Analysis

- How to reason about the performance of algorithms

Defining Efficiency

"Runs fast on typical real problem instances"

Pro:

sensible, bottom-line-oriented

Con:

moving target (diff computers, compilers)

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

Efficiency

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

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

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 (+, ÷)
- communication
- different computers have different primitive instructions

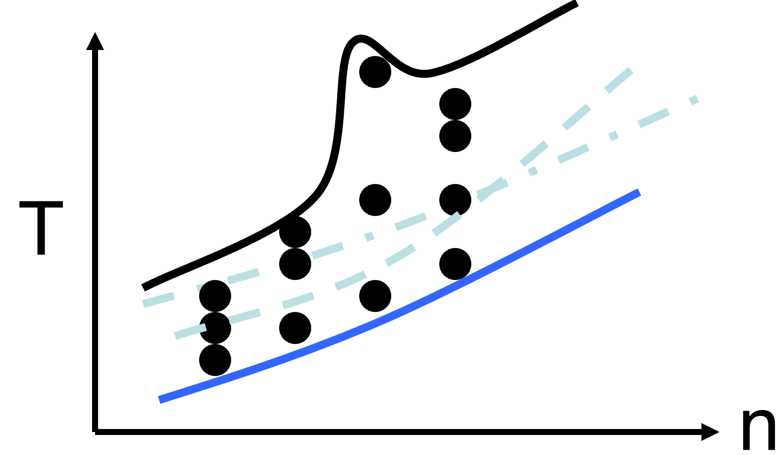
However,

- one can usually tune implementations so that the hierarchy, etc., is not a huge factor

Problem

- Algorithms can have different running times on different inputs!
- Smaller inputs take less time, larger inputs take more time.

Solution



Measure performance on problem 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:

Average-case

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

Worst-case

+ a fast algorithm has a comforting guarantee
+ analysis easier
+ useful in real-time applications (space shuttle, nuclear reactors)
- may be too pessimistic

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

Complexity

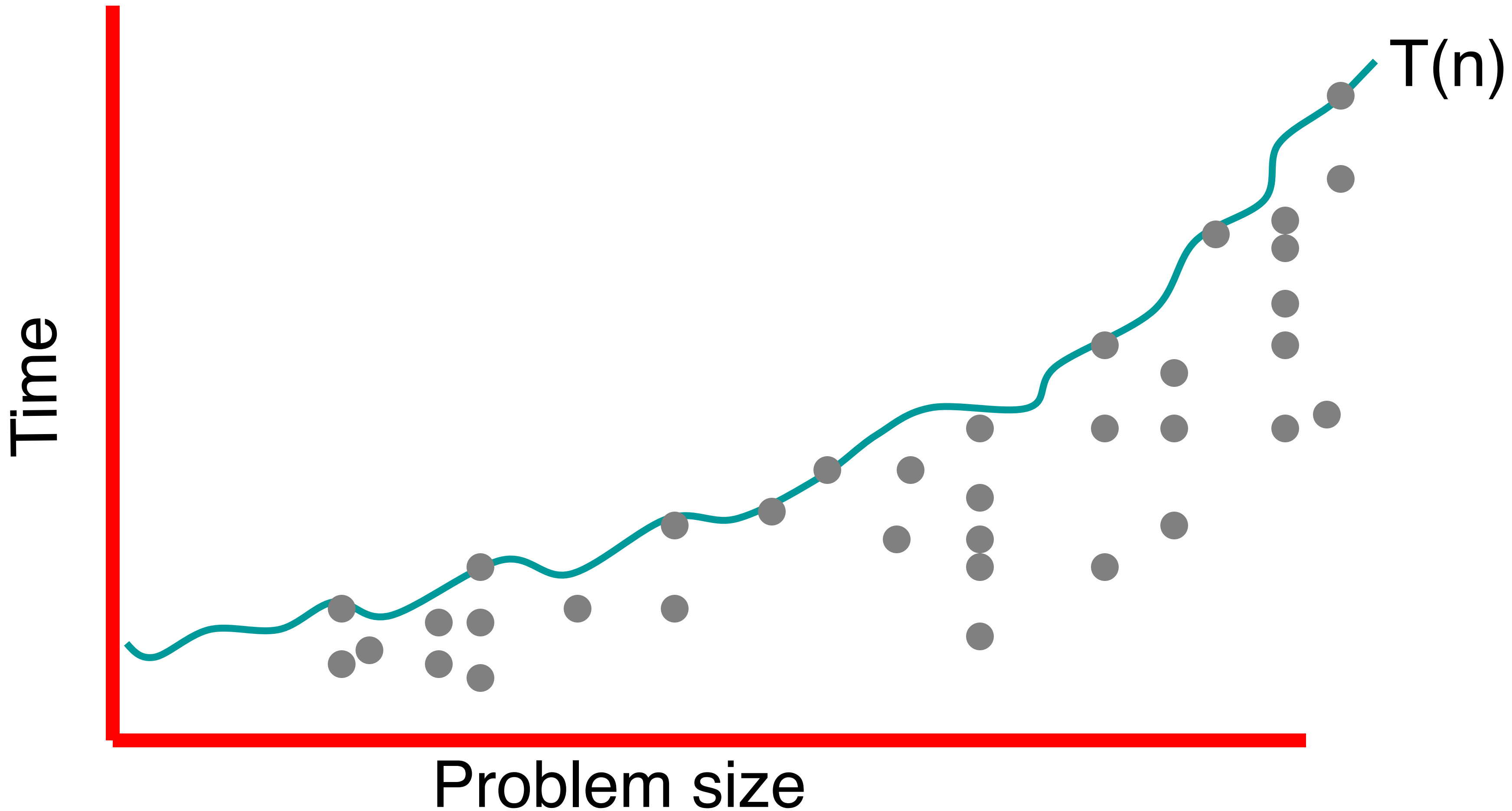
The *complexity* of an algorithm associates a number $T(n)$, the worst-case time the algorithm takes on problems of size n, with each problem size n.

Mathematically,

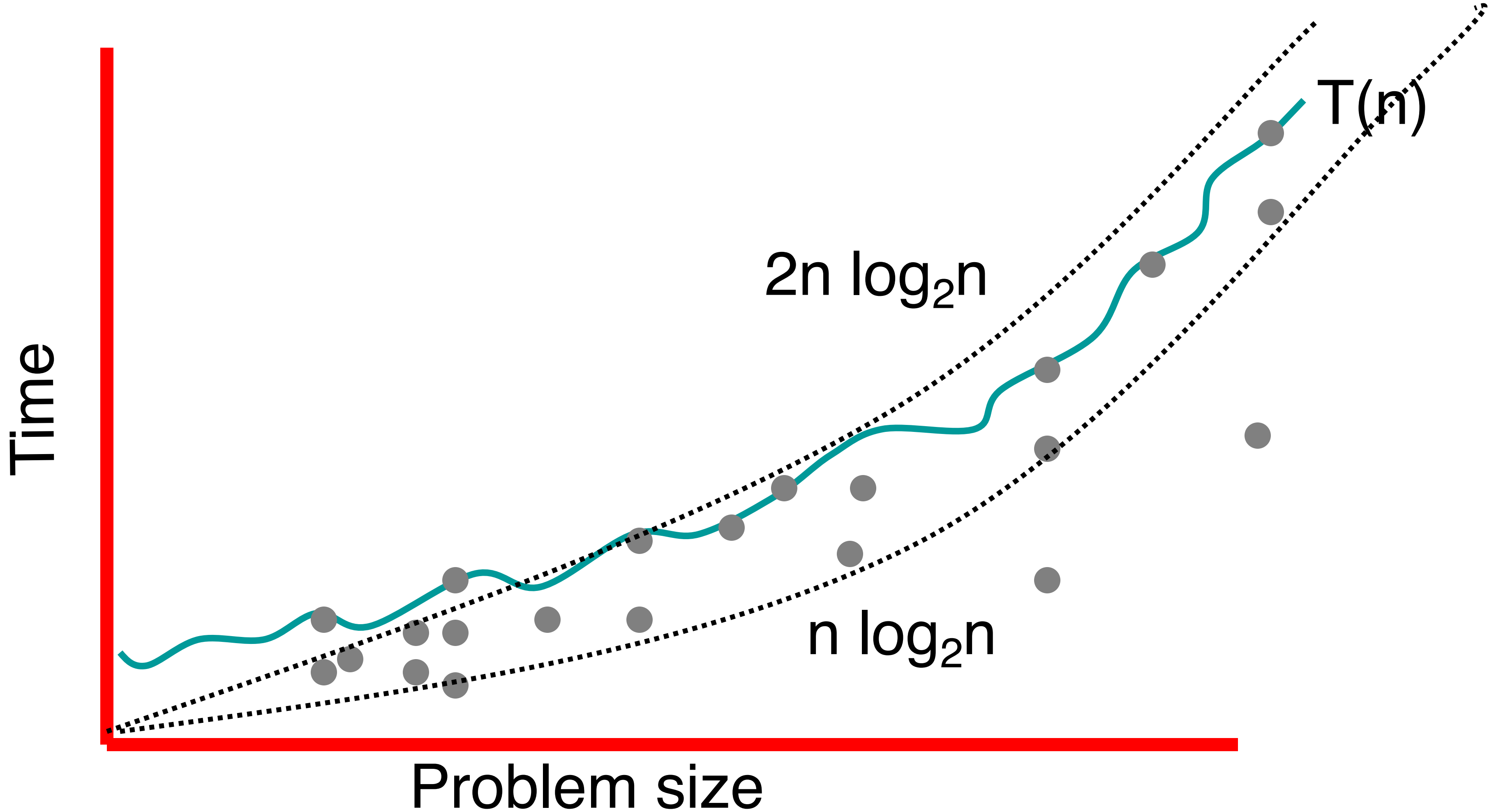
$T: N^+ \text{->} R^+$

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->R

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

f(n) is $\Omega(g(n))$ iff there is a constant $c>0$ so that
f(n) is eventually always > c g(n)

f(n) is $\Theta(g(n))$ iff there are constants c_1, $c_2>0$ so that
eventually always $c_1 g(n) < f(n) < c_2 g(n)$

Examples

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

$10n^2-16n+100 < 10n^2$ for all $n > 10$

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

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

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 $g = \Theta(h)$ 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:

$\log_a n = \Theta(\log_b n)$ for any constants $a,b > 1$

Logarithms:

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

log grows slower than every polynomial

"One-Way Equalities"

2 + 2 is 4

2 + 2 = 4

4 = 2 + 2

$2n^2 + 5\ n$ is $O(n^3)$

$2n^2 + 5\ n = O(n^3)$

$\cancel{O(n^3) = 2n^2 + 5\ n}$

Bottom line:

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

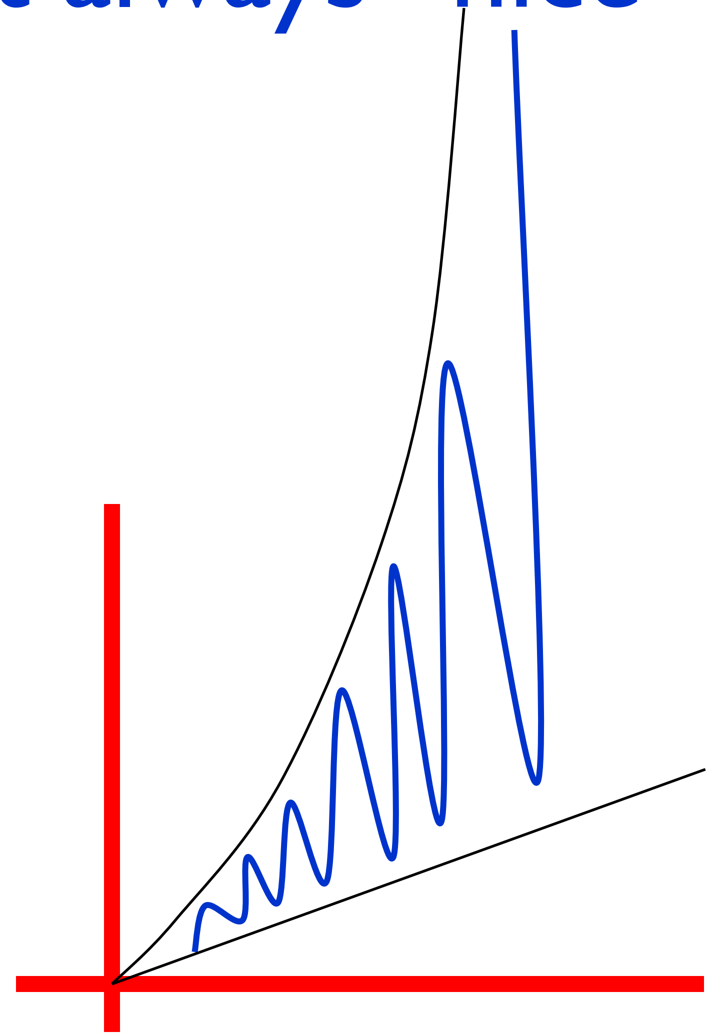
[Better, but uncommon, notation: $T(n) < O(f(n))$.]

Big-Theta, etc. not always "nice"

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

f(n) is not $\Theta(n^a)$ for any *a*.

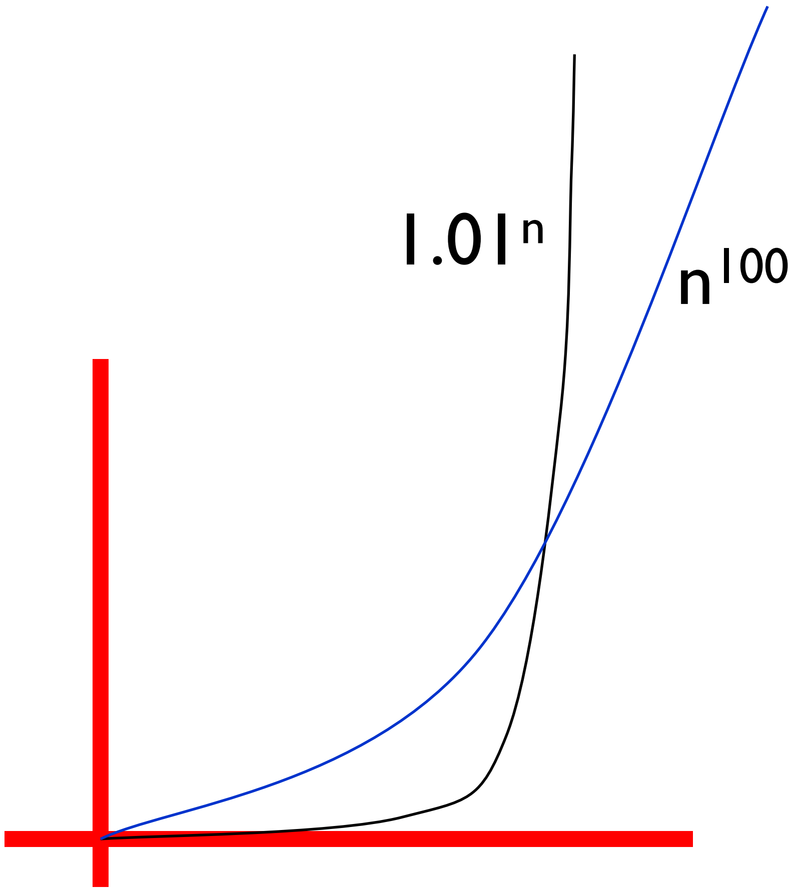
Fortunately, such nasty cases are rare



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 $n \rightarrow n+d$ increases time by a factor of more than c.

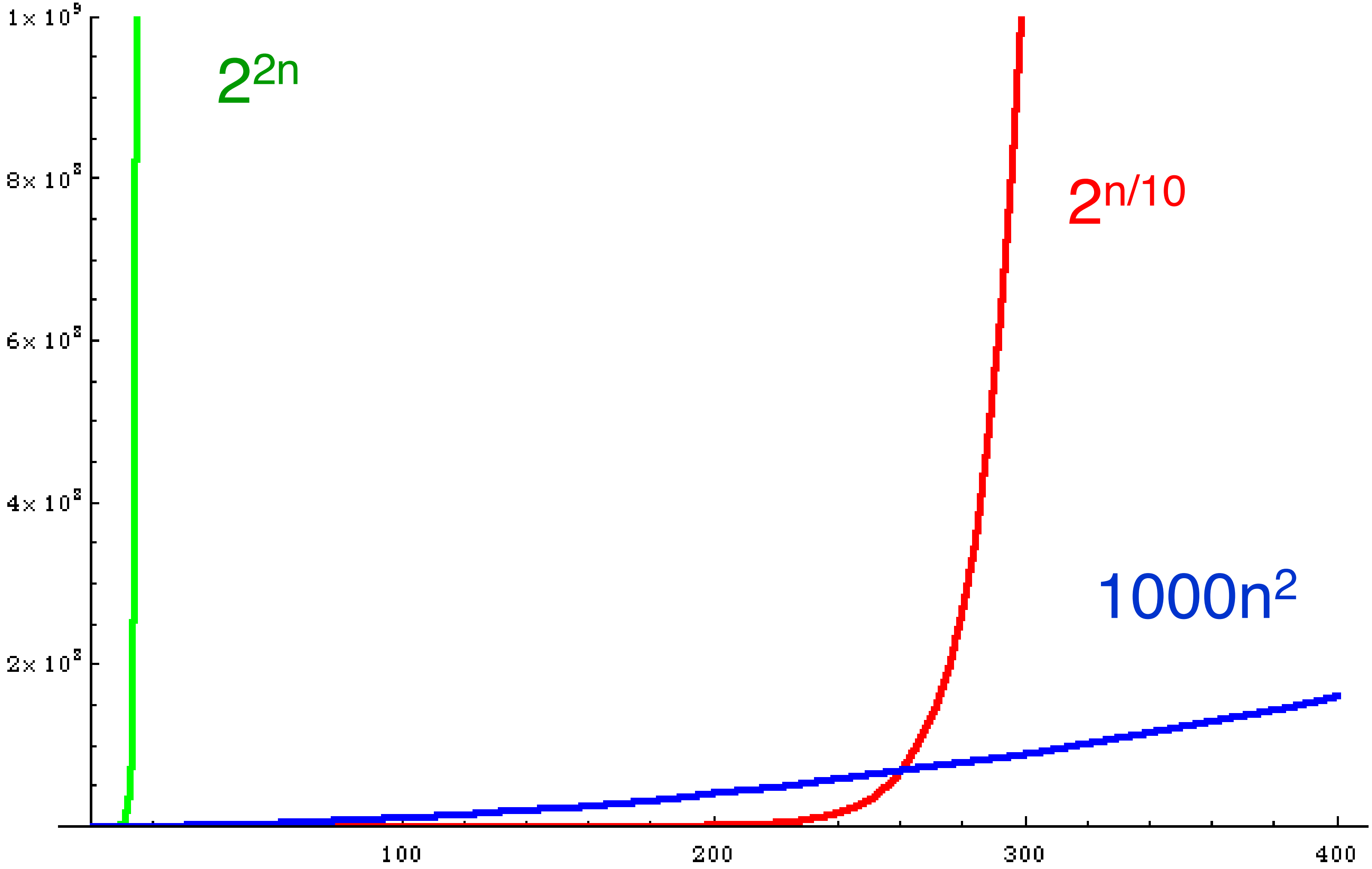
(E.g., 2^n vs 2^{n+1})

Polynomial time

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

Behaves well under composition: if algorithm has a polynomial running time with polynomial number of calls to a subroutine that has polynomial running time, then overall running time is still polynomial.

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

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 < b then n^a is $O(n^b)$

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

Note:
if f(n) is $\Omega(g(n))$ then it cannot be o(g(n))

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!