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;

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

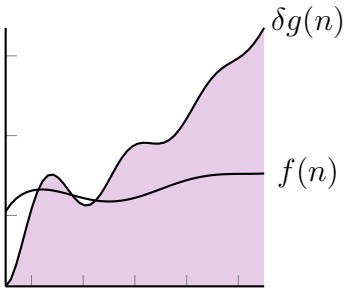
$$f \preceq g \triangleq \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \qquad \text{also written as } f \in O(g(n))$$

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

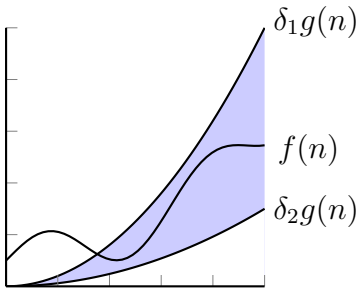
$$f \succeq g \triangleq \limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0 \qquad \text{also written as } f \in \Omega(g(n))$$

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

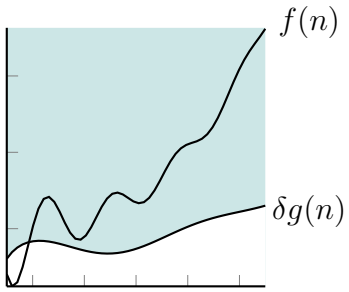
Visually, we can represent this in the following three graphs. Note that $\delta, \delta_1, \delta_2$ are just constant multipliers. The first plot shows that as n gets larger $f(n)$ will exist within the shaded region bounded above by $\delta g(n)$, and similarly (on the other extreme) the third plot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



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



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

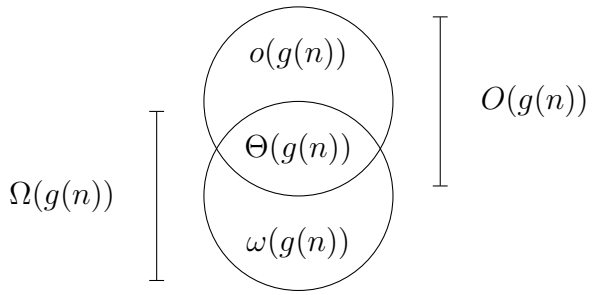


$f(n) \in \Omega(g(n))$
$f \succeq g$

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]
```

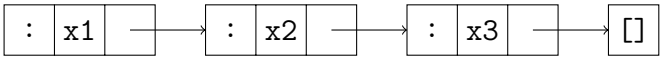
This creates two values, called constructors (RHS of data declaration, hence we can pattern match on these unlike other functions);

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

In Haskell, data structures are persistent by default.

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

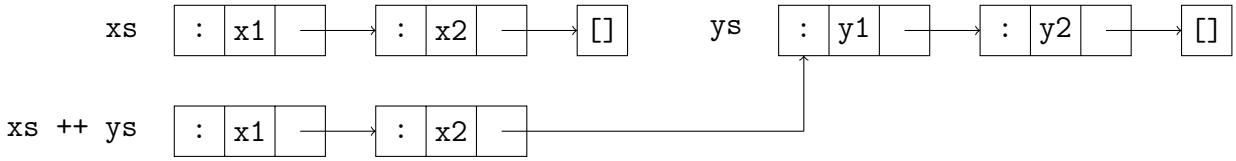
Visually, we can look at this as "cells" with their constructors and arguments;



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

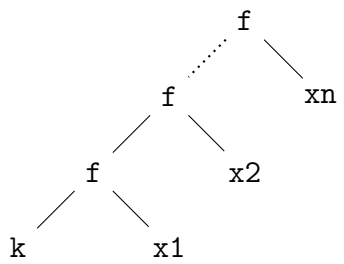
The structure of lists is completed reduced by the `foldr` function;

```
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;

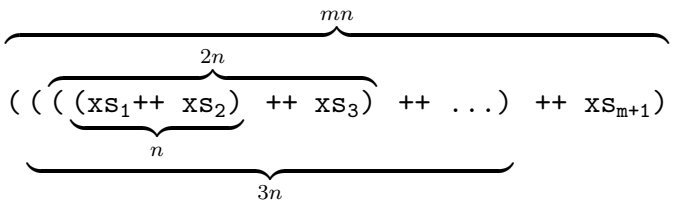
```
lconcat :: [[a]] -> [a]
lconcat = foldl (++) []
```

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 `DList`s, and vice versa;

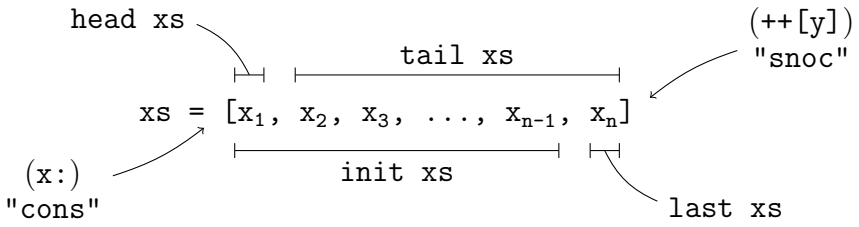
```
-- technically called the abstraction function
toList :: DList a -> [a]
toList (DList fxs) = fxs []

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

We also need a way of adding `DList`s 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. Note that `List` is the name of the type class, and `list` is the variable of abstraction.

```
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 $\Theta(1)$ 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 (let it be equation (1)), 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;

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

Let this be equation 2.

25th October 2019

Size Function

One way of thinking about the size function is that it is similar to a bank; when a cheap operation is performed, we can put money in, and when an expensive operation is performed, we take money out.

If we show equation 2, we have shown equation 1 - given that $S(\texttt{xs}_0) = 0$.

Example on Deques

First, we can assign costs as follows;

$$\begin{aligned} C_{\texttt{cons}}(\texttt{xs}) &= 1 \\ C_{\texttt{snoc}}(\texttt{xs}) &= 1 \\ C_{\texttt{head}}(\texttt{xs}) &= 1 \\ C_{\texttt{last}}(\texttt{xs}) &= 1 \\ C_{\texttt{tail}}(\texttt{Deque xs ys}) &= k \qquad \text{where } k = \texttt{length ys} \end{aligned}$$

Now, we can assign amortised costs. Regardless of the operation, we can say the charge is 2, hence we are overcharging by 1 for each of the constant time operations, in the hope that when we get to `tail`, we have enough "saved up";

$$A_{\texttt{op}}(\texttt{Deque xs ys}) = 2$$

Finally, we can define the size function to be the difference in size of `xs` and `ys`. The expensive operation of doing `tail` happens when one of the two lists is empty, and reversal has to be done. Once this "distance" increases to a certain amount, we have to charge the expensive operation.

$$S(\texttt{Deque xs ys}) = |\texttt{length xs} - \texttt{length ys}|$$

We can now attempt to prove equation 2. Given `Deque xs' ys' = tail (Deque xs ys)`, we have the following sizes (in the worst case, where `xs` is a singleton list, and `ys` is of length k, and after the operation the difference is at most 1, since the list is (approximately) symmetric);

$$S(\texttt{Deque xs ys}) = k - 1$$
$$S(\texttt{Deque xs' ys'}) = 1$$

Substituting gives the following;

$$\underbrace{C_{\texttt{tail}}(\texttt{Deque xs ys})}_{=k} \leq \underbrace{A_{\texttt{tail}}(\texttt{Deque xs ys})}_{=2} + \underbrace{S(\texttt{Deque xs ys})}_{=k-1} - \underbrace{S(\texttt{Deque xs' ys'})}_{=1}$$
$$\Leftrightarrow$$
$$k \leq 2 + (k-1) - 1 = k \text{ (which is true)}$$

Note that if we overcharged (had a number higher than 2), we'd still have a constant cost.

Counting

Once again, we encounter Peano numbers (see **CO142** from last year);

```
data Peano = Zero | Succ Peano

inc :: Peano -> Peano
inc n = Succ n

-- notice this errors for Zero
dec :: Peano -> Peano
dec (Succ n) = n

add :: Peano -> Peano -> Peano
add Zero n = n
add (Succ m) n = Succ (add m n)
```

This has a duality with lists;

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

-- mirrors inc
cons :: a -> [a] -> [a]
cons x xs = x:xs

-- mirrors dec (notice the error case)
tail :: [a] -> [a]
tail (x:xs) = xs

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

Counting in binary is as follows (our representation has the least significant bit first);

```
type Binary = [Bit] -- [] is decimal 0
data Bit = 0 | 1

inc :: Binary -> Binary
inc [] = [1]
inc (0:bs) = 1:bs
inc (1:bs) = 0:inc bs
```

Notice that, in the worst case, `inc` has complexity n, where n is the number of bits. Once again, doing amortised analysis, we define the following;

$$C_{\texttt{inc}}(\texttt{bs}) = t + 1 \quad \text{where } t = \texttt{length (takeWhile (== 1) bs)}$$
$$A_{\texttt{inc}}(\texttt{bs}) = 2 \quad \text{overcharge for amortised cost}$$
$$S(\texttt{bs}) = \texttt{length (filter (== 1) bs)}$$

Size function is some kind of measure of how much "money" we have in the "bank". We're "paying" when we have a lot of `1`s, therefore for any `1` we put down, we have to pay for it later - hence we define the size function as the number of `1`s in the bit string. Additionally, recall that having $S(\texttt{xs}_0) = 0$ allows us to show equation 1, by showing equation 2 (which can be done with the inequality, without the summation).

Given `bs' = inc bs`, we want to verify the following;

$$\underbrace{C_{\texttt{inc}}(\texttt{bs})}_{=t+1} \leq \underbrace{A_{\texttt{inc}}(\texttt{bs})}_{=2} + \underbrace{S(\texttt{bs})}_{=b} - \underbrace{S(\texttt{bs'})}_{=b'}$$

Note that we can relate b and b' as $b' = b - t + 1$, since we flip all the leading `1`s to `0`s (hence the $-t$), and then flip a `0` to a `1` (hence the $+1$). By doing those substitutions, we can conclude the following;

$$t + 1 \leq 2 + b - (b - t + 1) = t + 1 \text{ (which is true)}$$

Note that we want A to be as tight of a bound as possible - while the inequality would still hold true if we set A to be the number of bits, we'd end up with a poor estimate.

Relation with Lists

Comparing binary numbers to Peano numbers, we are going up in powers of two, and therefore we can communicate larger numbers with less space. Relating this back to lists, searching for an item by traversing the entire list mirrors Peano numbers.

Binary numbers are in the form;

$$b_0 b_1 b_2 \dots b_n \text{ meaning } 2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \dots 2^n b_n$$

We will work with a list representation that has $1, 2, 4, \dots, 2^n$ elements. Therefore, for a list;

$$[\texttt{t}_0, \texttt{t}_1, \dots, \texttt{t}_n]$$

We will store 1 element in the $\texttt{t}_0$ position, 2 elements in the $\texttt{t}_1$ position, and 2^n elements in the $\texttt{t}_n$ position. Depending on how many elements we want, we will either store or not store in these positions - consider having 6 elements; nothing will be stored in the $\texttt{t}_0$ position, but we will store 2 elements in $\texttt{t}_1$, and 4 elements in $\texttt{t}_2$. To do this, we will be using trees, which also gives us logarithmic access.

Looking up in a list is expensive (takes $\Theta(k)$ instructions);

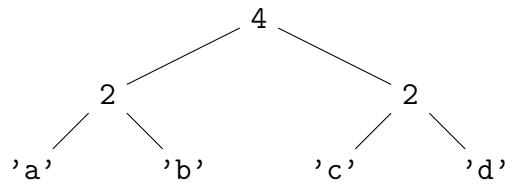
```
(!!) :: [a] -> Int -> a
(x:xs) !! 0 = x
(x:xs) !! k = xs !! (k - 1)
```

To improve upon this, we seek $O(\log n)$ instead.

Trees

This can be done with trees (note that the `Int` in the `Node` constructor is the size of the tree);

```
data Tree a = Leaf a | Node Int (Tree a) (Tree a)
```




```
size :: Tree a -> Int
size (Leaf x) = 1
size (Node n l r) = n
```

In lieu of taking the sum of the subtrees, we can simply maintain an invariant that has `n` be the size of the tree. This can be done with a **smart constructor**;

```
node :: Tree a -> Tree a -> Tree a
node l r = Node (size l + size r) l r
```

We can imagine that this has an instance of the `List` class;

```
instance List Tree where
  toList :: Tree a -> [a]
  toList (Leaf x) = [x]
  toList (Node n l r) = toList l ++ toList r -- we can do better here
  ...
  (!!) :: Tree a -> Int -> a
  Leaf x !! 0 = x
  Node n l r !! k
    | k < m     = l !! k
    | otherwise = r !! k - m
    where
      m = size l
```

Random Access Lists (RALists)

Lists can be represented using binary numbers as follows;

```
type RAList a = [Maybe (Tree a)]

instance List RAList where -- can't actually do this on type synonyms
  toList :: RAList a -> [a]
  toList = concat . map to
    where
      to :: Maybe (Tree a) -> [a]
      to Nothing = []
      to (Just t) = toList t -- note this is toList :: Tree a -> [a]
  (!!) :: RAList a -> Int -> a
  (Nothing : ts) !! k = ts !! k
  (Just t : ts) !! k
    | k < m     = t !! k -- (!!) for Tree (overloading)
    | otherwise = ts !! k - m
    where
      m = size t
```

We have the following extensional equality;

$$\texttt{toList txs !! k} = \texttt{txs !! k} \text{ where } \texttt{tks :: RAList a}$$

However, the RHS performs better, as the LHS constructs the list, and then performs a walk down it. The complexity of `(!!)` (on `RAList`) is $O(\log k)$.

To implement `cons`, we need to mirror `inc` on binary numbers;

```
cons :: a -> RAList a -> RAList a
cons x xs = consT (Leaf x) xs
  where
    consT :: Tree a -> RAList a -> RAList a
    consT t [] = [Just t]
    consT t (Nothing:ts) = (Just t):ts
    consT t ((Just t'):ts) = Nothing:consT (node t t') ts
```

29th October 2019

Divide and Conquer, Merge Sort

This strategy has the following three parts;

1. divide a problem into subproblems
2. apply the strategy to subproblems to make subsolutions
3. combine subsolutions to make a solution

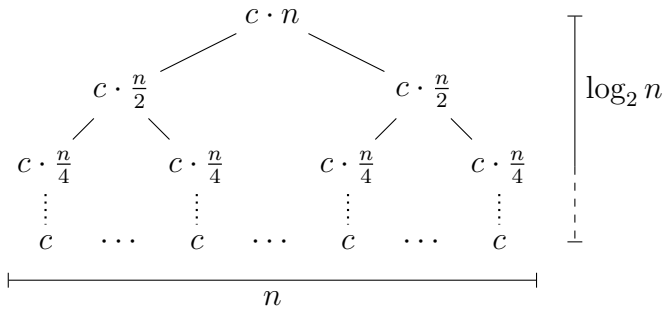
A classic example is merge sort;

```
msort :: [Int] -> [Int]
msort [] = []
msort [x] = [x]
msort xs = merge (msort us) (msort vs)
  where
    (us, vs) = splitAt (div n 2) xs
    n = length xs

merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x:merge xs (y:ys)
  | otherwise = y:merge (x:xs) ys
```

We can perform the following recurrence relation to obtain the complexity;

$$
\begin{aligned}
T_{\texttt{msort}}(0) &= 1 \\
T_{\texttt{msort}}(1) &= 1 \\
T_{\texttt{msort}}(n) &= \underbrace{T_{\texttt{length}}(n)}_{O(n)} + \underbrace{T_{\texttt{splitAt}}\left(\frac{n}{2}\right)}_{O(n)} + \underbrace{T_{\texttt{merge}}\left(\frac{n}{2}\right)}_{O(n)} + 2 \cdot T_{\texttt{msort}}\left(\frac{n}{2}\right)
\end{aligned}
$$

For this, we can draw a recursion tree to represent the structure (note the initial $c \cdot n$ represents the three $O(n)$ costs combined, and so on);



Therefore, we can conclude the following; $T_{\texttt{msort}}(n) \in \Theta(n \log n)$

Quicksort

For this implementation, we will use the head of the list as the pivot - using an arbitrary element is better.

```
qsort :: [Int] -> [Int]
qsort [] = []
qsort [x] = [x]
```

```
qsort (x:xs) = qsort us ++ [x] ++ qsort vs
  where
    (us, vs) = partition (< x) xs

partition :: (a -> Bool) -> [a] -> ([a], [a])
partition p xs = (filter p xs, filter (not . p) xs)
```

In the best case, the cost is the same as `msort`, since we are splitting the list in the middle. However, in the worst case we have the following;

$$\begin{aligned} T_{\texttt{qsort}}(n) &= \underbrace{T_{\texttt{partition}}(n)}_{O(n)} + \underbrace{T_{(\texttt{++})}(n-1)}_{O(n)} + T_{\texttt{qsort}}(1) + T_{\texttt{qsort}}(n-1) \\ &= c \cdot n + T_{\texttt{qsort}}(n-1) \\ &= \underbrace{c \cdot n + c \cdot n + \cdots + c \cdot n}_{n} + 1 \\ &= c \cdot n^2 + 1 \end{aligned}$$

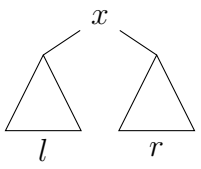
Hence $T_{\texttt{qsort}} \in O(n^2)$.

Binary Search Trees / AVL Trees

We start with a tree which mirrors `qsort`;

```
data QTree a = QNil | Node (QTree a) a (QTree a)
```

The idea is that, for the diagram on the right, elements in l are less than x and elements in r are greater than x;




```
fromList :: [Int] -> QTree Int
fromList [] = QNil
fromList (x:xs) = QNode (fromList us) x (fromList vs)
  where
    (us, vs) = partition (< x) xs
```

Notice that this essentially encapsulates the call structure of `qsort` in memory. Flattening this tree will give us a sorted list, equivalent to `qsort`. The `fromList` function has the same complexity as `qsort`, which is $O(n \log n)$ in the best case, and $O(n^2)$ in the worst case.

The goal is to change the data structure so that it takes $O(n \log n)$ on average. To achieve this performance, we want to keep the tree balanced - hence we want to store the height of the tree (in lieu of traversing it, since we will use this operation frequently). A binary search tree has this representation (with the `Int` representing height, and a smart constructor `bnode` to maintain this property);

```
data BTree a = BNil | BNode Int (BTree a) a (BTree a)

height :: BTree a -> Int
height BNil = 0
height (BNode h l x r) = h

bnode :: BTree a -> a -> BTree a -> BTree a
bnode l x r = BNode h l x r
  where
    h = 1 + max (height l) (height r)
```

The algorithm will consist of repeated inserts into the tree - the `fromList` function will take a list of elements and `foldr` over it. The `insert` function must maintain the balance of the tree.

```
fromList :: [Int] -> BTree Int
fromList xs = foldr insert BNil

insert :: Int -> BTree Int -> BTree Int
insert x BNil = bnode BNil x BNil
insert x (BNode h l y r)
  | x < y  = balance (insert x l) y r
  | x == y = BNode h l y r -- discarding if equal
  | x > y  = balance x y (insert x r)
```

Note that `balance` is another smart constructor.

1st November 2019

BTrees (Continued)

The intuition is that the tree given to `insert` is roughly balanced, such that the difference in heights of the left and right subtrees are at most 1. We define a tree to be **biased** if its subtrees differ in height;

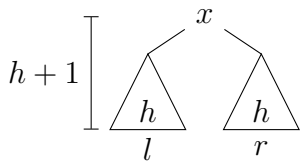
```
bias :: BTree a -> Int
bias BNil = 0
bias (BNode h l x r) = height l - height r
```

Therefore, we are off by at most 2 after performing the insertion. The `balance` smart construction ensures the trees are balanced when they are originally off by at most 2;

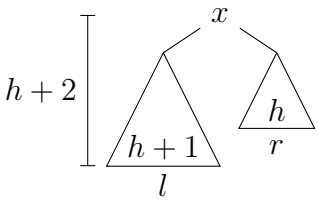
```
balance :: BTree Int -> Int -> BTree Int -> BTree Int
```

Given `balance l x r`, we have the following cases;

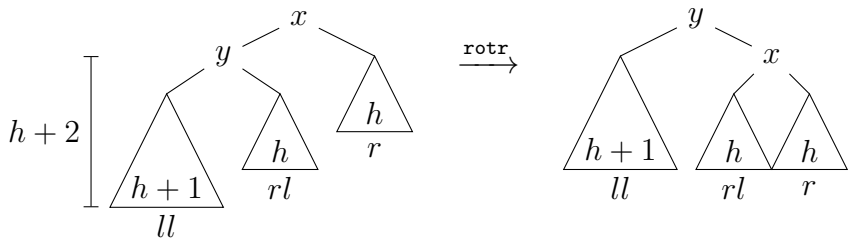
1. trees differ by at most 1;
   (a) the tree is balanced



   (b) the tree is off by 1 (showing only left tree being larger, without loss of generality)

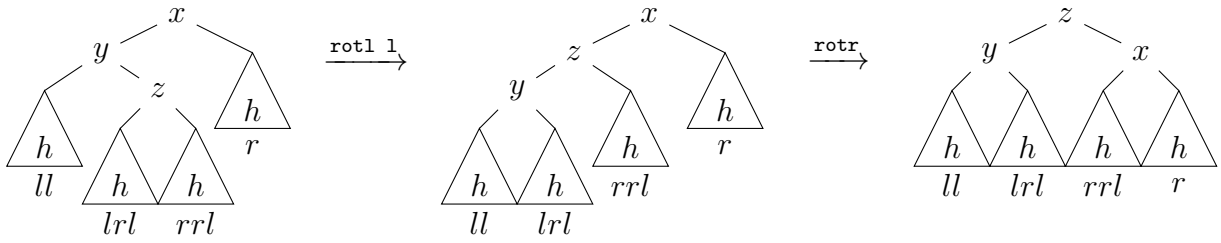


   Regardless, this is fine, hence `balance l x r = bnode l x r`
2. the trees are biased by 2;
   (a) assume that `height l` $>$ `height r`, and also `height height ll` $\geq$ `height rl` (where `rl` is the right subtree of the left subtree)



```
balance l x r = rotr (node l x r)

rotr :: BTree a -> BTree a
rotr (BNode _ (BNode _ ll y rl) x r)
  = bnode ll y (bnode rl x r)
```

(b) assume that `height l` > `height r`, and also `height height ll` < `height rl`;




```
balance l x r = rotr (node (rotl l) x r)

rotl :: BTree a -> BTree a
rotl (BNode _ ll y (BNode _ lrl z rrl))
  = bnode (bnode ll y lrl) z rrl
```

Putting all of this together;

```
balance :: BTree a -> a -> BTree a -> BTree a
balance l x r
  | abs b <= 1 = bnode l x r                          -- (case 1a, 1b)
  | b == 2     =
      if 0 <= bias l then rotr (node l x r)          -- (case 2a)
                     else rotr (node (rotl l) x r) -- (case 2b)
  | b == -2    =
      if bias r <= 0 then rotl (node l x r)          -- (symmetry of 2a)
                     else rotl (node l x (rotr r)) -- (symmetry of 2b)
```

Since `balance`, `rotr`, and `rotl` have no recursion, and are straight pattern matches, the process of balancing a tree is constant time.

However, we have recursion in `insert`. Since we have balanced the tree, we can therefore say that the size of the left and right subtrees are roughly equal, hence the height of the tree is approximately $\log_2 n$; and since we are doing a constant time operation at each level;

$$T_{\texttt{insert}}(n) \in O(\log n)$$

Similarly, since it is a balanced tree, looking up values is also $O(\log n)$.

By our definition of `fromList`, we are folding over a list of size n, and performing a $\log n$ operation for each, hence we have $O(n \log n)$.

Sorting with BTrees

Since the tree is ordered, we a natural sorting algorithm;

```
sort = toList .  fromList
```

Since we already know `fromList` is $O(n \log n)$, `toList` is the limiting factor - if we can get it to $O(n \log n)$ or better, the overall cost of the sort is $O(n \log n)$.

```
toList :: BTree a -> [a]
toList BNil = []
toList (Node _ l x r) = toList l ++ [x] ++ toList r
```

Assuming we made `toList` use $O(n)$ time (the implementation above takes $O(n^2)$, but we could've used a `DList`), we cannot get a better `fromList`, since it would lower the overall cost of `sort` to $O(n)$, which isn't possible due to a lower bound on sorting algorithms.

Recall that `DList` uses function composition, which gives preferable associativity. Therefore, we want the following;

```
toList l ++ [x] ++ toList r
          ↓
(toList' l .  (x:)  .  toList' r) []
```

Note that the function `toList'` takes a `BTree a`, and gives a **function** that takes a `[a]` and gives back `[a]`;

```
toList' :: BTree a -> ([a] -> [a])
toList' BNil xs = xs
toList' (BNode _ l x r) xs = (toList' l . (x:) . toList' r) xs
```

Therefore, we can write the following;

```
toList t = toList' t []
```

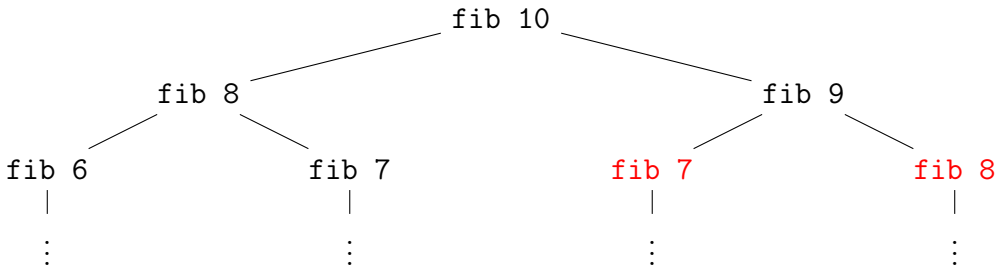
Hence our `toList` is $O(n)$.

November 5th 2019

Dynamic Programming

The idea behind dynamic programming is that we start with a recursive function, and repeated calls do not get executed again. We are trading space for speed. An example is calculating Fibonacci numbers (note that `Int` is bounded by the CPU, and `Integer` is larger and bounded by RAM);

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



However, notice that some values are recomputed, which wastes time. Ideally, we'd want to store the values we compute in a table, giving us constant time lookup.

```
fib' :: Int -> Integer
fib' n = table ! n
  where
    table :: Array Int Integer
    table = tabulate (0, n) mfib

    mfib 0 = 0
    mfib 1 = 1
    mfib n = table ! (n - 2) + table ! (n - 1)
```

Note that `(!)` is $O(1)$, and has the following type;

```
(!)  ::  Ix i => Array i e -> i -> e
```

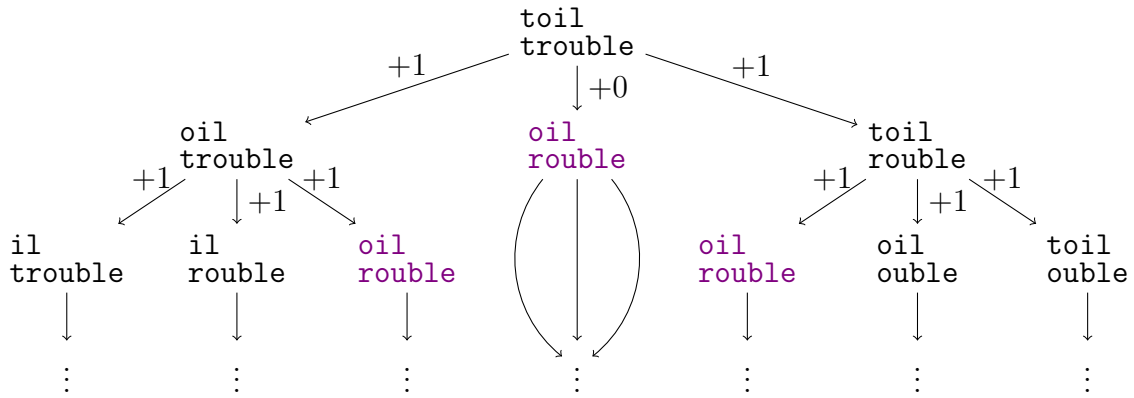
Memoisation is idea that if an argument has been computed the result can be retrieved immediately, but if it isn't, it needs to be computed and stored. Tabulation is a way to implement memoisation.

To construct an array we use the `array` function, which is used by `tabulate` (which also requires the `range` function);

```
range :: Ix i => (i, i) -> [i]
array :: Ix i => (i, i) -> [(i, e)] -> Array i e

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

Edit-Distance Problem

This works out how many times we need to update, delete, or insert a character to get from one string to another (the *Levenshtein* distance between two strings). We can better formalise this as a decision tree, where each node has 3 possible options. We can remove the first character from the first string, which costs 1, "remove" the first characters of both strings, which costs 0 if they are equal, and 1 otherwise, or remove the first character from the second string, which costs 1. At the end, we will have an empty string compared with something else - which has a cost of however long the other string is. The edit-distance is the **minimum** cost of all the paths to the root.



However, notice that the comparison between `oil` and `rouble` is done multiple times. Formally, we can write it as a recursive function;

```
dist :: String -> String -> Int
dist xs [] = length xs
dist [] ys = length ys
dist (x:xs) (y:ys) = minimum
  [dist xs (y:ys) + 1
  ,dist xs ys + if x == y then 0 else 1
  ,dist (x:xs) ys + 1]
```

However, notice that our current function uses two strings - and in order to construct a table indexing on a pair of strings, we'd need to enumerate an extremely large number of values. Since we are decomposing the values systematically, we can instead consider an integer.

```
dist' :: String -> String -> Int -> Int -> Int
dist' xs ys 0 j = j
dist' xs ys i 0 = i
dist' xs ys i j = minimum
  [dist' xs ys (i - 1) j + 1
  ,dist' xs ys (i - 1) (j - 1) + if x == y then 0 else 1
```

```
7     ,dist' xs ys i (j - 1) + 1]
8     where
9       x = xs !! i
10      y = ys !! j
```

Note that we are now checking the strings in reverse. This can be tabulated with a pair `(Int, Int)`;

```
1  dist'' :: String -> String -> Int
2  dist'' xs ys = table ! (m, n)
3    where
4      m = length xs
5      n = length ys
6
7      table = tabulate ((0, 0), (m, n)) mdist
8
9      mdist 0 j = j
10     mdist i 0 = i
11     mdist i j = minimum
12       [table ! ((i - 1), j) + 1
13       ,table ! ((i - 1), (j - 1)) + if  x == y then 0 else 1
14       ,table ! (i, (j - 1)) + 1]
15       where
16         x = xs !! i
17         y = ys !! j
```

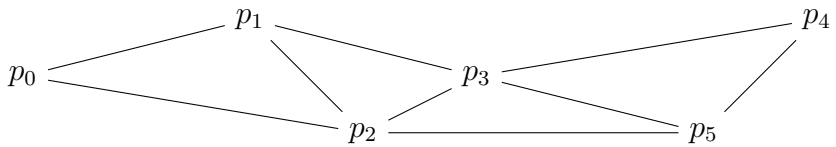
While `x = xs !! i` and `y = ys !! j` isn't constant time, we assume we can simply tabulate it to obtain constant time access, and ignore it. The original version was $O(3^{m+n})$, and this has been reduced to $O(mn)$.

November 8th 2019

Note that this lecture is 3 hours long, with the first hour being in **CO220**. It starts with a remark that the examples of dynamic programming we've looked at so far only gives us the answer, and not how the answer was reached.

The Travelling Salesman Problem

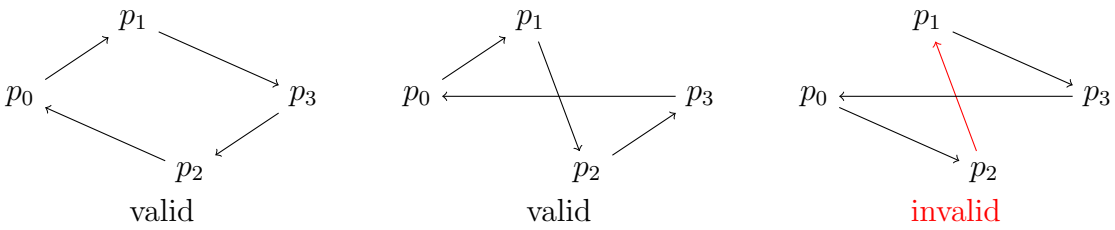
The travelling salesman problem is notoriously difficult to solve efficiently. Given $p_0, \ldots, p_n$, the goal is to find the shortest path in a graph that visits all these nodes and returns to the start.



Bitonic Travelling Salesman

The bitonic version, abbreviated to BTSP, has nodes on a plane instead. We then make the following assumptions;

1. every node is connected to every other node, and the distance (cost) is simply the Euclidean distance
2. no two points exist on top of each other (such that no two points have the same x co-ordinate, if we looked at the xy plane)
3. we have a bitonic path - this means that it starts from the left-most node and goes to the right-most node, and then back again (see the examples below)



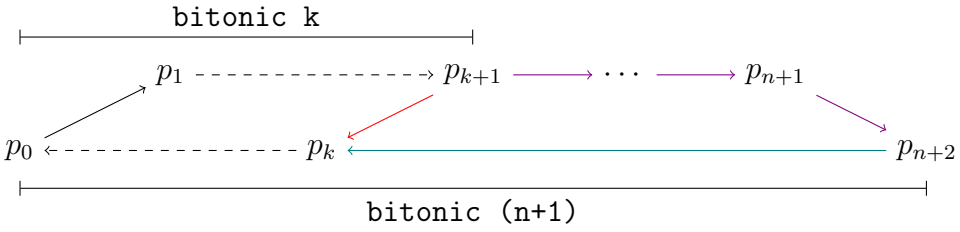
Note that the left-most node must be connected to the second left-most node, and the right-most node must be connected to the second right-most node.

This can be solved using dynamic programming in $O(m^2)$ time, for m nodes. Note that for the cases below, `bitonic i` means the shortest path up to `i+1`, and we write $\overline{p_i p_j}$ to denote the (Euclidean) distance between p_i and p_j;

- **base case** $\texttt{bitonic 0} = 2 \cdot \overline{p_0 p_1}$

- **recursive case**

  Let `bitonic i` be the shortest bitonic path for some value i. We can also assume that we've calculated `bitonic i` for all $i \leq n$ (strong induction).



  p_{n+2} must connect to p_{n+1}, and also some p_k. Bitonicity means that $p_{k+1} \to \cdots \to p_{n+1}$ must be connected. Therefore, we can conclude;

$$\texttt{bitonic (n+1)} = \texttt{bitonic k} + \overline{p_{k+1} \cdots p_{n+2}} - \overline{p_{k+1} p_k} + \overline{p_{n+2} p_k}$$

  As a side note, if there was a better path that uses some point after p_{k+1}, that would be picked up by taking the minimum value for all `k`s.

We must now find this k, but it must be minimal (assume `d :: Int -> Int -> Double` gives us the distance between two points, such that `d i j` $= \overline{p_i p_j}$);

```
1  import Data.Array
2  import Data.List
3
4  type Point = (Double, Double)
5
6  ps :: Array Int Point
7  ps = array (0, 3) (zip [0 .. ] [(0,0), (1,2), (2,3), (3,3)])
8
9  dist :: Array Int Point -> Int -> Int -> Double
10 dist ps i j = sqrt (x*x + y*y)
11   where
12     (xi, yi) = ps ! i
13     (xj, yj) = ps ! j
```

```
    x = xi - xj
    y = yi - yj

bitonic :: (Int -> Int -> Double) -> Int -> Double
bitonic d 0 = 2 * d 0 1
bitonic d n =
  minimum [ bitonic d k - d k (k + 1) + d k (n + 1)
          + sum [ d (i) (i + 1) | i <- [k + 1 .. n]]]
          | k <- [0 .. n - 1]]

-- returns edges as pairs of points
bitonic' :: (Int -> Int -> Double) -> Int -> (Double, [(Int, Int)]
bitonic' d 0 = (2 * d 0 1, [(0, 1)])
bitonic' d n =
  minimum [ bitonic' d k ~- (k, k + 1) ~+ (k, n + 1)
          ~+~ ssum [(d i (i + 1), [(i, (i + 1)])) | i <- [k + 1 .. n]]
          | k <- [0 .. n - 1]]
  where
   (~-) :: (Double, [(Int, Int)]) -> (Int, Int) -> (Double [(Int, Int)])
   (x, ps) ~- (i, j) = (x - d i j, ps \\ [(i, j), (j, i)])

   (~+) :: (Double, [(Int, Int)]) -> (Int, Int) -> (Double [(Int, Int)])
   (x, ps) ~+ (i, j) = (x + d i j, (i, j):ps)

   (~+~) :: (Double, [(Int, Int)]) -> (Double, [(Int, Int)]) -> (Double, [(Int,
      Int)])
   (x, ps) ~+~ (y, qs) = (x + y, ps ++ qs)

   ssum :: [(Double, [(Int, Int)])] -> (Double, [(Int, Int)])
   ssum = foldr1 (~+~)
```

Note that the "strange" operators are used to perform operations on the pairs.

12th November 2019

Introduction to Red-Black Trees

These trees have the same asymptotic complexity as binary search (AVL) trees, which is $O(\log n)$ for access, and insertion. In terms of pragmatics, binary trees had balancing operations after every insertion, which was constant, but still a large overhead. Red-Black trees have a cheaper rebalancing, but the trees become more lopsided (but has more expensive access). In addition, we had to keep the height of the tree with binary trees (an integer), but Red-Black trees have a single bit.

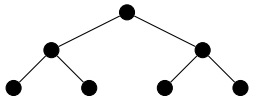
Assuming an empty node is a black node, RB trees have the following invariants;

1. every red node has a black parent
2. every path from the root to a leaf must have the same number of black nodes

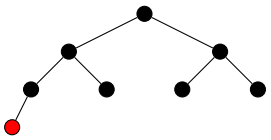
For invariant 2, if we had no red nodes, we will have a perfectly balanced tree. Due to invariant 1, the most lopsided a tree can become is if we alternated black, red, black, red, and so on, as it must still have the same number of black nodes.

Some examples are as follows;

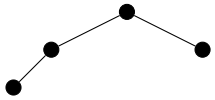
- **valid RB tree** perfectly balanced tree, with no red nodes



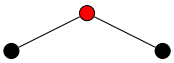
- **valid RB tree** same number of black nodes in each path



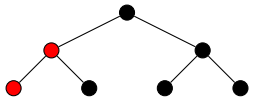
- **invalid RB tree** invariant 2 does not hold



- **invalid RB tree** invariant 1 does not hold



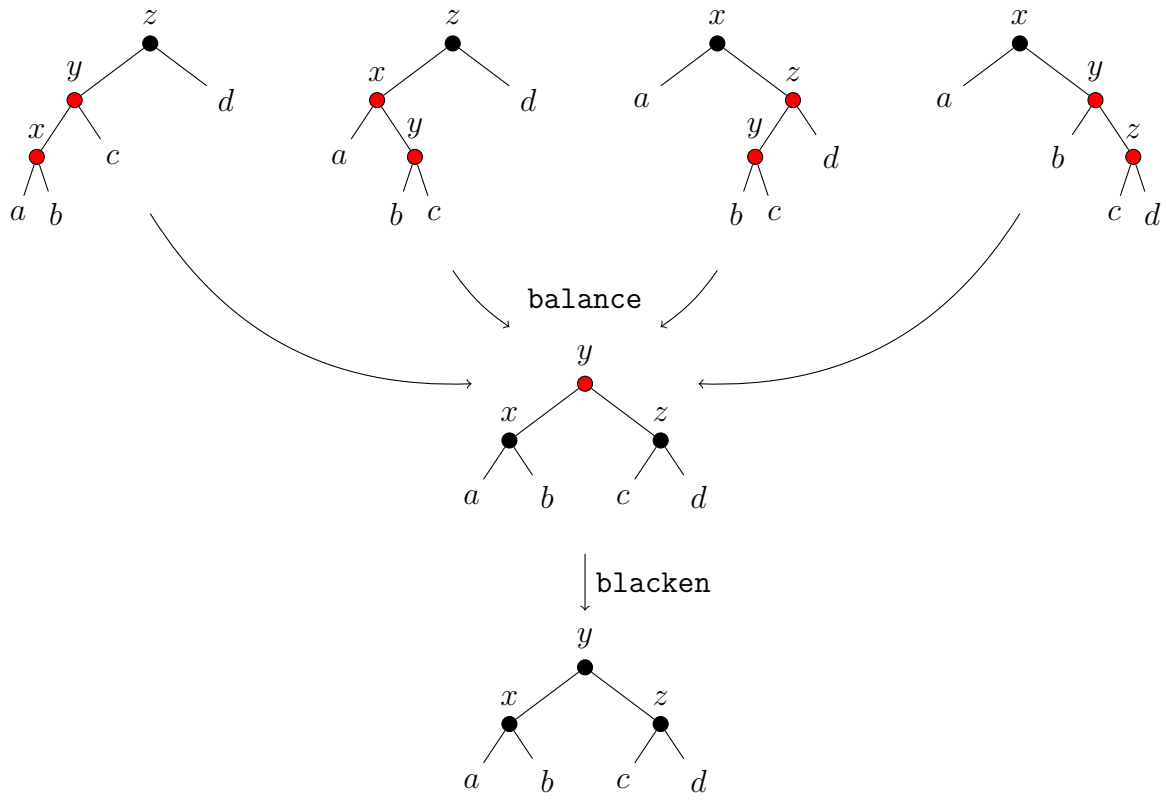
- **invalid RB tree** neither invariants hold



Similarly, we are also ensuring that elements in the left subtree are smaller, and elements in the right subtree are larger. The implementation is as follows;

```
data Colour = R | B                                    -- (R)ed | (B)lack
data RBTree a = E | N Colour (RBTree a) a (RBTree a) -- (E)mpty | N(ode)

member :: Ord a => a -> RBTree a -> Bool
member x E = False
member x (N _ l y r)
  | x < y  = member x l
  | x == y = True
  | x > y  = member x r

insert :: Ord a => a -> RBTree a -> RBTree a
insert x t = blacken (insx t)
  where
    insx E = N R E x E
    insx (N c l y r)
      | x < y  = balance c (insx l) y r
      | x == y = N c l y r
      | x > y  = balance c l y (insx r)

-- ensures invariant 1 holds
blacken :: RBTree a -> RBTree a
blacken E = E
blacken (N _ l y r) = N B l y r
```

For `balance`, there are 4 possible cases (we can assume that the subtrees, a, b, c, d, have already been fixed due to recursion);



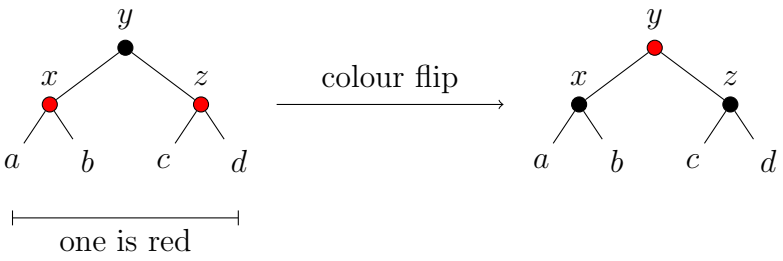
As shown above, `balance` is a simple pattern match (each `B` case corresponds to a case above);

```
balance :: Colour -> RBTree a -> a -> RBTree a -> RBTree a
balance B (N R (N R a x b) y c) z d = t
balance B (N R a x (N R b y c)) z d = t
balance B a x (N R (N R b y c) z d) = t
balance B a x (N R b y (N R c z d)) = t
-- case without a black root
balance c l x r = N c l x r
  where -- not valid Haskell, can't span across multiple patterns
    t = N R (N B a x b) y (N B c z d)
```

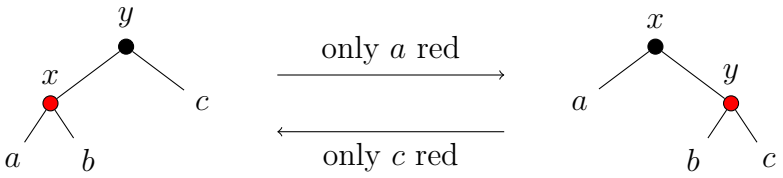
Persistent vs Mutable Data Structures

Note that with a persistent data structure, the code above will take in a "pointer" to the root of the tree, and return a **new** pointer, which is desired. This is useful in the case where we have multiple threads running, as we don't want the data structure to change unexpectedly. On the other hand, a mutable data structure modifies the tree, but has more efficient transformations (and is more complex in terms of cases).

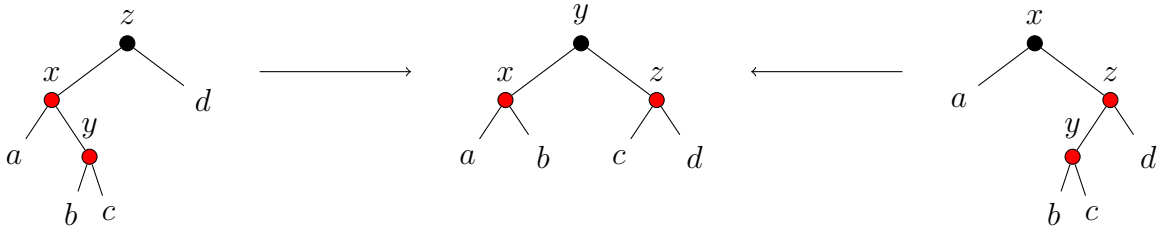
- **colour flip**



- **rotation**



- **double rotation**



In an imperative setting, for the colour flip, only three values need to be changed (a, b, c, d no longer cause a red-red conflict, as the parents are now black). Note that the transformations can be aborted early, unlike in the persistent setting, since only parts of the tree need to be modified.

November 15th 2019

Note that this lecture has no sound.

Construction of Red-Black Tree

Constructing this tree can be done with repeated addition, similar to before;

```
1 fromList :: Ord a => [a] -> RBTree a
2 fromList = foldr insert E
```

Applying it to the list `[1..8]`, we get the following;