CO202 - Algorithms II

8th October 2019

Introduction

Note that this course is taught in Haskell, and in the style of Dijkstra (structure of algorithms), instead of Knuth (analysis and complexity).

List Insertion

An algorithm to insert elements in a sorted list;

```
1 insert :: Int -> [Int] -> [Int]
2 insert x [] = [x]
3 insert x (y:ys)
4   | x <= y    = x:y:ys
5   | otherwise = y:insert x ys
```

In Haskell, we do this by case analysis, first looking at the base case (line 2) - where the list is empty. The second case (line 3) considers the non-empty list. The evaluation is as follows, for a simple example;

	`insert 4 [1,3,6,7,9]`	
$\rightsquigarrow$	`1:insert 4 [3,6,7,9]`	definition of `insert`
$\rightsquigarrow$	`1:3:insert 4 [6,7,9]`	definition of `insert`
$\rightsquigarrow$	`1:3:4:6:[7,9]`	definition of `insert`

To give a cost, we will measure the number of steps, which approximates time - the number of steps is essentially each transition from the LHS of `=` to the RHS. The measure of input will be $n =$ `length xs`. We write a recurrence relationship that ties together$n$ with the algorithm;

$T(0) = 1$	1 transition
$T(n) = 1 + T(n-1)$	looking at worst case, line 5

The structure of the complexity should follow the structure of the algorithm itself. However, we are interested in a closed form for $T(n)$, where we can directly obtain the value without evaluating recursively. The easiest way to do this is to unroll the definition, and look for patterns;

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + (1 + T(n-2)) \\ &= 1 + (1 + \cdots + T(n-n)) \\ &= 1 + n \end{aligned}$$

Insertion Sort

The previous algorithm can be used as the basis for insertion sort. For each element in the unsorted list, we insert it into the sorted list (which is initially empty).

```
1 isort :: [Int] -> [Int]
2 isort [] = []
3 isort (x:xs) = insert x (isort xs)
```

We assume that `insert`, and `isort` both give us a sorted list, assuming the input lists were also sorted. An example of this on a small list is as follows;

	`isort [3,1,2]`	
$\rightsquigarrow$	`insert 3 (isort [1,2])`	definition of `isort`

```
⇝ insert 3 (insert 1 (isort [2]))                    definition of isort
⇝ insert 3 (insert 1 (insert 2 (isort [])))          definition of isort
⇝ insert 3 (insert 1 (insert 2 []))                  definition of isort
⇝ insert 3 (insert 1 [2])                            definition of insert
⇝ insert 3 (1:2:[])                                  definition of insert
⇝ 1:insert 3 (2:[])                                  definition of insert
⇝ 1:2:(insert 3 [])                                  definition of insert
⇝ 1:2:[3]                                            definition of insert
```

This cost 9 steps to evaluate. The recurrence relation generalises this (similarly $n =$ `length xs`);

$$T_{\texttt{isort}}(0) = 1$$
$$T_{\texttt{isort}}(n) = 1 + T_{\texttt{insert}}(n-1) + T_{\texttt{isort}}(n-1)$$

However, we want to find this in closed form;

$$\begin{aligned} T_{\texttt{isort}}(n) &= 1 + n + T_{\texttt{isort}}(n-1) \\ &= 1 + n + (1 + n - 1 + T_{\texttt{isort}}(n-2)) \\ &= \ldots \\ &= \frac{n(n+1)}{2} + 1 + n \end{aligned}$$

A more thorough analysis will teach us about;

- evaluation strategies and cost
- counting carefully and crudely
- abstract interfaces
- data structures

11th October 2019

Laziness

In the last lecture, we saw `isort` sorts in approximately n^2 steps.

```
1 minimum :: [Int] -> Int
2 minimum = head . isort
```

The evaluation of `minimum` takes n steps, when given a sorted list;

```
  minimum [1,2,3]
⇝ head (sort [1,2,3])
⇝ ...
⇝ head (insert 1 (insert 2 (insert 3 [])))
⇝ head (insert 1 (insert 2 [3]))
⇝ head (insert 1 (2:[3]))
⇝ head 1:2:[3]
⇝ 1
```

The worst case is a reversed list, as follows;

```
minimum [3,2,1]
```

$\rightsquigarrow$ ...

$\rightsquigarrow$ `head (insert 3 (insert 2 (insert 1 [])))`

$\rightsquigarrow$ `head (insert 3 (insert 2 [1]))`

$\rightsquigarrow$ `head (insert 3 (1:insert 2 []))`

$\rightsquigarrow$ `head (1:insert 3 (insert 2 []))`

The important part is to note that the minimum value, `1`, is floated to the left, for a total of n steps. Therefore, this still takes linear time. This evaluation relies on laziness, hence we can build the large computation on the RHS of the `:`.

Normal Forms

There are three normal forms that values can take;

- **normal form (NF)**

  This is fully evaluated, and there is no more work to be done - an expression is in NF if it is;

  - a constructor applied to arguments in NF
  - a λ-abstraction (function) whose body is in NF

- **head normal form (HNF)**

  An expression is in HNF if it is;

  - a constructor applied to arguments in any form
  - a λ-abstraction (function) whose body is in HNF

- **weak head normal form (WHNF)**

  An expression is in WHNF if it is;

  - a constructor applied to arguments in any form
  - a λ-abstraction (function) whose body is in any form

Looking at the last line in the previous evaluation, we have two constructors; cons (`:`) and the empty list (`[]`). The LHS of `:` is in normal form, but the RHS isn't, and therefore it cannot be in normal form.

Evaluation Order

There are two main evaluation strategies;

- **applicative order** (eager / strict evaluation) goes to normal form

  Evaluates as much as possible, until it ends up in normal form. It evaluates the left-most, inner-most expression first. For example, in the final step `head (1:insert 3 (insert 2 []))`, it would first evaluate `2`, then `[]`, and then `insert 2 []`, and so on.

- **normal order** (lazy evaluation) goes to weak head normal form

  This evaluates the left-most, outer-most expression first.

Counting Carefully

Here we are concerned at counting the steps mechanically in strict evaluation. This is done for a simplified language, containing constants, variables, functions, conditionals, and pattern matching. We will write e^T to denote the number of steps it takes to reduce e. Additionally, if f is a primitive function, then $f^T\ e_1\ \ldots\ e_n = 0$, otherwise $f\ e_1\ \ldots\ e_n = e$, and $f^T\ e_1\ \ldots\ e_n = 1 + e^T$.

$$k^T = 0 \qquad \text{constants}$$

$$x^T = 0 \qquad \text{evaluated variables}$$

$$(f\ e_1\ \ldots\ e_n)^T = (f^T\ e_1\ \ldots\ e_n) + e_1^T + \cdots + e_n^T \qquad \text{function with arguments}$$

$$(\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2)^T = p^T + (\texttt{if}\ p\ \texttt{then}\ e_1^T\ \texttt{else}\ e_2^T) \qquad \text{conditional}$$

$$\left(\texttt{case}\ e\ \texttt{of}\ \begin{cases} p_1 & \to e_1 \\ & \vdots \\ p_n & \to e_n \end{cases}\right)^T = e^T + \left(\texttt{case}\ e\ \texttt{of}\ \begin{cases} p_1 & \to e_1^T \\ & \vdots \\ p_n & \to e_n^T \end{cases}\right) \qquad \text{pattern matching}$$

This is very involved for tiny examples, and becomes much more complex for lazy evaluation.

Counting Crudely

We mainly use asymptotic notation to achieve this. Certain functions dominate others when given enough time - as the input increases.

L-functions are the smallest class of one-valued functions on real variables $n \in \mathbb{R}$, containing constants, the variable n, and are closed under arithmetic, exponentiation, and logarithms. They tend to be monotonic after a given time, and tend to a value.

Consider $f(n) = 2n$, and $g(n) = \frac{n^2}{4}$ - at $n = 1$, $f(1) > g(1)$, however at some point on the number line, g begins to dominate. Comparing functions can be achieved by studying their ratios (with well-behaved functions, like L-functions, the ratio will tend to 0, infinity, or a constant);

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

Any L-function is ultimately continuous of constant sign, monotonic, and approaches 0, ∞, or some definite limit as $n \to \infty$. Furthermore, $\frac{f}{g}$ is an L-function if both f and g are. We can now introduce notation compare function;

$$f \prec g \triangleq \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad \text{also written as } f \in o(g(n))$$

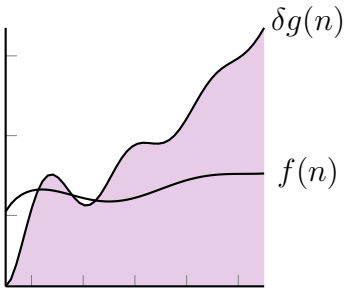
$$f \preceq g \triangleq \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \qquad \text{also written as } f \in O(g(n))$$

$$f \asymp g \triangleq f \in (O(g(n)) \cap \Omega(g(n))) \qquad \text{also written as } f \in \Theta(g(n))$$

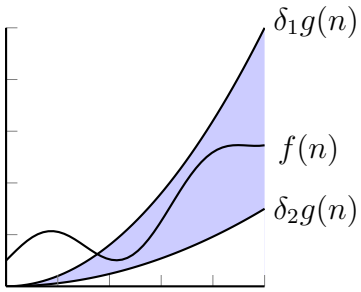
$$f \succeq g \triangleq \limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0 \qquad \text{also written as } f \in \Omega(g(n))$$

$$f \succ g \triangleq \lim_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| = \infty \qquad \text{also written as } f \in \omega(g(n))$$

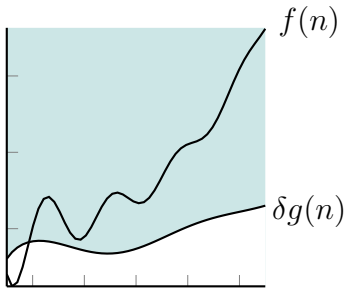
Visually, we can represent this in the following three graphs. Note that $\delta, \delta_1, \delta_2$ are just constant multipliers. The first plot shows that as n gets larger $f(n)$ will exist within the shaded region bounded above by $\delta g(n)$, and similarly (on the other extreme) the third plot shows that as n gets larger, $f(n)$ will exist within the region bounded below by $\delta g(n)$. If f is constrained (as time progresses) within the region bounded by $\delta_1 g(n)$ and $\delta_2 g(n)$, then we have the second plot.



$f(n) \in O(g(n))$
$f \preceq g$



$f(n) \in \Theta(g(n))$
$f \asymp g$

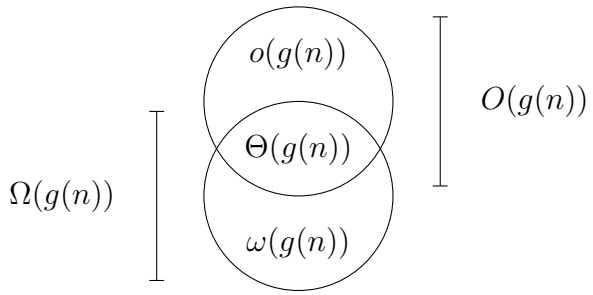


$f(n) \in \Omega(g(n))$
$f \succeq g$

If f and g are L-functions, then either;

$$f \in o(g),\ f \in \Theta(g), \text{ or } f \in \Omega(g)$$

Another method of visualising this is as a Venn diagram, with the upper circle being $O(g(n))$, and the lower circle being $\Omega(g(n))$;



Finally, this can also be defined by the following;

$$\begin{aligned}
o(g(n)) &= \{f \mid \forall \delta > 0.\ \exists n_0 > 0.\ \forall n > n_0.\ |f(n)| < \delta g(n)\} \\
O(g(n)) &= \{f \mid \exists \delta > 0.\ \exists n_0 > 0.\ \forall n > n_0.\ |f(n)| \leq \delta g(n)\} \\
\Theta(g(n)) &= \left\{ f \;\middle|\; \begin{array}{c} (\exists \delta > 0.\ \exists n_0 > 0.\ \forall n > n_0.\ |f(n)| \leq \delta g(n)) \\ \wedge \\ (\exists \delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ f(n) \geq \delta g(n)) \end{array} \right\} \\
&= O(g(n)) \cap \Omega(g(n)) \\
\Omega(g(n)) &= \{f \mid \exists \delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ f(n) \geq \delta g(n)\} \\
\omega(g(n)) &= \{f \mid \forall \delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ f(n) > \delta g(n)\}
\end{aligned}$$

15th October 2019

Basic Lists

In Haskell, lists are given by **two** constructors;

```
1 data [a] = [] | (:) a [a]
```

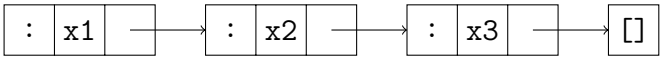
This creates two values, called constructors (RHS of data declaration, hence we can pattern match on these unlike other functions);

```
1 [] :: [a]
2 (:) :: a -> [a] -> [a]
```

In Haskell, data structures are persistent by default.

```
[x1,x2,x3] = x1:x2:x3:[]
```

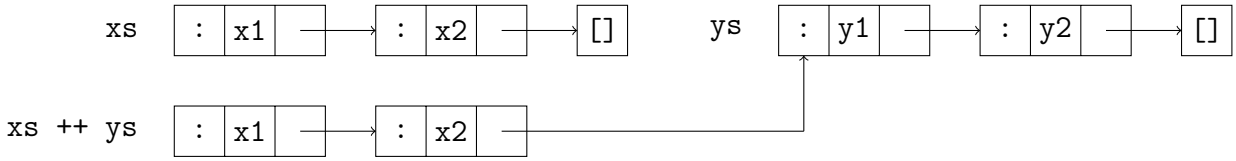
Visually, we can look at this as "cells" with their constructors and arguments;



Appending lists together is achieved with `(++)`;

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

When we do `xs ++ ys`, the final structure points to `ys`. The trade-off here is that we didn't have to modify `ys`, but we had to create a new `x1`, and `x2`;



Therefore, the complexity is linear, but only depends on `xs`, and not `ys`, hence we can say;

$$T_{(\texttt{++})} \in O(n), \text{ where } n = \texttt{length xs} \text{ in } \texttt{xs ++ ys}$$

Folding

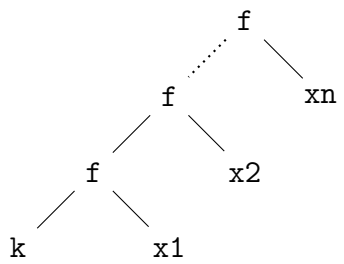
The structure of lists is completed reduced by the `foldr` function;

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k [] = k
foldr f k (x:xs) = f x (foldr f k xs)
```

This replaces `(:)` with `f`, and `[]` with `k`;

Note that doing `foldr (:) []` is the same as `id`, but with more work, and also `xs ++ ys` is equivalent to `foldr (:) ys xs`. Similarly, we have `foldl`;

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f k [] = k
foldl f k (x:xs) = foldl f (f k s) xs
```



Consider a binary operator ($\diamond$);

$$x \diamond (y \diamond z) = (x \diamond y) \diamond z \qquad \diamond - \text{associative}$$
$$\epsilon \diamond y = y \qquad \epsilon - \text{left-unit}$$
$$x \diamond \epsilon = x \qquad \epsilon - \text{right-unit}$$

When folding with $\diamond$ and ϵ, `foldr` and `foldl` coincide.

List Concatenation

An application of this is list concatenation, consider `concat`;

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

This can be written as a `foldr`;

```
rconcat :: [[a]] -> [a]
rconcat = foldr (++) []
```

Since `(++)` is associate with a neutral element `[]`, we can write a `foldl` version;

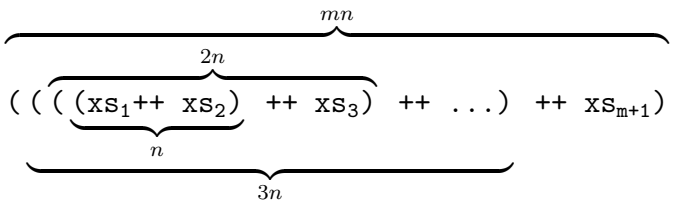
```
lconcat :: [[a]] -> [a]
lconcat = foldl (++) []
```

While `lconcat = rconcat`, this only represents extensional (what values are produced) equality. They are not intensionally (how those values are produced) equal.

18th October 2019

Concatenation and Associativity

The complexity of `xs ++ ys` is; $T_{(++)}(m, n) \in O(m)$, where $m =$ `length xs`, and $n =$ `length ys` - note that it doesn't consider n as expected. We want to consider the complexity of;



In lieu of having individual variables, we can approximate all the lists `xs1` to $\texttt{xs}_{\texttt{m+1}}$ as having the same length n. This means that if $m =$ `length xss`, then `concat xss` has complexity;

$$n + 2n + 3n + \cdots + mn = n(1 + 2 + 3 + \cdots + m) \in O(nm^2)$$

If we now look at the right associative version;

$$\texttt{xs}_1 \texttt{ ++ (xs}_2 \texttt{ ++ (xs}_3 \texttt{ ++ (... ++ (xs}_\texttt{m} \texttt{ ++ xs}_{\texttt{m+1}}\texttt{))))}$$

Note that each of the `++` operations cost n, and we have m of them, under the same assumption. Therefore, we have $O(mn)$.

Note that adding to the right of a list is similar to doing the first version, which is left associative, and also more expensive. Preferably, we'd add to the left of the list, however this isn't always possible, as we'd also like to be able to add to the right of the list.

Function Composition

We do not want to suffer the consequences of poor associativity. To solve this, we notice that function composition pays no associativity penalty. First note that function composition is as follows;

```
(f . g) x = f (g x)
```

We can also show that function composition is both extensionally and intensionally equal, by unrolling the definition;

```
((f . g) . h) x = f (g (h x))
```

vs

```
(f . (g . h)) x = f (g (h x))
```

For our lists, we will replace with;

$$((xs_1\ ++)\ .\ (xs_2\ ++)\ .\ \ldots\ .\ (xs_{m+1}\ ++))\ []$$

Note that each of $(xs_i\ ++)$ is a partially applied function that takes a list, and adds xs_i to the start. This is equivalent to the right associative version we had before, which has a better time complexity.

Difference List (DList)

The idea is to replace lists with functions that take in a list and give a list back. Previously, we'd consider xs_1 a list, but we now use $(xs_1\ ++)$, which needs to be applied to the empty list. A value of type `a` in the list is now a function of type `[a] -> [a]`. We can create a new datatype;

```
data DList a = DList ([a] -> [a])
```

We need ways of making lists from DLists, and vice versa;

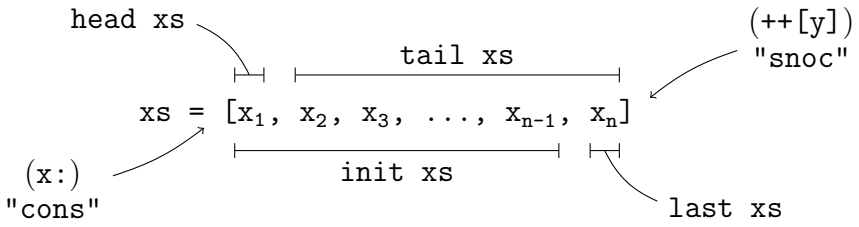
```
-- technically called the abstraction function
toList :: DList a -> [a]
toList (DList fxs) = fxs []

fromList :: [a] -> DList a
fromList xs = DList (xs ++)
-- or         DList (\ys -> xs ++ ys)
```

We also need a way of adding DLists together (note the notes use $\diamond$, but I can't really get that with the listing environment, so I will use `<>`);

```
(<>) :: DList a -> DList a -> DList a
DList fxs <> DList fys = DList (fxs . fys)
-- or                    DList (\zs -> fxs (fys zs))
```

Note that taking the head or tail of this list now requires constructing the entire list, which will take linear time.

List Anatomy



List Interface

Rather than dealing with lists directly, we want to work with a specification instead. For this, we are creating a type class - an abstraction mechanism that has a class of objects, which have certain operations - and depending on what is given the operations can operate slightly differently.

```
class List list where
empty :: 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
length :: list a -> Int
```

```
(++) :: list a -> list a -> list a
null :: list a -> Bool
single :: list a -> Bool

toList :: list a -> [a]
fromList :: [a] -> list a
```

Note that `toList` and `fromList` must behave nicely when composed;

$$\texttt{id = toList . fromList}$$

However, composing the functions in reverse does not necessarily yield the same result. Consider the case with our DList - it would be perfectly valid to have `(DList reverse)`; however, since the space of functions is much greater than the space of lists, we cannot reasonably convert `revert` into a list, and then hope to get `reverse` back out.

However, we can say that if we had something representing a list, converting it to a list and then back to our implementation of a list, should yield something similar.

$$\texttt{normalise = fromList . toList}$$

Exercises

1. Prove formally that $(n+1)^2 \in \Theta(n^2)$ by exhibiting the necessary constants.

   We set $f(n) = (n+1)^2$, and $g(n) = n^2$. By our definition of Θ, we need to satisfy both the following;

   $$O(g(n)) = \{f \mid \exists \delta > 0.\ \exists n_0 > 0.\ \forall n > n_0.\ |f(n)| \leq \delta g(n)\}$$
   $$\Omega(g(n)) = \{f \mid \exists \delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ f(n) \geq \delta g(n)\}$$

   The first case, we want to find δ such that;

   $$\begin{aligned} (n+1)^2 &\leq \delta n^2 && && \Leftrightarrow \\ n^2 + 2n + 1 &\leq \delta n^2 && && \Leftrightarrow \\ 0 &\leq (\delta - 1)n^2 - 2n - 1 && \text{assuming } \delta - 1 = 3, && \Rightarrow \\ 0 &\leq (3n+1)(n-1) \end{aligned}$$

   Hence we have $\delta = 4$. The second case, Ω, is trivial to prove with $\delta = 1$.

2. Give examples of expressions that are;

   1. in head normal form, but not in normal form — `5:take 2 [1,2,3]`
   2. in weak head normal form, but not in head normal form — `\x -> take 2 [1,2,3]`
   3. in no normal form of any kind — `take 2 [1,2,3]`
      it's not a constructor, and there isn't a λ-abstraction
   4. in normal form, but not in weak head normal form — not possible

3. Using the definition of $f \in O(g(n))$ as a set, derive the definition of $f \in \omega(g(n))$ as a set.

   Recall that $f \in \omega(g(n)) \Leftrightarrow f \notin O(g(n))$, and the following definition;

   $$O(g(n)) = \{f \mid \exists \delta > 0.\ \exists n_0 > 0.\ \forall n > n_0.\ |f(n)| \leq \delta g(n)\}$$

Using equivalences for first-order logic;

$$\begin{aligned}
&\neg\exists\delta > 0.\ \exists n_0 > 0.\ \forall n > n_0.\ |f(n)| \leq \delta g(n) \\
\equiv\ &\forall\delta > 0.\ \neg\exists n_0 > 0.\ \forall n > n_0.\ |f(n)| \leq \delta g(n) \\
\equiv\ &\forall\delta > 0.\ \forall n_0 > 0.\ \neg\forall n > n_0.\ |f(n)| \leq \delta g(n) \\
\equiv\ &\forall\delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ \neg(|f(n)| \leq \delta g(n)) \\
\equiv\ &\forall\delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ f(n) > \delta g(n)
\end{aligned}$$

Hence we can write ω as the following;

$$\omega(g(n)) = \{f \mid \forall\delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ f(n) > \delta g(n)\}$$

4. The composition rule gives the time complexity of two functions f and g when they are composed. By using the definition of e^T, derive the accurate cost of $f\ (g\ x)$ using strict evaluation.

$$\begin{aligned}
((f \circ g)\ x)^T &= 1 + (f(g(x)))^T \\
&= 1 + f^T(g(x)) + (g(x))^T \\
&= 1 + f^T(g(x)) + g^T x + x^T \\
&= 1 + f^T(g(x)) + g^T x
\end{aligned}$$

5. Justify whether each of the following is true or false;

   1. $2n^2 + 3n \in \Theta(n^2)$ — true
   2. $2n^2 + 3n \in O(n^3)$ — true
   3. $n \log n \in O(n\sqrt{n})$ — ($\log n$ grows slower than $\sqrt{n}$) true
   4. $n + \sqrt{n} \in O(n \log n)$ — false
   5. $2^{\log n} \in O(n)$ — (use laws of logarithms) true

22nd October 2020

Double-Ended Queues (Deque)

These are sometimes called symmetric lists. A normal list has the following complexities;

- `cons x xs` $O(1)$
- `head xs` $O(1)$
- `tail xs` $O(1)$
- `snoc x xs` $O(n)$
- `last xs` $O(n)$
- `init xs` $O(n)$

We define a Deque as two lists, where the first list is in normal order, and the second list, containing the remainder of the whole list, is in reverse order (see the `toList` function)

```
1 data Deque a = Deque [a] [a]
2
3 toList :: Deque a -> [a]
4 toList (Deque xs ys) = xs ++ reverse ys
```

To add to the end of the list, we add to the front of the second list, hence we have constant time `snoc`, as well as constant time `cons`.

To get desirable asymptotic complexities, there are two invariants that we require;

- `null xs` $\Rightarrow$ `null ys` $\vee$ `single ys`
- `null ys` $\Rightarrow$ `null xs` $\vee$ `single xs`

To make a deque from a list, some symmetry is preferable.

```
1  fromList :: [a] -> Deque [a]
2  fromList xs = Deque us (reverse vs)
3    where
4      (us, vs) = splitAt (div n 2) xs
5      n = length xs
```

Some operations are easy to analyse and define;

```
1  snoc :: a -> Deque a -> Deque a
2  snoc y (Deque [] ys) = Deque ys [y]
3  snoc y (Deque xs ys) = Deque xs y:ys
```

This has $O(1)$ complexity. It's important to note that we can do line 2 due to our invariant - if we know `xs` is empty, then `ys` must also be empty, or be a singleton list - and therefore it can be placed as the first list without a reversal.

```
1  last :: Deque a -> a
2  last (Deque xs []) = head xs -- or last xs
3  last (Deque xs y:ys) = y
```

This is also $O(1)$ complexity. Similarly, we know that `xs` is either empty (which would cause an error anyways), or it is singleton - thus the last item is the only item in the deque.

```
1  empty :: Deque a
2  empty = Deque [] []
3
4  tail :: Deque a -> Deque a
5  tail (Deque [] ys) = empty
6  tail (Deque [x] ys) = Deque (reverse vs) us
7    where
8      (us, vs) = splitAt (div n 2) ys
9      n = length ys
10 tail (Deque xs ys) = Deque (tail xs) ys
```

Here we define the empty deque, which is used in the first case (and doesn't cause an error). In the third case, we know that there is more than one element in `xs` (since it didn't match the second case), and therefore we simply take the tail (since we won't have an empty list, unlike the second case). In the second case, we discard the single `x`. Symmetry is maintained by using the same split as `fromList`. Note that `ys` is reversed, hence we can write `ys = [`y_n`, `y_{n-1}`, ..., `y_0`]`, and therefore `(us, vs) = ([`y_n`, `y_{n-1}`, ..., `y_j`], [`y_{j-1}`, `y_{j-2}`, ..., `y_0`])`, therefore `vs` needs to be reversed and put first. Both the first, and third cases are $O(1)$, but the worst case complexity is the second case, which has the reversal, and is $O(n)$.

Amortised Analysis

In the worst case, the cost is $O(n)$, where $n =$ `length ys`. However, we rarely encounter this case.

The idea of amortised analysis is that while it may have a high complexity in a single instance of the worst case, but it may not cost that amount in a sequence of operations. The cost of some functions is better expressed in terms of how it performs in a wider context.

Consider repeated applications of `tail`, in a chain (such that we apply `tail` to the result of the previous application) - each successive call to `tail` will cost less;

$$\texttt{xs}_0 \overset{\texttt{tail}}{\rightsquigarrow} \texttt{xs}_1 \overset{\texttt{tail}}{\rightsquigarrow} \texttt{xs}_2 \overset{\texttt{tail}}{\rightsquigarrow} \ldots \overset{\texttt{tail}}{\rightsquigarrow} \texttt{xs}_n$$

We can use amortised analysis to work out the true cost.

$$\texttt{xs}_0 \overset{\texttt{op}_0}{\rightsquigarrow} \texttt{xs}_1 \overset{\texttt{op}_1}{\rightsquigarrow} \texttt{xs}_2 \overset{\texttt{op}_2}{\rightsquigarrow} \ldots \overset{\texttt{op}_n}{\rightsquigarrow} \texttt{xs}_{n+1}$$

To perform amortised analysis, we need to define the following;

- **cost** assign a real cost $C_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i})$ for each operation $\texttt{op}_\texttt{i}$ on data $\texttt{xs}_\texttt{i}$
- **amortised cost** assign an amortised cost $A_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i})$ for each operation $\texttt{op}_\texttt{i}$ on data $\texttt{xs}_\texttt{i}$
- **size** define $S(\texttt{xs})$ that calculates the size of the data $\texttt{xs}$

We use these to establish the following;

$$\underbrace{\sum_{i=0}^{n-1} C_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i})}_{\text{total actual cost}} \leq \underbrace{\sum_{i=0}^{n-1} A_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i})}_{\text{total amortised cost}}$$

For example, if we choose $A_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i}) = 1$, then the total cost is $O(n)$. To prove this inequality, we use the size function (physicist's approach);

$$C_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i}) \leq A_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i}) + \overbrace{\underbrace{S(\texttt{xs}_\texttt{i})}_{\text{input size}} - \underbrace{S(\texttt{xs}_{\texttt{i+1}})}_{\text{output size}}}^{\text{dif between data structure}}$$

We can sum over this as follows;

$$\sum_{i=0}^{n-1} C_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i}) \leq \sum_{i=0}^{n-1} A_{\texttt{op}_\texttt{i}}(\texttt{xs}_\texttt{i}) + S(\texttt{xs}_0) - S(\texttt{xs}_\texttt{n})$$