Chapter 5

Divide and Conquer
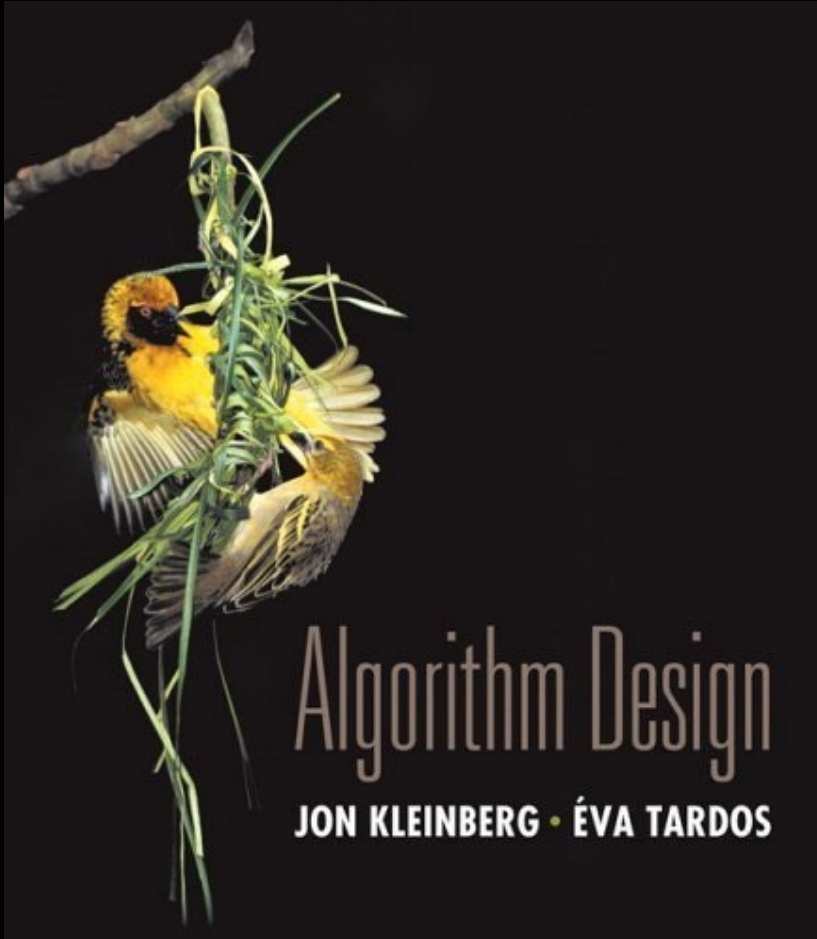
Algorithm Design

JON KLEINBERG · ÉVA TARDOS

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 ½n.
- 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

Sorting

Sorting. Given n elements, rearrange in ascending order.

Obvious sorting applications.
    List files in a directory.
    Organize an MP3 library.
    List names in a phone book.
    Display Google PageRank
    results.

Problems become easier once sorted.
    Find the median.
    Find the closest pair.
    Binary search in a database.
    Identify statistical outliers.
    Find duplicates in a mailing
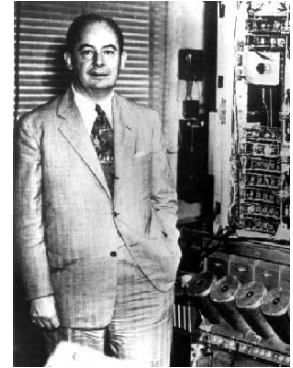   list.

Non-obvious sorting applications.
    Data compression.
    Computer graphics.
    Interval scheduling.
    Computational biology.
    Minimum spanning tree.
    Supply chain management.
    Simulate a system of
    particles.
    Book recommendations on
    Amazon.
    Load balancing on a parallel
    computer.
    . . .

Mergesort

Mergesort.

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

Jon von Neumann (1945)

| A | L | G | O | R | I | T | H | M | S |

| A | L | G | O | R | | I | T | H | M | S | divide | $O(1)$ |

| A | G | L | O | R | | H | I | M | S | T | sort | $2T(n/2)$ |

| A | G | H | I | L | M | O | R | S | T | merge | $O(n)$ |

Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.

A	G	L	O	R		H	I	M	S	T

A	G	H	I						

Challenge for the bored. In-place merge. [Kronrud, 1969]

↑

using only a constant amount of extra storage

A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.
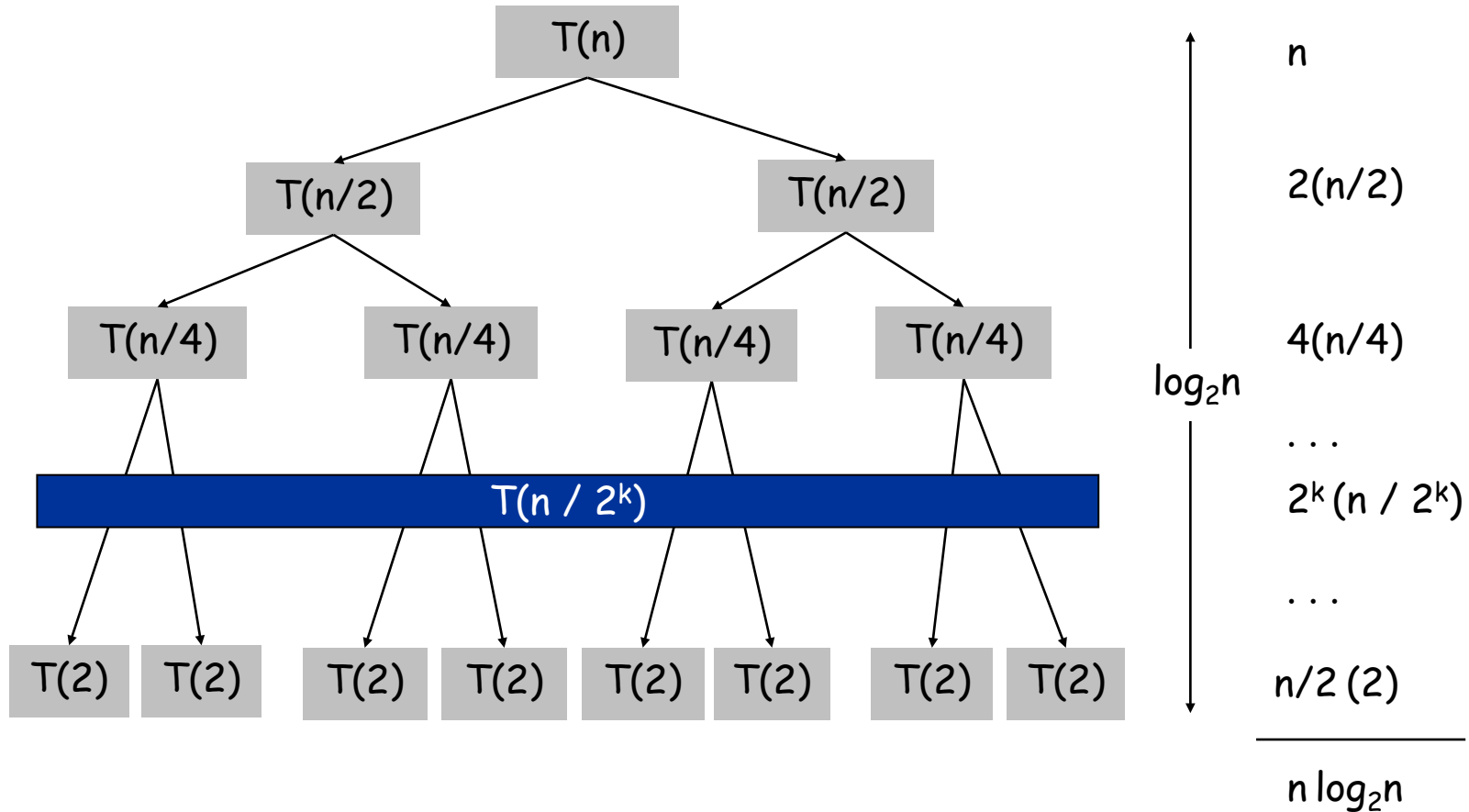
Mergesort recurrence.

$$T(n) \leq \begin{cases} 0 & \text{if } n=1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(n \log_2 n)$.

Assorted proofs. We describe several ways to prove this recurrence.
Initially we assume n is a power of 2 and replace $\leq$ with =.

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

T(n)

T(n/2)   T(n/2)

T(n/4)   T(n/4)   T(n/4)   T(n/4)

T(n / 2^k)

T(2) T(2)   T(2) T(2)   T(2) T(2)   T(2) T(2)

$\log_2 n$

n

$2(n/2)$

$4(n/4)$

. . .

$2^k(n / 2^k)$

. . .

$n/2\,(2)$

―――――――

$n \log_2 n$

Proof by Telescoping

Claim. If T(n) satisfies this recurrence, then T(n) = n log$_2$ n.

↑
assumes n is a power of 2

$$
T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}
$$

Pf. For n > 1:

$$\frac{T(n)}{n} = \frac{2T(n/2)}{n} + 1$$

$$= \frac{T(n/2)}{n/2} + 1$$

$$= \frac{T(n/4)}{n/4} + 1 + 1$$

$$\dots$$

$$= \frac{T(n/n)}{n/n} + \underbrace{1 + \dots + 1}_{\log_2 n}$$

$$= \log_2 n$$

Proof by Induction

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

↑
assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: n = 1.
- Inductive hypothesis: T(n) = n $\log_2$ n.
- Goal: show that T(2n) = 2n $\log_2$ (2n).

$$
\begin{aligned}
T(2n) &= 2T(n) + 2n \\
&= 2n \log_2 n + 2n \\
&= 2n\big(\log_2(2n) - 1\big) + 2n \\
&= 2n \log_2(2n)
\end{aligned}
$$

Analysis of Mergesort Recurrence

Claim. If T(n) satisfies the following recurrence, then $T(n) \leq n \lceil \lg n \rceil$.

$\uparrow$
$\log_2 n$

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: n = 1.
- Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.
- Induction step: assume true for 1, 2, ... , n–1.

$$\begin{aligned} T(n) &\leq T(n_1) + T(n_2) + n \\ &\leq n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &\leq n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &= n \lceil \lg n_2 \rceil + n \\ &\leq n(\lceil \lg n \rceil - 1) + n \\ &= n \lceil \lg n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\leq \lceil 2^{\lceil \lg n \rceil} / 2 \rceil \\ &= 2^{\lceil \lg n \rceil} / 2 \\ \Rightarrow \lg n_2 &\leq \lceil \lg n \rceil - 1 \end{aligned}$$

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, …, n.
- Your rank: $a_1, a_2, …, a_n$.
- Songs i and j inverted if i < j, but $a_i > a_j$.

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).

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 |

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

Divide: O(1).

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

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

| 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 |

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.

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

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

- Assume each half is sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole.
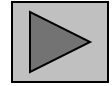
to maintain sorted invariant

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

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

Count: O(n)

2	3	7	10	11	14	16	17	18	19	23	25

Merge: O(n)

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

Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] A and B are sorted.
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
    (r_A, A) ← Sort-and-Count(A)
    (r_B, B) ← Sort-and-Count(B)
    (r , L) ← Merge-and-Count(A, B)

    return r = r_A + r_B + r and the sorted list L
}
```

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.
- 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.

↑
to make presentation cleaner

Divide. Sub-divide region into 4 quadrants.

Closest Pair of Points: First Attempt

Divide. Sub-divide region into 4 quadrants.

Obstacle. Impossible to ensure n/4 points in each piece.
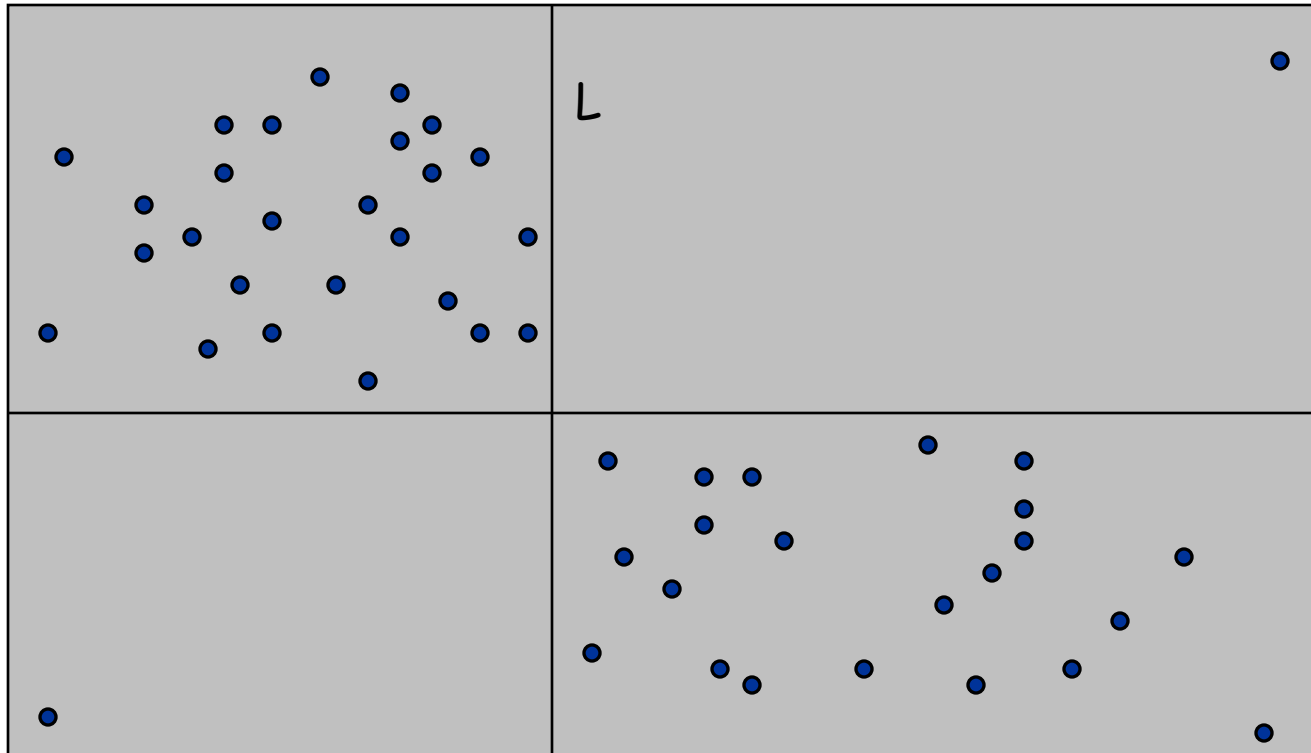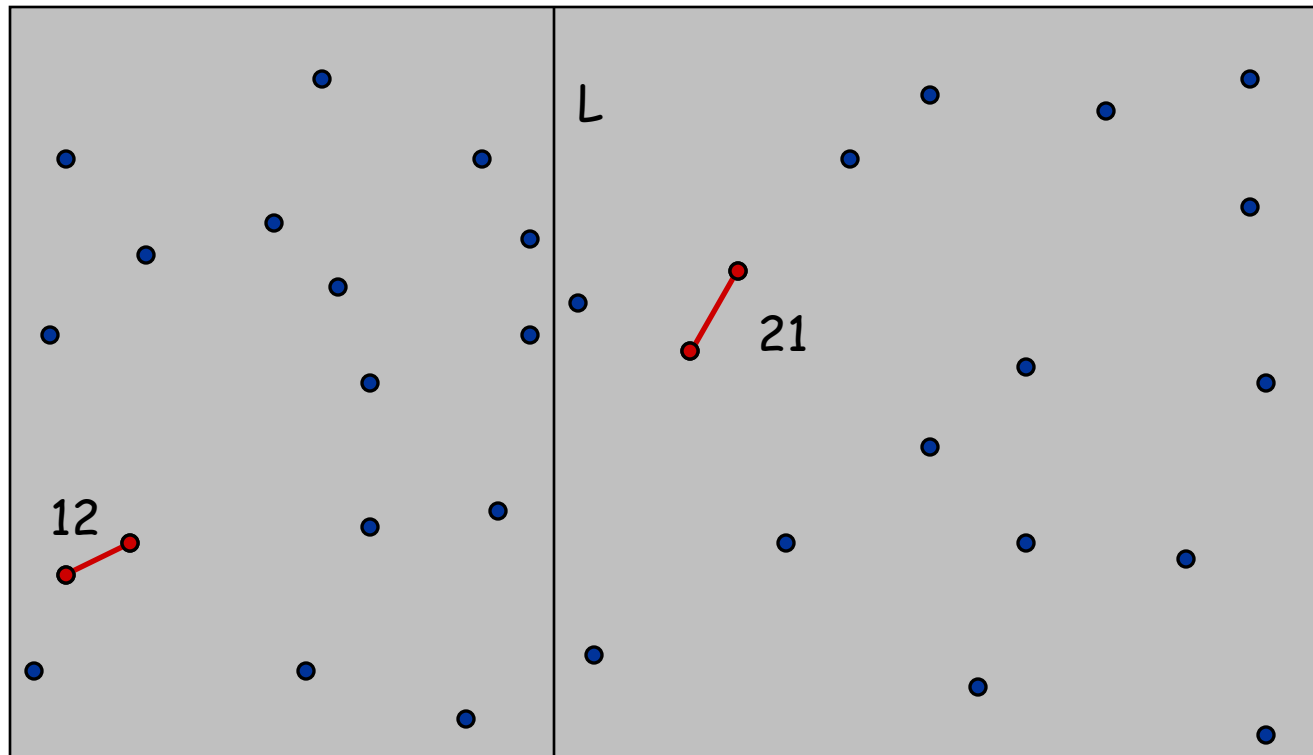
Closest Pair of Points

Algorithm.

- Divide: draw vertical line L so that roughly ½n points on each side.
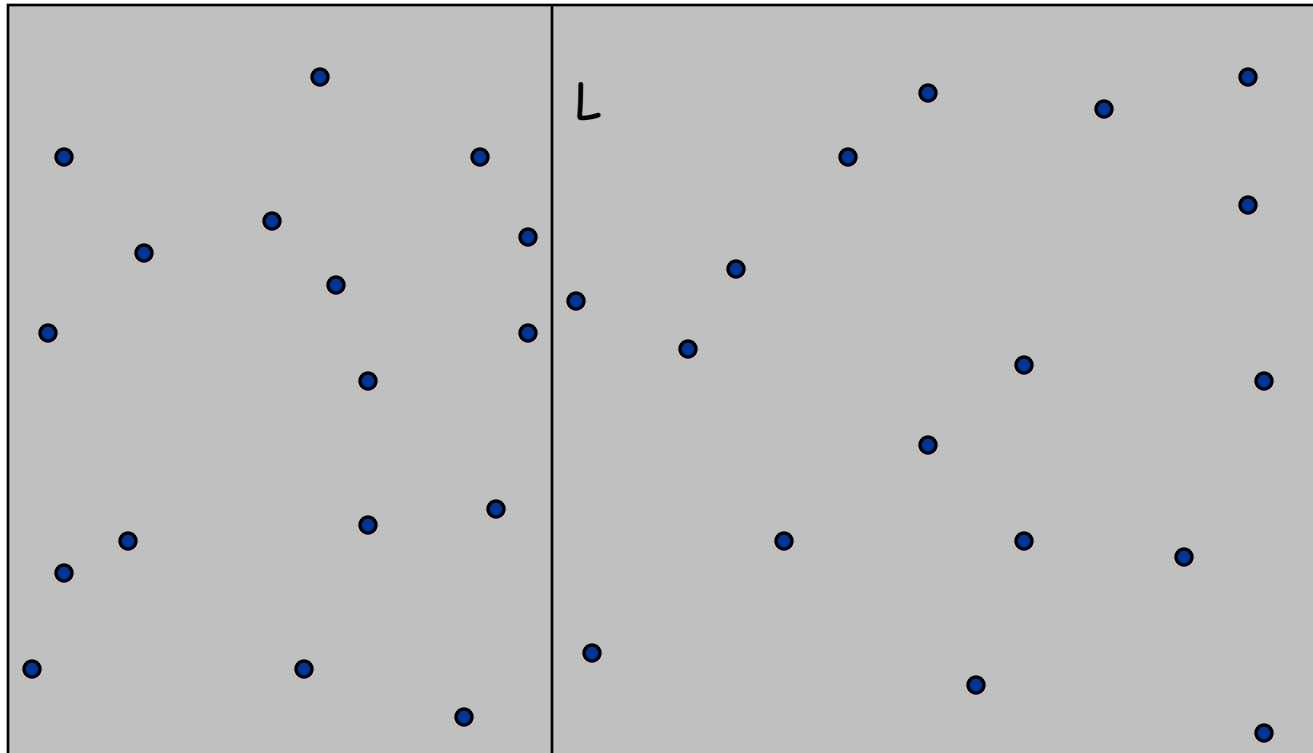
Closest Pair of Points

Algorithm.

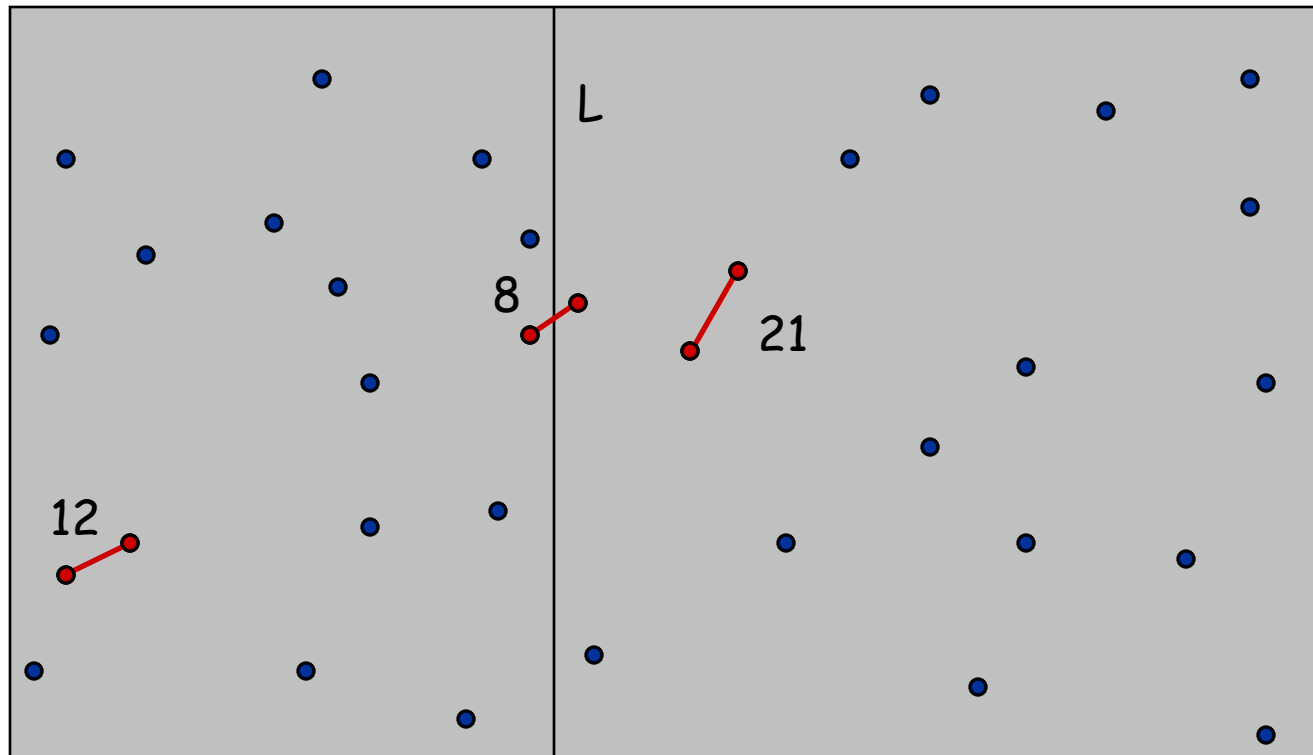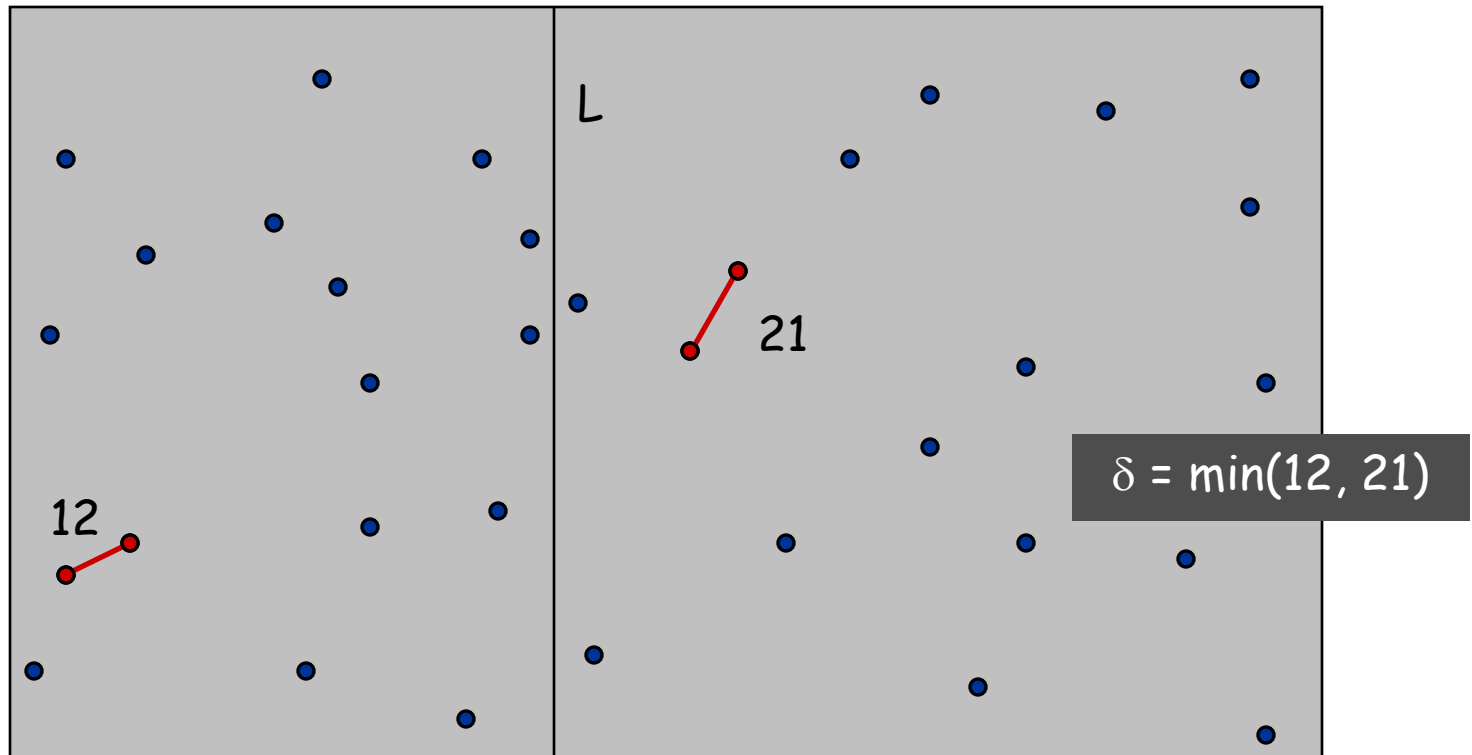- Divide: draw vertical line L so that roughly ½n points on each side.
- Conquer: find closest pair in each side recursively.

Closest Pair of Points

Algorithm.

- Divide: draw vertical line L so that roughly ½n 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.
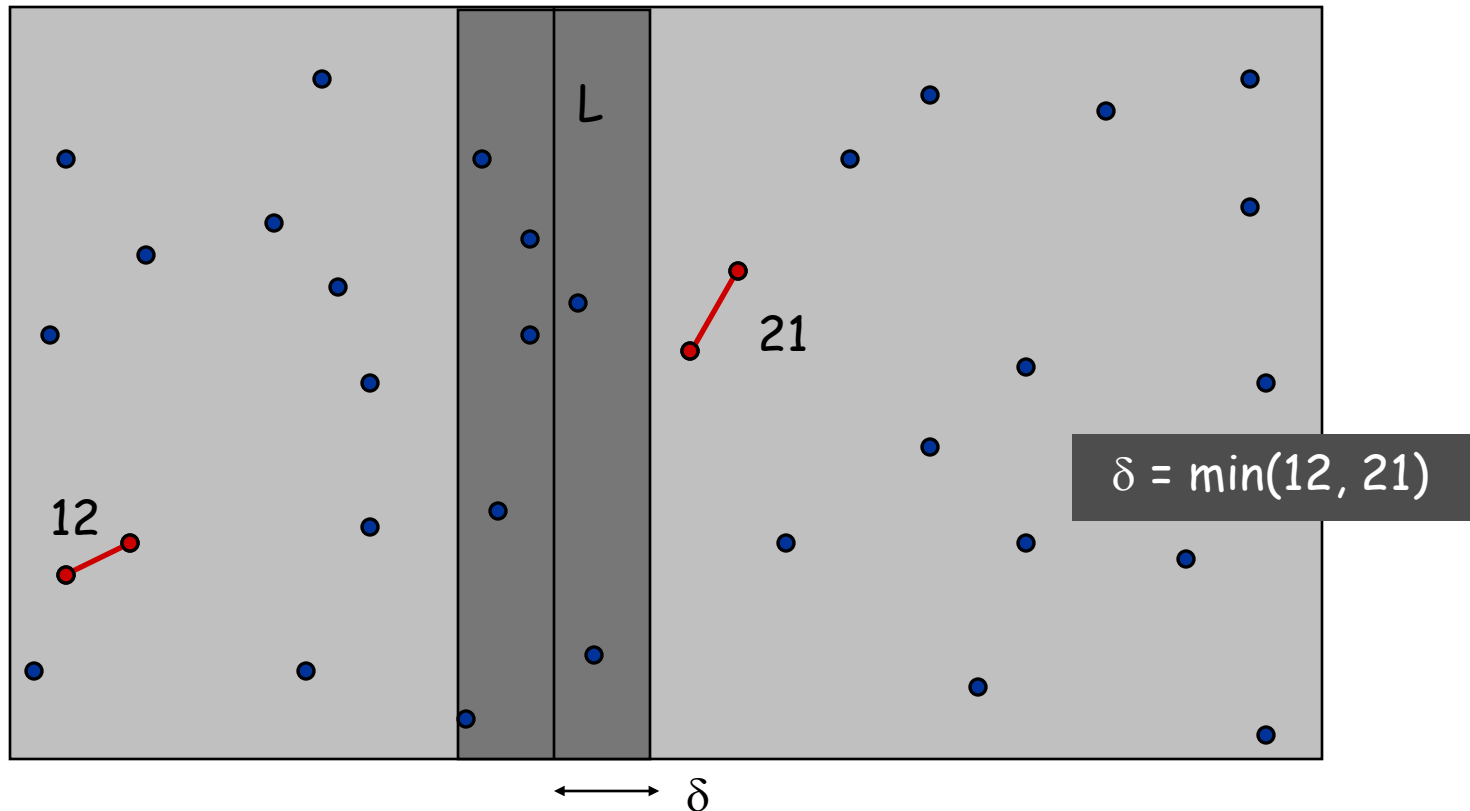
Closest Pair of Points

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

L

21

12

δ = min(12, 21)

Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.
- Observation: only need to consider points within δ of line L.

L

21

12

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

δ

Closest Pair of Points

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

- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.

δ = min(12, 21)

Closest Pair of Points

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

- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.
- Only check distances of those within 11 positions in sorted list!
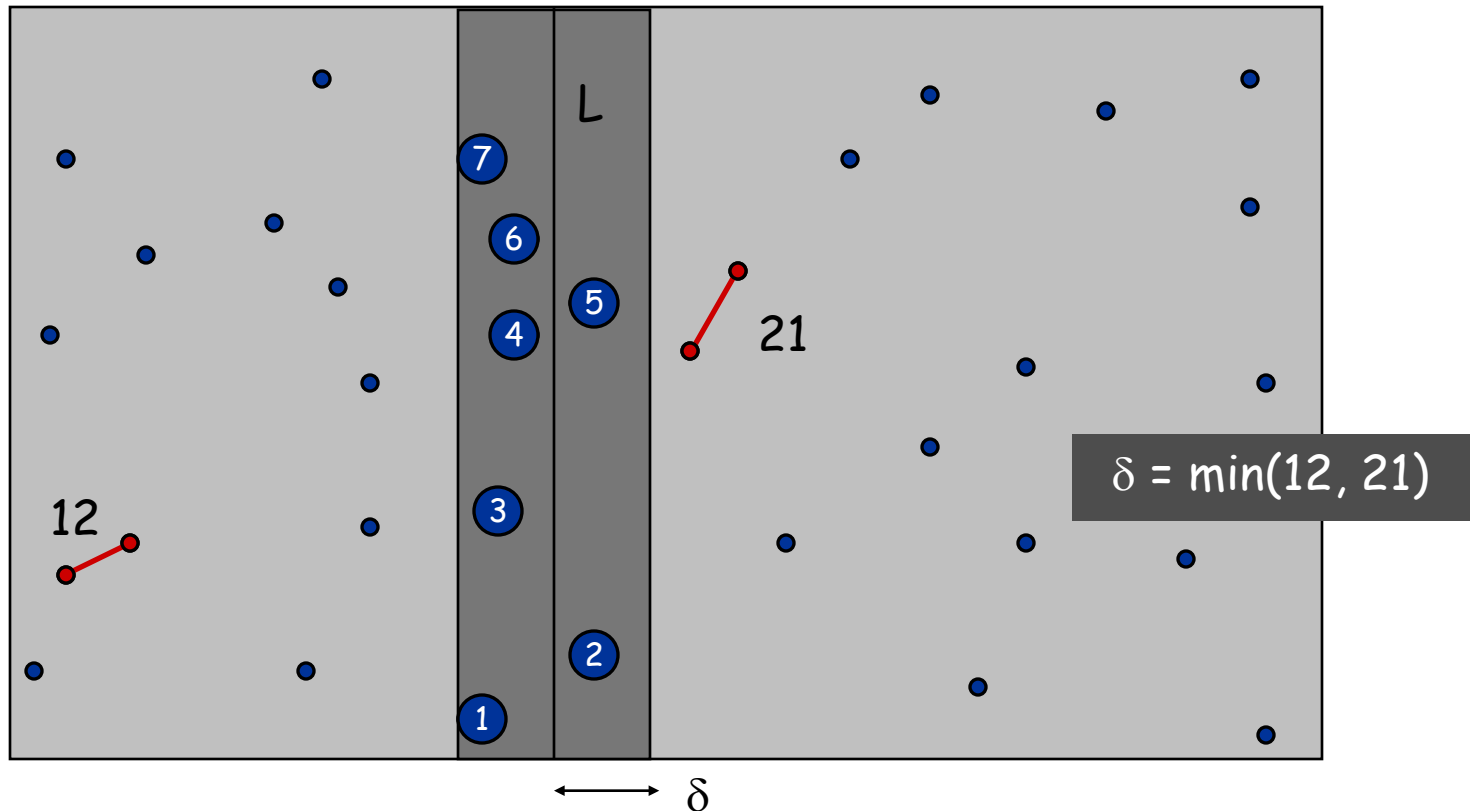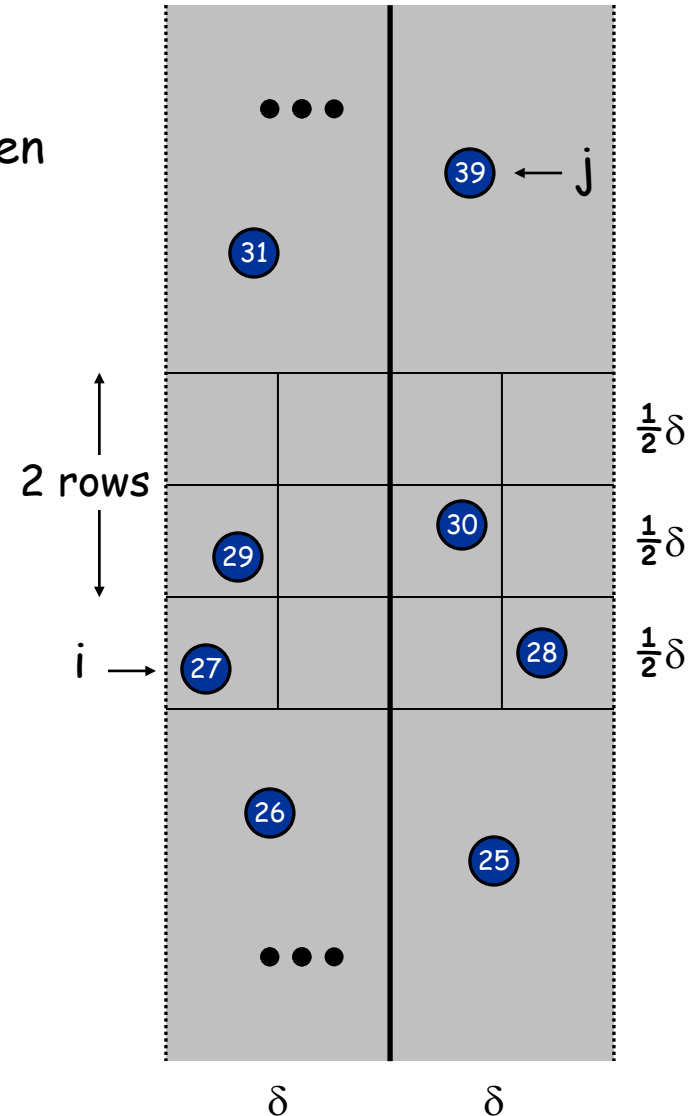
Closest Pair of Points

Def. Let s_i be the point in the 2δ-strip, with the ith smallest y-coordinate.

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

Pf.

- No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
- Two points at least 2 rows apart have distance $\geq\ 2(\frac{1}{2}\delta)$. ∎

Fact. Still true if we replace 12 with 7.

Closest Pair Algorithm

```
Closest-Pair(p₁, …, pₙ) {
    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 by y-coordinate.

    Scan points in y-order and compare distance between
    each point and next 11 neighbors. If any of these
    distances is less than δ, update δ.

    return δ.
}
```

$O(n \log n)$

$2T(n / 2)$

$O(n)$

$O(n \log n)$

$O(n)$

Closest Pair of Points: Analysis

Running time.

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

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

A. Yes. Don't sort points in strip from scratch each time.
   ▫ Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
   ▫ Sort by merging two pre-sorted lists.

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