

---

# CSE 421: Intro Algorithms

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

Our correct TSP algorithm was incredibly slow

No matter what computer you have

As a 2<sup>nd</sup> 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 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)

“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”?)

“Runs fast on a specific suite of benchmarks”

Pro:

again sensible, bottom-line-oriented

Con:

all the problems above

are benchmarks representative

algorithms can be “tuned” to the well-known benchmarks

generating/maintaining benchmarks is a burden

benchmarking a new algorithm is a lot of work

Instead:

a) Give up on detailed timing, focus on *scaling*

Nanoseconds matter of course, but we often want to push to bigger problems tomorrow than we can solve today, so an algorithm that scales as  $n^2$ , say, will very likely beat one that grows as  $2^n$  or  $n^{10}$  or even  $n^3$ , even if the later uses fewer nanoseconds for today's  $n$ .

b) Give up on “typical,” focus on *worst case* behavior

Over all inputs of size  $n$ , how fast are we on the worst?  
Removes all debate about “typical” / “average.”

Overall, these yield a big win in terms of technology independence, ease of analysis, robustness

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

Mathematically,

$$T: \mathbb{N}^+ \rightarrow \mathbb{R}$$

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

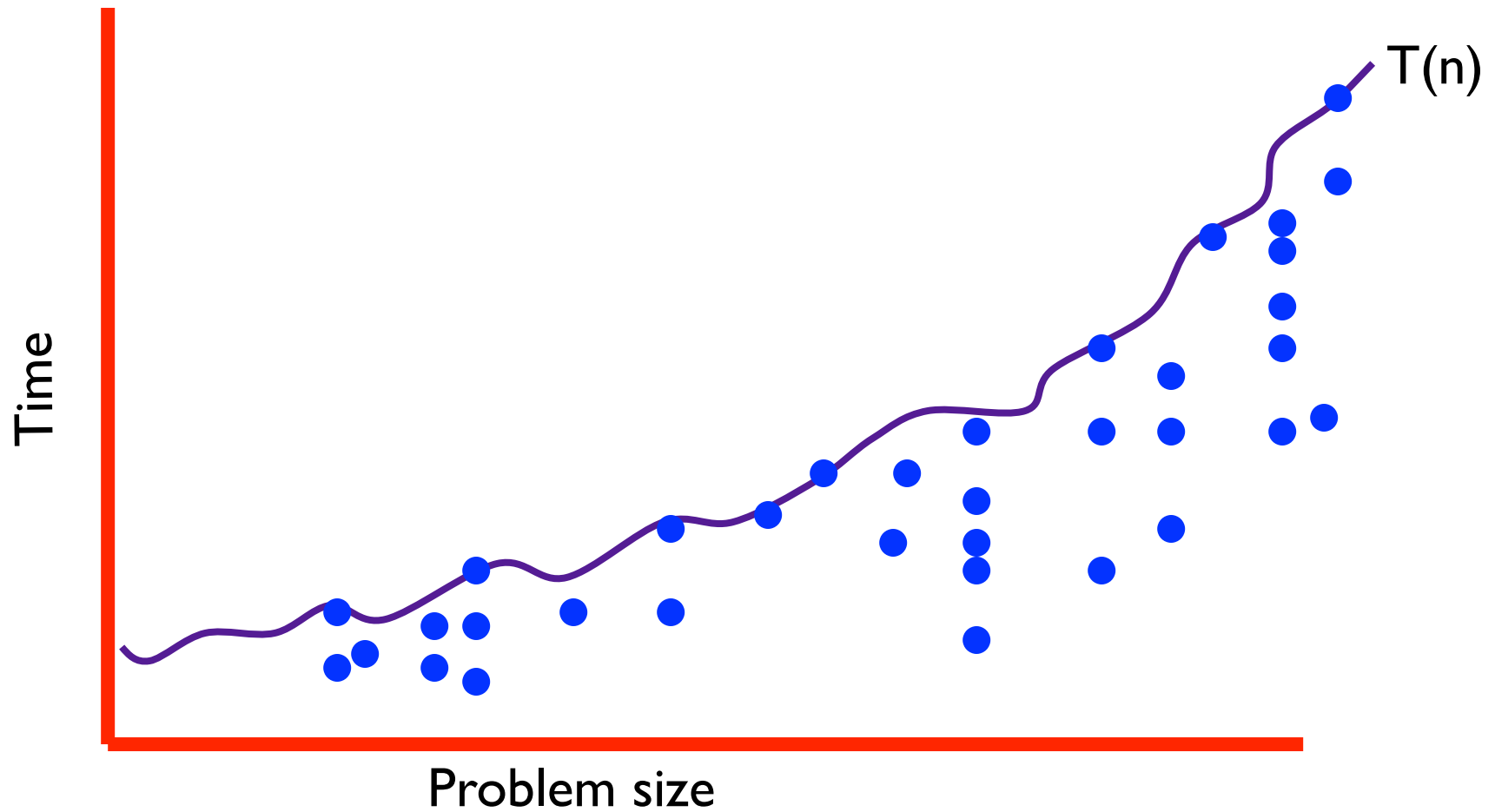
“Reals” so, e.g., we can say  $\sqrt{n}$  instead of  $\lceil \sqrt{n} \rceil$

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



# computational complexity

---



Appropriate for time-critical applications

E.g. avionics, nuclear reactors

Unlike Average-Case, no debate over the right definition

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

Analysis often much easier

Result is often representative of “typical” problem instances

Of course there are exceptions...

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.

---

**Big-O:** math notation for an upper bound on the asymptotic growth rate of a function

E.g., if  $f(n)$  = value of the  $n^{\text{th}}$  prime,  $f(n) = O(n \log n)$

In CS, commonly used to describe run time of algorithms, usually worst case run time, but could be other run time functions.

E.g., for Quicksort

$$T_{\text{best}}(n) = O(n)$$

$$T_{\text{avg}}(n) = O(n \log n)$$

$$T_{\text{worst}}(n) = O(n^2)$$

---

What's big-O: definition and related concepts

Given two functions  $f$  and  $g: \mathbb{N}^+ \rightarrow \mathbb{R}$

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

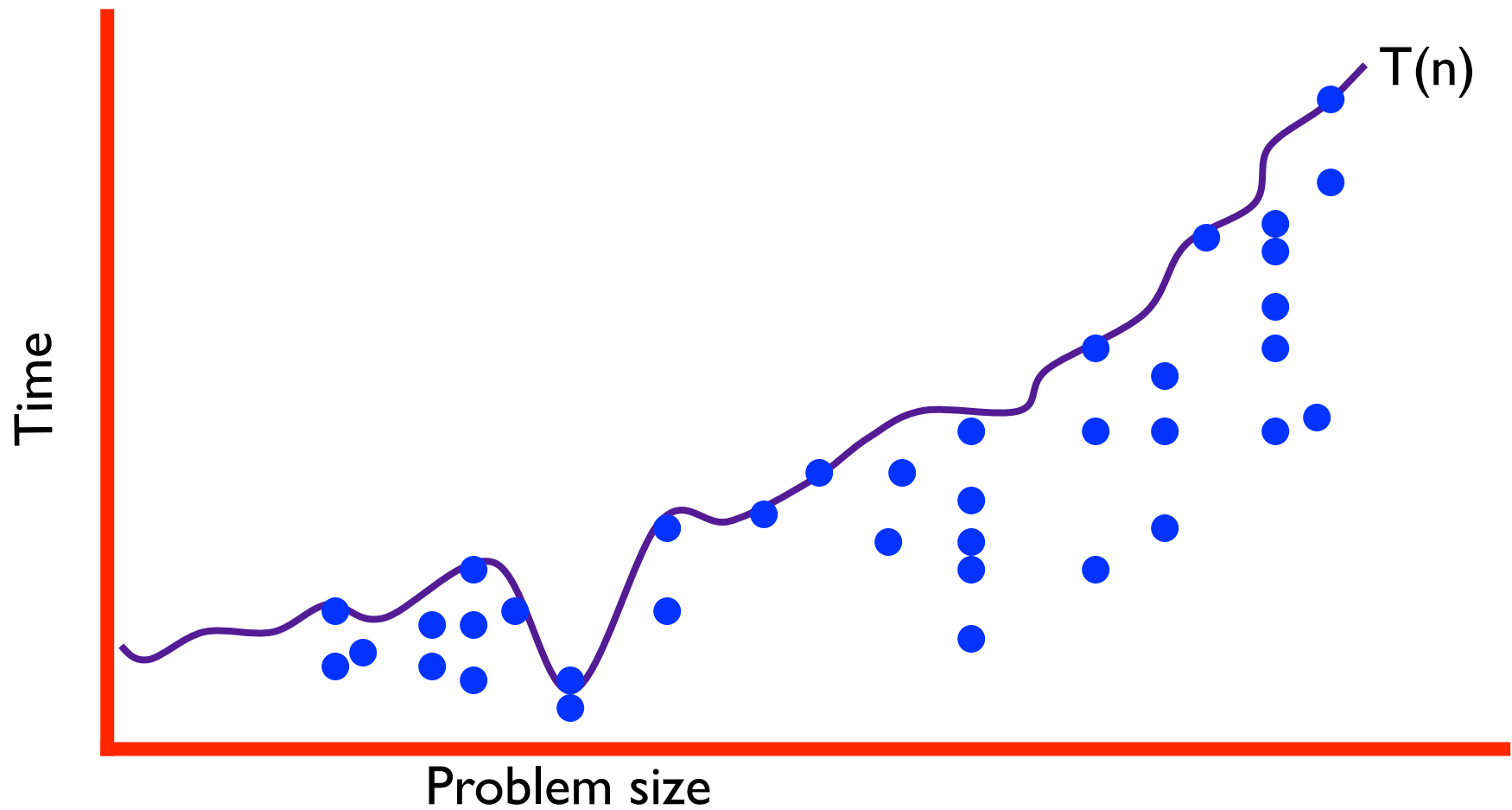
$f(n)$  is  $\Omega(g(n))$  iff there is a constant  $c > 0$  so that  
 $f(n)$  is eventually always  $\geq c g(n)$  Lower  
Bounds

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

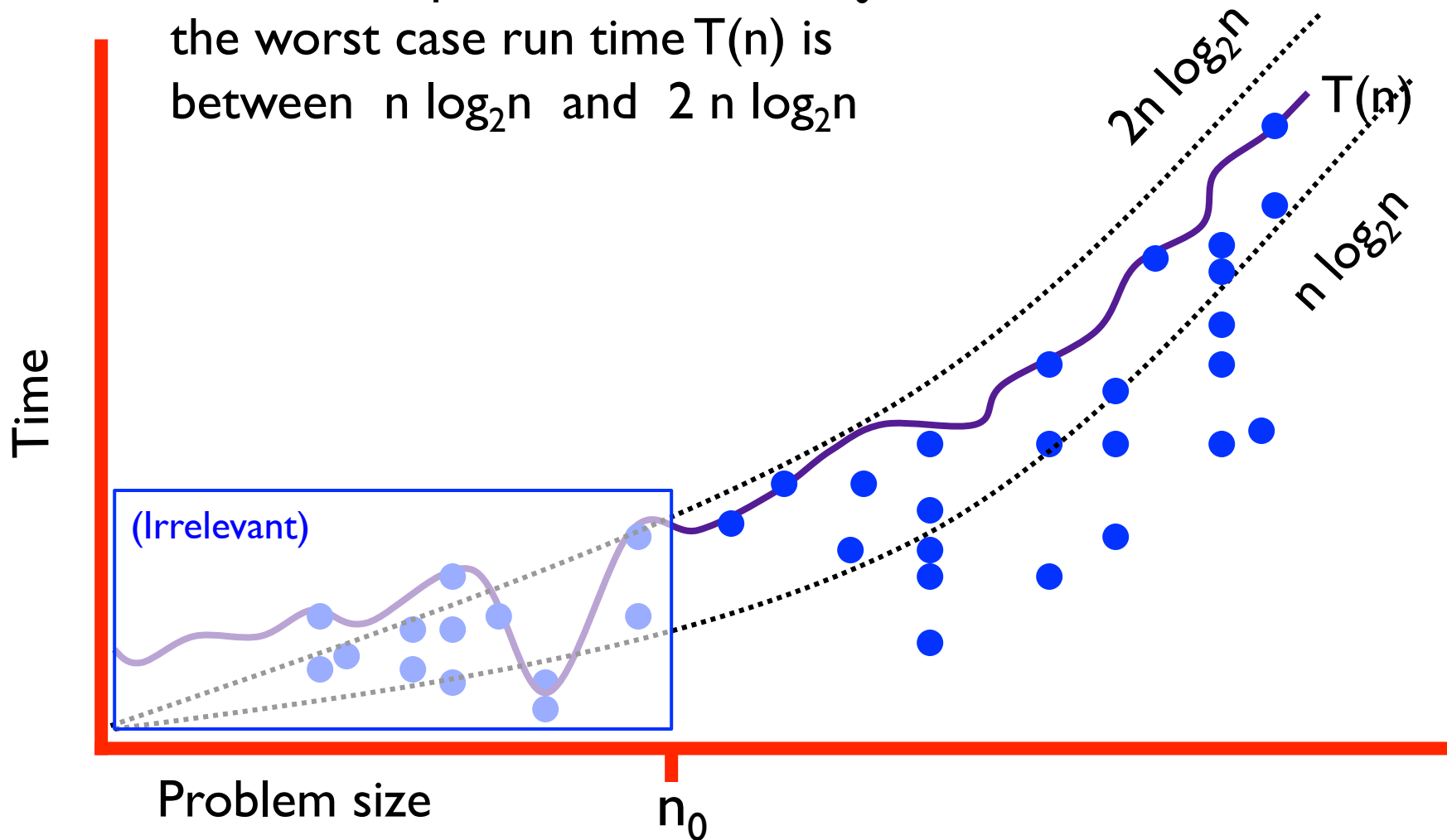
“Eventually always  $P(n)$ ” means “ $\exists n_0$  s.t.  $\forall n > n_0$   $P(n)$  is true.” I.e., there can be exceptions, but only for finitely many “small” values of  $n$ .

# computational complexity

---

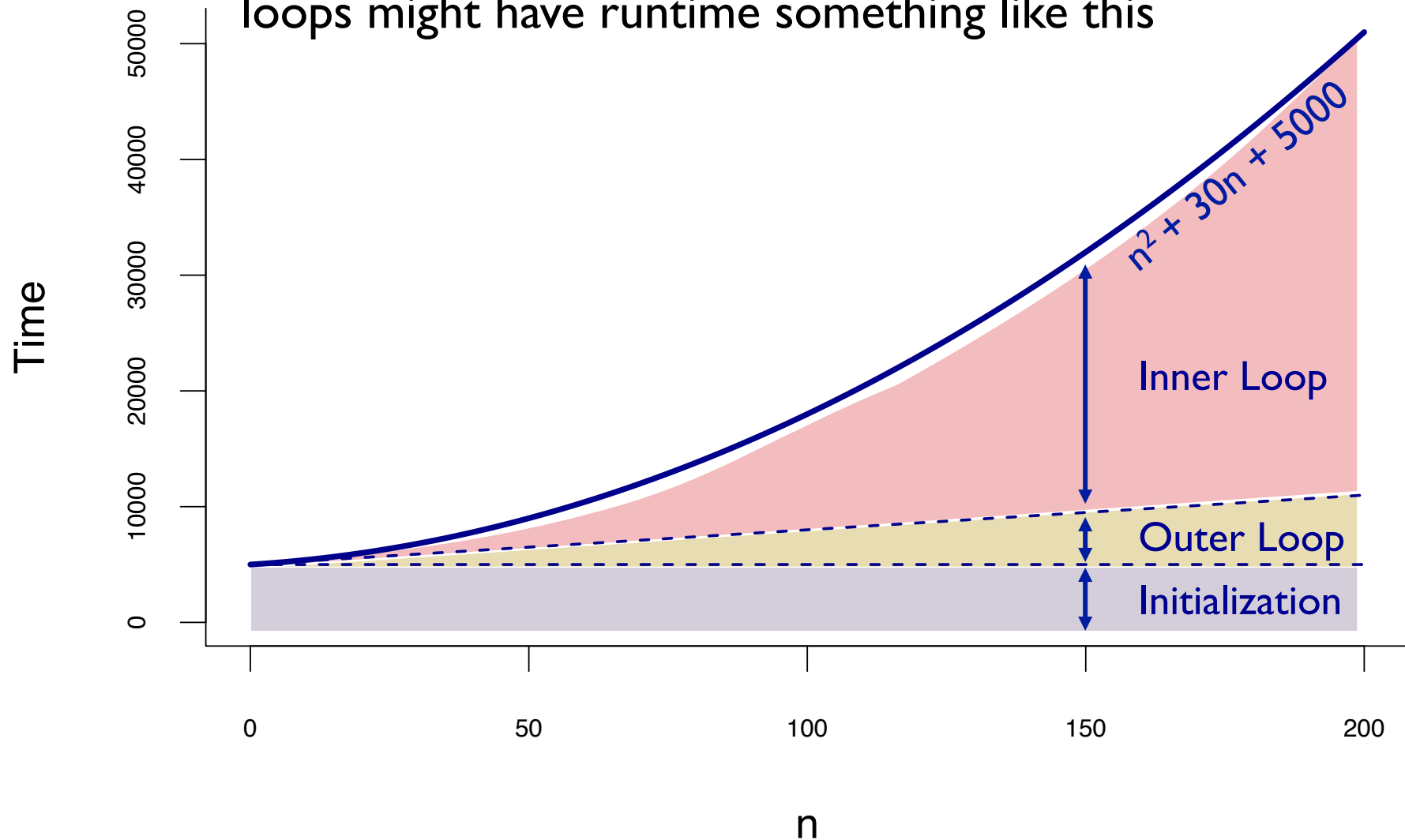


Example:  $T(n) = \Theta(n \log_2 n)$   
 since for all problem sizes  $n > n_0$ ,  
 the worst case run time  $T(n)$  is  
 between  $n \log_2 n$  and  $2n \log_2 n$

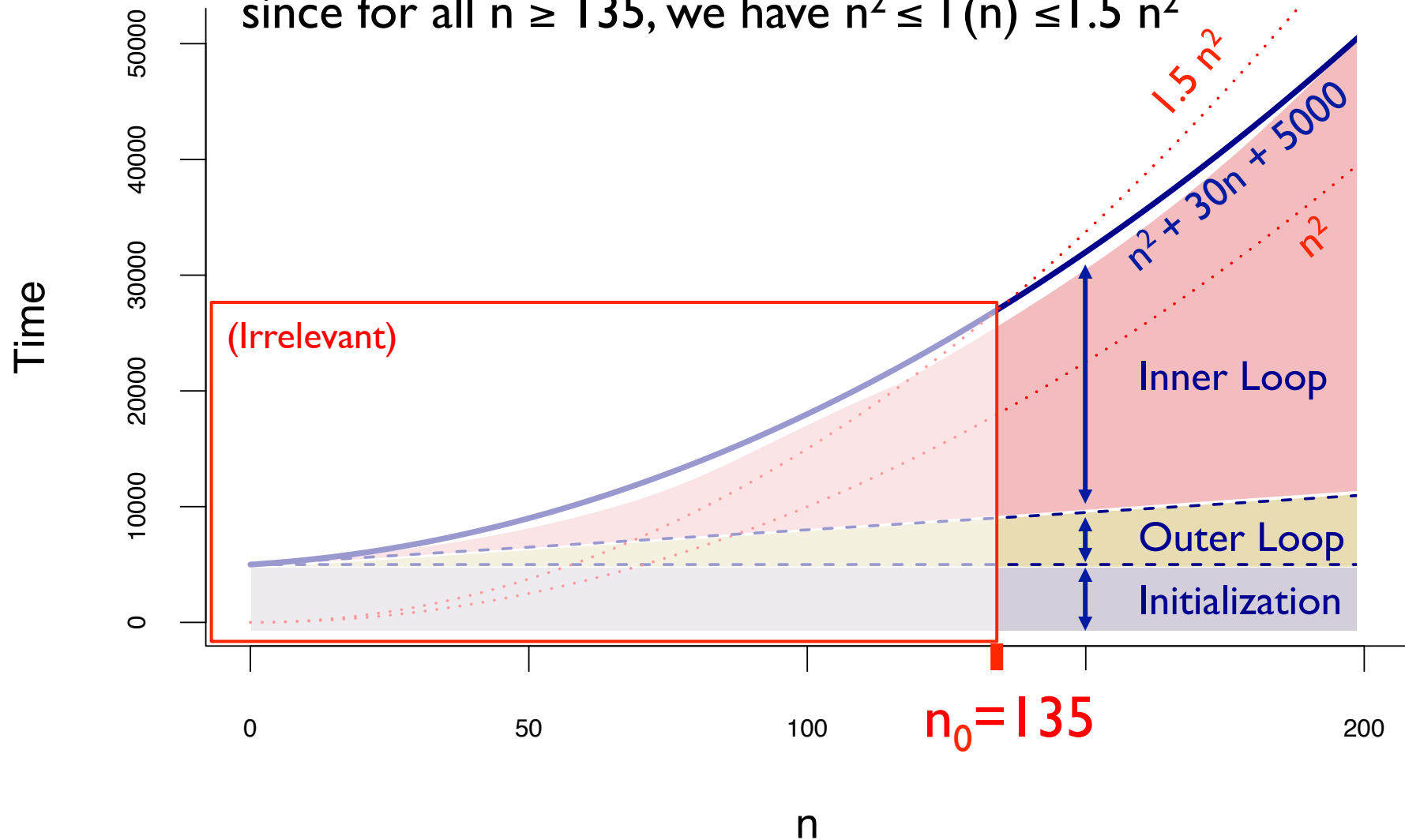




A typical program with initialization and two nested loops might have runtime something like this



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

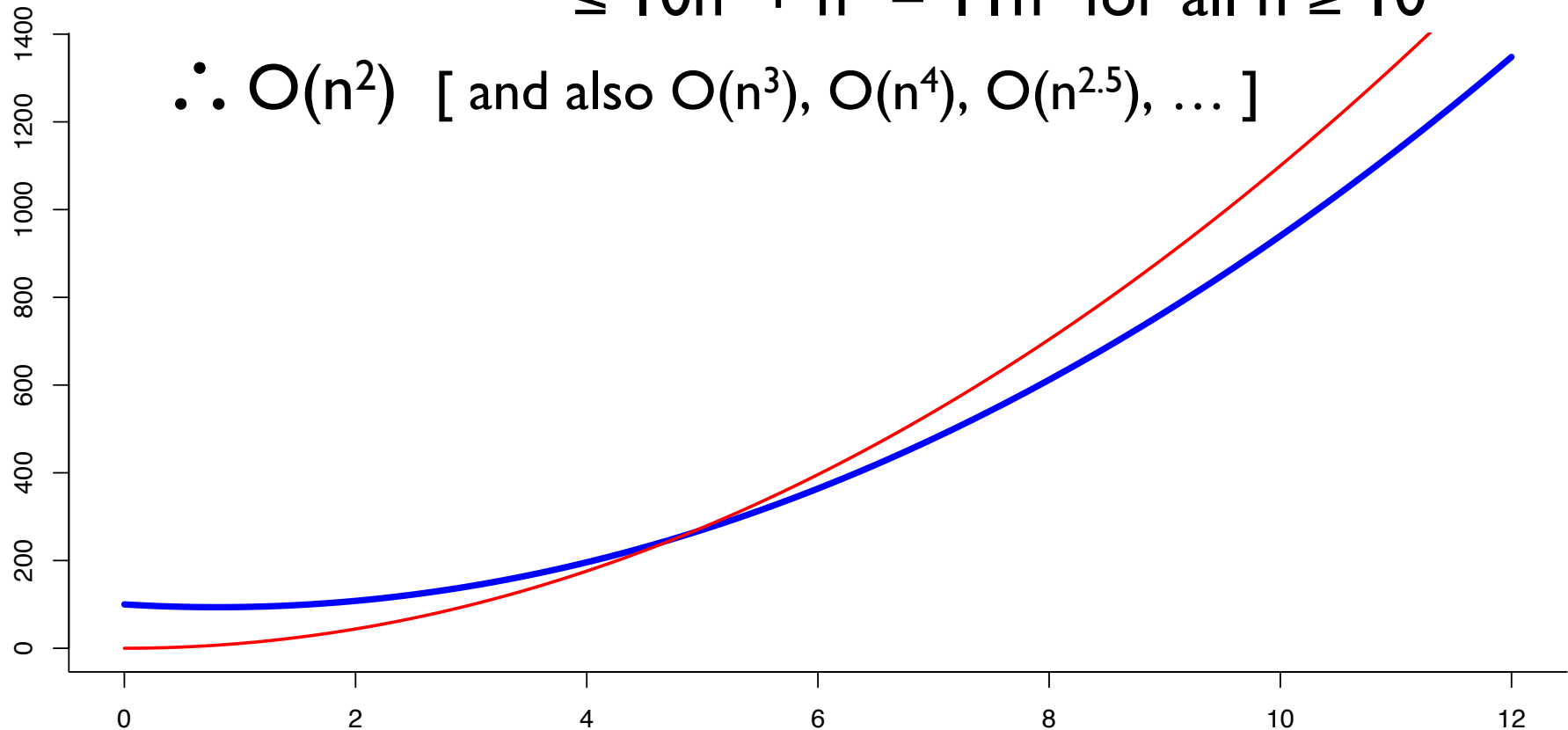
Show  $10n^2 - 16n + 100$  is  $O(n^2)$  :

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

$$= 10n^2 + 10^2$$

$$\leq 10n^2 + n^2 = 11n^2 \text{ for all } n \geq 10$$

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



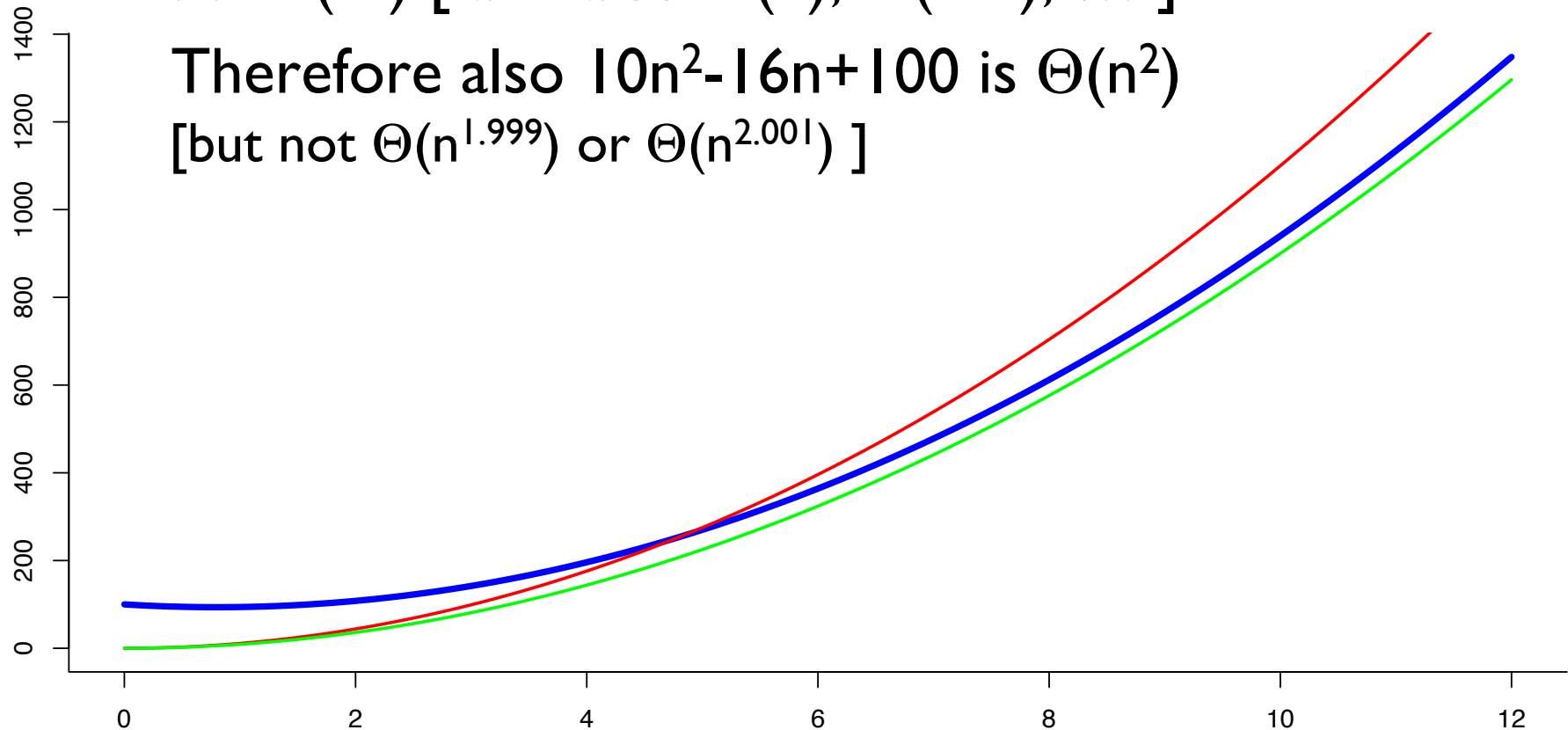
Show  $10n^2 - 16n + 100$  is  $\Omega(n^2)$  :

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

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

$\therefore \Omega(n^2)$  [ 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:

$$p(n) = a_0 + a_1 n + \dots + a_d n^d \text{ is } \Theta(n^d) \text{ if } a_d > 0$$

Proof:

$$\begin{aligned} p(n) &= a_0 + a_1 n + \dots + a_d n^d \\ &\leq |a_0| + |a_1| n + \dots + a_d n^d \\ &\leq |a_0| n^d + |a_1| n^d + \dots + a_d n^d \quad (\text{for } n \geq 1) \\ &= c n^d, \text{ where } c = (|a_0| + |a_1| + \dots + |a_{d-1}| + a_d) \end{aligned}$$

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

**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.**

## another example of working with $O$ - $\Omega$ - $\Theta$ notation

---

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

$$\begin{aligned}(n+a)^b &\leq (2n)^b && \text{for } n \geq |a| \\ &= 2^b n^b \\ &= cn^b && \text{for } c = 2^b\end{aligned}$$

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

$$\begin{aligned}(n+a)^b &\geq (n/2)^b && \text{for } n \geq 2|a| \text{ (even if } a < 0) \\ &= 2^{-b} n^b \\ &= c'n && \text{for } c' = 2^{-b}\end{aligned}$$

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

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

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

Proof:

(a) An upper bound: each term is  $\leq$  the max term

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

(b) A lower bound: each term is  $\geq$  the min term

$$\sum_{1 \leq i \leq n} i \geq \sum_{1 \leq i \leq n} 1 = n = \Omega(n)$$

This is valid, but a weak bound.

Better: pick a large subset of large terms

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



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

Proofs are left as exercises.

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

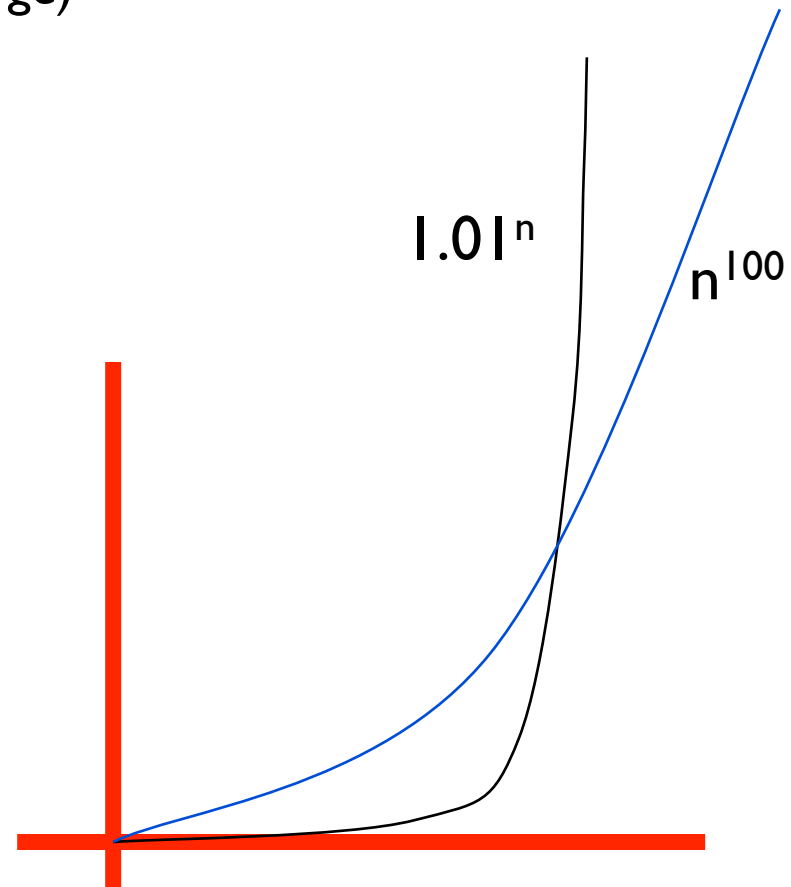
## polynomial vs exponential

---

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!

(proof below)



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

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

definition

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

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

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

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

So :

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

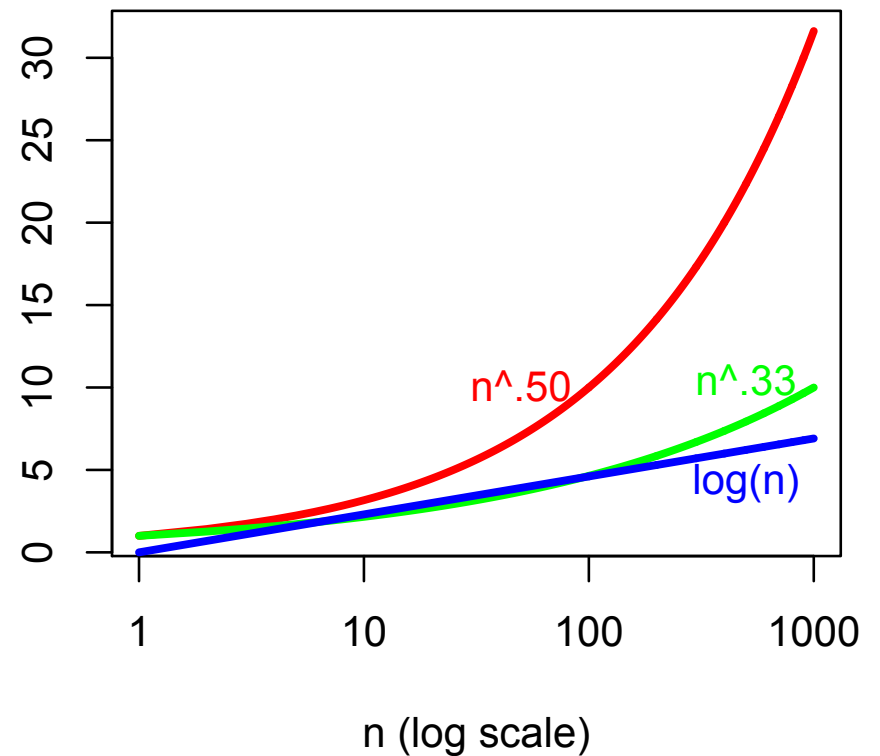
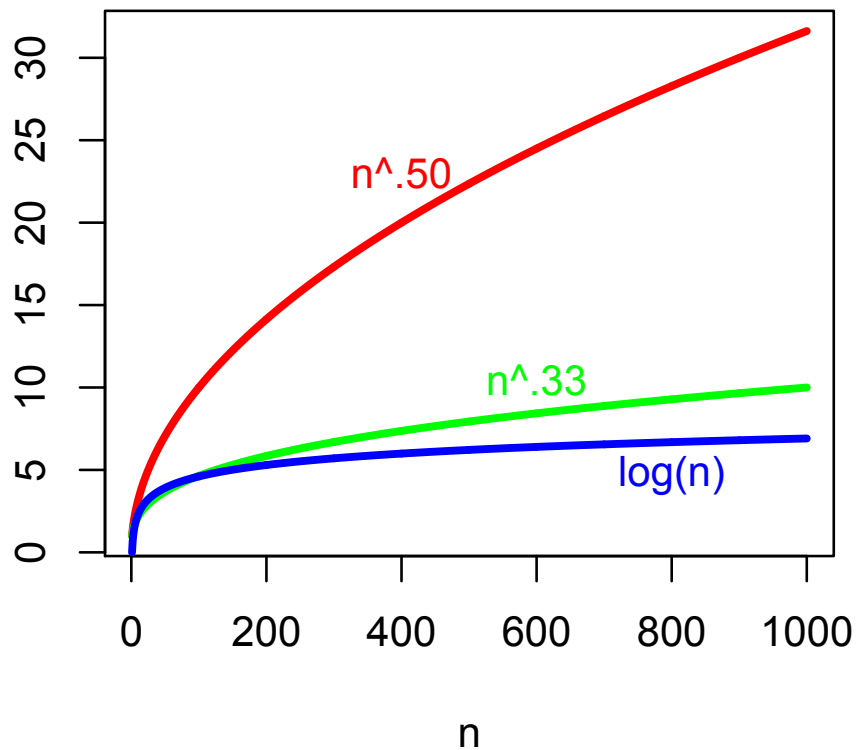
Corollary: base of a log factor is usually irrelevant, asymptotically. E.g. “ $O(n \log n)$ ” [but  $n^{\log_2 8} \neq O(n^{\log_8 8})$ ]

# polynomial vs logarithm

Logarithms:

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

*log grows slower than every polynomial*



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

$f(n) = O(g(n))$  vs  $f(n) = o(g(n))$  are analogs to  $\leq$  vs  $<$

Note:

if  $f(n)$  is  $\Theta(g(n))$  then it cannot be  $o(g(n))$

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

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

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

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

## polynomial vs exponential

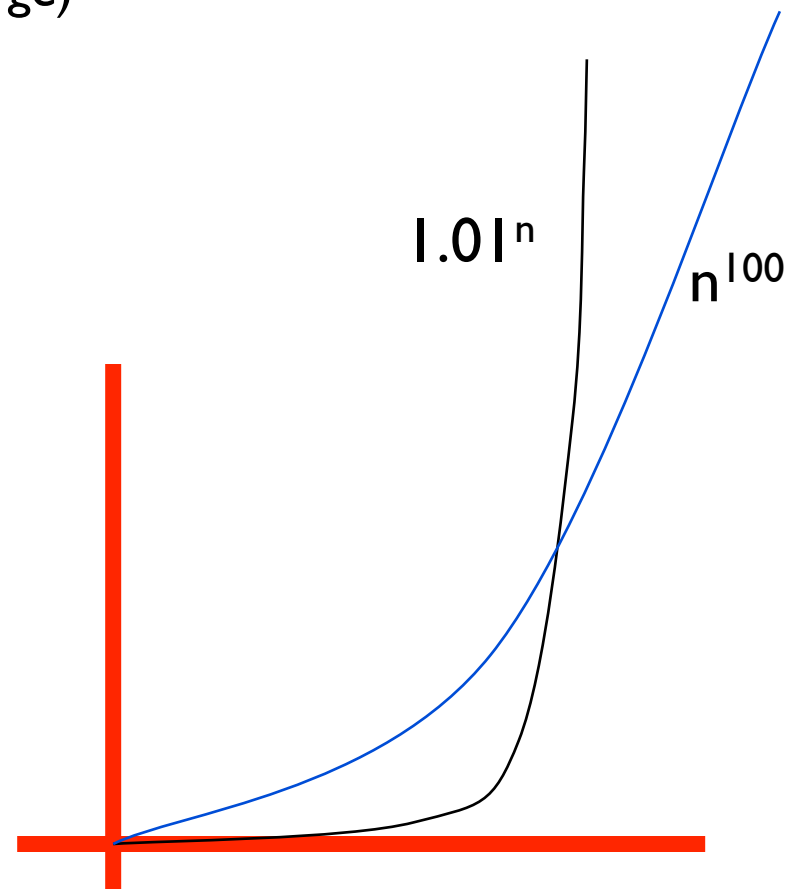
---

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

$n^d = o(r^n)$ , even

Exercise: prove this, using  
tricks from previous slide

In short, every exponential  
grows faster than every  
polynomial!



Given two functions  $f(n)$  and  $g(n)$ , if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \begin{cases} c & \text{for some constant } c > 0 \\ 0 \end{cases},$$

then  $\left\{ \begin{array}{l} f(n) = \Theta(g(n)) \\ f(n) = o(g(n)) [\Rightarrow O(g(n))] \end{array} \right\}$ , respectively.

Inconclusive if the limit doesn't exist. E.g., no limit for  $f/g$  at right, but  $g(n) \leq f(n) = O(f(n))$

$$\begin{array}{l} f(n) = \begin{cases} n & \text{if } n \text{ is even} \\ n^2 & \text{otherwise} \end{cases} \\ g(n) = n \end{array}$$



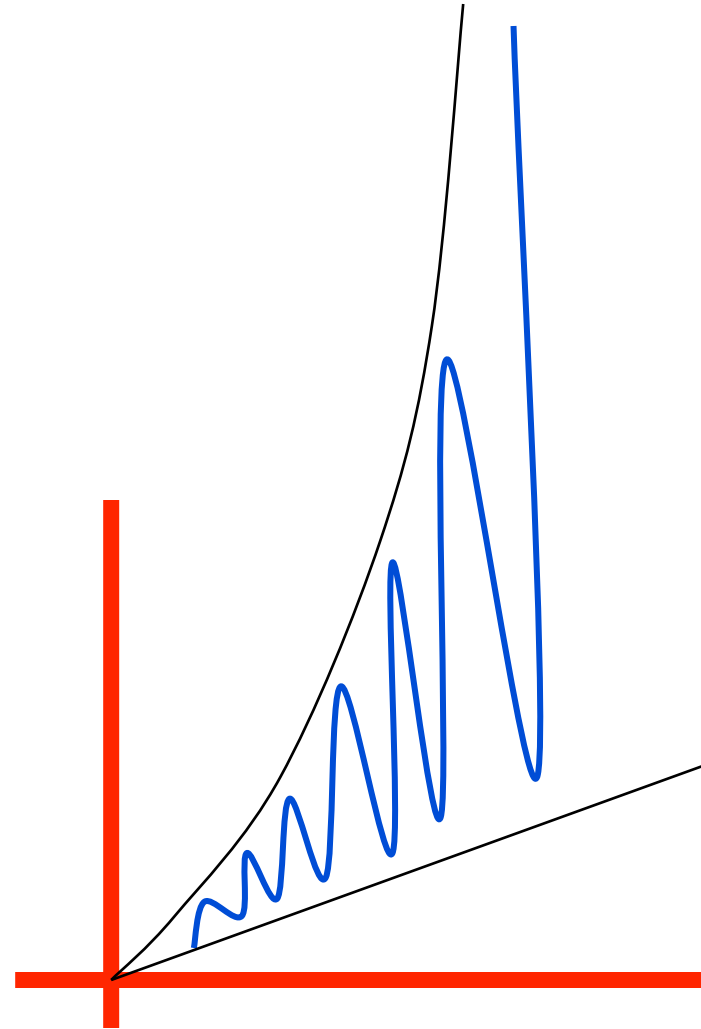
## big-theta, etc. are not always “nice”

---

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

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

Fortunately, such nasty cases are rare



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

“Theorem”:  $\sum_{1 \leq i \leq n} i = O(n)$

“Proof:” (by induction on  $n$ )

basis:  $\sum_{1 \leq i \leq 1} i = 1 = O(1)$

induction step:

$$\begin{aligned} \sum_{1 \leq i \leq n} i &= \left( \sum_{1 \leq i \leq n-1} i \right) + n \\ &= O(n-1) + n \quad (\text{by ind. hyp.}) \\ &= O(n) \end{aligned}$$

**FALSE!**

Q. Where’s the flaw??

A. Never use “big-O” like this in an induction; instead, explicitly show the implicit constant “c”; in the above “proof,” you’ll see “c” become “c+1”...

## “One-Way Equalities”

---

2 + 2 is 4

2 + 2 = 4

4 = 2 + 2

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

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

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

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 less common, notation:  $T(n) \in O(f(n))$ .

I.e.,  $O(f(n))$  is the *set of all functions* that grow no more rapidly than some constant times  $f$ .

Replace “=” by “ $\in$ ” or “ $\subseteq$ ” as appropriate: e.g.:

$$2n^2 + 5n \in O(n^2) \subseteq O(n^3)$$

---

## Polynomial Time

## the complexity class P: 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  $c$ .

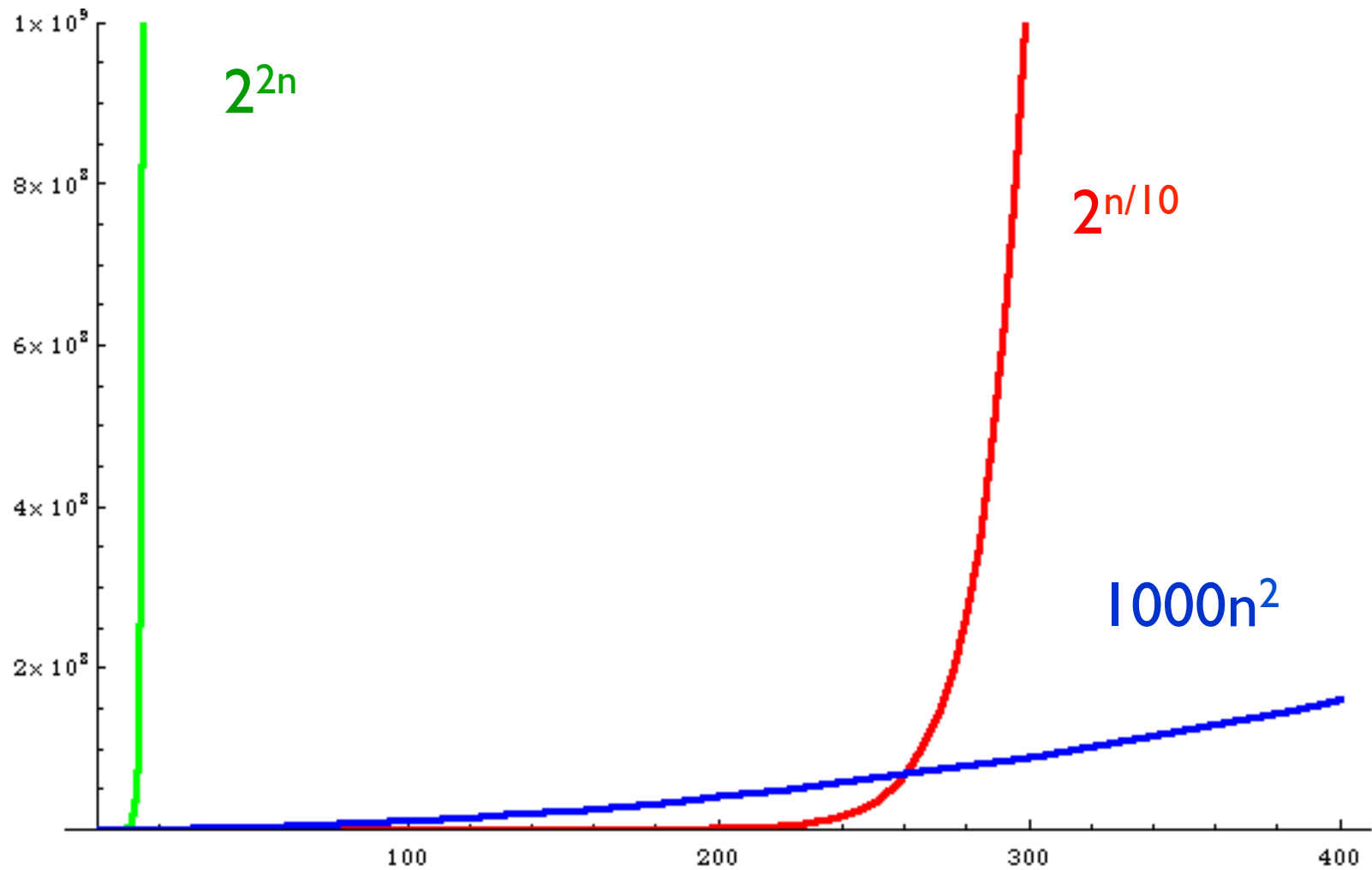
(E.g.,  $c \sim 2^d$ )

**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

---



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



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

Big  $O$  is a *math* notation defining an *upper bound* on *growth rate* of a function (typically a function lacking a simple analytic formula.)

In CS, that function is often the *worst case run time* of some algorithm (as a function of input size,  $n$ , where “worst case” means max time over all inputs of size  $n$ .)

BUT, it can also be used for other functions, like best- or average-case time/space/..., so be clear/careful re defn.

Big  $\Omega$  is analogous math notation for lower bounds

Big  $\Theta$ : upper and lower bounds simultaneously

These notations deliberately define growth rate only *up to a (hidden) constant factor*, essentially because (a) scaling matters more than the constant, and (b) the constant is strongly technology-dependent (language, code, compiler, processor, ...) making it much more work to pin down.

So, a typical initial goal for algorithm analysis is to find a

reasonably tight,	←	i.e., $\Theta$ if possible
asymptotic,	←	i.e., $O$ or $\Theta$
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.