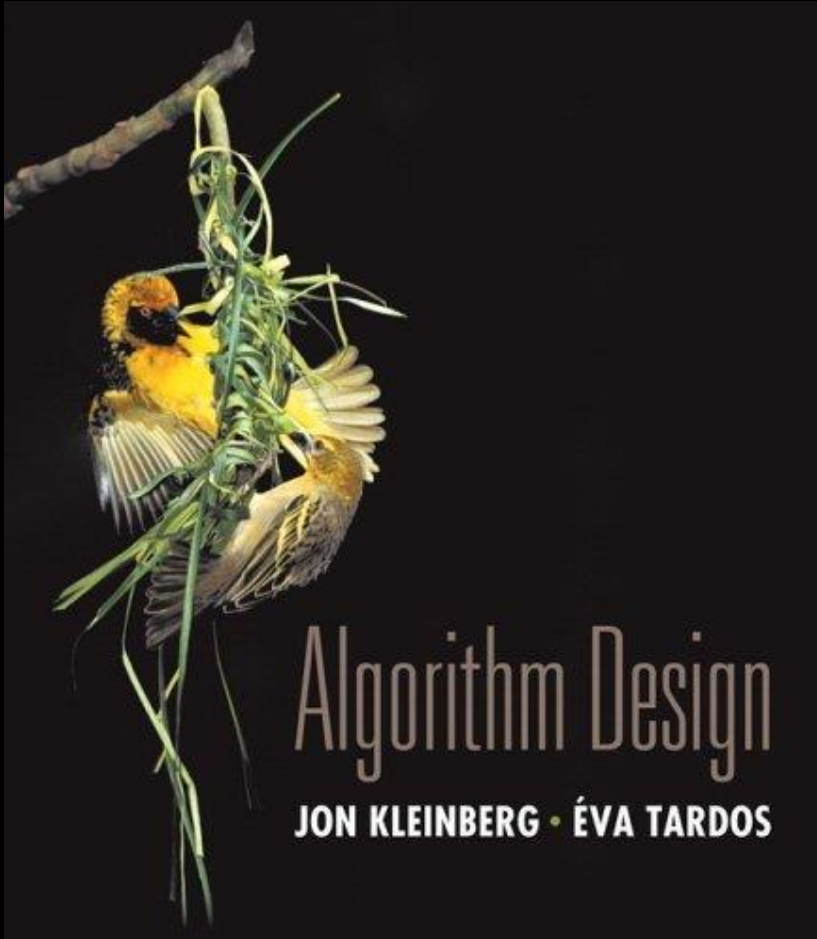


Chapter 5

Divide and Conquer



Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.

- Break up problem of size n into **two** equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively.
- Combine two solutions into overall solution in **linear time**.

Consequence.

- Brute force: n^2 .
- Divide-and-conquer: $n \log n$.

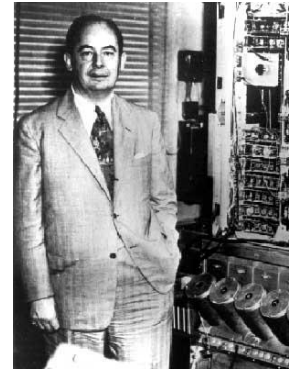
Divide et impera.
Veni, vidi, vici.
- *Julius Caesar*

5.1 Mergesort

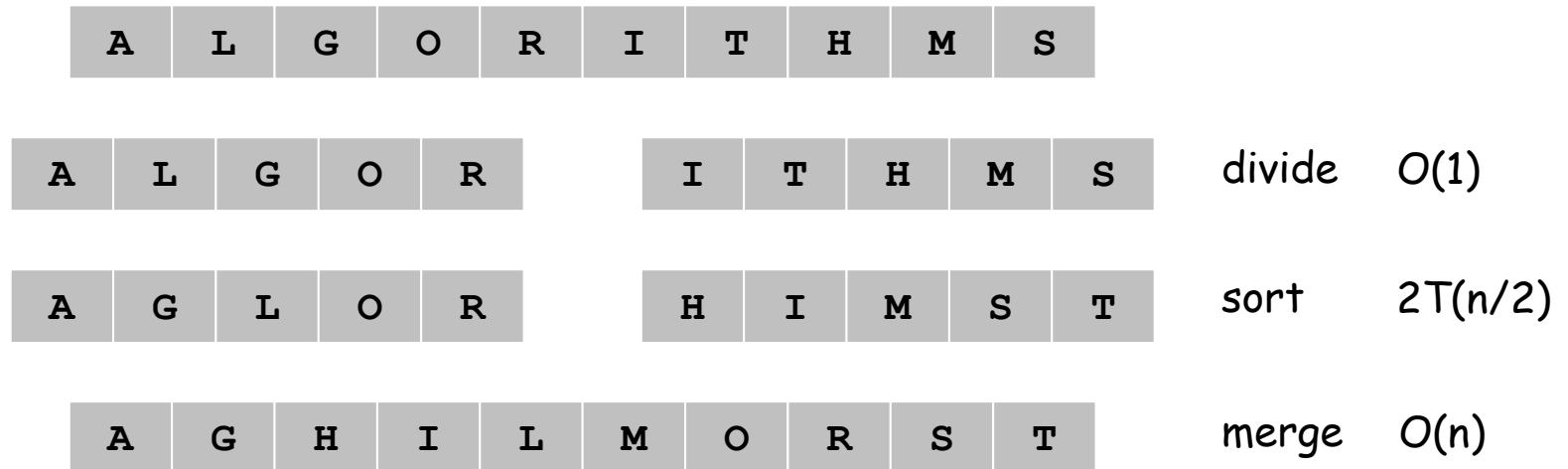
Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.



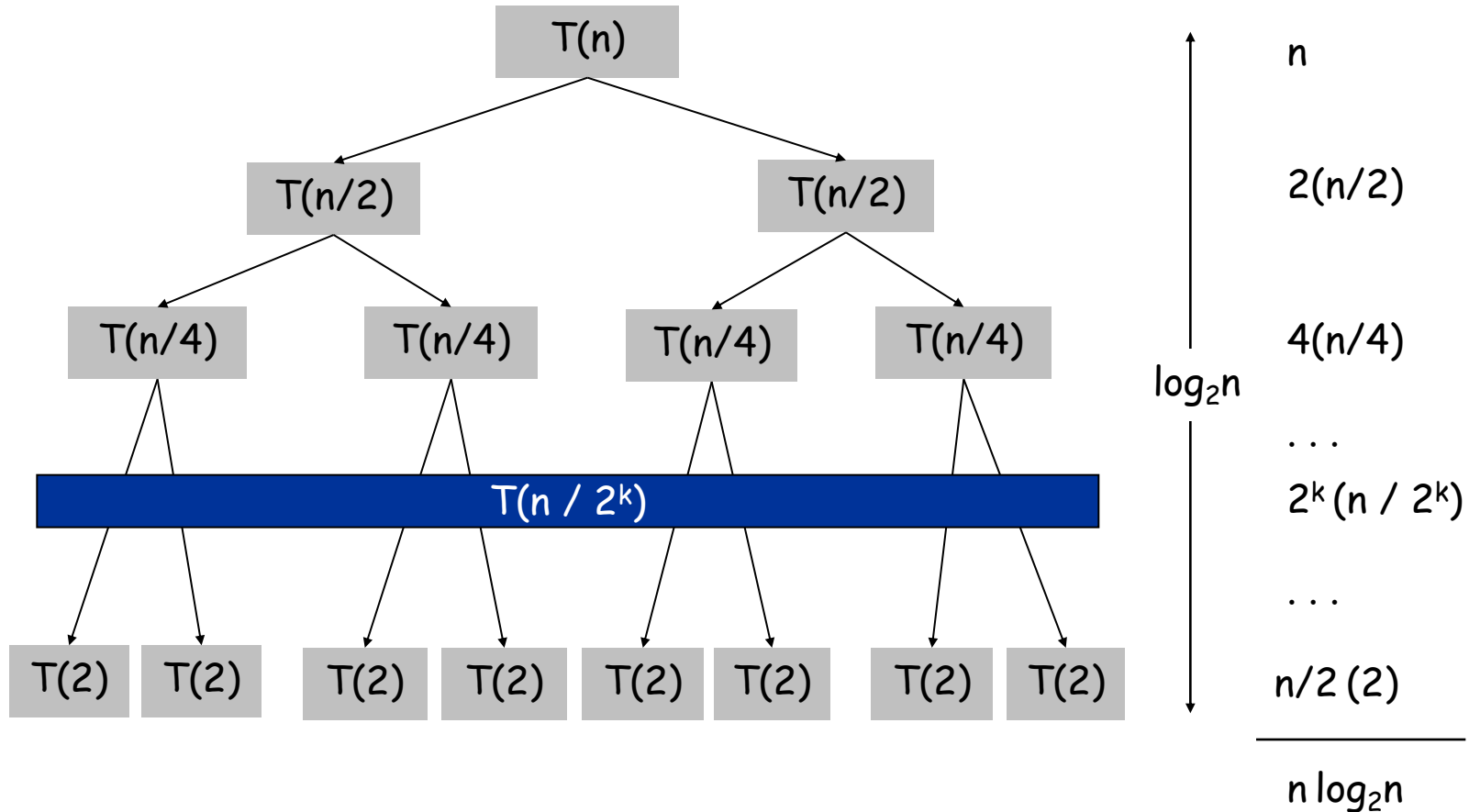
Jon von Neumann (1945)



Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

$\frac{2T(n/2)}{14243}$
sorting both halves
 $+ \{n\}$
merging



5.3 Counting Inversions

Counting Inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with **similar** tastes.

Similarity metric: number of inversions between two rankings.

- My rank: $1, 2, \dots, n$.
- Your rank: a_1, a_2, \dots, a_n .
- Songs i and j **inverted** if $i < j$, but $a_i > a_j$.

Want: Count number of inversions

	Songs				
	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions
3-2, 4-2

Brute force: check all $\Theta(n^2)$ pairs i and j .

Applications

Applications.

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's Tau distance).

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

1	5	4	8	10	2	6	9	12	11	3	7
---	---	---	---	----	---	---	---	----	----	---	---

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide:** separate list into two pieces.

1	5	4	8	10	2	6	9	12	11	3	7
---	---	---	---	----	---	---	---	----	----	---	---

Divide: $O(1)$.

1	5	4	8	10	2	6	9	12	11	3	7
---	---	---	---	----	---	---	---	----	----	---	---

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- **Conquer**: recursively count inversions in each half.



Divide: $O(1)$.



Conquer: $2T(n / 2)$

5 blue-blue inversions

8 green-green inversions

5-4, 5-2, 4-2, 8-2, 10-2

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide:** separate list into two pieces.
- **Conquer:** recursively count inversions in each half.
- **Combine:** count inversions where a_i and a_j are in different halves, and return sum of three quantities.



Divide: $O(1)$.



5 blue-blue inversions

8 green-green inversions

Conquer: $2T(n / 2)$

9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

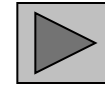
Combine: ???

Total = $5 + 8 + 9 = 22$.

Counting Inversions: Combine

Combine: count blue-green inversions

Q: What happens if each half is **sorted**?



A: Can count blue-green inversions in $O(n)$



13 blue-green inversions: $6 + 3 + 2 + 2 + 0 + 0$

For $j = 1$ to $|R|$,

- Find first element L_{pos} larger than R_j ; all further elements in L are larger than $R_j \Rightarrow \text{count}(j) = |L| - pos.$

Then add all of the $\text{count}(j)$'s

Notice sorting **inside** blue and green arrays does not change the number of blue-green inversions

Counting Inversions: Implementation

```
Count_Inversions(L) {  
    if list L has one element  
        return 0 and the list L  
  
    Divide the list into two halves A and B  
    (rA) ← Count_Inversions(A)  
    (A) ← Sort(A)  
    (rB) ← Count_Inversions(B)  
    (B) ← Sort(B)  
    r ← Count_Inversions_Between(A, B)  
  
    return rA + rB + r  
}
```

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n \log n) \Rightarrow T(n) =$$

Counting Inversions: Implementation

```
Count_Inversions(L) {  
    if list L has one element  
        return 0 and the list L  
  
    Divide the list into two halves A and B  
    (rA) ← Count_Inversions(A)  
    (A) ← Sort(A)  
    (rB) ← Count_Inversions(B)  
    (B) ← Sort(B)  
    r ← Count_Inversions_Between(A, B)  
  
    return rA + rB + r  
}
```

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n)$$

Counting Inversions: Combine Revised

We want faster, $O(n \log n)$... Cannot spend $O(n \log n)$ to sort in each recursion of `Count_Inversions`

Idea: Make `Count_Inversions` return sorted list

```
Count_Inversions(L) {  
    if list L has one element  
        return 0 and the list L  
  
    Divide the list into two halves A and B  
    (rA) ← Count_Inversions(A)  
    (A) ← Sort(A)  
    (rB) ← Count_Inversions(B)  
    (B) ← Sort(B)  
    r ← Count_Inversions_Between(A, B)  
  
    return rA + rB + r  
}
```

Need to combine A, B
into sorted list



Counting Inversions: Implementation

Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {  
    if list L has one element  
        return 0 and the list L  
  
    Divide the list into two halves A and B  
    (rA, A) ← Sort-and-Count(A)  
    (rB, B) ← Sort-and-Count(B)  
    r ← Count_Inversions_Between(A, B)  
    L ← Merge(A,B)  
  
    return r = rA + rB + r and the sorted list L  
}
```

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \Rightarrow T(n) = O(n \log n)$$

Exercise

Exercise 1: You have the sequence of predicted prices $p(1), p(2), \dots, p(n)$ of a single stock. Give an algorithm that tells when is the best time to buy and sell this stock (if there is no profitable buying/selling, then you don't buy).

The algorithm should be $O(n \log n)$

From now on $O(n \log n)$
is not necessarily about sorting

Ex: $p(1)=9, p(2)=1, p(3)=5 \Rightarrow$ buy on 2, sell on 3, profit of $5-1=4$

Solution: [See also the solution in page 245 of the Kleinberg-Tardos book]

- Break list of prices into $L = (p(1), \dots, p(n/2))$, $R = (p(n/2) + 1, \dots, p(n))$
- Recursively find best buying/selling inside L , and inside R
- Find best way of buying inside L and selling inside $R \Rightarrow$ buy in minimum price in L , sell in maximum price in $R \Rightarrow O(n)$

5.4 Closest Pair of Points

Closest Pair of Points

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

Fundamental geometric primitive.

- Games, graphics, computer vision, geographic information systems, molecular modeling, **air traffic control**.

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

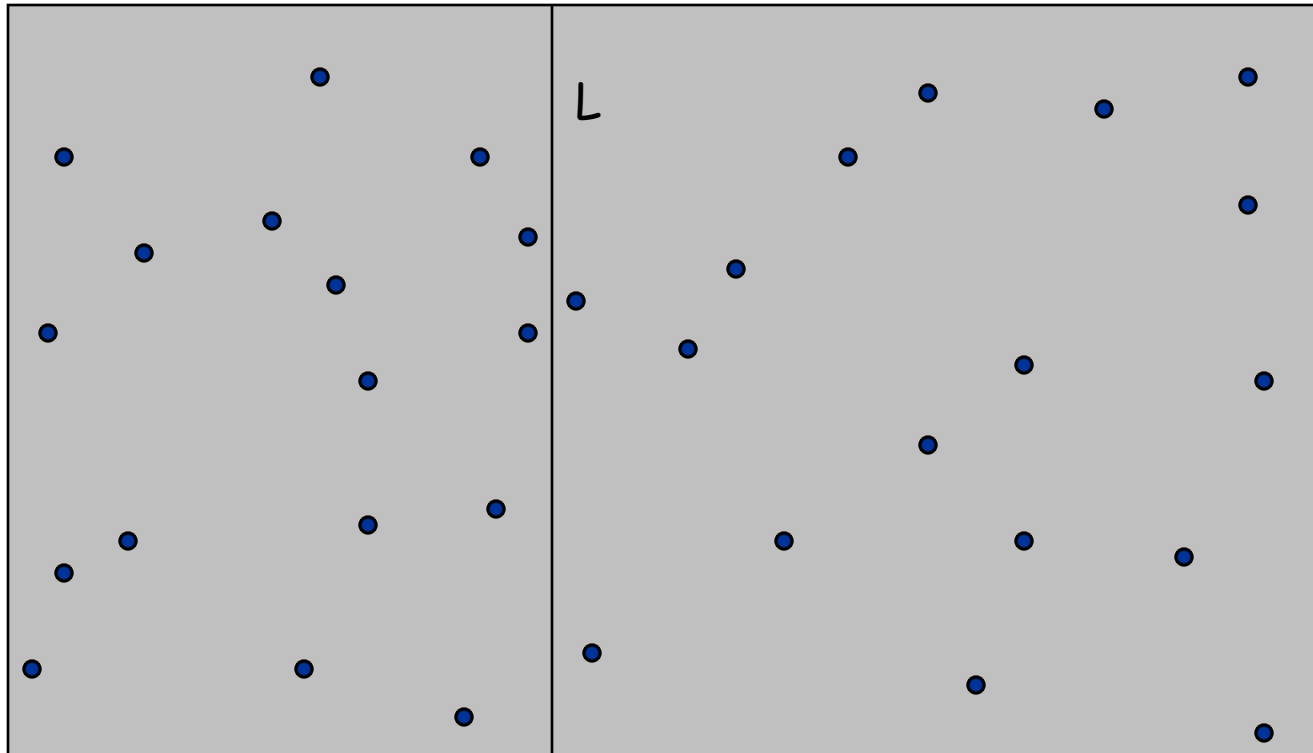
1-D version. $O(n \log n)$ easy if points are on a line.

Assumption to simplify presentation: No two points have same x coordinate.

Closest Pair of Points

Algorithm.

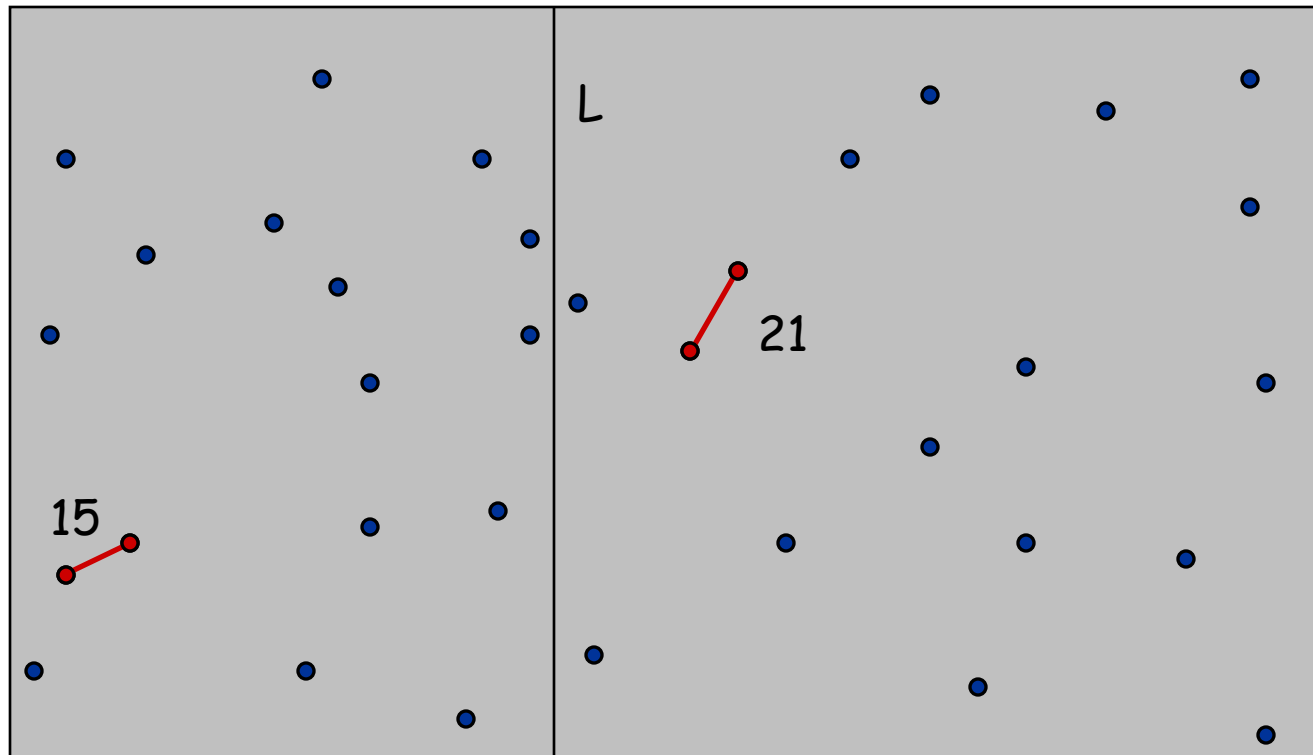
- **Divide:** draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.



Closest Pair of Points

Algorithm.

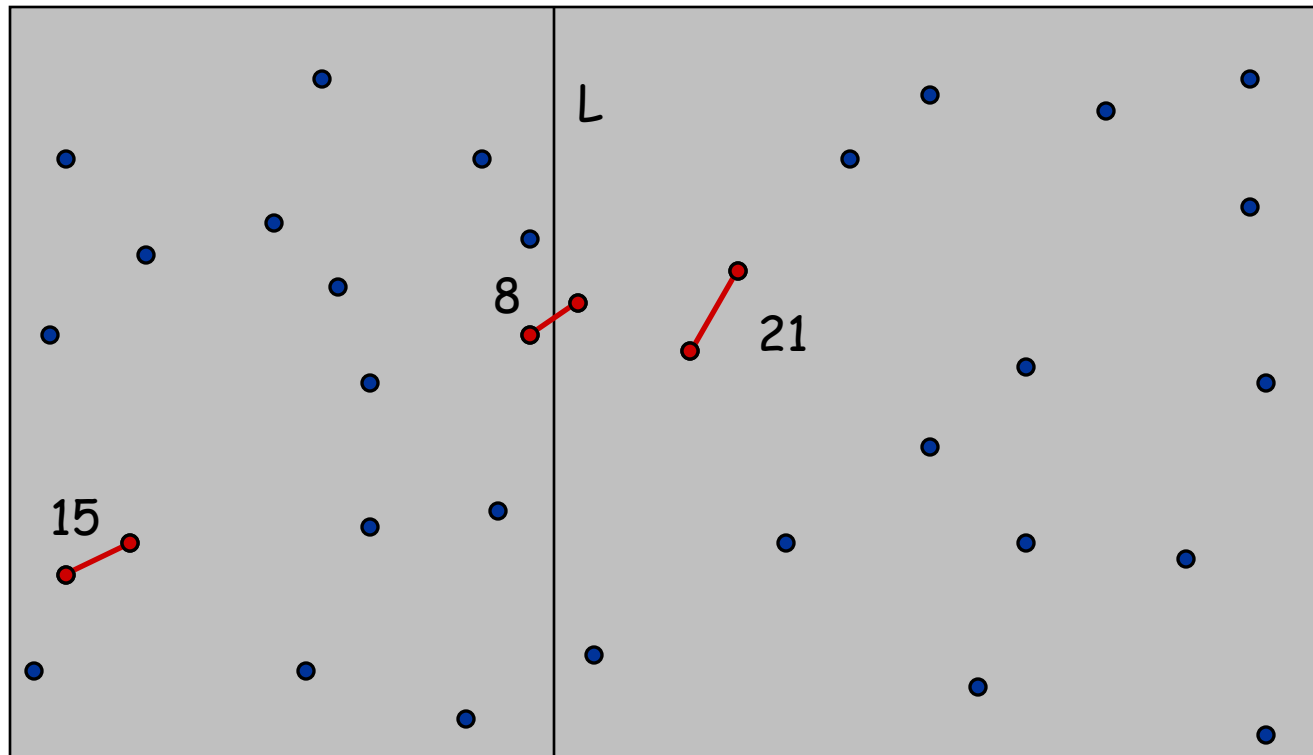
- Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
- **Conquer**: find closest pair in each side recursively.



Closest Pair of Points

Algorithm.

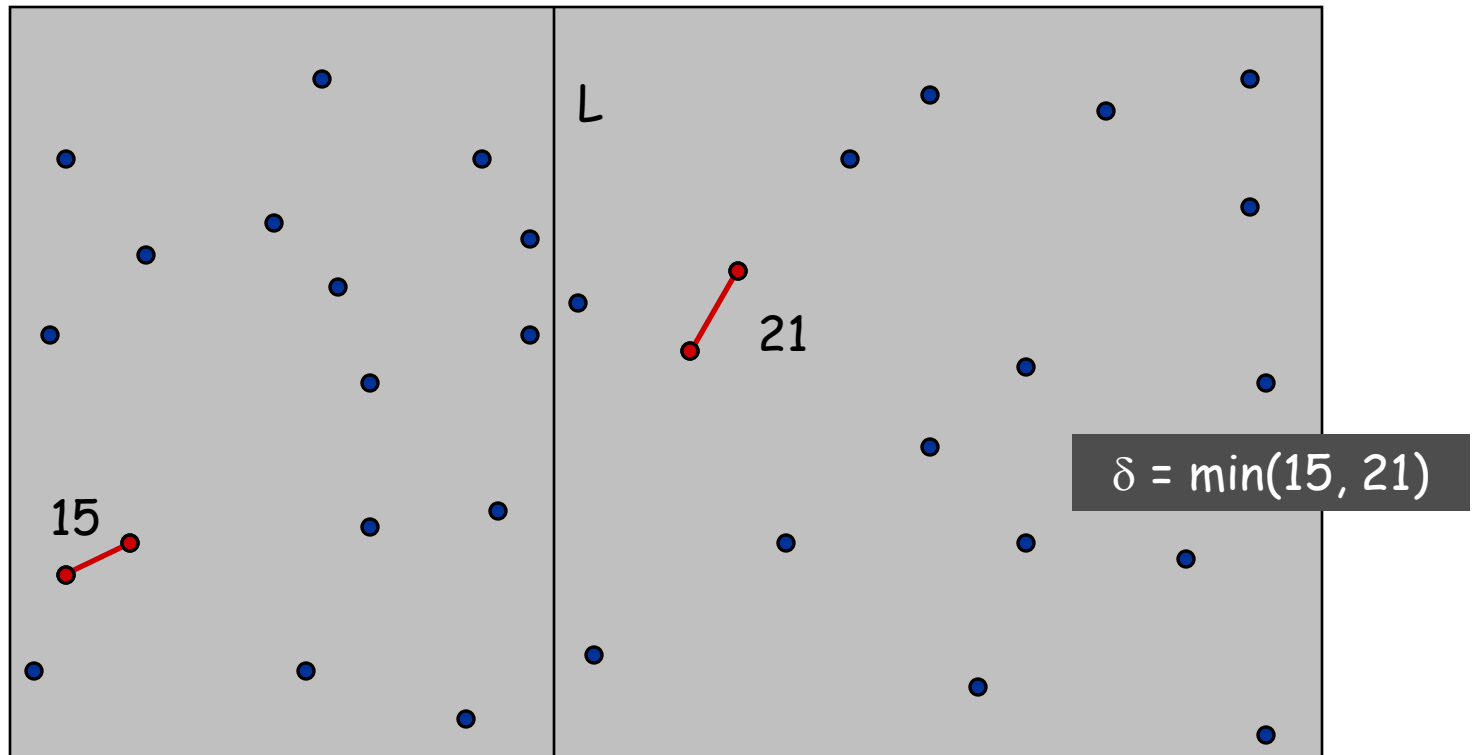
- Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
- Conquer: find closest pair in each side recursively.
- **Combine**: find closest pair with one point in each side. ← seems like $\Theta(n^2)$
- Return best of 3 solutions.



Closest Pair of Points

Let δ be the smallest closest distance found so far

Idea 1: Only need to consider pairs with one point in each side at distance less than δ .

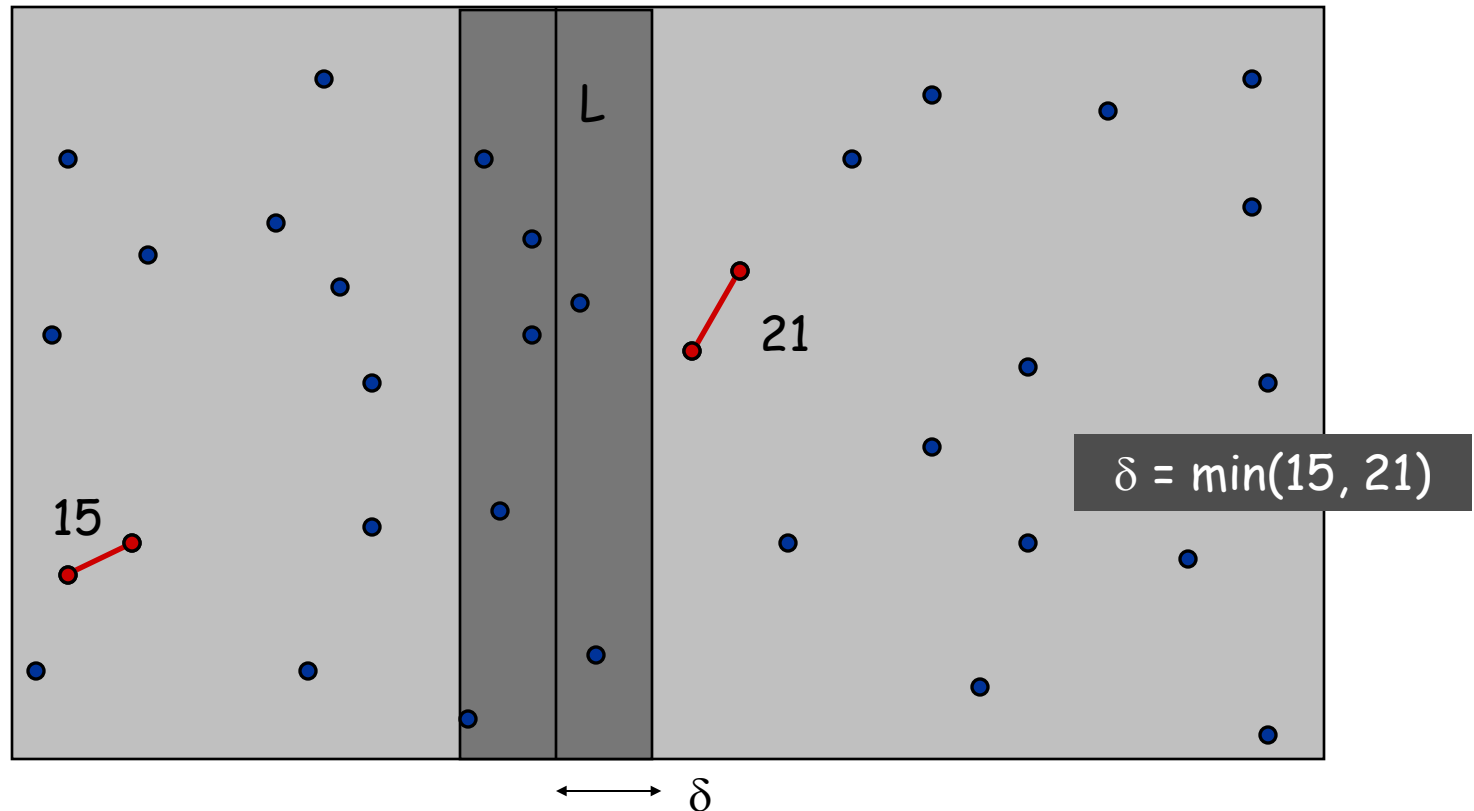


Closest Pair of Points

Let δ be the smallest closest distance found so far

Idea 1: Only need to consider pairs with one point in each side at distance less than δ .

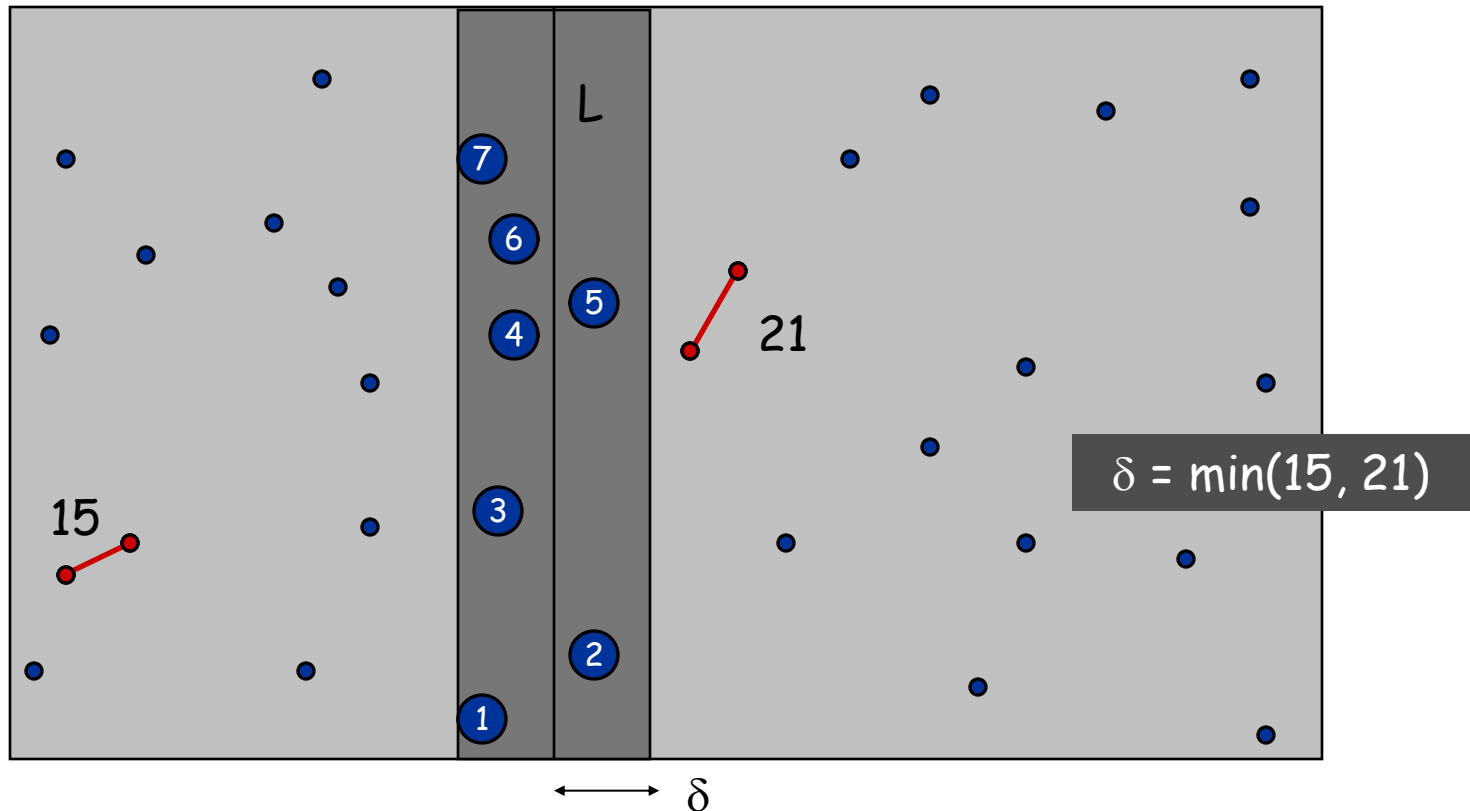
So only need to consider points within δ of line L



Closest Pair of Points

Find closest pair with one point in each side:

- Sort points in 2δ -strip by their y coordinate.

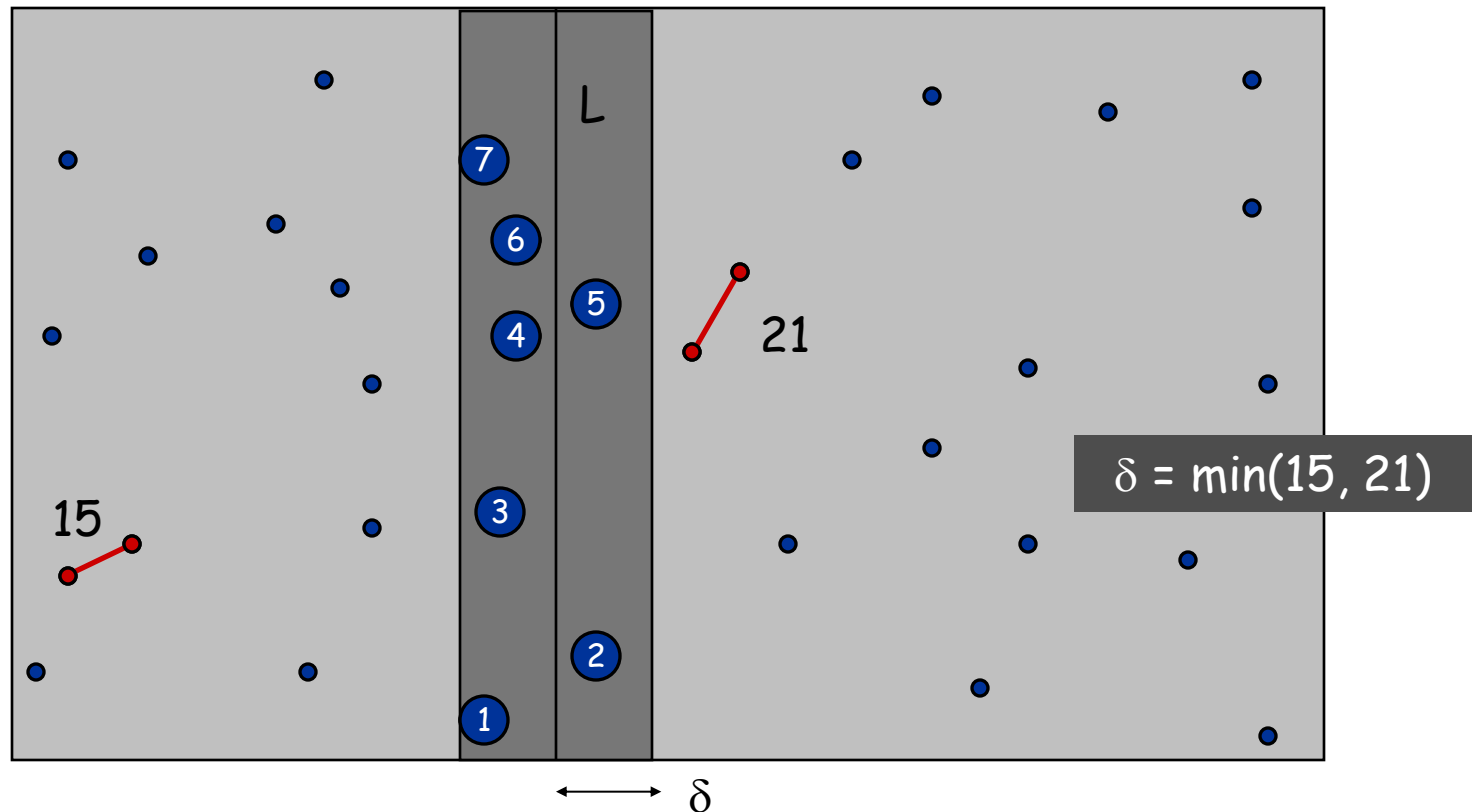


Closest Pair of Points

Find closest pair with one point in each side:

- Sort points in 2δ -strip by their y coordinate.
- Only check distances of those within **11 positions** in sorted list!

[So only need to compute distance between 1 and 2, 3, ..., 12
2 and 3, 4, ..., 13]



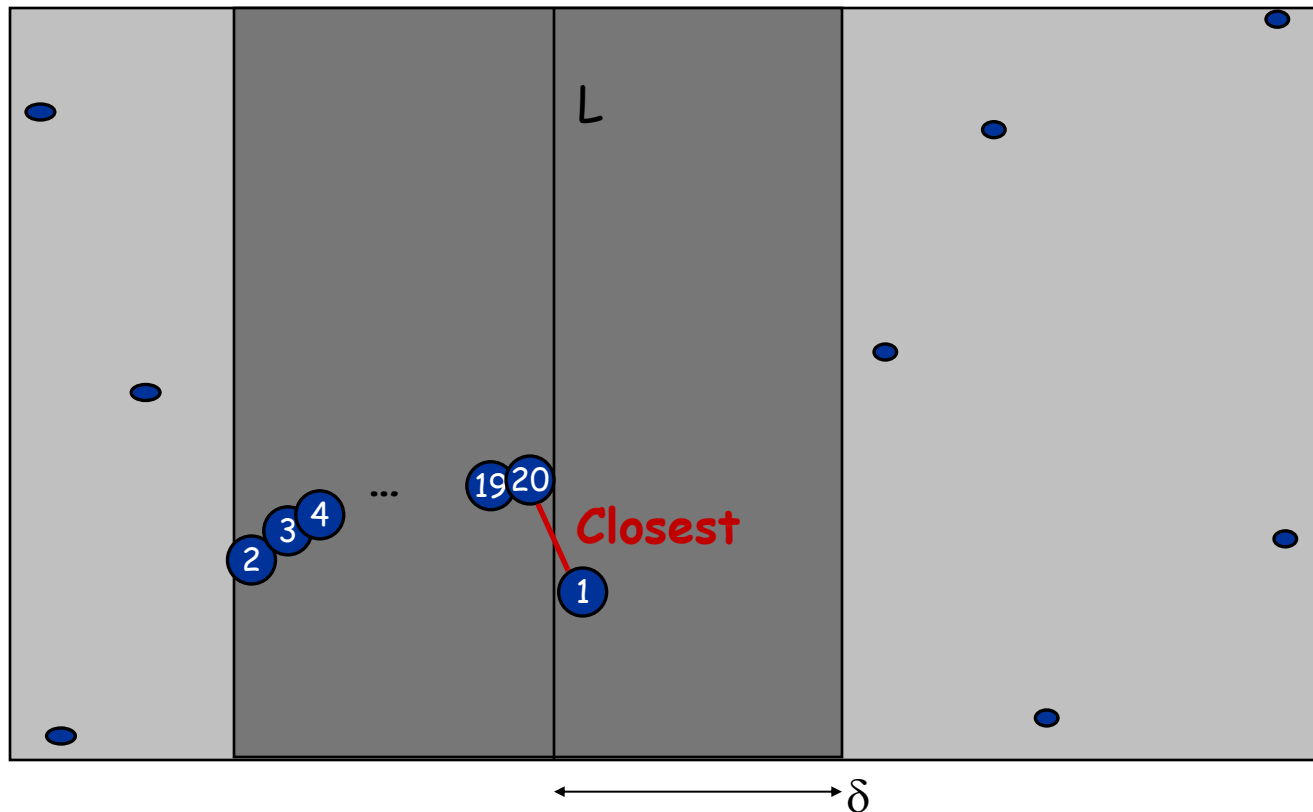
Closest Pair of Points

Find closest pair with one point in each side:

- Sort points in 2δ -strip by their y coordinate.
- Only check distances of those within **11 positions** in sorted list!

Q: Why is this enough? Can the following happen?

A: No! Points on each side are at distance at least δ from each other



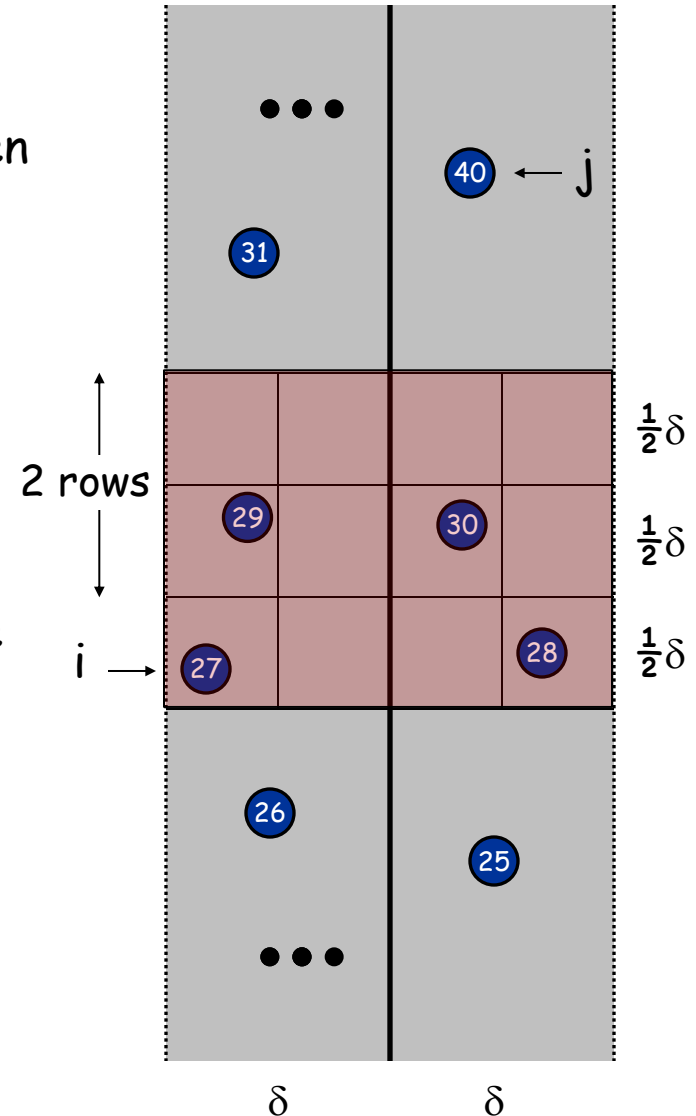
Closest Pair of Points

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

Claim. If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ .

Proof.

- Partition the strip into $\frac{1}{2}\delta$ -by- $\frac{1}{2}\delta$ boxes
- No two points lie in same $\frac{1}{2}\delta$ -by- $\frac{1}{2}\delta$ box.
- Consider box of i -th point...
- ...box of point $j=i+12$ is at least 3 rows above
- \Rightarrow already have distance $\geq 2(\frac{1}{2}\delta)$ from i to j



Closest Pair Algorithm

Sort all points according to x-coordinate.

$O(n \log n)$

`Closest-Pair(p_1, \dots, p_n) {`

`δ_1 = Closest-Pair(left half)`

`δ_2 = Closest-Pair(right half)`

`δ = min(δ_1, δ_2)`

$2T(n/2)$

Delete all points further than δ from separation line L

$O(n)$

Sort remaining points by y-coordinate.

$O(n \log n)$

Scan points in y-order and compare distance between each point and next (in this order) 12 points.

If any of these distances is less than δ , update δ .

$O(n)$

`return δ .`

`}`

Closest Pair of Points: Analysis

Running time.

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

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

A. Yes. Like counting inversions, don't sort points in strip from scratch each time.

- Each recursive call returns the lists of all points sorted by y coordinate
- Sort by **merging** two pre-sorted lists.

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

Closest Pair Algorithm: $O(n \log n)$

[Shamos, Hoey '75]

Sort all points according to x-coordinate.

$O(n \log n)$

```
SortbyY_and_Closest-Pair( $p_1, \dots, p_n$ ) {
```

```
  ( $A, \delta_1$ ) = SortbyY_and_Closest-Pair(left half)
```

```
  ( $B, \delta_2$ ) = SortbyY_and_Closest-Pair(right half)
```

```
   $\delta = \min(\delta_1, \delta_2)$ 
```

$2T(n/2)$

```
   $S \leftarrow$  Merge( $A, B$ ) by y-coordinate.
```

$O(n)$

```
  Let  $S'$  be the list obtained from  $S$  by deleting all  
  points further than  $\delta$  from separation line  $L$ 
```

$O(n)$

```
  Scan points of  $S'$  in y-order and compare distance  
  between each point and next 12 neighbors. If any of  
  these distances is less than  $\delta$ , update  $\delta$ .
```

$O(n)$

```
  return  $\delta$  and  $S$ .
```

```
}
```


Closest Pair of Points

Fastest algorithms

- $O(n \log \log n)$ [Fortune, Hopcroft '79]
- Expected time $O(n)$ [Khuller, Matias '95]

5.5 Integer Multiplication

Divide-and-Conquer Multiplication: Warmup

To multiply two n -digit integers:

- Multiply four $\frac{1}{2}n$ -digit integers.
- Add two $\frac{1}{2}n$ -digit integers, and shift to obtain result.

$$\begin{aligned}x &= 2^{n/2} \cdot x_1 + x_0 \\y &= 2^{n/2} \cdot y_1 + y_0 \\xy &= (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0) = 2^n \cdot x_1y_1 + 2^{n/2} \cdot (x_1y_0 + x_0y_1) + x_0y_0\end{aligned}$$

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



assumes n is a power of 2

Karatsuba Multiplication

To multiply two n -digit integers:

- Add two $\frac{1}{2}n$ digit integers.
- Multiply **three** $\frac{1}{2}n$ -digit integers.
- Add, subtract, and shift $\frac{1}{2}n$ -digit integers to obtain result.

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

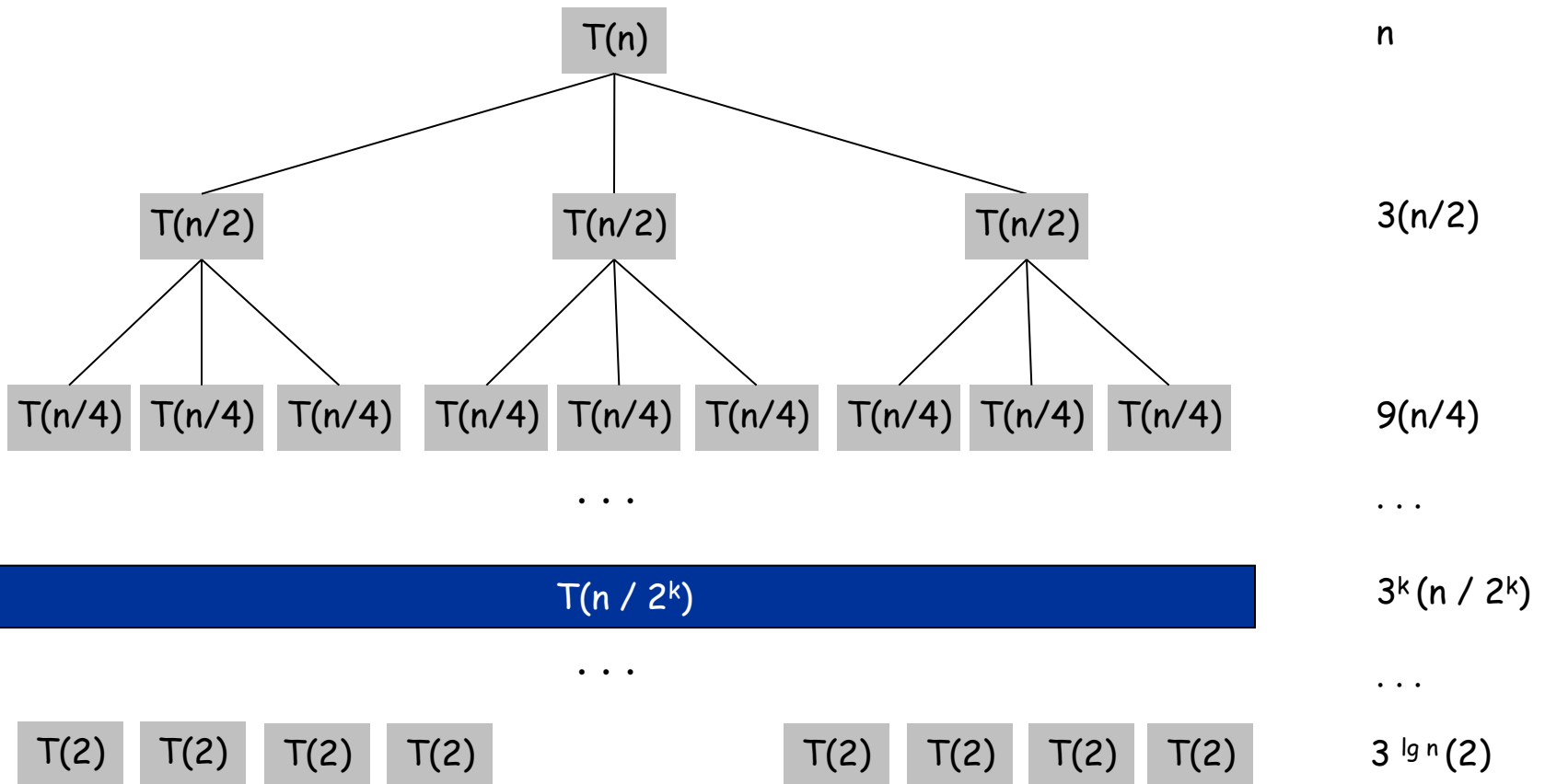
A B A C C

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

Karatsuba: Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\log_2 n} n \left(\frac{3}{2}\right)^k = \frac{\left(\frac{3}{2}\right)^{1+\log_2 n} - 1}{\frac{3}{2} - 1} = 3n^{\log_2 3} - 2$$



Matrix Multiplication

Matrix Multiplication

Matrix multiplication. Given two n -by- n matrices A and B , compute $C = AB$.

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \text{L} & c_{1n} \\ c_{21} & c_{22} & \text{L} & c_{2n} \\ \text{M} & \text{M} & \text{O} & \text{M} \\ c_{n1} & c_{n2} & \text{L} & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \text{L} & a_{1n} \\ a_{21} & a_{22} & \text{L} & a_{2n} \\ \text{M} & \text{M} & \text{O} & \text{M} \\ a_{n1} & a_{n2} & \text{L} & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \text{L} & b_{1n} \\ b_{21} & b_{22} & \text{L} & b_{2n} \\ \text{M} & \text{M} & \text{O} & \text{M} \\ b_{n1} & b_{n2} & \text{L} & b_{nn} \end{bmatrix}$$

Brute force. $\Theta(n^3)$ arithmetic operations.

Fundamental question. Can we improve upon brute force?

Matrix Multiplication: Warmup

Divide-and-conquer.

- Divide: partition A and B into $\frac{1}{2}n$ -by- $\frac{1}{2}n$ blocks.
- Conquer: multiply 8 $\frac{1}{2}n$ -by- $\frac{1}{2}n$ recursively.
- Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$T(n) = \underbrace{8}_{14243} T(n/2) + \underbrace{1443}_{\Theta(n^2)} \Rightarrow T(n) = \Theta(n^3)$$

recursive calls add, form submatrices

Matrix Multiplication: Key Idea

Key idea. multiply 2-by-2 block matrices with only **7** multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

$$P_1 = A_{11} \times (B_{12} - B_{22})$$

$$P_2 = (A_{11} + A_{12}) \times B_{22}$$

$$P_3 = (A_{21} + A_{22}) \times B_{11}$$

$$P_4 = A_{22} \times (B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$

$$P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

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

- 7 multiplications.
- 18 = 10 + 8 additions (or subtractions).

Fast Matrix Multiplication

Fast matrix multiplication. (Strassen, 1969)

- Divide: partition A and B into $\frac{1}{2}n$ -by- $\frac{1}{2}n$ blocks.
- Compute: 14 $\frac{1}{2}n$ -by- $\frac{1}{2}n$ matrices via 10 matrix additions.
- Conquer: multiply 7 $\frac{1}{2}n$ -by- $\frac{1}{2}n$ matrices recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

- Assume n is a power of 2.
- $T(n) = \#$ arithmetic operations.

$$T(n) = \underbrace{7}_{\text{recursive calls}} T\left(\frac{n}{2}\right) + \underbrace{14}_{\text{add, subtract}} \frac{n^2}{4} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Fast Matrix Multiplication in Practice

Implementation issues.

- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.

Common misperception: "Strassen is only a theoretical curiosity."

- Advanced Computation Group at Apple Computer reports 8x speedup on G4 Velocity Engine when $n \sim 2,500$.
- Range of instances where it's useful is a subject of controversy.

Remark. Can "Strassenize" $Ax=b$, determinant, eigenvalues, and other matrix ops.

Fast Matrix Multiplication in Theory

Q. Multiply two 2-by-2 matrices with only 7 scalar multiplications?

A. Yes! [Strassen, 1969]

Q. Multiply two 2-by-2 matrices with only 6 scalar multiplications?

A. Impossible. [Hopcroft and Kerr, 1971]

Q. Two 3-by-3 matrices with only 21 scalar multiplications?

A. Also impossible.

Q. Two 70-by-70 matrices with only 143,640 scalar multiplications?

A. Yes! [Pan, 1980]

Decimal wars.

- December, 1979: $O(n^{2.521813})$.
- January, 1980: $O(n^{2.521801})$.

Fast Matrix Multiplication in Theory

Best known. ~~$O(n^{2.376})$~~ [Coppersmith-Winograd, 1987.]

$O(n^{2.3728639})$ [Williams '13, Le Gall '14]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

Exercise

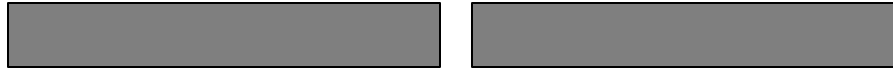
Max interval sum: Given a list with n values (positive or negative), compute the interval with largest sum in time $O(n \log n)$.

Ex: On input $A = [-2, 3, -1, 3, -4]$, return the interval with largest sum starts at position 2 and ends at position 4, with the sum $3 - 1 + 3 = 5$

Exercises of Kleinberg-Tardos (Chapter 5): 2, 3

Another type of divide-and-conquer: binary search

In all examples so far we have recursed on **both** halves of the instance



But another important class of div-and-conquer algorithms are like binary search: only recurse on **one** half



Typically these have running time $O(\log n)$

Another type of divide-and-conquer: binary search

Exercise: Consider a sequence of distinct numbers a_1, a_2, \dots, a_n that is **unimodal**, that is, it is increasing up to some point a_i , and then decreasing after that. This maximum element a_i is called the **mode**.

Give a $O(\log n)$ algorithm for finding the mode of the sequence

A: Look at the middle of the sequence and check if it the mode:

if $a_{n/2-1} < a_{n/2} > a_{n/2+1}$, return $a_{n/2}$

Else, compute "slope" in the middle of the sequence to see which half should recurse on:

if $a_{n/2} > a_{n/2+1}$, recurse on $a_1, \dots, a_{n/2}$

else recurse on $a_{n/2+1}, \dots, a_n$

Another type of divide-and-conquer: binary search

Exercise: Consider a **strictly increasing** sequence a_1, a_2, \dots, a_n of integers. Find if there is an element with $a_i = i$ in time $O(\log n)$.

A: Look at the middle of the sequence:

if $a_i > i$, since strictly increasing we have $a_{i+1} > i + 1, a_{i+2} > i + 2 \dots$
 \Rightarrow recurse on first half of sequence

if $a_i < i$, similarly we have $a_{i-1} > i - 1, a_{i-2} > i - 2 \dots$
 \Rightarrow recurse on second half of sequence