(a) Write a function \[ \text{treeMax :: Tree -> Int} \]
that returns the maximum label value in the given tree.

\[
\text{treeMax \ t} = \text{maximum} \ \$ \ \text{nodeList \ (Node \ label \ left \ right)}
\]

(b) Write a function \[ \text{nodeList :: Tree -> [Int]} \]
that returns a list of all label values in the tree in no particular order.

\[
\begin{align*}
\text{nodeList \ Empty} &= [] \\
\text{nodeList \ (Node \ label \ left \ right)} &= \text{[label]} ++ \text{nodeList \ left} ++ \text{nodeList \ right}
\end{align*}
\]

(c) Write a function \[ \text{nodeDepth :: Tree -> [(Int, Int)]} \]
that returns a list of pairs, with one pair for each node, in no particular order. The first element in each pair is the label of the node, and the second one is its depth. Remember from kindergarten that the depth of the root is zero, and the depth of each other node is the number of edges that you have to travel to reach it, starting from the root. \textbf{Hint:} You need a function with an additional input so that you can remember how deep you are in the tree during the recursive function calls. Probably the best way to do this is to have \text{nodeDepth} call a helper function that you attach to it using a where clause.

\[
\begin{align*}
\text{nodeDepth \ t} &= \text{goDepth \ t} \ 0 \\
&\quad \text{where} \ \text{goDepth \ Empty} \ = \ [] \\
&\quad \text{goDepth \ (Node \ label \ left \ right)} \ n = \ [(\text{label}, \ n)] \ ++ \\
&\quad \text{goDepth \ left} \ (n+1) \ ++ \ \text{goDepth \ right} \ (n+1)
\end{align*}
\]
(d) Write a function \( \text{treeHeight} :: \text{Tree} \rightarrow \text{Int} \) that returns the height of the tree, which is the maximum depth among its nodes. This is of course easy once you have solved part (c).

\[
\text{treeHeight } t = \text{maximum depth}
\]

where \((\_, \text{depth}) = \text{unzip} \circ \text{nodeDepth}\) \(t\)

**Question 5 (Bonus Question): Bubbling**

In class we talked about the quicksort algorithm, which is a very efficient way to sort the elements in a list. As you know, there are many other ways. One of the simplest algorithms is called bubblesort. It starts by looking at the first two elements in the list. If the first one is greater than the second, it swaps the two elements, otherwise it leaves them in place. Then it looks at the (new) second and third elements and swaps them if the second is greater than the third one. This way it traverses the entire list. Afterwards, the new last element in the list must already be in the correct position. So the algorithm starts over again but this time stops before looking at the last list element. In the next cycle, it stops before the second last one, and so on, until no more comparisons need to be made.

(a) Write a function \( \text{bubbleSort} :: \text{Ord } a => [a] \rightarrow [a] \) that sorts a list of any type as long as an ordering \((<, >)\) is defined on it, using the bubblesort algorithm.

\[
\begin{align*}
\text{bubbleSort1 } (x:y:xs) & = y \circ [x] \circ \text{bubbleSort1}(x:xs) \\
\text{bubbleSort1 } x & = x
\end{align*}
\]

\[
\text{bubbleSortn } xs\ 0\ =\ xs
\]

\[
\text{bubbleSortn } xs\ len\ =\ \text{bubbleSort } (\text{bubbleSort1 } (\text{take } len\ xs) \circ [\text{drop } len\ xs])\ (len\ -\ 1)
\]

\[
\text{bubbleSort } xs\ =\ \text{bubbleSortn } xs\ (\text{length } xs)
\]

(b) In some cases, a variant of bubblesort can actually be extremely efficient and easily outperform quicksort. Imagine that we have a list that is almost perfectly sorted, i.e., each element is at most \(n\) positions away from its correct (ordered) place. Then the list will be perfectly sorted after \(n\) cycles. In order to take advantage of this, we need an additional termination criterion: If during any cycle no elements are swapped, computation can stop, because then all elements must already be in their correct order. Write a function \( \text{bubbleSmart} :: \text{Ord } a => [a] \rightarrow [a] \) that implements this (sometimes) efficient version of bubblesort.
bubbleSmart1 xs = (newList, or swaps)
  where (newList, swaps) = unzip $ bubbleHelper xs
  bubbleHelper (x:y:xs)
    | x > y     = (y, True):bubbleHelper (x:xs)
    | otherwise = (x, False):bubbleHelper (y:xs)
  bubbleHelper [x] = [(x, False)]
  bubbleHelper [] = []

bubbleSmartn xs 0   = xs
bubbleSmartn xs len = if swapped then bubbleSmartn (newList ++
  drop len xs) (len - 1) else xs
  where (newList, swapped) = bubbleSmart1 (take len xs)

bubbleSmart xs = bubbleSmartn xs (length xs)

**Question 6 (Required only for CS670):**

Write a function \texttt{powerSet :: [a] -\rightarrow [[a]]} that receives a set as its input (as a list of its elements) and outputs its power set, i.e., the set of all its subsets (as a list of lists). For example, \texttt{powerSet [1, 2, 3]} should output \texttt{[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]} (the order does not matter).

\texttt{powerSet [] = [[]]}
\texttt{powerSet (x:xs) = powerSet xs ++ map (x:) (powerSet xs)}