Lecture 6 Asymptotic Analysis

Definition Revisit

Suppose f and g are two positive-valued mathematical functions defined on (at least) the natural numbers. We say that $f(n)$ is $O(g(n))$ if there exists constant $c \in \mathbb{N}$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.

Consider $f(n) = n^2$ and $g(n) = n^2 + n + 3$. $f(n) = O(g(n))$ and $g(n) = O(f(n))$. Or, f and g have the same asymptotic complexity (quadratic).

Analysis of Two Functions

Consider our function `append` and `rev` before.

```
fun @ (nil : int list, L2 : int list) : int list = L2
  | @ (x::xs, L2) = x :: (@ (xs, L2))
infixr @

fun rev (nil : int list) : int list = nil
  | rev (x::xs) = (rev xs) @ [x]
```

The `append` Function

So `@` has two arguments. Consider `n` length of first list, `m` length of second list.

Analyze "work" of the function: $W(n, m)$.

For `n = 0`, $W(0, m) = c_0$, which is a constant.

For `n >= 1`, $W(n, m) = c_1 + W(n - 1, m)$, where c_1 is the constant time of `cons`.

> Note: This is not yet a solution. It's just a recursive analysis of the recurrent work.

An informal argument for the proof:

```
W(n, m) = c1 + W(n - 1, m)
        = c1 + c1 + W(n - 2, m)
        = c1 + c1 + c1 + W(n - 3, m)
        = ...
        = c1 +...+ c1 + c0
```

This is called "unrolling the recurrence".

By observation, we can **conjecture** the complexity is linear.

Then, we can prove this formally by **induction**.

The `rev` Function

Now, let's analyze the function `rev`.

`rev` has one argument. Let `n` be the length of the list.

For `n = 0`, $W(0) = c_0$, which is a constant.

For `n >= 1`, $W(n) = c_1 + W_{@}(n - 1, 1) + W(n - 1)$.

> Note: In order to determine the arguments for the complexity of `append` used here, we have to know that function `rev` does not change the length of the list input.

We know the complexity of the function `append`, so we can substitute and continue our analysis:

```
W(n) <= k0 + k1(n - 1) + W(n - 1)
     <= k0' + k1'(n) + W(n - 1)
     ::
     <= k0' + k1'(n) + k0' + k1'(n - 1) + W(n - 2)
     <= k0' + k1'(n) + k0' + k1'(n - 1) + k0' + k1'(n - 2) + W(n - 3)
     <= ...
     == (n)(k0') + (k1')(1 +...+ n) + k0
```

Thus, we **conjecture** this is of order n^2.

The Tail Recursive `rev`

Now, let's analyze the *tail recursive* version of the function.

```
fun trev (nil : int list, acc : int list) : int list = acc
  | trev (x::xs, acc) = trev(xs, x::acc)
```

Consider n the length of first list, m the length of the accumulator.

For `n = 0`, $W(0, m) = c_0$. We are just returning, **not** copying anything.

For `n >= 1`, $W(n, m) = c_1 + W(n-1, m+1)$.

> Note: It's important to think carefully about the **size** of the arguments.

By observing the recurrent structure, we **conjecture** this is again of linear time.

Analysis of Trees

Consider this definition of the datatype `tree`:

```
datatype tree = Empty | Node of tree * int * tree

(* sum : tree -> int *)
fun sum (Empty : tree) : int = 0
  | sum (Node (left, x, right) = sum left + x + sum right
```

Let's consider the complexity of the function `sum`.

Consider "work" in terms of the size of the tree n.

> Note: Sometimes we consider "work" or "span" in terms of the **depth** of the tree.

> Important: The size n here refers to the number of **nodes**. Sometimes we might want it different.

For `n = 0`, $W(0) = c_0$, a constant, for an `Empty` tree.

For a non-`Empty` tree, $W(n) = c_1 + W(n_{left}) + W(n_{right})$.

We also know that $n_{left} + n_{right} = n$.

Conjecture: $W(n) \leq k_1 + k_2 \cdot n$.

> Consider: In fact, considering the work done in each `Node`, should be all c_1.

Analysis of the "Span"

Consider still the computation of the tree above.

$S(0) = c_0$.

$S(n) = c_1 + max(S(n_{left}), S(n_{right}))$.

Suppose the tree is balanced, $S(n) \approx c_1 + max(S(n/2), S(n/2))$.

Consider the "span" in terms of **depth**:

$S(0) = c_0$.

$S(d) = c_1 + max(S(d-1), S(d-1))$.

So, this concludes that $S(d)$ is of $O(d)$, i.e. $O(logn)$. (This is for a balanced tree.)

Thus, we conclude that what really matters is the **depth** of the tree (for parallel).

> Side Note: The SML implementation we have now is running **sequentially**.

Sorting and Analysis of Sorting

Datatype `order` in ML

```
datatype order = LESS | EQUAL | GREATER
Int.compare : int * int -> order
(* This gives a three-way result. All pre-defined. *)
```

Definition of "Being Sorted"

Definition. A list of integers `L` is sorted if-and-only-if for every `x` and `y` in `L`, if `x` appears to the left of `y` in `L`, then $x \leq y$.

> Side Note: We can define in terms of indices, but we don't want to.

Analysis of a Sort

```
(* ins : int list * int -> int list
 * REQUIRES: L is sorted
 * ENSURES: ins(L, x) is sorted list consisting of all elements in L plus x
 *)
fun ins ([] : int list, x : int) : int list = [x]
  | ins (y::ys, x) =
    case Int.compare(x, y)
      of GREATER => y::(ins(ys, x))
       | _ => x::y::ys

(* isort : int list -> int list *)
fun isort ([] : int list) : int list = []
  | isort (x::xs) = ins(isort xs, x)      (* recursively sort the rest, then insert x *)
```

The work of function `ins` is linear, and the work of `isort` is quadratic.