CSE 421: Intro Algorithms

2: Analysis

Summer 2011

Larry Ruzzo

Defining Efficiency

"Runs fast on typical real problem instances"

Pro:

sensible, bottom-line-oriented

Con:

moving target (diff computers, compilers, Moore's law)

highly subjective (how fast is "fast"? What's "typical"?)

Efficiency

Our correct TSP algorithm was incredibly slow

Basically slow no matter what computer you have

We want a general theory of "efficiency" that is

- Simple
- Objective
- Relatively independent of changing technology
- But still predictive – "theoretically bad" algorithms should be bad in practice and vice versa (usually)

Measuring efficiency

Time ≈ # of instructions executed in a simple programming language

- only simple operations (+,*,-,=,if,call,…)
- each operation takes one time step
- each memory access takes one time step
- no fancy stuff (add these two matrices, copy this long string,…) built in; write it/charge for it as above

No fixed bound on the memory size

We left out things but...

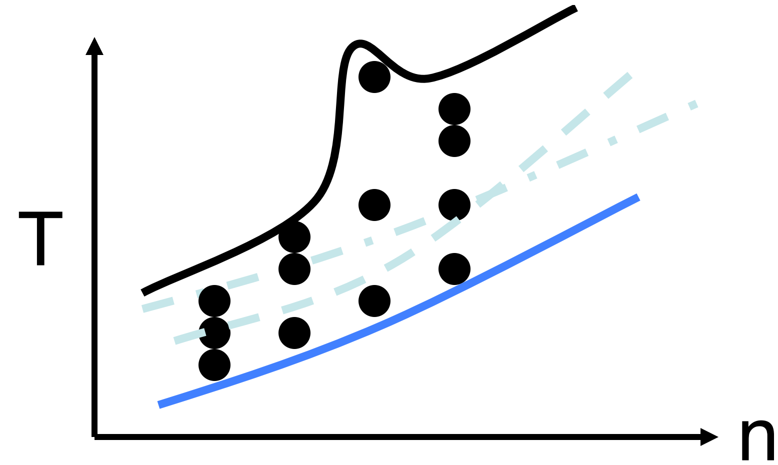
Things we've dropped

- memory hierarchy
  - disk, caches, registers have many orders of magnitude differences in access time
- not all instructions take the same time in practice (+, ÷)
- different computers have different primitive instructions

However,

- the "RAM" model is useful for designing algorithms and measuring their efficiency
- one can usually tune implementations so that the hierarchy, etc., is not a huge factor

Complexity analysis



Problem size n

- Best-case complexity: min # steps algorithm takes on any input of size n
- Average-case complexity: avg # steps algorithm takes on inputs of size n
- Worst-case complexity: max # steps algorithm takes on any input of size n

Pros and cons:

Best-case

- unrealistic oversell

Average-case

- over what probability distribution? (different people may have different "average" problems)
- analysis often hard

Worst-case

+ a fast algorithm has a comforting guarantee
- may be too pessimistic

Why Worst-Case Analysis?

Appropriate for time-critical applications

    E.g. avionics, nuclear reactors

Unlike Average-Case, no debate about what the right definition is

    If worst $\gg$ average, then (a) alg is doing something pretty subtle, & (b) are hard instances really that rare?

Analysis often easier

Result is often representative of "typical" problem instances

Of course there are exceptions…

General Goals

Characterize *growth rate* of (worst-case) run time as a function of problem size, up to a constant factor

Why not try to be more precise?

- Technological variations (computer, compiler, OS, …) easily 10x or more
- Being more precise is a ton of work
- A key question is "scale up": if I can afford to do it today, how much longer will it take when my business problems are twice as large? (E.g., today: cn^2, next year: $c(2n)^2 = 4cn^2$: 4 x longer.)

Complexity

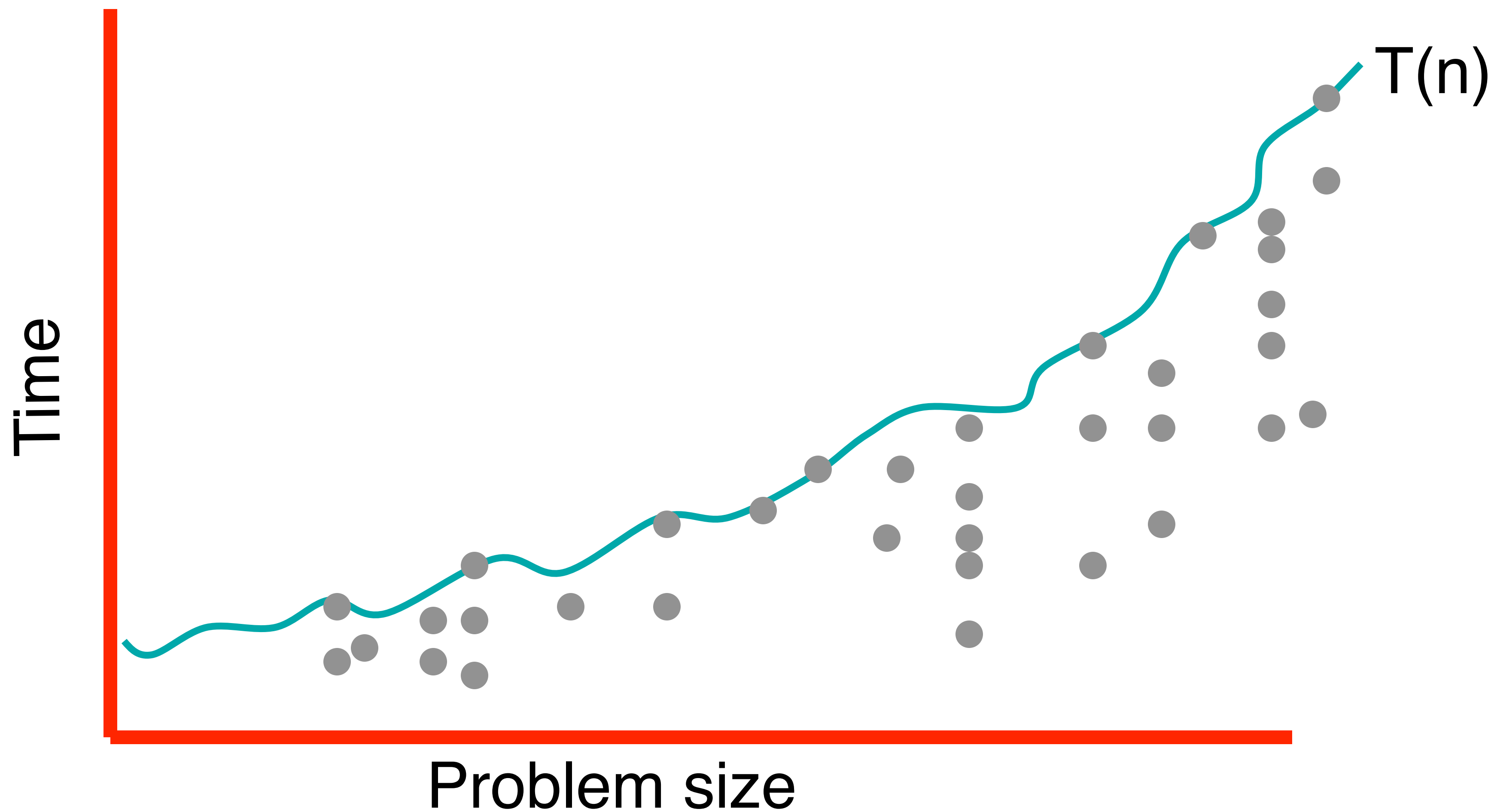
The *complexity* of an algorithm associates a number $T(n)$, the worst-case time the algorithm takes on problems of size n, with each problem size n.

Mathematically,

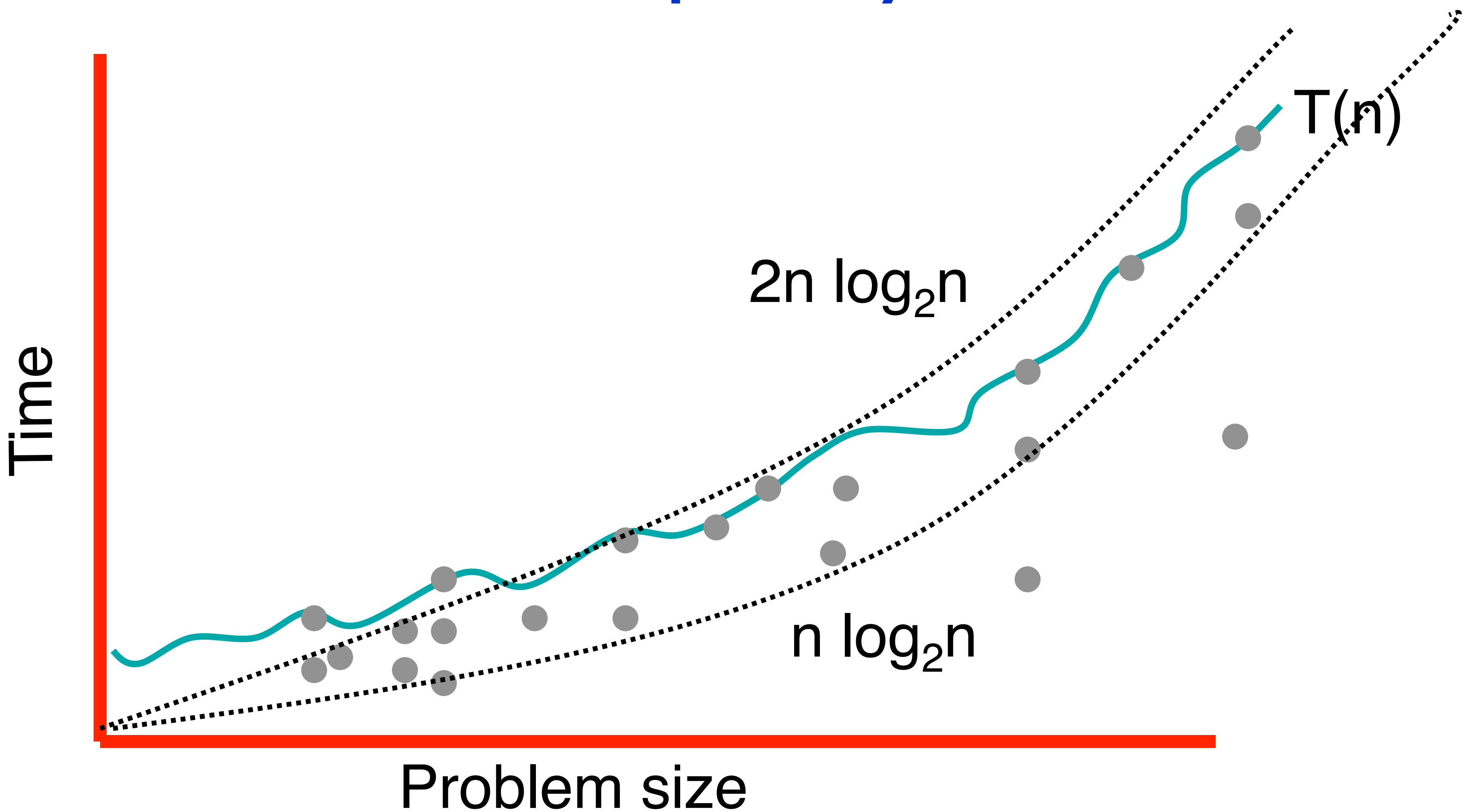
$T: N+ \rightarrow R+$

I.e., T is a function that maps positive integers (problem sizes) to positive real numbers (number of steps).

Complexity



Complexity



O-notation, etc.

Given two functions f and $g: N \to R$

f(n) is O(g(n)) iff there is a constant $c>0$ so that

f(n) is eventually always $\leq c\, g(n)$

f(n) is Ω (g(n)) iff there is a constant $c>0$ so that

f(n) is eventually always $\geq c\, g(n)$

f(n) is Θ (g(n)) iff there is are constants c_1, $c_2>0$ so that

eventually always $c_1 g(n) \leq f(n) \leq c_2 g(n)$

Examples

$10n^2$-$16n+100$ is $O(n^2)$ also $O(n^3)$

$10n^2-16n+100 \le 11n^2$ for all $n \ge 10$

$10n^2$-$16n+100$ is $\Omega\ (n^2)$ also $\Omega\ (n)$

$10n^2-16n+100 \ge 9n^2$ for all $n \ge 16$

Therefore also $10n^2-16n+100$ is $\Theta\ (n^2)$

$10n^2$-$16n+100$ is not $O(n)$ also not $\Omega\ (n^3)$

Properties

Transitivity.

If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.

If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.

If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.

If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.

If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.

If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.

Asymptotic Bounds for Some Common Functions

Polynomials:

$a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$

Logarithms:

$O(\log_a n) = O(\log_b n)$ for any constants $a,b > 0$

Logarithms:

For all $x > 0$, $\log n = O(n^x)$

log grows slower than every polynomial

"One-Way Equalities"

2 + 2 is 4	$2n^2 + 5\ n$ is $O(n^3)$
2 + 2 = 4	$2n^2 + 5\ n = O(n^3)$
4 = 2 + 2	$\cancel{O(n^3) = 2n^2 + 5\ n}$
All dogs are mammals	~~All mammals are dogs~~

Bottom line:

OK to put big-O in R.H.S. of equality, but not left.

[Better, but uncommon, notation: $T(n) \in O(f(n))$.]

Working with O-Ω-Θ notation

Claim: For any a, and any b>0, $(n+a)^b$ is $\Theta(n^b)$

$(n+a)^b \le (2n)^b$ for $n \ge |a|$
$= 2^b n^b$
$= cn^b$ for $c = 2^b$

so $(n+a)^b$ is $O(n^b)$

$(n+a)^b \ge (n/2)^b$ for $n \ge 2|a|$ (even if $a < 0$)
$= 2^{-b} n^b$
$= c'n$ for $c' = 2^{-b}$

so $(n+a)^b$ is $\Omega(n^b)$

Working with O-Ω-Θ notation

Claim: For any a, b>1 $\log_a n$ is Θ ($\log_b n$)

$$\log_a b = x \text{ means } a^x = b$$

$$a^{\log_a b} = b$$

$$(a^{\log_a b})^{\log_b n} = b^{\log_b n} = n$$

$$(\log_a b)(\log_b n) = \log_a n$$

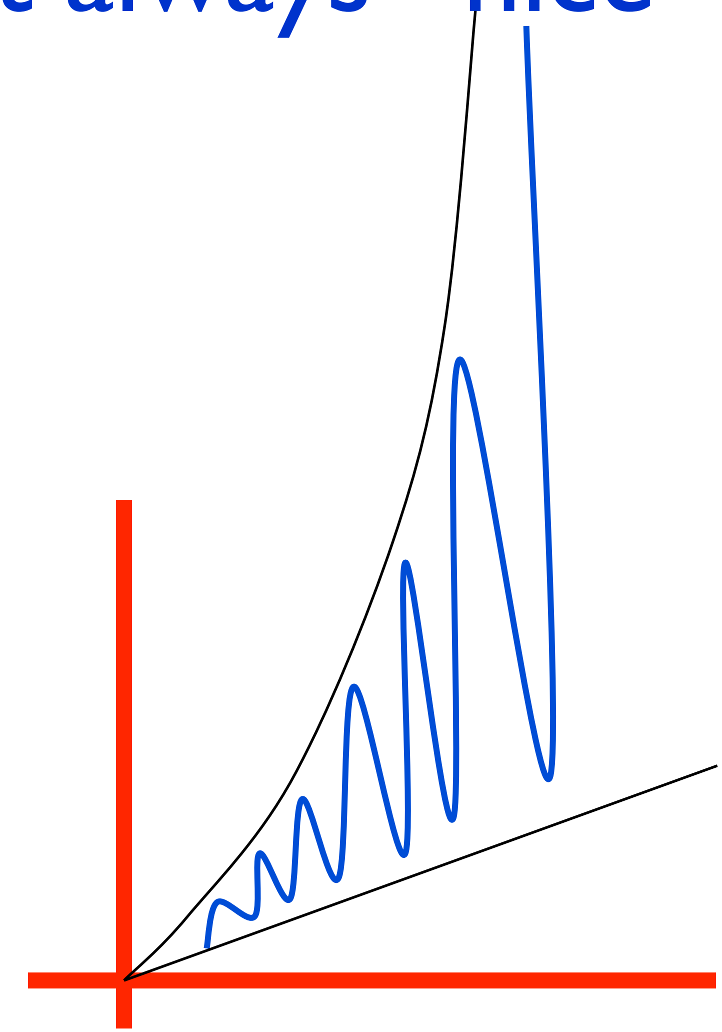
$$c \log_b n = \log_a n \text{ for the constant c} = \log_a b$$

$$\text{So}:$$

$$\log_b n = \Theta(\log_a n) = \Theta(\log n)$$

Big-Theta, etc. not always "nice"

$$f(n) = \begin{cases} n^2, & n \text{ even} \\ n, & n \text{ odd} \end{cases}$$



$f(n) \neq \Theta(n^a)$ for any a.

Fortunately, such nasty cases are rare

$f(n \log n) \neq \Theta(n^a)$ for any *a, either, but at least it's simpler*.

A Possible Misunderstanding?

We have looked at

- type of complexity analysis
  - worst-, best-, average-case
- types of function bounds
  - O, Ω, Θ

These two considerations are independent of each other

- one can do any type of function bound with any type of complexity analysis - measuring different things with same yardstick

Insertion Sort:

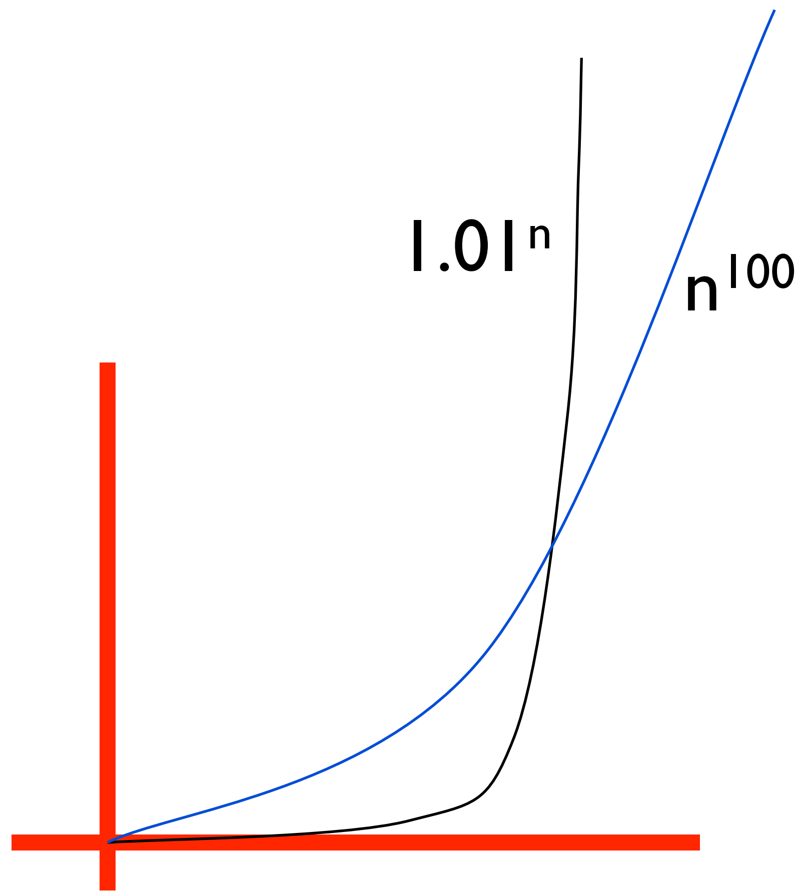
$\Omega(n^2)$ (worst case)

$O(n)$ (best case)

Asymptotic Bounds for Some Common Functions

Exponentials.
For all $r > 1$
and all $d > 0$,
$n^d = O(r^n)$.

every exponential grows faster than every polynomial



Polynomial time

$\boxed{\text{P: Running time is } O(n^d) \text{ for some constant d independent of the input size n.}}$

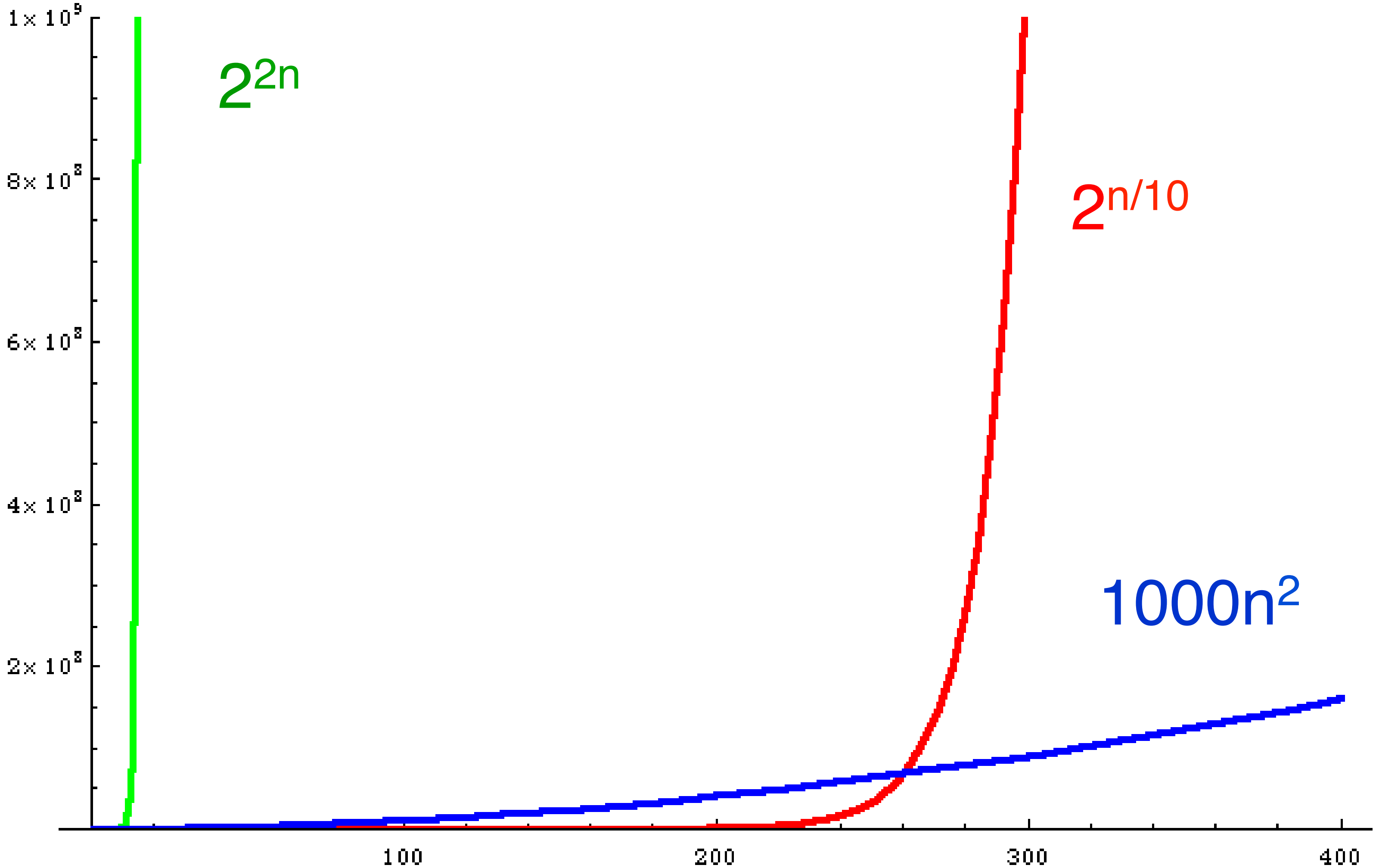
Nice scaling property: there is a constant c s.t. *doubling* n, time increases only by a factor of c.

(E.g., $c \sim 2^d$)

Contrast with exponential: For any constant c, there is a d such that $n \to n+d$ increases time by a factor of more than c.

(E.g., 2^n vs 2^{n+1})

polynomial vs exponential growth



Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

not only get very big, but do so *abruptly*, which likely yields erratic performance on small instances

another view of poly vs exp

Next year's computer will be 2x faster. If I can solve problem of size n_0 today, how large a problem can I solve in the same time next year?

Complexity	Increase	E.g. $T=10^{12}$		
$O(n)$	$n_0 \to 2n_0$	10^{12}	$\to$	2×10^{12}
$O(n^2)$	$n_0 \to \sqrt{2}\, n_0$	10^6	$\to$	1.4×10^6
$O(n^3)$	$n_0 \to \sqrt[3]{2}\, n_0$	10^4	$\to$	1.25×10^4
$2^{n/10}$	$n_0 \to n_0+10$	400	$\to$	410
2^n	$n_0 \to n_0+1$	40	$\to$	41

Domination

f(n) is o(g(n)) iff $\lim_{n\to\infty} f(n)/g(n)=0$

that is g(n) *dominates* f(n)

If $a \leq b$ then n^a is $O(n^b)$

If $a < b$ then n^a is $o(n^b)$

Note:
if f(n) is Θ (g(n)) then it cannot be o(g(n))

Working with little-o

$n^2 = o(n^3)$ [Use algebra]:

$$\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$$

$n^3 = o(e^n)$ [Use L'Hospital's rule 3 times]:

$$\lim_{n\to\infty} \frac{n^3}{e^n} = \lim_{n\to\infty} \frac{3n^2}{e^n} = \lim_{n\to\infty} \frac{6n}{e^n} = \lim_{n\to\infty} \frac{6}{e^n} = 0$$

Summary

Typical initial goal for algorithm analysis is to find a

- reasonably tight ← i.e., Θ if possible
- *asymptotic* ← i.e., O or Θ
- bound on ← usually upper bound
- *worst case* running time
- as a function of problem *size*

This is rarely the last word, but often helps separate good algorithms from blatantly poor ones - so you can concentrate on the good ones!

why “polynomial”?

Point is not that n^{2000} is a nice time bound, or that the differences among n and 2n and n^2 are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials, so more amenable to theoretical analysis.

- “My problem is in P” is a starting point for a more detailed analysis
- “My problem is not in P” may suggest that you need to shift to a more tractable variant, or otherwise readjust expectations