Today

• Work and span
  • sequential and parallel runtime

• Recurrences
  • exact and asymptotic solutions

• Improving efficiency
  • careful program design

program $\rightarrow$ recurrence
what matters

• Correctness

TYPE f : t₁ -> t₂
REQUIRES (value) x (:t₁) such that …
ENSURES f x ⟺* v (:t₂) such that …

• Efficiency

Information about evaluation time of f x

f x ⟺ h(x) steps v

- exact number of steps h(x) depends on x and definition of f

An asymptotic estimate is good enough!

- h(x) is \( O(g(size x)) \) for some notion of size
asymptotic

• We want to estimate the runtime $W_f(n)$ for evaluating $f(n)$, for large $n$
  
  assuming basic operations take constant time

• We will give a big-$O$ classification

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

$\forall n \geq N, \ W_f(n) \leq c \ g(n)$
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
motivation
motivation

Why make trillions when we could make billions?
motivation

Why take linear time when we can solve the problem in log time?
motivation

Why take linear time when we can solve the problem in log time?

isqrt_0 123456789
Why take linear time when we can solve the problem in log time?
motivation

Why take linear time when we can solve the problem in log time?

isqrt_0 123456789
isqrt_1 123456789
isqrt_2 123456789
asymptotically

- **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
  
  \[ O(n^2) \subset O(n^3) \subset O(n^4) \]

- Use common terminology
  
  logarithmic, linear, quadratic, polynomial, exponential
asymptotically

• **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
  
  \[ O(n^2) \subset O(n^3) \subset O(n^4) \]

• Use common terminology
  
  logarithmic, linear, quadratic, polynomial, exponential

\[ O(10^9) = O(10^6) \]
rules of thumb

To calculate work, span for recursive functions we can use recurrence relations, e.g.

$$W_f(n) = k \times W_f(n-1) + c \quad \text{for } n>0$$

where $k, c$ are constants

- Additive constants don’t matter
  
  WLOG let $c = 1$

- Multiplicative constants do matter
  
  $W_f(n)$ is $O(k^n)$
  
  $O(2^n)$ is not the same as $O(3^n)$
work

- $W(e)$, the work of $e$, is the time to evaluate $e$ sequentially, on a single processor
  \[
  \text{work} = \text{total number of operations}
  \]

- 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 $n$
Span

- $S(e)$, the span of $e$, is the time to evaluate $e$, using parallel evaluation for independent code.

- Often we have a function $f$ and a notion of size for argument values, and want $S_f(n)$, the span of $f(v)$ when $v$ has size $n$. 
rules of thumb

• Most primitive ops are constant-time
  • but not @ on lists (it does a bunch of :: operations)

• To calculate work,
  - add the work for sub-expressions

• To calculate span,
  - max the span for independent sub-expressions
  - add the span for dependent sub-expressions
dependence

• if \( b \) then \( e_1 \) else \( e_2 \) \( b \) before \( e_1 \) or \( e_2 \)
• \((\text{fn } x \Rightarrow e_2) \) \( e_1 \) \( e_1 \) before \( [x:v_1]e_2 \)
• let val \( x = e_1 \) in \( e_2 \) end \( e_1 \) before \( [x:v_1]e_2 \)

independence

• \((e_1, \ldots, e_n)\) tuple components
• \( e_1 + e_2 \) summands
work rules

\[ W(n) = 0 \]

\[ W(e_1 + e_2) = W(e_1) + W(e_2) + 1 \]

\[ W(e_1, e_2) = W(e_1) + W(e_2) \]

\[ W(e_1 \oplus e_2) = W(e_1) + W(e_2) + \text{length}(e_1) + 1 \]

\[ W(\text{if } b \text{ then } e_1 \text{ else } e_2) \]
\[ \leq W(b) + \max(W(e_1), W(e_2)) + 1 \]
span rules

\[ S\ (\text{null}) = 0 \]

\[ S\ (e_1 + e_2) = \max(S\ e_1, S\ e_2) + 1 \]

\[ S\ (e_1, e_2) = \max(S\ e_1, S\ e_2) \]

\[ S\ (e_1 @ e_2) = \max(S\ e_1, S\ e_2) + \text{length } e_1 + 1 \]

\[ S\ (\text{if } b \text{ then } e_1 \text{ else } e_2) \leq S\ b + \max(S\ e_1, S\ e_2) + 1 \]
work and evaluation

- An evaluation step $e \implies e'$ represents a basic op, so the exact work for $e$ is the number of steps
work and evaluation

- An evaluation step $e \Rightarrow e'$ represents a basic op, so the exact work for $e$ is the number of steps

$$\text{If } e \Rightarrow^{(k)} v \text{ then } W(e) = k$$
work and evaluation

- An evaluation step $e \implies e'$ represents a basic op, so the exact work for $e$ is the number of steps.

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

$(2+2)+(2+2) \implies 4+(2+2)
\implies 4+4
\implies 8$
work and evaluation

- An evaluation step $e \implies e'$ represents a basic op, so the exact work for $e$ is the number of steps.

$$W((2+2) + (2+2)) = 3$$

\[
\begin{align*}
(2+2) + (2+2) & \implies 4 + (2+2) \\
& \implies 4 + 4 \\
& \implies 8
\end{align*}
\]

If $e \implies^{(k)} v$ then $W(e) = k$
work and evaluation

- An evaluation step $e \implies e'$ represents a basic op, so the exact work for $e$ is the number of steps.

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

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

$$W((2+2) + (2+2)) = 3$$

$$W(e_1+e_2) = W(e_1) + W(e_2) + 1$$
work and application

If \( e_1 \xrightarrow{\ast} (\text{fn } x \Rightarrow e) \) and \( e_2 \xrightarrow{\ast} v \),
then \( W(e_1, e_2) = W(e_1) + W(e_2) + \text{W}(\forall x:v \ e) + 1 \)

\[
\begin{align*}
(\text{fn } x \Rightarrow x+x) \ (2+2) &
\Rightarrow (\text{fn } x \Rightarrow x+x) \ 4 \\
\Rightarrow 4+4 \\
\Rightarrow 8 \quad \text{(3 steps)}
\end{align*}
\]

\[
\begin{align*}
W \ ((\text{fn } x \Rightarrow x+x) \ (2+2)) 
\end{align*}
\]
work and application

If $e_1 \Rightarrow^* (\text{fn } x \Rightarrow e)$ and $e_2 \Rightarrow^* v$, then $W(e_1 e_2) = W(e_1) + W(e_2) + W([x:v]e) + 1$

$(\text{fn } x \Rightarrow x+x) \ (2+2)$

$\Rightarrow (\text{fn } x \Rightarrow x+x) \ 4$

$\Rightarrow 4+4$

$\Rightarrow 8 \quad (3 \text{ steps})$

$W ((\text{fn } x \Rightarrow x+x) \ (2+2))$

$= 0 + 1 + W(4+4) + 1$
If $e_1 \Rightarrow^* (\text{fn } x \Rightarrow e)$ and $e_2 \Rightarrow^* v$, then $W(e_1 \; e_2) = W(e_1) + W(e_2) + W([x:v]e) + 1$

\[(\text{fn } x \Rightarrow x+x) \; (2+2)\]  
\[\Rightarrow (\text{fn } x \Rightarrow x+x) \; 4\]  
\[\Rightarrow 4+4\]  
\[\Rightarrow 8 \quad (3 \text{ steps})\]  

\[W \; ((\text{fn } x \Rightarrow x+x) \; (2+2))\]  
\[= 0 + 1 + W(4+4) + 1\]  
\[= 0 + 1 + 1 + 1 + 1\]
work and application

If \( e_1 \Rightarrow^* (\text{fn } x \Rightarrow e) \) and \( e_2 \Rightarrow^* v \), then \( W(e_1, e_2) = W(e_1) + W(e_2) + W(\\langle x:v \rangle e) + 1 \)

\[
(\text{fn } x \Rightarrow x+x) (2+2) \\
\Rightarrow (\text{fn } x \Rightarrow x+x) 4 \\
\Rightarrow 4+4 \\
\Rightarrow 8 \quad (3 \text{ steps})
\]

\[
W ((\text{fn } x \Rightarrow x+x) (2+2)) \\
= 0 + 1 + W(4+4) + 1 \\
= 0 + 1 + 1 + 1 \\
= 3
\]
fun exp (n:int):int = 
  if n=0 then 1 else 2 * exp (n-1)

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

exp 4 ⇒(1) M 4
fun exp (n:int):int = 
    if n=0 then 1 else 2 * exp (n-1)

Let M be \((\text{fn} \ n \Rightarrow \text{if} \ n=0 \ \text{then} \ 1 \ \text{else} \ 2 \ast \text{exp}(n-1))\)

exp 4 \Rightarrow^{(1)} M 4

\Rightarrow^{(5)} 2 \ast (M 3)
fun exp (n:int):int = 
    if n=0 then 1 else 2 * exp (n-1)

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

exp 4 ⟹ (M 4)
    ⟹ (2 * (M 3))
fun exp (n:int):int =  
    if n=0 then 1 else 2 * exp (n-1)

Let M be \((\text{fn } n \Rightarrow \text{if } n=0 \text{ then } 1 \text{ else } 2 \times \exp(n-1))\)

\[
\begin{align*}
\text{exp 4} & \Rightarrow^{(1)} M 4 \\
& \Rightarrow^{(5)} 2 \times (M 3)
\end{align*}
\]
fun exp (n:int):int = 
  if n=0 then 1 else 2 * exp (n-1)

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

exp 4 \Rightarrow^{(1)} M 4
\Rightarrow^{(5)} 2 * (M 3)
\Rightarrow^{(5)} 2 * (2 * (M 2))
fun exp (n:int):int = 
if n=0 then 1 else 2 * exp (n-1)

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

exp 4 ⟹(1) M 4
  ⟹(5) 2 * (M 3)
  ⟹(5) 2 * (2 * (M 2))
fun exp (n:int):int = 
    if n=0 then 1 else 2 * exp (n-1)

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

exp 4 ⇒(1) M 4
    ⇒(5) 2 * (M 3)
    ⇒(5) 2 * (2 * (M 2))
fun exp (n:int):int = 
  if n=0 then 1 else 2 * exp (n-1)

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

exp 4 \Rightarrow (1) M 4
  \Rightarrow (5) 2 * (M 3)
  \Rightarrow (5) 2 * (2 * (M 2))
  \Rightarrow (5) 2 * (2 * (2 * (M 1))))
fun exp (n:int):int = 
  if n=0 then 1 else 2 * exp (n-1)

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

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

exp 4 \Rightarrow^{(1)} M 4
\Rightarrow^{(5)} 2 * (M 3)
\Rightarrow^{(5)} 2 * (2 * (M 2))
\Rightarrow^{(5)} 2 * (2 * (2 * (M 1)))
\Rightarrow^{(5)} 2 * (2 * (2 * (2 * (M 0))))
\Rightarrow^{(3)} 2 * (2 * (2 * (2 * 1)))
fun exp (n:int):int = 
if n=0 then 1 else 2 * exp (n-1)

Let M be \( \text{fn } n \Rightarrow \text{if } n=0 \text{ then } 1 \text{ else } 2 * \text{exp}(n-1) \)
fun exp (n:int):int = 
  if n=0 then 1 else 2 * exp (n-1)

Let M be (fn n => 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

exp 4 ⟹(28) 16
exp

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

\[
\exp n \iff (6n+4) \ k,
\]

where \( k \) is the numeral for \( 2^n \).
exp

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

$$\exp n \implies (6^n + 4) \cdot k,$$

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

But it’s tedious, and why be so accurate?
It’s not hard to prove that for all $n \geq 0$,

$$\exp n \Rightarrow (6n+4) \, k,$$

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

But it’s tedious, and why be so accurate?

Does $6n+4$ really tell us about actual *runtime* in milliseconds?
It’s not hard to prove that for all $n \geq 0$,

$$\exp n \implies (6n+4) \ k,$$

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

But it’s tedious, and why be so accurate?

Does $6n+4$ really tell us about actual runtime in milliseconds?

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

• It’s best to classify runtimes *asymptotically*

• This ignores irrelevant constants…
  (which may be machine-dependent, so not very significant)

• … and ignores runtime on small inputs
  (which may have been special-cased in the code)

\[ \exp n \Rightarrow O(n) \ 2^n \]

If we *double* \( n \),
the runtime… *doubles*
recurrences

• Given a recursive definition for function $f$ and a non-negative size function that decreases in every recursive call

• Extract a recurrence relation for the applicative work of $f$

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

- Given a recursive definition for function $f$ and a non-negative size function 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 $W_f(m)$, $0 \leq m < n$.
recurrences

• Given a recursive definition for function $f$ and a non-negative size function that decreases in every recursive call

• Extract a recurrence relation for the applicative work of $f$

  $W_f(n) =$ work of $f \, v$ on values $v$ of size $n$

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

Q: When can this method succeed?
recurrences

• Given a *recursive definition for function* $f$ and a non-negative *size* function 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 $\text{Fib}(0) = 1$
| $\text{Fib}(1) = 1$
| $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$

$\text{W}_{\text{Fib}}(0) = c_0$
$\text{W}_{\text{Fib}}(1) = c_0$
$\text{W}_{\text{Fib}}(n) = \text{W}_{\text{Fib}}(n-1) + \text{W}_{\text{Fib}}(n-2) + c_1$

for some constants $c_0, c_1$

size is value of $n$
solving a recurrence

WLOG let additive constants be 1

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
Let $W_{\text{exp}}(0) = c_0$

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

for some constants $c_0$ and $c_1$

$c_0$ : cost for test $n=0$
$c_1$ : cost for test $n=0$, multiply by 2

$\text{fun exp (n:int):int = if n=0 then 1 else 2 * exp (n-1)}$

For $n \geq 0$, $\text{exp n} \implies * 2^n$
solution

• Easy to prove by induction on $n$ that

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

$W_{\text{exp}}(n)$ is $O(n)$

The work for $\text{exp}(n)$ is \textit{linear}
If we’d simplified by letting constants be 1,

\[ W_{\text{exp}}(0) = 1 \]

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

we’d have gotten  \[ W_{\text{exp}}(n) = 1 + n \]

\[ W_{\text{exp}}(n) \text{ is } O(n) \]

The simpler recurrence has the same solution, asymptotically.
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 
  (assuming that basic operations are constant time)

• Can we do better?
use parallelism?
(with the same \texttt{exp} function)

\begin{verbatim}
fun exp (n:int):int =
  if n=0 then 1 else 2 * exp (n-1)
\end{verbatim}

• Give a recurrence for the \textit{span} of \texttt{exp \ n}

It will be \textit{identical} to the recurrence we gave for work, with the same asymptotic solution... why?

There is no advantage to be gained by parallel evaluation here!
a faster method?

- The definition of `exp` relies on the fact that

  \[ 2^n = 2 \cdot (2^{n-1}) \quad \text{when } n > 0 \]

- Everybody knows that

  \[ 2^n = (2^n \div 2)^2 \quad \text{when } n \text{ is even} \]

Let’s define

\[
\text{fastexp : int -> int}
\]

based on this idea…
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)
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?

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)
is it faster?

```plaintext
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)
```

Code design leads to recurrence…
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)

Code design leads to recurrence...

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

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)

Code design leads to recurrence…

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

for some constants \( k_0, k_1, k_2 \)

\( k_0 \): cost for test \( n=0 \)
\( k_1 \): cost for tests \( n=0, n \mod 2 = 0 \), squaring
\( k_2 \): cost for tests \( n=0, n \mod 2 = 0 \), multiplication by 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)
```

Expand, then set constants to 1

( asymptotically same as original recurrence )
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)

Expand, then set constants to 1

\[ W_{fastexp}(0) = 1 \]

\[ W_{fastexp}(1) = 1 \]

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

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

(asymptotically same as original recurrence)
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)

Expand, then set constants to 1

(asymptotically same as original recurrence)
is it faster?

```haskell
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)
```

Expand, then set constants to 1

\[ W_{fastexp}(0) = 1 \]

\[ W_{fastexp}(1) = 1 \]

\[ W_{fastexp}(n) = W_{fastexp}(n \div 2) + 1 \quad \text{for } n > 1 \]

(asymptotically same as original recurrence)
approx solution

- $\mathcal{W}_{\text{fastexp}}(n)$ is defined like $\log_2(n)$

  $$\log_2 n = \begin{cases} 
  0 & \text{if } n=1 \\
  \log_2 (n \div 2) + 1 & \text{else}
  \end{cases}$$

  $$\mathcal{W}_{\text{fastexp}}(n) = \begin{cases} 
  1 & \text{if } n<2 \\
  \mathcal{W}_{\text{fastexp}}(n \div 2) + 1 & \text{else}
  \end{cases}$$

- It follows that $\mathcal{W}_{\text{fastexp}}(n)$ is $O(\log n)$
exercise

• Using ML, discover the relationship between the functions

\[ \text{fun } \log n = \begin{cases} 0 & \text{if } n=1 \\ 1 + \log(n \text{ div } 2) & \text{else} \end{cases} \]

\[ \text{fun } W n = \begin{cases} 1 & \text{if } n<2 \\ 1 + W(n \text{ div } 2) & \text{else} \end{cases} \]

(see previous slide)
it’s really faster

- Work of $\text{exp}(n)$ 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 asymptotically faster than $\text{exp}$
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

Runtime of rev(L) depends on length of L but not the contents of L

length(rev L)= length(L)
work of rev

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

Let $W_{rev}(n)$ be work of $rev L$ when length $L = n$

$W_{rev}(0) = 1$

$W_{rev}(n) = W_{rev}(n-1) + (n-1) + 1$ for $n>0$
work of rev

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

Let $W_{rev}(n)$ be work of rev L when length $L = n$

$W_{rev}(0) = 1$

$W_{rev}(n) =$ for $n>0$
work of rev

\[
\text{fun } \text{rev } [ ] = [ ] \\
| \text{rev } (x::L) = (\text{rev } L) @ [x]
\]

Let \( W_{\text{rev}}(n) \) be work of \( \text{rev } L \) when length \( L = n \)

\[
W_{\text{rev}}(0) = 1 \\
W_{\text{rev}}(n) = W_{\text{rev}}(n-1) + n \quad \text{for } n > 0
\]
work of rev

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

Let $W_{\text{rev}}(n)$ be work of rev $L$ when length $L = n$

$W_{\text{rev}}(0) = 1$

$W_{\text{rev}}(n) = W_{\text{rev}}(n-1) + n$ for $n > 0$

$= W_{\text{rev}}(n-2) + (n-1) + n$
work of rev

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

Let \( W_{\text{rev}}(n) \) be work of rev \( L \) when length \( L = n \)

\[
W_{\text{rev}}(0) = 1
\]

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

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

\[
= 1 + 2 + \ldots + (n-1) + n
\]
work of rev

\begin{align*}
\text{fun } \text{rev} \ [\ ] &= \ [\ ] \\
| \ \text{rev} \ (x::\text{L}) &= (\text{rev} \ \text{L}) \ @ \ [x]
\end{align*}

Let $W_{\text{rev}}(n)$ be work of $\text{rev} \ \text{L}$ when length $\text{L} = n$

$W_{\text{rev}}(0) = 1$

$W_{\text{rev}}(n) = W_{\text{rev}}(n-1) + n$ for $n > 0$

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

\[
= 1 + 2 + \ldots + (n-1) + n
\]

$W_{\text{rev}}(n)$ is $O(n^2)$
work of rev

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

Let $W_{rev}(n)$ be work of $\text{rev } L$ when length $L = n$

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

$= W_{rev}(n-2) + (n-1) + n$

$= 1 + 2 + \ldots + (n-1) + n$

$W_{rev}(n)$ is $O(n^2)$ SLOW!
faster rev

**Surely** \( O(n) \) should be feasible...

- Use an extra argument to *accumulate* the reversed list

  \[
  \text{revver : int list } \ast \text{ int list } \rightarrow \text{ int list}
  \]

- Instead of *append* after the recursive call, do a *cons* before the recursive call

\[
\text{fun revver([ ], } A) = A \\
| \text{revver(x::L, A) = revver(L, x::A)}
\]
faster rev

**Surely** $O(n)$ should be feasible…

- Use an extra argument to accumulate the reversed list.
- Instead of `append` after the recursive call, do a `cons` before the recursive call.

```plaintext
fun revver([], A) = A  
|  revver(x::L, A) = revver(L, x::A)
```

`revver : int list * int list -> int list`

Surely $O(n)$ should be feasible…

Yes you can do it in $O(n)$. And don’t call me Shirley.
faster rev

fun revver([ ], A) = A
  | revver(x::L, A) = revver(L, x::A)

fun Rev L = revver(L, [ ])

For all L, A, \( \text{revver}(L, A) = (\text{rev } L) \odot A \)

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

Explain why \( W_{\text{Rev}}(n) \) is \( \Omega(n) \)

Hint: analyze \( W(\text{revver } (L, A)) \)
even more faster?

- The definition of \texttt{fastexp} relies on
  \[
  2^n = (2^{n \div 2})^2 \quad \text{if } n \text{ is even}
  \]
  \[
  2^n = 2 \times (2^{n-1}) \quad \text{if } n \text{ is odd}
  \]

- A moment’s thought tells us that
  \[
  2^n = 2 \times (2^{(n \div 2)})^2 \quad \text{if } n \text{ is odd}
  \]

Let’s define
\[
\text{pow} : \text{int} -> \text{int}
\]

based on this idea…
fun pow (n:int):int =
    case n of
    0 => 1
| 1 => 2
| _ => let
    val k = pow(n div 2)
    in
    if n mod 2 = 0 then k*k else 2*k*k
    end
work of $\text{pow}(n)$

\[
\begin{align*}
W_{\text{pow}}(0) &= 1 \\
W_{\text{pow}}(1) &= 1 \\
W_{\text{pow}}(n) &= 1 + W_{\text{pow}}(n \text{ div } 2) \quad \text{for } n > 1
\end{align*}
\]

Same recurrence as $W_{\text{fastexp}}$

Same asymptotic behavior

$\text{pow}(n)$ is $O(\log n)$
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 $\text{badpow}(n)$

$\text{badpow}(0) = 1$

$\text{badpow}(1) = 1$

$\text{badpow}(n) = 1 + 2 \text{badpow}(n \div 2)$

for $n > 1$

- This implies that $\text{badpow}(n)$ is $O(n)$

But $\text{pow}(n)$ is $O(\log n)$ (faster!)

Bad code design leads to poor performance
summary

Use recurrences for work/span

• recurrence form *mimics* function syntax

• OK to be sloppy with *additive* constants
  • let $c = 1$, or add/subtract 1

Asymptotic estimates are *robust*

• independent of architecture

• give information about *scaling*
exercise

• Recall the functions

\[
\begin{align*}
\text{isqrt}_0 : \text{int} & \rightarrow \text{int} \\
\text{isqrt}_1 : \text{int} & \rightarrow \text{int} \\
\text{isqrt}_2 : \text{int} & \rightarrow \text{int}
\end{align*}
\]

• Figure out the asymptotic work for

\[
\begin{align*}
\text{isqrt}_0 n & \\
\text{isqrt}_1 n & \\
\text{isqrt}_2 n
\end{align*}
\]

using recurrences

Try them out on large values of \( n \) and see the differences!
fun isqrt_0 (n : int) : int = 
  if n=0 then 0 else
  let
    fun loop i = if n < i*i then i-1 else loop(i+1)
  in
    loop 1
  end

• \( W_{isqrt\_0}(0) = 1 \)

• \( W_{isqrt\_0}(n) = W_{loop}(1) \) for \( n>0 \)

How can this be? RHS doesn’t seem to use \( n \)
isqrt_0

- The `loop` function used by `isqrt_0(n)` does use the value of `n`
  ```
  fun loop i = if n < i*i then i-1 else loop(i+1)
  ```

- Let `k` be the integer square root of `n`, so
  \[ l^2 \leq 2^2 \leq \ldots \leq k^2 \leq n < (k+1)^2 \]
  \[
  W_{loop}(i) = 1 + W_{loop}(i+1) \quad \text{for } i=1, \ldots, k
  \]
  \[
  W_{loop}(k+1) = 1
  \]
  Hence \( W_{loop}(1) \) is \( O(k) \)
The loop function used by \texttt{isqrt\_0}(n) does use the value of \texttt{n}

\begin{verbatim}
fun loop i = if n < i*i then i-1 else loop(i+1)
\end{verbatim}

Let \texttt{k} be the integer square root of \texttt{n}, so

\[1^2 \leq 2^2 \leq \ldots \leq k^2 \leq n < (k+1)^2\]

\[W_{\text{loop}}(i) = 1 + W_{\text{loop}}(i+1) \quad \text{for } i = 1, \ldots, k\]

\[W_{\text{loop}}(k+1) = 1\]

Hence \[W_{\text{loop}}(1)\] is \(O(k)\)

So \[W_{\text{isqrt\_0}}(n)\] is \(O(\sqrt{n})\)
\textbf{isqrt\_l}

\begin{verbatim}
fun isqrt\_l(n) = 
  if n=0 then 0 else
  let
    val r = isqrt\_l(n - 1) + 1
  in
    if n<r\^r then r-1 else r
  end
\end{verbatim}

- \(W_{isqrt\_l}(0) = 1\)
- \(W_{isqrt\_l}(n) = 1 + W_{isqrt\_l}(n - 1)\) for \(n > 0\)

\(W_{isqrt\_l}(n)\) is \(O(n)\)
isqrt_1

fun isqrt_1(n) =
    if n=0 then 0 else
    let
        val r = isqrt_1(n - 1) + 1
    in
        if n<r*r then r-1 else r
    end

• \( W_{isqrt_1}(0) = 1 \)

• \( W_{isqrt_1}(n) = 1 + W_{isqrt_1}(n - 1) \) for \( n > 0 \)

\[ W_{isqrt_1}(n) \text{ is } O(n) \]
**isqrt_2**

\[
\text{fun } \text{isqrt}_2(n) = \\
\text{if } n=0 \text{ then 0 else} \\
\text{let} \\
\quad \text{val } r = 2 \times \text{isqrt}_2(n \div 4) + 1 \\
\text{in} \\
\quad \text{if } n<r^2 \text{ then } r-1 \text{ else } r \\
\text{end}
\]

- \( W_{\text{isqrt}_2}(0) = 1 \)

- \( W_{\text{isqrt}_2}(n) = 1 + W_{\text{isqrt}_2}(n \div 4) \) for \( n > 0 \)

\( W_{\text{isqrt}_2}(n) \) is \( \Theta(\log n) \)
\texttt{isqrt\_2}

\begin{verbatim}
fun isqrt\_2(n) = 
  if n=0 then 0 else 
  let 
    val r = 2 * isqrt\_2(n div 4) + 1 
  in 
    if n<r*r then r-1 else r 
  end 
end
\end{verbatim}

- \( W_{\text{isqrt\_2}}(0) = 1 \)

- \( W_{\text{isqrt\_2}}(n) = 1 + W_{\text{isqrt\_2}}(n \text{ div } 4) \) for \( n > 0 \)

\[ W_{\text{isqrt\_2}}(n) \text{ is } O(\log n) \]
summary

- Asymptotic work analysis “explains” runtime experience

\[ \text{isqrt}_0 \ 123456789 \quad \text{fast} \]
\[ \text{isqrt}_1 \ 123456789 \quad \text{slowest} \]
\[ \text{isqrt}_2 \ 123456789 \quad \text{fastest} \]

\[ O(\log n) \subset O(\sqrt{n}) \subset O(n) \]