CO202 - Algorithms II — Tutorial Sheets

Tutorial 1

Not sure if I'll actually cover it, since I've done these questions already in the notes.

Tutorial 2

Note that the `List` class is as follows;

```
class List list where
  fromList :: [a] -> list a
  toList :: list a -> [a]
  normalize :: list a -> list a

  empty :: list a
  single :: a -> list a

  cons :: a -> list a -> list a
  snoc :: a -> list a -> list a
  head :: list a -> a
  tail :: list a -> list a
  init :: list a -> list a
  last :: list a -> a

  isEmpty :: list a -> Bool
  isSingle :: list a -> Bool

  length :: list a -> Int
  (++) :: list a -> list a -> list a
```

1. The `List` typeclass overloads the functions `empty`, `cons`, `snoc`, `head`, `tail`, `init`, `last`, `null`, `length`, and `(++)` into the `List` class given above. It is possible to give default implementations for all of these functions. For instance, the definition of `normalize` is

   `normalize = fromList .  toList`

   Give all the other default implementations by appropriate conversion using `toList` and `fromList`;

```
empty = fromList []
single x = fromList [x]

cons x xs = fromList (x:toList xs)
snoc x xs = fromList ((toList xs) ++ [x])
head xs = Prelude.head (toList xs)
tail xs = fromList (Prelude.tail (toList xs))
init xs = fromList (Prelude.init (toList xs))
last xs = Prelude.last (toList xs)

isEmpty xs = null (toList xs)
isSingle xs = case (toList xs) of [_] -> True
                                   _   -> False

length xs = Prelude.length (toList xs)
(++) xs ys = fromList (toList xs ++ toList ys)
```

2. Give the trivial instance of `List` class for ordinary lists by giving the minimal definition of `instance List []`.

```
instance List [] where
  fromList = id
  toList = id
  normalize = id

  empty = []
  single x = [x]

  cons x xs = x:xs
  snoc x xs = xs ++ [x]
  head = Prelude.head
  tail = Prelude.tail
  init = Prelude.init
  last = Prelude.last

  isEmpty = null

  isSingle [_] = True
  isSingle _   = False

  length = Prelude.length
  (++) = Prelude.(++)
```

3. Implement the instance of the `List` class for the `DList` datatype. State the complexity of each of these functions.

```
instance List DList where
  fromList xs = DList (xs ++)
  toList (DList fxs) = fxs []

  DList fxs ++ DList fys = DList (fxs . fys)
```

Generally, the time complexities are the same, except for `tail` (since the whole list must now be rebuilt). The benefit is that `(++)` is now constant time.

4. Prove that the definition of `(++)` for `DList`s is correct by showing;

`fromList xs ++ fromList ys = fromList (xs ++ ys)`

`fromList xs` gives `DList (xs ++)`, and similarly `fromList ys` gives `DList (ys ++)`. By our definition of `(++)`, we know that `fromList xs ++ fromList ys` gives `DList ((xs ++) . (ys ++))`. Intuitively, that is equivalent to `DList ((xs ++ ys) ++)`, which is the result of `fromList (xs ++ ys)`.

5. Explain the time complexity of the following definition of `reverse`;

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

This has a complexity of $O(n^2)$, due to the left nested chain of appends.

$$\text{let } n = \texttt{length xs} \qquad\qquad \text{for } \texttt{reverse xs}$$
$$T_{\texttt{reverse}}(0) = 1$$

$$T_{\texttt{reverse}}(n) = T_{\texttt{reverse}}(n-1) + \underbrace{(n-1)}_{T_{(\texttt{++})}(n-1)}$$

6. Show how to modify the previous definition of `reverse` to produce a version `reverse' :: DList a -> DList a`, and give the time complexity of the resulting function.

```
reverse' :: DList a -> DList a
reverse' xs
  | isEmpty xs = empty
  | otherwise  = reverse' (tail xs) ++ single (head xs)
```

   This has a time complexity in $O(n)$, as `(++)` is right associated.

7. Give a trivial representation of lists where `length` takes $O(1)$, and that does not affect the complexity of other operations.

```
data LList a = LList Int [a]

instance List LList where
  fromList xs = LList (length xs) xs
  toList (LList _ xs) = xs
  cons x (LList n xs) = LList (n + 1) (x:xs)
  length (LList n _) = n
```

   This simply stores the length of the list as a parameter.