CSE 417: Algorithms and Computational Complexity

Lecture 2: Analysis

Larry Ruzzo

- Why big-O: measuring algorithm efficiency
- What's big-O: definition and related concepts
- Reasoning with big-O: examples & applications
    - polynomials
    - exponentials
    - logarithms
    - sums
- Polynomial Time

Why big-O: measuring algorithm efficiency

Let p_n = n^{th} prime, n >= 1, e.g.:

$p_1 = 2$

$p_2 = 3$

$p_3 = 5$

$p_4 = 7$

$p_5 = 11$

After much study, we know $p_n \sim n \log n$

Better: $$\log n + \log\log n - 1 < \frac{p_n}{n} < \log n + \log\log n \quad \text{for } n \geq 6.$$

Great to have that precision, but sometimes

$p_n = O(n \log n)$ is all you need

- Our correct TSP algorithm was incredibly slow
  - No matter what computer you have
- As a 2nd example, for large problems, mergesort beats insertion sort – n log n *vs* n^2 matters a lot
  - Even tho the alg is more complex & inner loop is slower
  - No matter what computer you have
- We want a general theory of "efficiency" that is
  - Simple
  - Objective
  - Relatively independent of changing technology
  - Measures *algorithm*, not code
  - But still *predictive* – "theoretically bad" algorithms should be bad in practice and vice versa (usually)

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

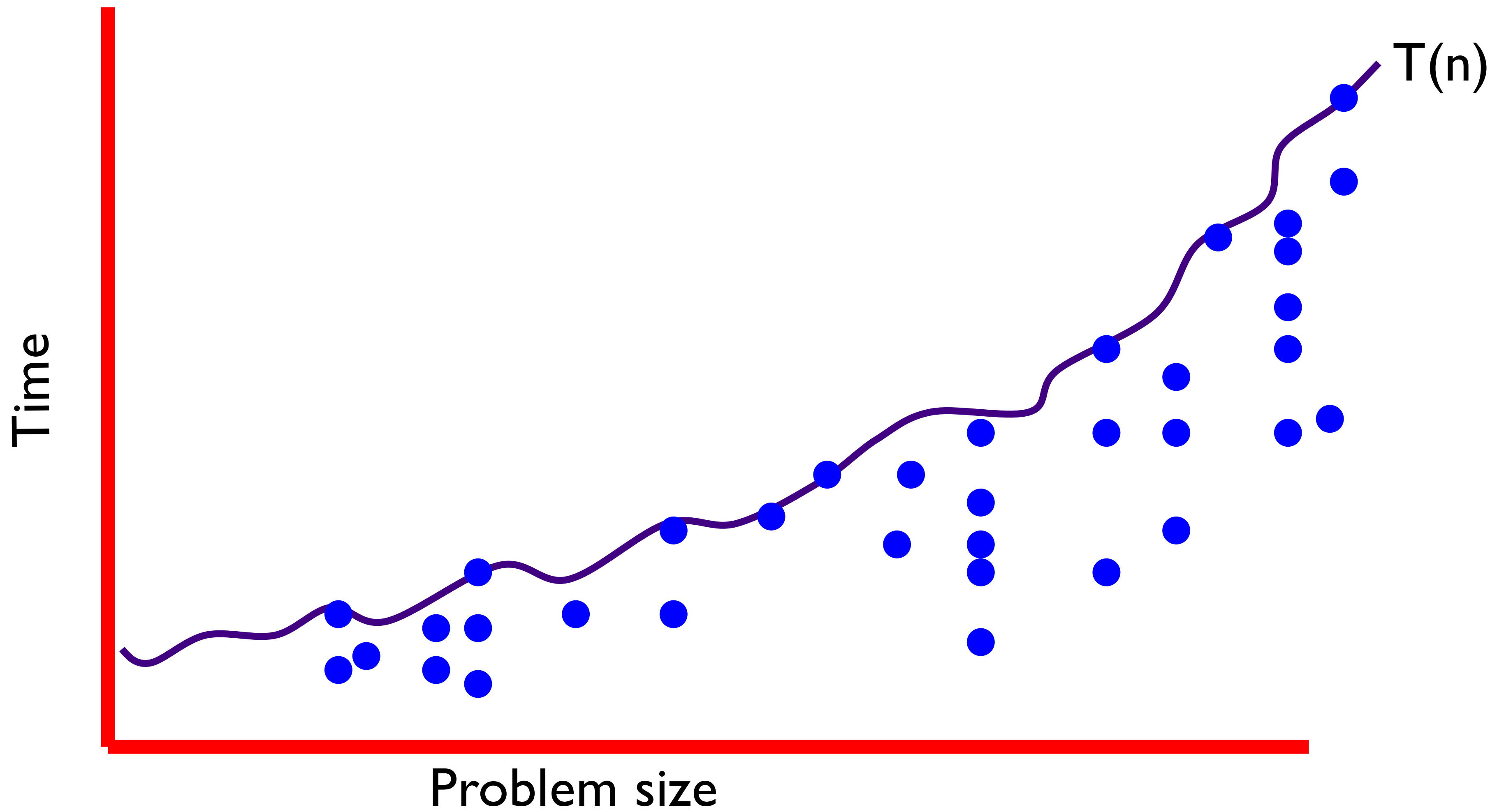
Mathematically,

$T: N+ \to R$

i.e., T is a function mapping positive integers (problem sizes) to positive real numbers (number of steps).

"Reals" so, e.g., we can say sqrt(n) instead of $\lceil \mathrm{sqrt}(n) \rceil$

"Positive" so, e.g., log(n) and $2^n/n$ aren't problematic


T(n)
Time
Problem size

Asymptotic growth rate, i.e., characterize growth rate of worst-case run time as a function of problem size, up to a constant factor, e.g. $T(n) = O(n^2)$

Why not try to be more precise?

- Average-case, e.g., is hard to define, analyze
- Technological variations (computer, compiler, OS, …) easily 10x or more
- Being more precise is *much* more work
- A key question is "*scale up*": if I can afford this today, how much longer will it take when my business is 2x larger? (E.g. today: cn^2, next year: $c(2n)^2 = 4cn^2$: 4 x longer.) Big-O analysis is adequate to address this.

What's big-O: definition and related concepts

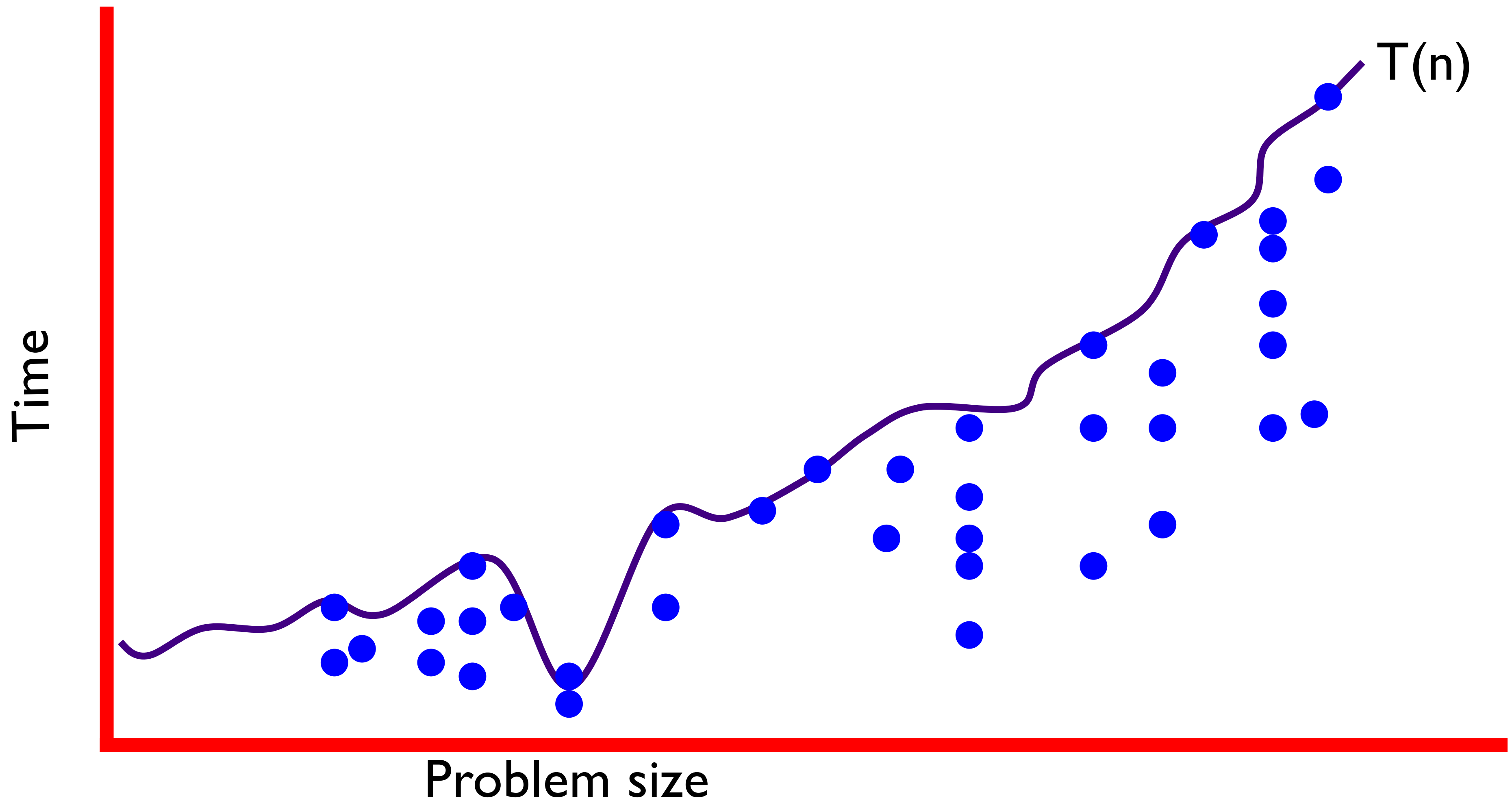
Given two functions f and g: N+ $\rightarrow$ R

f(n) is O(g(n)) iff there is a constant $c > 0$ so that f(n) is eventually always $\leq c\, g(n)$ — Upper Bounds

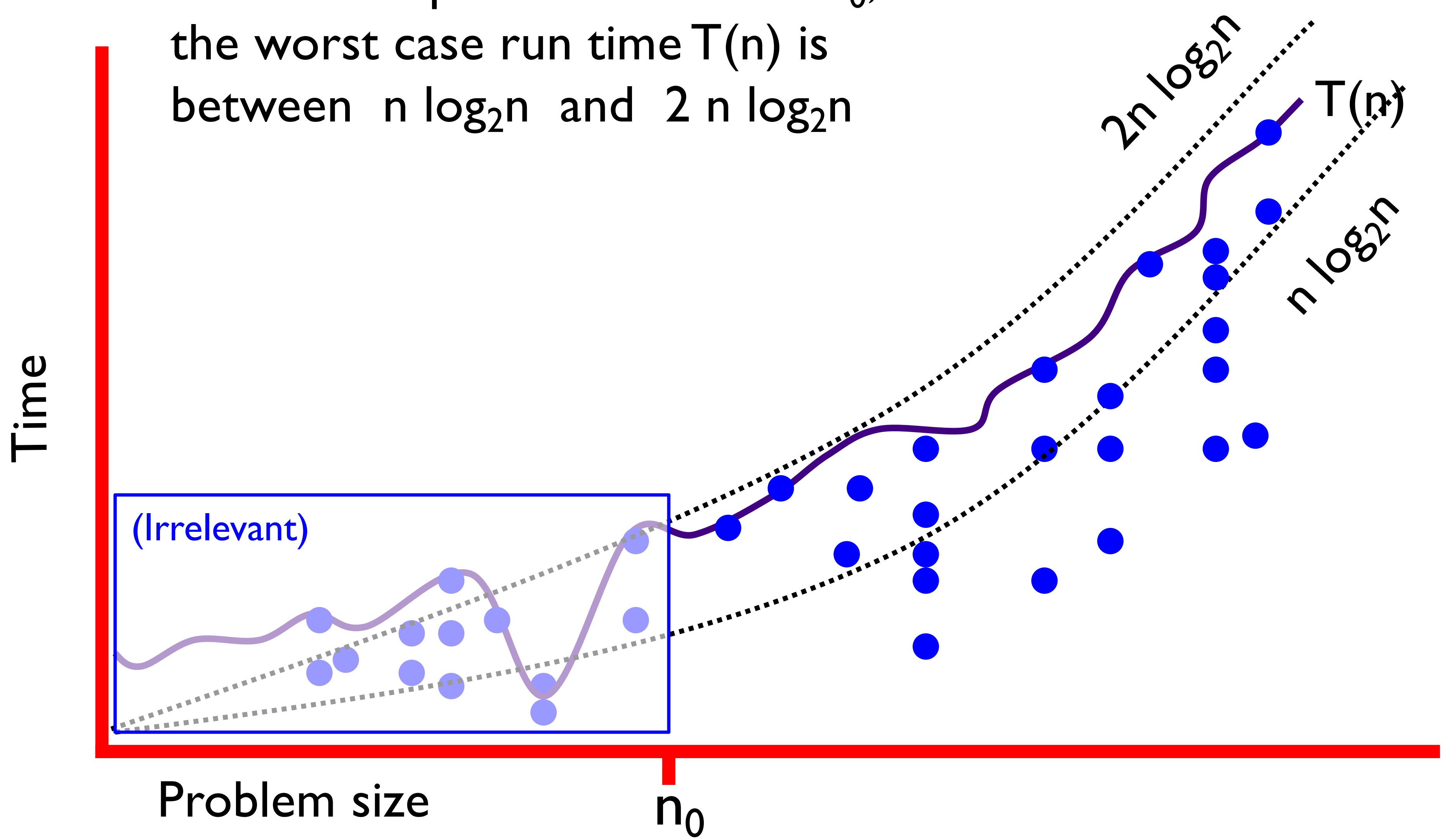
f(n) is Ω(g(n)) iff there is a constant $c > 0$ so that f(n) is eventually always $\geq c\, g(n)$ — Lower Bounds

f(n) is Θ(g(n)) iff there is are constants $c_1, c_2 > 0$ so that eventually always $c_1 g(n) \leq f(n) \leq c_2 g(n)$ — Both

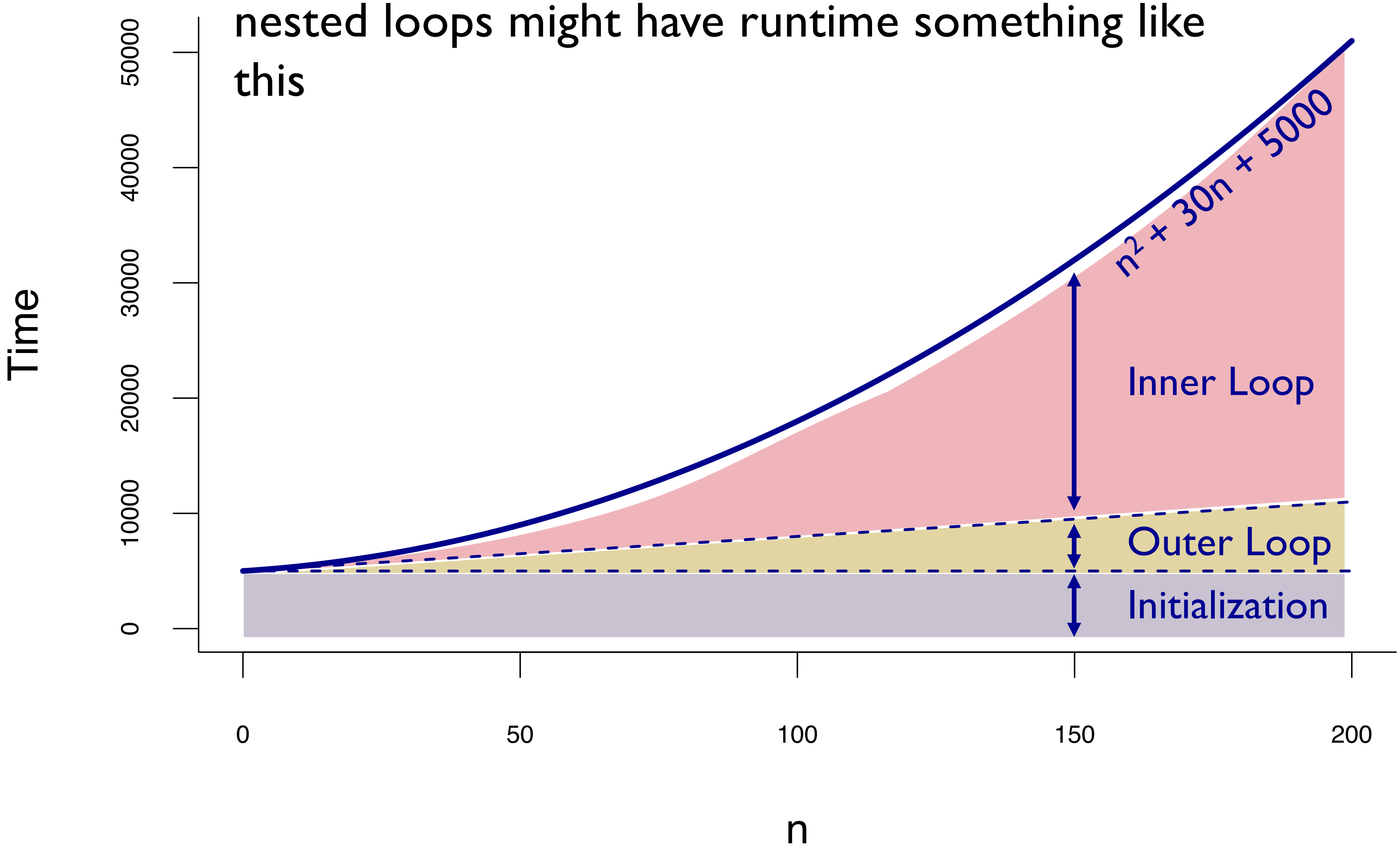
"Eventually always P(n)" means "$\exists n_0$ s.t. $\forall n > n_0$ P(n) is true." I.e., there can be exceptions, but only for finitely many "small" values of n.


T(n)
Time
Problem size

Example: $T(n) = \Theta(n \log n)$
since for all problem sizes $n > n_0$,
the worst case run time $T(n)$ is
between $n \log_2 n$ and $2 n \log_2 n$

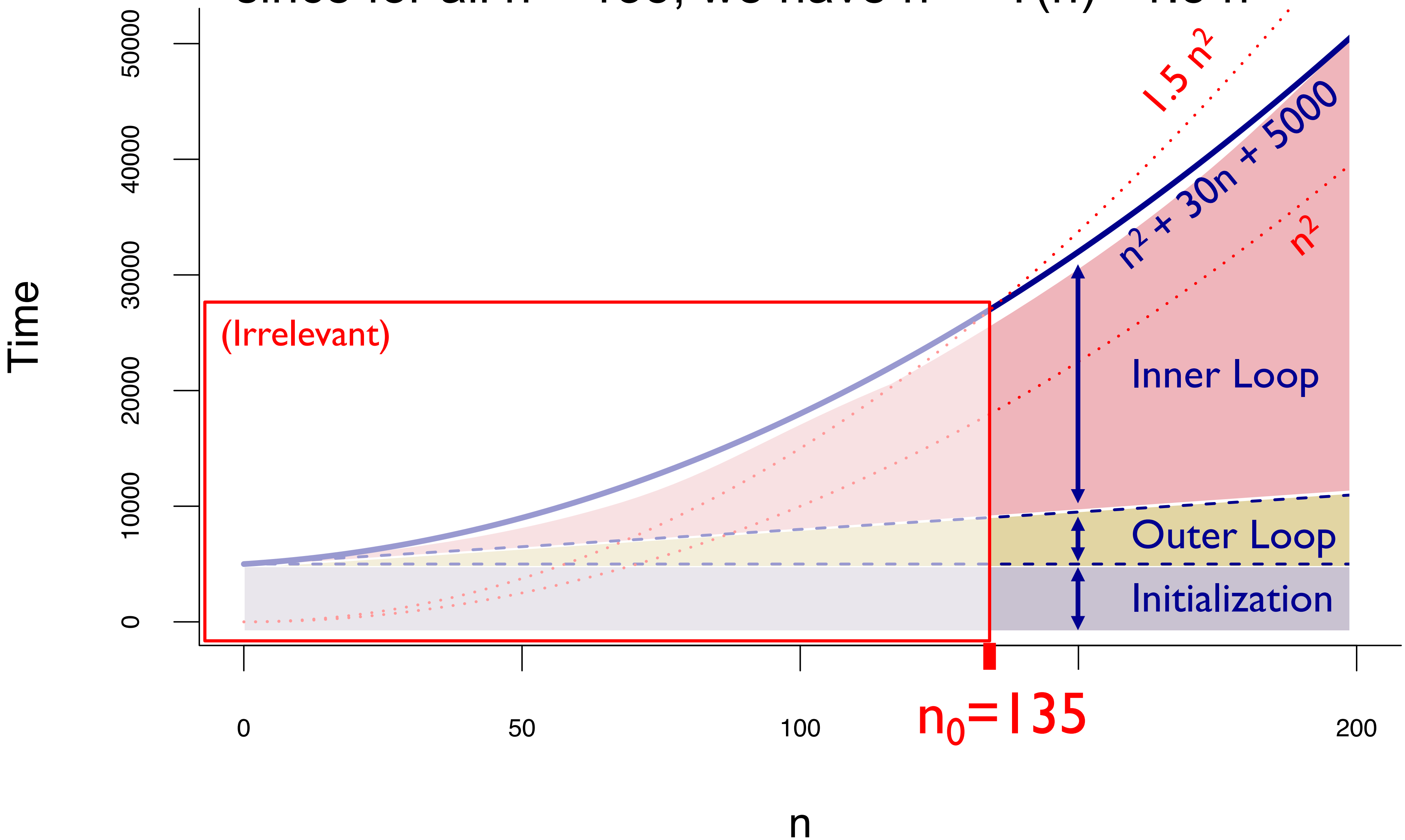


A typical program with initialization and two nested loops might have runtime something like this



Lecture 2 Jan 9 END Lect 3 Jan 11 Start

If $T(n) = n^2 + 30n + 5000$, then $T(n) = \Theta(n^2)$,
since for all $n \geq 135$, we have $n^2 \leq T(n) \leq 1.5\, n^2$



Reasoning with big-O: examples & applications

polynomials

exponentials

logarithms

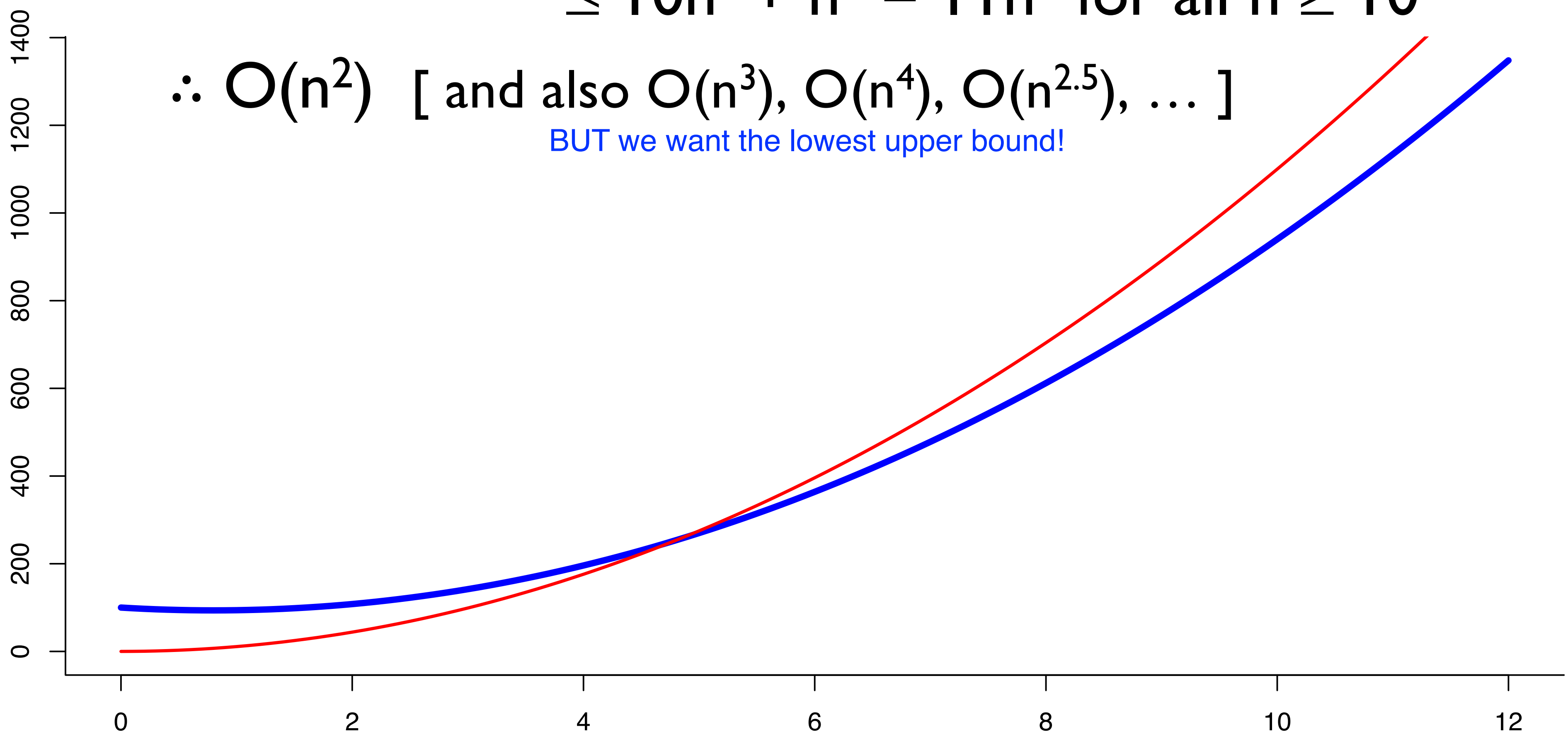
sums

Show $10n^2-16n+100$ is $O(n^2)$:

$$10n^2-16n+100 \le 10n^2 + 100$$
$$= 10n^2 + 10^2$$
$$\le 10n^2 + n^2 = 11n^2 \text{ for all } n \ge 10$$

$\therefore O(n^2)$ [and also $O(n^3)$, $O(n^4)$, $O(n^{2.5})$, …]

BUT we want the lowest upper bound!



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

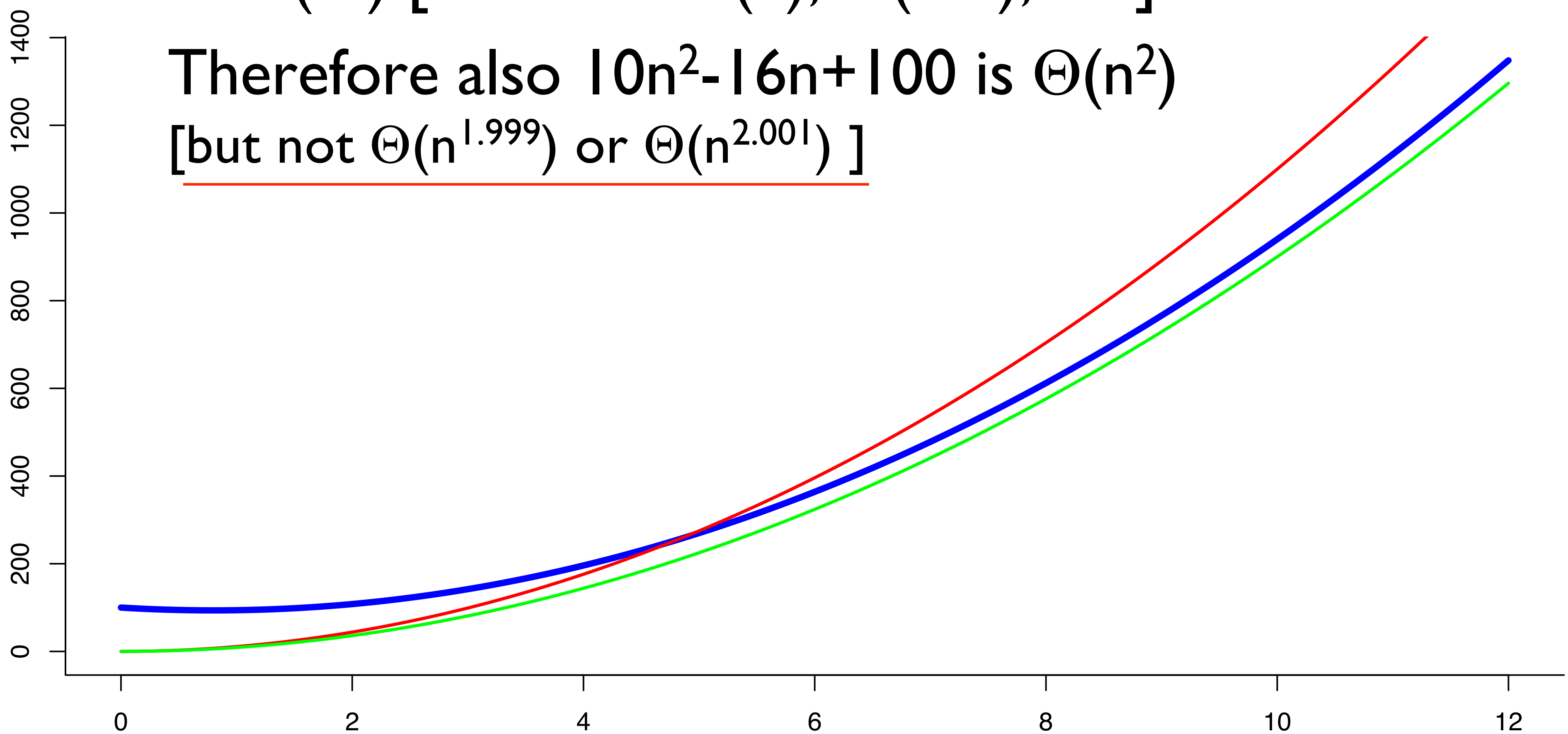
$10n^2-16n+100 \geq 10n^2 - 16n$

$\geq 10n^2 - n^2 = 9n^2$ for all $n \geq 16$

$\therefore \Omega(n^2)$ [and also $\Omega(n)$, $\Omega(n^{1.5})$, …]

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

[but not $\Theta(n^{1.999})$ or $\Theta(n^{2.001})$]



Polynomials:

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

Proof:

$$
\begin{aligned}
p(n) &= a_0 + a_1 n + \ldots + a_d n^d \\
&\le |a_0| + |a_1| n + \ldots + a_d n^d \\
&\le |a_0| n^d + |a_1| n^d + \ldots + a_d n^d \qquad \text{(for } n \ge 1\text{)} \\
&= c\, n^d, \text{ where } c = (|a_0| + |a_1| + \ldots + |a_{d-1}| + a_d)
\end{aligned}
$$

$\therefore p(n) = O(n^d)$

Exercise: show that $p(n) = \Omega(n^d)$

Hint: this direction is trickier; focus on the "worst case" where all coefficients except a_d are negative.

Example: 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 $\Omega\ (n^b)$

more examples: tricks for sums

Example: $\sum_{1 \le i \le n} i = \Theta(n^2)$

E.g. : for i = 1..n {
for j=1 to i {
. . .
}}

sum i = 1 to n = n(n+1) / 2 is clearly theta(n^2)

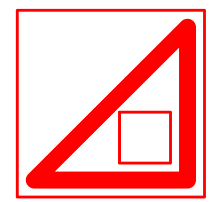
Proof:

(a) An upper bound: each term is ≤ the max term

$$\sum_{1 \le i \le n} i \le \sum_{1 \le i \le n} n = n^2 = O(n^2)$$

(b) A lower bound: each term is ≥ the min term

$$\sum_{1 \le i \le n} i \ge \sum_{1 \le i \le n} 1 = n = \Omega(n)$$



This is valid, but a weak bound.

Better: pick a large subset of large terms

$$\sum_{1 \le i \le n} i \ge \sum_{n/2 \le i \le n} n/2 \ge \lfloor n/2 \rfloor^2 = \Omega(n^2)$$

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 $f + g = \Theta(h)$.

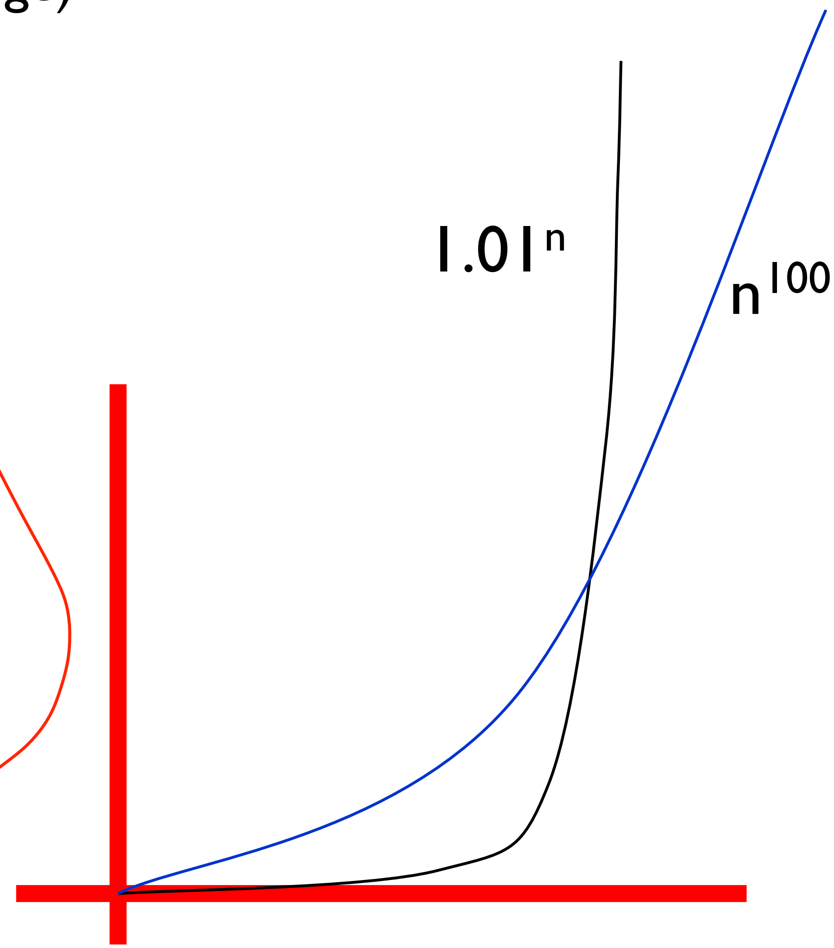
Useful, e.g., for analyzing programs with subroutines.

Proofs are left as exercises.

For all $r > 1$ (no matter how small)
and all $d > 0$, (no matter how large)
$n^d = O(r^n)$

In short, every exponential grows faster than every polynomial!

(To prove this, use calculus tricks like L'Hospital's rule.)



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

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

definition

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

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

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

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

$$\text{So}:$$

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

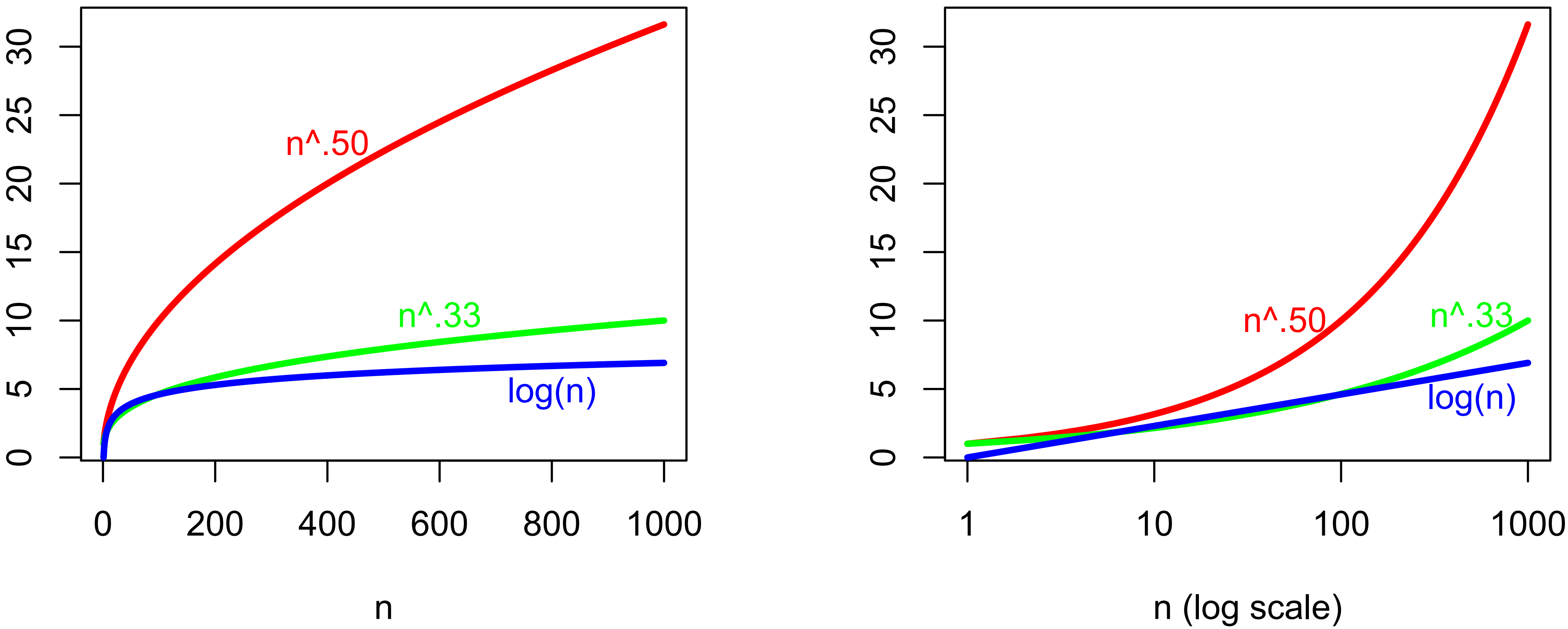
Corollary: base of a log *factor* is usually irrelevant, asymptotically. E.g. "O(n log n)" [but $n^{\log_2 8} \neq \Theta(n^{\log_8 8})$]

i.e. we could use the change of base formula for logs to change to base 10 with an extra constant multiple.

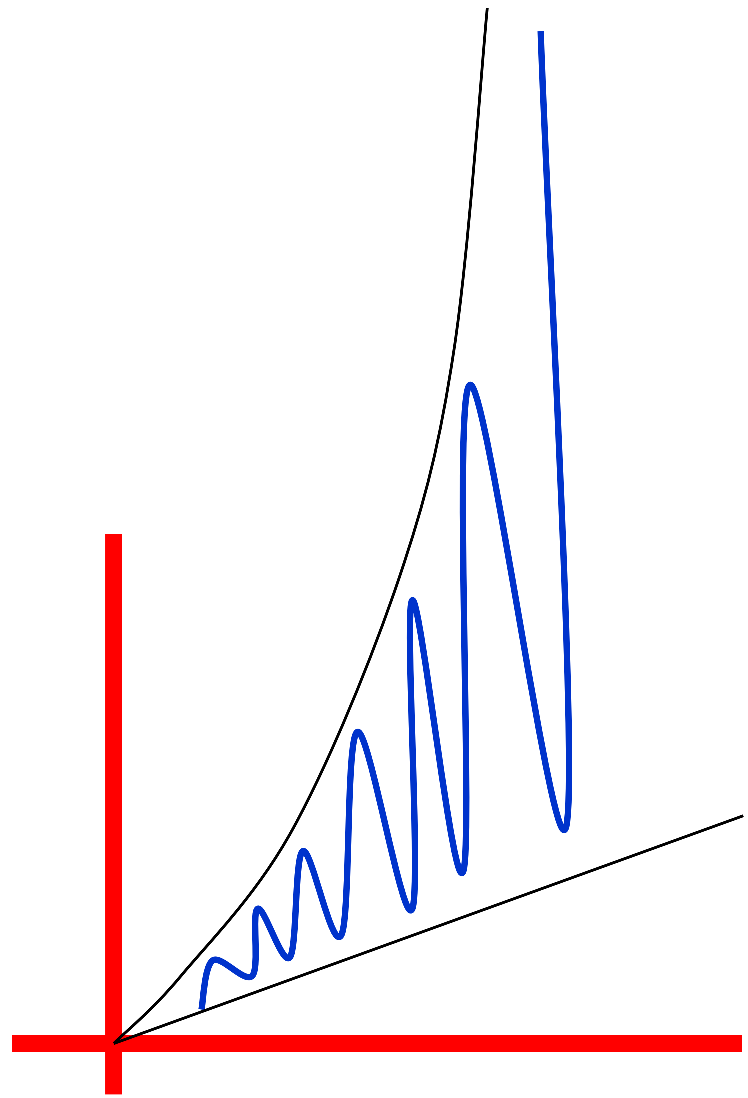
Logarithms:

For all x > 0, *(no matter how small)* $\log n = O(n^x)$

log grows slower than every polynomial



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



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

Fortunately, such nasty cases are rare

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

Polynomial Time

P: The set of problems solvable by algorithms with running time $O(n^d)$ for some constant d

(d is a constant 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., c = 100 and d = 7 for 2^n vs 2^{n+7})

polynomial vs exponential growth

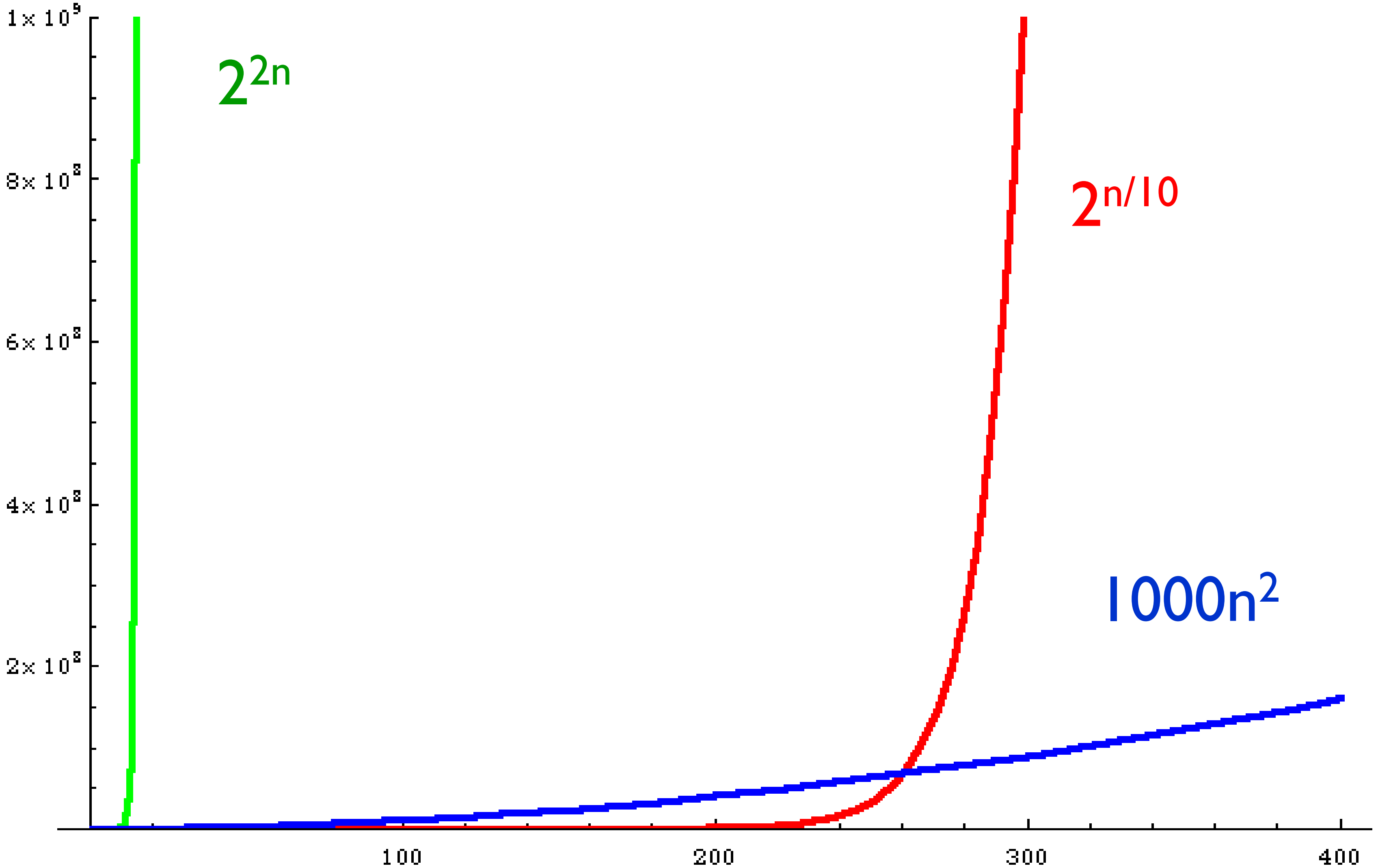


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	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

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

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

Summary

A 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!

As one important example, poly time algorithms are almost always preferable to exponential time ones.