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.

Mock Exam

1. This question is about dynamic programming.

(a) The *edit-distance* between two strings is the minimum number of insertions, deletions, and updates of characters required to turn one string into the other. For example. `"change"` can be turned into `"hunger"` in 3 steps with the sequence `["hange", "hunge", "hunger"]` by deleting `'c'`, updating `'a'` to `'u'`, and inserting `'r'`.

i) Write a recursive function called `dist` that calculates the edit distance between two strings. The function should have the following signature;

```
dist :: String -> String -> Int
```

For example, `dist "change" "hunger" = 3`. You may use the function `minimum :: [Int] -> Int` which returns the minimum value in a non-empty list of integers.

Start with the example(s) given, if there is any doubt. Since we're told to do induction on strings, we can do case analysis on lists of characters.

```
1 dist :: String -> String -> Int
2 dist [] [] = 0
3 dist [] ys = length ys
4 dist xs [] = length xs
5 dist xxs@(x:xs) yys@(y:ys)
6   = minimum [1 + dist xs yys,
7             ,1 + dist xxs ys,
8             ,(if (x == y) then 0 else 1) + dist xs ys]
```

ii) What is the time complexity of `dist xs ys`?

Let m = `length xs`, and n = `length ys`. The second and third cases, on lines 3 and 4, have complexities $O(n)$ and $O(m)$. However, the final case has branching, which suggests an exponential complexity. Therefore, the overall complexity is in $O(3^{m+n})$.

(b) The function `tabulate` can be used to build an array.

```
tabulate :: (Enum i, Ix i) => (i, i) -> (i -> a) -> Array i a
tabulate (m, n) f = array (m, n) [(i, f i) | i <- range (m, n)]
```

The resulting array allows fast access to its elements, where given an array `as`, the expression `as ! i` accesses the `i`th element in constant time. Define `dist'`, an efficient version of `dist` that uses dynamic programming. You may assume that the relevant `Enum` and `Ix` instances exist for the `tabulate` function, and that indexing into strings takes constant-time.

Here, we want to replace all recursive calls with lookups.

```
dist' :: String -> String -> Int
dist' xs ys = table ! (m, n)
  where
    m = length xs
    n = length ys

    table = tabulate (m + 1, n + 1) mdist

    mdist 0 0 = 0
    mdist 0 j = j
    mdist i 0 = i
    mdist i j =
      minimum [1 + table ! (i - 1, j)
              ,1 + table ! (i, j - 1)
              ,(if (x == y) then 0 else 1) + table (i - 1, j - 1)]
      where
        x = xs !! (i - 1)
        y = ys !! (j - 1)
```

(c) Define a recursive function `dists`, where `dists xs ys` is a sequence of strings of shortest length needed to turn `xs` into `ys`. This should have the signature;

```
dists :: String -> String -> [String]
```

You do not need to worry about efficiency. You may use standard list functions so long as you give their type and briefly explain what they do.

Note that this question tends to be harder, and worth less. Therefore, it should be attempted at the end, when the rest of the marks are secured. For this, we want to define a function `inits :: [a] -> [[a]]`, for example `inits "hunger" = ["", "h", "hu", ..., "hunger"]`. Similarly, we want to define `tails`, such that `tails "change" = ["change", "hange", "ange", ..., ""]`. Now that we've defined these functions, we can use it as follows;

```
dists :: String -> String -> [String]
dists [] [] = []
dists [] ys = inits ys
dists xs [] = tails xs
dists (x:xs) (y:ys)
  = minimums [xs : dists xs (y:ys)
             ,map (y:) ((x:xs) dists (x:xs) ys)
             ,if (x == y) then (map (x:) dists xs ys) else (map (y:) (xs :
                dists xs ys))]
```

`minimums` can be easily defined, which returns the list of minimum length.

2. This question is about divide & conquer, and randomised algorithms.

(a) The numbers in Fibonacci sequence are given by this recursive function:

```
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

The sequence starts `[0, 1, 1, 2, 3, 5, ...]`.

The n^{th} Fibonacci number can also be computed by the following equation;

$$\texttt{fib n} = \frac{\psi^{\texttt{n}}}{\sqrt{5}} \text{ where } \psi = \frac{1+\sqrt{5}}{2}$$

i) Give an upper bound for the complexity of the recursive `fib` function.

$$O(2^n)$$

ii) By passing the previous two results around as arguments, write a recursive function called `fib'` that calculates `fib n` in linear time.

```
fib' :: Int -> Integer
fib' n = loop n 0 1
  where
    loop 0 x y = x -- base case, when we are "done"
    loop n x y = loop (n - 1) (x + y) x
```

iii) Use the golden ratio (ψ) to define `fib''`, a divide & conquer version of `fib`. You may assume you can calculate the golden ratio accurately in constant time, and that multiplication takes constant time. State the complexity of `fib''`.

```
fib'' :: Int -> Integer
fib'' n = round (psi n / sqrt 5)

psi :: Int -> Double
psi 0 = 1
psi 1 = 1 + (sqrt 5) / 2
psi n
  | even n     = pk * pk
  | otherwise = pk * pk * (psi 1)
  where
    k = div n 2
    pk = psi k
```

Since we're halving at each stage, we can see that `fib''` is in $O(\log_2 n)$.

(b) i) Recall the following functions for working with random numbers:

```
mkStdGen :: Int -> StdGen
random :: Random a => StdGen -> (a, StdGen)
randomR :: Random a => (a, a) -> StdGen -> (a, StdGen)
```

Explain what these functions do.

See notes.

ii) Define a randomised Monte Carlo algorithm `sqrt5` to find an approximation of $\sqrt{5}$ after 100000 trials. It should have the following signature;

```
sqrt5 :: Double
```

By considering the ratio of random numbers drawn uniformly between 0 and 3, we can do this as follows (the first solution is iterative, with a `loop`).

```
sqrt5 :: Double
sqrt5 = loop 100000 0
  where
    loop :: Int -> Int -> StdGen -> Double
    loop 0 k seed = 3 * (fromIntegral k / fromIntegral 100000)
    loop n k seed = loop (n - 1) k' seed'
      where
        (x, seed') = randomR (0, 3) seed
        k' = if (x * x <= 5) then k + 1 else k
```

See the notes for a more functional method.

3. This question is about abstract data representation and amortised analysis.

(a) The following is the interface for a `Stack`, which is a data structure that holds integers. The function `look` has a default implementation that can be overridden.

```
class Stack stack where
  empty :: stack
  push :: Int -> stack -> stack
  pop :: stack -> stack
  peek :: stack -> Maybe Int

  look :: Int -> stack -> Maybe Int
  look 0 xs = peek xs
  look i xs = look (i - 1) (pop xs)

  -- the following must hold for any implementation;
  -- pop (empty) = empty
  -- peek (empty) = Nothing
  -- pop (push x xs) = xs
  -- peek (push x xs) = Just x
```

i) Implement the list instance for `Stack`. Give the time complexity of the functions `push`, `pop`, and `peek`.

```
instance Stack [Int] where
  empty = []

  push x xs = x:xs        -- O(1)

  pop [] = []             -- O(1)
  pop (x:xs) = xs

  peek [] = Nothing       -- O(1)
  peek (x:xs) = Just x
```

ii) Explain the time complexity of the default implementation of `look` for lists.

The complexity of `look n xs` is in $O(n)$, since we are decrementing the n.

(b) The type `Array Int a` represents an array of values of type `a` indexed by values of type `Int`. A new array can be constructed with the `fromList` function;

```
fromList :: [Int] -> Array Int Int
```

Given a list of values `xs`, the array given by `fromList xs` takes $O(n)$ time to construct, where $n =$ `length xs`. The function `(!) :: Array Int a -> Int -> a` is such that given an array `ar` and an index `i`, the result of `ar ! i` is the value in the array `ar` at index `i`. The function `modify :: Array Int a -> Int -> a -> Array Int a` is such that given an array `ar`, an index `i`, and a value `x`, the result of `modify ar i x` is the array `ar` except that the value at `i` is now `x`. Assume this takes constant time.

i) Define a data type `StackArray` that contains `Array Int Int` and two `Int`s; the number of elements in the stack, and the maximum capacity of the array.

```
data StackArray = StackArray (Array Int Int) Int Int
```

ii) Define the `Stack StackArray` instance where the complexities of `empty`, `pop`, `peek`, and `look` are constant, as is the amortised complexity of `push`.

```
instance Stack StackArray where
  empty = StackArray (fromList [0]) 0 1

  pop (StackArray a 0 n) = StackArray a 0 n
  pop (StackArray a m n) = StackArray a (m - 1) n

  peek (StackArray a 0 n) = Nothing
  peek (StackArray a m n) = Just (a ! (n - m))

  look i (StackArray a m n)
    | i < m     = Just (a ! (n - m + i))
    | otherwise = Nothing

  push x (StackArray a m n)
    | m < n     = StackArray (modify a (m - m - 1) x) (m + 1) n
    | otherwise = StackArray (fromList ((replicate n x) ++ (elems a))) (m + 1) (2 * n)
```

iii) Prove that the amortised complexity of `push` is indeed constant.

For the amortised complexity, we must define A, C and S, and check that the following holds;

$$C_{\mathtt{op_i}}(\mathtt{xs_i}) \leq A_{\mathtt{op_i}}(\mathtt{xs_i}) + S(\mathtt{xs_i}) - S(\mathtt{xs_{i+1}})$$

This gives us;

$$\sum_{i=0}^{n-1} C_{\mathtt{op_i}}(\mathtt{xs_i}) \leq \sum_{i=0}^{n-1} A_{\mathtt{op_i}}(\mathtt{xs_i}) + S(\mathtt{xs_0}) - S(\mathtt{xs_n})$$

If $S(\mathtt{xs_0}) = 0$, then the following holds;

$$\sum_{i=0}^{n-1} C_{\mathtt{op_i}}(\mathtt{xs_i}) \leq \sum_{i=0}^{n-1} A_{\mathtt{op_i}}(\mathtt{xs_i})$$

We define this as follows, and we aim to prove the violet inequality;

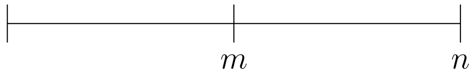
$$\begin{aligned}
C_{\mathtt{empty}}(\mathtt{xs}) &= 1 \\
C_{\mathtt{pop}}(\mathtt{xs}) &= 1 \\
C_{\mathtt{peek}}(\mathtt{xs}) &= 1 \\
C_{\mathtt{look}}(\mathtt{xs}) &= 1 \\
C_{\mathtt{push}}(\mathtt{StackArray\ a\ m\ n}) &= 2n \\
A_{\mathtt{op}}(\mathtt{xs}) &= 2
\end{aligned}$$

Given `StackArray a m n`, we want to show the following;

$$2n \leq 2 + S(\mathtt{xs_i}) - S(\mathtt{xs_{i+1}})$$

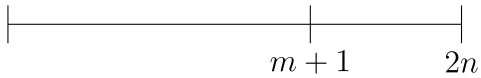
From this, we want to find the size as something that is close to $2n$ **before** the push, and close to 0 **after** the push.

Generally, the array is as follows (on the number line);



However, in the worst case, we have the following (this should ideally become size $2n$);

And after the push, we have the following (this should ideally become size 0);



As such, we can define the size function to be as follows;

$$S(\texttt{StackArray a m n}) = 2(2m - n)$$

Not sure if the above works with $A_{\texttt{op}}(\texttt{xs}_\texttt{i}) = 2$, but it does work with $= 4$. The screen was slightly cut off.