Algorithms and Computational Complexity: an Overview

Autumn 2011

Larry Ruzzo

Thanks to Paul Beame, James Lee, Kevin Wayne for some slides

Design of Algorithms – a taste

- design methods
- common or important types of problems
- analysis of algorithms - efficiency

Complexity & intractability – a taste

- solving problems in principle is not enough
  - algorithms must be efficient
- some problems have no efficient solution
- NP-complete problems
  - important & useful class of problems whose solutions (seemingly) cannot be found efficiently

Cryptography (e.g. RSA, SSL in browsers)

Secret: p,q prime, say 512 bits each

Public: n which equals p x q, 1024 bits

In principle

there is an algorithm that given n will find p and q:
try all $2^{512} \approx 1.3 \times 10^{154}$ possible p's (but that's kinda big…
for comparison, the age of the universe is $\approx 5 \times 10^{29}$ picosec)

In practice

no fast algorithm known for this problem (on non-quantum computers)

security of RSA depends on this fact

(and research in "quantum computing" is strongly driven by the possibility of changing this)

Moore's Law and the exponential improvements in hardware...

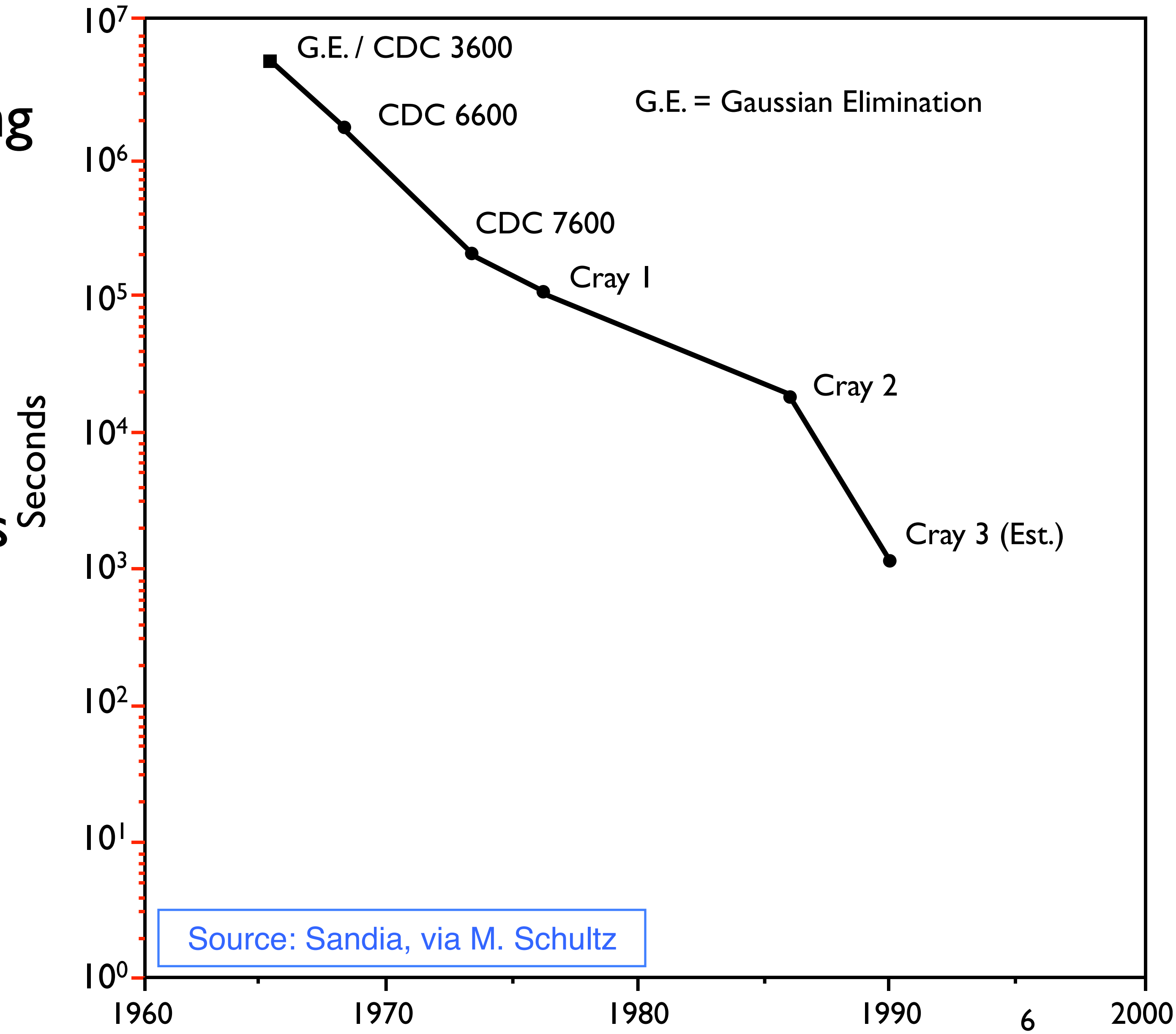
Ex: sparse linear equations over 25 years

10 orders of magnitude improvement!

algorithms or hardware?

25 years progress solving sparse linear systems

Hardware alone: 4 orders of magnitude

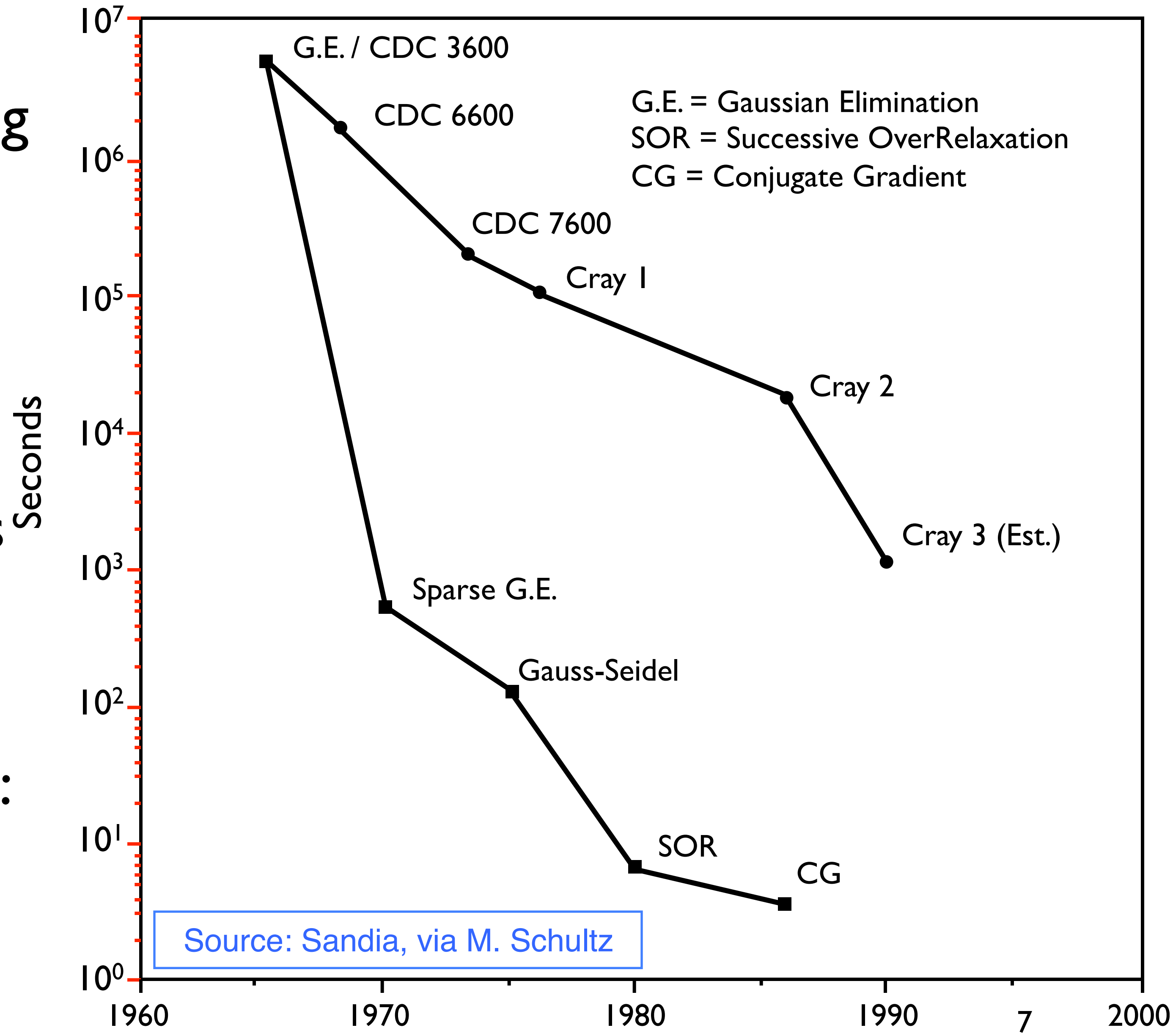


algorithms or hardware?

25 years progress solving sparse linear systems

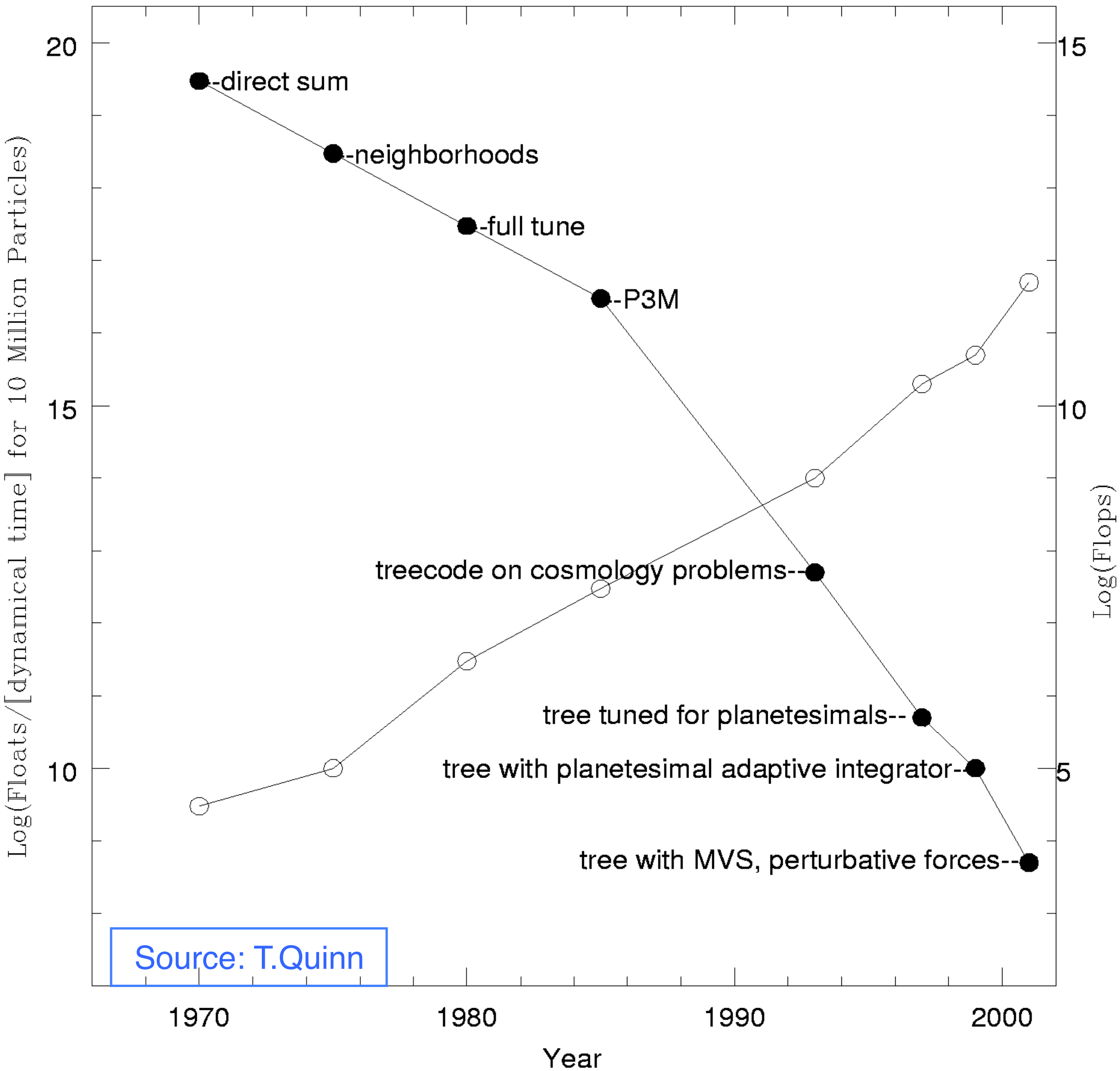
Hardware alone: 4 orders of magnitude

Software alone: 6 orders of magnitude



The N-Body Problem:

in 30 years
- 10^7 hardware
- 10^{10} software



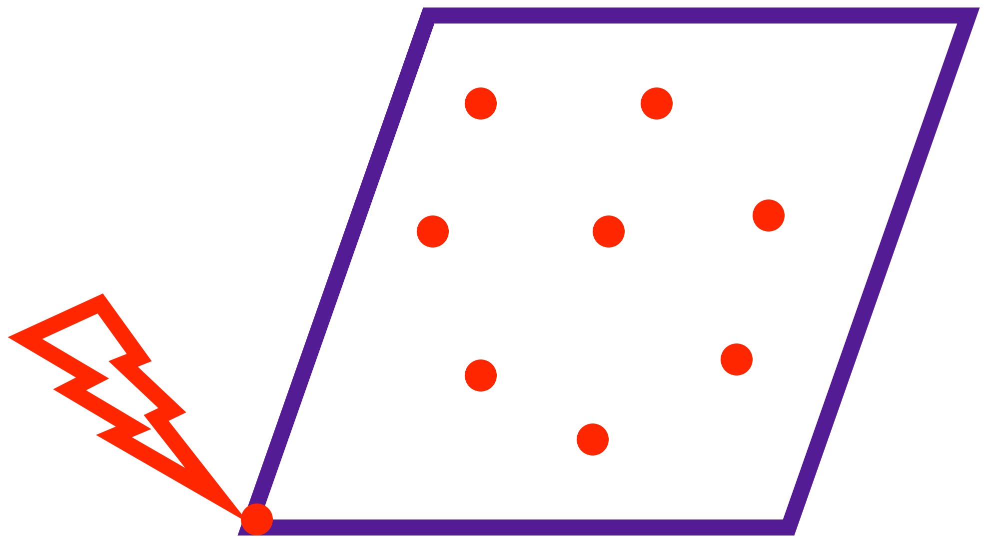
Procedure to accomplish a task or solve a well-specified problem

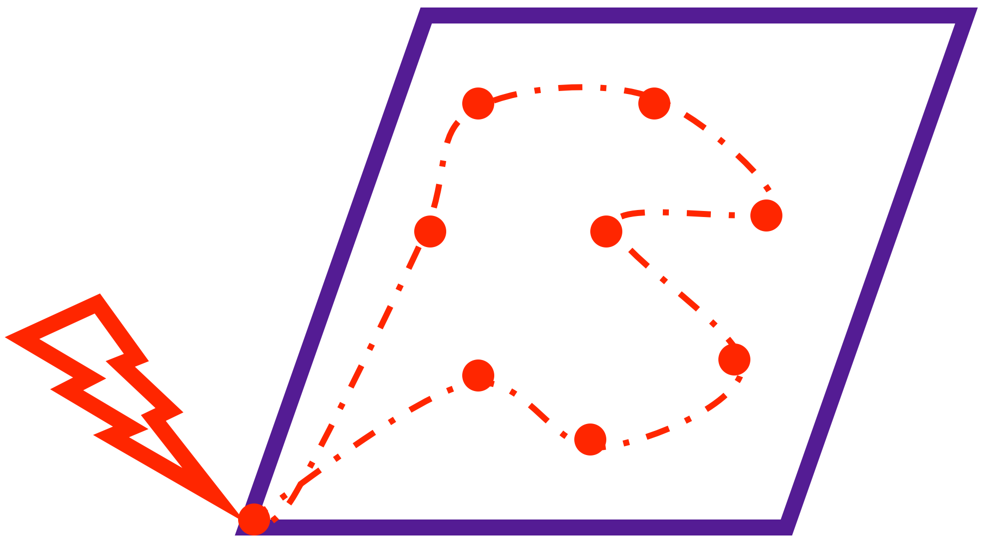
- Well-specified: know what all possible inputs look like and what output looks like given them
- “accomplish” via simple, well-defined steps
- Ex: sorting names (via comparison)
- Ex: checking for primality (via +, -, *, /, ≤)

Printed circuit-board company has a robot arm that solders components to the board

Time: proportional to total distance the arm must move from initial rest position around the board and back to the initial position

For each board design, find best order to do the soldering





Input: Given a set *S* of *n* points in the plane

Output: The shortest cycle tour that visits each point in the set *S*.

Better known as "TSP"

How might you solve it?

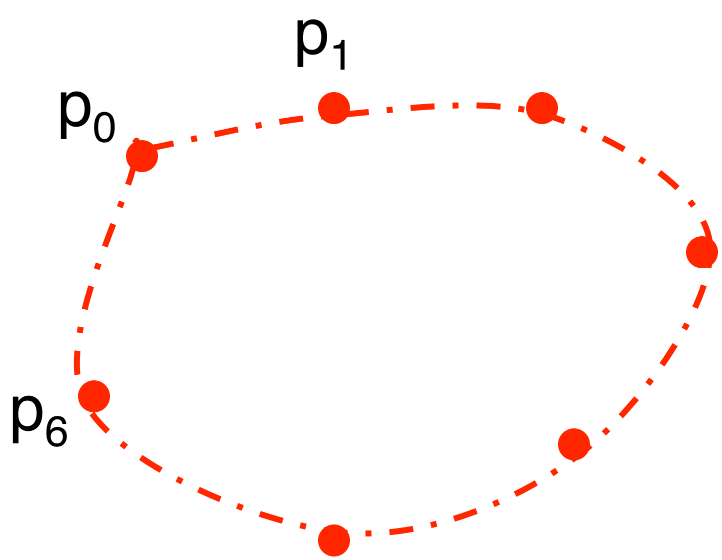
Start at some point p_0

Walk first to its nearest neighbor p_1

Walk to the nearest unvisited neighbor p_2, then nearest unvisited p_3, … until all points have been visited

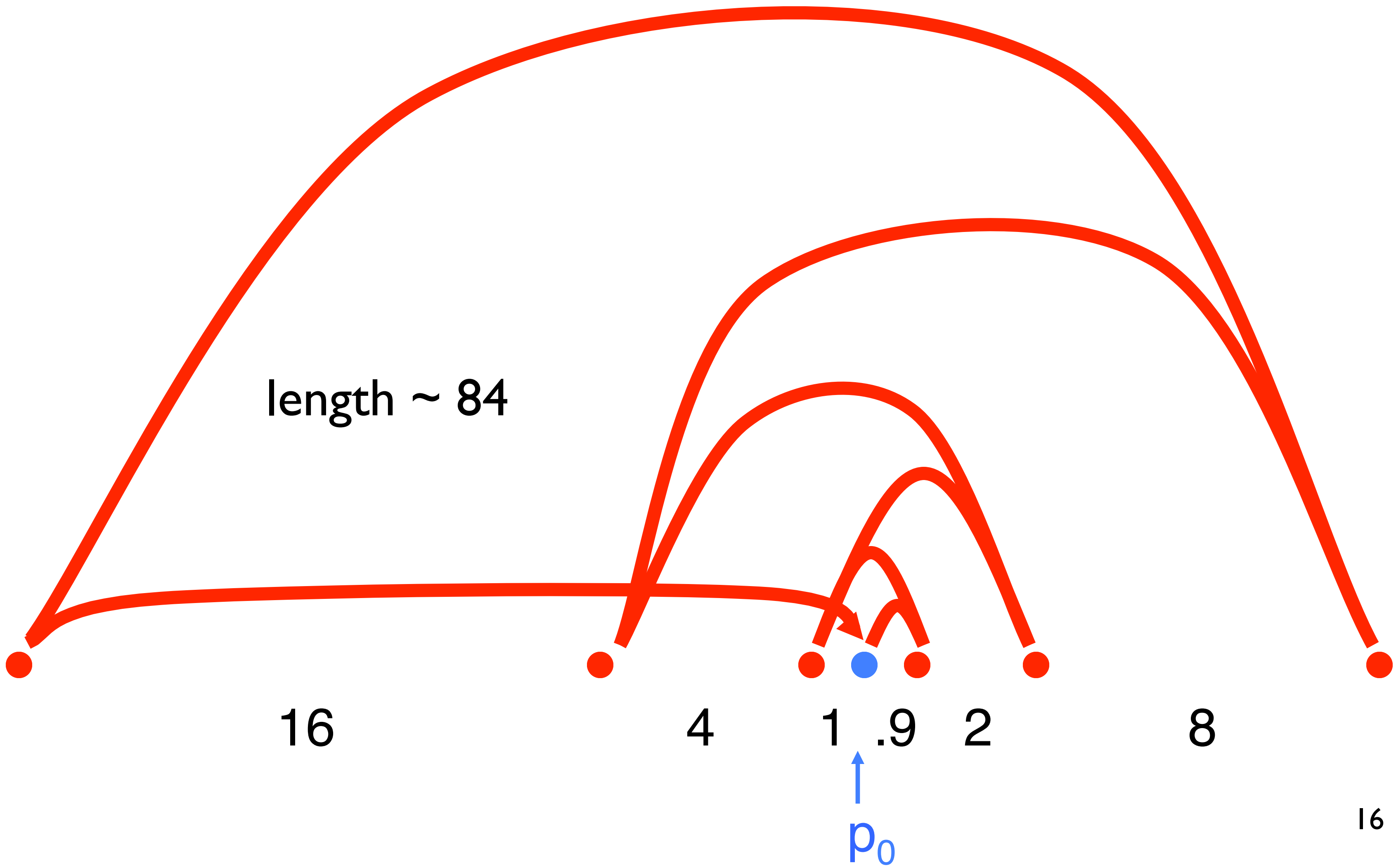
Then walk back to p_0

heuristic:
A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. May be good, but usually *not* guaranteed to give the best or fastest solution.
(And often difficult to analyze precisely.)

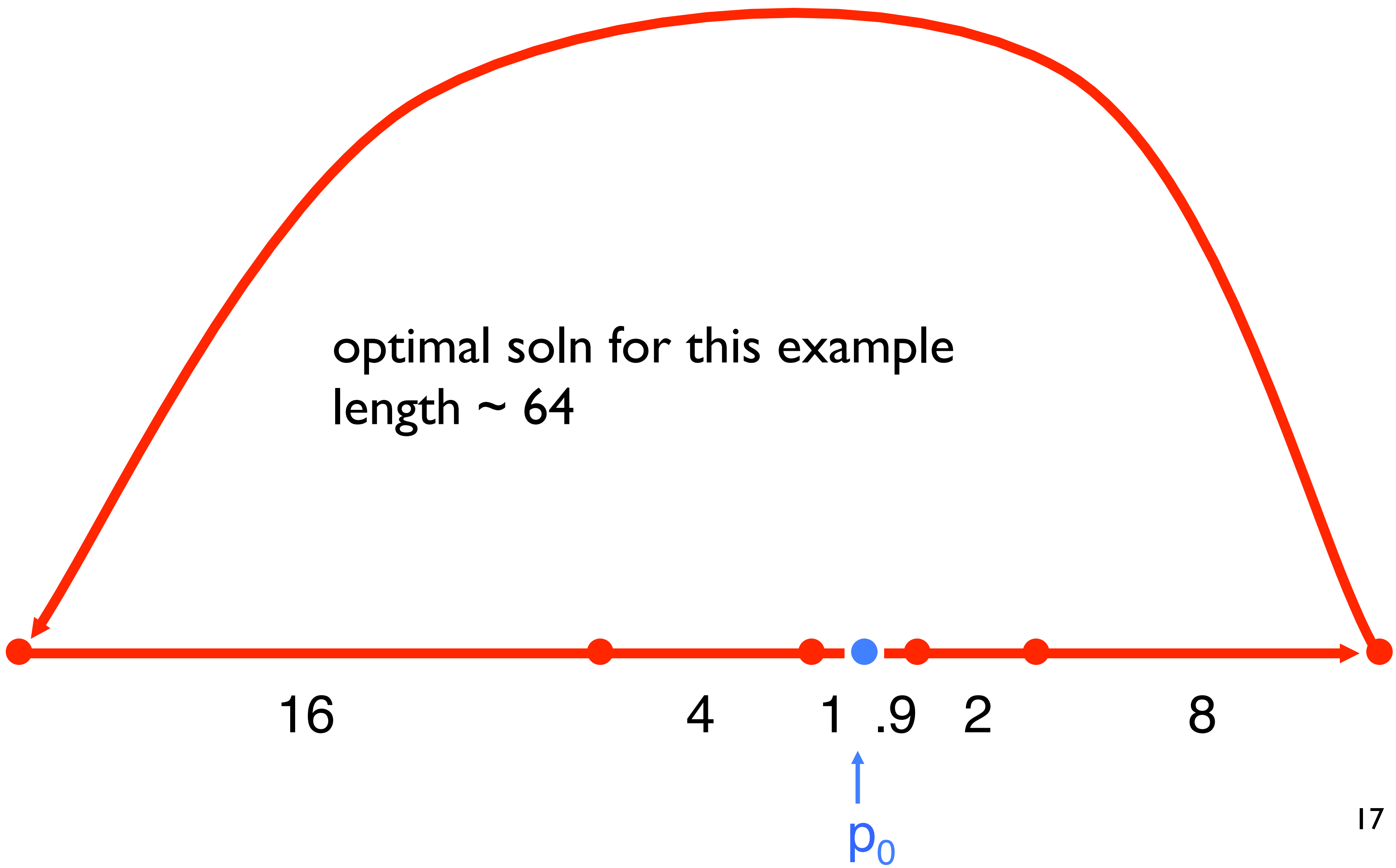


p_1
p_0
p_6

an input where nn works badly

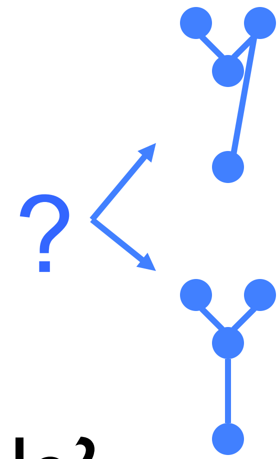


an input where nn works badly

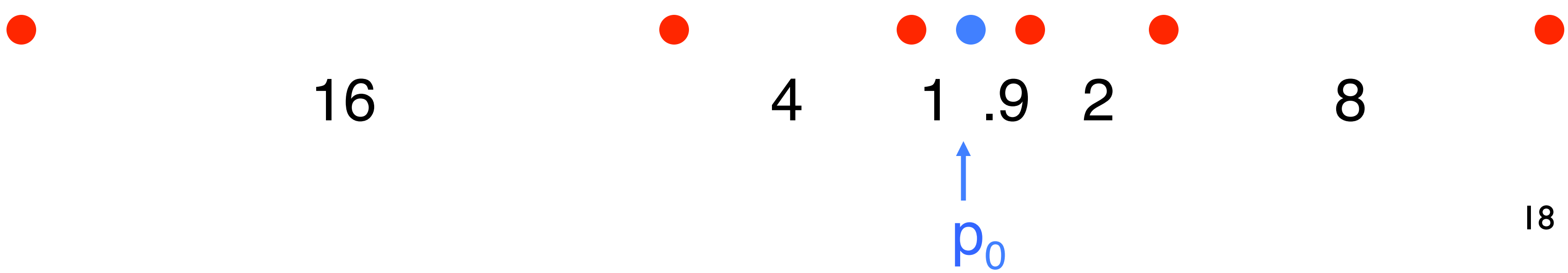


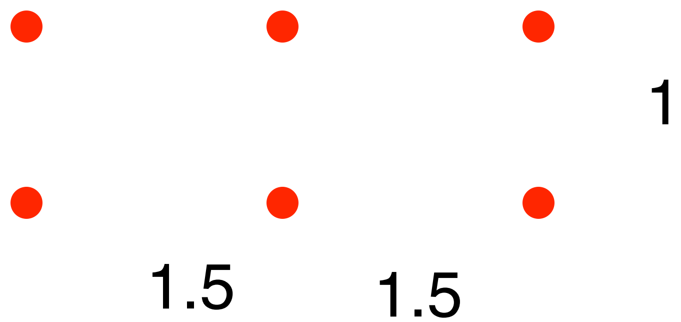
Repeatedly join the closest pair of points

(such that result can still be part of a single loop in the end. I.e., join endpoints, but not points in middle, of path segments already created.)

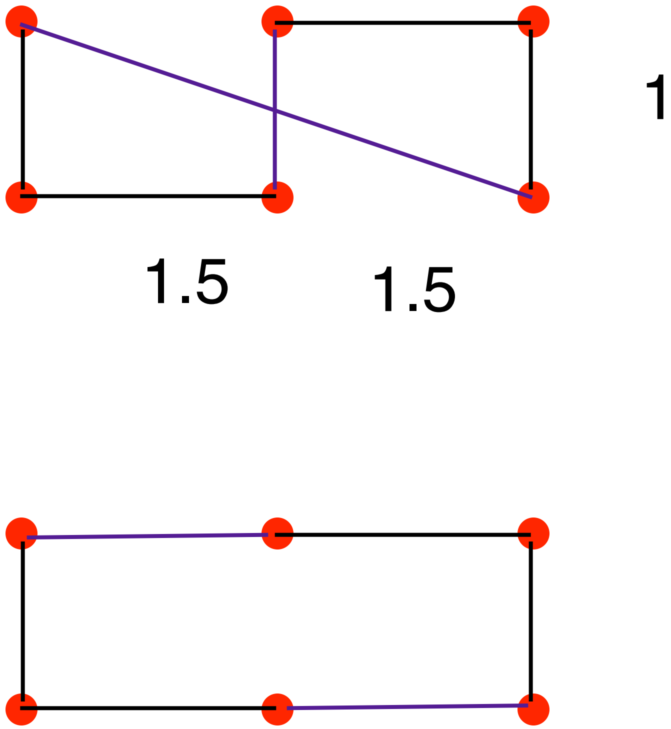


How does this work on our bad example?




1
1.5
1.5

a bad example for closest pair



$6+\sqrt{10} = 9.16$

vs

8

"Brute Force Search":

For each of the $n! = n(n-1)(n-2)\ldots 1$ orderings of the points, check the length of the cycle;

Keep the best one

The two *in*correct algorithms were greedy

- Often very natural & tempting ideas
- They make choices that look great "locally" (and never reconsider them)
- When greed works, the algorithms are typically efficient
- BUT: often does not work - you get boxed in

Our correct alg avoids this, but is *incredibly* slow

- 20! is so large that checking one billion per second would take 2.4 billion seconds (around 70 years!)
- And *growing*: $n! \sim \sqrt{2 \pi n} \cdot (n/e)^n \sim 2^{O(n \log n)}$

- Algorithms are important
  - Many performance gains outstrip Moore's law
- Simple problems can be hard
  - Factoring, TSP, *many* others
- Simple ideas don't always work
  - Nearest neighbor, closest pair heuristics
- Simple algorithms can be very slow
  - Brute-force factoring, TSP
- A point we hope to make: for some problems, even the *best* algorithms are slow

A brief overview of the theory of algorithms

- Efficiency & asymptotic analysis
- Some scattered examples of simple problems where clever algorithms help

A brief overview of the theory of intractability

- Especially NP-complete problems

"Basics every educated CSE student should know"

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

Mathematically,

$T: N+ \rightarrow R+$

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

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. $T(n) = O(n^2)$

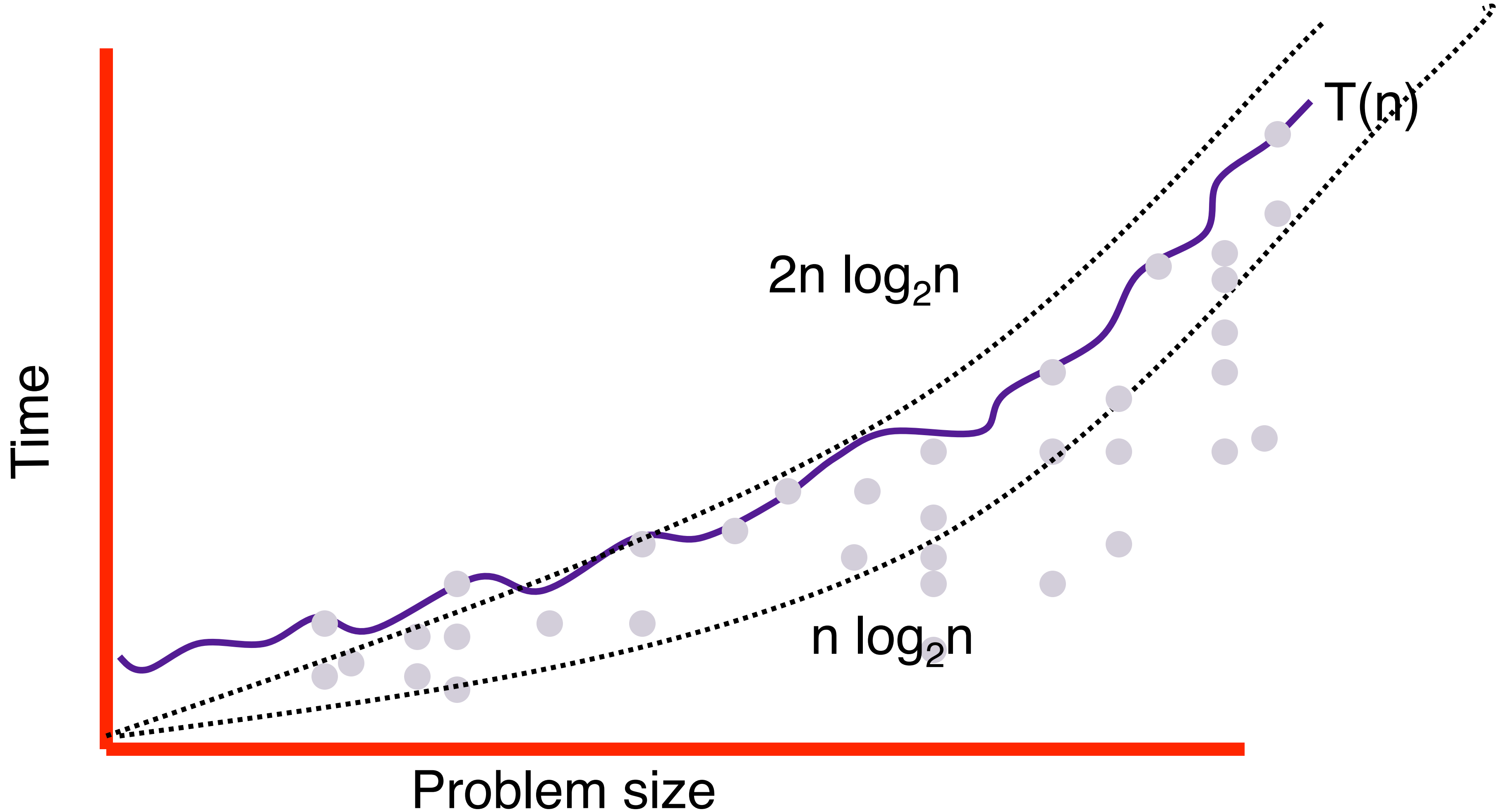
Why not try to be more precise?

Average-case, e.g., is hard to define, analyze

Technological variations (computer, compiler, OS, …) easily 10x or more

Being more precise is a ton of work

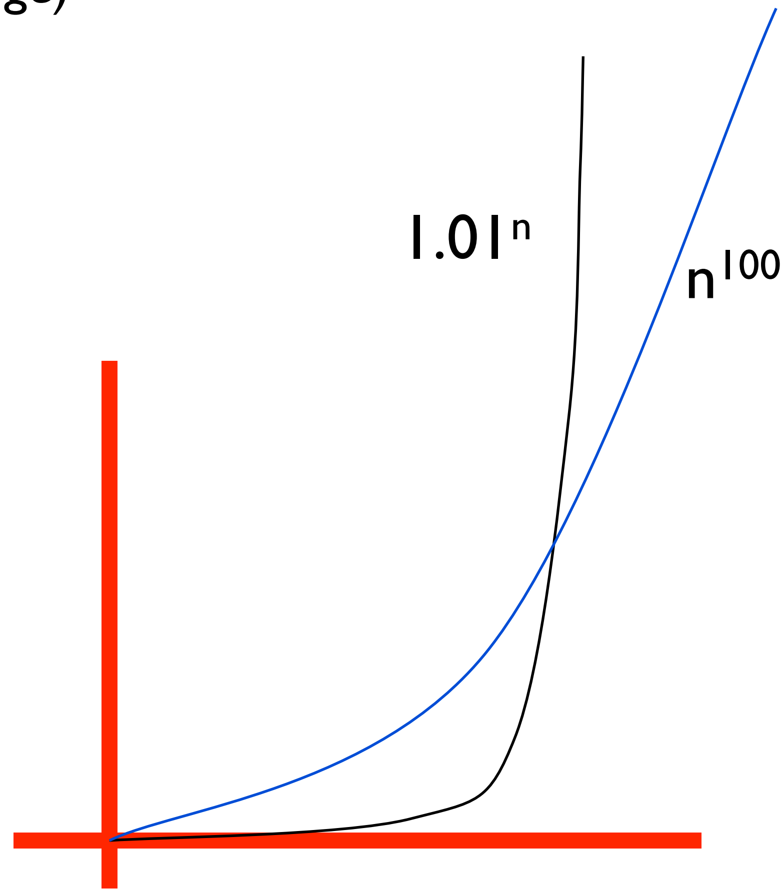
A key question is "scale up": if I can afford this today, how much longer will it take when my business is 2x larger? (E.g. today: cn^2, next year: $c(2n)^2 = 4cn^2$: 4 x longer.) Big-O analysis is adequate to address this.



T(n)
$2n \log_2 n$
$n \log_2 n$
Time
Problem size

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!



P: Running time $O(n^d)$ for some constant d

(d is 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

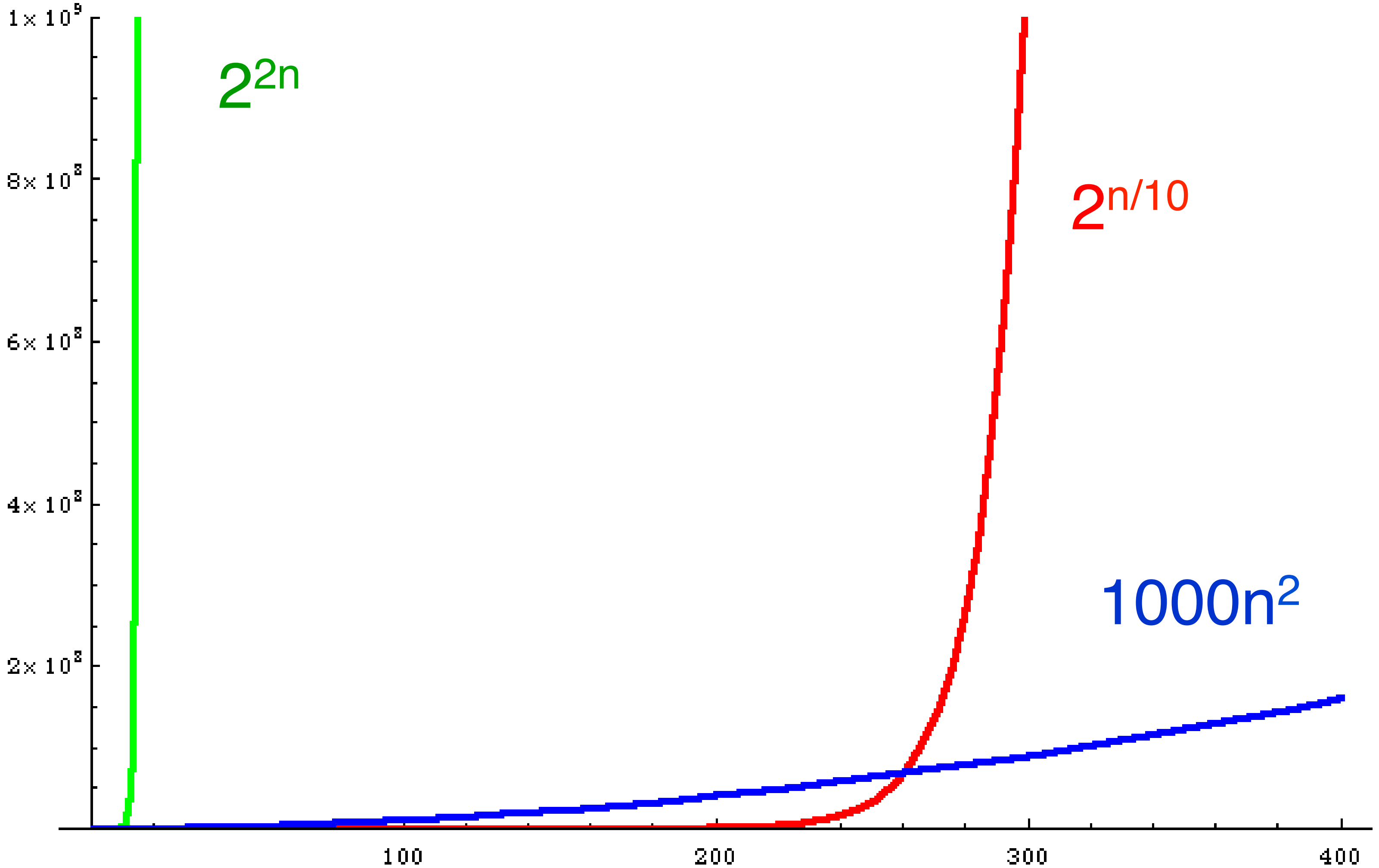


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	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	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

Typical initial goal for algorithm analysis is to find an

- asymptotic
- upper bound on
- 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 - concentrate on the good ones!

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