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

$$
\begin{aligned}
T(0) &= 1 && \text{1 transition} \\
T(n) &= 1 + T(n-1) && \text{looking at worst case, line 5}
\end{aligned}
$$

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).

```
isort :: [Int] -> [Int]
isort [] = []
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]`	
$\leadsto$ `insert 3 (isort [1,2])`	definition of `isort`

$\leadsto$ `insert 3 (insert 1 (isort [2]))` definition of `isort`
$\leadsto$ `insert 3 (insert 1 (insert 2 (isort [])))` definition of `isort`
$\leadsto$ `insert 3 (insert 1 (insert 2 []))` definition of `isort`
$\leadsto$ `insert 3 (insert 1 [2])` definition of `insert`
$\leadsto$ `insert 3 (1:2:[])` definition of `insert`
$\leadsto$ `1:insert 3 (2:[])` definition of `insert`
$\leadsto$ `1:2:(insert 3 [])` definition of `insert`
$\leadsto$ `1:2:[3]` definition of `insert`

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

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

However, we want to find this in closed form;

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

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.

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

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

`minimum [1,2,3]`
$\leadsto$ `head (sort [1,2,3])`
$\leadsto$ `...`
$\leadsto$ `head (insert 1 (insert 2 (insert 3 [])))`
$\leadsto$ `head (insert 1 (insert 2 [3]))`
$\leadsto$ `head (insert 1 (2:[3]))`
$\leadsto$ `head 1:2:[3]`
$\leadsto$ `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.

$$
\begin{aligned}
k^T &= 0 && \text{constants} \\
x^T &= 0 && \text{evaluated variables} \\
(f\ e_1\ \dots\ e_n)^T &= (f^T\ e_1\ \dots\ e_n) + e_1^T + \dots + e_n^T && \text{function with arguments} \\
(\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2)^T &= p^T + (\texttt{if}\ p\ \texttt{then}\ e_1^T\ \texttt{else}\ e_2^T) && \text{conditional} \\
\left(\texttt{case}\ e\ \texttt{of}\ \begin{cases} p_1 & \to e_1 \\ & \vdots \\ p_n & \to e_n \end{cases}\right)^T &= e^T + \left(\texttt{case}\ e\ \texttt{of}\ \begin{cases} p_1 & \to e_1^T \\ & \vdots \\ p_n & \to e_n^T \end{cases}\right) && \text{pattern matching}
\end{aligned}
$$

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.

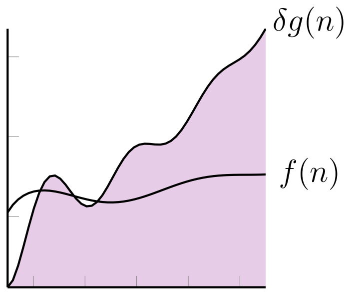
Consider $f(n) = 2n$, and $g(n) = \frac{n^2}{4}$ - at $n = 1$, $f(1) > g(1)$, however at some point on the number line, g begins to dominate. Comparing functions can be achieved by studying their ratios (with well-behaved functions, like L-functions, the ratio will tend to 0, infinity, or a constant);

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

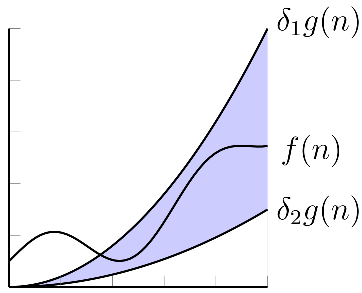
Any L-function is ultimately continuous of constant sign, monotonic, and approaches 0, ∞, or some definite limit as $n \to \infty$. Furthermore, $\frac{f}{g}$ is an L-function if both f and g are. We can now introduce notation compare function;

$$
\begin{aligned}
f \prec g &\triangleq \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 && \text{also written as } f \in o(g(n)) \\
f \preceq g &\triangleq \limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty && \text{also written as } f \in O(g(n)) \\
f \asymp g &\triangleq f \in (O(g(n)) \cap \Omega(g(n))) && \text{also written as } f \in \Theta(g(n)) \\
f \succeq g &\triangleq \limsup_{n \to \infty} \left|\frac{f(n)}{g(n)}\right| > 0 && \text{also written as } f \in \Omega(g(n)) \\
f \succ g &\triangleq \lim_{n \to \infty} \left|\frac{f(n)}{g(n)}\right| = \infty && \text{also written as } f \in \omega(g(n))
\end{aligned}
$$

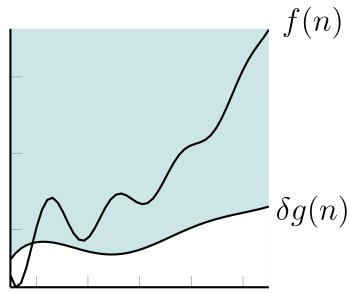
Visually, we can represent this in the following three graphs. Note that $\delta, \delta_1, \delta_2$ are just constant multipliers. The first plot shows that as n gets larger $f(n)$ will exist within the shaded region bounded above by $\delta g(n)$, and similarly (on the other extreme) the third plot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



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



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

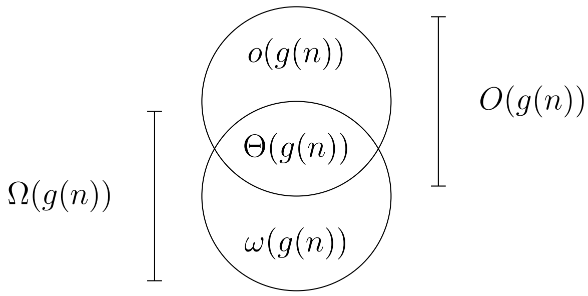


$f(n) \in \Omega(g(n))$
$f \succeq g$

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;

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