Algorithms and Computational Complexity: an Overview

Winter 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} possible p's, but an astronomical number

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)

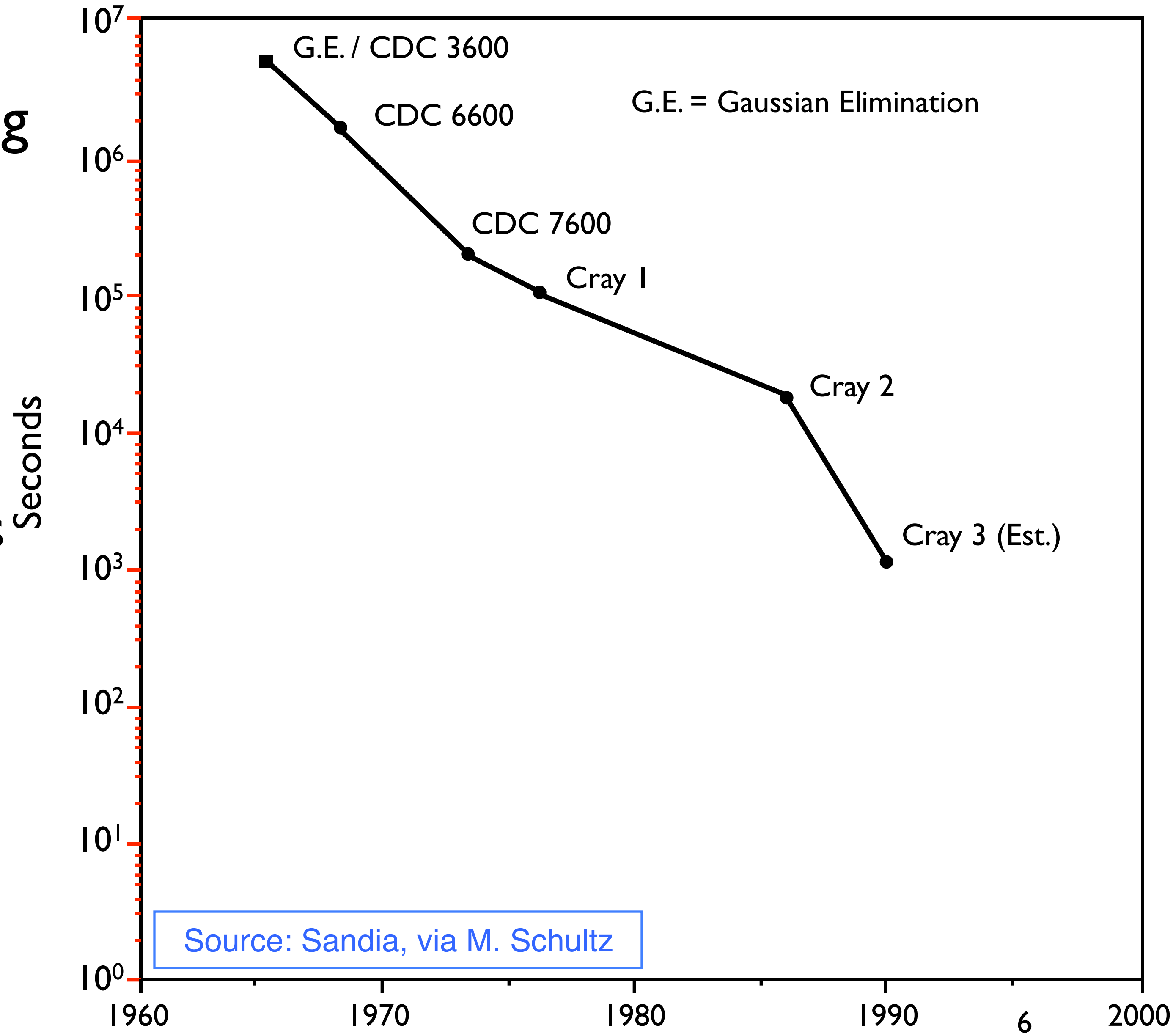
Moore's Law and the exponential improvements in hardware...

Ex: sparse linear equations over 25 years

10 orders of magnitude improvement!

25 years progress solving sparse linear systems

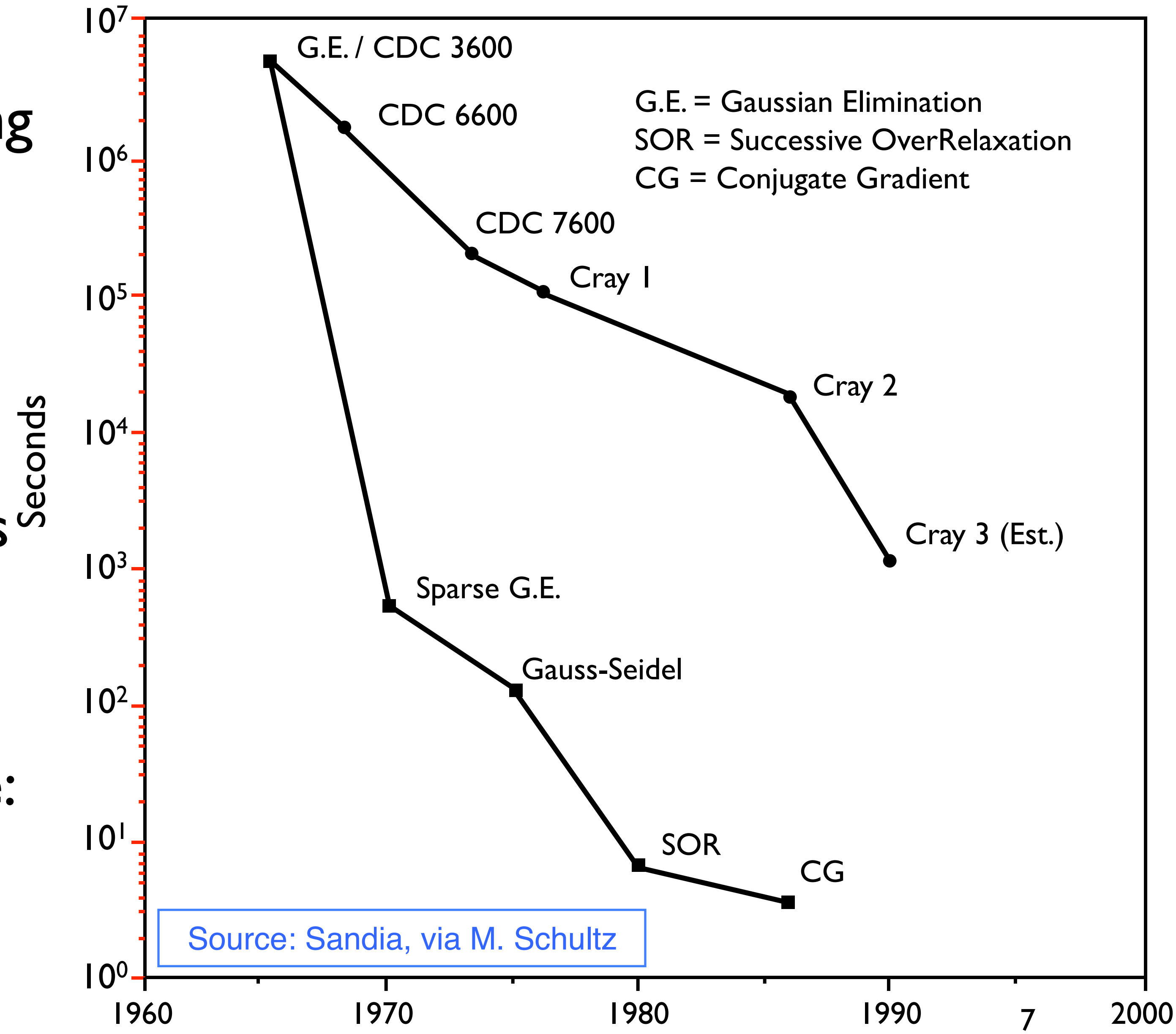
Hardware alone: 4 orders of magnitude



25 years progress solving sparse linear systems

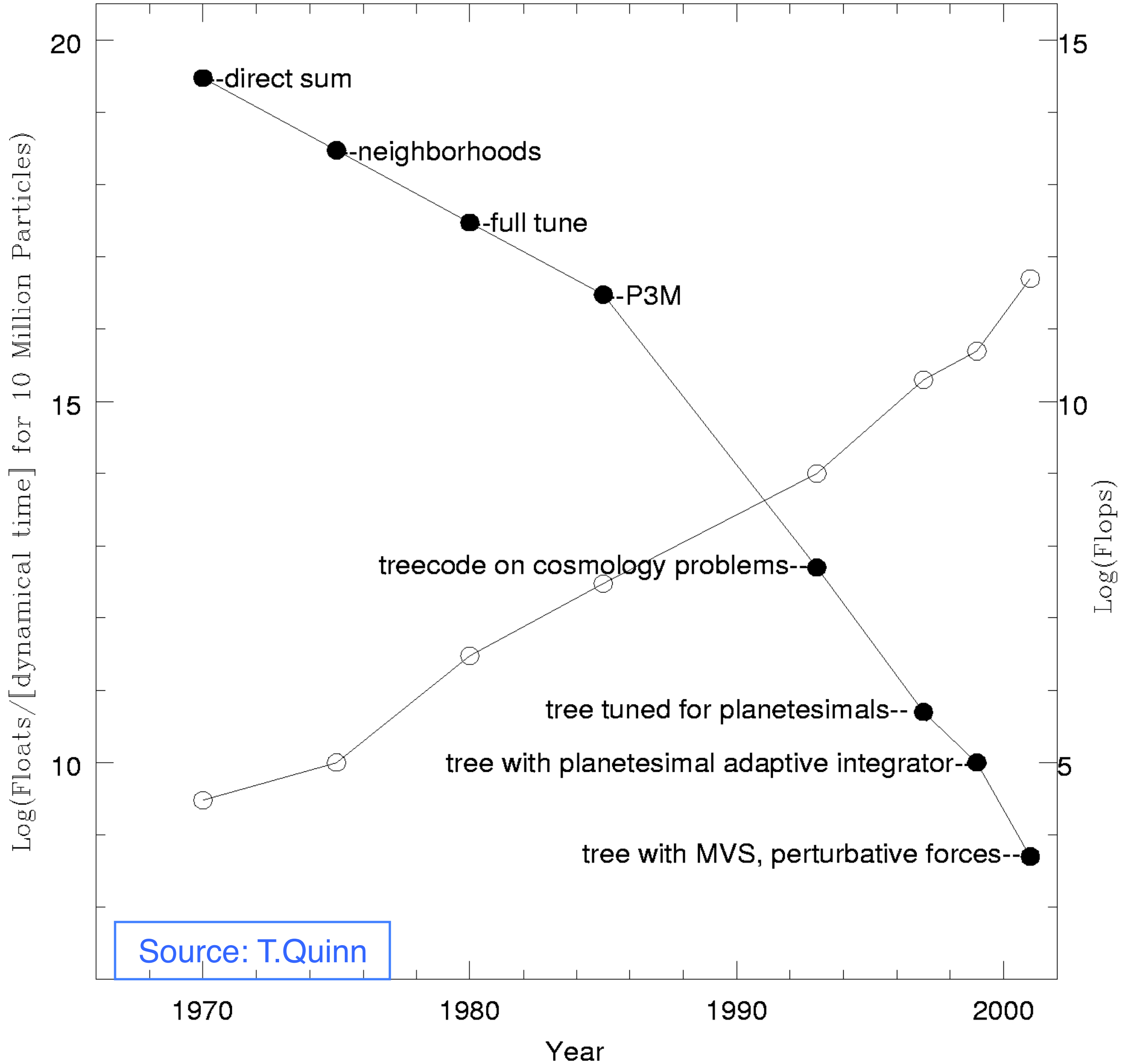
Hardware alone: 4 orders of magnitude

Software alone: 6 orders of magnitude



The N-Body Problem:

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



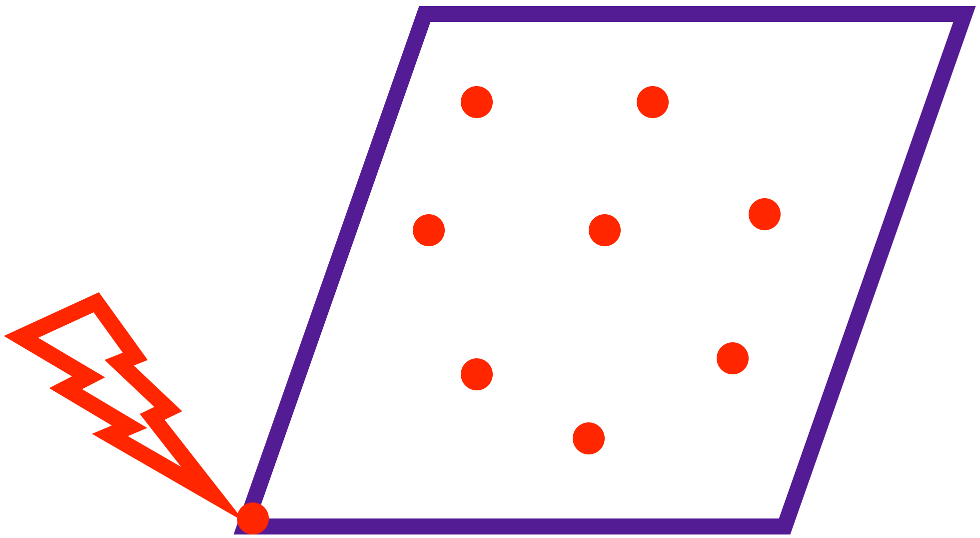
Procedure to accomplish a task or solve a well-specified problem

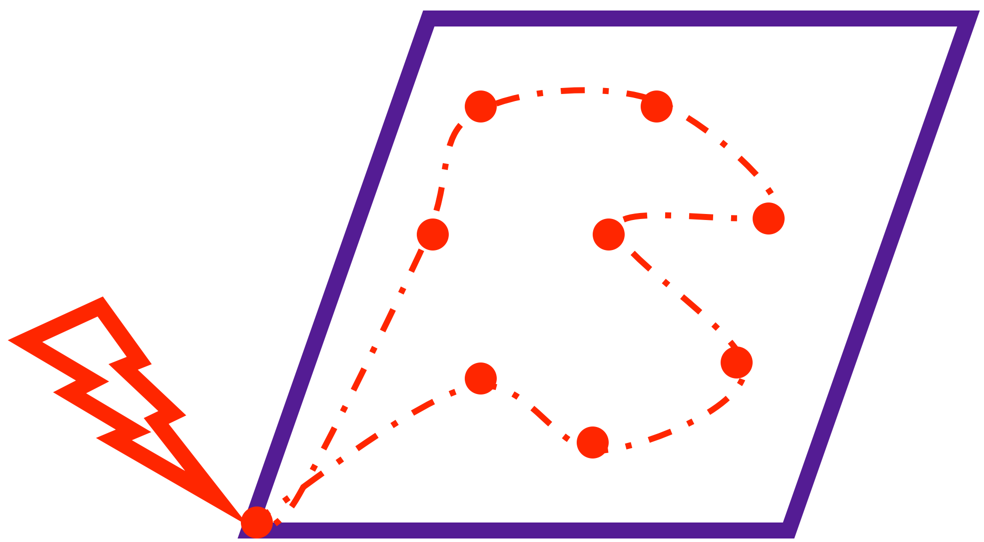
- Well-specified: know what all possible inputs look like and what output looks like given them
- "accomplish" via simple, well-defined steps
- Ex: sorting names (via comparison)
- Ex: checking for primality (via +, -, *, /, ≤)

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?

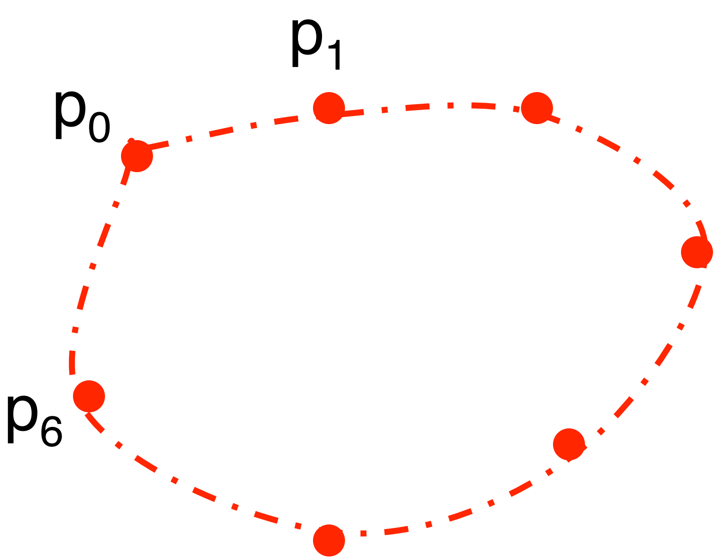
Start at some point p_0

Walk first to its nearest neighbor p_1

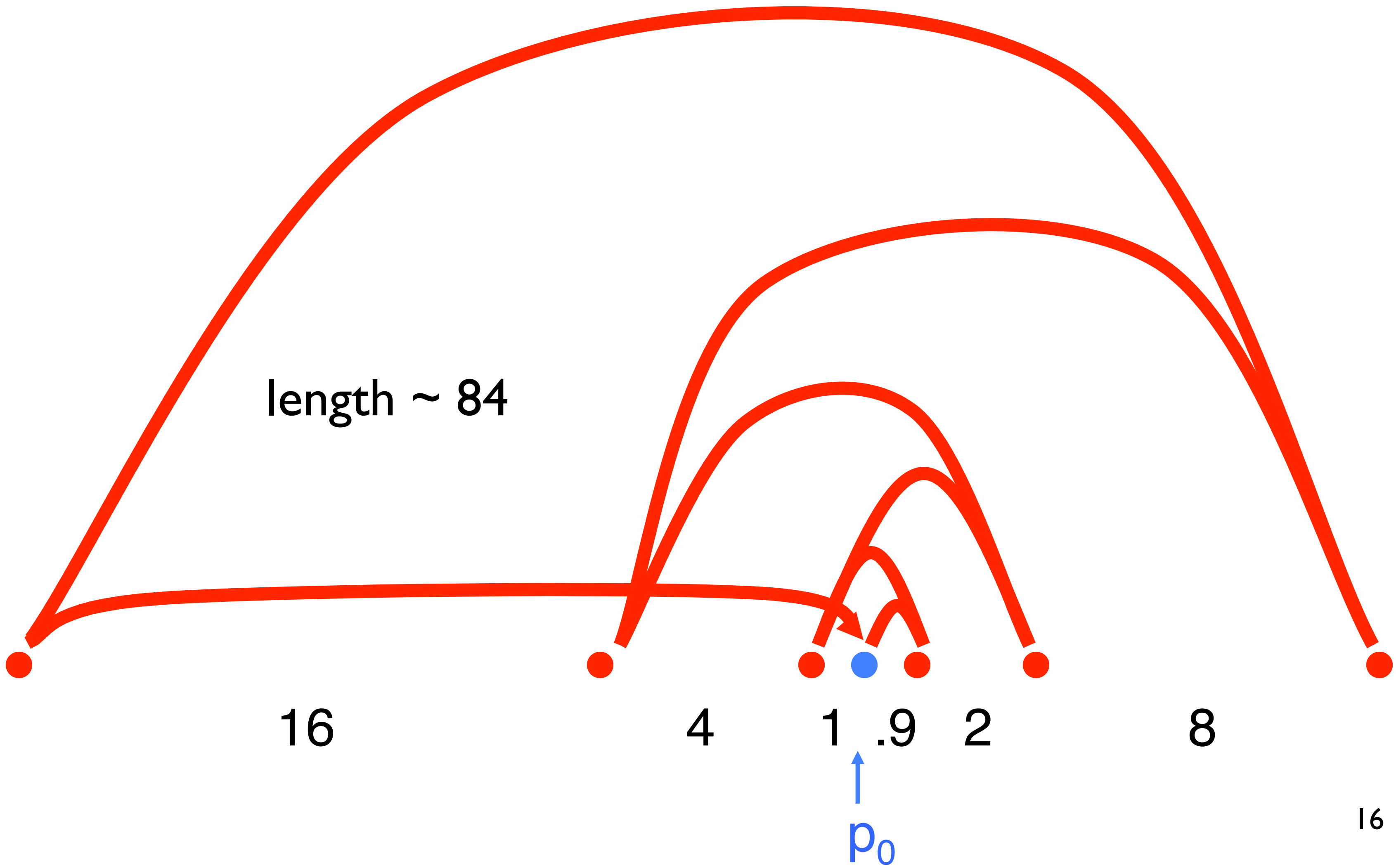
Walk to the nearest unvisited neighbor p_2, then nearest unvisited p_3, … until all points have been visited

Then walk back to p_0

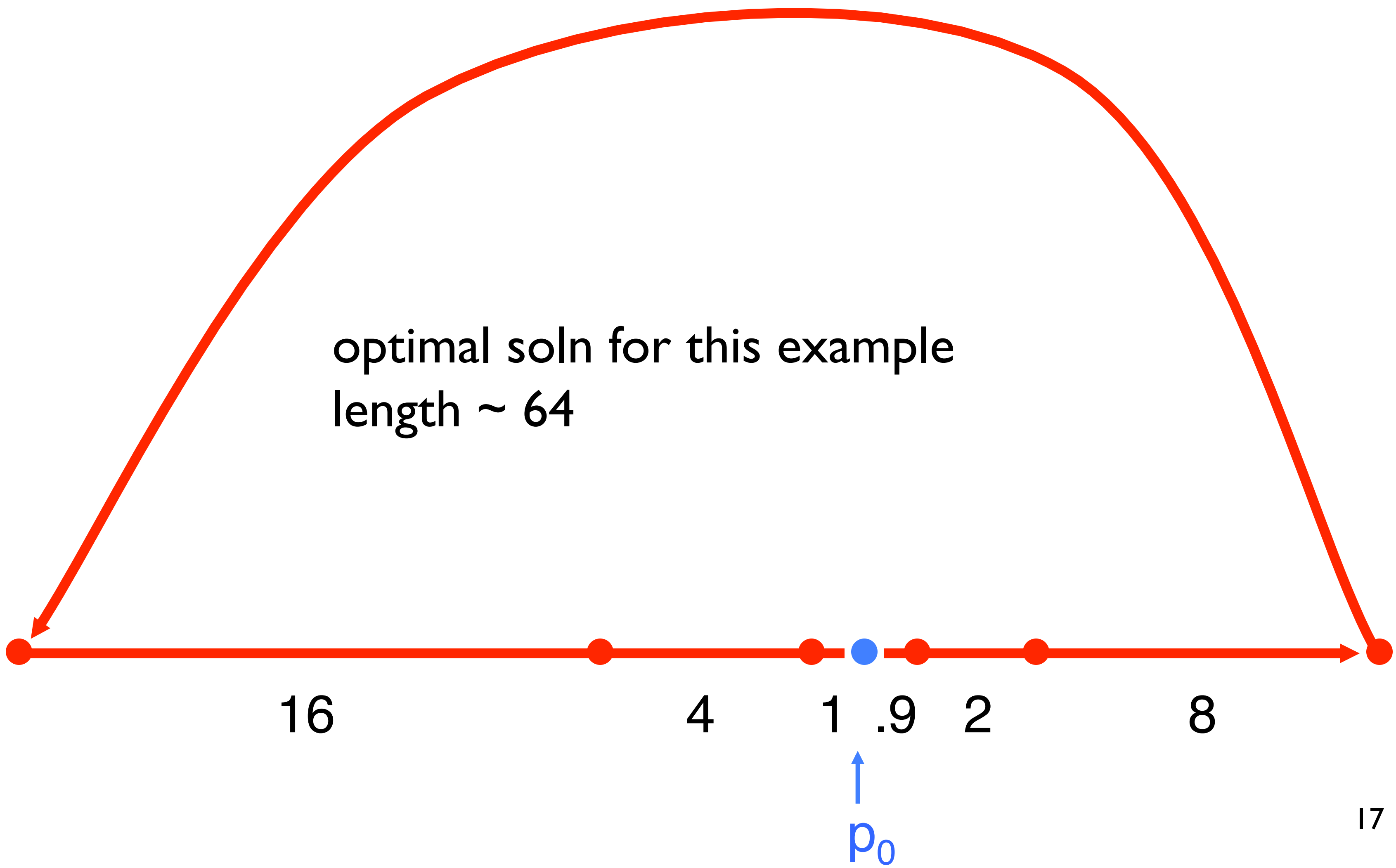
heuristic:
A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. May be good, but usually *not* guaranteed to give the best or fastest solution.


p1
p0
p6

an input where nn works badly

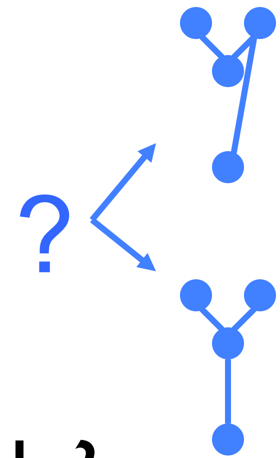


an input where nn works badly

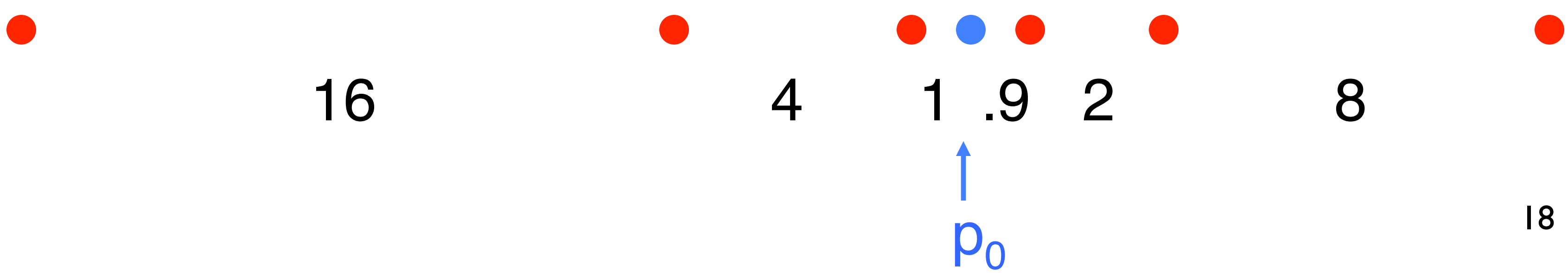


Repeatedly join the closest pair of points

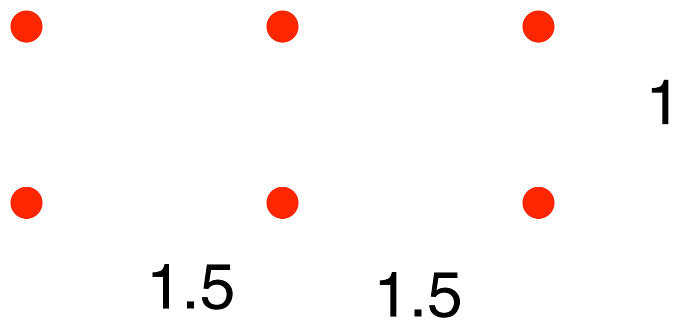
(such that result can still be part of a single loop in the end. I.e., join endpoints, but not points in middle, of path segments already created.)

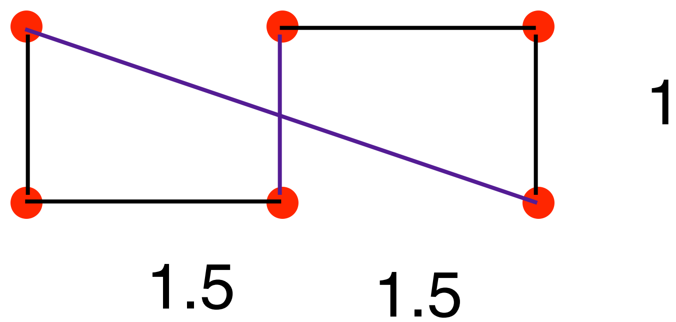


How does this work on our bad example?



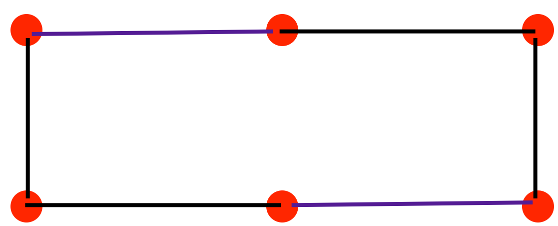
a bad example for closest pair





$6+\sqrt{10} = 9.16$

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

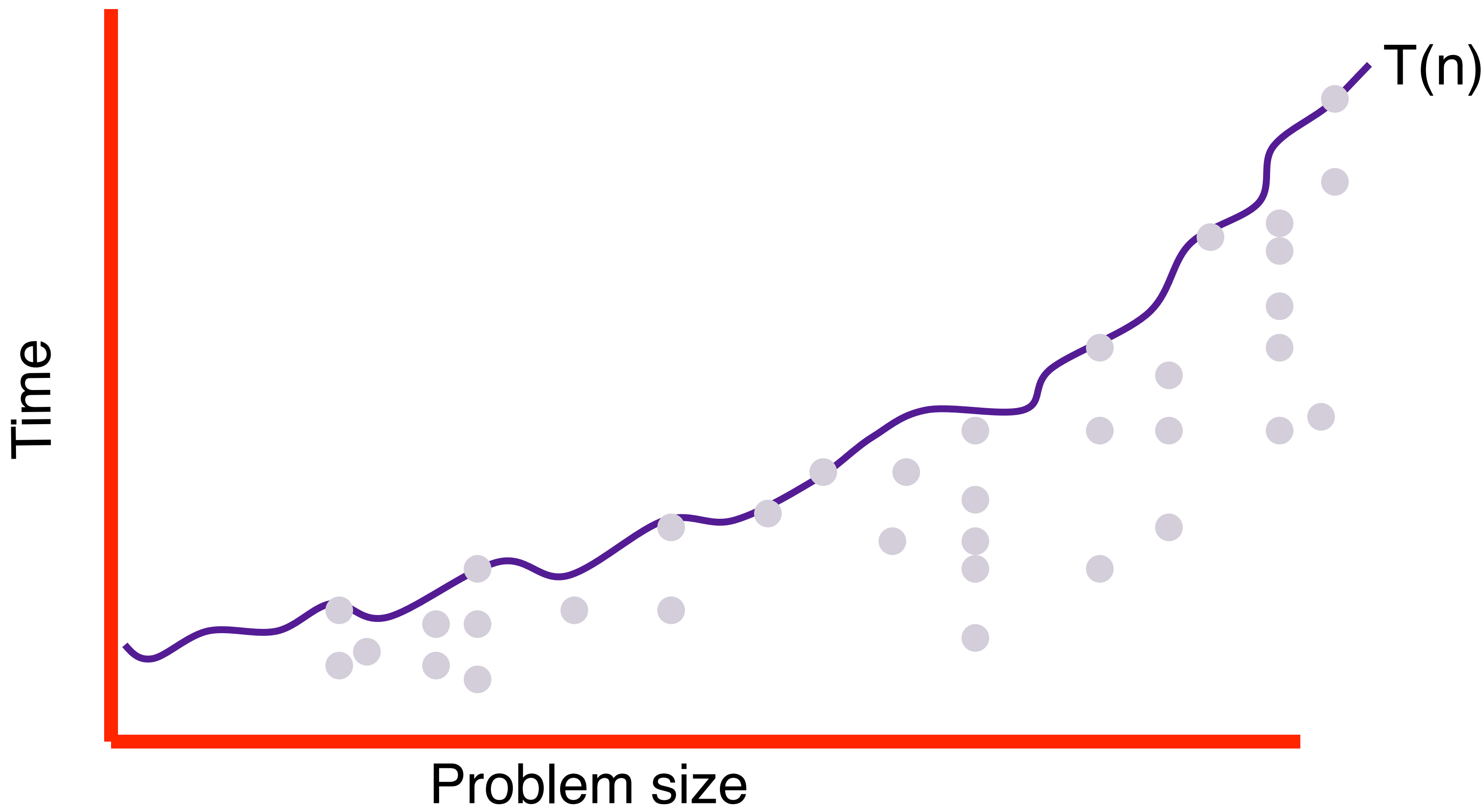
"Basics every educated CSE student should know"

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

Mathematically,

$T: N+ \rightarrow R+$

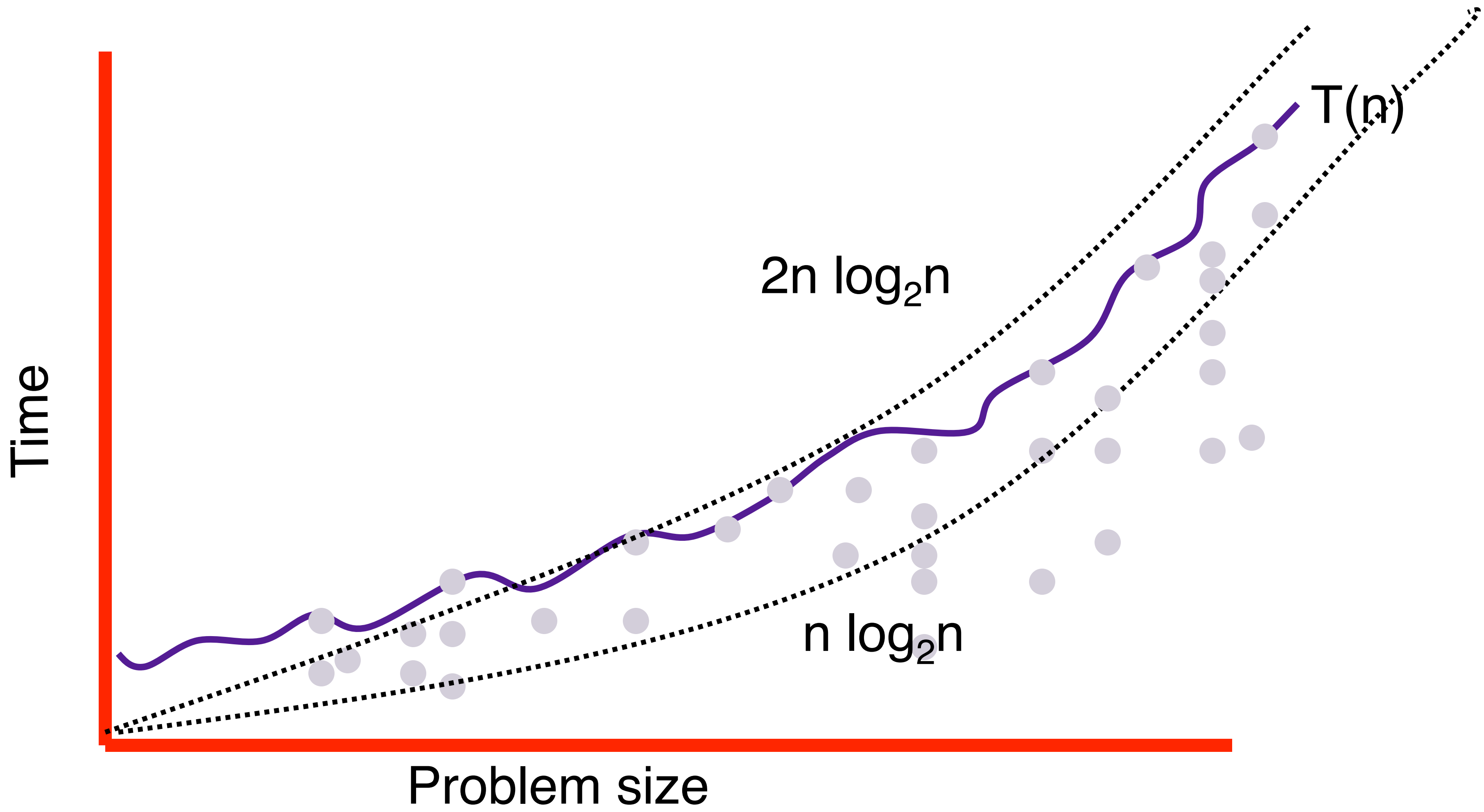
i.e., T is a function mapping positive integers (problem sizes) to positive real numbers (number of steps).


T(n)
Time
Problem size

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?

- 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.)



2n log2n
T(n)
n log2n
Time
Problem size

Given two functions f and g: $N \rightarrow R$, $f(n)$ is $O(g(n))$ iff

$\exists$ constant $c > 0$ so that $f(n)$ is eventually always $\leq c\, g(n)$

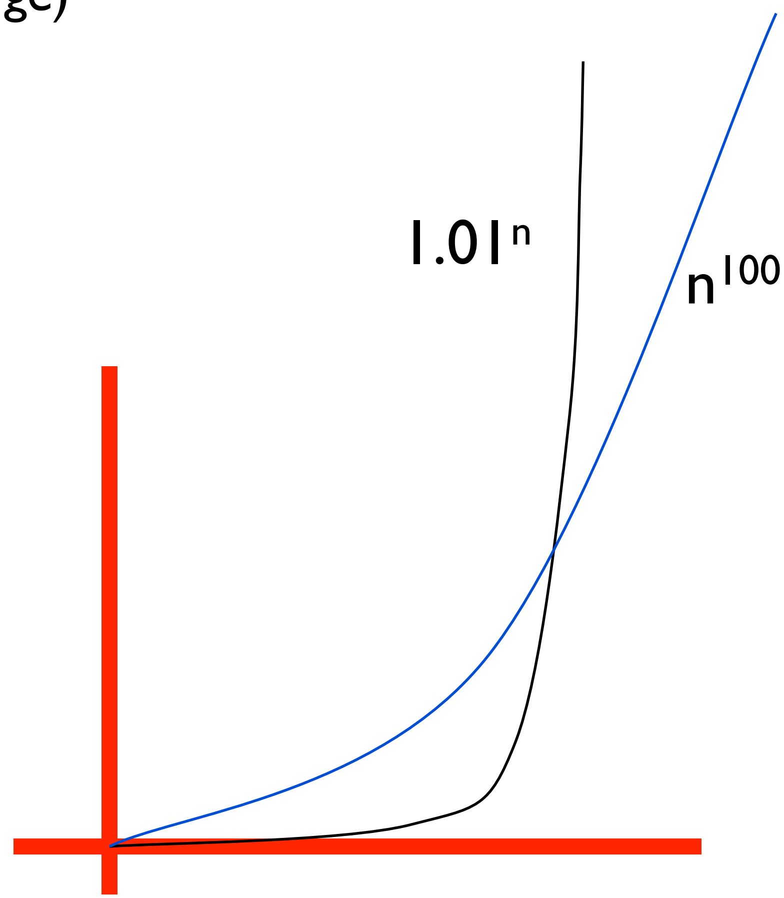
Example:

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

why?:

$10n^2-16n+100 \leq 11n^2$ for all $n \geq 10$

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

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

polynomial vs exponential growth

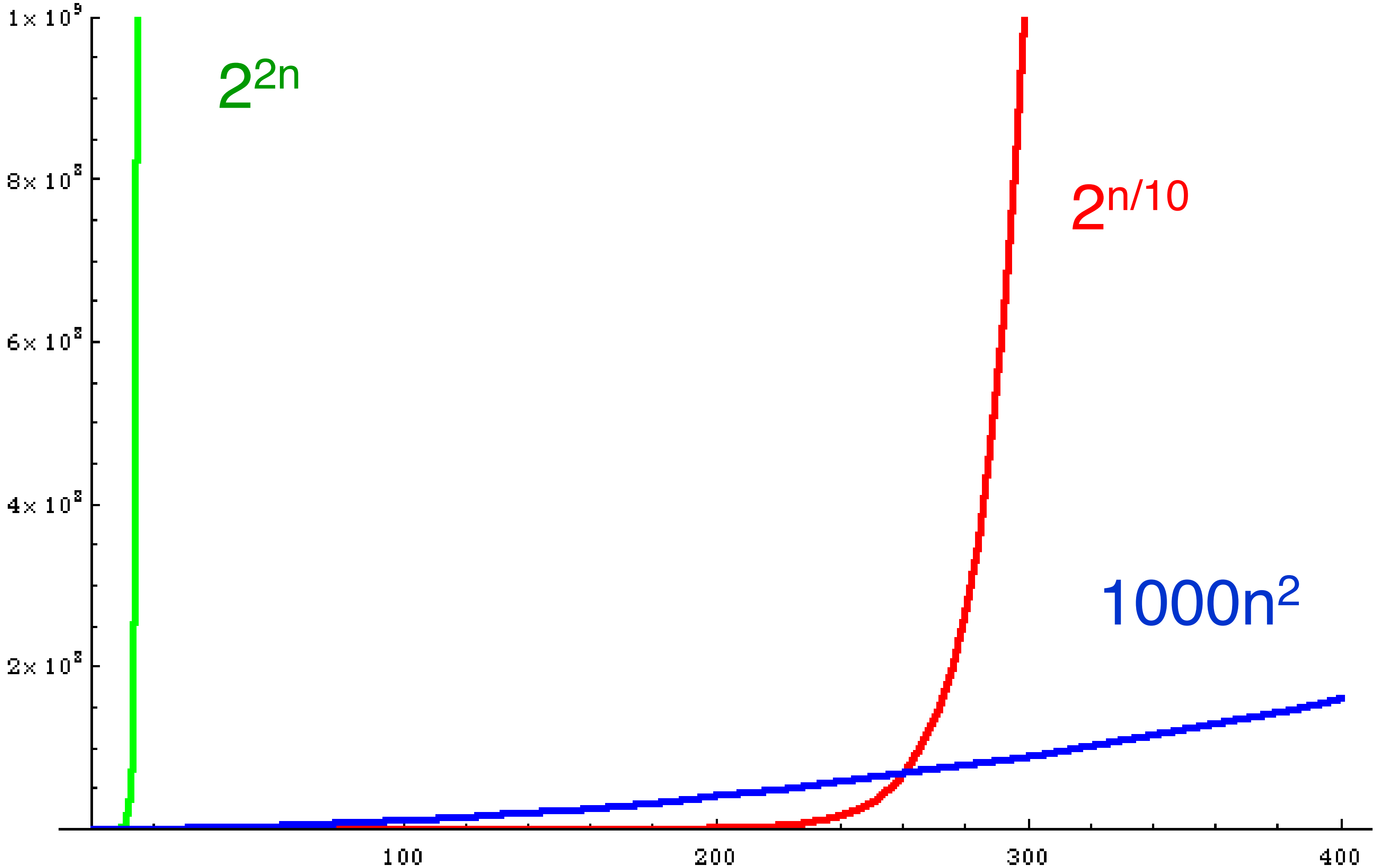


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

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

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

algorithm design techniques

We will survey two:

Later: Dynamic programming

Orderly solution of many smaller sub-problems, typically non-disjoint

Can give exponential speedups compared to more brute-force approaches

Today: Divide & Conquer

Reduce problem to one or more sub-problems of the same type, typically disjoint

Often gives significant, usually polynomial, speedup

Divide & Conquer

Reduce problem to one or more sub-problems of the same type

Each sub-problem's size a fraction of the original

Subproblem's typically disjoint

Often gives significant, usually polynomial, speedup

Examples:

- Mergesort, Binary Search, Strassen's Algorithm, Quicksort (roughly)

Suppose we've already invented DumbSort, taking time n^2

Try *Just One Level* of divide & conquer:

DumbSort(first n/2 elements)

DumbSort(last n/2 elements)

Merge results

Time: $\boxed{2\,(n/2)^2 + n = n^2/2 + n \ll n^2}$

D&C in a nutshell

Almost twice as fast!

Moral 1: "two halves are better than a whole"

Two problems of half size are *better* than one full-size problem, even given O(n) overhead of recombining, since the base algorithm has *super-linear* complexity.

Moral 2: "If a little's good, then more's better"

Two levels of D&C would be almost 4 times faster, 3 levels almost 8, etc., even though overhead is growing.

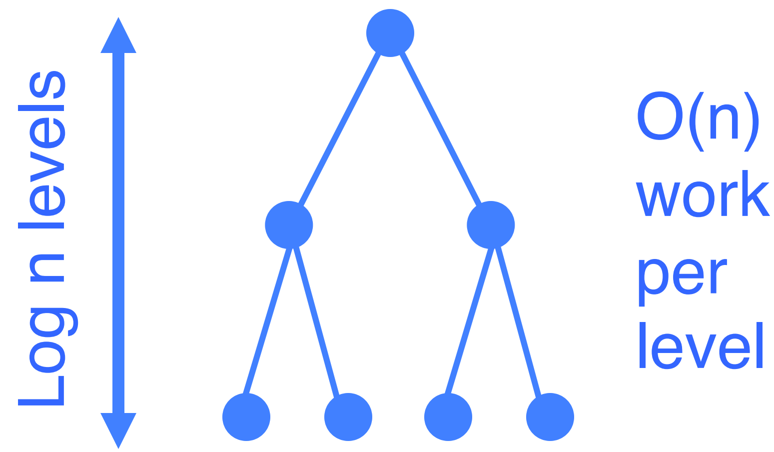
In the limit: you've just rediscovered mergesort.

Mergesort: (recursively) sort 2 half-lists, then merge results.

$T(n) = 2T(n/2)+cn, \;\; n \geq 2$

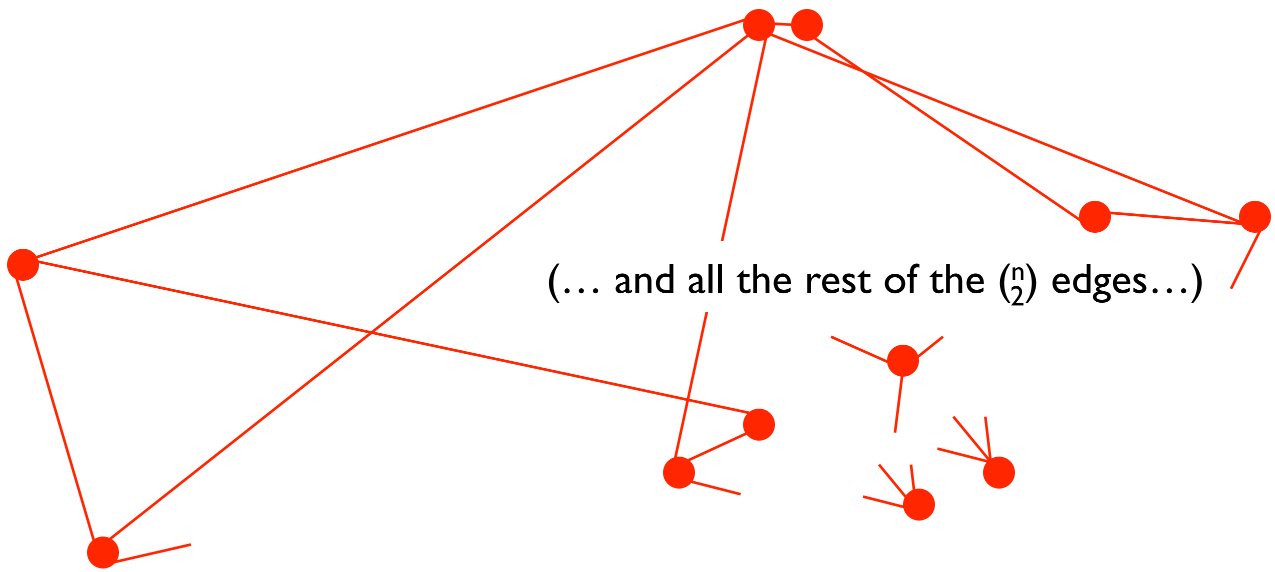
$T(1) = 0$

Solution: $O(n \log n)$



A Divide & Conquer Example: Closest Pair of Points

Given n points and arbitrary distances between them, find the closest pair. (E.g., think of distance as airfare – definitely not Euclidean distance!)

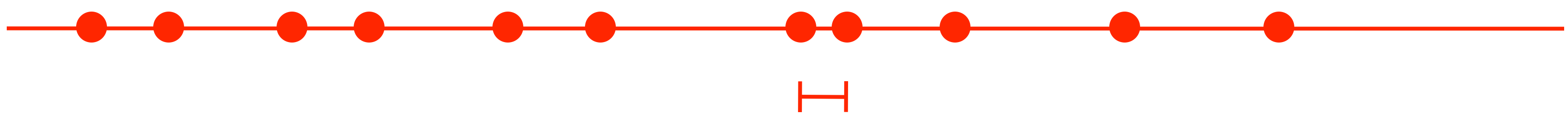


Must look at all n choose 2 pairwise distances, else any one you didn't check might be the shortest.

Also true for Euclidean distance in 1-2 dimensions?

closest pair of points: 1 dimensional version

Given n points on the real line, find the closest pair



Closest pair is *adjacent* in ordered list

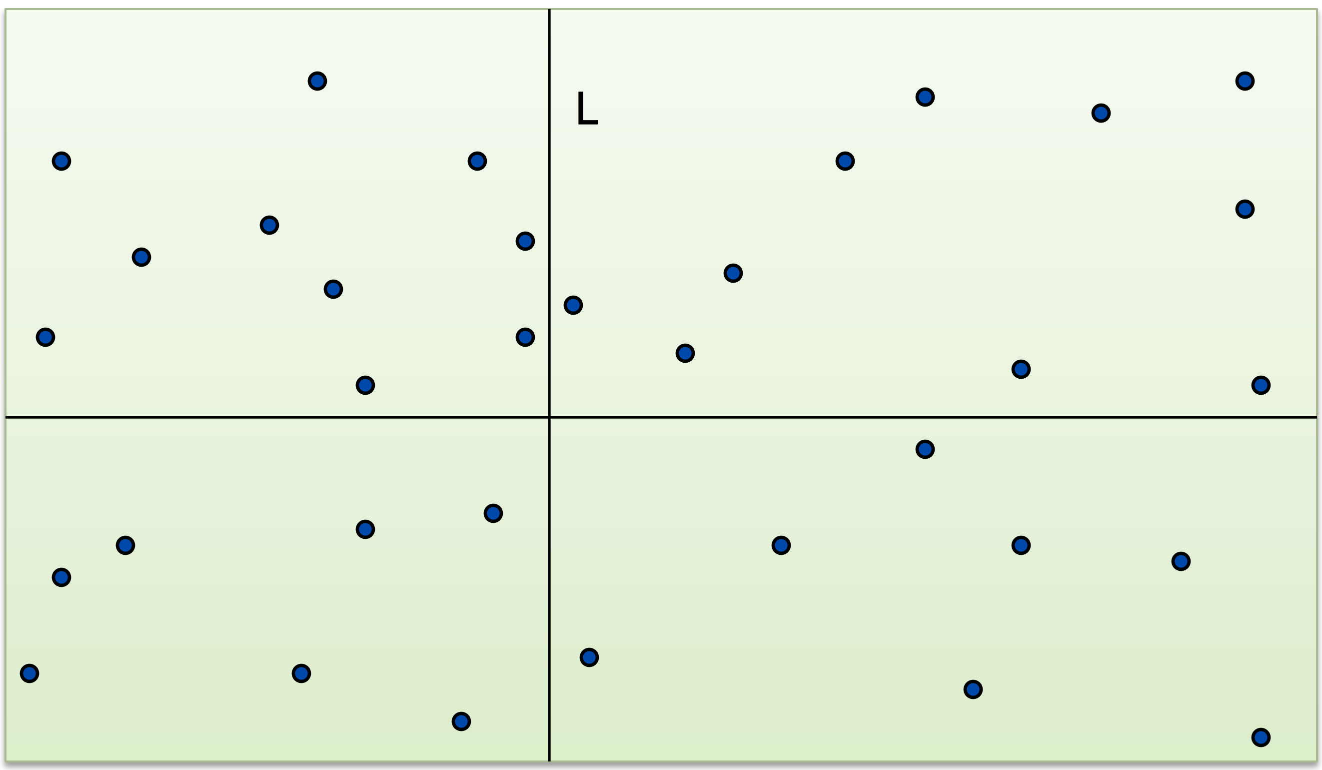
Time O(n log n) to sort, if needed

Plus O(n) to scan adjacent pairs

Key point: do *not* need to calc distances between all pairs: exploit geometry + ordering

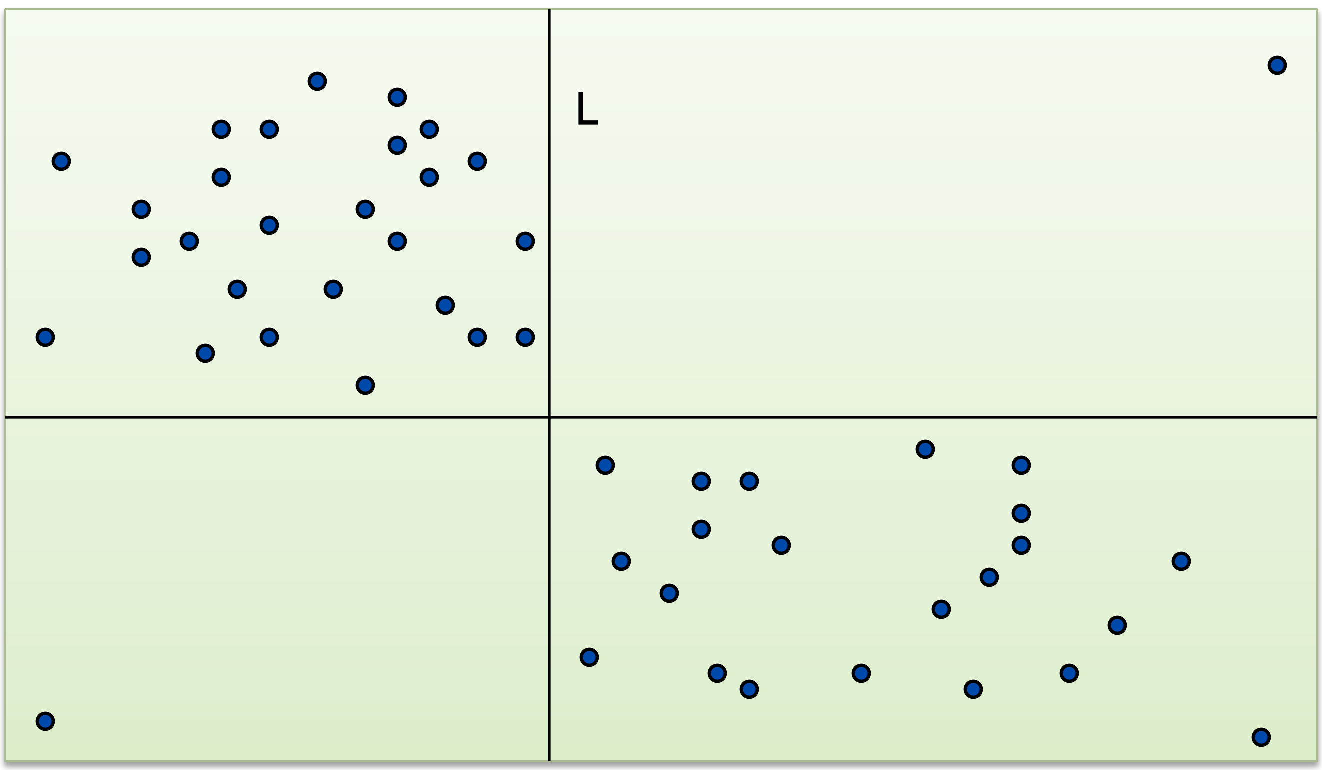
closest pair of points. 2d, Euclidean distance: 1st try

Divide. Sub-divide region into 4 quadrants.



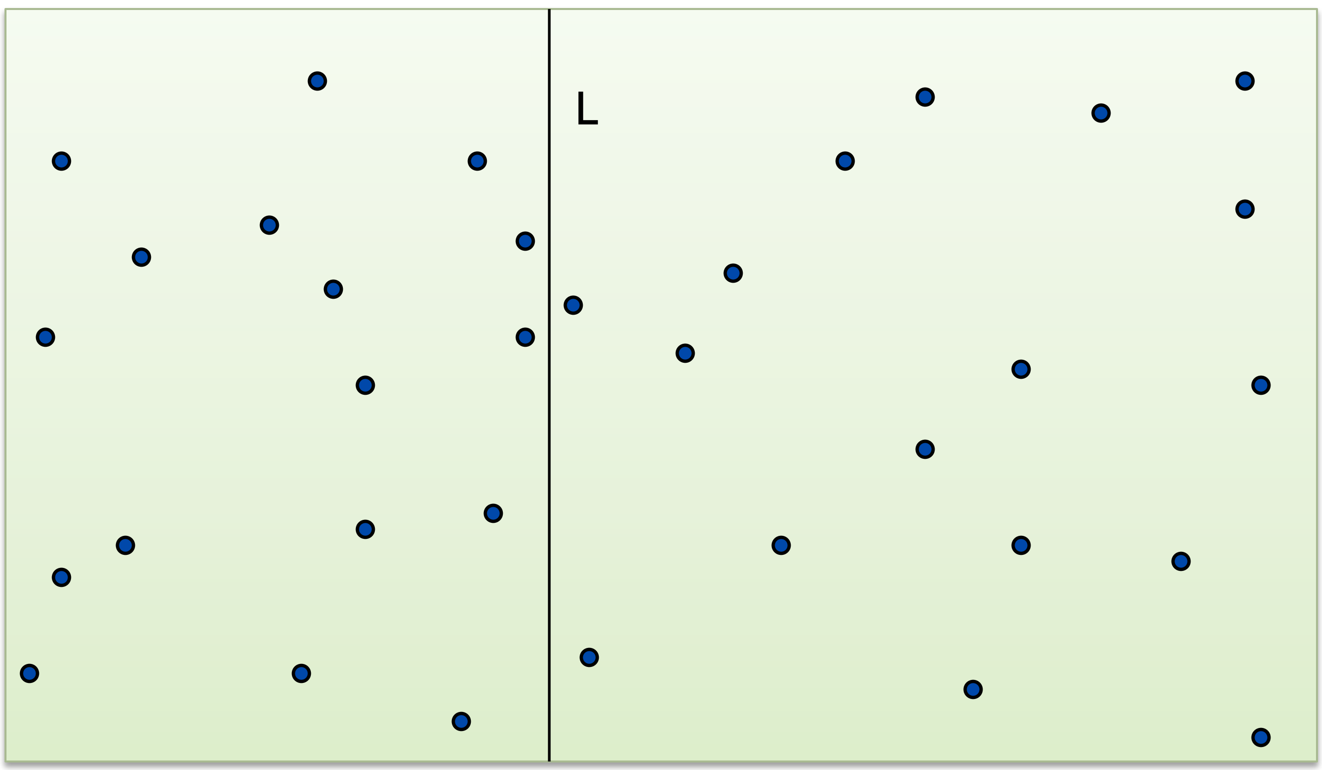
Divide. Sub-divide region into 4 quadrants.

Obstacle. Impossible to ensure n/4 points in each piece.



Algorithm.

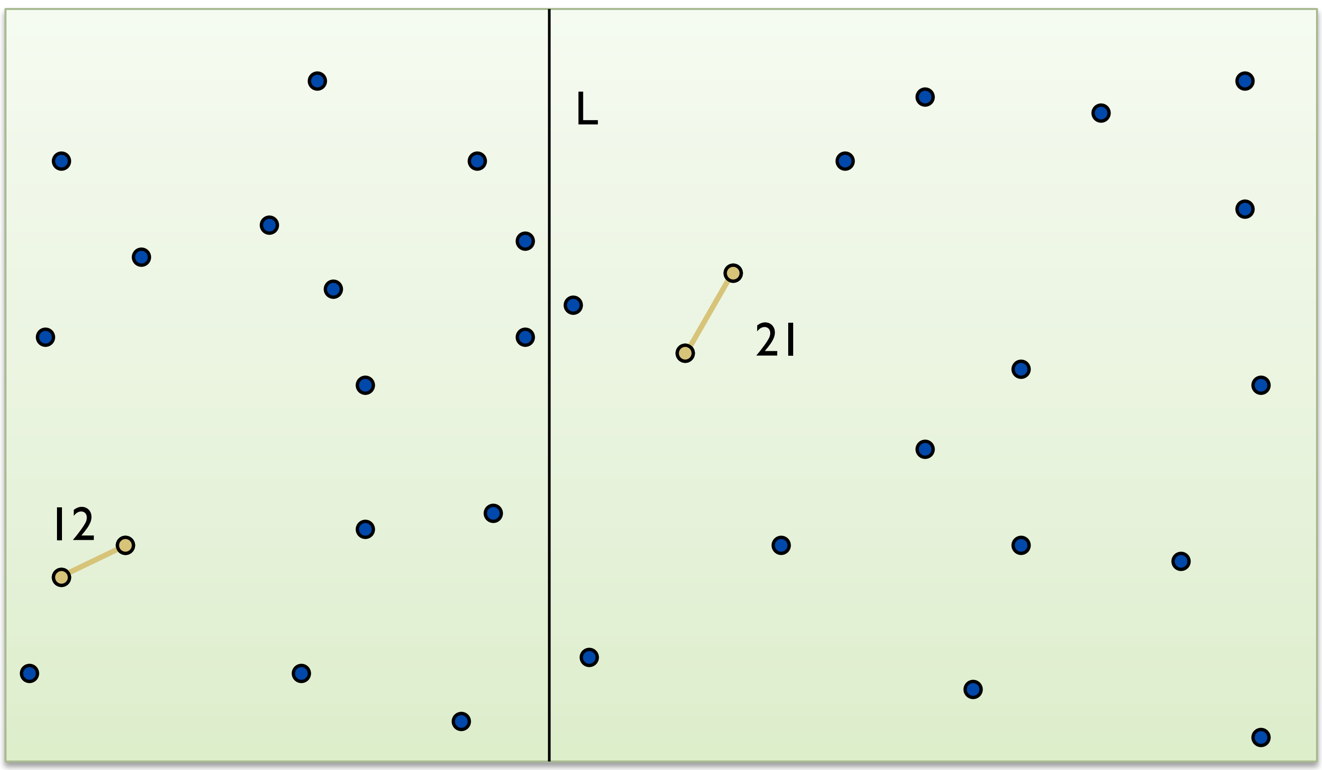
Divide: draw vertical line L with ≈ n/2 points on each side.



Algorithm.

Divide: draw vertical line L with ≈ n/2 points on each side.

Conquer: find closest pair on each side, recursively.



Algorithm.

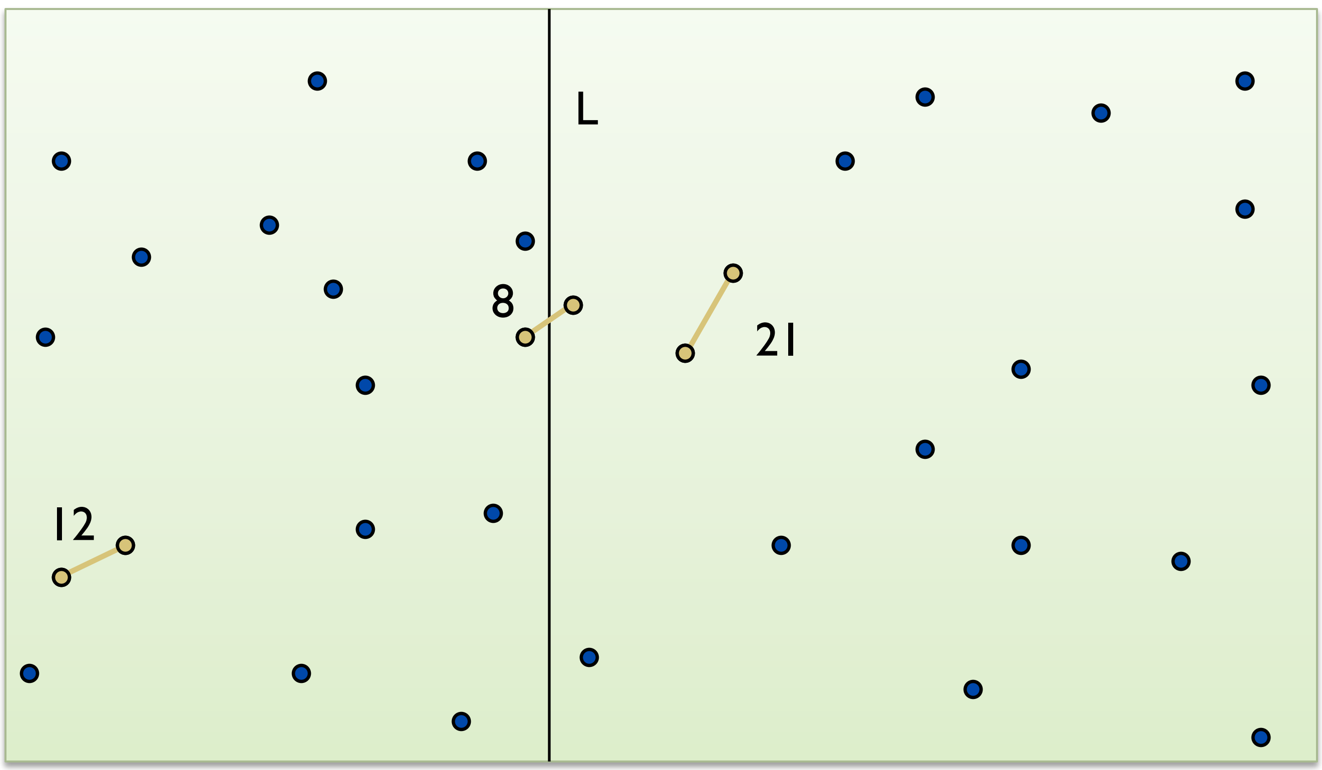
Divide: draw vertical line L with ≈ n/2 points on each side.

Conquer: find closest pair on each side, recursively.

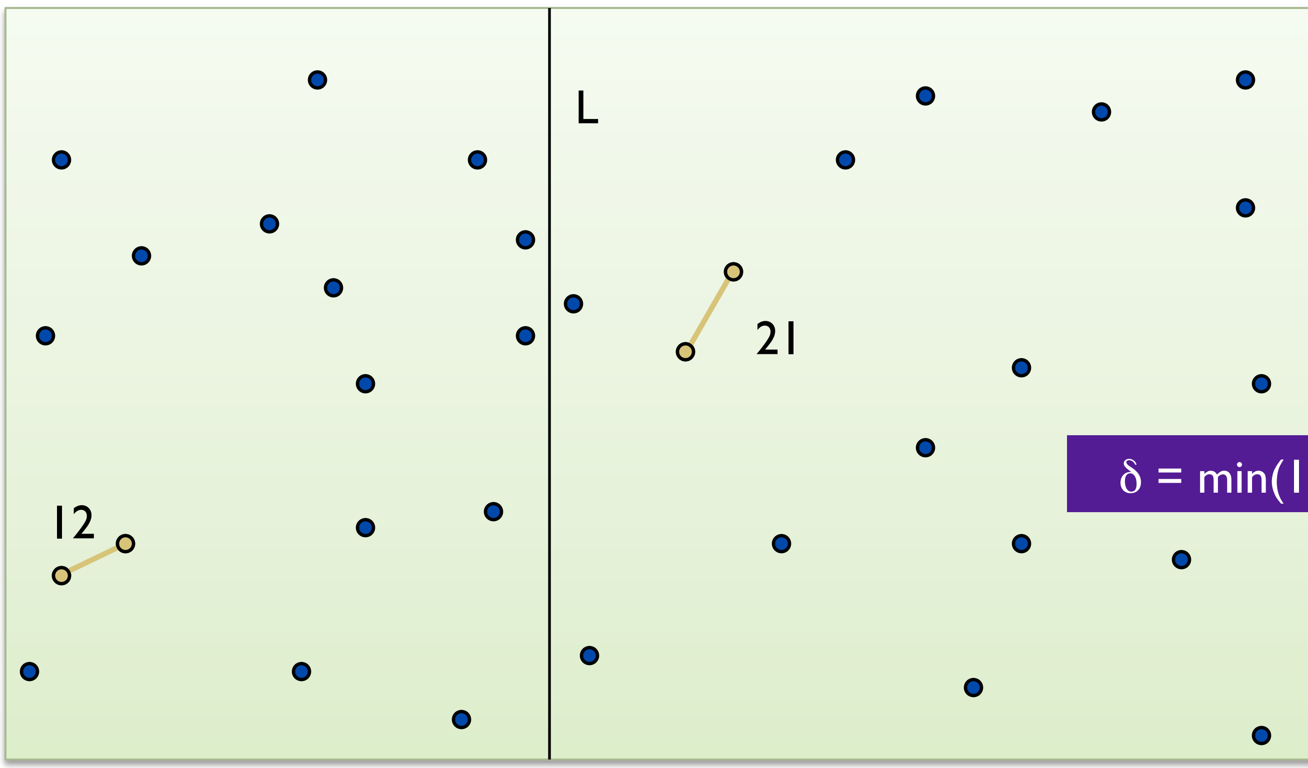
Combine: find closest pair with one point in each side.

Return best of 3 solutions.

seems like $\Theta(n^2)$?



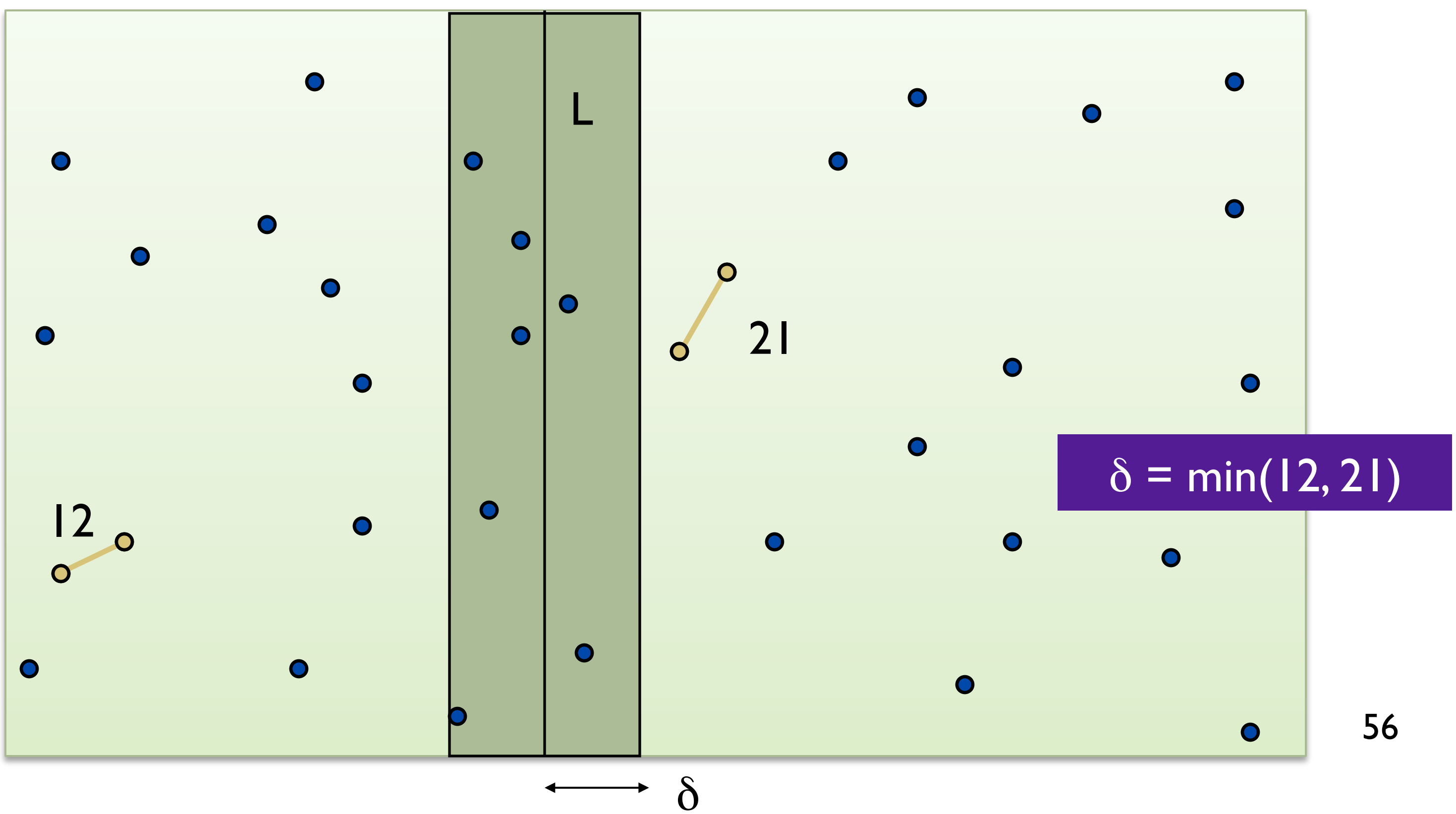
Find closest pair with one point in each side, assuming that distance < δ.



δ = min(12, 21)

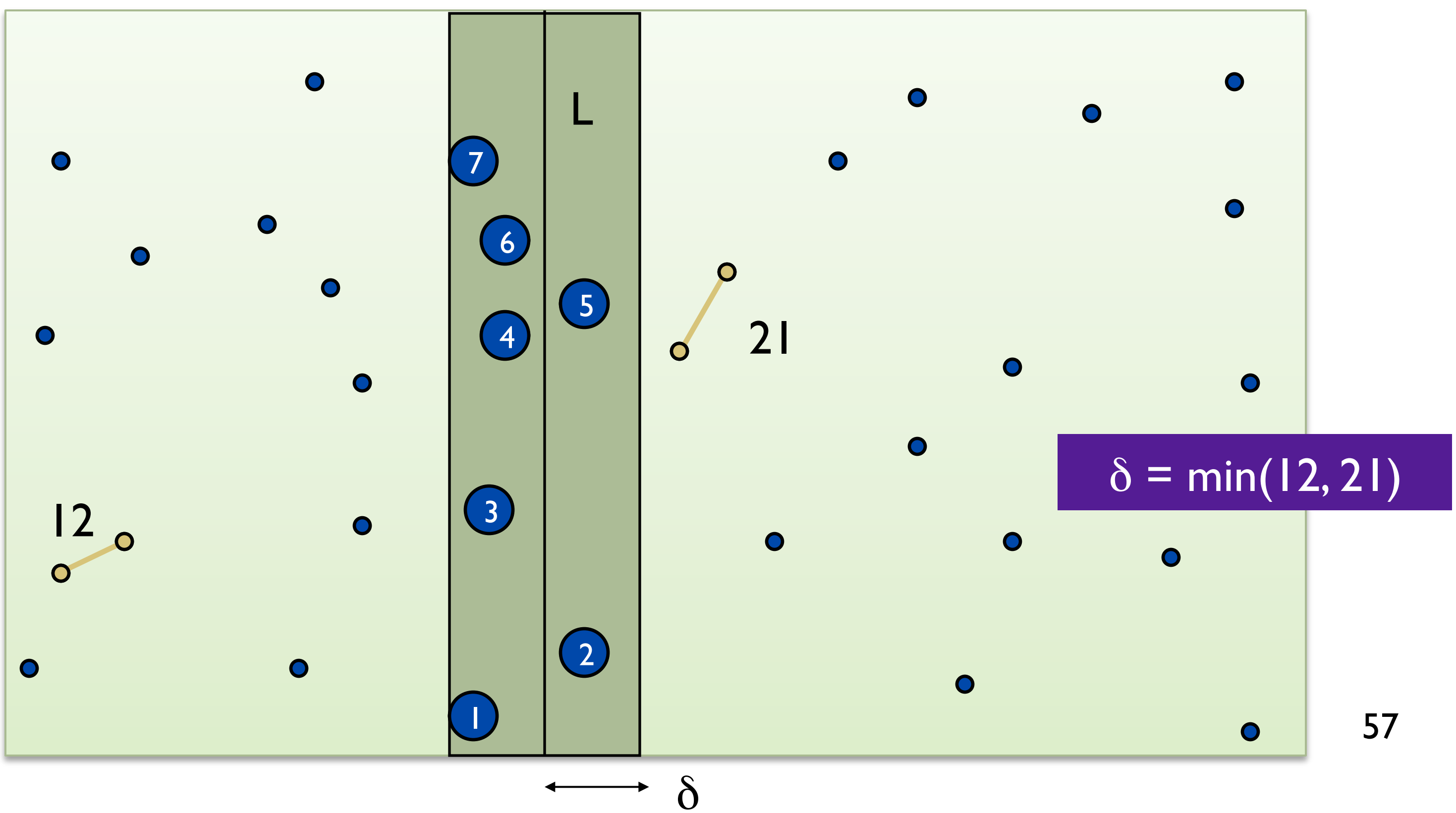
Find closest pair with one point in each side, assuming that distance < δ.

Observation: suffices to consider points within δ of line L.



Find closest pair with one point in each side, assuming that distance < δ.

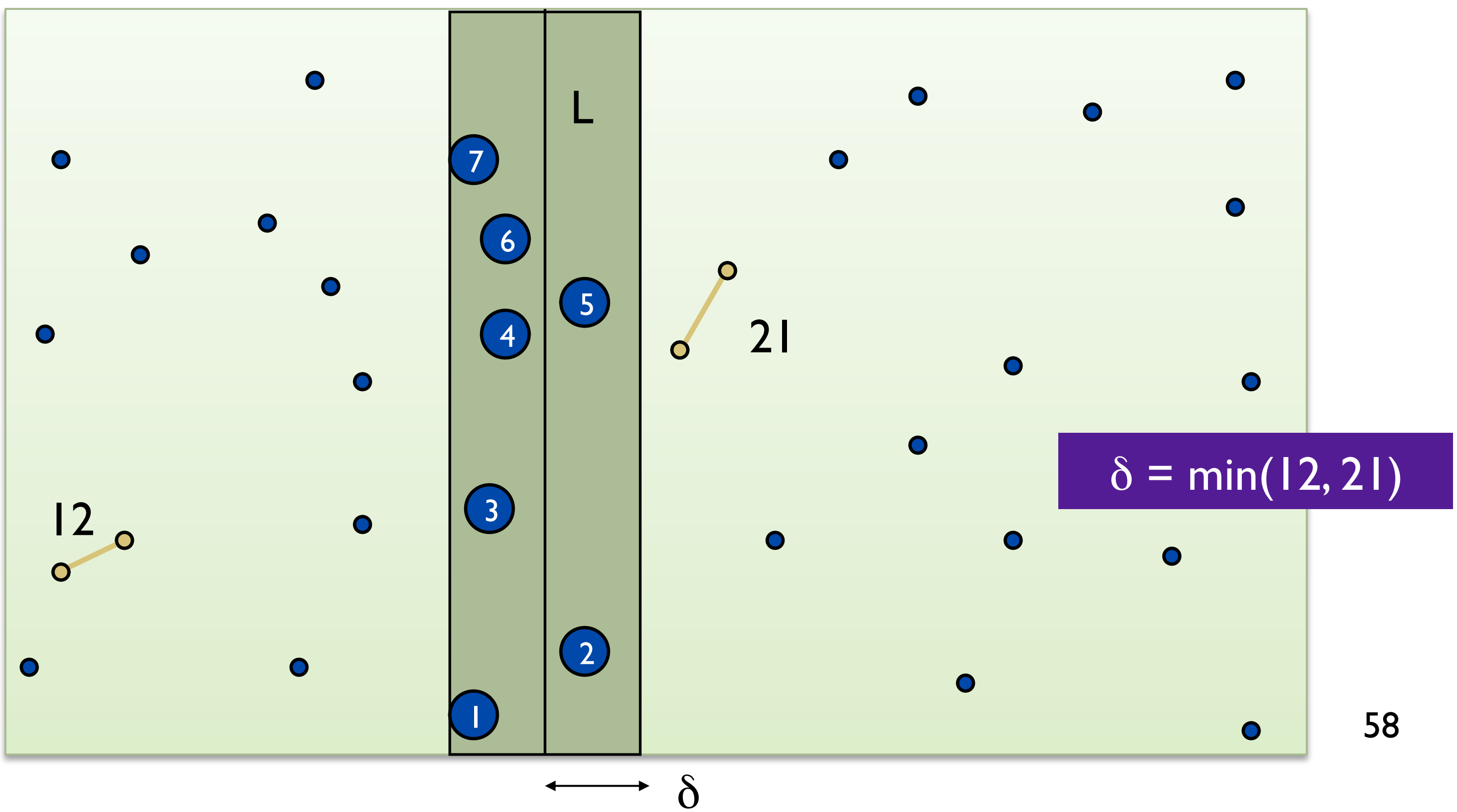
- Observation: suffices to consider points within δ of line L.
- Almost the one-D problem again: Sort points in 2δ-strip by their y coordinate.



Find closest pair with one point in each side, assuming that distance < δ.

Observation: suffices to consider points within δ of line L.

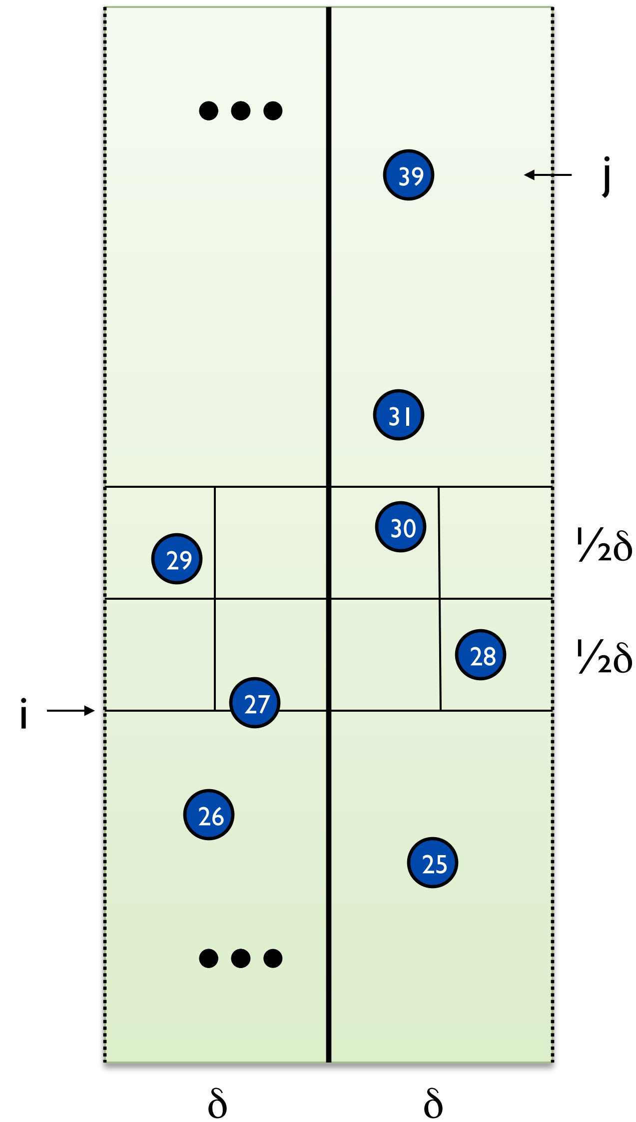
Almost the one-D problem again: Sort points in 2δ-strip by their y coordinate. Only check pts within 8 in sorted list!



Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

Claim. If $|i - j| > 8$, then the distance between s_i and s_j is $> \delta$.

Pf: No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box; only 8 boxes within δ



Number of pairwise distance calculations:

$$D(n) \leq \begin{cases} 0 & n = 1 \\ 2D(n/2) + 7n & n > 1 \end{cases} \Rightarrow D(n) = O(n \log n)$$

(A mostly superfluous detail: straightforward implementation gives a running time that is a factor of log n larger, due to sorting in the various subproblems. Run time can be reduced to O(n log n) also, roughly by the trick of sorting by x at the top level, and returning/merging y-sorted lists from the subcalls.

Regardless of this nuance, the big picture is the same: divide-and-conquer allows sharp speed gain over a naive n^2 method.)

Integer Multiplication

Add. Given two n-digit integers a and b, compute a + b.

O(n) bit operations.

	1	1	1	1	1	1	0	1	
		1	1	0	1	0	1	0	1
Add	+	0	1	1	1	1	1	0	1
	1	0	1	0	1	0	0	1	0

Add. Given two n-digit integers a and b, compute a + b.

O(n) bit operations.

Add

1	1	1	1	1	1	0	1	
	1	1	0	1	0	1	0	1
+	0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Multiply. Given two n-digit integers a and b, compute a × b.
The "grade school" method:

$\Theta(n^2)$ bit operations.

Multiply

								1	1	0	1	0	1	0	1
							*	0	1	1	1	1	1	0	1
								1	1	0	1	0	1	0	1
							0	0	0	0	0	0	0	0	
						1	1	0	1	0	1	0	1		
					1	1	0	1	0	1	0	1			
				1	1	0	1	0	1	0	1				
			1	1	0	1	0	1	0	1					
		1	1	0	1	0	1	0	1						
	0	0	0	0	0	0	0	0							
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1

divide-and-conquer multiplication: warmup

To multiply two 2-digit integers:

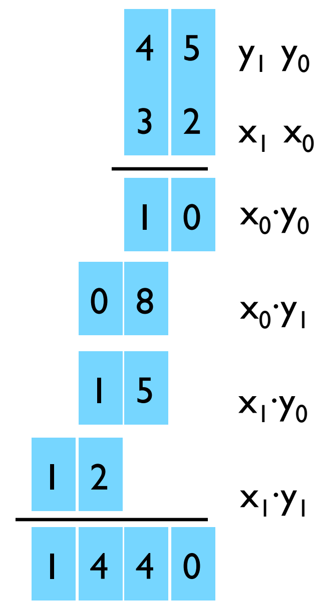
Multiply four 1-digit integers.

Add, shift some 2-digit integers to obtain result.

$$
\begin{aligned}
x &= 10\cdot x_1 + x_0 \\
y &= 10\cdot y_1 + y_0 \\
xy &= \left(10\cdot x_1 + x_0\right)\left(10\cdot y_1 + y_0\right) \\
&= 100\cdot x_1y_1 + 10\cdot\left(x_1y_0 + x_0y_1\right) + x_0y_0
\end{aligned}
$$

Same idea works for *long* integers –

can split them into 4 half-sized ints



To multiply two n-digit integers:

Multiply four n/2-digit integers.

Add, shift some n/2-digit integers to obtain result.

$$
\begin{aligned}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) \\
&= 2^{n} \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0
\end{aligned}
$$

$$
\mathrm{T}(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow \mathrm{T}(n) = \Theta(n^2)
$$

assumes n is a power of 2

		1 1 0 1	0 1 0 1	y_1 y_0
		* 0 1 1 1	1 1 0 1	x_1 x_0
		0 1 0 0	0 0 0 1	$x_0 \cdot y_0$
	1 0 1 0	1 0 0 1		$x_0 \cdot y_1$
	0 0 1 0	0 0 1 1		$x_1 \cdot y_0$
0 1 0 1	1 0 1 1			$x_1 \cdot y_1$
0 1 1 0	1 0 0 0	0 0 0 0	0 0 0 1	

key trick: 2 multiplies for the price of 1

$$\begin{aligned}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) \\
&= 2^{n} \cdot x_1 y_1 + 2^{n/2} \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0
\end{aligned}$$

Well, ok, 4 for 3 is more accurate…

$$\begin{aligned}
\alpha &= x_1 + x_0 \\
\beta &= y_1 + y_0 \\
\alpha\beta &= \left(x_1 + x_0\right)\left(y_1 + y_0\right) \\
&= x_1 y_1 + \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0 \\
\left(x_1 y_0 + x_0 y_1\right) &= \alpha\beta - x_1 y_1 - x_0 y_0
\end{aligned}$$

To multiply two n-digit integers:

Add two ½n digit integers.

Multiply three ½n-digit integers.

Add, subtract, and shift ½n-digit integers to obtain result.

$$
\begin{aligned}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= 2^{n} \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
&= 2^{n} \cdot \underbrace{x_1 y_1}_{A} + 2^{n/2} \cdot \big(\underbrace{(x_1 + x_0)(y_1 + y_0)}_{B} - \underbrace{x_1 y_1}_{A} - \underbrace{x_0 y_0}_{C} \big) + \underbrace{x_0 y_0}_{C}
\end{aligned}
$$

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit ops.

$$
T(n) \le \underbrace{3\,T(n/2)}_{\text{recursive calls}} + \underbrace{O(n)}_{\text{add, subtract, shift}}
$$

$$
\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})
$$

Naïve: $\Theta(n^2)$

Karatsuba: $\Theta(n^{1.59\ldots})$

Amusing exercise: generalize Karatsuba to do 5 size n/3 subproblems $\rightarrow \Theta(n^{1.46\ldots})$

Best known: $\Theta(n \log n \operatorname{loglog} n)$

- "Fast Fourier Transform"
- but mostly unused in practice (unless you need really big numbers - a billion digits of π, say)

High precision arithmetic IS important for crypto

Idea:

- "Two halves are better than a whole"
  - if the base algorithm has super-linear complexity.
- "If a little's good, then more's better"
  - repeat above, recursively

Applications: Many.

Binary Search, Merge Sort, (Quicksort), Closest points, Integer multiply,…