15-150, Spring 2020

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

$$f(n) \leq c \cdot g(n)$$

for all $n \geq N$. 
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

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

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

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

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

$$O(1), O(n), O(n^2), O(n^3), \ldots,$$

$$O(\log n), O(n \cdot \log n), O(2^n), \ldots$$
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 \texttt{append}:

\begin{verbatim}
  fun [] @ Y = Y
  | (x::xs) @ Y = x::(xs @ Y)
\end{verbatim}

Work analysis of \texttt{append}:

\begin{equation}
  W_{@} (n,m) \text{ with } n \text{ and } m \text{ the sizes of the input lists.}
\end{equation}

Equation for base case:

\begin{equation}
  W_{@} (0,m) = c_0, \text{ for some } c_0, \text{ all } m.
\end{equation}

Equation for recursive clause, for \( n > 0 \):

\begin{equation}
  W_{@} (n,m) = c_1 + W_{@} (n-1,m), \text{ for some } c_1, \text{ all } m.
\end{equation}
Solving: \[ W_{@}(0,m) = c_0 \]

\[ W_{@}(n,m) = c_1 + W_{@}(n-1,m) \]

Unrolling:

\[ W_{@}(n,m) = c_1 + c_1 + W_{@}(n-2,m) \]
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:

\[ W_@ (n,m) = c_1 + c_1 + W_@ (n-2,m) \]

\[ = c_1 + c_1 + c_1 + W_@ (n-3,m) \]

\[ \ldots = n \cdot c_1 + c_0 \quad \text{(can prove this by induction)} \]

So evaluation of \((X @ Y)\) has \(O(n)\) work, with \(n\) the length of \(X\).
**Code for** \texttt{rev}: 

\begin{verbatim}
fun rev [] = []
    | rev (x::xs) = (rev xs) @ [x]
\end{verbatim}

**Work analysis of** \texttt{rev}: 

\[ W_{\text{rev}}(n) \] with \( n \) the size of the input list.

Equation for base case:

\[ W_{\text{rev}}(0) = c_0, \text{ for some } c_0. \]

Equation for recursive clause, for \( n > 0 \):

\[ W_{\text{rev}}(n) = c_1 + W_{\text{rev}}(n-1) + W_{\text{rev}}(n-1,1), \text{ some } c_1. \]

Why?
**Code for rev:**

```ml
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:

$W_{rev}(n) \leq c_1 + W_{rev}(n-1) + c_2(n-1)$, some $c_2$. 
Solving:  \( W_{\text{rev}}(0) = c_0 \)
\[ W_{\text{rev}}(n) \leq c_1 + W_{\text{rev}}(n-1) + c_2(n-1) \]

Unrolling:
\[ W_{\text{rev}}(n) \leq c_1 + c_2 \cdot n + W_{\text{rev}}(n-1) + \left\{ c_1 + c_2(n-1) + W_{\text{rev}}(n-2) \right\} \]
Solving:

\[ W_{rev}(0) = c_0 \]

\[ W_{rev}(n) \leq c_1 + W_{rev}(n-1) + c_2(n-1) \]

Unrolling:

\[ W_{rev}(n) \leq c_1 + c_2 \cdot n + W_{rev}(n-1) \]

\[ W_{rev}(n) \leq c_1 + c_2 \cdot n + \{c_1 + c_2(n-1) + W_{rev}(n-2)\} \]

\[ \leq c_1 + c_2 \cdot n + c_1 + c_2(n-1) \]

\[ + \{c_1 + c_2(n-2) + W_{rev}(n-3)\} \]
Solving: \( w_{\text{rev}}(0) = c_0 \)
\[
 w_{\text{rev}}(n) \leq c_1 + w_{\text{rev}}(n-1) + c_2(n-1)
\]

Unrolling:
\[
 w_{\text{rev}}(n) \leq c_1 + c_2 \cdot n + w_{\text{rev}}(n-1)
\]

\[
 w_{\text{rev}}(n) \leq c_1 + c_2 \cdot n + \{c_1 + c_2(n-1) + w_{\text{rev}}(n-2)\}
\]

\[
 \leq c_1 + c_2 \cdot n + c_1 + c_2(n-1)
\]

\[
 + \{c_1 + c_2(n-2) + w_{\text{rev}}(n-3)\}
\]

\[
 \ldots \leq c_0 + n \cdot c_1 + \left(n(n+1)/2\right) \cdot c_2
\]
Solving: \( w_{rev}(0) = c_0 \)
\[
w_{rev}(n) \leq c_1 + w_{rev}(n-1) + c_2(n-1) \]

Unrolling:
\[
w_{rev}(n) \leq c_1 + c_2 \cdot n + w_{rev}(n-1) \]

\[
w_{rev}(n) \leq c_0 + n \cdot c_1 + \frac{n(n+1)}{2} \cdot c_2 \]

So evaluation of \( rev(L) \) has \( O(n^2) \) work, with \( n \) the length of \( L \).
Analyzing `trev`

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

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

```ml
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:

\[ W_{trev}(0,m) = c_0, \text{ for some } c_0, \text{ all } m. \]

Equation for recursive clause, for \( n > 0 \):

\[ W_{trev}(n,m) = c_1 + W_{trev}(n-1,m+1), \text{ some } c_1, \text{ all } m. \]

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) + x + (sum r)
Code for `sum`:

\[
\begin{align*}
\text{fun sum Empty} &= 0 \\
\text{sum (Node(l,x,r))} &= \text{(sum l) + x + (sum r)}
\end{align*}
\]

Work analysis of `sum`:

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

Equation for base case:

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

Equation for recursive clause, for \( n > 0 \):

\( W_{\text{sum}}(n) = c_1 + W_{\text{sum}}(n_l) + W_{\text{sum}}(n_r) \), 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.
Solving: \[ W_{\text{sum}}(0) = c_0 \]
\[ W_{\text{sum}}(n) = c_1 + W_{\text{sum}}(n_l) + W_{\text{sum}}(n_r) \]

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

\[ W_{\text{sum}}(n) = c_1 n + c_0 (n+1) \]

Fact: A binary tree has \( n \) nodes iff it has \( n+1 \) leaves.

So evaluation of \( \text{sum}(T) \) has \( O(n) \) work.

(can also prove this by induction)
Code for sum:

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

Is there any opportunity for parallelism?

**YES:** The recursive calls to sum can occur in parallel.
Code for \texttt{sum}:

\begin{verbatim}
fun sum Empty = 0
    | sum (Node(l, x, r)) = (sum l) + x + (sum r)
\end{verbatim}

Span analysis of \texttt{sum}:

\( S_{\text{sum}}(n) \) with \( n \) the number of nodes in the tree.

Equation for base case:

\[ S_{\text{sum}}(0) = c_0, \text{ for some } c_0. \]

Equation for recursive clause, for \( n > 0 \):

\[ S_{\text{sum}}(n) = c_1 + \max\{S_{\text{sum}}(n_l), S_{\text{sum}}(n_r)\}, \text{ some } c_1. \]

Notice how \texttt{max} replaces \texttt{+} in the cost analysis.
Solving:  \( S_{\text{sum}}(0) = c_0 \)
\[
S_{\text{sum}}(n) = c_1 + \max\{S_{\text{sum}}(n_l), S_{\text{sum}}(n_r)\}
\]

**ALAS!** It could be that \( n_l = n-1 \) and \( n_r = 0 \).

Then the recursive equation becomes:
\[
S_{\text{sum}}(n) = c_1 + S_{\text{sum}}(n-1)
\]

Therefore \( S_{\text{sum}}(n) \) is \( O(n) \), 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:

\[ S_{\text{sum}}(0) = c_0 \]

\[ S_{\text{sum}}(n) \approx c_1 + \max\{S_{\text{sum}}(n/2), S_{\text{sum}}(n/2)\} \]
Suppose however that the tree is balanced.

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

\[
S_{\text{sum}}(0) = c_0 \\
S_{\text{sum}}(n) = c_1 + \max\{S_{\text{sum}}(n/2), S_{\text{sum}}(n/2)\}
\]

Then:

\[
S_{\text{sum}}(n) = c_1 + S_{\text{sum}}(n/2) \\
= c_1 + c_1 + S_{\text{sum}}(n/4) \\
\ldots = c_1 + c_1 + \cdots + c_1 + c_0 \\
\underbrace{(\lfloor \log_2 n \rfloor + 1)}_{\text{many times}}
\]

Now \( S_{\text{sum}}(n) \) is \( O(\log(n)) \), meaning parallelism is significant.
We could also have obtained this result by expressing span as $S_{sum}(d)$, with $d$ the depth of the tree.

Then: 

$S_{sum}(0) = c_0$

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

So 

$S_{sum}(d) = c_1 + S_{sum}(d-1)$

Thus $S_{sum}(d)$ is $O(d)$. 

This result holds for all trees. (d=n is possible)

For balanced trees, $d$ is $O(\log(n))$, and we again see that parallelism helps.
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

\[
\text{datatype order} = \text{LESS} \mid \text{EQUAL} \mid \text{GREATER}
\]

For lists:

\[
\text{L is sorted iff } \text{compare}(x,y) \Rightarrow \text{LESS or EQUAL}
\]

whenever \( x \) appears to the left of \( y \) in \( L \).
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:

\[
[\ldots, x, \ldots \ LEQ \ 
   \ldots, y, \ldots ]
\]  )
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 \texttt{ins}:

\begin{verbatim}
  fun ins (x, []) = [x]
  | ins (x, y::ys) = case compare(x, y) of
    GREATER => y::ins(x, ys)
  | _       => x::y::ys
\end{verbatim}

\textbf{Work:}

\[ \texttt{W}_{\text{ins}}(n) \] with \( n \) the list length.

\textbf{Equations:}

\[ \texttt{W}_{\text{ins}}(0) = c_0 \]

\[ \texttt{W}_{\text{ins}}(n) = c_1 + \texttt{W}_{\text{ins}}(n-1) \), for first \texttt{case} clause \]

\[ \texttt{W}_{\text{ins}}(n) = c_2 \), for second \texttt{case} clause \]

Consequently, \( \texttt{W}_{\text{ins}}(n) \) is \( O(n) \).

Also, observe: no opportunity for parallel speedup.
Code for `isort`:

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

Work:

\[ W_{isort}(n) \] with \( n \) the list length.

Equations:

\[
W_{isort}(0) = c_0
\]

\[
W_{isort}(n) = c_1 + W_{isort}(n-1) + W_{ins}(n-1)
\]

So:

\[
W_{isort}(n) \leq c_1 + c_2 \cdot n + W_{isort}(n-1)
\]

(That should remind you of the recurrence for \( \text{rev} \))

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

Again, no opportunity for parallel speedup.
<table>
<thead>
<tr>
<th></th>
<th>list isort</th>
<th>list merge sort</th>
<th>tree merge sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work</td>
<td>O($n^2$)</td>
<td>O($n \cdot \log n$)</td>
<td>O($n \cdot \log n$)</td>
</tr>
<tr>
<td>Span</td>
<td>O($n^2$)</td>
<td>O($n$)</td>
<td>O($((\log n)^3)$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>O($((\log n)^2)$)</td>
</tr>
<tr>
<td>(next week)</td>
<td></td>
<td></td>
<td>(next week)</td>
</tr>
<tr>
<td>(in 15-210)</td>
<td></td>
<td></td>
<td>(in 15-210)</td>
</tr>
</tbody>
</table>