15-150 Fall 2023

Stephen Brookes

Lecture 5

Work and span
Efficiency analysis

Today

- **Work and span**
  - sequential and parallel runtime
- **Recurrences**
  - *exact* and *asymptotic* solutions
- **Designing efficient code**

what matters

- **Correctness**

  TYPE $f : t_1 \text{ -> } t_2$
  REQUIRES x such that …
  ENSURES $f\ x \Longrightarrow^{*} v$ such that …

- **Efficiency**

  Information about *evaluation time*

  $f\ x \Longrightarrow^{(h(x)\ \text{steps})} v$

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

  An ***asymptotic*** estimate is good enough!

  - $h(x)$ is $\mathbf{O}(g(\text{size } x))$ for some notion of size

asymptotic

- We want to estimate the *runtime* $W_f(n)$ for evaluating $f(n)$, for ***large*** n

  - basic operations take ***constant time***

- We will give a **big-O** classification

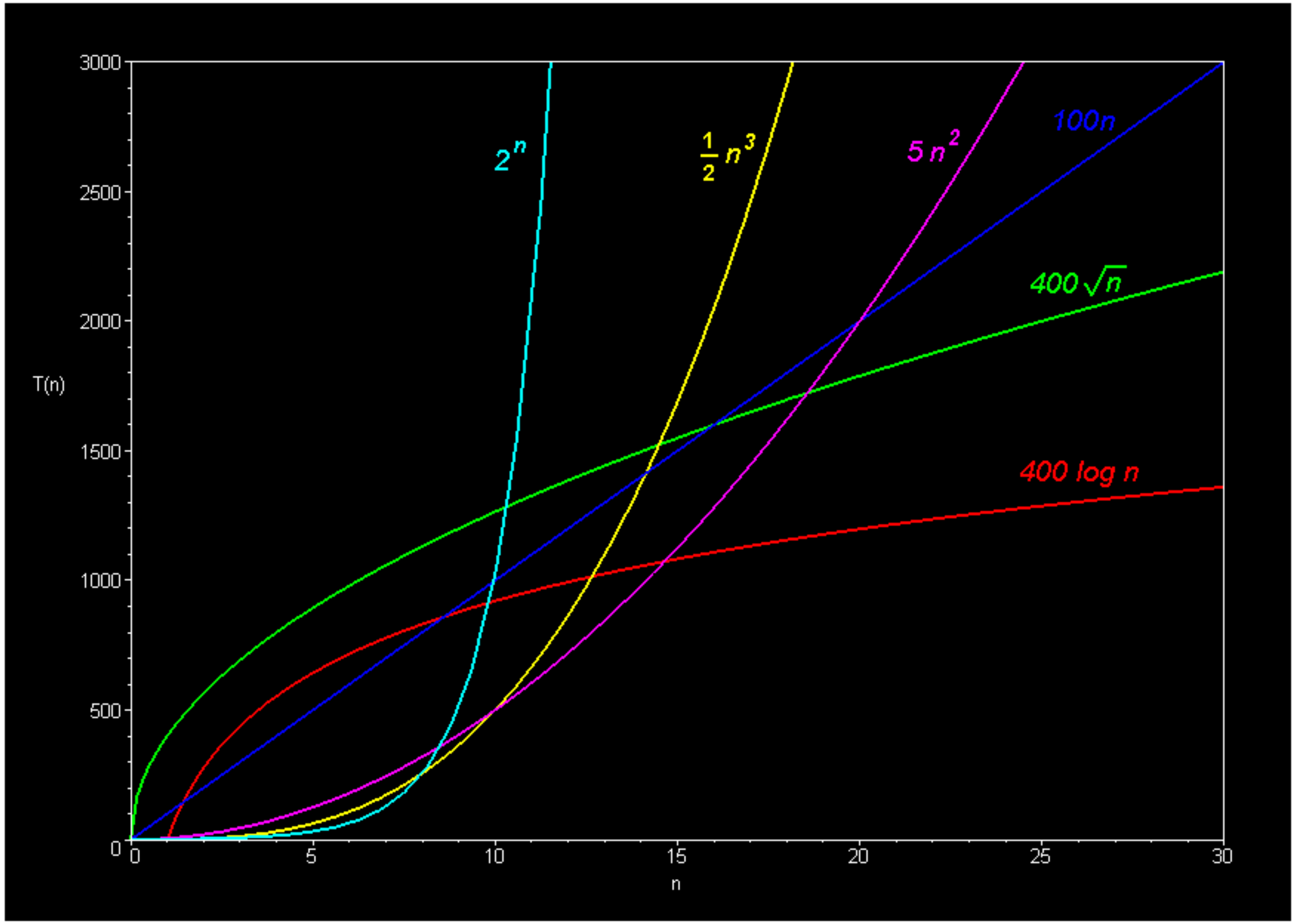
$W_f(n)$ is $O(g(n))$

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)



isqrt_1

isqrt_0

isqrt_2

we will see this later

asymptotically

- ***Ignore*** additive constants

  $n^5+1000000$ is $O(n^5)$

- ***Absorb*** multiplicative constants

  $1000000n^5$ is $O(n^5)$

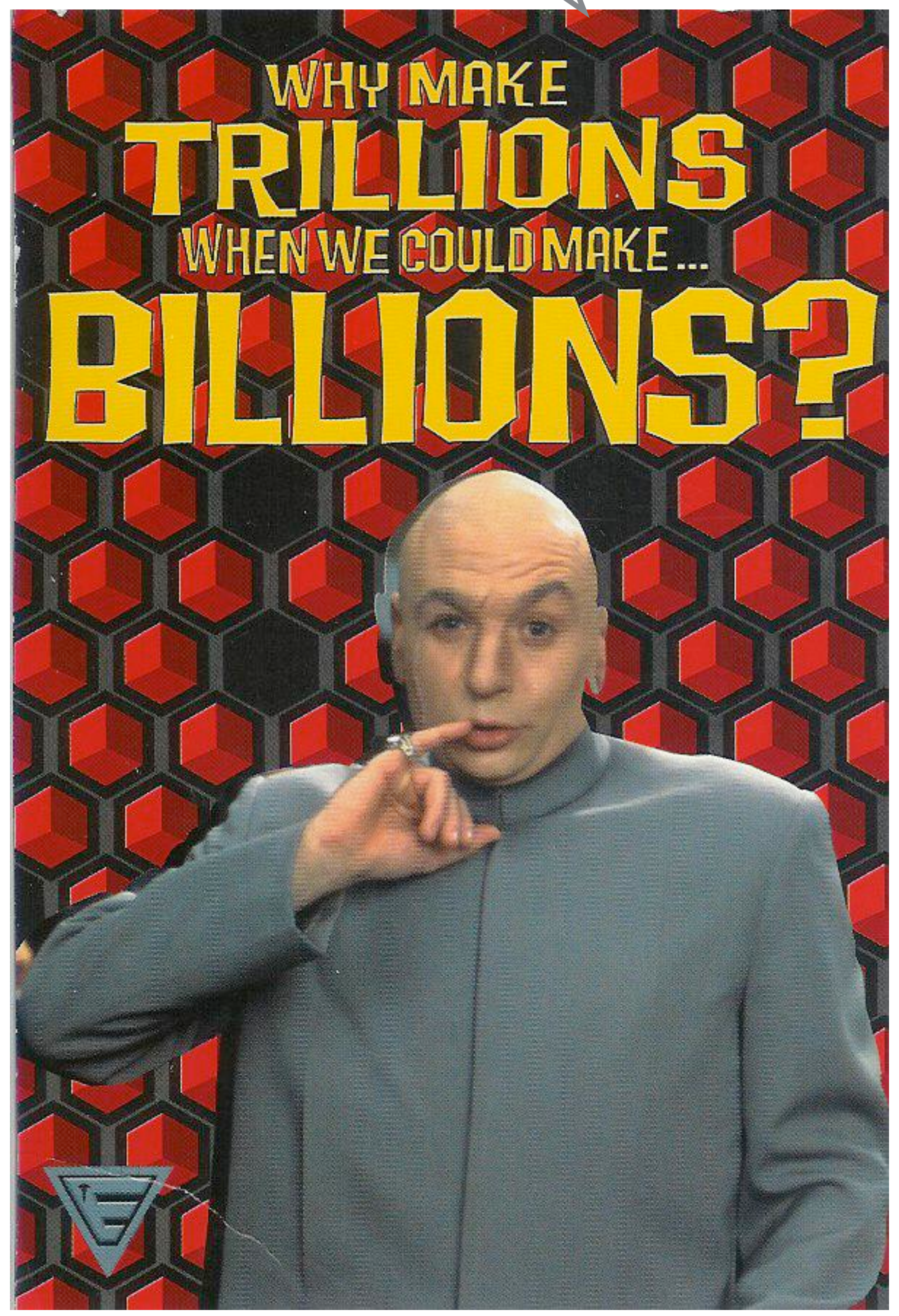
- Be ***accurate***

  $O(n^2) \subset O(n^3) \subset O(n^4)$

- Use common terminology

  constant time, logarithmic, linear, quadratic, polynomial, exponential

$O(10^9) = O(10^6)$



recurrences

To calculate the applicative work or span for a *recursive function* we can use a ***recurrence relation***.

```
fun fact(n:int) : int =
   if n=0 then 1 else n * fact(n-1)
```

$$W_{fact}(0) = c_0$$
$$W_{fact}(n) = c_1 + W_{fact}(n-1) \quad n > 0$$

for some constants $c_0, c_1 > 0$

recurrences

$$W_f(n) = k * W_f(n-1) + c$$

where $k > 1$, $c > 0$ are constants

- Additive constants ***don't*** matter

  $W_f(n)$ is $O(k^n)$ for ANY choice of c

  WLOG let $c = 1$

- Multiplicative constants ***do*** matter

  $W_f(n)$ is $O(k^n)$ … depends on k

  $O(2^n)$ is not the same as $O(3^n)$

work

number of steps

- $W(e)$, the *work* of e, is the time to evaluate e *sequentially*, on a single processor

- Given a function f and a notion of *size* we may want the *applicative work* $W_f(n)$
    - The work of $f\ v$, when v is a value of 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 operations 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

dependent

- **if** b **then** e_1 **else** e_2 — b *before* e_1 *or* e_2
- (**fn** x => e_2) e_1 — e_1 *before* $[\![x{:}v_1]\!]e_2$
- **let val** $x = e_1$ **in** e_2 **end** — e_1 *before* $[\![x{:}v_1]\!]e_2$

independent

- $(e_1, \ldots, e_n)$ — *tuple components*
- $e_1 + e_2$ — *summands*

work rules

$$W(\mathbf{n}) = 0$$

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

$$W((e_1, e_2)) = W\, e_1 + W\, e_2$$

$$W(e_1 @ e_2) = W\, e_1 + W\, e_2 + \text{length } e_1 + 1$$

$$W(\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2)$$
$$\leq W\, b + \max(W\, e_1 , W\, e_2) + 1$$

span rules

$S\ (\mathbf{n}) = 0$

$S\ (e_1 + e_2) = \max(S\ e_1, S\ e_2) + 1$

$S\ ((e_1, e_2)) = \max(S\ e_1, S\ e_2)$

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

$S\ (\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2)$
$\leq S\ b + \max(S\ e_1\ ,\ S\ e_2\ ) + 1$

work and evaluation

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

$$\boxed{\text{If } \; e \Longrightarrow^{(k)} v \; \text{ then } \; W(e) = k}$$

$$\begin{aligned}(2+2)+(2+2) &\Longrightarrow 4+(2+2)\\ &\Longrightarrow 4+4\\ &\Longrightarrow 8\end{aligned}$$

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

$$\boxed{W(e_1+e_2) = W(e_1) + W(e_2) + 1}$$

work and application

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

$(\textbf{fn}\ x \Rightarrow x{+}x)\ (2{+}2)$
$\Longrightarrow (\textbf{fn}\ x \Rightarrow x{+}x)\ 4$
$\Longrightarrow 4{+}4$
$\Longrightarrow 8$ (3 steps)

$W\ ((\textbf{fn}\ x \Rightarrow x{+}x)\ (2{+}2))$
$= 0 + 1 + W(4{+}4) + 1$
$= 0 + 1 + 1 + 1$
$= 3$

exp

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))

$\text{exp } 4 \Longrightarrow^{(1)} \text{M } 4$
$\Longrightarrow^{(5)} 2 * (\text{M } 3)$
$\Longrightarrow^{(5)} 2 * (2 * (\text{M } 2))$
$\Longrightarrow^{(5)} 2 * (2 * (2 * (\text{M } 1)))$
$\Longrightarrow^{(5)} 2 * (2 * (2 * (2 * (\text{M } 0))))$
$\Longrightarrow^{(3)} 2 * (2 * (2 * (2 * 1)))$
$\Longrightarrow^{(4)} 16$

$\text{M } 4 \Longrightarrow \text{if } 4{=}0 \text{ then} \ldots$
$\Longrightarrow \text{if false then} \ldots$
$\Longrightarrow 2 * \text{exp } (4-1)$
$\Longrightarrow 2 * \text{M } (4-1)$
$\Longrightarrow 2 * (\text{M } 3)$

$\text{M } 3 \Longrightarrow^{(5)} 2 * (\text{M } 2)$

$\text{exp } 4 \Longrightarrow^{(28)} 16$

exp

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

exp n $\Longrightarrow^{(6n+4)}$ 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)

$$\text{exp n} \Longrightarrow^{O(n)} 2^n$$

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

worst-case work, over all values of size n

$W_f(n)$ = 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?

A: *If the work of* f v *depends only on the size of* v (!)

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

exp

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

For $n \geq 0$, exp n $\Longrightarrow^* 2^n$

Let $W_{exp}(n)$ be the runtime for exp(n)

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

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

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

solution

- Easy to prove by induction on n that

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

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

The work for exp(n) is ***linear***

comment

If we'd simplified by letting constants be 1,

$W_{exp}(0) = 1$

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

we'd have gotten $W_{exp}(n) = 1 + n$

$W_{exp}(n)$ 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 exp function)

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

- The recurrence for the *span* of exp n

  will be *identical* to the recurrence we gave for work, with the same asymptotic solution… why?

No speed-up here using parallel evaluation!

a **faster** method?

- The definition of exp relies on the fact that

$$2^n = 2\ (2^{n-1}) \qquad \text{when } n > 0$$

- Everybody knows that

$$2^n = (2^{n \text{ div } 2})^2 \qquad \text{when } n \text{ is even}$$

Let's define

fastexp : int -> int

based on this idea…

fastexp

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

This code design leads to a recurrence…

$W_{fastexp}(0) = k_0$

$W_{fastexp}(n) = W_{fastexp}(n \text{ div } 2) + k_1$ for $n>0$, even

$W_{fastexp}(n) = W_{fastexp}(n-1) + k_2$ for $n>0$, 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

$W_{fastexp}(0) = 1$

$W_{fastexp}(1) = 1$

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

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

(asymptotically same as original recurrence)

approx solution

- $W_{fastexp}(n)$ *is defined like* $\log_2(n)$

  $\log_2 n =$
  if $n=1$ **then** 0 **else** $\log_2 (n \text{ div } 2) + 1$

  $W_{fastexp}(n) =$
  if $n<2$ **then** 1 **else** $W_{fastexp}(n \text{ div } 2) + 1$

- It follows that $W_{fastexp}(n)$ is $O(\log n)$

it's really faster

- Work of exp(n) is O(n)
- Work of fastexp(n) is O(log n)
- O(log n) is a proper subset of O(n)
- fastexp is *asymptotically faster* than exp

even more faster?

- The definition of fastexp relies on

$$2^n = (2^{n \text{ div } 2})^2 \qquad \text{if n is even}$$

$$2^n = 2\ (2^{n-1}) \qquad \text{if n is odd}$$

- A moment's thought tells us that

$$2^n = 2\ (2^{(n \text{ div } 2)})^2 \qquad \text{if n is odd}$$

Let's define

pow : int -> int

based on this idea…

pow

```
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 pow(n)

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

Same recurrence as $W_{fastexp}$

Same asymptotic behavior

pow(n) is $O(\log n)$

badpow

```
fun badpow (n:int):int =
   case n of
      0 => 1
   |  1 => 2
   |  _ => let
              val k = badpow(n div 2)*badpow(n div 2)
           in
              if n mod 2 = 0 then k else 2*k
           end
```

bad idea:
does same
recursive call
twice

work of badpow(n)

$W_{badpow}(0) = 1$
$W_{badpow}(1) = 1$
$W_{badpow}(n) = 1 + 2\, W_{badpow}(n \text{ div } 2)$
for $n>1$

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

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

Bad code design leads to poor performance

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 $n>0$

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

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

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

faster rev

Surely O(n) should be feasible…

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

  revver : int list * int list -> int list

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

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

faster rev

```
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

Explain why $W_{Rev}(n)$ is $O(n)$

Hint: analyze W(revver (L, A))

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

  isqrt_0 : int -> int
  isqrt_1 : int -> int
  isqrt_2 : int -> int

- Figure out the asymptotic work for

  isqrt_0 n
  isqrt_1 n
  isqrt_2 n

  using recurrences

  Try them out on large values of n and see the differences!

isqrt_0

```
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 $1^2 \le 2^2 \le \ldots \le k^2 \le n < (k+1)^2$

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

$$W_{loop}(k+1) = 1$$

Hence $W_{loop}(1)$ is $O(k)$

So $W_{isqrt\_0}(n)$ is $O(\sqrt{n})$

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 $n > 0$

$$\boxed{W_{isqrt\_1}(n) \quad \text{is } O(n)}$$

isqrt_2

```
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
```

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

$$\boxed{W_{isqrt\_2}(n) \quad \text{is } O(\log n)}$$

summary

- Asymptotic work analysis
  "explains" runtime experience

isqrt_0	123456789	*fast*
isqrt_1	123456789	*slowest*
isqrt_2	123456789	*fastest*

$$O(\log n) \subset O(\sqrt{n}) \subset O(n)$$

isqrt_2 isqrt_0 isqrt_1