15-150

Principles of Functional Programming

Slides for Lecture 6

Asymptotic Cost Analysis

February 1, 2024

Michael Erdmann

Asymptotic Cost Analysis

- Big-O complexity classes
- Recurrence Relations
- Work and Span
- Application: Sorting

Big-O Complexity Classes

Suppose $f(n)$ and $g(n)$ are positive-valued mathematical functions (with n a natural number).

We say that "$f(n)$ is $O(g(n))$" if there exist N and c such that

$$f(n) \le c*g(n) \text{ for all } n \ge N.$$

Big-O Complexity Classes

Suppose $f(n)$ and $g(n)$ are positive-valued mathematical functions (with n a natural number).

We say that "$f(n)$ is $O(g(n))$" if there exist N and c such that

$$f(n) \leq c*g(n) \text{ for all } n \geq N.$$

$n^2 + n + 3$ is $O(n^2)$ for instance.

(use $N=3$ and $c=2$)

(e.g., $7^2 + 7 + 3 \leq 2*7^2$)

Big-O Complexity Classes

Suppose $f(n)$ and $g(n)$ are positive-valued mathematical functions (with n a natural number).

We say that "$f(n)$ is $O(g(n))$" if there exist N and c such that

$$f(n) \leq c * g(n) \text{ for all } n \geq N.$$

$n^2 + n + 3$ is $O(n^2)$ for instance.

for this example also n^2 is $O(n^2 + n + 3)$

Big-O Complexity Classes

Suppose $f(n)$ and $g(n)$ are positive-valued mathematical functions (with n a natural number).

We will let f measure work or span in terms of some size parameter n (sometimes tree depth d) and obtain complexity classes

$O(1)$, $O(n)$, $O(n^2)$, $O(n^3)$, ...,

$O(\log n)$, $O(n \cdot \log n)$, $O(2^n)$, ...

Analyzing **append** and **rev**

```
(* op @ : int list * int list -> int list *)
infixr @
fun [] @ Y = Y
  | (x::xs) @ Y = x::(xs @ Y)

(* rev : int list -> int list
   REQUIRES: true
   ENSURES:  rev(L) returns a list consisting
             of L's elements in reverse order.
*)
fun rev [] = []
  | rev (x::xs) = (rev xs) @ [x]
```

Code for **append**:

```
fun [] @ Y = Y
  | (x::xs) @ Y = x::(xs @ Y)
```

Work analysis of **append**:

$W_{@}(n,m)$ with n and m the sizes of the input lists.

Equation for base case:

$W_{@}(0,m) = c_0$, for some c_0, all m.

Equation for recursive clause, for $n > 0$:

$W_{@}(n,m) = c_1 + W_{@}(n-1,m)$, for some c_1, all m.

Solving: $\mathbf{W_@(0,m) = c_0}$

$$\mathbf{W_@(n,m) = c_1 + W_@(n-1,m)}$$

Unrolling:

$$\mathbf{W_@(n,m) = c_1 + \overline{c_1 + W_@(n-2,m)}}$$

Solving: $W_@(0,m) = c_0$

$W_@(n,m) = c_1 + W_@(n-1,m)$

Unrolling:

$$W_@(n,m) = c_1 + c_1 + W_@(n-2,m)$$

$$= c_1 + c_1 + c_1 + W_@(n-3,m)$$

Solving: $W_@(0,m) = c_0$

$W_@(n,m) = c_1 + W_@(n-1,m)$

Unrolling:

$$W_@(n,m) = c_1 + c_1 + W_@(n-2,m)$$

$$= c_1 + c_1 + c_1 + W_@(n-3,m)$$

$$\ldots \quad = n \cdot c_1 + c_0$$

(can prove this by induction)

So evaluation of **(X @ Y)** has **O(n)** work,
with **n** the length of **X**.

Code for **rev**:

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

Work analysis of **rev**:

$W_{rev}(n)$ with **n** the size of the input list.

Equation for base case:

$W_{rev}(0) = c_0$, for some c_0.

Equation for recursive clause, for $n > 0$:

$W_{rev}(n) = c_1 + W_{rev}(n-1) + W_{@}(n-1,1)$, some c_1.

Why?

(use a little lemma that tells us)

For all list values **`L`**,

$$\texttt{length (rev L)} \cong \texttt{length L}$$

Code for `rev`:

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

Work analysis of `rev`:

$W_{rev}(n)$ with **n** the size of the input list.

Equation for base case:

$W_{rev}(0) = c_0$, for some c_0.

Equation for recursive clause, for $n > 0$:

$W_{rev}(n) = c_1 + W_{rev}(n-1) + W_{@}(n-1, 1)$, some c_1.

So:

$W_{rev}(n) \le c_1 + W_{rev}(n-1) + c_2(n-1)$, some c_2.

Solving: $W_{rev}(0) = c_0$

$$W_{rev}(n) \le c_1 + W_{rev}(n-1) + c_2(n-1)$$

$$W_{rev}(n) \le c_1 + c_2 \cdot n + W_{rev}(n-1)$$

Unrolling:

$$W_{rev}(n) \le c_1 + c_2 \cdot n + \overline{\left\{ c_1 + c_2(n-1) + W_{rev}(n-2) \right\}}$$

Solving:

$$W_{rev}(0) = c_0$$

$$W_{rev}(n) \le c_1 + W_{rev}(n-1) + c_2(n-1)$$

$$W_{rev}(n) \le c_1 + c_2 \cdot n + W_{rev}(n-1)$$

Unrolling:

$$W_{rev}(n) \le c_1 + c_2 \cdot n + \{c_1 + c_2(n-1) + W_{rev}(n-2)\}$$

$$\le c_1 + c_2 \cdot n + c_1 + c_2(n-1)$$

$$+ \{c_1 + c_2(n-2) + W_{rev}(n-3)\}$$

Solving: $W_{rev}(0) = c_0$

$$W_{rev}(n) \le c_1 + W_{rev}(n-1) + c_2(n-1)$$

$$W_{rev}(n) \le c_1 + c_2 \cdot n + W_{rev}(n-1)$$

Unrolling:

$$W_{rev}(n) \le c_1 + c_2 \cdot n + \{c_1 + c_2(n-1) + W_{rev}(n-2)\}$$

$$\le c_1 + c_2 \cdot n + c_1 + c_2(n-1)$$

$$+ \{c_1 + c_2(n-2) + W_{rev}(n-3)\}$$

$$\ldots \le c_0 + n \cdot c_1 + (n(n+1)/2) \cdot c_2$$

Solving: $\mathbf{W_{rev}(0) = c_0}$

$$\mathbf{W_{rev}(n) \le c_1 + W_{rev}(n-1) + c_2(n-1)}$$

$$\mathbf{W_{rev}(n) \le c_1 + c_2 \cdot n + W_{rev}(n-1)}$$

Unrolling:

$\mathbf{W_{rev}(n) \le}$

$$\mathbf{\ldots}$$

$$\mathbf{\le c_0 + n \cdot c_1 + \big(n(n+1)/2\big) \cdot c_2}$$

So evaluation of **`rev(L)`** has $O(n^2)$ work, with **`n`** the length of **`L`**.

Analyzing **trev**

```
(* trev : int list * int list -> int list *)

fun trev ([], acc) = acc
  | trev (x::xs, acc) = trev(xs, x::acc)
```

Code for **trev**:

```
fun trev ([], acc) = acc
  | trev (x::xs, acc) = trev(xs, x::acc)
```

Work analysis of **trev**:

$W_{trev}(n,m)$ with n and m the sizes of the input lists.

Equation for base case:

$W_{trev}(0,m) = c_0$, for some c_0, all m.

Equation for recursive clause, for $n > 0$:

$W_{trev}(n,m) = c_1 + W_{trev}(n-1,m+1)$, some c_1, all m.

Unrolling:

$$W_{trev}(n,m) = c_1 + c_1 + W_{trev}(n-2,m+2)$$
$$\ldots = n \cdot c_1 + c_0, \text{ which is } O(n).$$

Analyzing tree summation

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

(* sum : tree -> int  *)
   REQUIRES: true
   ENSURES:  sum(T) adds all integers in T.
*)

fun sum(Empty : tree) : int = 0
  | sum(Node(ℓ,x,r)) = (sum ℓ) + (sum r) + x
```

Code for sum:

```
fun sum Empty = 0
  | sum (Node(ℓ,x,r)) = (sum ℓ) + (sum r) + x
```

Work analysis of sum:

$W_{sum}(n)$ with n the number of nodes in the tree.

Equation for base case:

$W_{sum}(0) = c_0$, for some c_0.

Equation for recursive clause, for $n > 0$:

$W_{sum}(n) = c_1 + W_{sum}(n_\ell) + W_{sum}(n_r)$, some c_1,

with now n_ℓ the number of nodes in the left subtree

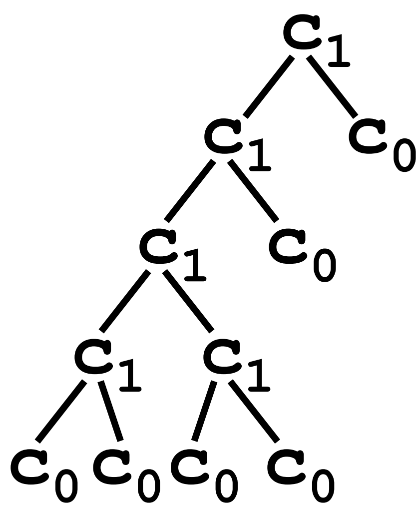
and n_r the number of nodes in the right subtree.

<u>Solving:</u>

$$\mathbf{W_{sum}(0) = c_0}$$

$$\mathbf{W_{sum}(n) = c_1 + W_{sum}(n_\ell) + W_{sum}(n_r)}$$

<u>Tree Method:</u> (write down work that occurs at each node/leaf)



Note: Tree need not be balanced.

$$\mathbf{W_{sum}(n) = c_1 n + c_0(n+1)}$$

Fact: A binary tree has **n** nodes iff it has **n+1** leaves.

So evaluation of **sum(T)** has **O(n)** work.

(can also prove this by induction)

Side remark for the curious student

The fact that a binary tree has `n` nodes iff it has `n+1` leaves is a special instance of the **Euler Characteristic**.

A slightly more general instance:

In an undirected graph:

#vertices - #edges = #components - #cycles

Code for **`sum`**:

```
fun sum Empty = 0
  | sum (Node(ℓ,x,r)) = (sum ℓ) + (sum r) + x
```

Is there any opportunity for parallelism?

YES: The recursive calls to **`sum`** can occur in parallel.

Code for **sum**:

```
fun sum Empty = 0
  | sum (Node(ℓ,x,r)) = (sum ℓ) + (sum r) + x
```

Span analysis of **sum**:

$S_{sum}(n)$ with **n** the number of nodes in the tree.

Equation for base case:

$S_{sum}(0) = c_0$, for some c_0.

Equation for recursive clause, for $n > 0$:

$S_{sum}(n) = c_1 + \max\{S_{sum}(n_\ell), S_{sum}(n_r)\}$, some c_1.

Notice how **max** replaces **+** in the cost analysis.

Solving: $\mathbf{S_{sum}(0) = c_0}$

$$\mathbf{S_{sum}(n) = c_1 + max\{S_{sum}(n_\ell), S_{sum}(n_r)\}}$$

ALAS! It could be that $\mathbf{n_\ell = n-1}$ **and** $\mathbf{n_r = 0}$.

Then the recursive equation becomes:

$$\mathbf{S_{sum}(n) = c_1 + S_{sum}(n-1)}$$

Therefore $\mathbf{S_{sum}(n)}$ is $\mathbf{O(n)}$,

meaning we haven't gained anything from parallel evaluation.

Suppose however that the tree is *balanced*.

(This means that roughly half the remaining nodes appear in each subtree as one descends the tree.)

Then: $S_{sum}(0) = c_0$

$$S_{sum}(n) \approx c_1 + \max\{S_{sum}(n/2), S_{sum}(n/2)\}$$

Suppose however that the tree is *balanced*.

(This means that roughly half the remaining nodes appear in each subtree as one descends the tree.)

Then: $\mathbf{S_{sum}(0) = c_0}$

$$\mathbf{S_{sum}(n) = c_1 + max\{S_{sum}(n/2), S_{sum}(n/2)\}}$$

So

$$\begin{aligned} \mathbf{S_{sum}(n)} &= \mathbf{c_1 + S_{sum}(n/2)} \\ &= \mathbf{c_1 + c_1 + S_{sum}(n/4)} \\ \mathbf{...} \quad &= \underbrace{\mathbf{c_1 + c_1 + \cdots + c_1}}_{(\lfloor \log_2 n \rfloor + 1) \text{ many times}} \mathbf{+ c_0} \end{aligned}$$

Now $\mathbf{S_{sum}(n)}$ is $\mathbf{O(log(n))}$, meaning parallelism is significant.

We could also have obtained this result by expressing span as $\mathbf{S_{sum}(d)}$, with $\mathbf{d}$ the depth of the tree.

Then: $\mathbf{S_{sum}(0) = c_0}$

$$\mathbf{S_{sum}(d) = c_1 + max\{S_{sum}(d-1), S_{sum}(d')\}}$$

Note: $\mathbf{d' < d}$

We could also have obtained this result by expressing span as $\mathbf{S}_{\mathtt{sum}}(\mathbf{d})$, with $\mathbf{d}$ the depth of the tree.

Then: $\mathbf{S}_{\mathtt{sum}}(0) = \mathbf{c}_0$

$$\mathbf{S}_{\mathtt{sum}}(\mathbf{d}) = \mathbf{c}_1 + \mathbf{max}\{\mathbf{S}_{\mathtt{sum}}(\mathbf{d}-1), \mathbf{S}_{\mathtt{sum}}(\mathbf{d}')\}$$

So $\mathbf{S}_{\mathtt{sum}}(\mathbf{d}) = \mathbf{c}_1 + \mathbf{S}_{\mathtt{sum}}(\mathbf{d}-1)$

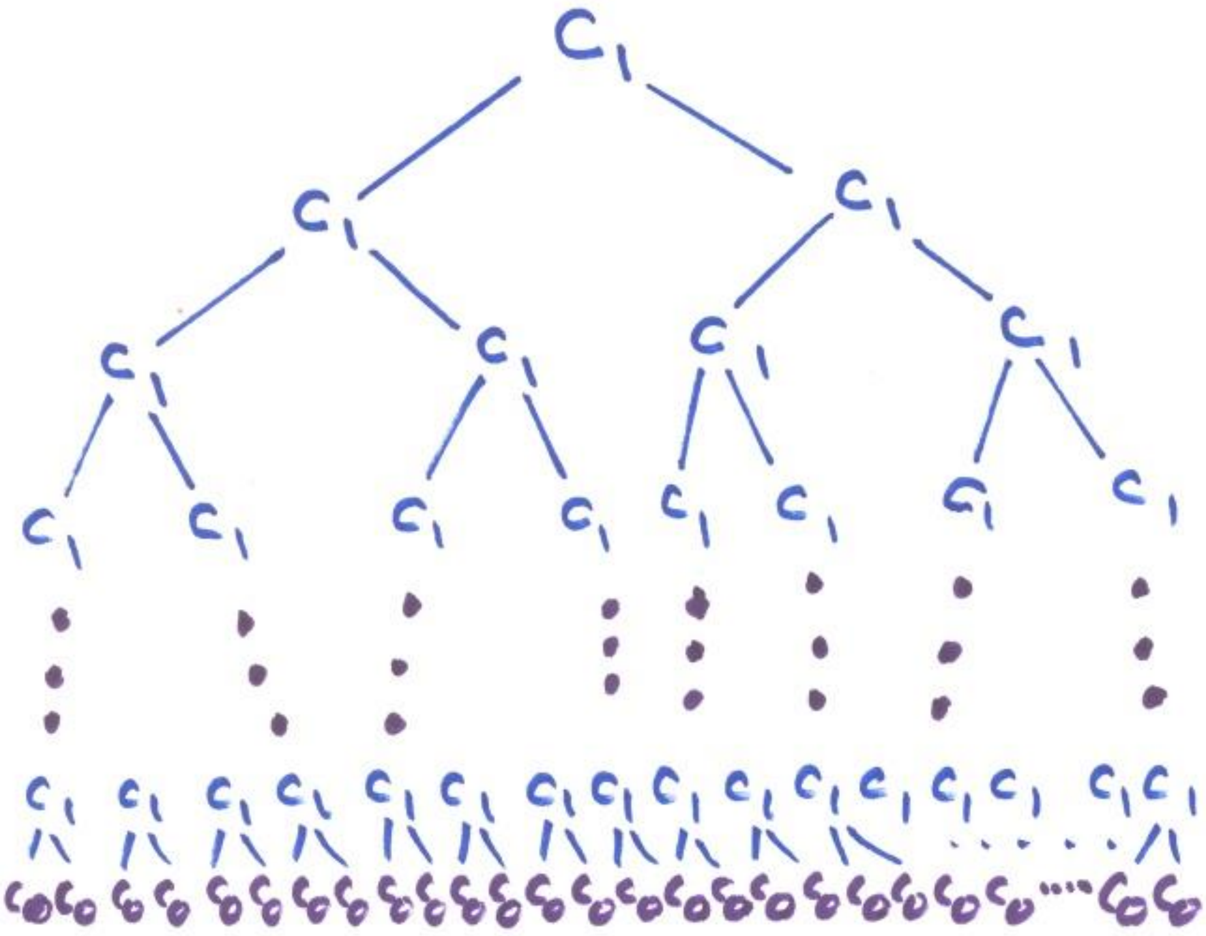
Thus $\mathbf{S}_{\mathtt{sum}}(\mathbf{d})$ is $\mathbf{O}(\mathbf{d})$.

This result holds for <u>all</u> trees. ($\mathbf{d}=\mathbf{n}$ is possible)

For balanced trees, $\mathbf{d}$ is $\mathbf{O}(\mathbf{log}(\mathbf{n}))$,

and we again see that parallelism helps.

Tree Method for balanced trees:



This tree is *perfectly balanced*.

We use it as a model for balanced trees more generally.

Tree Method for balanced trees:

Definition: A binary tree is *balanced* if it is either

(i) **`Empty`**

or (ii) a **`Node`** whose two subtrees are balanced with depths differing by at most **1**.

This tree is *perfectly balanced*.

We use it as a model for balanced trees more generally.

Tree Method for balanced trees:

More generally: A binary tree is *balanced* if it is either

(i) **`Empty`**

or (ii) a **`Node`** whose two subtrees are balanced

with depths differing by at most a constant **`c`**.

This tree is *perfectly balanced*.

We use it as a model for balanced trees more generally.

Tree Method for balanced trees:

Another definition (consequence of previous defs):

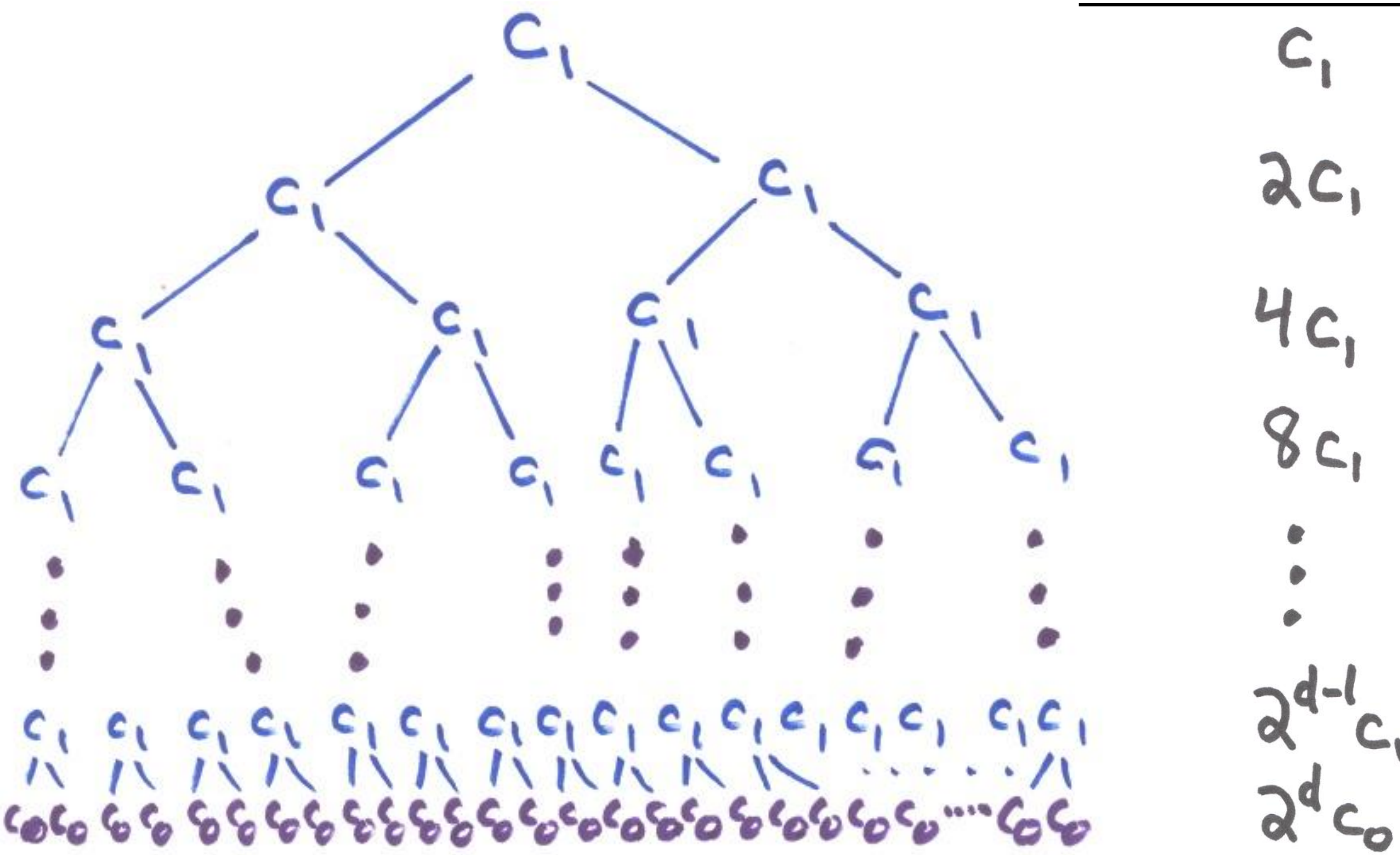
A binary tree is *balanced* if its depth **d** is roughly **log(n)**, with **n** the number of nodes in the tree.

This tree is *perfectly balanced*.

We use it as a model for balanced trees more generally.

Tree Method for balanced trees:

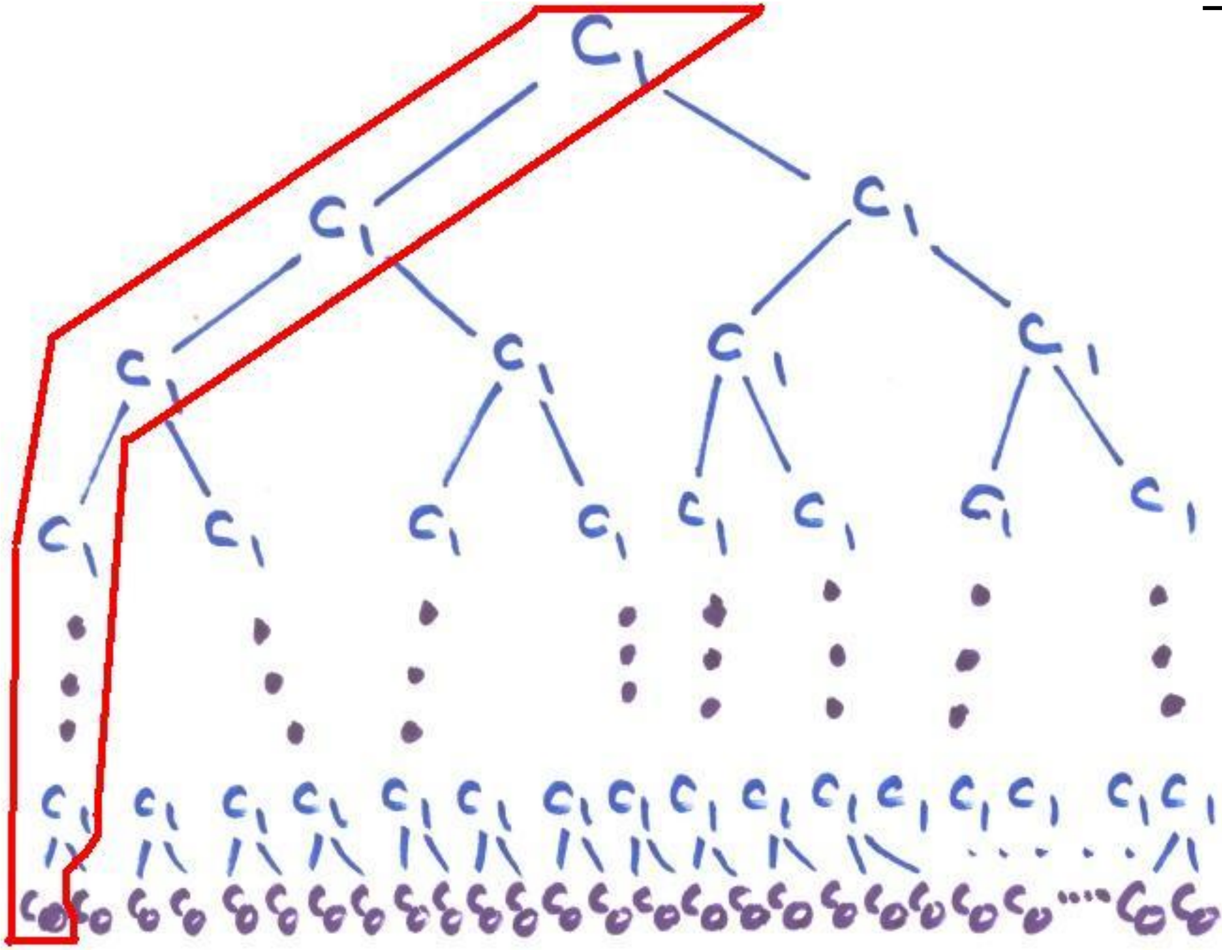
Work at each level



$$W(n) = c_1(1+2+\cdots+2^{d-1})+c_0 2^d \leq c2^{d+1}, \text{ so } O(n).$$

$(c = \max(c_1, c_0))$

Tree Method for balanced trees:



Work at each level

c_1

$2c_1$

$4c_1$

$8c_1$

$\vdots$

$2^{d-1}c_1$

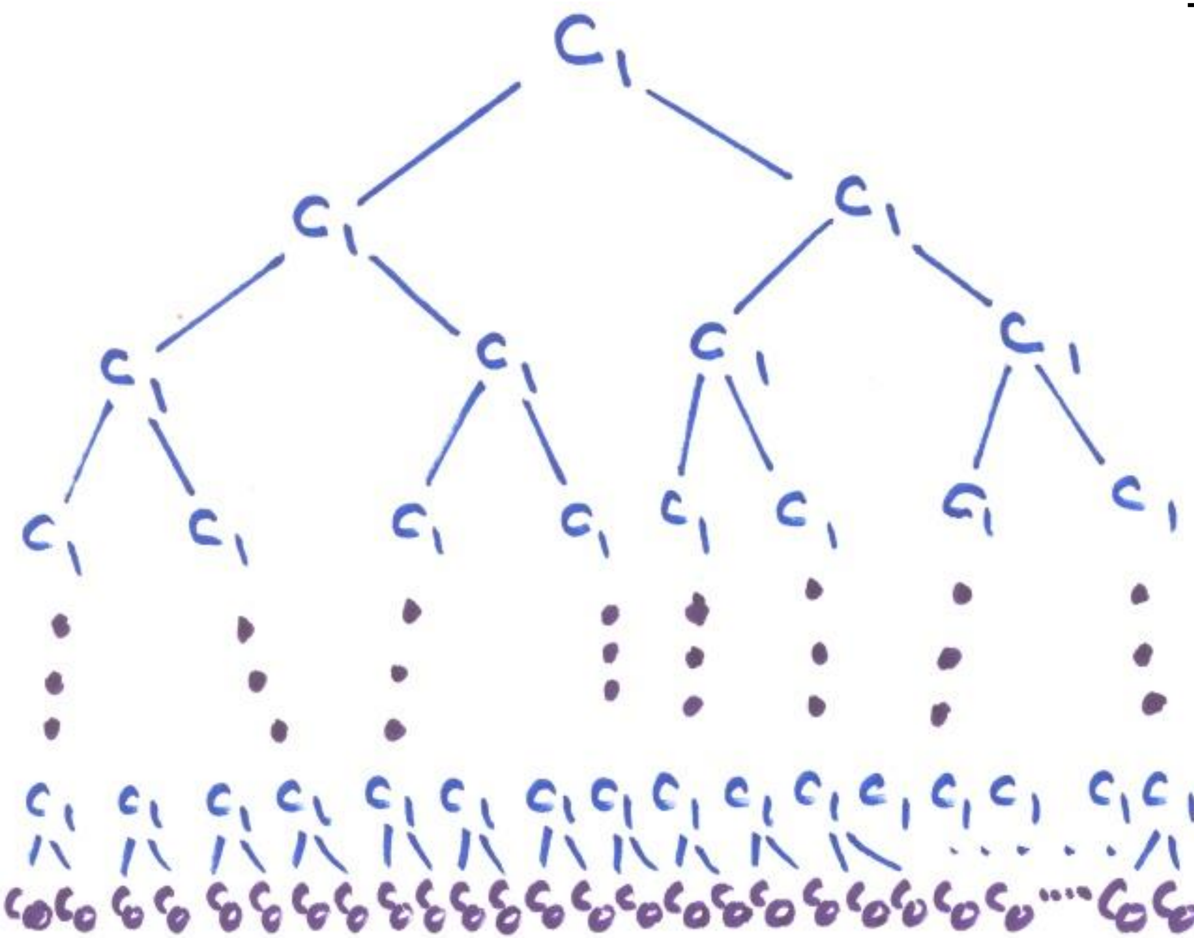
$2^d c_0$

$$W(n) = c_1(1+2+\cdots+2^{d-1}) + c_0 2^d \le c2^{d+1}, \text{ so } O(n).$$

$$S(n) =$$

$(c = \max(c_1, c_0))$

Tree Method for balanced trees:



Work at each level

c_1

$2c_1$

$4c_1$

$8c_1$

$\vdots$

$2^{d-1}c_1$

$2^d c_0$

$$\mathbf{W(n)} = c_1(1+2+\cdots+2^{d-1})+c_0 2^d \leq c2^{d+1}, \text{ so } O(n).$$

$$\mathbf{S(n)} = c_1(1+1+\cdots+1)+c_0 \leq c(d+1), \text{ so } O(\log(n)).$$

$(c = \max(c_1, c_0))$

Sorting

```
datatype order = LESS | EQUAL | GREATER

Int.compare : int * int -> order

String.compare : string * string -> order
```

More generally, for some type `t` may have

```
compare : t * t -> order
```

Sorting

```
datatype order = LESS | EQUAL | GREATER
```

For lists:

`L` is ***sorted*** iff `compare(x,y)` $\Longrightarrow$ `LESS` or `EQUAL`

whenever `x` appears to the left of `y` in `L`.

```
                  LESS|EQUAL
[...,x,...                    ...,y,...]
```

insertion sort for lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x,L) ==> a sorted permutation of x::L
*)

fun ins (x, []) = [x]
  | ins (x, y::ys) = (case compare(x, y) of
                           GREATER => y::ins(x, ys)
                          |  _     => x::y::ys)
```

(Remember our definition of a sorted list:

[…,x,… LESS|EQUAL …,y,…])

insertion sort for lists

```
(* ins : int * int list -> int list
   REQUIRES: L is sorted
   ENSURES: ins(x,L) ==> a sorted permutation of x::L
*)

fun ins (x, []) = [x]
  | ins (x, y::ys) = (case compare(x, y) of
                             GREATER => y::ins(x, ys)
                              |  _    => x::y::ys)

(* isort : int list -> int list
   REQUIRES: true
   ENSURES:  isort(L) ==> a sorted permutation of L
*)

  fun isort [] = []
    | isort (x::xs) = ins (x, isort xs)
```

Code for **`ins`**:

```
fun ins (x, []) = [x]
  | ins (x, y::ys) = (case compare(x, y) of
                           GREATER => y::ins(x, ys)
                          |   _    => x::y::ys)
```

Work:

$\mathbf{W_{ins}(n)}$ with **n** the list length.

Equations:

$\mathbf{W_{ins}(0) = c_0}$

$\mathbf{W_{ins}(n) = c_1 + W_{ins}(n-1)}$, for first **`case`** clause

$\mathbf{W_{ins}(n) = c_2}$, for second **`case`** clause

Consequently, $\mathbf{W_{ins}(n)}$ is $\mathbf{O(n)}$.

Also, observe: no opportunity for parallel speedup.

Code for `isort`:

```
fun isort [] = []
  | isort (x::xs) = ins (x, isort xs)
```

Work:

$\mathbf{W}_{\mathtt{isort}}(\mathbf{n})$ with $\mathbf{n}$ the list length.

Equations:

$$W_{isort}(0) = c_0$$

$$W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)$$

So: $W_{isort}(n) \le c_1 + c_2 \cdot n + W_{isort}(n-1)$

(that should remind you of the recurrence for **`rev`**)

Consequently, $W_{isort}(n)$ is $O(n^2)$.

Again, no opportunity for parallel speedup.

Sorting

	list isort	list merge sort	tree merge sort
Work	$O(n^2)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$
Span	$O(n^2)$	$O(n)$	$O((\log n)^3)$ $O((\log n)^2)$
		(next week)	(next week) (in 15-210)

That is all.

Please have a good weekend.

See you Tuesday.