
CSE 417
Algorithms:
Divide and Conquer

Larry Ruzzo

Thanks to Richard Anderson, Paul Beame, Kevin Wayne for some slides

algorithm design paradigms: divide and conquer

Outline:

- General Idea

- Review of Merge Sort

- Why does it work?

 - Importance of balance

 - Importance of super-linear growth

- Some interesting applications

 - Inversions

 - Closest points

 - Integer Multiplication

- Finding & Solving Recurrences

Divide & Conquer

Reduce a problem to one or more (smaller) sub-problems of the same type

Typically, each sub-problem is at most a constant fraction of the size of the original problem

Subproblems typically disjoint

Often gives significant, usually polynomial, speedup

Examples:

Binary Search, Mergesort, Quicksort (roughly),
Strassen's Algorithm, integer multiplication, powering,
FFT, ...

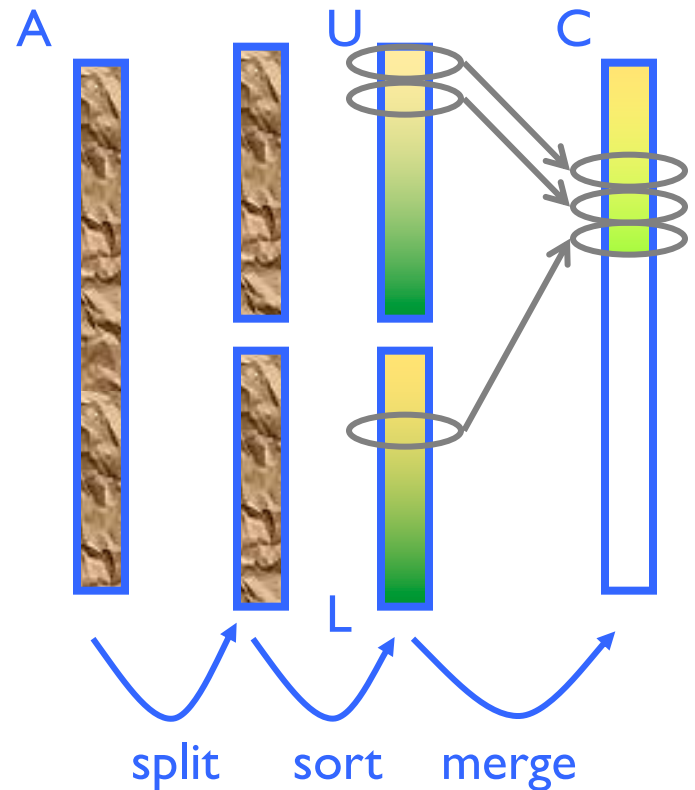
Motivating Example: Mergesort

```

MS(A: array[1..n]) returns array[1..n] {
  If(n=1) return A;
  New U:array[1:n/2] = MS(A[1..n/2]);
  New L:array[1:n/2] = MS(A[n/2+1..n]);
  Return(Merge(U,L));
}
    
```

```

Merge(U,L: array[1..n]) {
  New C: array[1..2n];
  a=1; b=1;
  For i = 1 to 2n
    "C[i] = smaller of U[a], L[b] and correspondingly a++ or b++,
    while being careful about running past end of either";
  Return C;
}
    
```



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

Why does it work? Suppose we've already invented DumbSort, taking time n^2

Try *Just One Level* of divide & conquer:

DumbSort(first $n/2$ elements) $O((n/2)^2)$

DumbSort(last $n/2$ elements) $O((n/2)^2)$

Merge results $O(n)$

Time: $2 (n/2)^2 + n = n^2/2 + n \ll n^2$

Almost twice as fast!



D&C in a
nutshell

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!

Moral 3: unbalanced division good, but less so:

$$(.1n)^2 + (.9n)^2 + n = .82n^2 + n$$

The 18% savings compounds significantly if you carry recursion to more levels, actually giving $O(n \log n)$, but with a bigger constant. So worth doing if you can't get 50-50 split, but balanced is better if you can.

This is intuitively why Quicksort with random splitter is good – badly unbalanced splits are rare, and not instantly fatal.

Moral 4: but consistent, completely unbalanced division doesn't help much:

$$(1)^2 + (n-1)^2 + n = n^2 - n + 2$$

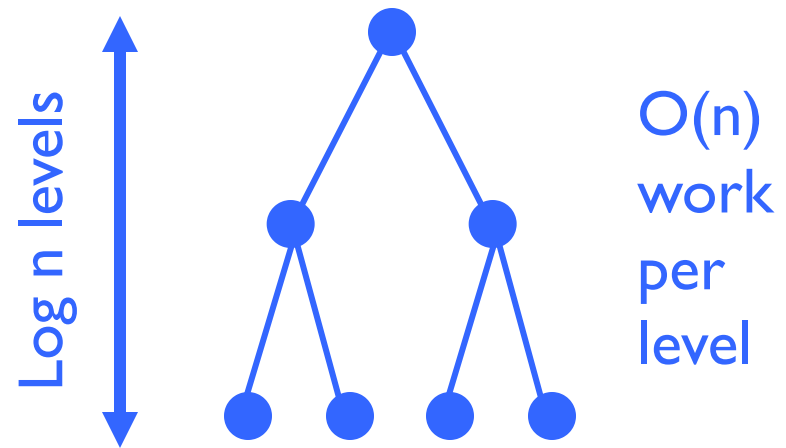
Little improvement here.

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

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

$$T(1) = 0$$

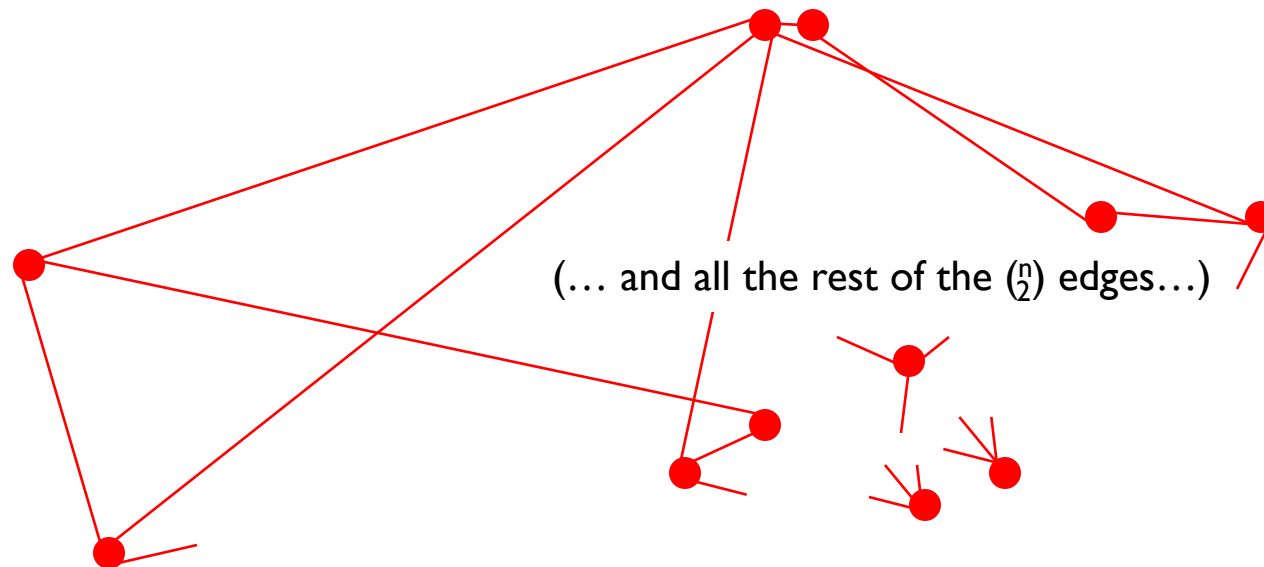
Solution: $\Theta(n \log n)$
(details later)



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!)

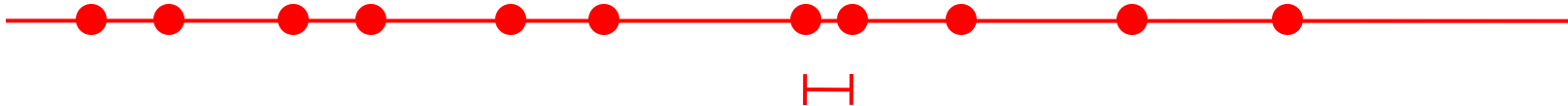


Must look at all n choose 2 pairwise distances, else any one you didn't check might be the shortest.

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.

↑
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.

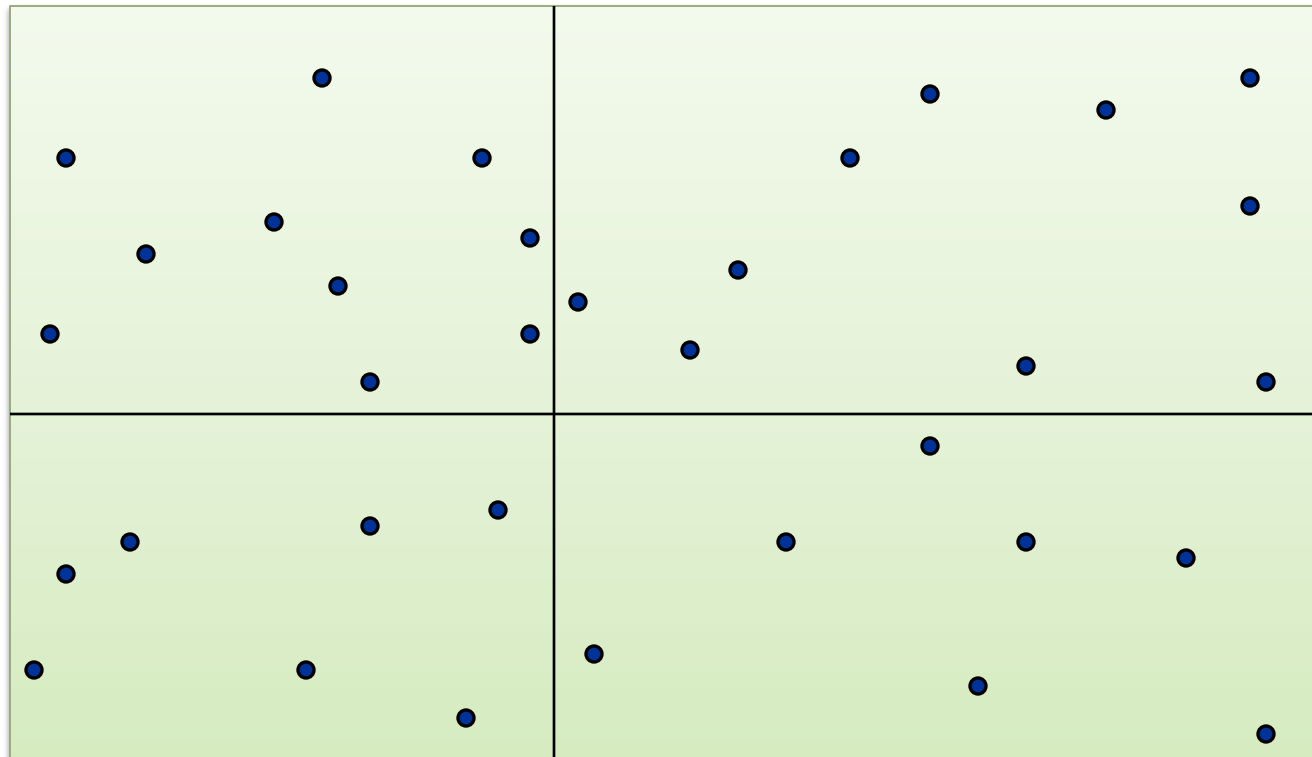
Can we do as well in 2-D?

Just to simplify presentation

Assumption. No two points have same x coordinate.

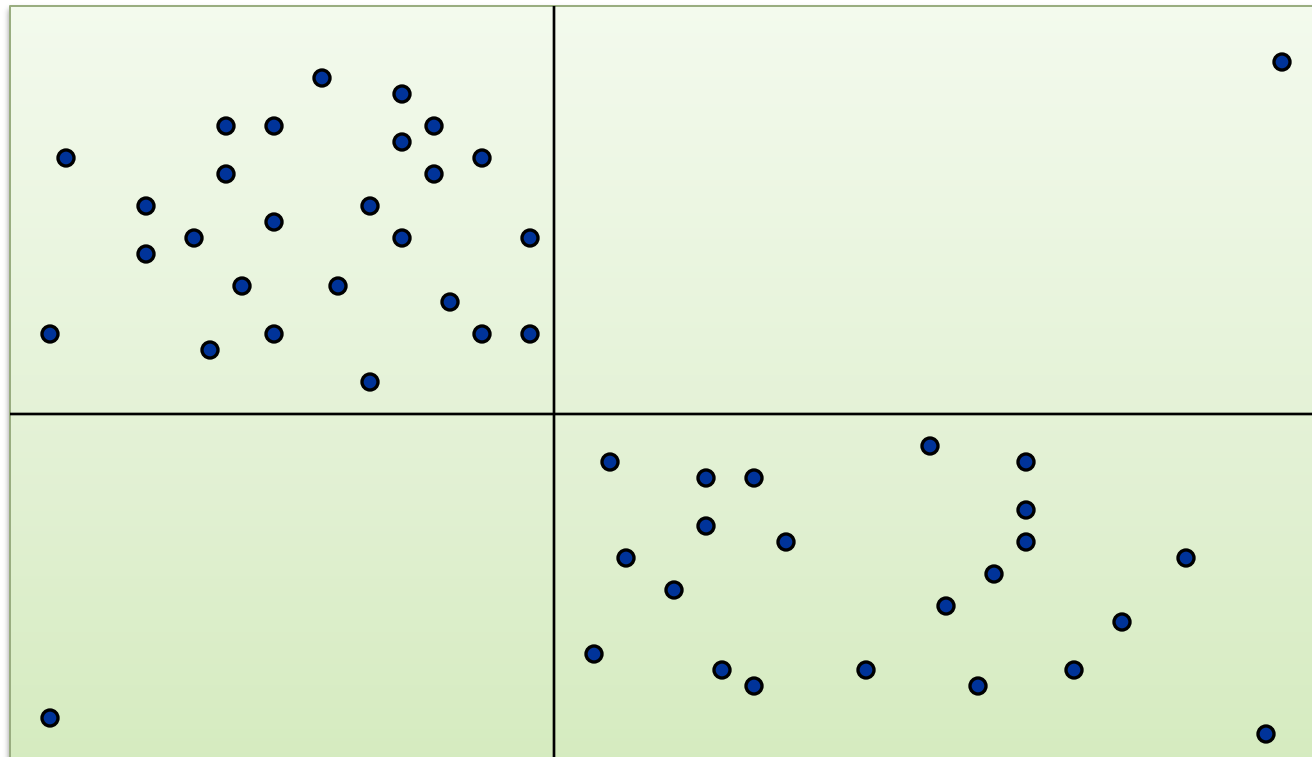
closest pair of points. 2d, Euclidean distance: 1st try

Divide. Sub-divide region into 4 quadrants.



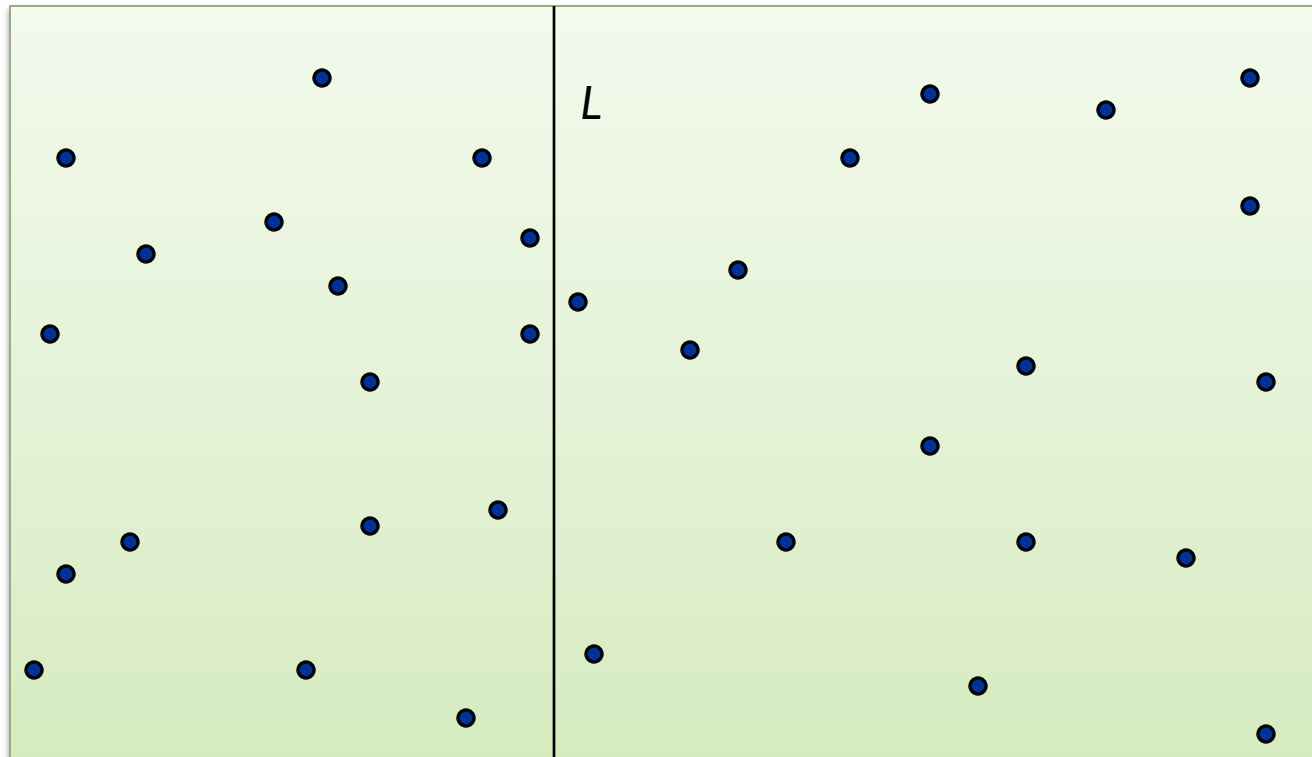
Divide. Sub-divide region into 4 quadrants.

Obstacle. Impossible to ensure $n/4$ points in each piece, so the “balanced subdivision” goal may be elusive/problematic.



Algorithm.

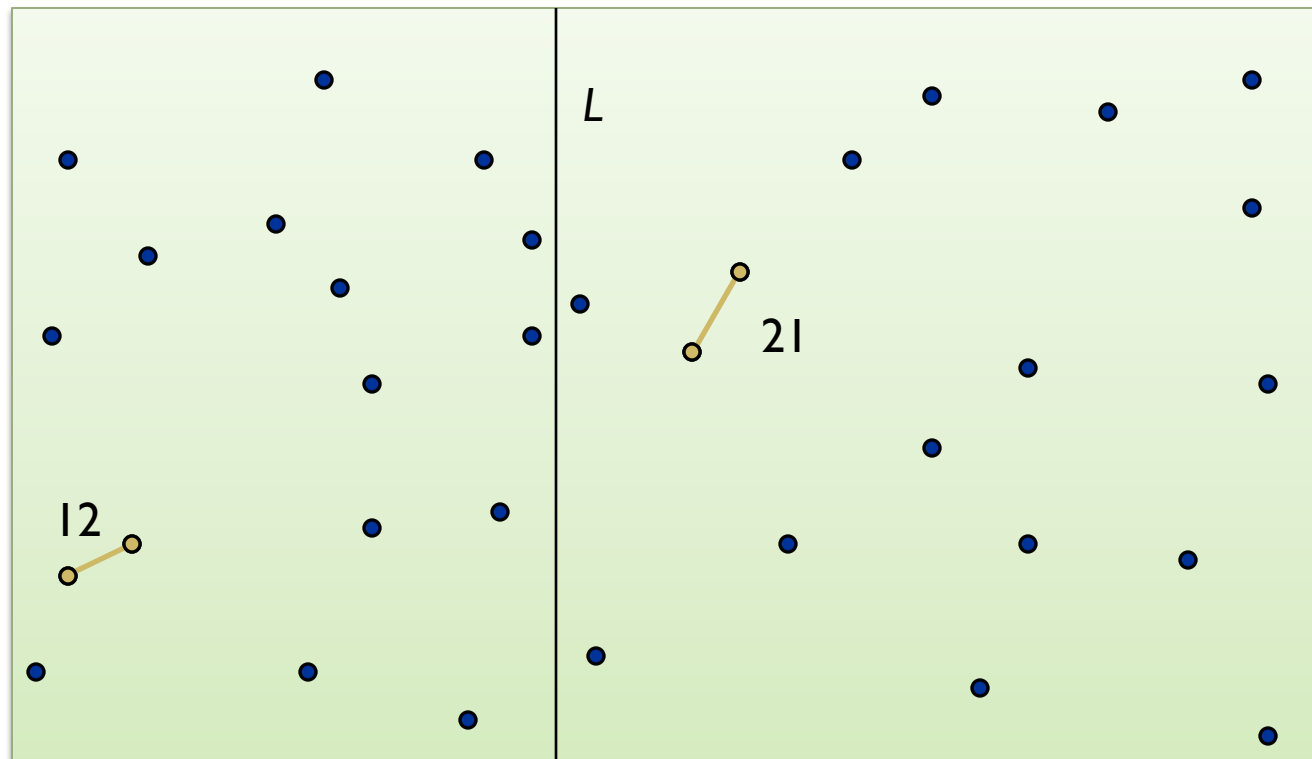
Divide: draw vertical line L with $\approx n/2$ points on each side.



Algorithm.

Divide: draw vertical line L with $\approx n/2$ points on each side.

Conquer: find closest pair on each side, recursively.



Algorithm.

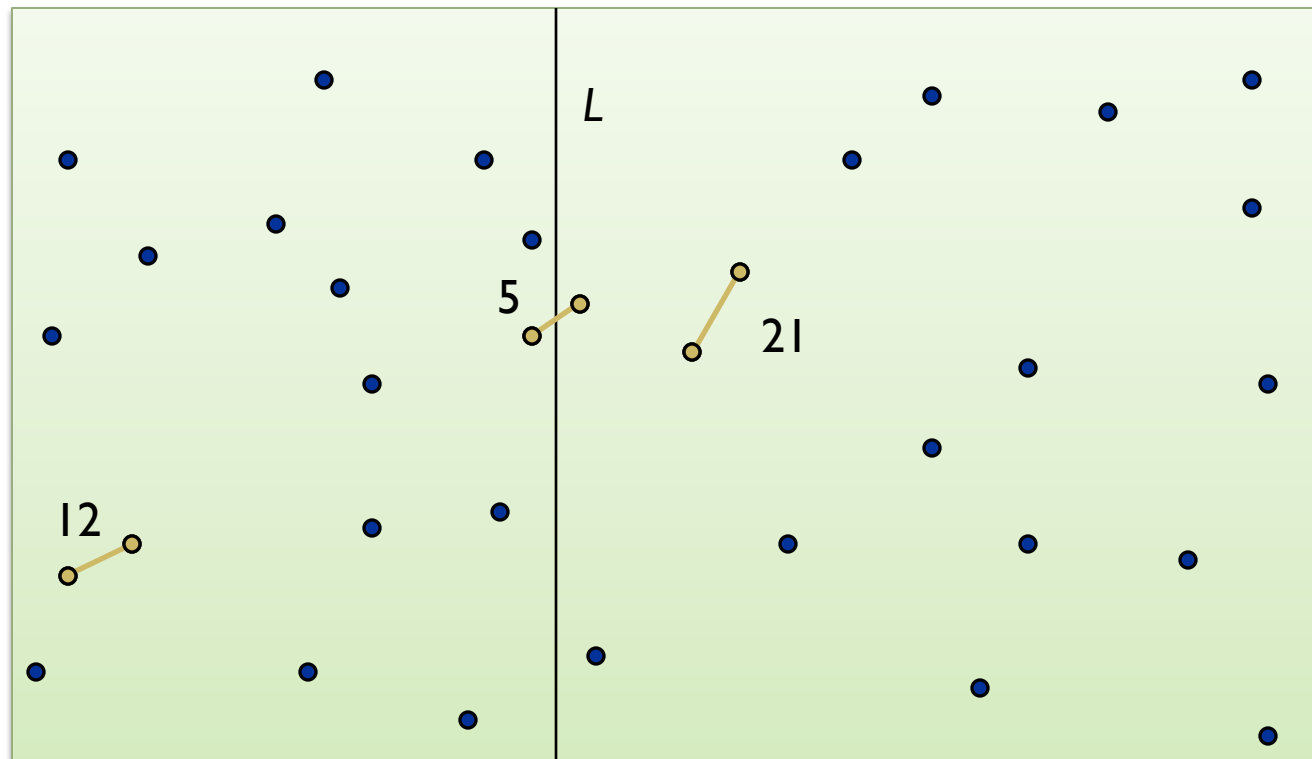
Divide: draw vertical line L with $\approx n/2$ points on each side.

Conquer: find closest pair on each side, recursively.

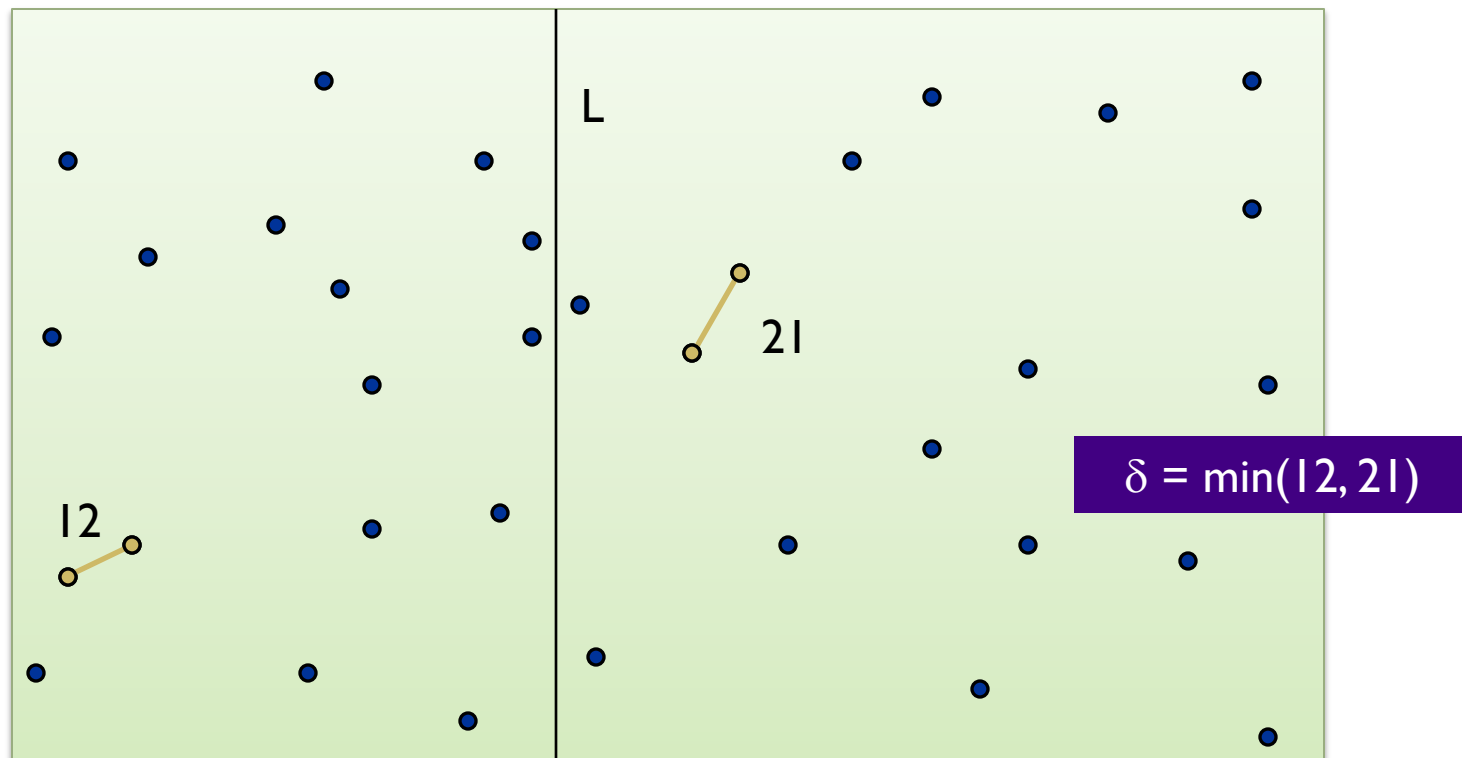
Combine: find closest pair with one point in each side.

Return best of 3 solutions.

←
seems
like
 $\Theta(n^2)$?

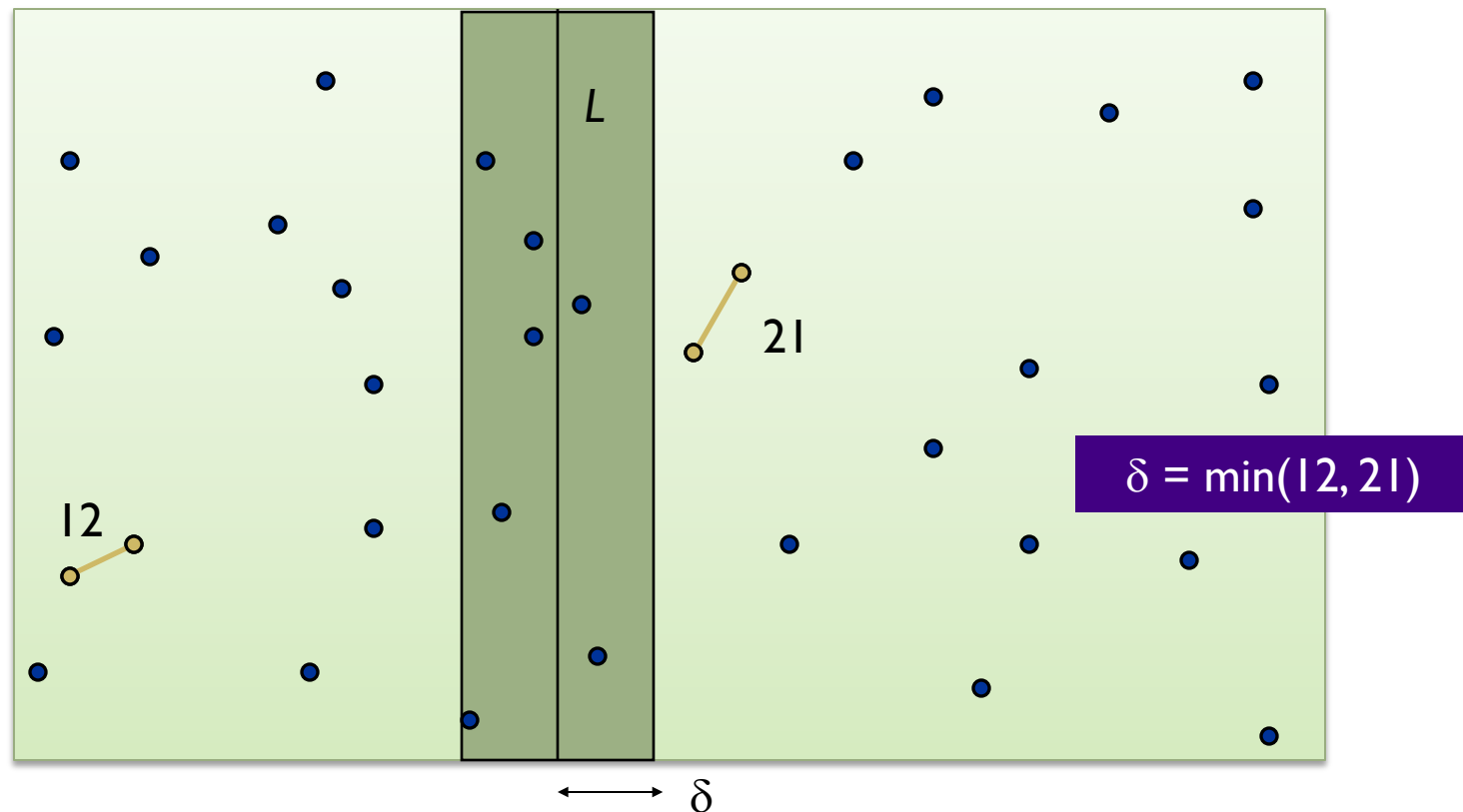


Find closest pair with one point in each side,
assuming distance $< \delta$.



Find closest pair with one point in each side,
assuming distance $< \delta$.

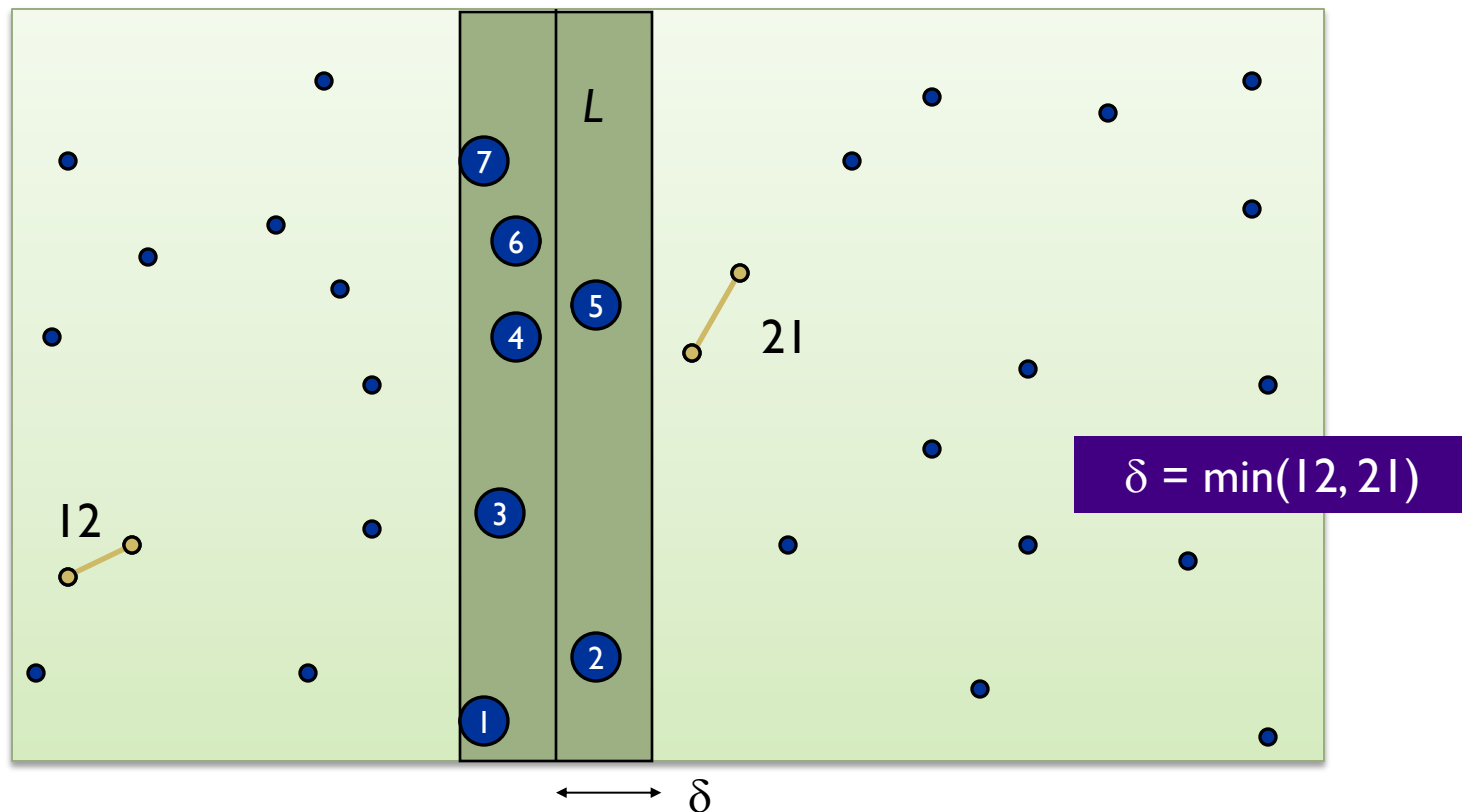
Observation: suffices to consider points within δ of line L .



Find closest pair with one point in each side, assuming distance $< \delta$.

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

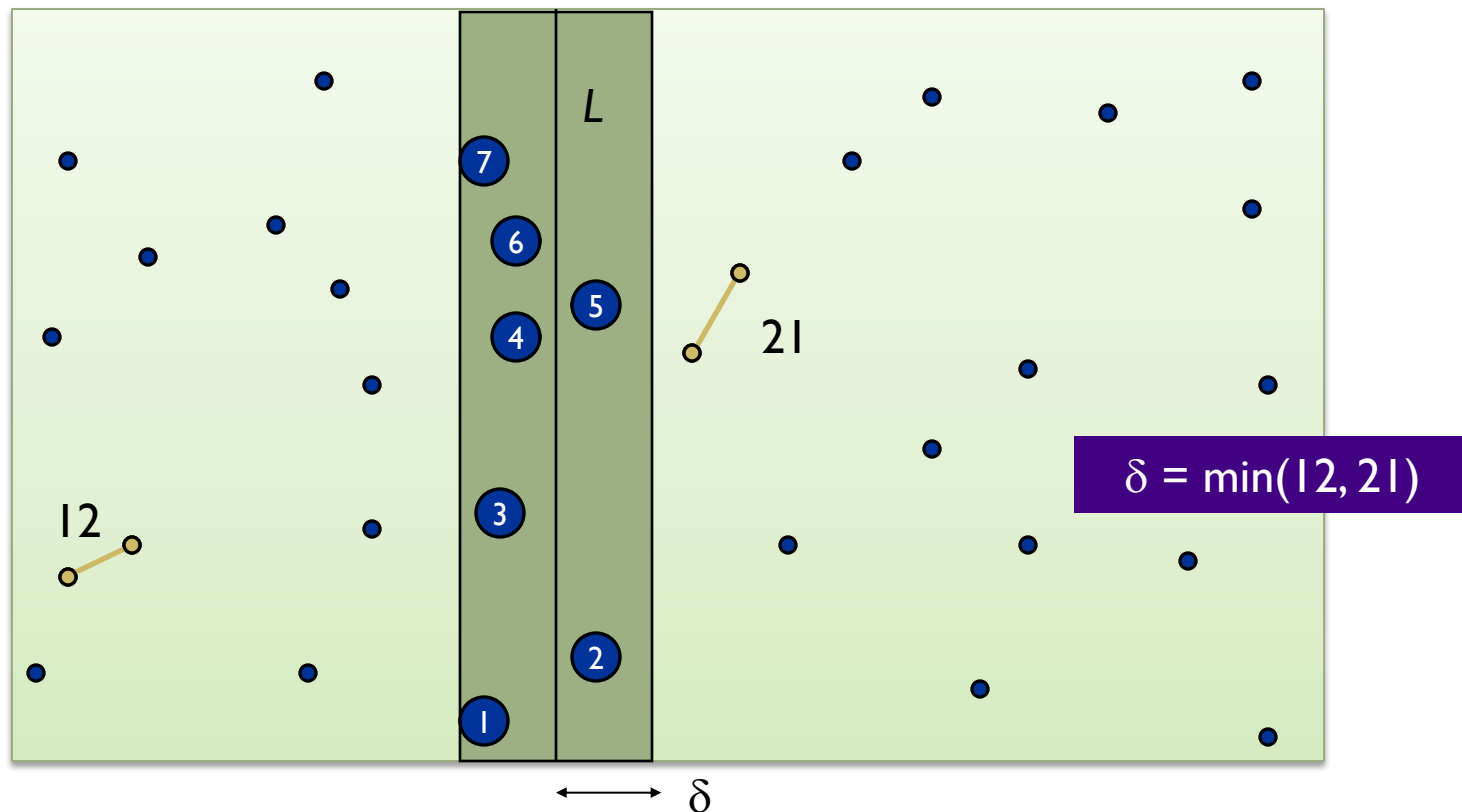
“Almost” the one-D problem again: Sort points in 2δ -strip by their y coordinate.



Find closest pair with one point in each side, assuming distance $< \delta$.

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!



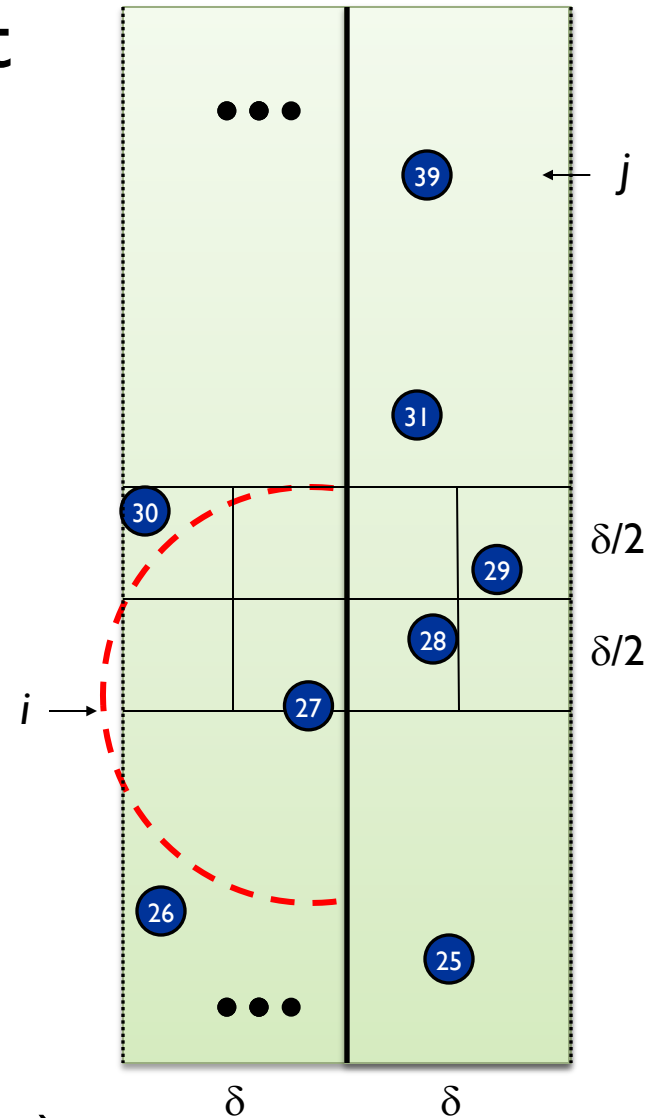
Def. Let s_i have the i^{th} smallest y -coordinate among points in the 2δ -width-strip.

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

Pf: No two points lie in the same $\delta/2$ -by- $\delta/2$ square:

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

so ≤ 7 points within $+\delta$ of $y(s_i)$.



closest pair algorithm

```
Closest-Pair( $p_1, \dots, p_n$ ) {  
    if( $n \leq ??$ ) return ??  
  
    Compute separation line  $L$  such that half the points  
    are on one side and half on the other side.  
  
     $\delta_1$  = Closest-Pair(left half)  
     $\delta_2$  = Closest-Pair(right half)  
     $\delta$  = min( $\delta_1, \delta_2$ )  
  
    Delete all points further than  $\delta$  from separation line  $L$   
  
    Sort remaining points  $p[1] \dots p[m]$  by  $y$ -coordinate.  
  
    for  $i = 1..m$   
         $k = 1$   
        while  $i+k \leq m \ \&\& \ p[i+k].y < p[i].y + \delta$   
             $\delta = \min(\delta, \text{distance between } p[i] \text{ and } p[i+k]);$   
             $k++$ ;  
  
    return  $\delta$ .  
}
```


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

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

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

What if we counted comparisons?

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

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

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.

$$T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$$

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

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 $n \geq 1$ ”

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

Write Recurrence(s)

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

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

Base case

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.)

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 $n \geq 1$ 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( $p_1, \dots, p_n$ ) {  
  if( $n \leq 1$ ) return  $\infty$ 
```

Base Case

0

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

```
 $\delta_1$  = Closest-Pair(left half)  
 $\delta_2$  = Closest-Pair(right half)  
 $\delta$  = min( $\delta_1, \delta_2$ )
```

Recursive calls (2)

$2D(n / 2)$

Delete all points further than δ from separation line L

Sort remaining points $p[1] \dots p[m]$ by y-coordinate.

```
for  $i = 1 \dots m$   
   $k = 1$ 
```

```
    while  $i+k \leq m$  &&  $p[i+k].y < p[i].y + \delta$   
       $\delta = \min(\delta, \text{distance between } p[i] \text{ and } p[i+k]);$   
       $k++;$ 
```

Basic operations at
this recursive level

One
recursive
level

$7n$

```
  return  $\delta$ .  
}
```

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

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

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

What if we counted comparisons?

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 $n \geq 1$ 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

Base Case

```
Closest-Pair( $p_1, \dots, p_n$ ) {  
  if ( $n \leq 1$ ) return  $\infty$ 
```

Recursive calls (2)

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

```
     $\delta_1 = \text{Closest-Pair}(\text{left half})$   
     $\delta_2 = \text{Closest-Pair}(\text{right half})$   
     $\delta = \min(\delta_1, \delta_2)$ 
```

```
    Delete all points further than  $\delta$  from separation line  $L$ 
```

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

```
    for  $i = 1..m$   
       $k = 1$ 
```

```
      while  $i+k \leq m \ \&\& \ p[i+k].y < p[i].y + \delta$   
         $\delta = \min(\delta, \text{distance between } p[i] \text{ and } p[i+k]);$   
         $k++;$ 
```

```
    return  $\delta$ .
```

```
}
```

Basic operations at
this recursive level

0

$k_1 n \log n$

$2C(n/2)$

1

$k_2 n$

$k_3 n \log n$

$8n$

$7n$

One
recursive
level

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

$$C(n) \leq \begin{cases} 0 & n = 1 \\ 2C(n/2) + k_4 n \log_2 n & n > 1 \end{cases} \Rightarrow C(n) = O(n \log^2 n)$$

for $k_4 = k_1 + k_2 + k_3 + 16$

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.

$$T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$$

Integer Multiplication

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	1	0	1
<hr/>									
	1	0	1	0	1	0	0	1	0

$O(n)$ bit operations.

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

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

Multiply. Given two n -bit integers a and b , compute $a \times b$.
The “grade school” method:

 $\Theta(n^2)$ bit operations.[illegible]

divide & conquer multiplication: warmup

To multiply two 2-digit integers:

Multiply four 1-digit integers.

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

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

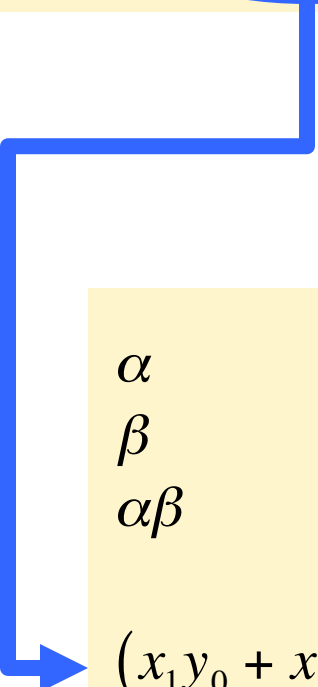
Same idea works for *long* integers –
can split them into 4 half-sized ints
("10" becomes "10^k", $k = \text{length}/2$)

4	5	$y_1 y_0$	
3	2	$x_1 x_0$	
<hr/>			
1	0	$x_0 y_0$	
0	8	$x_0 y_1$	
1	5	$x_1 y_0$	
1	2	$x_1 y_1$	
<hr/>			
1	4	4	0

key trick: 2 multiplies for the price of 1:

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

Well, ok, 4 for 3 is more accurate...


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

Karatsuba multiplication

To multiply two n -bit integers:

Add two pairs of $\frac{1}{2}n$ bit integers.

Multiply *three* pairs of $\frac{1}{2}n$ -bit integers.

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

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

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

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

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

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

Karatsuba multiplication

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

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

Best to solve it directly (but messy). Instead, it nearly always suffices to solve a simpler recurrence:

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

Intuition: If $T(n) = n^k$, then $T(n+1) = n^k + kn^{k-1} + \dots = O(n^k)$

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

(Proof later.)

Naïve: $\Theta(n^2)$

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

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

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, among other uses

Recurrences

Above: Where they come
from, how to find them

Next: how to solve them

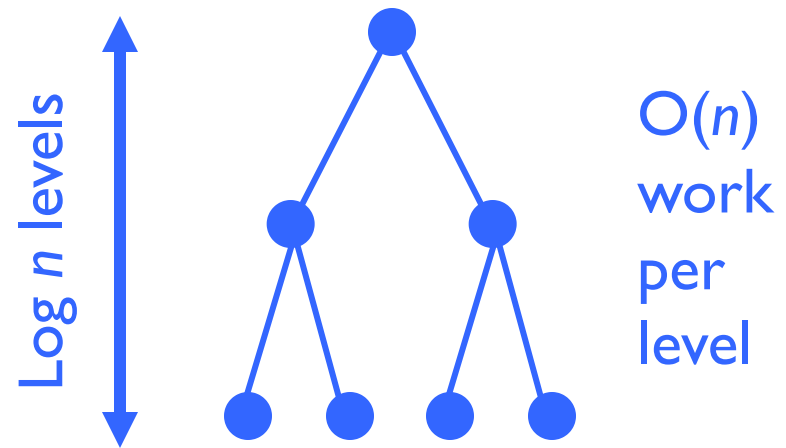
Mergesort: (recursively) sort 2 half-lists, then merge results.

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

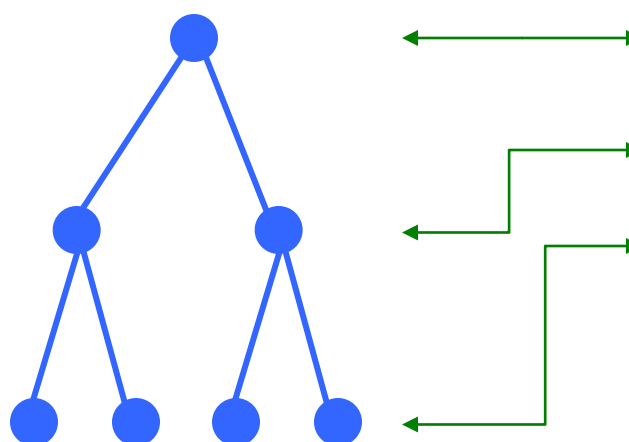
$$T(1) = 0$$

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

now!



Solve: $T(1) = c$
 $T(n) = 2 T(n/2) + cn$



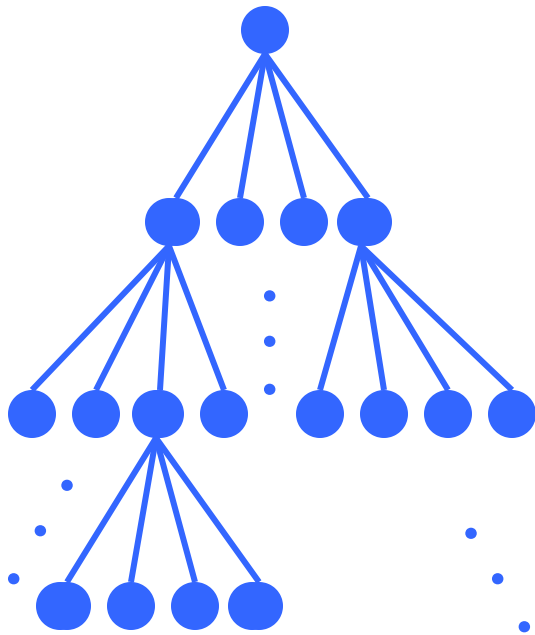
Level	Num	Size	Work
0	$1 = 2^0$	n	cn
1	$2 = 2^1$	$n/2$	$2cn/2$
2	$4 = 2^2$	$n/4$	$4cn/4$
...
i	2^i	$n/2^i$	$2^i c n/2^i$
...
$k-1$	2^{k-1}	$n/2^{k-1}$	$2^{k-1} c n/2^{k-1}$
k	2^k	$n/2^k = 1$	$2^k T(1)$

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

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

(add last col)

Solve: $T(1) = c$
 $T(n) = 4 T(n/2) + cn$



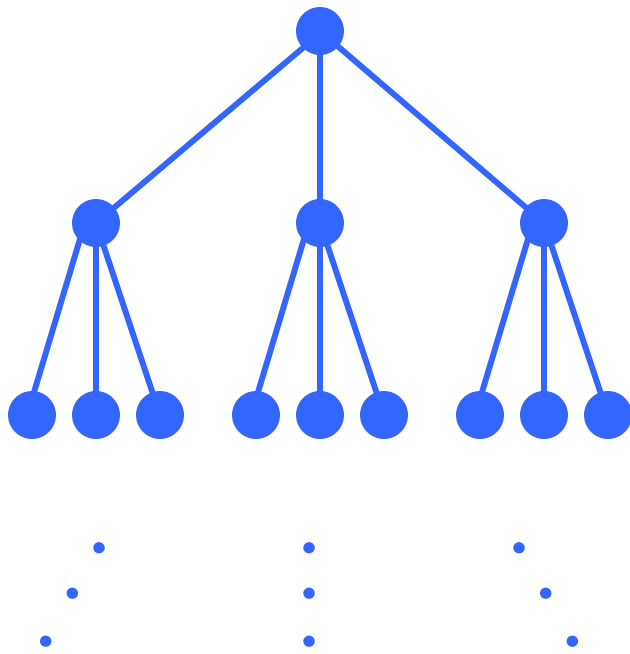
$$n = 2^k ; k = \log_2 n$$

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	$n/2^k = 1$	$4^k T(1)$

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: $T(1) = c$
 $T(n) = 3 T(n/2) + cn$



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

Total Work: $T(n) =$

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	$n/2^k = 1$	$3^k T(1)$

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



Theorem: for $x \neq 1$,

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

proof:

$$y = 1 + x + x^2 + x^3 + \dots + x^k$$

$$xy = x + x^2 + x^3 + \dots + x^k + x^{k+1}$$

$$xy - y = x^{k+1} - 1$$

$$y(x - 1) = x^{k+1} - 1$$

$$y = (x^{k+1} - 1)/(x - 1)$$

Solve: $T(1) = c$

$$T(n) = 3 T(n/2) + cn \quad (\text{cont.})$$

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

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

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

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



$$\begin{aligned} \sum_{i=0}^k x^i &= \\ \frac{x^{k+1} - 1}{x - 1} \\ (x \neq 1) \end{aligned}$$

Solve: $T(1) = c$

$$T(n) = 3 T(n/2) + cn \quad (\text{cont.})$$

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

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

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

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

Solve: $T(1) = c$

$$T(n) = 3 T(n/2) + cn \quad (\text{cont.})$$

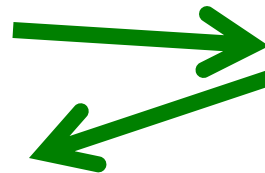
$$3cn \frac{3^k}{2^k} = 3cn \frac{3^{\log_2 n}}{2^{\log_2 n}}$$

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

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

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

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



$$\begin{aligned} & a^{\log_b n} \\ &= \left(b^{\log_b a} \right)^{\log_b n} \\ &= \left(b^{\log_b n} \right)^{\log_b a} \\ &= n^{\log_b a} \end{aligned}$$

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 \rightarrow leaves dominate]

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

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

Fine print:

$T(1) = d; a \geq 1; b > 1; c, d, k \geq 0; n = b^t$ 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

Expand recurrence as in earlier examples, to get

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

where $h = \log_b(a)$ (and n^h = number of tree leaves) 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 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 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 $S = x \cdot (x^{1+\log_b(n)} - 1)/(x-1)$. [And after some algebra, $n^h * S = O(n^k)$.]

Another Example: 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 $a^n = a^m \bullet a^m$

Power(a, n)

if $n = 0$ then return(1)

if $n = 1$ then return(a)

$x \leftarrow \text{Power}(a, \lfloor n/2 \rfloor)$

$x \leftarrow x \bullet x$

if n is odd then

$x \leftarrow a \bullet x$

return(x)

Let $M(n)$ be number of multiplies

Worst-case
recurrence:
$$M(n) = \begin{cases} 0 & n \leq 1 \\ M(\lfloor n/2 \rfloor) + 2 & n > 1 \end{cases}$$

By master theorem

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

More precise analysis:

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

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

Instead of a^n want $a^n \bmod N$

$$a^{i+j} \bmod N = ((a^i \bmod N) \cdot (a^j \bmod N)) \bmod N$$

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 $a^n \bmod N$ 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

Utility:

Correctness often easy; often faster

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

among others

Applications: Many.

Binary Search, Merge Sort, (Quicksort), Closest Points, Integer Multiply, Exponentiation,...