CSEP 521
Algorithms

Divide and Conquer

Richard Anderson

With Special Cameo Appearance by

Larry Ruzzo

Divide and Conquer Algorithms

Split into sub problems
Recursively solve the problem
Combine solutions

Make progress in the split and combine stages
- Quicksort – progress made at the split step
- Mergesort – progress made at the combine step

D&C Algorithms
- Strassen's Algorithm – Matrix Multiplication
- Inversions
- Median
- Closest Pair
- Integer Multiplication
- FFT
- …

divide & conquer – the key idea

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!

d&c approach, cont.

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.
Best is usually full recursion down to some small constant size (balancing "work" vs "overhead").

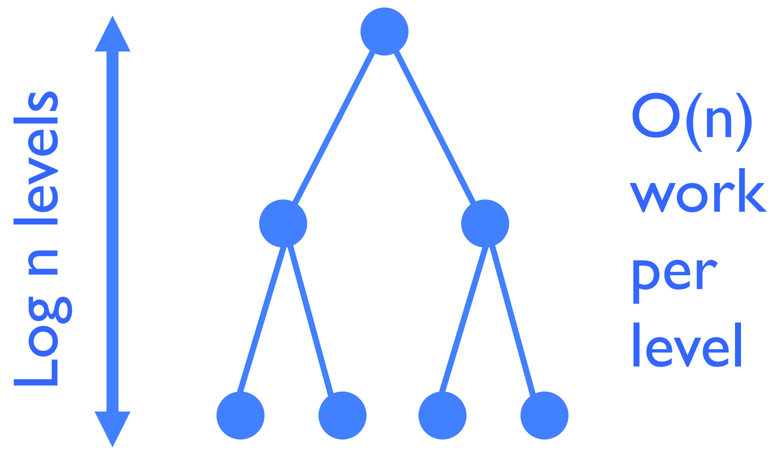
In the limit: you’ve just rediscovered mergesort!

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

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

$T(1) = 0$

Solution: $\Theta(n \log n)$
(details later)



What you really need to know about recurrences

Work per level changes geometrically with the level

Geometrically increasing $(x > 1)$

- The bottom level wins – count leaves

Geometrically decreasing $(x < 1)$

- The top level wins – count top level work

Balanced $(x = 1)$

- Equal contribution – top • levels (e.g. “n logn”)

$T(n) = aT(n/b) + n^c$

Balanced: $a = b^c$

Increasing: $a > b^c$

Decreasing: $a < b^c$

Recurrences

Next: how to solve them

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

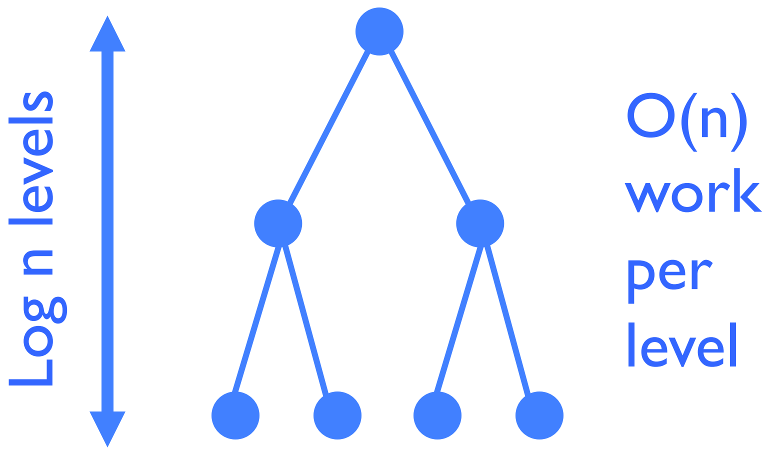
T(n) = 2T(n/2)+cn, n≥2

T(1) = 0

Solution: Θ(n log n)

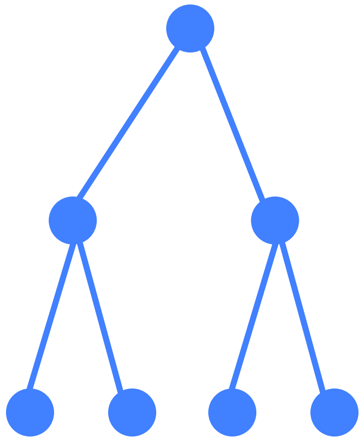
(~~details~~ later)

now



Solve: $T(1) = c$

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



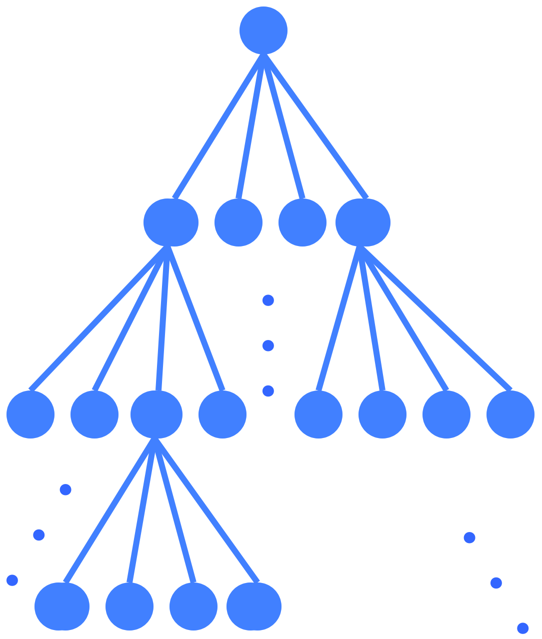
Level	Num	Size	Work
0	$1 = 2^0$	n	cn
1	$2 = 2^1$	$n/2$	$2cn/2$
2	$4 = 2^2$	$n/4$	$4cn/4$
…	…	…	…
i	2^i	$n/2^i$	$2^i\ c\ n/2^i$
…	…	…	…
$k-1$	2^{k-1}	$n/2^{k-1}$	$2^{k-1}\ c\ n/2^{k-1}$
k	2^k	$n/2^k = 1$	$2^k\,T(1)$

$n = 2^k$; $k = \log_2 n$

Total Work: $c\ n\ (1+\log_2 n)$ (add last col)

Solve: $T(1) = c$

$T(n) = 4\,T(n/2) + cn$



Level	Num	Size	Work
0	$1 = 4^0$	n	cn
1	$4 = 4^1$	$n/2$	$4cn/2$
2	$16 = 4^2$	$n/4$	$16cn/4$
…	…	…	…
i	4^i	$n/2^i$	$4^i\, c\, n/2^i$
…	…	…	…
$k-1$	4^{k-1}	$n/2^{k-1}$	$4^{k-1}\, c\, n/2^{k-1}$
k	4^k	$n/2^k = 1$	$4^k\, T(1)$

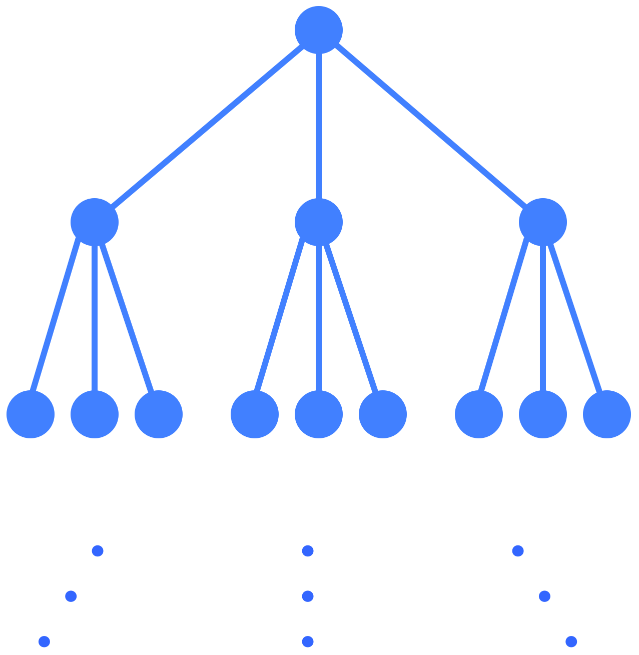
$n = 2^k$; $k = \log_2 n$

Total Work: $T(n) = \sum_{i=0}^{k} 4^i cn/2^i = O(n^2)$

$4^k = (2^2)^k = (2^k)^2 = n^2$

Solve: $T(1) = c$

$T(n) = 3\,T(n/2) + cn$



Level	Num	Size	Work
0	$1 = 3^0$	n	cn
1	$3 = 3^1$	$n/2$	$3cn/2$
2	$9 = 3^2$	$n/4$	$9cn/4$
…	…	…	…
i	3^i	$n/2^i$	$3^i\ c\ n/2^i$
…	…	…	…
$k-1$	3^{k-1}	$n/2^{k-1}$	$3^{k-1}\ c\ n/2^{k-1}$
k	3^k	$n/2^k = 1$	$3^k\,T(1)$

$n = 2^k$; $k = \log_2 n$

Total Work: $T(n) = \sum_{i=0}^{k} 3^i cn / 2^i$

a useful identity

Theorem:

$$1 + x + x^2 + x^3 + \ldots + x^k = (x^{k+1}-1)/(x-1)$$

proof:

$$\begin{aligned}
y &= 1 + x + x^2 + x^3 + \ldots + x^k \\
xy &= \phantom{1 + {}} x + x^2 + x^3 + \ldots + x^k + x^{k+1} \\
xy-y &= x^{k+1} - 1 \\
y(x-1) &= x^{k+1} - 1 \\
y &= (x^{k+1}-1)/(x-1)
\end{aligned}$$

Solve: $T(1) = c$
$T(n) = 3\,T(n/2) + cn$ (cont.)

$$T(n) = \sum_{i=0}^{k} 3^i cn / 2^i$$

$$= cn \sum_{i=0}^{k} 3^i / 2^i$$

$$= cn \sum_{i=0}^{k} \left(\tfrac{3}{2}\right)^i$$

$$= cn \frac{\left(\tfrac{3}{2}\right)^{k+1} - 1}{\left(\tfrac{3}{2}\right) - 1}$$

$$\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1} \quad (x \neq 1)$$

Solve: $T(1) = c$; $T(n) = 3\,T(n/2) + cn$ (cont.)

$$cn\frac{\left(\frac{3}{2}\right)^{k+1}-1}{\left(\frac{3}{2}\right)-1} = 2cn\left(\left(\frac{3}{2}\right)^{k+1}-1\right)$$

$$< 2cn\left(\frac{3}{2}\right)^{k+1}$$

$$= 3cn\left(\frac{3}{2}\right)^{k}$$

$$= 3cn\frac{3^k}{2^k}$$

Solve: T(1) = c
T(n) = 3 T(n/2) + cn (cont.)

$$3cn\frac{3^k}{2^k} = 3cn\frac{3^{\log_2 n}}{2^{\log_2 n}}$$

$$= 3cn\frac{3^{\log_2 n}}{n}$$

$$= 3c3^{\log_2 n}$$

$$= 3c\left(n^{\log_2 3}\right)$$

$$\boxed{= O\left(n^{1.585\ldots}\right)}$$

$$a^{\log_b n}$$

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

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

$$= n^{\log_b a}$$

divide and conquer – master recurrence

$T(n) = aT(n/b)+cn^k$ for $n > b$ then

$a > b^k \Rightarrow T(n) = \Theta(n^{\log_b a})$ [many subprobs → leaves dominate]

$a < b^k \Rightarrow T(n) = \Theta(n^k)$ [few subprobs → top level dominates]

$a = b^k \Rightarrow T(n) = \Theta(n^k \log n)$ [balanced → all log n levels contribute]

Fine print:

$a \geq 1$; $b > 1$; $c, d, k \geq 0$; $T(1) = d$; $n = b^t$ for some $t > 0$; a, b, k, t integers. True even if it is $\lceil n/b \rceil$ instead of n/b.

master recurrence: proof sketch

Expanding recurrence as in earlier examples, to get

$T(n) = n^h (d + c S)$

where $h = \log_b(a)$ (tree height) and $S = \sum_{j=1}^{\log_b n} x^j$, where $x = b^k/a$.

If $c = 0$ the sum S is irrelevant, and $T(n) = O(n^h)$: all the work happens in the base cases, of which there are n^h, one for each leaf in the recursion tree.

If $c > 0$, then the sum matters, and splits into 3 cases (like previous slide):

- if $x < 1$, then $S < x/(1-x) = O(1)$. [S is just the first log n terms of the infinite series with that sum].
- if $x = 1$, then $S = \log_b(n) = O(\log n)$. [all terms in the sum are 1 and there are that many terms].
- if $x > 1$, then $S = x \cdot (x^{1+\log_b(n)}-1)/(x-1)$. After some algebra, $n^h * S = O(n^k)$

Example:

Matrix Multiplication –

Strassen’s Method

Multiplying Matrices

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \bullet \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$= \begin{bmatrix} \boxed{a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42} & \circ & a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34} + a_{14}b_{44} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42} & \circ & a_{21}b_{14} + a_{22}b_{24} + a_{23}b_{34} + a_{24}b_{44} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} + a_{34}b_{41} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42} & \circ & a_{31}b_{14} + a_{32}b_{24} + a_{33}b_{34} + a_{34}b_{44} \\ a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31} + a_{44}b_{41} & a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} + a_{44}b_{42} & \circ & a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44} \end{bmatrix}$$

n^3 multiplications, n^3-n^2 additions

Simple Matrix Multiply

```
for i = 1 to n
    for j = 1 to n
        C[i,j] = 0
        for k = 1 to n
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

n^3 multiplications, n^3-n^2 additions

Multiplying Matrices

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \bullet \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42} & \circ & a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34} + a_{14}b_{44} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42} & \circ & a_{21}b_{14} + a_{22}b_{24} + a_{23}b_{34} + a_{24}b_{44} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} + a_{34}b_{41} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42} & \circ & a_{31}b_{14} + a_{32}b_{24} + a_{33}b_{34} + a_{34}b_{44} \\ a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31} + a_{44}b_{41} & a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} + a_{44}b_{42} & \circ & a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44} \end{bmatrix}$$

Multiplying Matrices

$$\begin{bmatrix} a_{11} & a_{12} & \boxed{a_{13} \quad a_{14}} \\ a_{21} & a_{22} & \boxed{a_{23} \quad a_{24}} \\ a_{31} & a_{32} & a_{33} \quad a_{34} \\ a_{41} & a_{42} & a_{43} \quad a_{44} \end{bmatrix} \bullet \begin{bmatrix} b_{11} \quad b_{12} & b_{13} & b_{14} \\ b_{21} \quad b_{22} & b_{23} & b_{24} \\ \boxed{b_{31} \quad b_{32}} & b_{33} & b_{34} \\ \boxed{b_{41} \quad b_{42}} & b_{43} & b_{44} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + \boxed{a_{13}b_{31} + a_{14}b_{41}} & a_{11}b_{12} + a_{12}b_{22} + \boxed{a_{13}b_{32} + a_{14}b_{42}} & \circ & a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34} + a_{14}b_{44} \\ a_{21}b_{11} + a_{22}b_{21} + \boxed{a_{23}b_{31} + a_{24}b_{41}} & a_{21}b_{12} + a_{22}b_{22} + \boxed{a_{23}b_{32} + a_{24}b_{42}} & \circ & a_{21}b_{14} + a_{22}b_{24} + a_{23}b_{34} + a_{24}b_{44} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} + a_{34}b_{41} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42} & \circ & a_{31}b_{14} + a_{32}b_{24} + a_{33}b_{34} + a_{34}b_{44} \\ a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31} + a_{44}b_{41} & a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} + a_{44}b_{42} & \circ & a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44} \end{bmatrix}$$

Multiplying Matrices

$$
\left[\begin{array}{cc|cc}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
\hline
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{array}\right]
\bullet
\left[\begin{array}{cc|cc}
b_{11} & b_{12} & b_{13} & b_{14} \\
b_{21} & b_{22} & b_{23} & b_{24} \\
\hline
b_{31} & b_{32} & b_{33} & b_{34} \\
b_{41} & b_{42} & b_{43} & b_{44}
\end{array}\right]
$$

$$
\left[\begin{array}{c|c}
A_{11} & A_{12} \\
\hline
A_{21} & A_{22}
\end{array}\right]
\left[\begin{array}{c|c}
B_{11} & B_{12} \\
\hline
B_{21} & B_{22}
\end{array}\right]
$$

$$
= \left[\begin{array}{cc|cc}
a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42} & \circ & a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34} + a_{14}b_{44} \\
a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42} & \circ & a_{21}b_{14} + a_{22}b_{24} + a_{23}b_{34} + a_{24}b_{44} \\
\hline
a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} + a_{34}b_{41} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42} & \circ & a_{31}b_{14} + a_{32}b_{24} + a_{33}b_{34} + a_{34}b_{44} \\
a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31} + a_{44}b_{41} & a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} + a_{44}b_{42} & \circ & a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44}
\end{array}\right]
$$

$$
\left[\begin{array}{c|c}
A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\
\hline
A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22}
\end{array}\right]
$$

Multiplying Matrices

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)\left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

$$= \left(\begin{array}{c|c} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ \hline A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \end{array}\right)$$

Counting arithmetic operations:

$T(n) = 8T(n/2) + 4(n/2)^2 = 8T(n/2) + n^2$

Multiplying Matrices

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8T(n/2) + n^2 & \text{if } n > 1 \end{cases}$$

By Master Recurrence, if

$T(n) = aT(n/b)+cn^k$ & $a > b^k$ then

$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

Strassen’s algorithm

Multiply 2x2 matrices using **7** instead of **8** multiplications (and lots more than 4 additions)

$T(n)=7\,T(n/2)+cn^2$

$7>2^2$ so $T(n)$ is $\Theta(n^{\log_2 7})$ which is $O(n^{2.81})$

Asymptotically fastest know algorithm uses $O(n^{2.376})$ time

not practical but Strassen’s may be practical provided calculations are exact and we stop recursion when matrix has size about 100 (maybe 10)

The algorithm

$P_1 = A_{12}(B_{11} + B_{21})$

$P_2 = A_{21}(B_{12} + B_{22})$

$P_3 = (A_{11} - A_{12})B_{11}$

$P_4 = (A_{22} - A_{21})B_{22}$

$P_5 = (A_{22} - A_{12})(B_{21} - B_{22})$

$P_6 = (A_{11} - A_{21})(B_{12} - B_{11})$

$P_7 = (A_{21} - A_{12})(B_{11} + B_{22})$

$C_{11} = P_1 + P_3$

$C_{12} = P_2 + P_3 + P_6 - P_7$

$C_{21} = P_1 + P_4 + P_5 + P_7$

$C_{22} = P_2 + P_4$

Example:
Counting Inversions

Inversion Problem

Let $a_1, \ldots a_n$ be a permutation of $1 \ldots n$

(a_i, a_j) is an inversion if $i < j$ and $a_i > a_j$

4, 6, 1, 7, 3, 2, 5

Problem: given a permutation, count the number of inversions

This can be done easily in $O(n^2)$ time

- Can we do better?

Application

Counting inversions can be use to measure closeness of ranked preferences

- People rank 20 movies, based on their rankings you cluster people who like the same types of movies

Can also be used to measure nonlinear correlation

Inversion Problem

Let $a_1, \ldots a_n$ be a permutation of $1 \ldots n$

(a_i, a_j) is an inversion if $i < j$ and $a_i > a_j$

4, 6, 1, 7, 3, 2, 5

Problem: given a permutation, count the number of inversions

This can be done easily in $O(n^2)$ time

Can we do better?

Counting Inversions

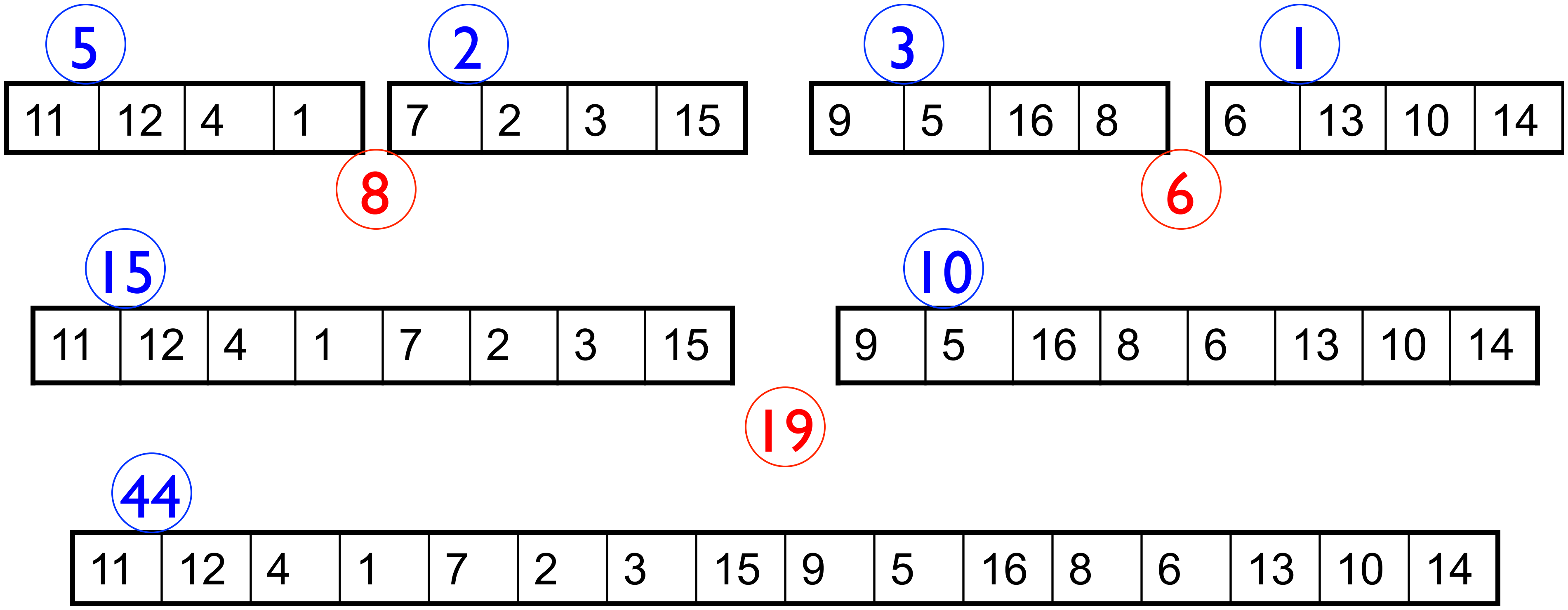
11	12	4	1	7	2	3	15	9	5	16	8	6	13	10	14

Count inversions on lower half

Count inversions on upper half

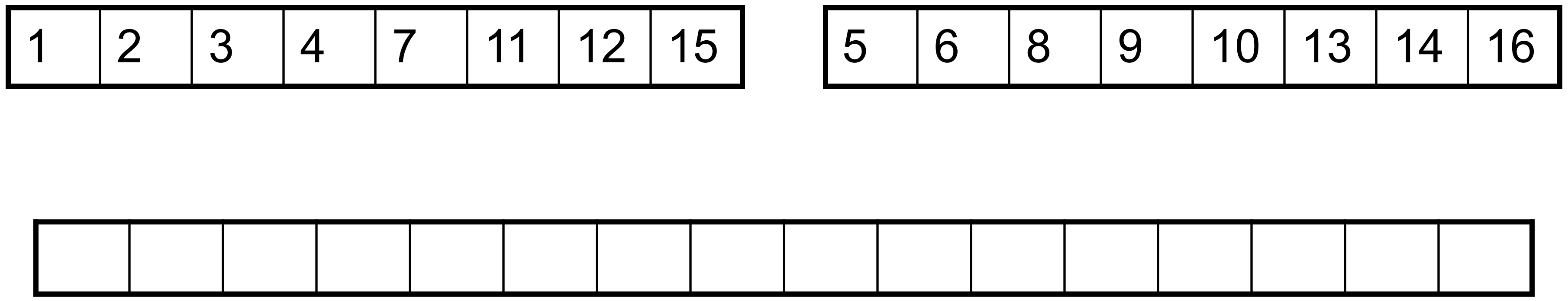
Count the inversions between the halves

Count the Inversions



Problem – how do we count inversions between sub problems in O(n) time?

Solution – Count inversions while merging



Standard merge algorithm – add to inversion count when an element is moved from the upper array to the solution

Counting inversions while merging

1	4	11	12

2	3	7	15

5	8	9	16

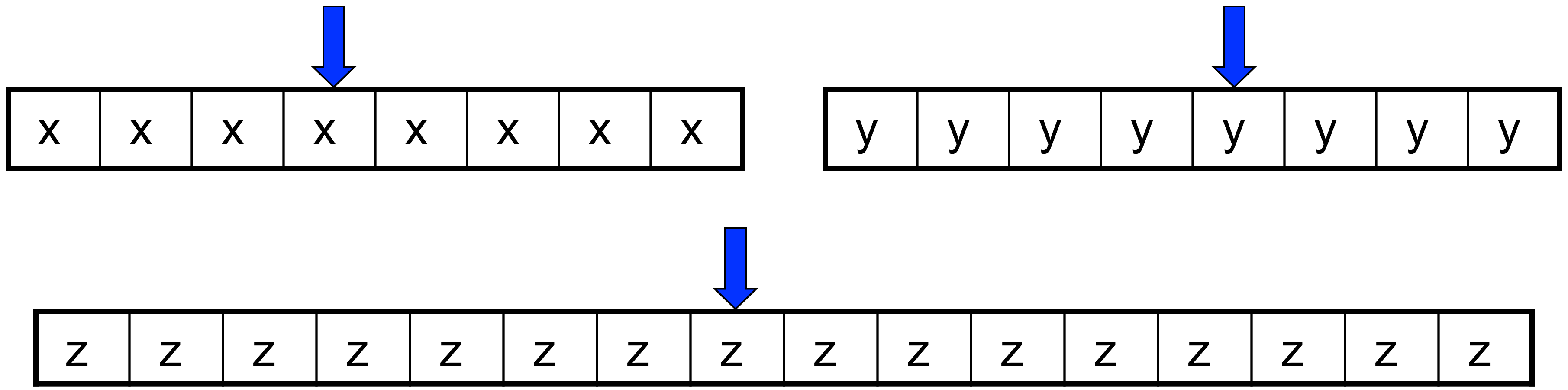
6	10	13	14

Indicate the number of inversions for each element detected when merging

Inversions

Counting inversions between two sorted lists

- O(1) per element to count inversions



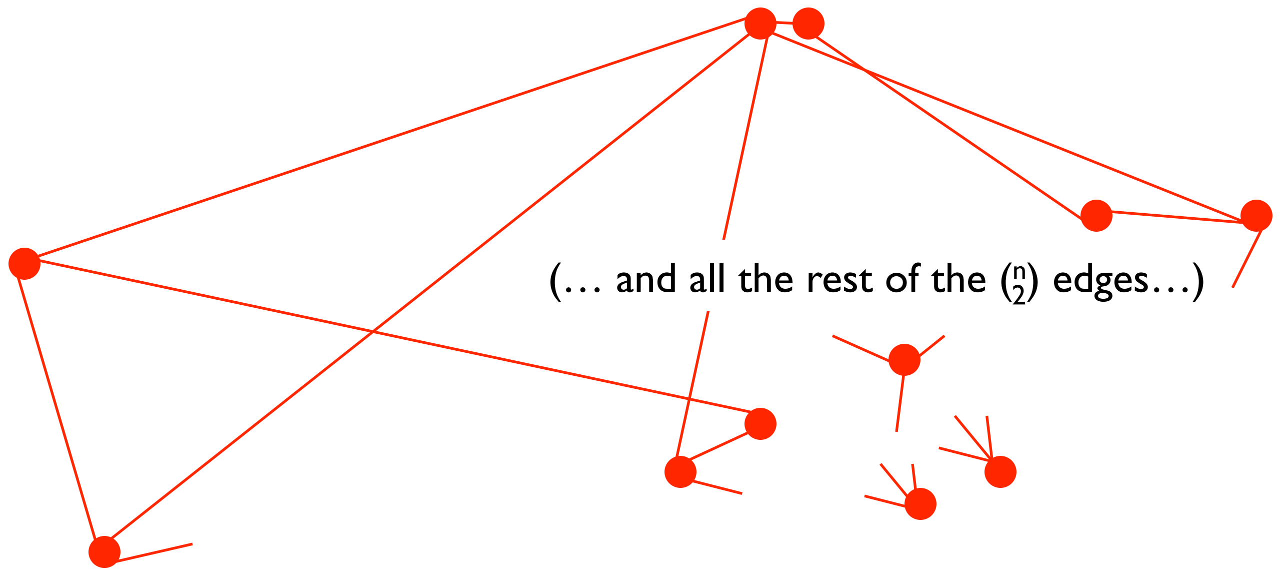
Algorithm summary

- Satisfies the "Standard recurrence"
- $T(n) = 2\,T(n/2) + cn$

A Divide & Conquer Example: Closest Pair of Points

closest pair of points: non-geometric version

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

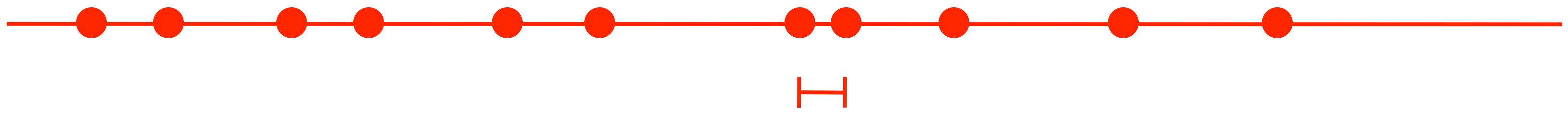


Must look at all n choose 2 pairwise distances, else any one you didn't check might be the shortest.

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: 2 dimensional version

Closest pair. Given n points in the plane, find a pair with smallest Euclidean distance between them.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

$\nwarrow$ fast closest pair inspired fast algorithms for these problems

Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

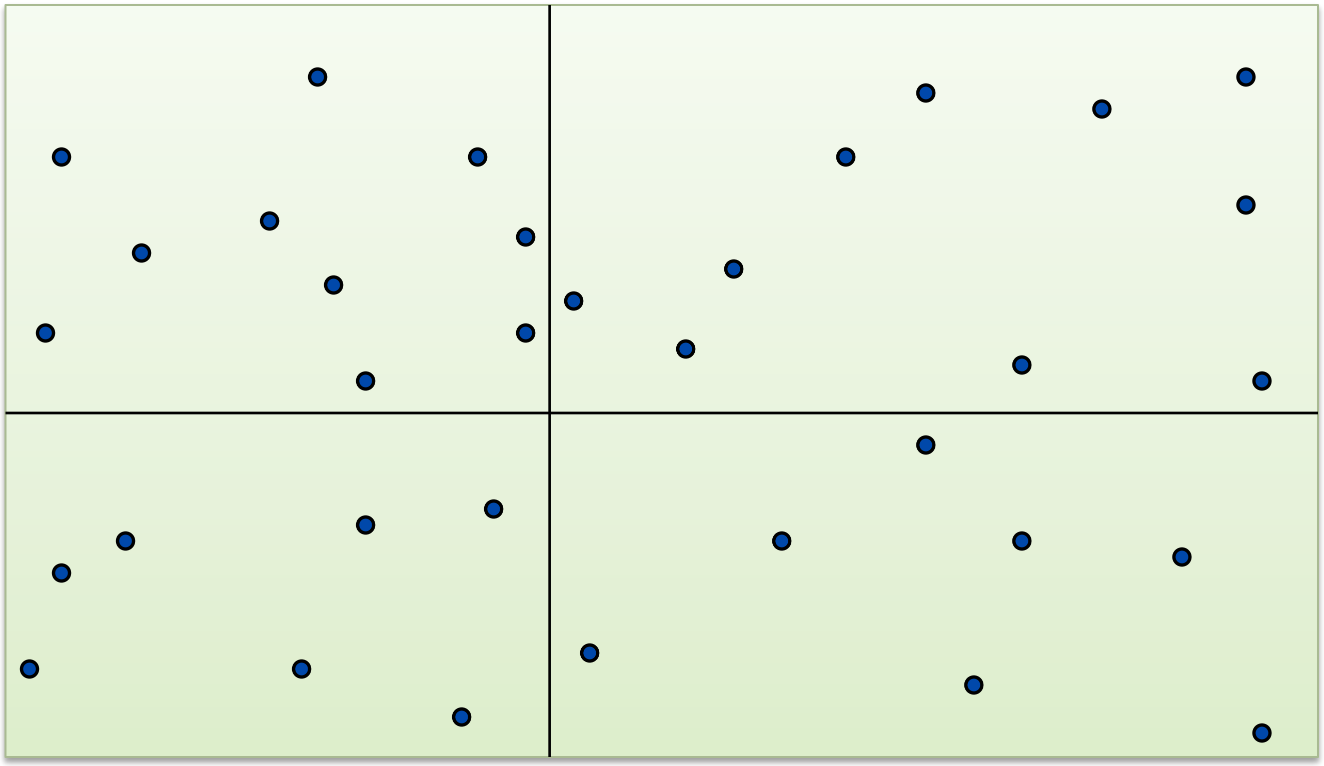
1-D version. O(n log n) easy if points are on a line.

Assumption. No two points have same x coordinate.

$\uparrow$ Just to simplify presentation

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

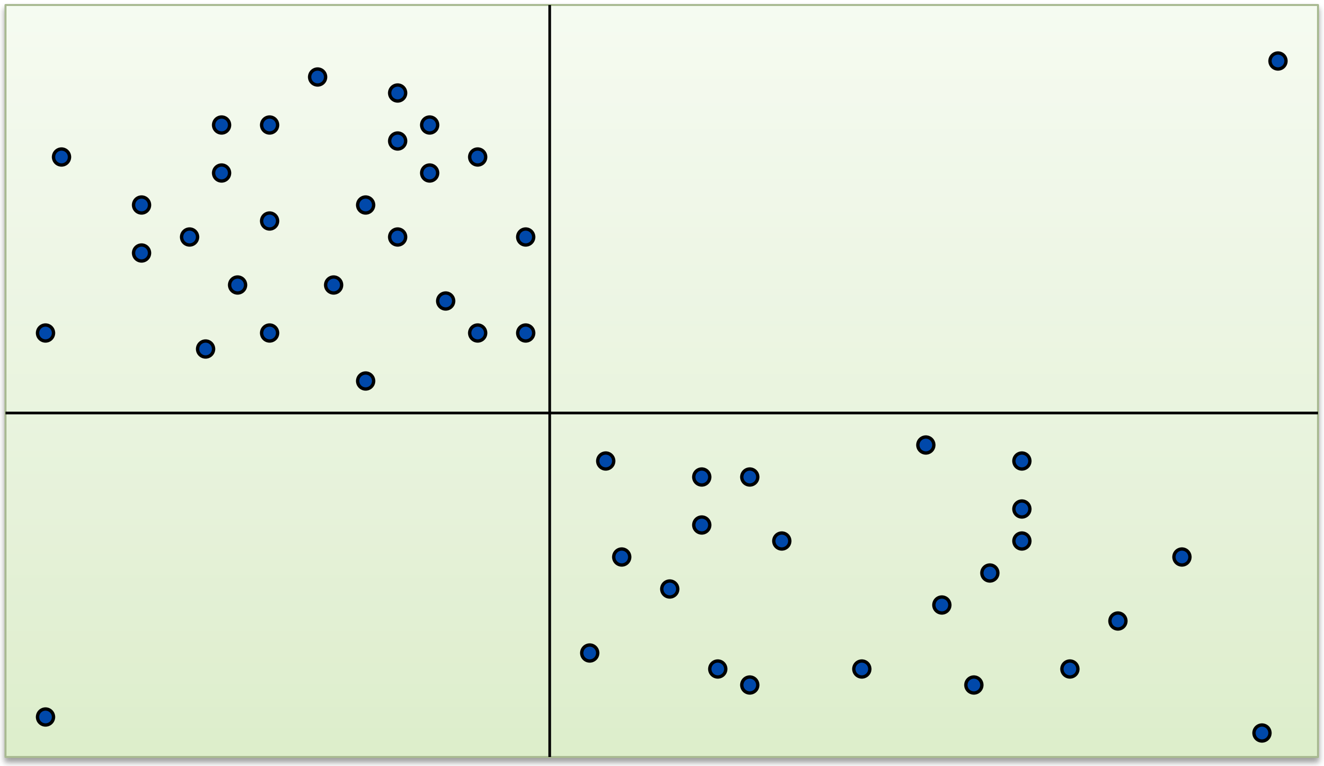
Divide. Sub-divide region into 4 quadrants.



closest pair of points: 1st try

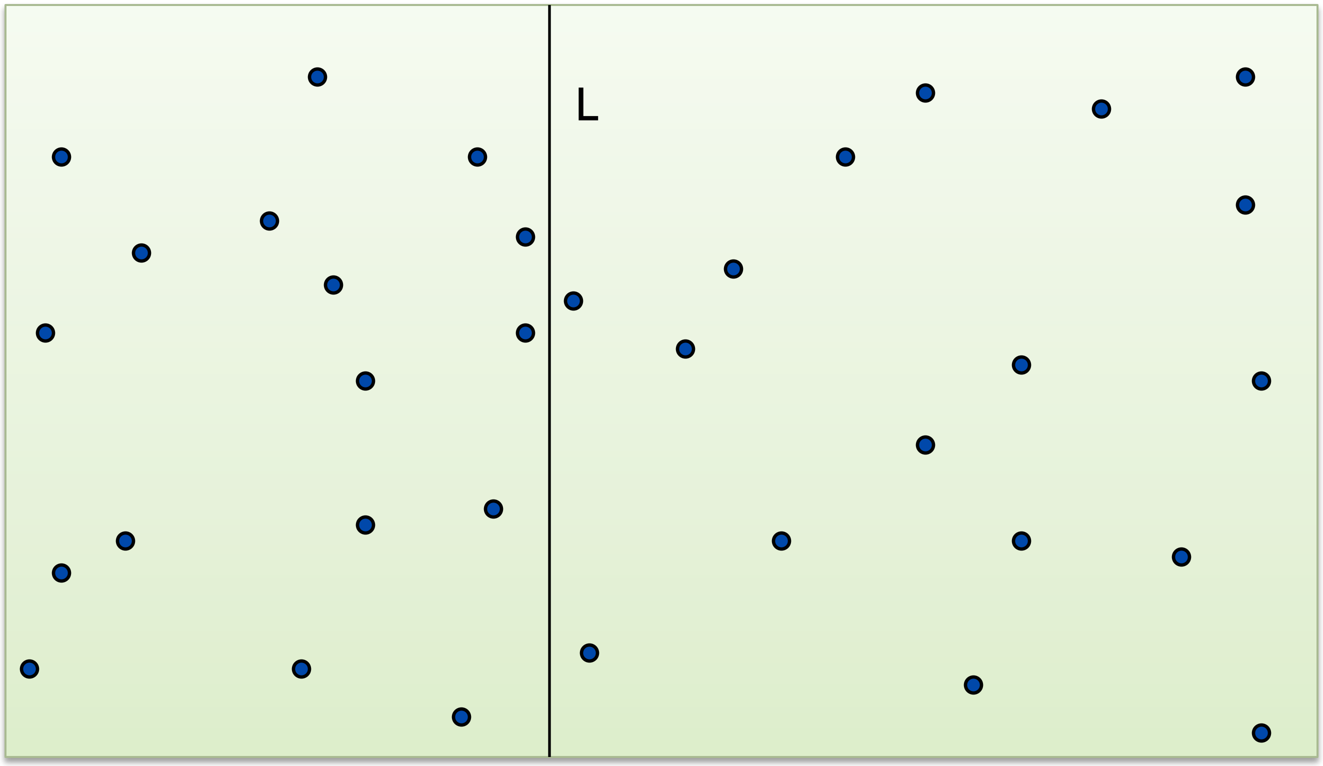
Divide. Sub-divide region into 4 quadrants.

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



Algorithm.

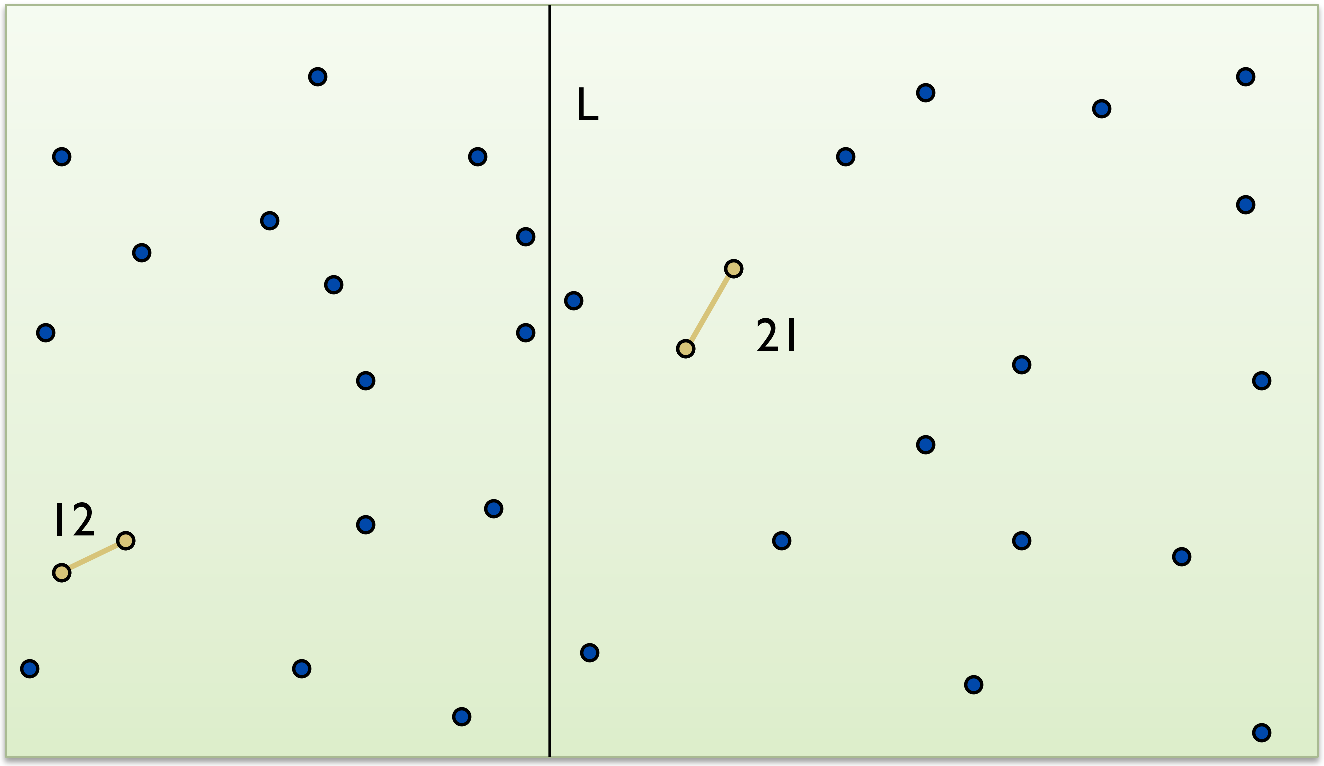
Divide: draw vertical line L with ≈ n/2 points on each side.



Algorithm.

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

Conquer: find closest pair on each side, recursively.



Algorithm.

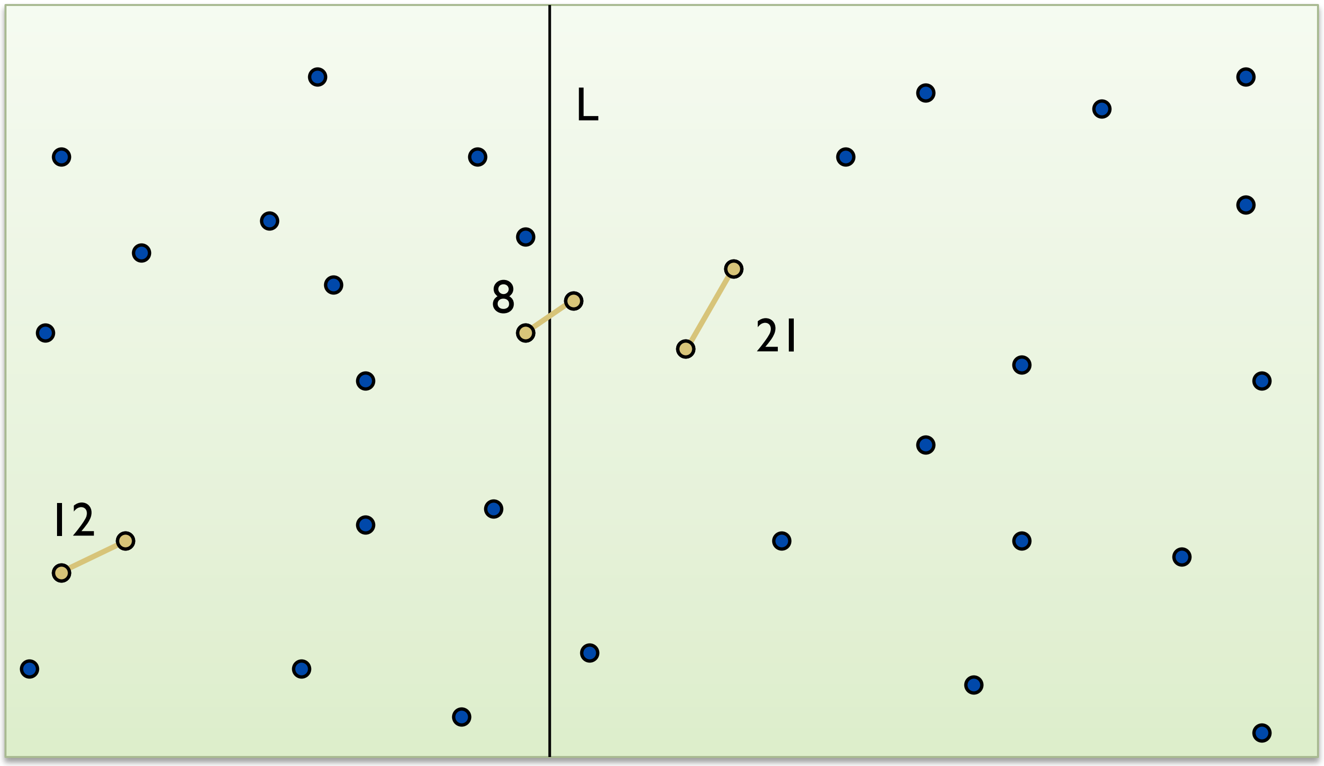
Divide: draw vertical line L with ≈ n/2 points on each side.

Conquer: find closest pair on each side, recursively.

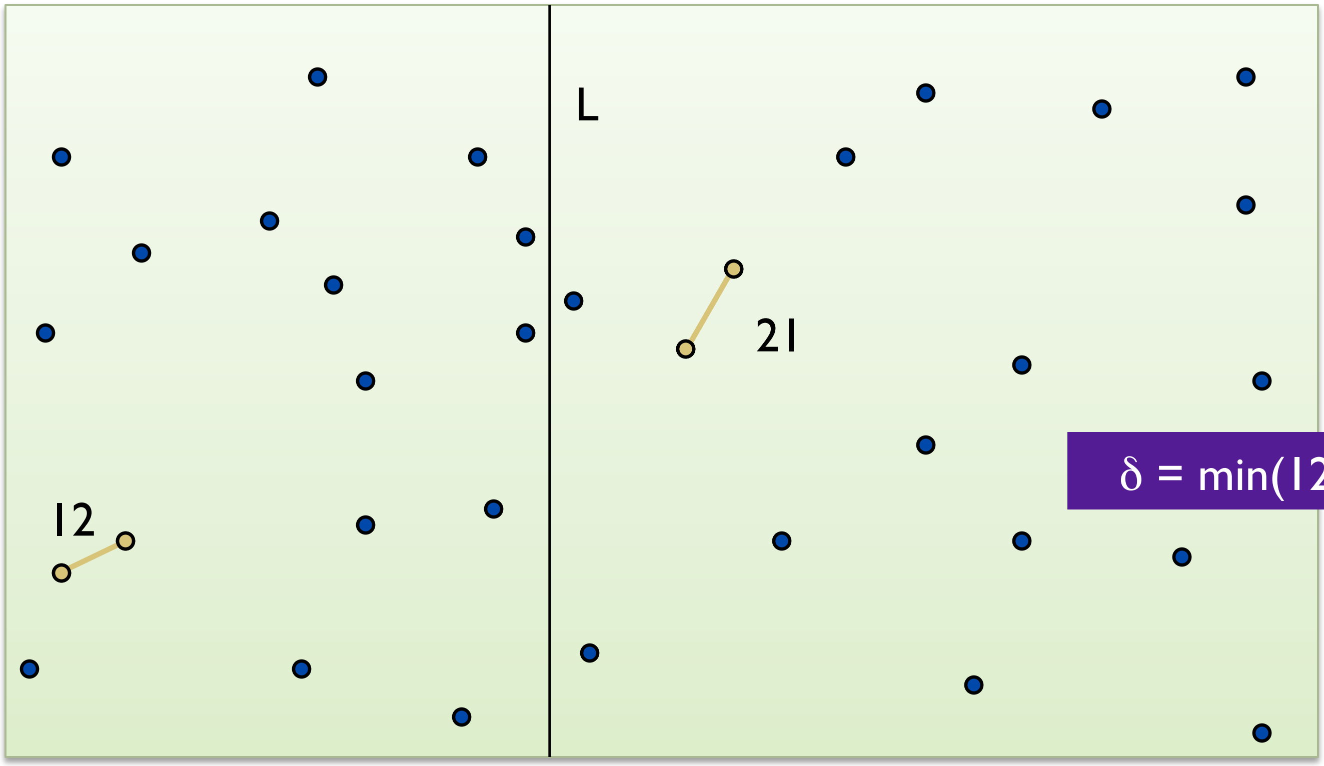
Combine: find closest pair with one point in each side.

Return best of 3 solutions.

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



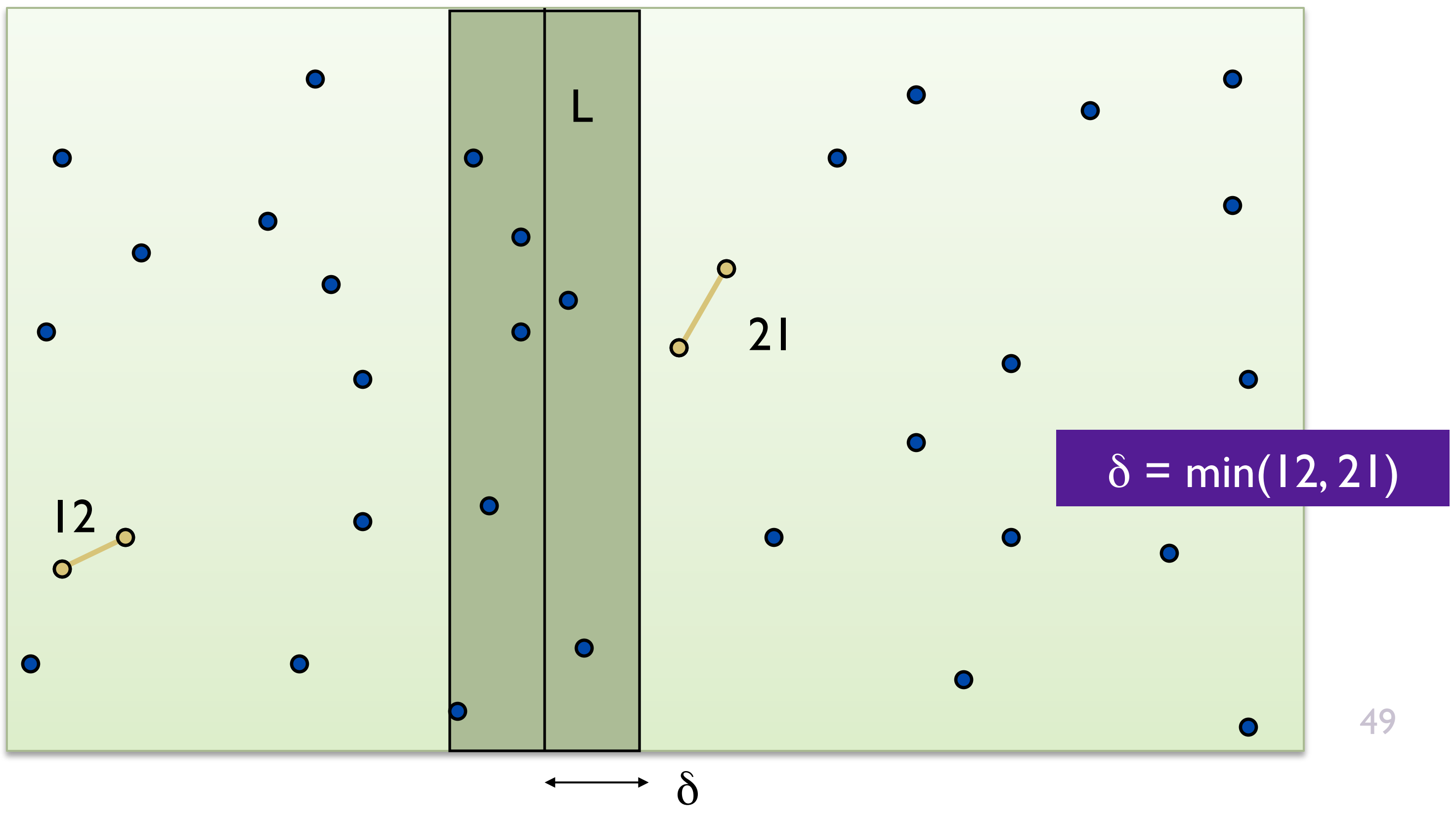
Find closest pair with one point in each side, *assuming distance < δ.*



δ = min(12, 21)

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

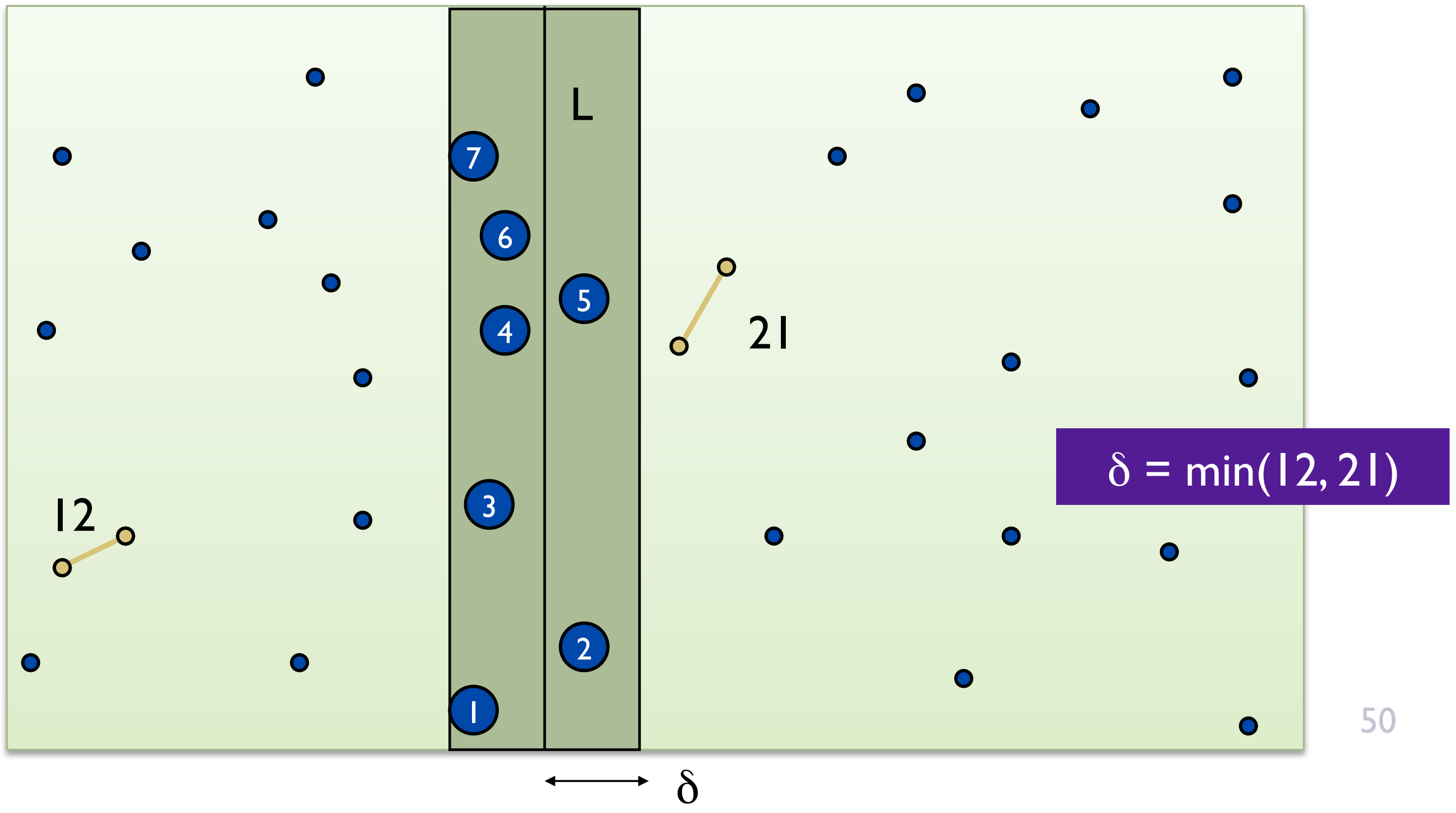
Observation: suffices to consider points within δ of line L.



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

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

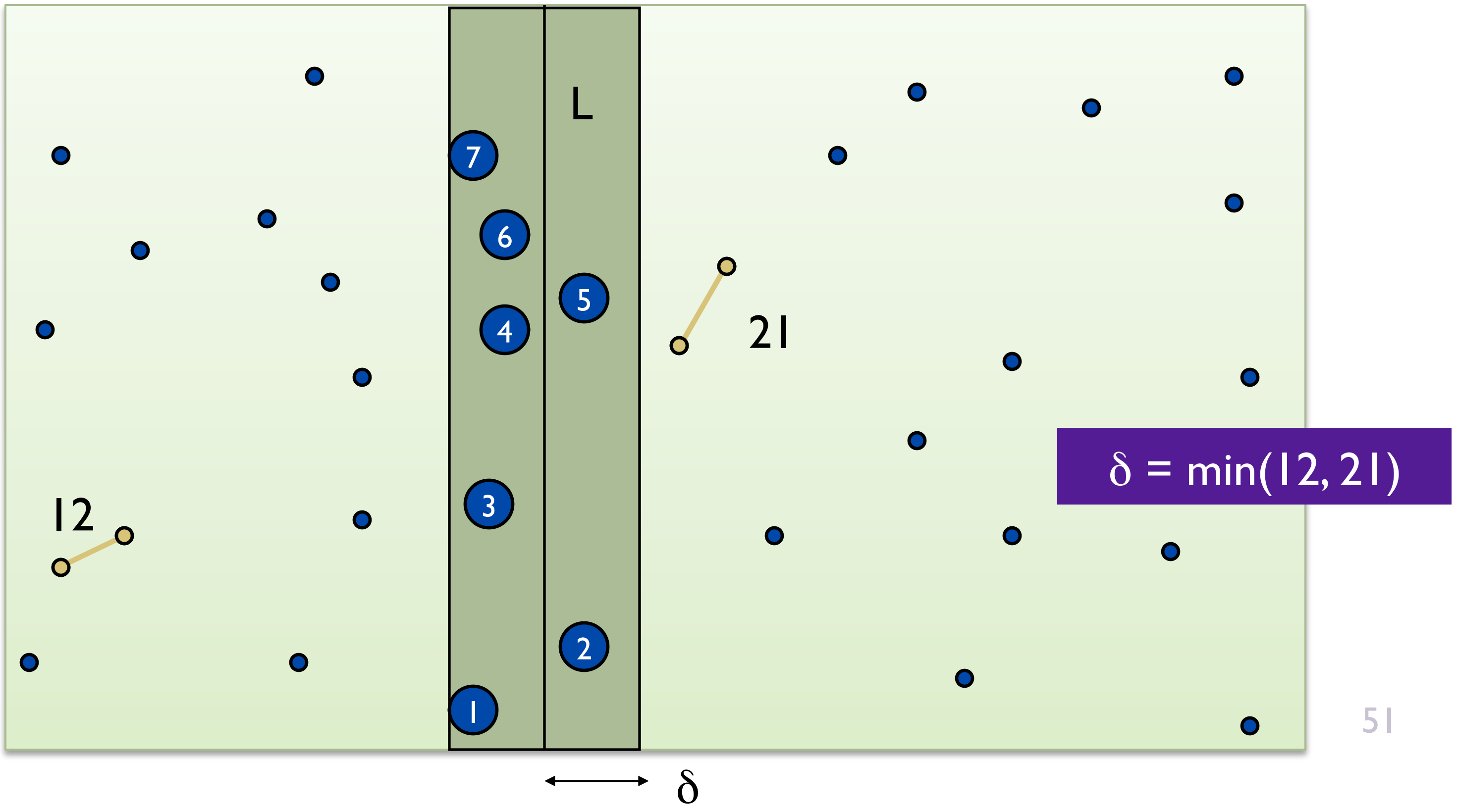
Almost the one-D problem again: Sort points in 2δ-strip by their y coordinate.



Find closest pair with one point in each side, *assuming* 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!



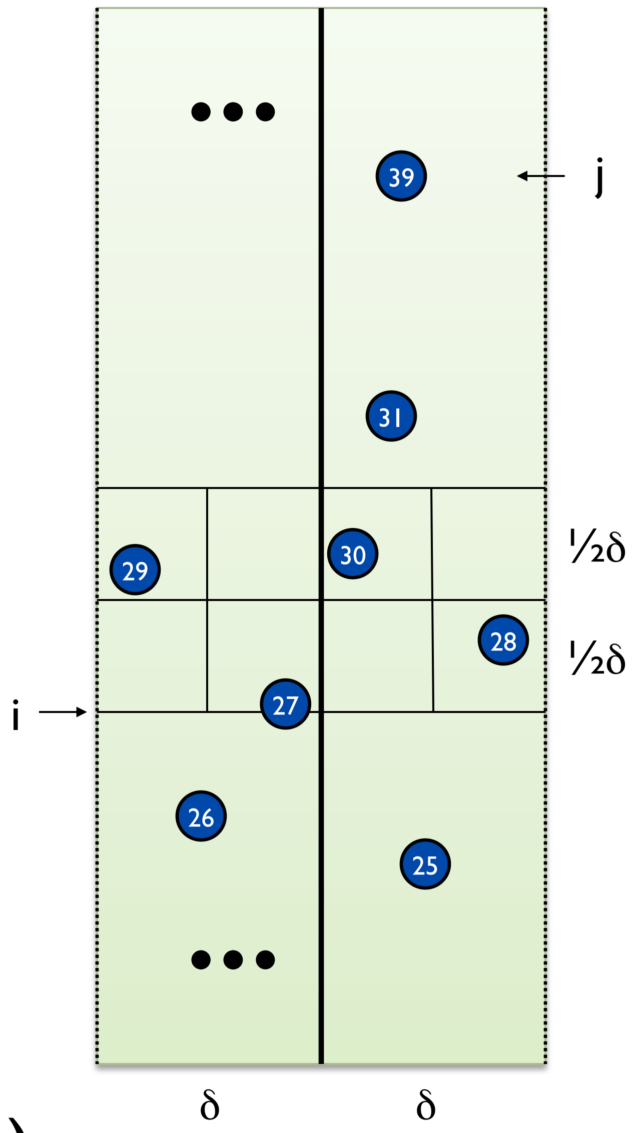
Def. Let s_i have the ith smallest y-coordinate among points in the 2δ-width-strip.

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

Pf: No two points lie in the same ½δ-by-½δ box:

$$\sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2} = \sqrt{\frac{1}{2}} = \frac{\sqrt{2}}{2} \approx 0.7 < 1$$

so ≤ 8 boxes within +δ of $y(s_i)$.



closest pair algorithm

```
Closest-Pair(p_1, …, p_n) {
   if(n <= ??) return ??

   Compute separation line L such that half the points
   are on one side and half on the other side.

   δ_1 = Closest-Pair(left half)
   δ_2 = Closest-Pair(right half)
   δ  = min(δ_1, δ_2)

   Delete all points further than δ from separation line L

   Sort remaining points p[1]…p[m] by y-coordinate.

   for i = 1..m
      k = 1
      while i+k <= m && p[i+k].y < p[i].y + δ
        δ = min(δ, distance between p[i] and p[i+k]);
        k++;

   return δ.
}
```

closest pair of points: analysis

Analysis, I: Let D(n) be the number of pairwise distance calculations in the Closest-Pair Algorithm when run on n ≥ 1 points

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

BUT – that's only the number of *distance calculations*

What if we counted comparisons?

closest pair of points: analysis

Analysis, II: Let C(n) be the number of comparisons between coordinates/distances in the Closest-Pair Algorithm when run on n ≥ 1 points

$$C(n) \le \left\{ \begin{array}{cc} 0 & n = 1 \\ 2C(n/2) + kn \log n & n > 1 \end{array} \right\} \Rightarrow C(n) = O(n \log^2 n)$$

for some constant k

Q. Can we achieve O(n log n)?

A. Yes. Don't sort points from scratch each time.

Sort by x at top level only.

Each recursive call returns δ *and* list of all points sorted by y

Sort by merging two pre-sorted lists.

$$T(n) \le 2T(n/2) + O(n) \Rightarrow \mathrm{T}(n) = O(n \log n)$$

is it worth the effort?

Code is longer & more complex

$O(n \log n)$ vs $O(n^2)$ may hide 10x in constant?

How many points?

n	**Speedup:** **$n^2 / (10\ n \log_2 n)$**
10	0.3
100	1.5
1,000	10
10,000	75
100,000	602
1,000,000	5,017
10,000,000	43,004

Going From Code to Recurrence

going from code to recurrence

Carefully define what you're counting, and *write it down*!

"Let C(n) be the number of comparisons between sort keys used by MergeSort when sorting a list of length n ≥ 1"

In code, clearly separate *base case* from *recursive case,* highlight *recursive calls,* and *operations being counted.*

Write Recurrence(s)

merge sort

Base Case

```
MS(A: array[1..n]) returns array[1..n] {
    If(n=1) return A;
    New L:array[1:n/2] = MS(A[1..n/2]);
    New R:array[1:n/2] = MS(A[n/2+1..n]);
    Return(Merge(L,R));
    }
Merge(A,B: array[1..n]) {
    New C: array[1..2n];
    a=1; b=1;
    For i = 1 to 2n {
        C[i] = "smaller of A[a], B[b] and a++ or b++";
    Return C;
    }
```

Recursive calls

One Recursive Level

Operations being counted

the recurrence

Base case

$$C(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2C(n/2) + (n-1) & \text{if } n > 1 \end{cases}$$

Recursive calls

One compare per element added to merged list, except the last.

Total time: proportional to C(n)

(loops, copying data, parameter passing, etc.)

going from code to recurrence

Carefully define what you're counting, and *write it down*!

> "Let D(n) be the number of pairwise distance calculations in the Closest-Pair Algorithm when run on n ≥ 1 points"

In code, clearly separate *base case* from *recursive case,* highlight *recursive calls,* and *operations being counted*.

Write Recurrence(s)

closest pair algorithm

Basic operations: distance calcs

```
Closest-Pair(p_1, …, p_n) {
   if(n <= 1) return ∞

   Compute separation line L such that half the points
   are on one side and half on the other side.

   δ_1 = Closest-Pair(left half)
   δ_2 = Closest-Pair(right half)
   δ  = min(δ_1, δ_2)

   Delete all points further than δ from separation line L

   Sort remaining points p[1]…p[m] by y-coordinate.

   for i = 1..m
      k = 1
      while i+k <= m && p[i+k].y < p[i].y + δ
        δ = min(δ, distance between p[i] and p[i+k]);
        k++;

   return δ.
}
```

Base Case

0

Recursive calls (2)

2D(n / 2)

One recursive level

Basic operations at this recursive level

7n

closest pair of points: analysis

Analysis, I: Let D(n) be the number of pairwise distance calculations in the Closest-Pair Algorithm when run on n ≥ 1 points

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

BUT – that's only the number of *distance calculations*

What if we counted comparisons?

going from code to recurrence

Carefully define what you're counting, and *write it down*!

> "Let D(n) be the number of comparisons between coordinates/distances in the Closest-Pair Algorithm when run on n ≥ 1 points"

In code, clearly separate *base case* from *recursive case,* highlight *recursive calls,* and *operations being counted*.

Write Recurrence(s)

closest pair algorithm

Basic operations:
comparisons

```
Closest-Pair(p₁, …, pₙ) {
   if(n <= 1) return ∞

   Compute separation line L such that half the points
   are on one side and half on the other side.

   δ₁ = Closest-Pair(left half)
   δ₂ = Closest-Pair(right half)
   δ  = min(δ₁, δ₂)

   Delete all points further than δ from separation line L

   Sort remaining points p[1]…p[m] by y-coordinate.

   for i = 1..m
      k = 1
      while i+k <= m && p[i+k].y < p[i].y + δ
         δ = min(δ, distance between p[i] and p[i+k]);
         k++;

   return δ.
}
```

Base Case

Recursive calls (2)

Basic operations at this recursive level

One recursive level

0

$k_1 n \log n$

$2C(n / 2)$

1

$k_2 n$

$k_3 n \log n$

$7n$

closest pair of points: analysis

Analysis, II: Let C(n) be the number of comparisons of coordinates/distances in the Closest-Pair Algorithm when run on n ≥ 1 points

$$C(n) \le \left\{ \begin{array}{cc} 0 & n=1 \\ 2C(n/2) + k_4 n \log n & n>1 \end{array} \right\} \Rightarrow C(n) = O(n \log^2 n)$$

for some $k_4 \le k_1 + k_2 + k_3 + 7$

Q. Can we achieve time O(n log n)?

A. Yes. Don't sort points from scratch each time.

Sort by x at top level only.

Each recursive call returns δ *and* list of all points sorted by y

Sort by merging two pre-sorted lists.

$$T(n) \le 2T(n/2) + O(n) \Rightarrow \mathrm{T}(n) = O(n \log n)$$

Integer Multiplication

integer arithmetic

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

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

O(n) bit operations.

integer arithmetic

Add. Given two n-bit 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-bit 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

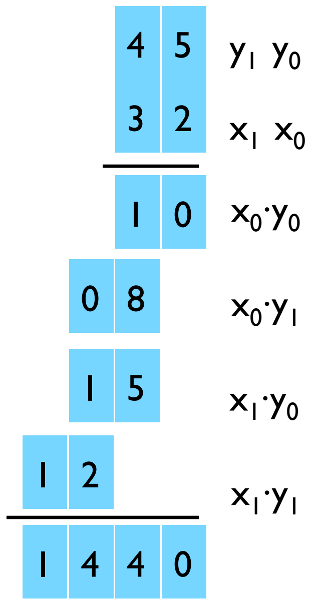
To multiply two 2-digit integers:

Multiply four 1-digit integers.

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

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

Same idea works for *long* integers – can split them into 4 half-sized ints



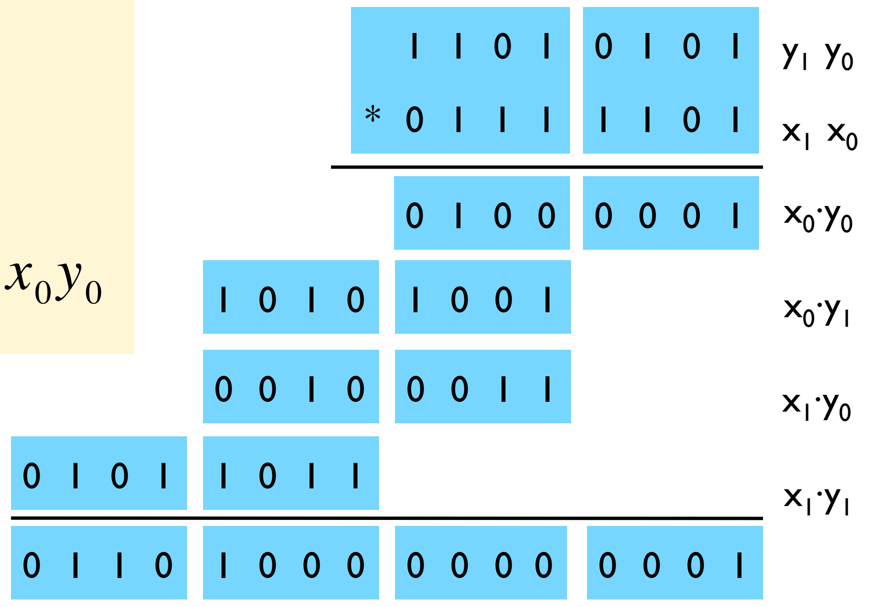
divide & conquer multiplication: warmup

To multiply two n-bit integers:

Multiply four ½n-bit integers.

Add two ½n-bit integers, and shift to obtain result.

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



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

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

$\uparrow$

assumes n is a power of 2

key trick: 2 multiplies for the price of 1:

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

Well, ok, 4 for 3 is more accurate…

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

Karatsuba multiplication

To multiply two n-bit integers:

- Add two ½n bit integers.
- Multiply three ½n-bit integers.
- Add, subtract, and shift ½n-bit integers to obtain result.

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

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

$$
\mathrm{T}(n) \le \underbrace{T\left(\lfloor n/2 \rfloor \right) + T\left(\lceil n/2 \rceil \right) + T\left(1 + \lceil n/2 \rceil \right)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}
$$

$$
\textit{Sloppy version}: \; T(n) \le 3T(n/2) + O(n)
$$

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

Karatsuba multiplication

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

$$\mathrm{T}(n) \le \underbrace{T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + T\left(1+\lceil n/2 \rceil\right)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}$$

$$\textit{Sloppy version}:\ T(n) \le 3T(n/2) + O(n)$$

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

multiplication – the bottom line

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: $\Theta(n \log n \log\log n)$

- "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

Polynomial Multiplication

Another D&C Example: Multiplying Polynomials

Similar ideas apply to polynomial multiplication

We'll describe the basic ideas by multiplying polynomials rather than integers

In fact, it's somewhat simpler: no carries!

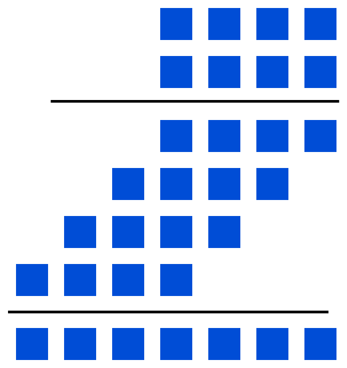
Notes on Polynomials

These are just formal sequences of coefficients so when we show something multiplied by x^k it just means shifted k places to the left – basically no work

Usual Polynomial Multiplication:

$$\begin{array}{r}
3x^2 + 2x + 2 \\
x^2 - 3x + 1 \\
\hline
3x^2 + 2x + 2 \\
-9x^3 - 6x^2 - 6x \phantom{{}+2} \\
3x^4 + 2x^3 + 2x^2 \phantom{{}- 6x + 2} \\
\hline
3x^4 - 7x^3 - x^2 - 4x + 2
\end{array}$$

Polynomial Multiplication



Given:

Degree $m-1$ polynomials P and Q

$$P = a_0 + a_1 x + a_2 x^2 + \ldots + a_{m-2}x^{m-2} + a_{m-1}x^{m-1}$$

$$Q = b_0 + b_1 x + b_2 x^2 + \ldots + b_{m-2}x^{m-2} + b_{m-1}x^{m-1}$$

Compute:

Degree $2m-2$ Polynomial PQ

$$PQ = a_0b_0 + (a_0b_1+a_1b_0)\,x + (a_0b_2+a_1b_1+a_2b_0)\,x^2 + \ldots + (a_{m-2}b_{m-1}+a_{m-1}b_{m-2})\,x^{2m-3} + a_{m-1}b_{m-1}\,x^{2m-2}$$

Obvious Algorithm:

Compute all a_ib_j and collect terms

$\Theta(m^2)$ time

Naïve Divide and Conquer

Assume $m=2k$

$$P = (a_0 + a_1 x + a_2 x^2 + \dots + a_{k-2} x^{k-2} + a_{k-1} x^{k-1}) + (a_k + a_{k+1} x + \dots + a_{m-2} x^{k-2} + a_{m-1} x^{k-1})\, x^k$$

$$= P_0 + P_1 x^k$$

$$Q = Q_0 + Q_1 x^k$$

$$PQ = (P_0+P_1x^k)(Q_0+Q_1x^k)$$
$$= P_0Q_0 + (P_1Q_0+P_0Q_1)x^k + P_1Q_1x^{2k}$$

4 sub-problems of size $k=m/2$ plus linear combining

$T(m)=4T(m/2)+cm$

Solution $T(m) = O(m^2)$

Karatsuba's Algorithm

A better way to compute terms

Compute

P_0Q_0

P_1Q_1

$(P_0+P_1)(Q_0+Q_1)$ which is $P_0Q_0\underline{+P_1Q_0+P_0Q_1}+P_1Q_1$

Then

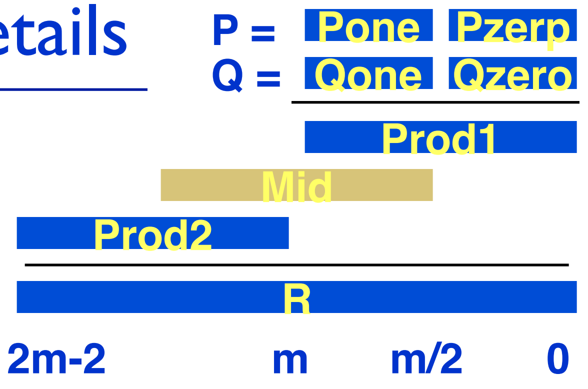
$P_0Q_1+P_1Q_0 = (P_0+P_1)(Q_0+Q_1) - P_0Q_0 - P_1Q_1$

3 sub-problems of size m/2 plus O(m) work

$T(m) = 3\ T(m/2) + cm$

$T(m) = O(m^\alpha)$ where $\alpha = \log_2 3 = 1.585\ldots$

Karatsuba: Details



```
PolyMul(P, Q):
    // P, Q are length m = 2k vectors, with P[i], Q[i] being
    // the coefficient of x^i in polynomials P, Q respectively.
    if (m==1) return (P[0]*Q[0]);
    Let Pzero be elements 0..k-1 of P; Pone be elements k..m-1
    Qzero, Qone : similar
    Prod1 = PolyMul(Pzero, Qzero);        // result is a (2k-1)-vector
    Prod2 = PolyMul(Pone, Qone);          // ditto
    Pzo = Pzero + Pone;                   // add corresponding elements
    Qzo = Qzero + Qone;                   // ditto
    Prod3 = PolyMul(Pzo, Qzo);            // another (2k-1)-vector
    Mid = Prod3 – Prod1 – Prod2;          // subtract corr. elements
    R = Prod1 + Shift(Mid, m/2) + Shift(Prod2,m) // a (2m-1)-vector
    Return( R );
```

Multiplication – The Bottom Line

Polynomials

- Naïve: $\Theta(n^2)$
- Karatsuba: $\Theta(n^{1.585\ldots})$
- Best known: $\Theta(n \log n)$
  - "Fast Fourier Transform"

Integers

- Similar, but some ugly details re: carries, etc. gives $\Theta(n \log n \operatorname{loglog} n)$,
  - but mostly unused in practice

Median and Selection

Computing the Median

Median: Given n numbers, find the number of rank n/2 (to be precise, say: $\lceil n/2 \rceil$)

Selection: given n numbers and an integer k, find the k-th largest

- E.g., Median is $\lceil n/2 \rceil$-nd largest

“order statistics”

Can find max with n-1 comparisons

Can find 2nd largest with another n-2

3rd largest with another n-3

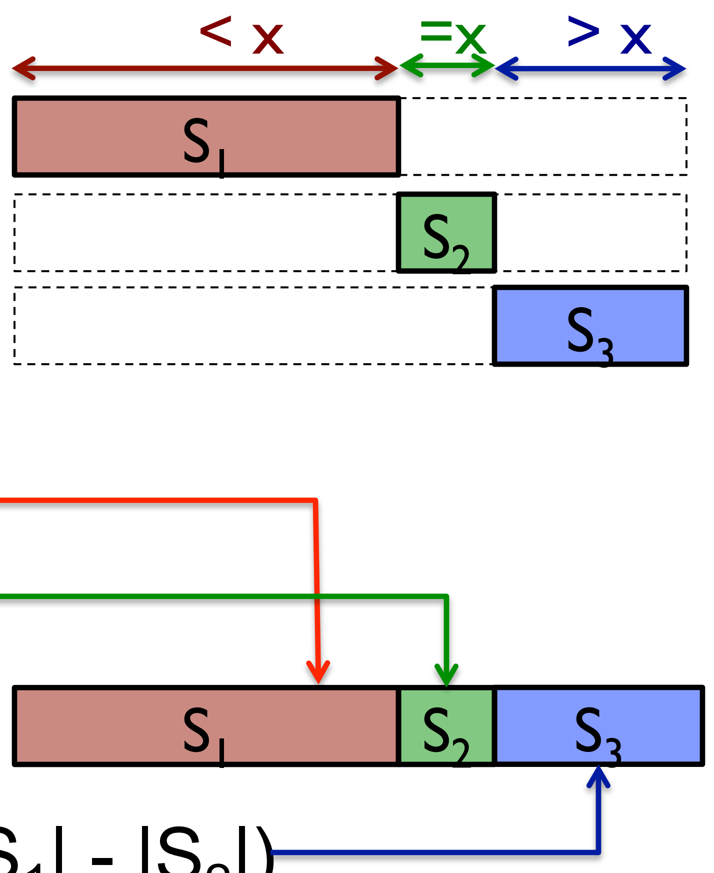
etc.: kth largest in O(kn)

What about k > log n?

Can we do better?

Select(A, k)

```
Select(A, k){
    Choose x from A
    $S_1$ = {y in A | y < x}
    $S_2$ = {y in A | y = x}
    $S_3$ = {y in A | y > x}
    if ($|S_1| \geq k$)
        return Select($S_1$, k)
    else if ($|S_1| + |S_2| \geq k$)
        return x
    else
        return Select($S_3$, k - $|S_1|$ - $|S_2|$)
}
```



Randomized Selection

Choose the element *at random*

Analysis (not here) can show that the algorithm has *expected* run time O(n)

Sketch: a random element eliminates, on average, ~ ½ of the data

Although worst case is $\Theta(n^2)$, albeit improbable (like Quicksort), for most purposes this is the method of choice

Worst case matters? Read on…

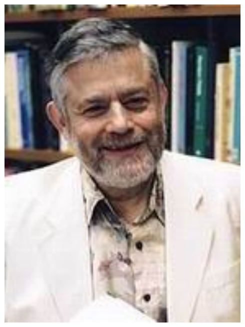
Deterministic Selection

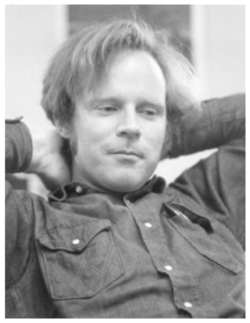
What is the run time of select if we can guarantee that "choose" finds an x such that $|S_1| < 3n/4$ and $|S_3| < 3n/4$

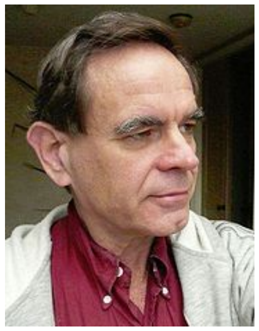
BFPRT Algorithm

A very clever "choose" algorithm . . .

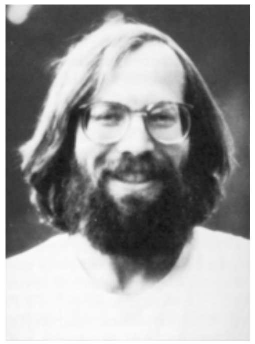
Split into n/5 sets of size 5
M be the set of medians of these sets
Return x = the median of M


M. Blum


R. Floyd


V. Pratt

R. Rivest


R. Tarjan

BFPRT runtime

Split into n/5 sets of size 5

Let M be the set of medians of these sets

Choose x to be the median of M

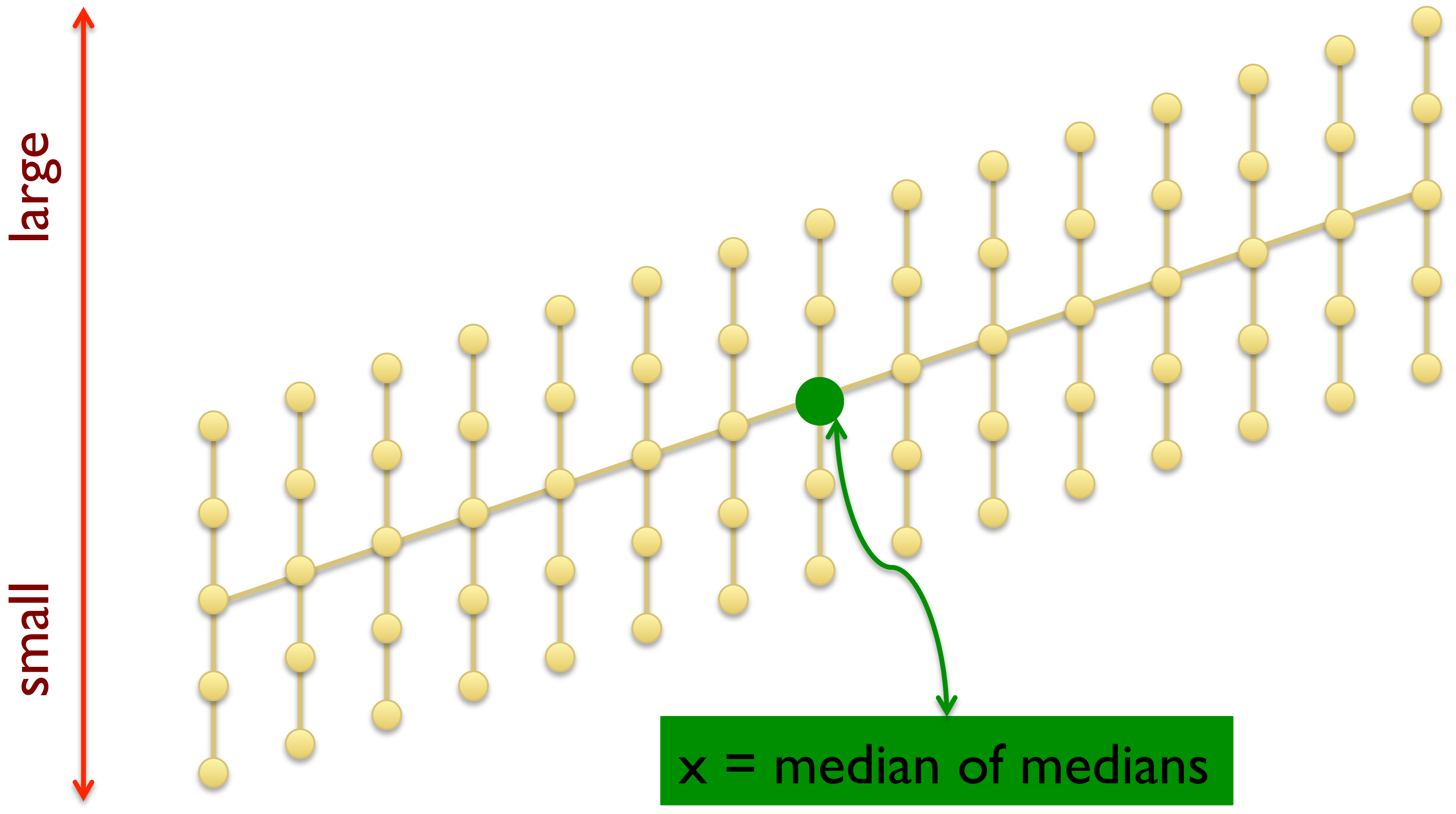
Construct S_1 , S_2 and S_3 as above

Recursive call in S_1 or S_3

To show: $|S_1| < 3n/4$, $|S_3| < 3n/4$

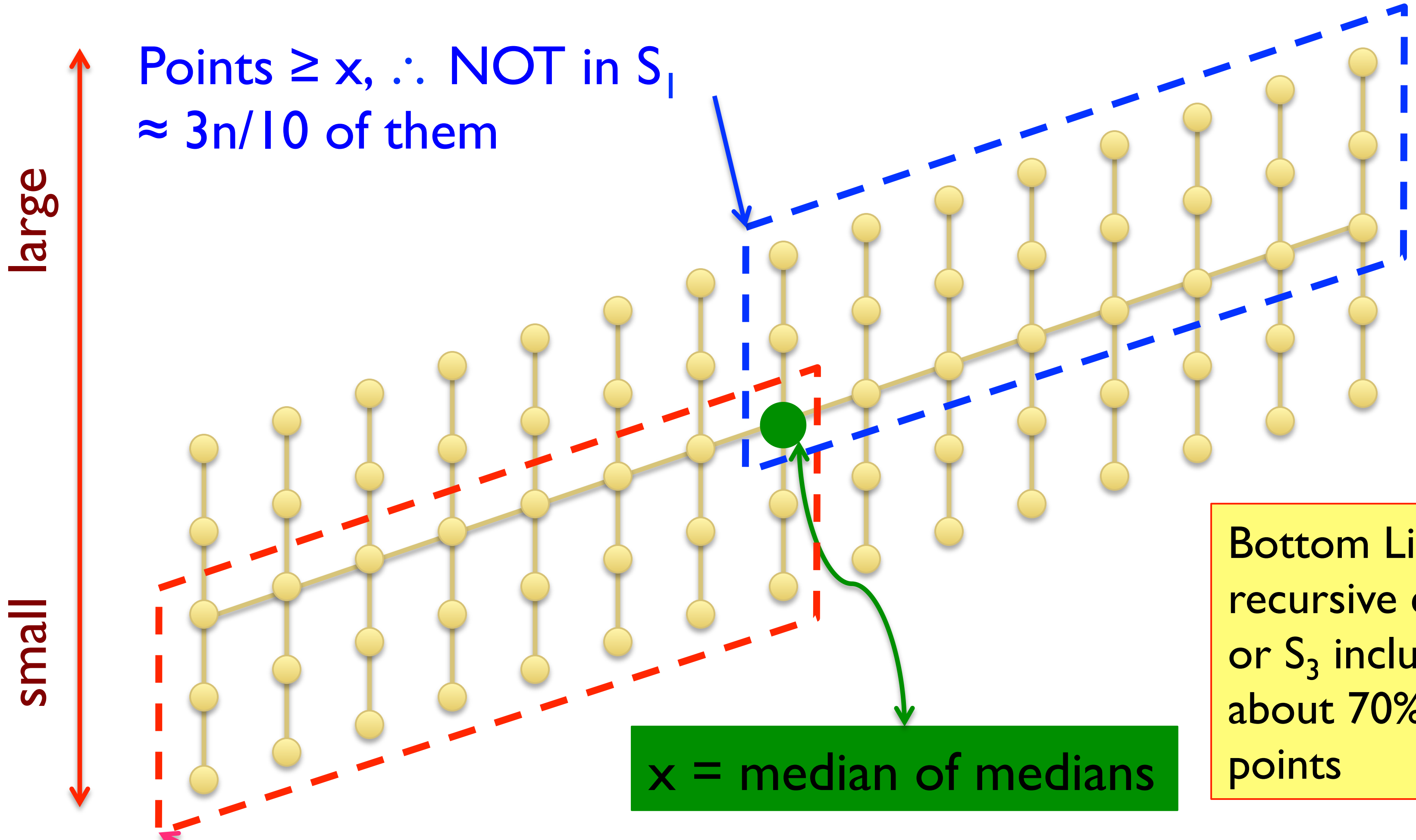
$n/5 + 3n/4 = 0.95n \Rightarrow O(n)$, worst case

Median of Medians



NB: conceptual; algorithm finds median(s), but does not sort

Median of Medians



Points ≥ x, ∴ NOT in S_1
≈ 3n/10 of them

x = median of medians

Points ≤ x, ∴ NOT in S_3
≈ 3n/10 of them

Bottom Line: recursive call on S_1 or S_3 includes only about 70% of points

NB: conceptual; algorithm finds median(s), but does not sort

BFPRT Recurrence

$\approx 7n/10$ points in subproblem

More precisely, various fussiness:

- $\lceil n/5 \rceil$ groups, all but (possibly) last of size 5
- Upper/lower half of $\geq \lfloor \lceil n/5 \rceil / 2 \rfloor$ groups excluded
- With some algebra, $\exists a,b,c$ such that:

$$T(n) \leq T(7n/10+a) + T(n/5+b) + c\,n$$

BFPRT Recurrence

$T(n) \leq T(7n/10+a) + T(n/5+b) + c\, n$

Prove that $T(n) <= 20\, c\, n$ for $n > 20(a+b)$

d & c summary

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,…

Exponentiation

another d&c example: fast exponentiation

Power(a,n)

Input: integer n and number a

Output: a^n

Obvious algorithm

$n-1$ multiplications

Observation:

if n is even, $n = 2m$, then $a^n = a^m \bullet a^m$

divide & conquer algorithm

```
Power(a,n)
    if $n = 0$ then return(1)
    if $n = 1$ then return(a)
    $x \leftarrow$ Power(a,$\lfloor n/2 \rfloor$)
    $x \leftarrow x \bullet x$
    if n is odd then
        $x \leftarrow a \bullet x$
    return(x)
```

Let M(n) be number of multiplies

Worst-case recurrence:

$$M(n) = \begin{cases} 0 & n \le 1 \\ M(\lfloor n/2 \rfloor) + 2 & n > 1 \end{cases}$$

By master theorem

M(n) = O(log n)   (a=1, b=2, k=0)

More precise analysis:

M(n) = $\lfloor \log_2 n \rfloor$ + (# of 1's in n's binary representation) - 1

Time is O(M(n)) if numbers < word size, else also depends on length, multiply algorithm

a practical application - RSA

Instead of a^n want a^n mod N

- $a^{i+j} \bmod N = ((a^i \bmod N) \cdot (a^j \bmod N)) \bmod N$
- same algorithm applies with each $x \cdot y$ replaced by
  - $((x \bmod N) \cdot (y \bmod N)) \bmod N$

In RSA cryptosystem (widely used for security)

- need a^n mod N where a, n, N each typically have 1024 bits
- Power: at most 2048 multiplies of 1024 bit numbers
  - relatively easy for modern machines
- Naive algorithm: 2^{1024} multiplies

d & c summary

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

Analysis: recursion tree or Master Recurrence

Applications: Many.

- Binary Search, Merge Sort, (Quicksort), counting inversions, closest points, median, integer/polynomial/matrix multiplication, FFT/convolution, exponentiation,…