Some of the exercises in this lab involve vectors, so you should import the relevant part of the standard library by writing

import Data.Vect

at the top of your idris file.

1. (a) Write a function \( \text{swapPair} : (a, b) \rightarrow (b, a) \)

(b) Write a function \( \text{swapEither} : (\text{Either} \ a \ b) \rightarrow (\text{Either} \ b \ a) \)

(c) Recall the identity function \( \text{id} : a \rightarrow a \) defined by \( \text{id} \ x = x \). Define functions:

\[ \text{there} : \text{Nat} \rightarrow \text{List Unit} \]

and

\[ \text{back} : \text{List Unit} \rightarrow \text{Nat} \]

such that for every \( x : \text{Nat} \)

\[ \text{back} \ (\text{there} \ x) = \text{id} \ x : \text{Nat} \]

and for every \( y : \text{List Unit} \)

\[ \text{there} \ (\text{back} \ y) = \text{id} \ y : \text{List Unit} \]

(d) Recall that \( \text{Fin} \ n \) is the type of natural numbers less than \( n \). Define a function \( \text{project} : \text{Fin} \ n \rightarrow \text{Nat} \) that returns the number, forgetting the part about it being less than \( n \).

(e) Write a function \( \text{listify} : \text{Vect} \ n \ a \rightarrow \text{List} \ a \). Your function should map the input vector to this list containing the same elements in the same order.

2. The reverse of a list contains the same elements, in reverse order. For example, the reverse of \([1,2,3]\) is \([3,2,1]\).

(a) Write a function \( \text{reverseList} : \text{List} \ a \rightarrow \text{List} \ a \) that reverses its argument.

(b) Write a function \( \text{reverseVect} \) that reverses a vector in a similar way.
(c) How efficient are your reverse functions? Is it possible to make them faster?

3. Consider the following type:

```haskell
data Tree : Type -> Type where
  Leaf : Tree a
  Node : Tree a -> a -> Tree a -> Tree a
```

The idea being that a `Tree a` is a binary tree, as in:

- `Leaf` => *
- `Node t1 x t2` => `x`  
  `/ \
   t1 t2`

- `Node Leaf 3 (Node (Node Leaf 4 Leaf) 5 Leaf)` =>
  
  3  
  `/ \n  * 5
  `/ \n  4 *
  `/ 
  * *
```

(a) Write a function `size : Tree a -> Nat` that returns the number of values stored in the given tree (Leaves don’t count).

(b) Write a function `depth : Tree a -> Nat` that returns the depth of the tree (Leaves have depth 0).

(c) Write a function `flatten : Tree a -> List a` that returns a list containing the elements of a tree. Are there any other ways to write this function? (hint: tree traversal algorithms).