CSE 421
Algorithms:
Divide and Conquer

Larry Ruzzo

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

    Closest points

    Integer Multiplication

Finding & Solving Recurrences

Divide & Conquer

Reduce problem to one or more 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:

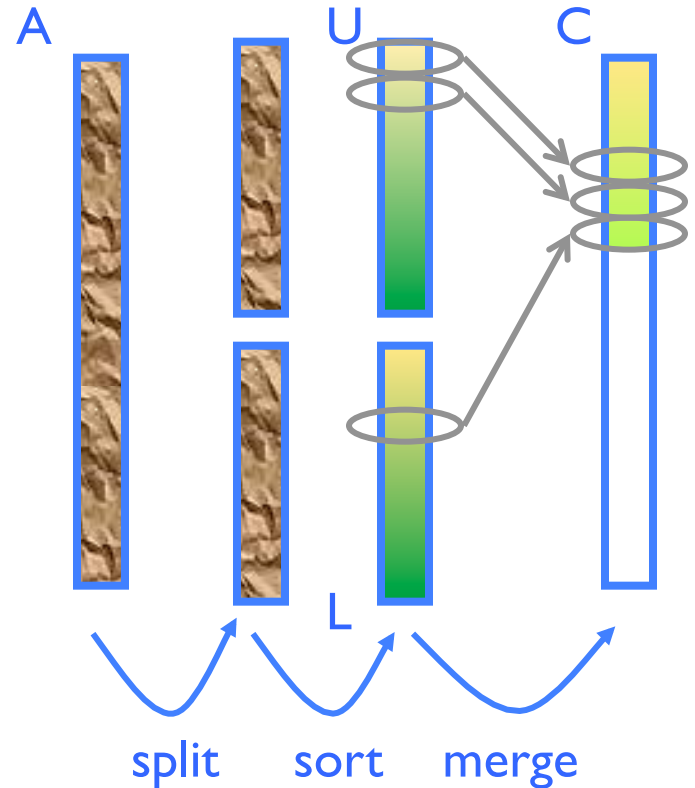Binary Search, Mergesort, Quicksort (roughly), Strassen's Algorithm, …

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++";
    Return C;
    }

A U C

L

split   sort   merge

6

Alternative "divide & conquer" algorithm:

Sort n-1

Sort last 1

Merge them

$T(n) = T(n-1)+T(1)+3n$ for $n \geq 2$

$T(1) = 0$

Solution: $3n + 3(n-1) + 3(n-2) \ldots = \Theta(n^2)$

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

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.
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 less good:

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

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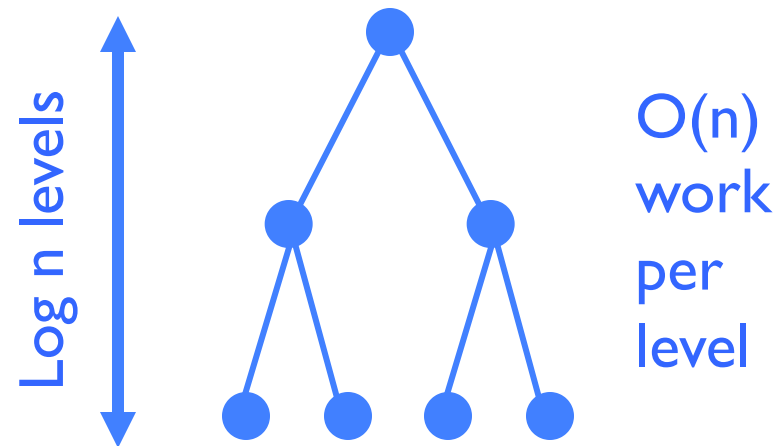
$(1)^2 + (n-1)^2 + n = n^2 - 2n + 2 + n$

> Little improvement here.

Mergesort: (recursively) sort 2 half-lists, then merge results.
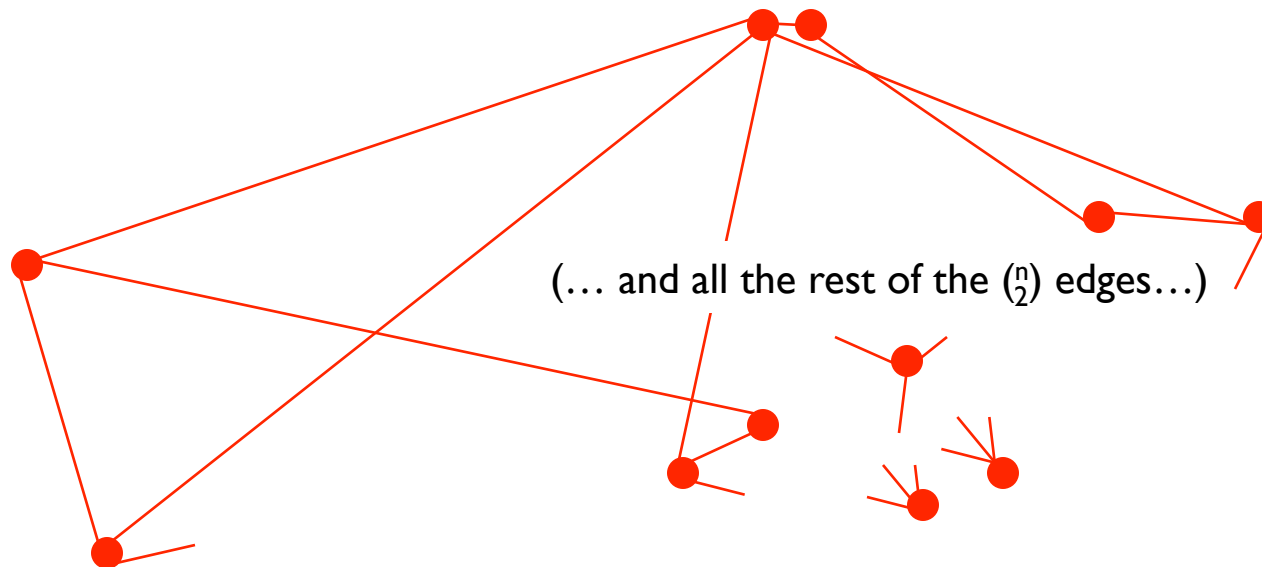
$T(n) = 2T(n/2)+cn, \quad n \geq 2$

$T(1) = 0$

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

Log n levels

O(n) work per level

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!)
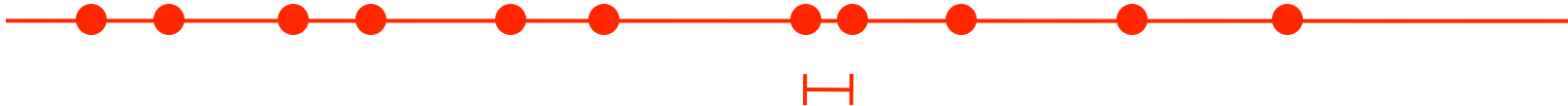

(… and all the rest of the $\binom{n}{2}$ edges…)

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?

13

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. 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

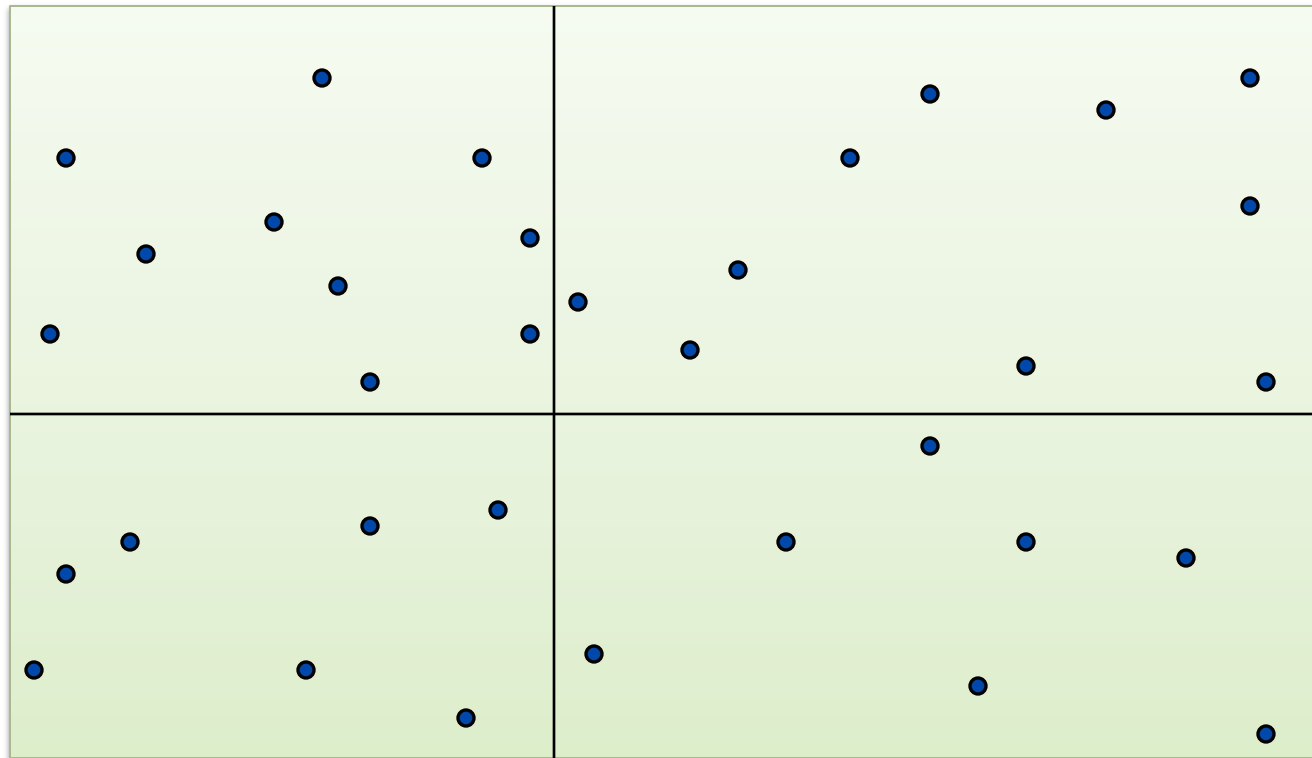Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

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

Assumption. No two points have same x coordinate.

Just to simplify presentation
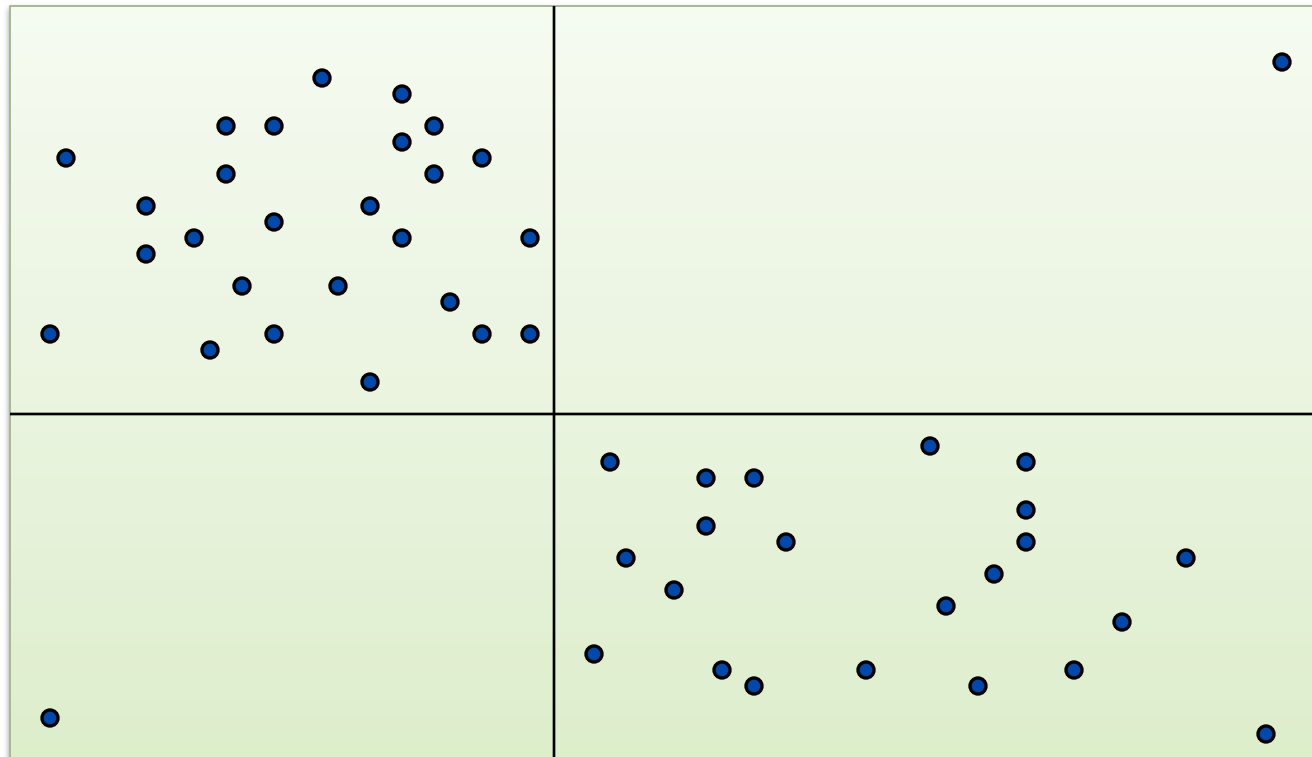
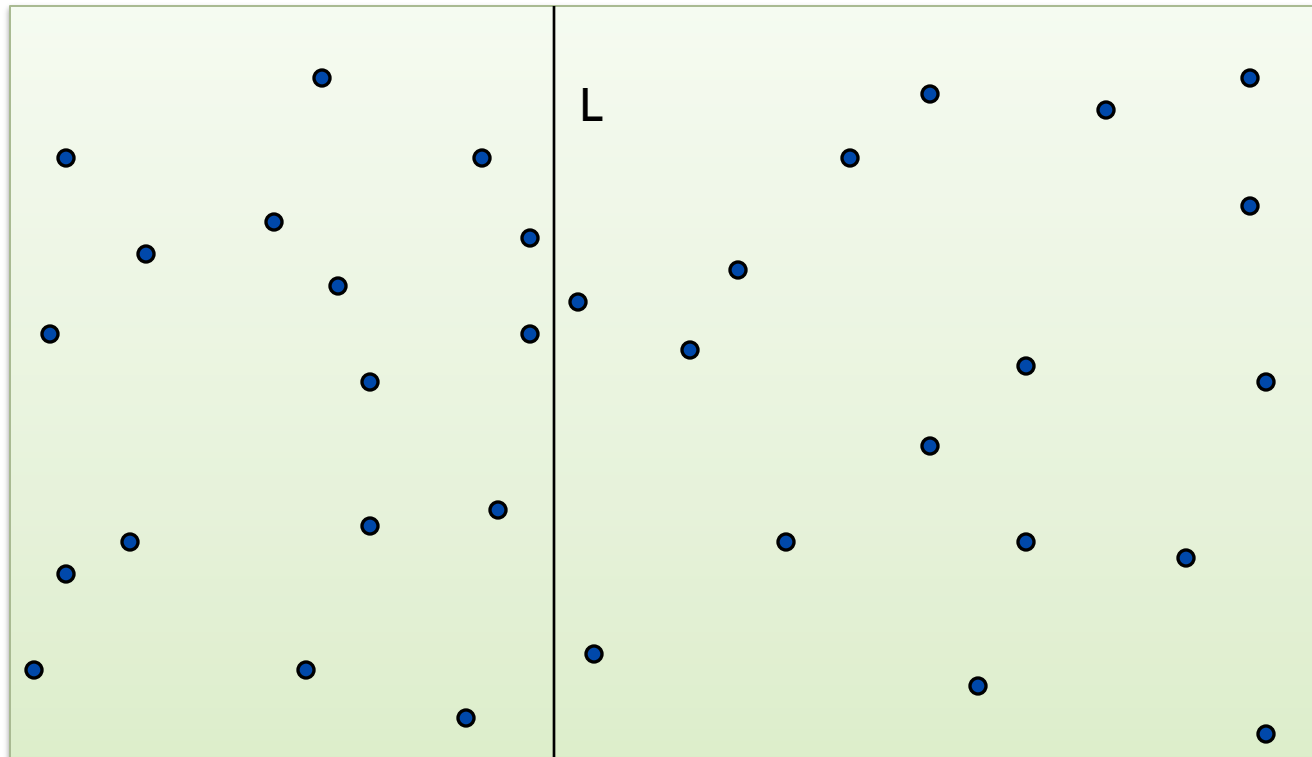# Divide. Sub-divide region into 4 quadrants.

16

Divide. Sub-divide region into 4 quadrants.

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

Algorithm.

Divide: draw vertical line L with $\approx$ 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.
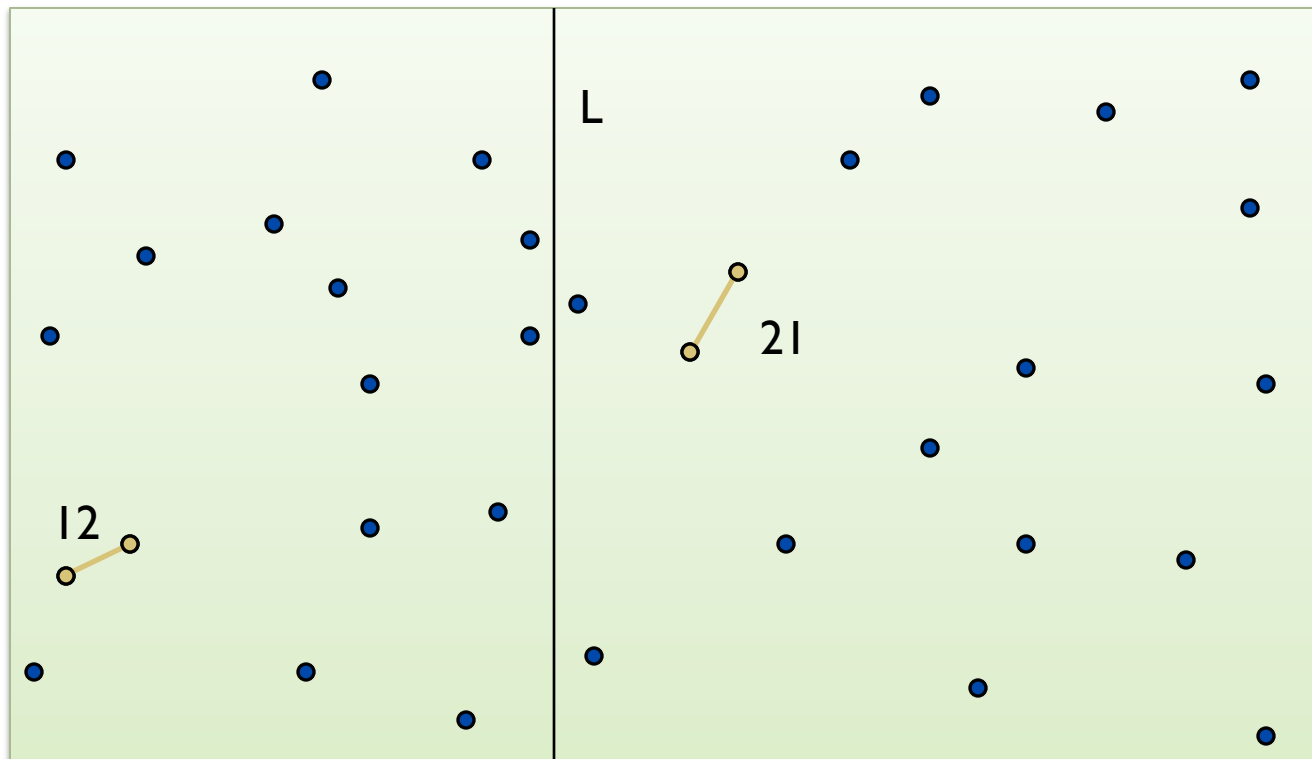
Algorithm.

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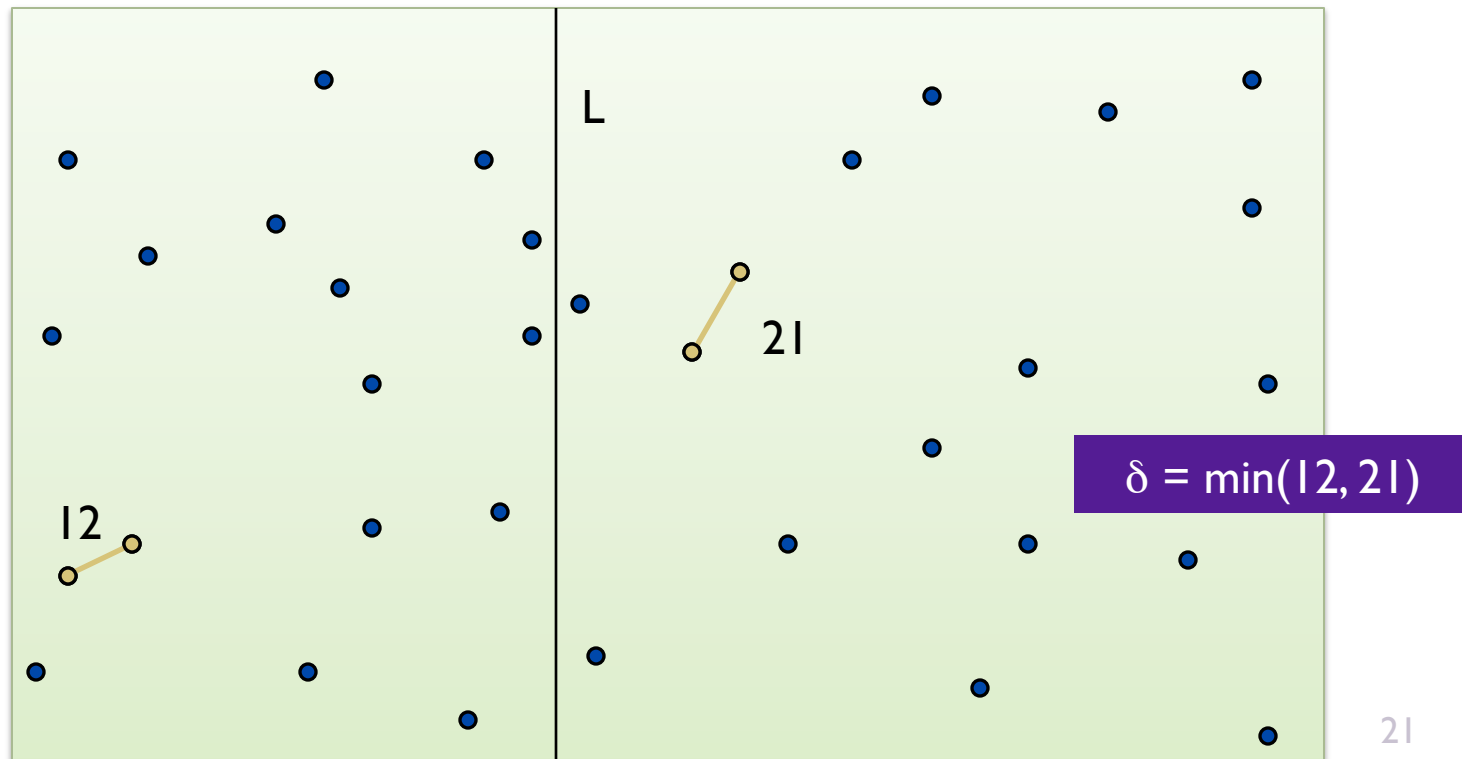
Find closest pair with one point in each side, *assuming* distance < δ.

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

L

21

δ = min(12, 21)

12

δ

22

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.

L

21

12

δ = min(12, 21)

δ

23

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!

L

7

6

5

4

21

3

$\delta = \min(12, 21)$

12

2

1

δ

24
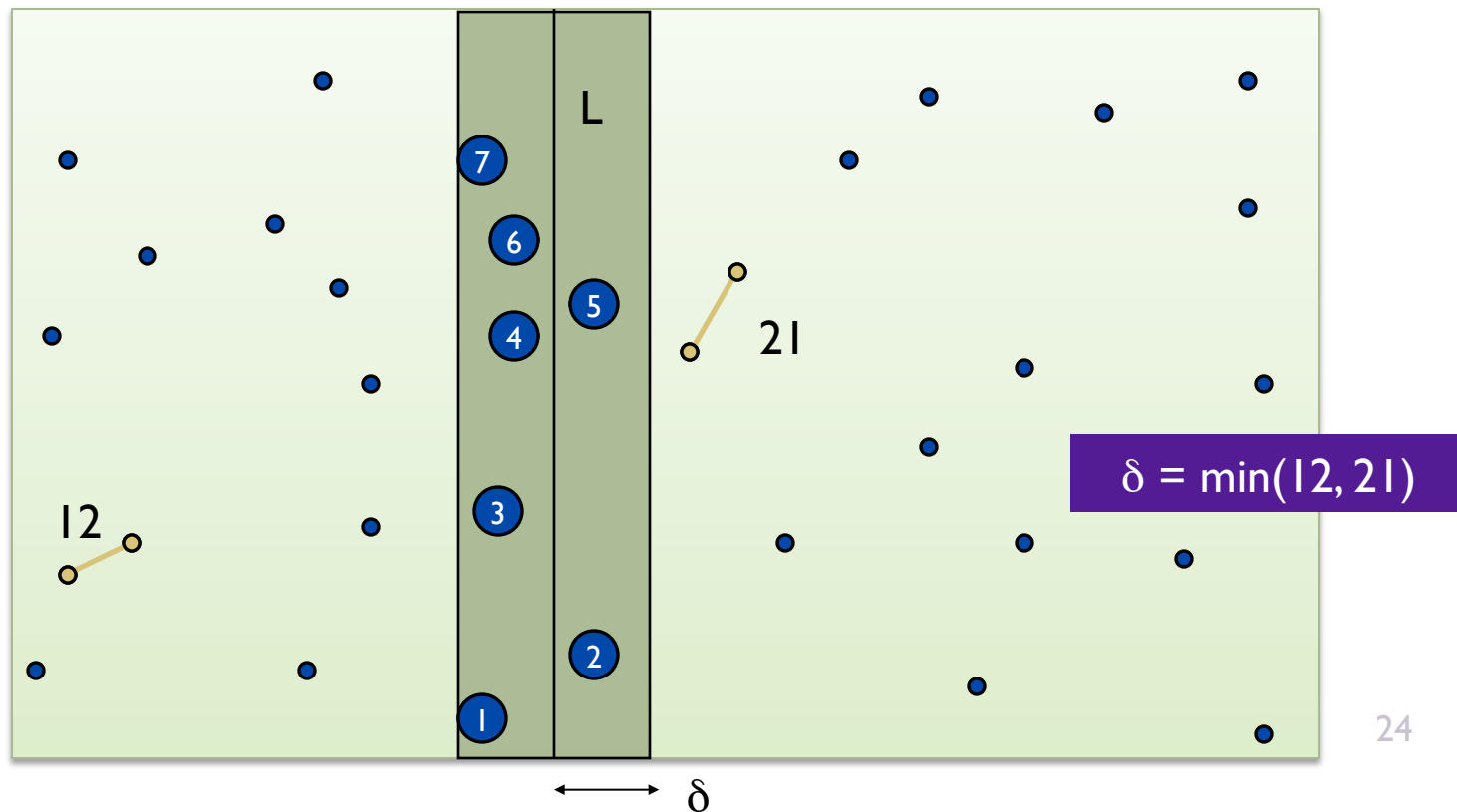
Def. Let s_i have the ith smallest y-coordinate among points in the 2δ-width-strip.

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

Pf: No two points lie in the same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ 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 $+\delta$ of $y(s_i)$.

25
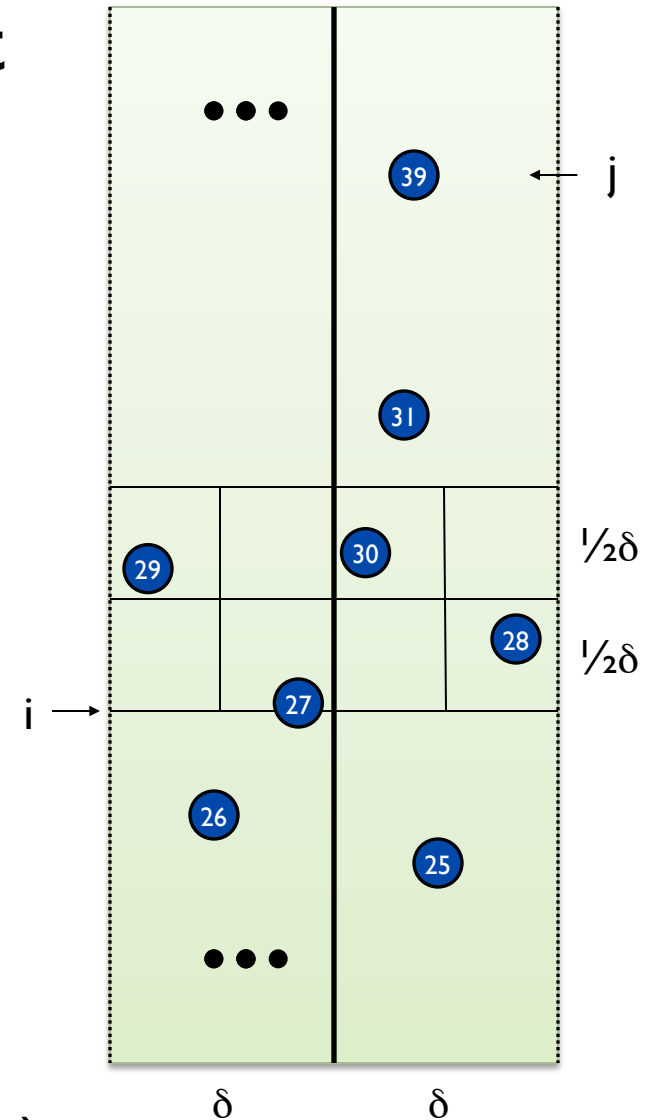
```
Closest-Pair(p₁, …, pₙ) {
   if(n <= ??) return ??

   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.
}
```

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

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

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 ≥ 1 points

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

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.

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

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 ≥ 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

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.)

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

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

Base Case

0

```
    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)
```

Recursive calls (2)

$2D(n / 2)$

```
    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.
}
```

One
recursive
level

Basic operations at
this recursive level

$O(n)$

36

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

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

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

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

Base Case

Recursive calls (2)

0

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

O(n log n)

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

2C(n / 2)

1

```
    Delete all points further than δ from separation line L
```

O(n)

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

O(n log n)

Basic operations at this recursive level

```
    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++;
```

O(n)

```
    return δ.
}
```

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 ≥ 1 points

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

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.

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

40

Integer Multiplication

Add. Given two n-bit 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
+		0	1	1	1	1	1	0	1
1	0	1	0	1	0	0	1	0

Add

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

O(n) bit operations.

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

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

Add

	I	I	I	I	I	I	0	I
	I	I	0	I	0	I	0	I
+ 0	I	I	I	I	I	0	I
I	0	I	0	I	0	0	I	0

Multiply

			I	I	0	I	0	I	0	I
		* 0	I	I	I	I	I	0	I
			I	I	0	I	0	I	0	I
		0	0	0	0	0	0	0	0
	I	I	0	I	0	I	0	I
I	I	0	I	0	I	0	I
I	I	0	I	0	I	0	I
I	I	0	I	0	I	0	I
I	I	0	I	0	I	0	I
0	0	0	0	0	0	0	0
0	I	I	0	I	0	0	0	0	0	0	0	0	0	0	I

To multiply two 2-digit integers:

Multiply four 1-digit integers.

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

$$x \quad = \quad 10 \cdot x_1 \; + \; x_0$$
$$y \quad = \quad 10 \cdot y_1 \; + \; y_0$$
$$xy \quad = \quad \left(10 \cdot x_1 + x_0\right)\left(10 \cdot y_1 + y_0\right)$$
$$= \quad 100 \cdot x_1 y_1 \; + \; 10 \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0$$

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

	4	5	$y_1 \; y_0$
	3	2	$x_1 \; x_0$
	1	0	$x_0 \cdot y_0$
0	8		$x_0 \cdot y_1$
1	5		$x_1 \cdot y_0$
1	2		$x_1 \cdot y_1$
1	4	4	0

To multiply two n-bit integers:

Multiply four ½n-bit integers.

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

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

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \Rightarrow T(n) = \Theta(n^2)$$
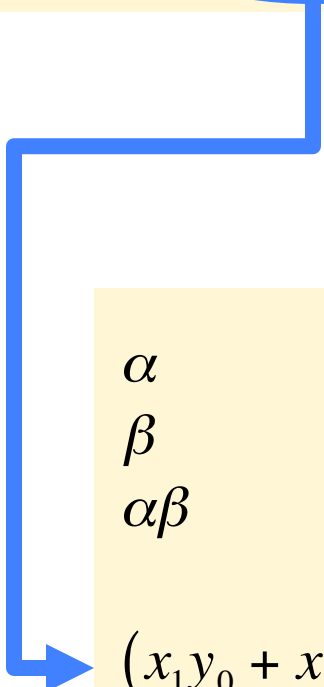
assumes n is a power of 2
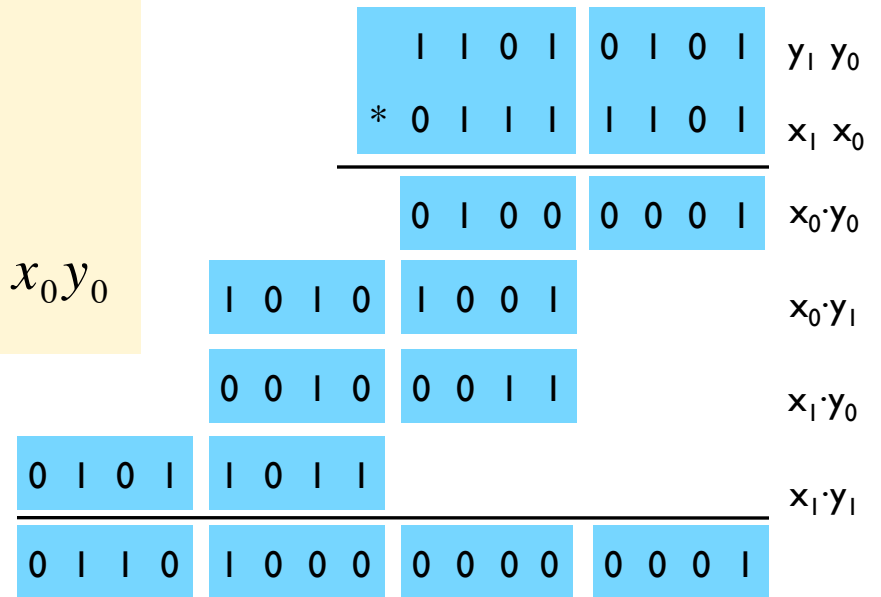
1 1 0 1	0 1 0 1	$y_1 \; y_0$
* 0 1 1 1	1 1 0 1	$x_1 \; x_0$
0 1 0 0	0 0 0 1	$x_0 \cdot y_0$

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

$$x = 2^{n/2} \cdot x_1 + x_0$$

$$y = 2^{n/2} \cdot y_1 + y_0$$

$$xy = \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right)$$

$$= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0$$

Well, ok, 4 for 3 is more accurate…

$$\alpha = x_1 + x_0$$

$$\beta = y_1 + y_0$$

$$\alpha\beta = \left(x_1 + x_0\right)\left(y_1 + y_0\right)$$

$$= x_1 y_1 + \left(x_1 y_0 + x_0 y_1\right) + x_0 y_0$$

$$\left(x_1 y_0 + x_0 y_1\right) = \alpha\beta - x_1 y_1 - x_0 y_0$$

To multiply two n-bit integers:

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

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

Add, subtract, and shift $\frac{1}{2}n$-bit integers to obtain result.

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

<div align="center">A   B   A   C   C</div>

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

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

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

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

47

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

48

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,…

Recurrences

Above: Where they come
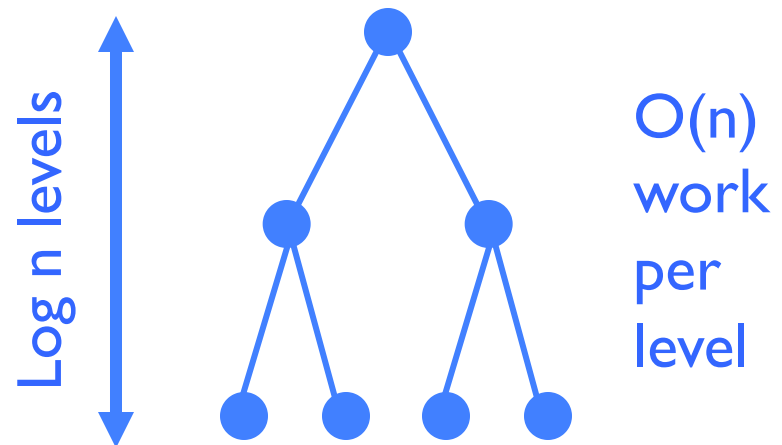from, how to find them

Next: how to solve them

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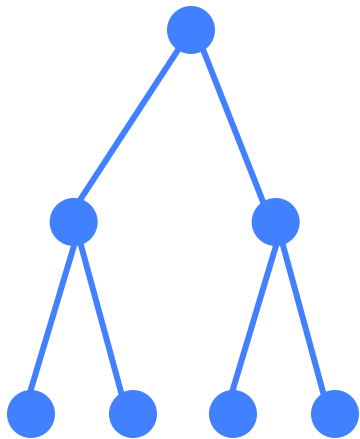merge results.

$T(n) = 2T(n/2)+cn, \quad n\geq2$

$T(1) = 0$

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

now

Log n levels

O(n)
work
per
level

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)

52

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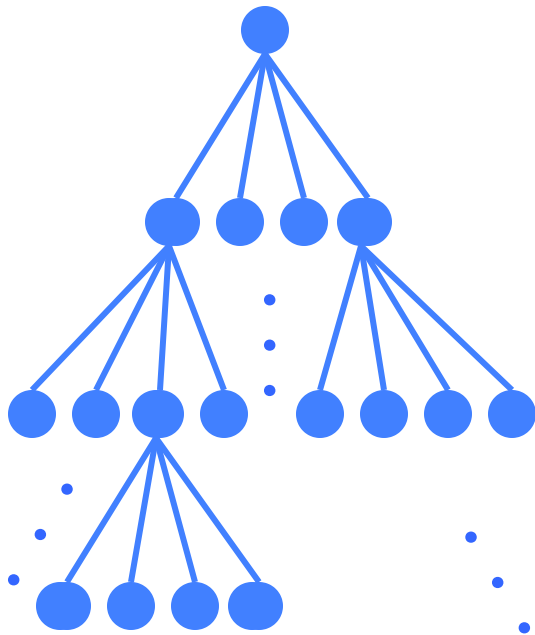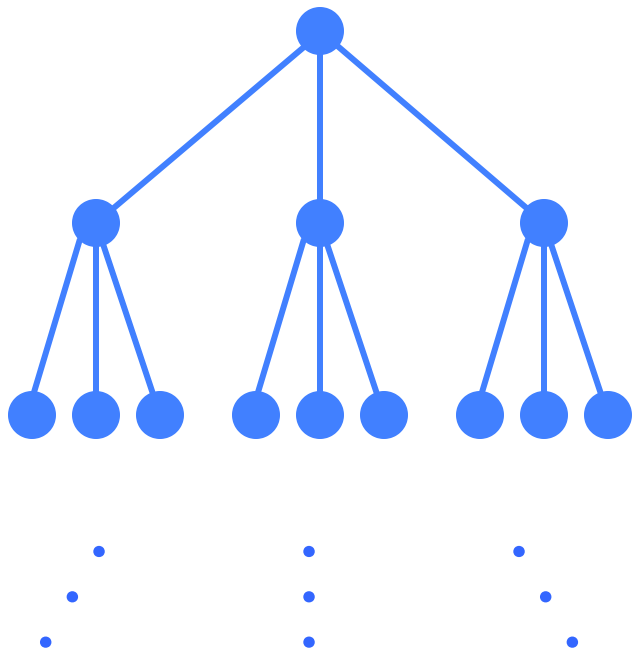

Level	Num	Size	Work
0	$1 = 4^0$	n	cn
1	$4 = 4^1$	$n/2$	$4cn/2$
2	$16 = 4^2$	$n/4$	$16cn/4$
...	...	...	...
i	4^i	$n/2^i$	$4^i\,c\,n/2^i$
...	...	...	...
$k-1$	4^{k-1}	$n/2^{k-1}$	$4^{k-1}\,c\,n/2^{k-1}$
k	4^k	$n/2^k = 1$	$4^k\,T(1)$

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

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

$4^k = (2^2)^k = (2^k)^2 = n^2$
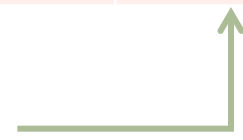
Solve: $T(1) = c$
       $T(n) = 3\,T(n/2) + cn$

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

54

Theorem:

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

proof:

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

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

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

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

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

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

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

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

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

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

$$\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1}$$

$$(x \neq 1)$$

56

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

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

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

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

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

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

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

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

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

$$\boxed{= O\left(n^{1.59\ldots}\right)}$$

$$a^{\log_b n}$$

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

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

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

$T(n) = aT(n/b) + cn^k$ 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.

Expanding recurrence as in earlier examples, to get

$$T(n) = n^g \, (\, d + c \, S \,)$$

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

If $c = 0$ the sum S is irrelevant, and $T(n) = O(n^g)$: all the work happens in the base cases, of which there are n^g, 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 $S = x * (x^{1+\log_b(n)}-1)/(x-1)$. After some algebra,
$n^g * S = O(n^k)$

Power(a,n)

Input: integer n and number a

Output: a^n

Obvious algorithm

$n\text{-}1$ 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)
   $x \leftarrow$ 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)   (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

Instead of a^n want a^n mod N

a^{i+j} mod N = $((a^i$ mod $N)$ • $(a^j$ mod $N))$ mod N

same algorithm applies with each x • y replaced by

$((x$ mod $N)$ • $(y$ mod $N))$ mod 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

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), Closest points, Integer multiply, exponentiation,…