# 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;

$$\begin{aligned}
&\texttt{insert 4 [1,3,6,7,9]} && \\
\rightsquigarrow\ &\texttt{1:insert 4 [3,6,7,9]} && \text{definition of } \texttt{insert} \\
\rightsquigarrow\ &\texttt{1:3:insert 4 [6,7,9]} && \text{definition of } \texttt{insert} \\
\rightsquigarrow\ &\texttt{1:3:4:6:[7,9]} && \text{definition of } \texttt{insert}
\end{aligned}$$

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;

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

### 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;

$$\begin{aligned}
&\texttt{isort [3,1,2]} && \\
\rightsquigarrow\ &\texttt{insert 3 (isort [1,2])} && \text{definition of } \texttt{isort}
\end{aligned}$$

| | |
|---|---|
| $\leadsto$ `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