Today

• Work
  • sequential runtime

• Recurrences
  • exact and approximate solutions

• Improving efficiency

program $\rightarrow$ recurrence $\rightarrow$ work
asymptotic

• Want the *runtime* of evaluating $f(n)$, for *large* $n$
  • *independent* of architecture
  • basic ops take *constant time*
• We will give a **big-O** classification

$f(n)$ is $O(g(n))$ if there are $N$ and $c$ such that

$$\forall n \geq N, f(n) \leq c \cdot g(n)$$
The graph below compares the running times of various algorithms.

- Linear -- $O(n)$
- Quadratic -- $O(n^2)$
- Cubic -- $O(n^3)$
- Logarithmic -- $O(\log n)$
- Exponential -- $O(2^n)$
- Square root -- $O(\sqrt{n})$
motivation

Why take exponential time when we can take quadratic time?
asymptotic

- **Ignore** additive constants
  
  \[ n^5 + 1000000 \text{ is } O(n^5) \]

- **Absorb** multiplicative constants
  
  \[ 1000000n^5 \text{ is } O(n^5) \]

- Be as accurate as you can
  
  \[ O(n^2) \subset O(n^3) \subset O(n^4) \]

- **Common terminology**
  
  logarithmic, linear, polynomial, exponential
**work**

- \( W(e) \), the *work* of \( e \), is the time to evaluate \( e \) sequentially, on a single processor
  - each basic operation is constant-time
  - work = total number of operations
- Often we have a function \( f \) and a notion of *size* for *argument values*, and want \( W_f(n) \), the work of \( f(v) \) when \( v \) has size \( n \)

May want either *exact* or *asymptotic* estimate
work and evaluation

- Evaluation steps $e \Rightarrow e'$ represent basic ops, so the work for $e$ is the number of steps

$$W((e_1 + e_2)) = W(e_1) + W(e_2) + 1$$

$$(2+2) + (2+2) \Rightarrow 4 + (2+2)$$
$$\Rightarrow 4 + 4$$
$$\Rightarrow 8$$

If $e \Rightarrow^{(k)} v$ then $W(e) = k$

$W((2+2) + (2+2)) = 3$
If $f$ is a function value and $e \Rightarrow v$ then $W(f \ e) = k + W(f \ v)$

<table>
<thead>
<tr>
<th>$f$</th>
<th>$(\text{fn } x \Rightarrow x+x) \ (2+2)$</th>
<th>$W((\text{fn } x \Rightarrow x+x) \ (2+2))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>=&gt;</td>
<td>$(\text{fn } x \Rightarrow x+x) \ 4$</td>
<td>$= 1 + W((\text{fn } x \Rightarrow x+x) \ 4)$</td>
</tr>
<tr>
<td>=&gt;</td>
<td>$4+4$</td>
<td>$= 1 + 1 + W(4+4)$</td>
</tr>
<tr>
<td>=&gt;</td>
<td>$8$</td>
<td>$= 3$</td>
</tr>
</tbody>
</table>
recurrences

• Given a recursive function \( f \) and a non-negative size that decreases in every recursive call

• Extract a recurrence relation for the applicative work of \( f \)

\[ W_f(n) = \text{work of } f \ v \text{ on values } v \text{ of size } n \]

Idea: express \( W_f(n) \) in terms of \( W_f(m), 0 \leq m < n \)

Q: When can this method succeed?

A: If the work of \( f \ v \) depends only on the size of \( v \) (!)
fun \( f(x) = \text{if } x=0 \text{ then } 1 \text{ else } x + f(x-1) \)

if \( x \geq 0 \), argument value is non-negative, decreases...

\[
W_f(n) = \text{if } n=0 \text{ then } k_1 \text{ else } k_2 + W_f(n-1)
\]

where \( k_1, k_2 \) are constants

\[
W_f(0) = k_1 \\
W_f(n) = k_2 + W_f(n-1) \quad \text{for } n>0
\]

\[
W_f(n) = k_1 + n \ k_2 \quad \text{for all } n \geq 0
\]
example

fun Fib(0) = 1
| Fib(1) = 1
| Fib(n) = Fib(n-1) + Fib(n-2)

A recurrence for the work to evaluate Fib(n)

\[ W_{Fib}(0) = c_0 \]
\[ W_{Fib}(1) = c_0 \]
\[ W_{Fib}(n) = W_{Fib}(n-1) + W_{Fib}(n-2) + c_1 \]
for some constants \( c_0, c_1 \)
finding solutions

Try to find a closed form solution for $W(n)$
(usually, by guessing and induction)

OR Code the recurrence in ML, test for small $n$,
look for a common pattern

OR Find solution to a simplified recurrence
with the same asymptotic properties

OR Appeal to table of standard recurrences
fun exp (n:int):int = 
  if n=0 then 1 else 2 * exp (n-1)

exp 4 => (1) M 4
    => (5) 2 * (M 3)
    => (5) 2 * (2 * (M 2))
    => (5) 2 * (2 * (2 * (M 1)))
    => (5) 2 * (2 * (2 * (2 * (M 0))))
    => (3) 2 * (2 * (2 * (2 * 1)))
    => (4) 16

Let M be (fn n => if n=0 then 1 else 2 * exp(n-1))

M 4 => if 4=0 then ...
    => if false then ...
    => 2 * exp (4-1)
    => 2 * M (4-1)
    => 2 * M 3

M 3 => (5) 2 * M 2

exp 4 => (28) 16
exp

It’s not hard to prove that for all $n \geq 0$,

$$\text{exp } n \Rightarrow (5n+8) \; k,$$

where $k$ is the numeral for $2^n$

But do we need to be so accurate?

And does $5n+8$ tell us about actual runtime in milliseconds?

No! But it does tell us runtime is linear.
big-O

- It’s useful to classify runtimes *asymptotically*

- This abstracts away from additive and multiplicative constants
  (which may be machine-dependent, so not very significant)

- And ignores runtime on small inputs
  (which may be special-cased in the code, so don’t imply much)

For \( f, g : \text{int} \rightarrow \text{int} \) we say that

\[ f \text{ is } O(g) \]

if there is a constant \( c \) and an integer \( N \) such that for all \( n \geq N \),

\[ |f(n)| \leq c \times |g(n)|. \]
Let $W_{\text{exp}}(n)$ be the runtime for $\text{exp}(n)$

\[
W_{\text{exp}}(0) = c_0
\]

\[
W_{\text{exp}}(n) = W_{\text{exp}}(n-1) + c_1 \quad \text{for } n>0
\]

for some constants $c_0$ and $c_1$

$c_0$ cost of $n=0$

$c_1$ cost of $n=0, n-1, \text{mult by } 2$
solution

- Can prove by induction on $n$ that

$$W_{\text{exp}}(n) = c_0 + n c_1 \quad \text{for } n \geq 0$$

the work of $\text{exp}(n)$ is *linear in* $n$
• \( W_{\text{exp}}(n) = c_0 + n c_1 \)
• \( W_{\text{exp}}(n) \) is \( O(n) \)

Let \( N=42 \), \( c = \max(c_0, c_1) + 1 \).

For all \( n \geq N \),
\[ W_{\text{exp}}(n) \leq c \cdot n \]

(would also work with \( N=1 \))

(would also work with an even bigger \( c \))
summary

• We’ve shown that for \( n \geq 0 \),
  \( \exp n \) computes the value of \( 2^n \)
  in \( O(n) \) steps

• This fact is independent of machine details
  (provided basic operations are constant time)

• Can we do better?
faster exp?

• The definition of exp relies on the fact that

\[ 2^n = 2 \cdot (2^{n-1}) \]

• Everybody knows that

\[ 2^n = (2^{n \div 2})^2 \text{ if } n \text{ is even} \]
fun square(x:int):int = x * x

fun fastexp (n:int):int =
  if n=0 then 1 else
  if n mod 2 = 0 then square(fastexp (n div 2))
  else 2 * fastexp(n-1)

fastexp 4 = square(fastexp 2)
  = square(square (fastexp 1))
  = square(square (2 * fastexp 0))
  = square(square (2 * 1))
  = square 4 = 16
is it faster?

\[
\text{fun fastexp (n:int):int =}
\begin{align*}
\text{if } n=0 \text{ then } & 1 \\
\text{else if } n \text{ mod } 2 = 0 \text{ then } & \text{square(fastexp (n div 2))} \\
\text{else } & 2 \times \text{fastexp(n-1)}
\end{align*}
\]

Let \( W_{\text{fastexp}}(n) \) be the work for fastexp(n)

\[
\begin{align*}
W_{\text{fastexp}}(0) &= k_0 \\
W_{\text{fastexp}}(n) &= W_{\text{fastexp}}(n \text{ div } 2) + k_1 \quad \text{for } n>0, \text{ even} \\
W_{\text{fastexp}}(n) &= W_{\text{fastexp}}(n-1) + k_2 \quad \text{for } n>0, \text{ odd}
\end{align*}
\]

for some constants \( k_0, k_1, k_2 \)
is it faster?

fun fastexp (n:int):int = 
  if n=0 then 1 else 
  if n mod 2 = 0 then square(fastexp (n div 2)) 
    else 2 * fastexp(n-1)

Let $W_{\text{fastexp}}(n)$ be the work for fastexp(n)

$W_{\text{fastexp}}(0) = c_0$

$W_{\text{fastexp}}(1) = c_1$

$W_{\text{fastexp}}(n) = W_{\text{fastexp}}(n \text{ div } 2) + c_2$ \quad \text{for } n>1, \text{ even}

$W_{\text{fastexp}}(n) = W_{\text{fastexp}}(n \text{ div } 2) + c_3$ \quad \text{for } n>1, \text{ odd}

for some constants $c_0, c_1, c_2, c_3$
solution?

• Not so obvious how to solve for $W_{\text{fastexp}}(n)$
• A closed form would involve $c_0, c_1, c_2, c_3$
• But we only care about asymptotic behavior
• So we can work with a simpler recurrence that has the same asymptotic properties

simplification:
choose each constant to be 1
simplified recurrence

Let $T_{\text{fastexp}}(n)$ be given by

$$T_{\text{fastexp}}(0) = 1$$
$$T_{\text{fastexp}}(1) = 1$$
$$T_{\text{fastexp}}(n) = T_{\text{fastexp}}(n \div 2) + 1 \text{ for } n>1$$

$W_{\text{fastexp}}(n)$ and $T_{\text{fastexp}}(n)$ are asymptotically equivalent
(belong to the same big-O class)
solution

• For $n > 1$, $T_{\text{fastexp}}(n)$ is defined like $\log(n)$

  \begin{verbatim}
  fun log n = 
    if n=1 then 0 else \log(n \text{ div } 2) + 1
  \end{verbatim}

• We know that $\log(n) = \log_2(n)$ for all $n > 0$

• Can show that there is a constant $c$ such that

  $T_{\text{fastexp}}(n) \leq c \log_2(n)$

  for all large enough $n$
classification

- \( T_{fastexp}(n) \) is \( O(\log_2 n) \)
- \( W_{fastexp}(n) \) depends on \( c_0, c_1, c_2, c_3 \)
- We can find constants \( c_{low} \) and \( c_{high} \) such that
  \[
  c_{low} T_{fastexp}(n) \leq W_{fastexp}(n) \leq c_{high} T_{fastexp}(n)
  \]
  and this implies that
  \( W_{fastexp}(n) \) is also \( O(\log_2(n)) \)
really, faster

- Work of $\exp(n)$ is $O(n)$
- Work of $\text{fastexp}(n)$ is $O(\log n)$
- $O(\log n)$ is a proper subset of $O(n)$
- $\text{fastexp}$ is \emph{asymptotically faster} than $\exp$
even faster?

• The definition of \texttt{fastexp} relies on

\[
2^n = (2^{n \text{ div } 2})^2 \quad \text{if } n \text{ is even}
\]
\[
2^n = 2 \times (2^{n-1}) \quad \text{if } n \text{ is odd}
\]

• A moment’s thought tells us that

\[
2^n = 2 \times (2^{(n \text{ div } 2)})^2 \quad \text{if } n \text{ is odd}
\]
fun pow (n:int):int =
  case n of
    0 => 1
    | 1 => 2
    | _ => let
      in
      in
        if n mod 2 = 0 then k*k else 2*k*k
      end
**work of pow(n)**

\[ W_{\text{pow}}(0) = c_0 \]
\[ W_{\text{pow}}(1) = c_1 \]
\[ W_{\text{pow}}(n) = c_2 + W_{\text{pow}}(n \div 2) \text{ for } n > 1 \]

Same recurrence as \( W_{\text{fastexp}} \)

Same asymptotic behavior

\( \text{pow}(n) \text{ is } O(\log n) \)
comparison

- \text{fastexp}(n) \text{ and } \text{pow}(n) \text{ have } O(\log n) \text{ work.}
- \text{For } n \geq 0, \text{fastexp}(n) = \text{pow}(n).
- \text{For } n < 0, \text{fastexp}(n) \text{ and } \text{pow}(n) \text{ fail to terminate.}
- \text{So fastexp and pow are extensionally equivalent and have the same asymptotic work classification.}
fun badpow (n:int):int = 
  case n of 
    0 => 1 
  | 1 => 2 
  | _ => let 
    val k2 = badpow(n div 2) * badpow(n div 2) 
  in 
    if n mod 2 = 0 then k2 else 2 * k2 
  end
work of \text{badpow}(n)

\begin{align*}
W_{\text{badpow}}(0) &= c_0 \\
W_{\text{badpow}}(1) &= c_1 \\
W_{\text{badpow}}(n) &= c_2 + 2W_{\text{badpow}}(n \text{ div } 2) \\
& \quad \text{for } n > 1
\end{align*}

Same asymptotic class as

\begin{align*}
T_{\text{badpow}}(0) &= 1 \\
T_{\text{badpow}}(1) &= 1 \\
T_{\text{badpow}}(n) &= 1 + 2T_{\text{badpow}}(n \text{ div } 2) \\
& \quad \text{for } n > 1
\end{align*}
examples

\[ T_{\text{badpow}}(2^0) = 1 \]
\[ T_{\text{badpow}}(2^1) = 1 + 2 \times T_{\text{badpow}}(2^0) \]
\[ = 1 + 2 \times 1 = 3 \]
\[ T_{\text{badpow}}(2^2) = 1 + 2 \times T_{\text{badpow}}(2^1) \]
\[ = 1 + 2 \times 3 = 7 \]
\[ T_{\text{badpow}}(2^m) = 2^{m+1} - 1 \]
analysis

\[ T_{\text{badpow}}(2^m) \text{ is } O(2^m) \]

• \[ W_{\text{badpow}}(2^m) \text{ is } O(2^m) \]

• This implies that \( W_{\text{badpow}}(n) \text{ is } O(n) \)

\[ W_{\text{pow}}(n) \text{ is } O(\log n) \]
\[ O(\log n) \subset O(n) \]

\textit{pow} is \textit{asymptotically faster} than \textit{badpow}
list reversal

fun rev [ ] = [ ]
| rev (x::L) = (rev L) @ [x]

For list values A and B, \( W(A, B) \) is *linear* in the length of A.

For all L, \( \text{length}(\text{rev L}) = \text{length}(L) \)

Runtime of \( \text{rev}(L) \)
depends only on *length of L*
work of rev

fun rev [ ] = [ ]
  | rev (x::L) = (rev L) @ [x]

• $W_{rev}(n) = \text{work to reverse a list of length } n$

  $W_{rev}(0) = 1$
  $W_{rev}(n) = W_{rev}(n-1) + (n-1) + 1$
solution

\[ W_{\text{rev}}(n) = n + W_{\text{rev}}(n-1) \]

\[ = n + (n-1) + W_{\text{rev}}(n-2) \]

\[ = n + (n-1) + \ldots + 1 + W_{\text{rev}}(0) \]

\[ = \frac{1}{2} n(n+1) + 1 \]

\[ W_{\text{rev}}(n) \text{ is } O(n^2) \]

quadratic runtime
faster \texttt{rev}

- Use an extra argument to \textit{accumulate} the reversed list
  
  \texttt{revver : int list * int list -> int list}

- Instead of \textit{append} after the recursive call, do a \textit{cons} before the recursive call
  
  \texttt{fun revver([ ],A) = A}
  \texttt{| revver(x::L,A) = revver(L, x::A)}
fun revver([], A) = A
  | revver(x::L, A) = revver(L, x::A)

fun Rev L = revver(L, [])

For all L, A,  revver(L, A) = (rev L) @ A
For all L,  Rev L = rev L
analysis

- Explain why $W_{\text{revver}}(n)$ is $O(n)$
standard results

- $T(n) = c + T(n-1)$ $\quad O(n)$
- $T(n) = c + n + T(n-1)$ $\quad O(n^2)$
- $T(n) = c + T(n \text{ div } 2)$ $\quad O(\log n)$
- $T(n) = c + 2T(n \text{ div } 2)$ $\quad O(n)$
- $T(n) = c + kT(n-1)$ $\quad O(k^n)$