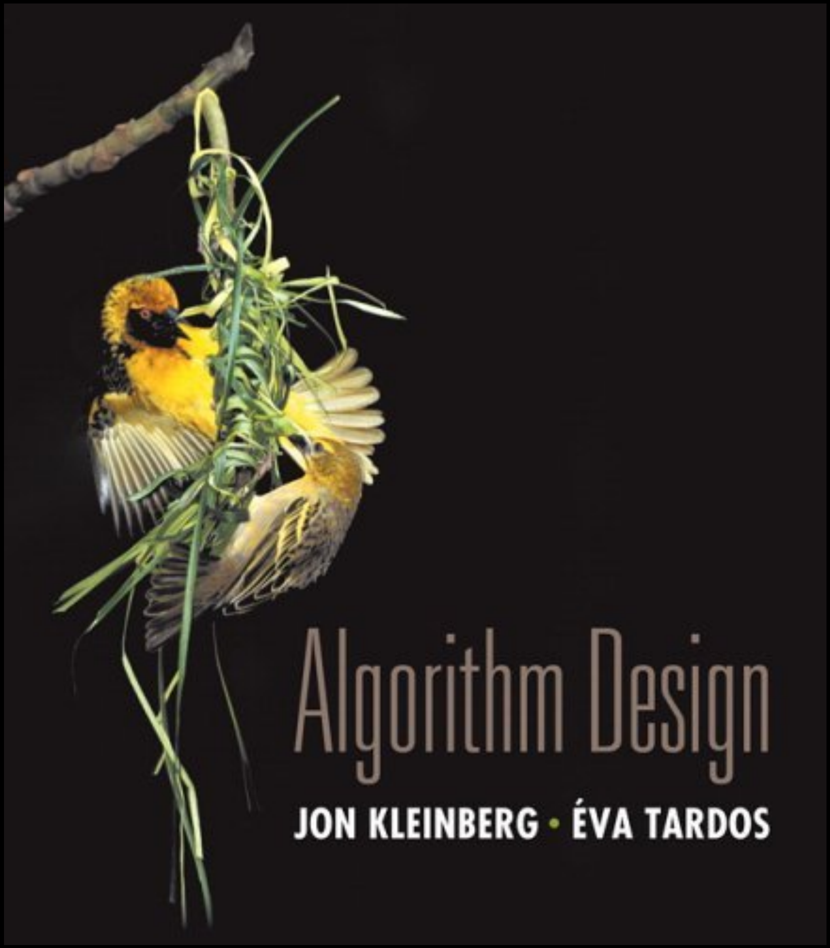


Chapter 5

Divide and Conquer

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

Sorting

Sorting. Given n elements, rearrange in ascending order.

Applications.

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

obvious applications

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

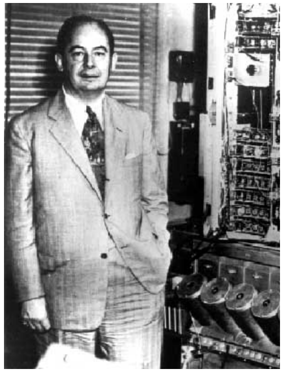
- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.

. . .

non-obvious applications

Mergesort

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



Jon von Neumann (1945)

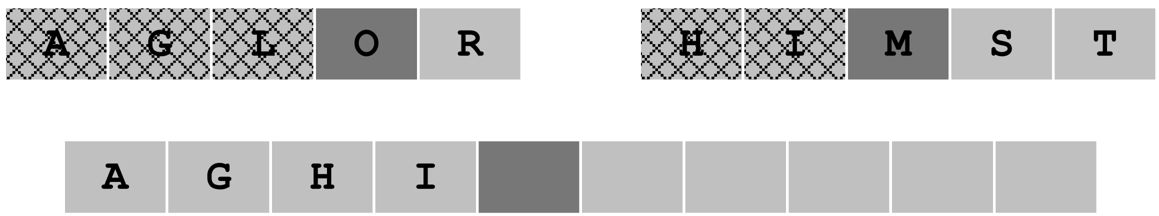
A L G O R I T H M S											
A L G O R	I T H M S	divide	$O(1)$								
A G L O R	H I M S T	sort	$2T(n/2)$								
A G H I L M O R S T		merge	$O(n)$								

Merging

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

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.

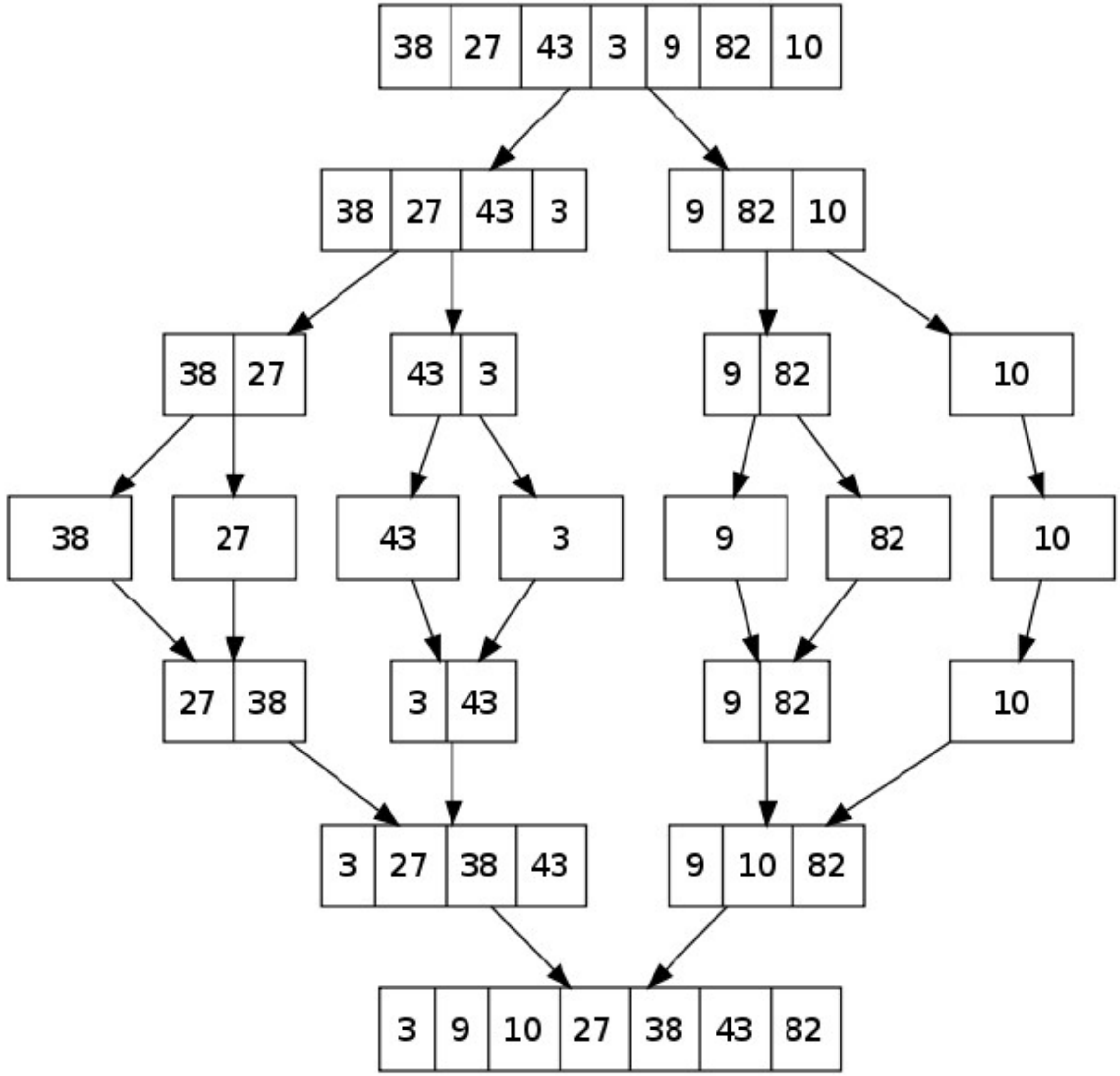


Challenge for the bored. In-place merge. [Kronrud, 1969]

using only a constant amount of extra storage

DEMO
(05-demo-merge)

Merge Sort



38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

A Useful Recurrence Relation

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

Mergesort recurrence.

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

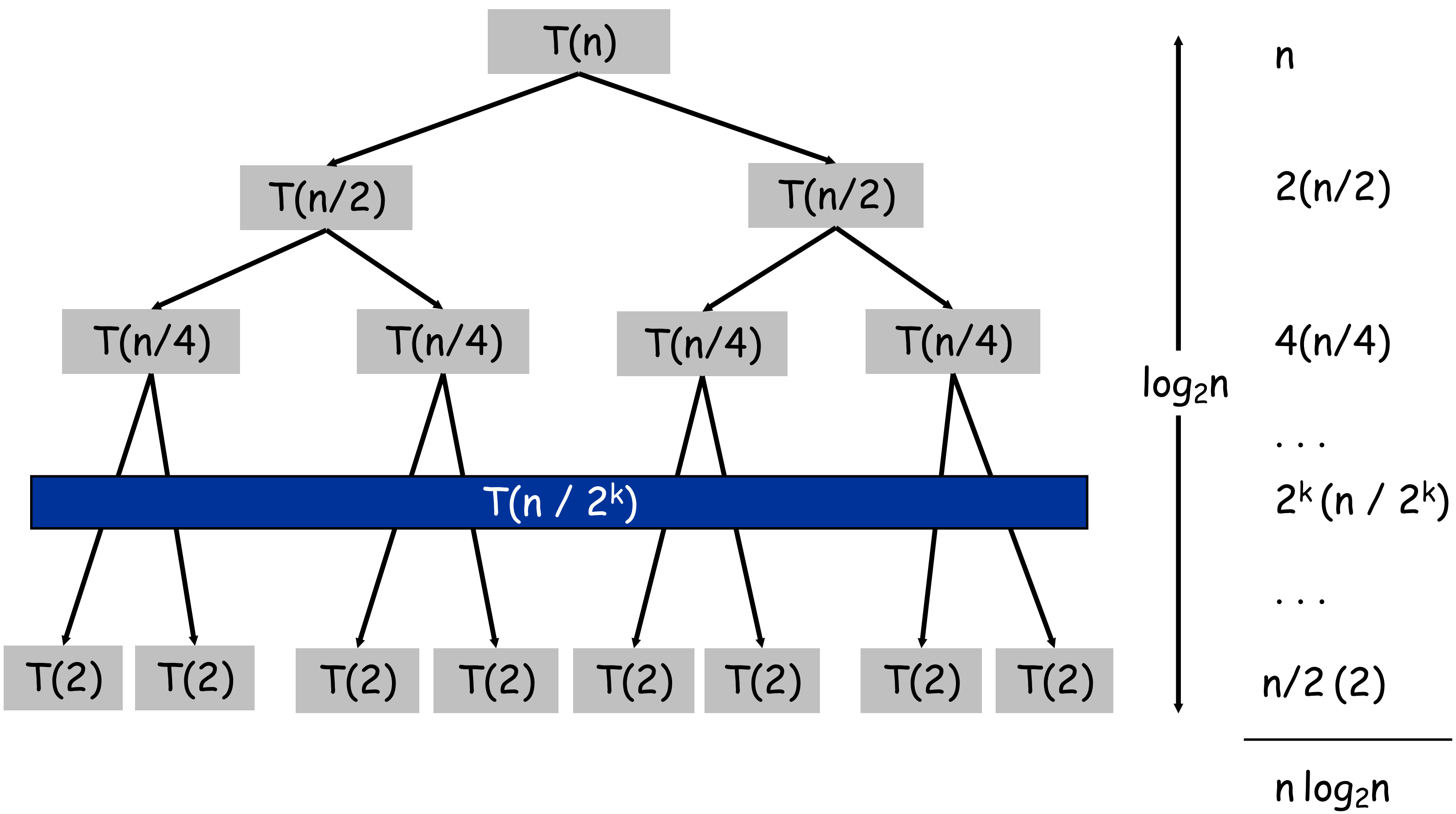
Solution. $T(n) = O(n \log_2 n)$.

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

Proof by Recursion Tree

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

assumes n is a power of 2



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:

$$\begin{aligned}
\frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\
&= \frac{T(n/2)}{n/2} + 1 \\
&= \frac{T(n/4)}{n/4} + 1 + 1 \\
&\cdots \\
&= \frac{T(n/n)}{n/n} + \underbrace{1 + \cdots + 1}_{\log_2 n} \\
&= \log_2 n
\end{aligned}$$

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 $T(n) \le n \lceil \lg n \rceil$.

$\log_2 n$

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

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.

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

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

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: $a_1, a_2, \ldots, a_n$.
- 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.

- 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).

Observations

- In the worst case, there are quadratic number of inversions ($O(n^2)$).
  - E.g., consider a list sorted in descending order.
- So to improve upon $O(n^2)$ bound asymptotically (e.g., $O(n \log n)$) an algorithm must count inversions without ever looking at each inversion individually.
- Key "combine" Idea:

  The cross-inversions between the two sorted halves A and B are precisely due to pairs

  $(a_i, b_j), (a_{i+1}, b_j), \ldots$ $\quad A \times B$ where $a_i > b_j$.

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

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

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

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

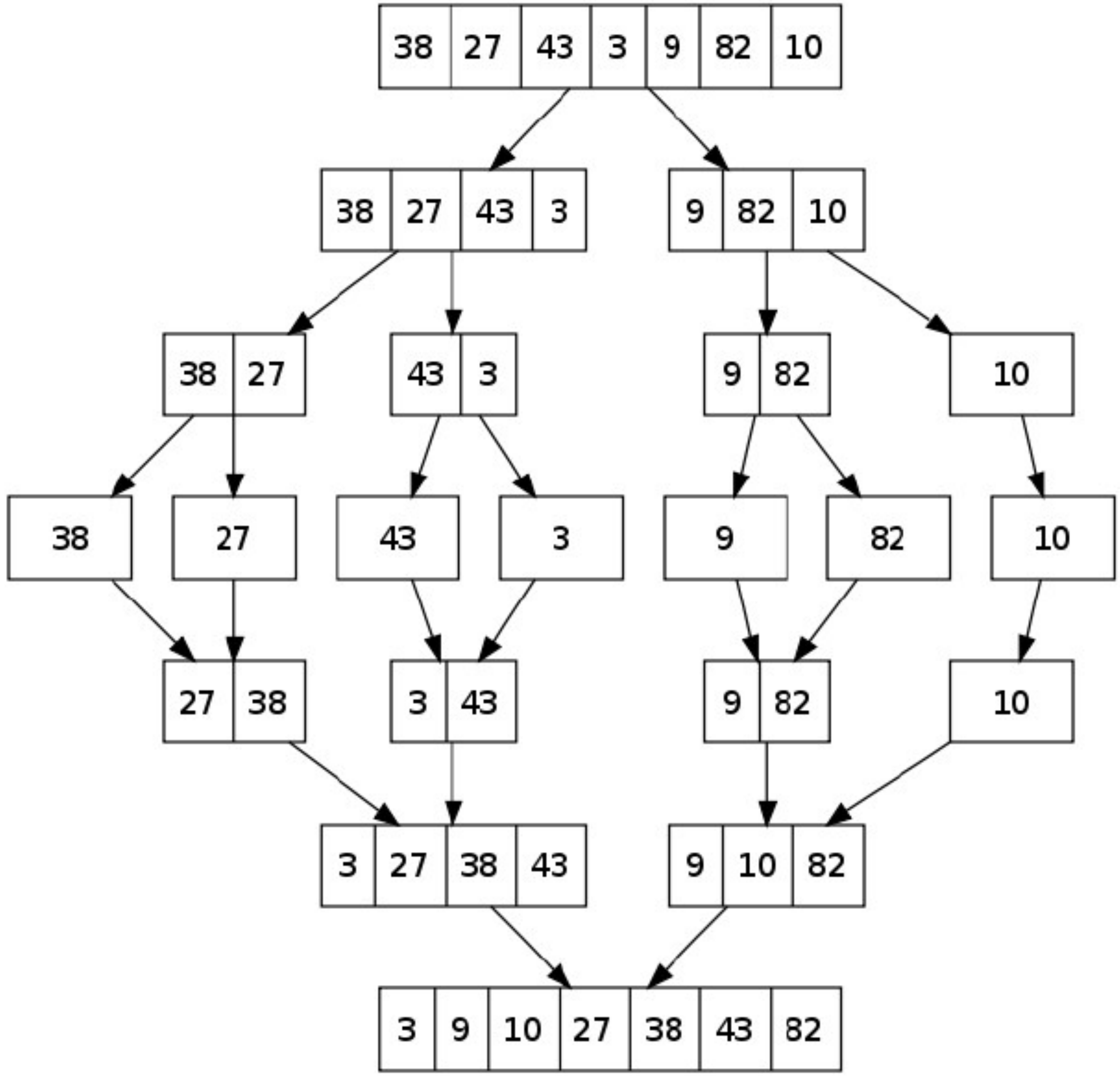
Merge: O(n)

$$T(n) \le T\left(\lfloor n/2 \rfloor\right) + T\left(\lceil n/2 \rceil\right) + O(n) \quad \Rightarrow \quad \mathrm{T}(n) = O(n \log n)$$

DEMO

(05demo-merge-invert)

Merge Sort


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Counting Inversions: Implementation

Pre-condition. [Sort-and-Count] A and B are sorted.

Post-condition. [Merge-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 , L) ← Merge-and-Count(A, B)

   return r = rA + rB + 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.

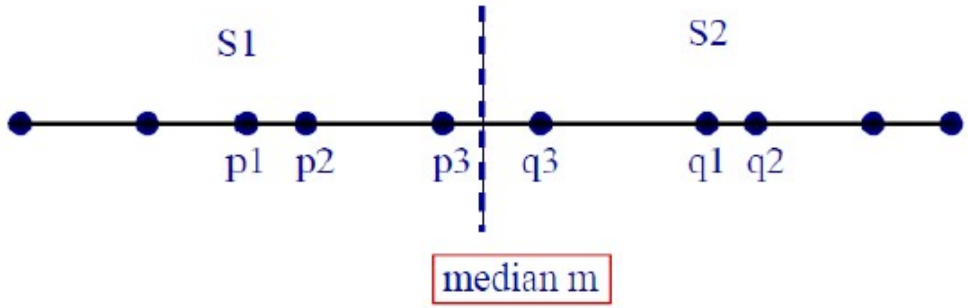
Assumption. No two points have same x coordinate.

to make presentation cleaner

1D version : Divide and Conquer

- Divide the points S into two sets S_1, S_2 by some x-coordinate so that $p < q$ for all $p \in S_1$ and $q \in S_2$.

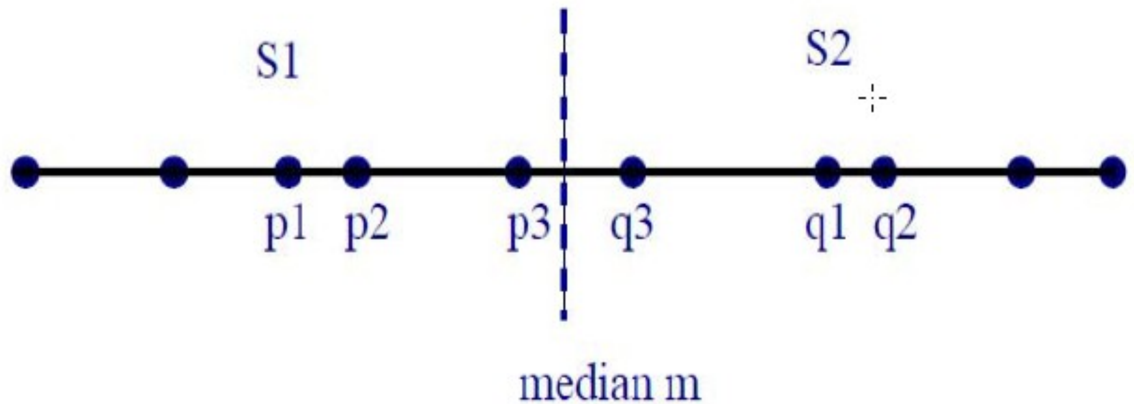
- Recursively compute closest pair (p_1, p_2) in S_1 and (q_1, q_2) in S_2.



- Let δ be the smallest separation found so far:

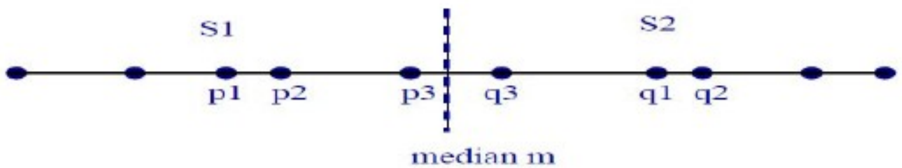
$$\delta = \min(|p_2 - p_1|,\ |q_2 - q_1|)$$

1D version : Divide and Conquer



- **The closest pair is $\{p_1, p_2\}$, or $\{q_1, q_2\}$, or some $\{p_3, q_3\}$ where $p_3 \in S_1$ and $q_3 \in S_2$.**
- **Key Observation: If m is the dividing coordinate, then p_3, q_3 must be within δ of m.**

1D version : Divide and Conquer



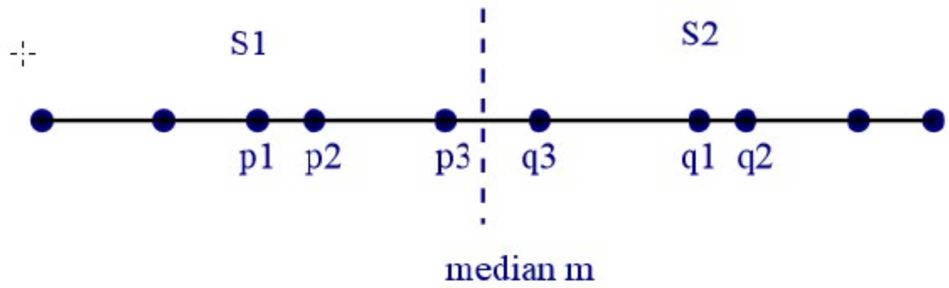
How many points of S_1 can lie in the interval $(m-\delta, m]$?

By definition of δ, at most one. Same holds for S_2.

So, we have just one pair (p3, q3) to check!

Note, problem decomposition (median computation) takes linear time and solution composition (max-min computation) takes linear time.

Recurrence is $T(n) = 2T(n/2) + O(n)$, which solves to $T(n) = O(n \log n)$.



- **Closest-Pair** (S).
- **If** $|S| = 1$, **output** $\delta = \infty$.
  If $|S| = 2$, **output** $\delta = |p_2 - p_1|$.
  Otherwise, do the following steps:

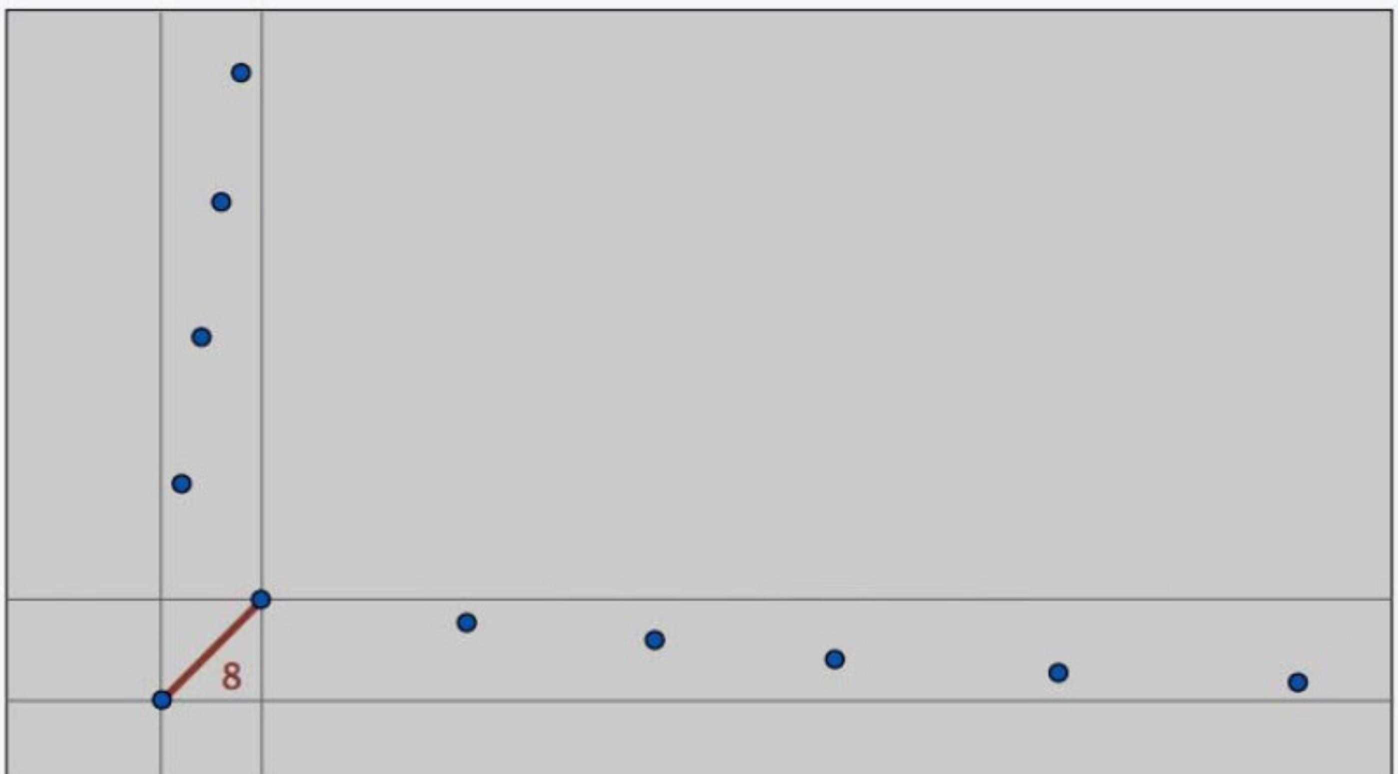
  1. **Let** $m = median(S)$.
  2. **Divide** S **into** S_1, S_2 **at** m.
  3. $\delta_1 =$ **Closest-Pair**(S_1).
  4. $\delta_2 =$ **Closest-Pair**(S_2).
  5. δ_{12} **is minimum distance across the cut.**
  6. **Return** $\delta = \min(\delta_1, \delta_2, \delta_{12})$.

- **Recurrence is** $T(n) = 2T(n/2) + O(n)$, **which solves to** $T(n) = O(n \log n)$.

Closest pair of points: first attempt

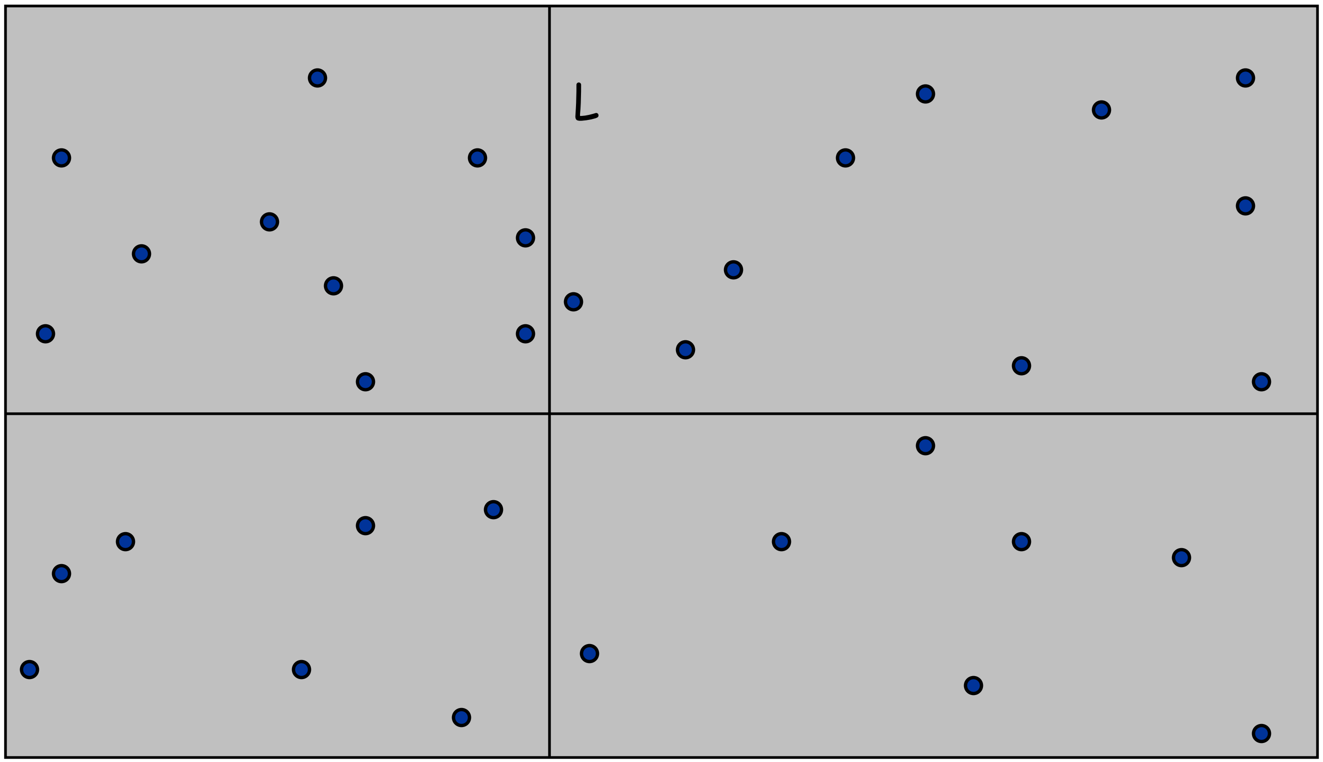
Sorting solution.

- Sort by x-coordinate and consider nearby points.
- Sort by y-coordinate and consider nearby points.



Closest Pair of Points: First Attempt

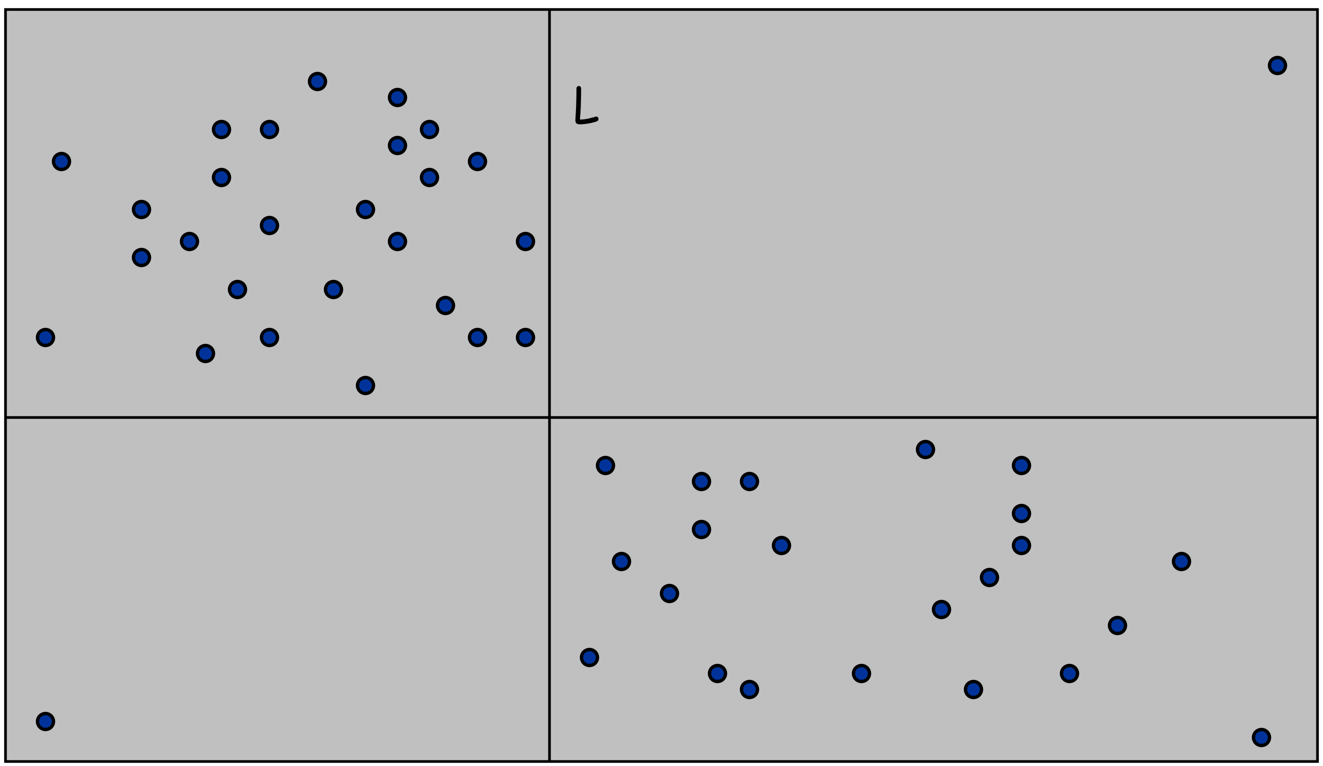
Divide. Sub-divide region into 4 quadrants.



Closest Pair of Points: First Attempt

Divide. Sub-divide region into 4 quadrants.

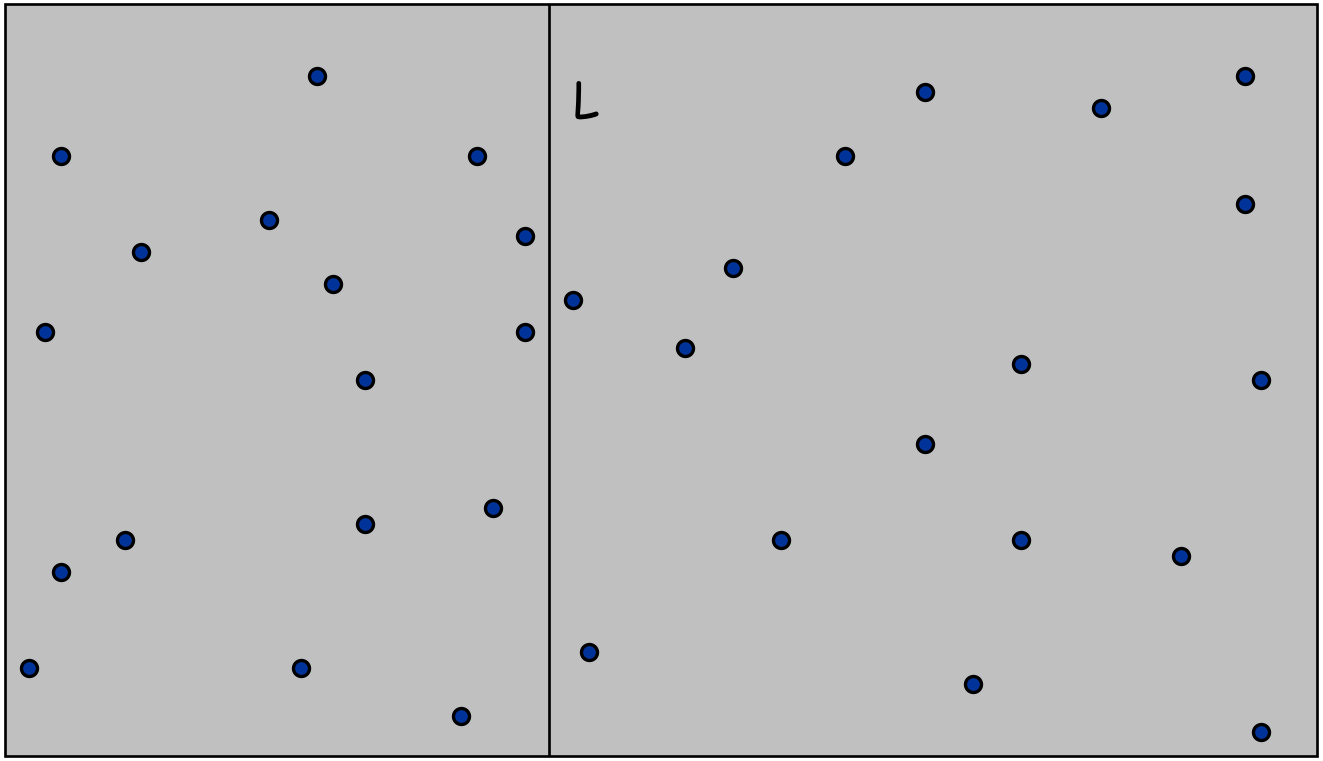
Obstacle. Impossible to ensure n/4 points in each piece.



Closest Pair of Points

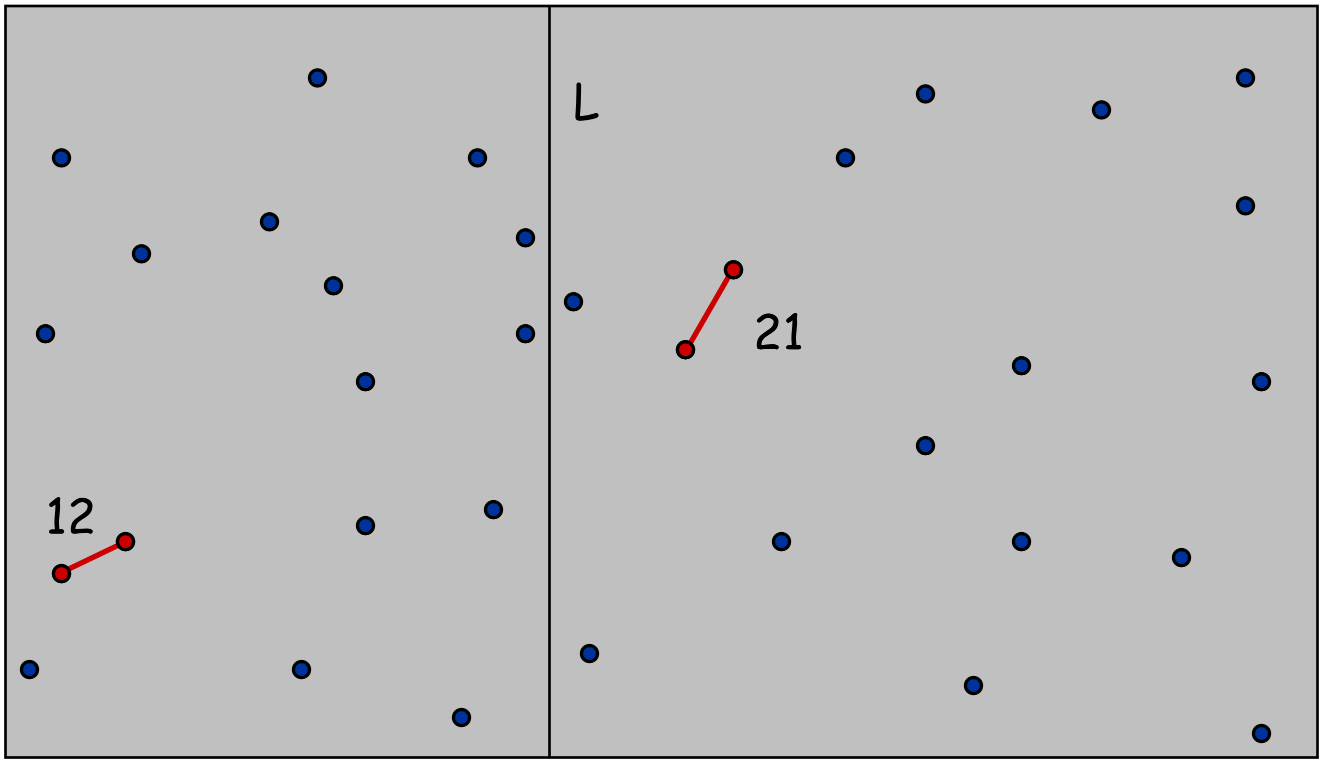
Algorithm.

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



Closest Pair of Points

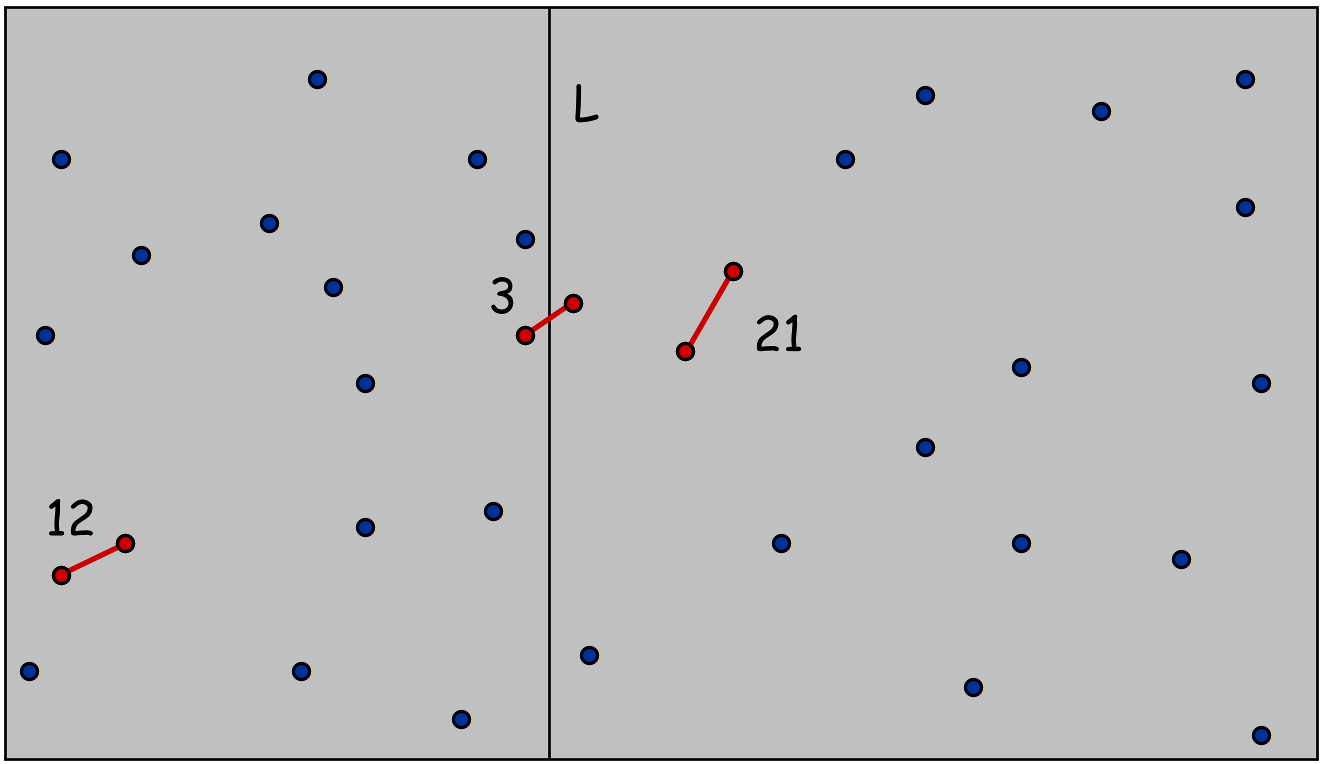
Algorithm.
- Divide: draw vertical line L so that roughly $\frac{1}{2}$n points on each side.
- Conquer: find closest pair in each side recursively.



Closest Pair of Points

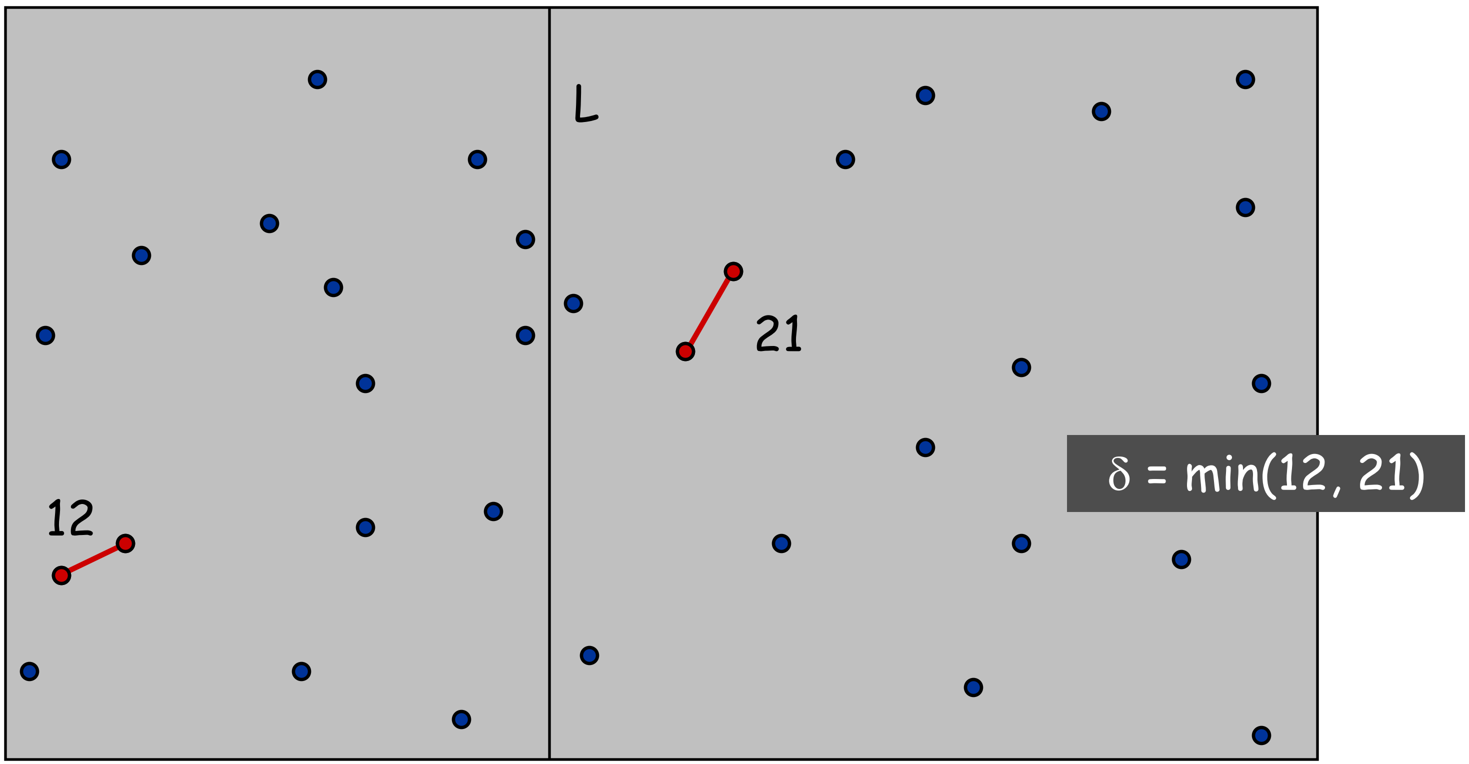
Algorithm.

- 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

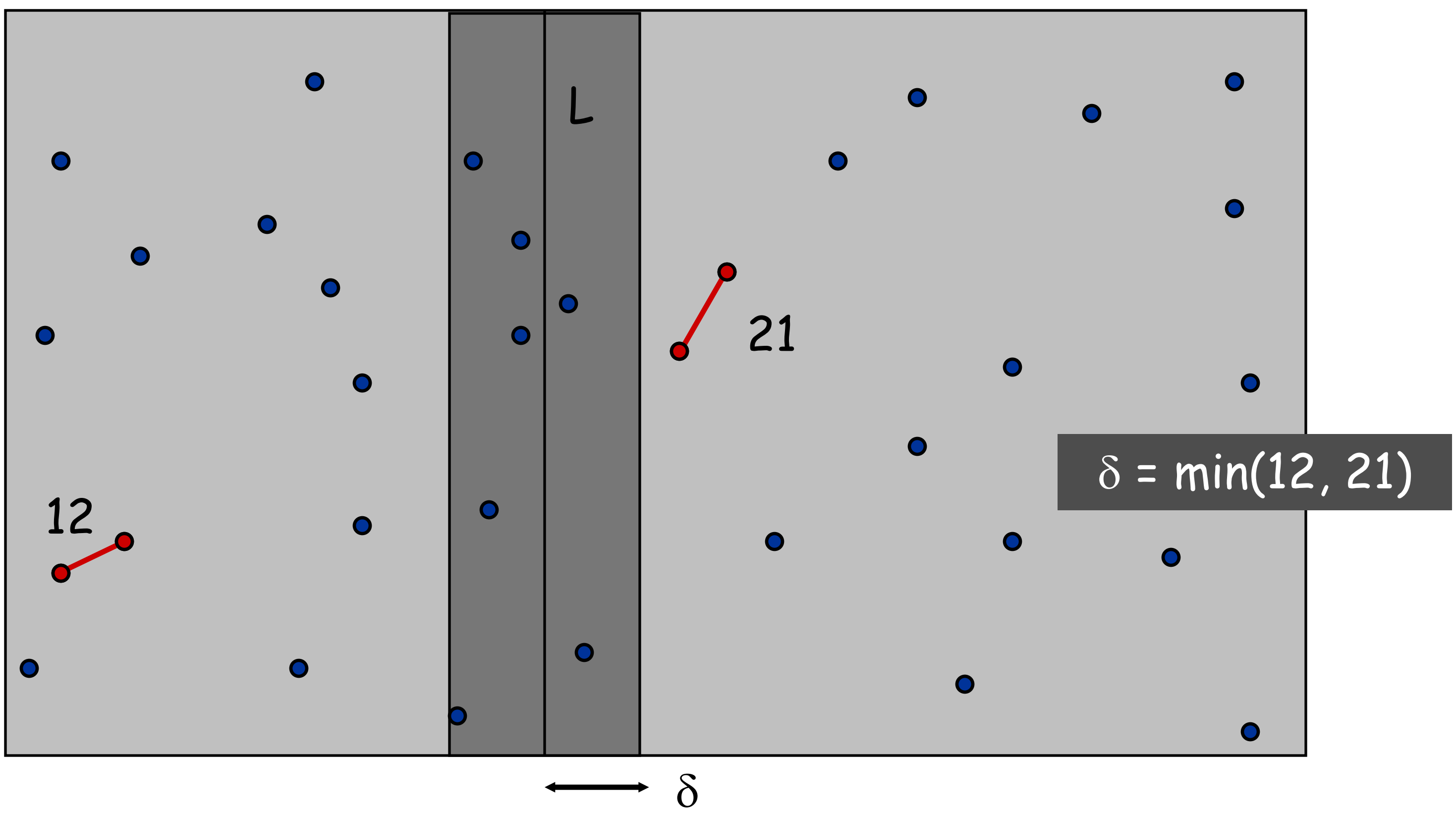
Find closest pair with one point in each side, assuming that distance < δ.



Closest Pair of Points

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

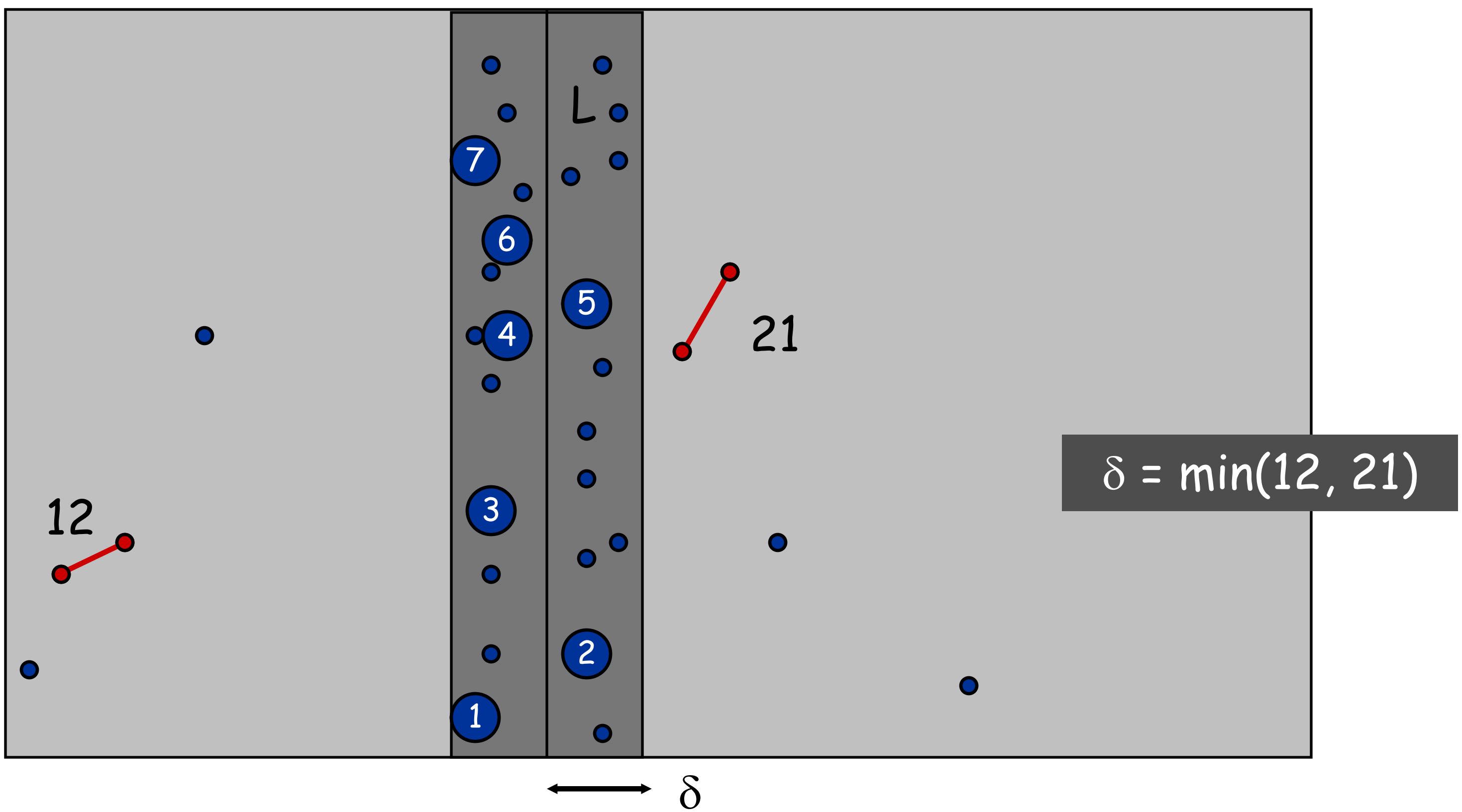
- Observation: only need to consider points within δ of line L.



Closest Pair of Points

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

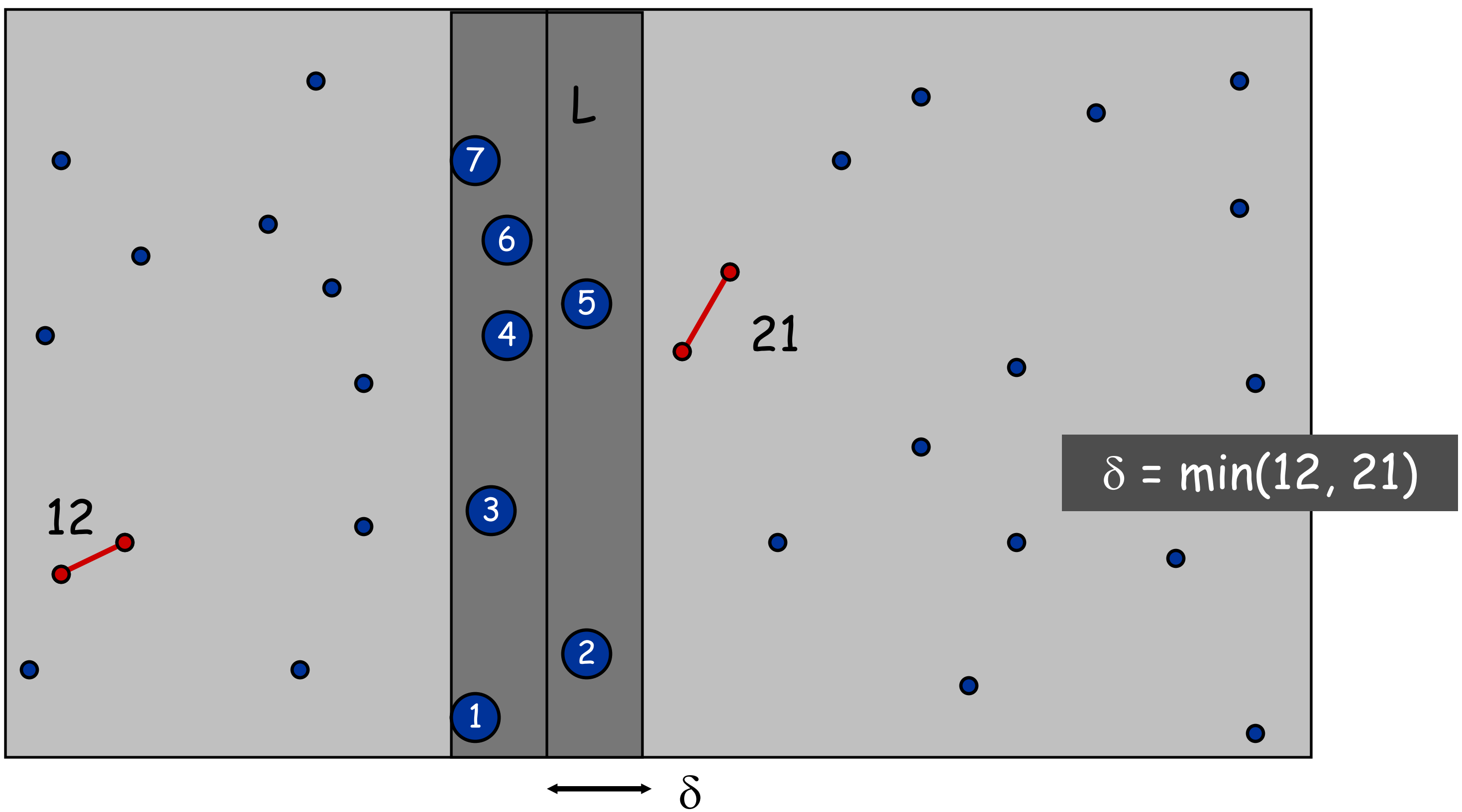
- Observation: only need to consider points within δ of line L.
- Unfortunately, this can degenerate into determining closest points among O(n/2) x O(n/2) point pairs in the worst case.



Closest Pair of Points

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

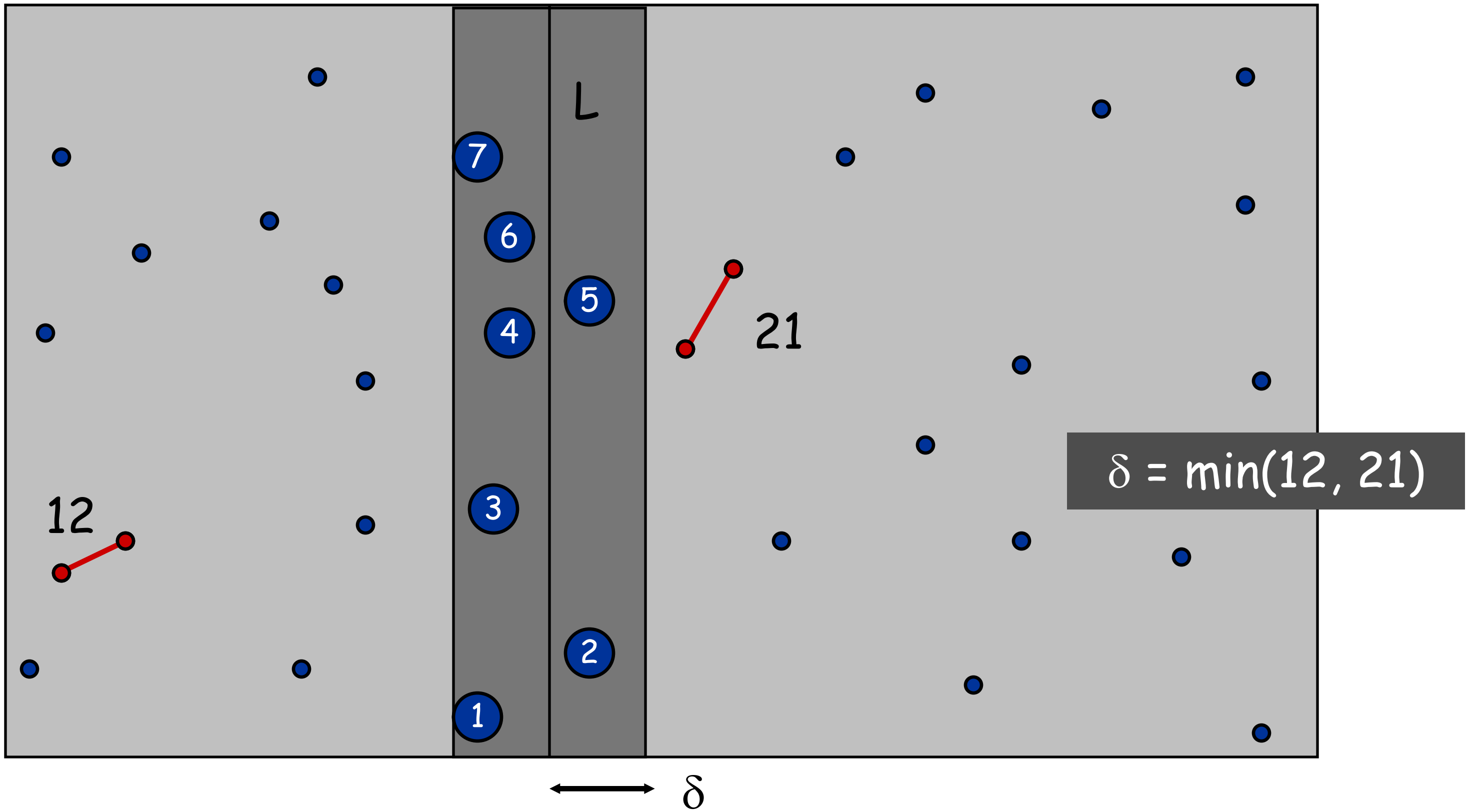
- Observation: only need to consider points within δ of line L.
- Unfortunately, this can degenerate into determining closest points among O(n/2) x O(n/2) point pairs in the worst case.



Closest Pair of Points

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

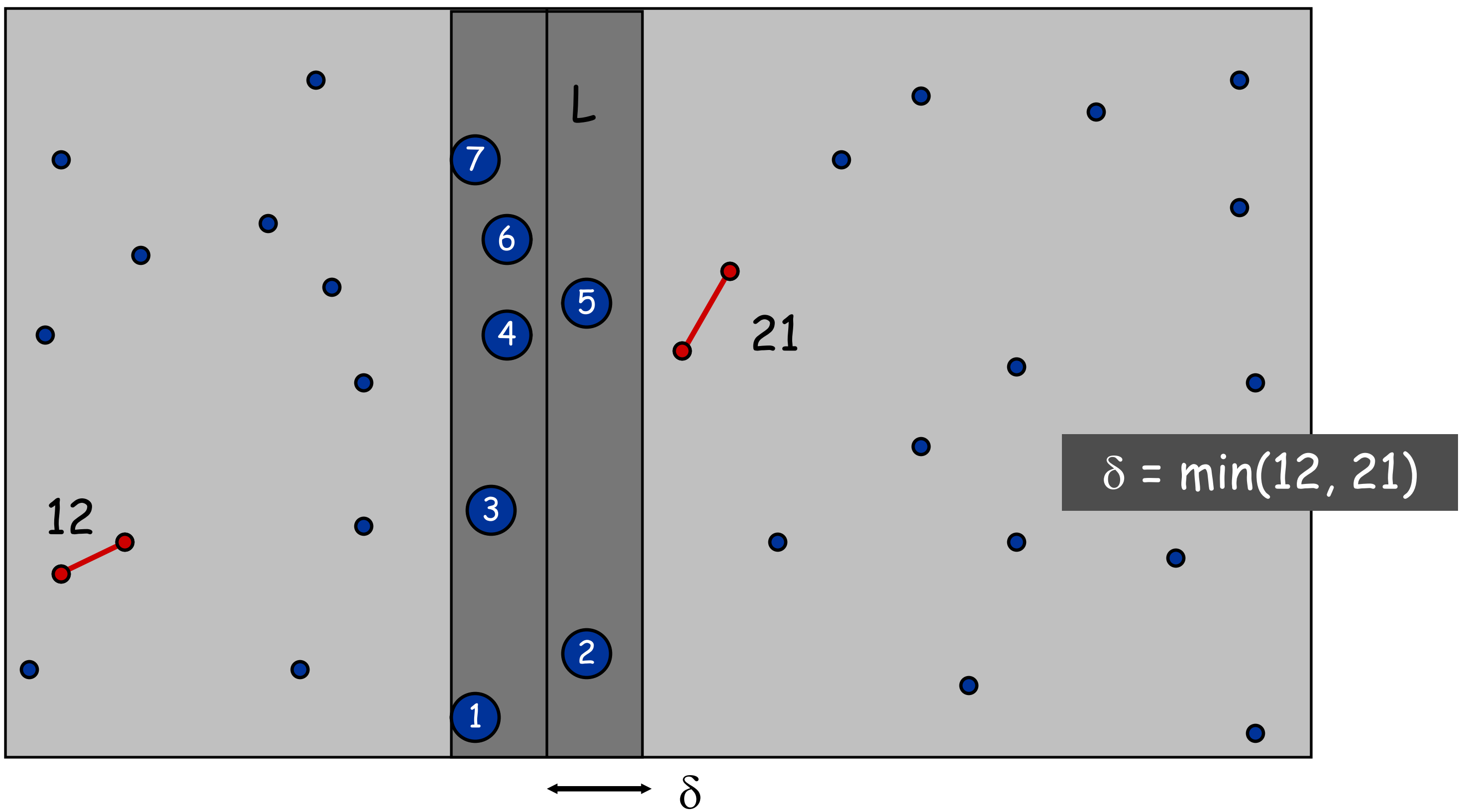
- Observation: only need to consider points within δ of line L.
- Unfortunately, this can degenerate into determining closest points among O(n/2) x O(n/2) point pairs in the worst case.
- Sort points in 2δ-strip by their y coordinate.



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.
- Key Insight: Check distances of only those points within 11 positions of each point in sorted list! (linear-time)



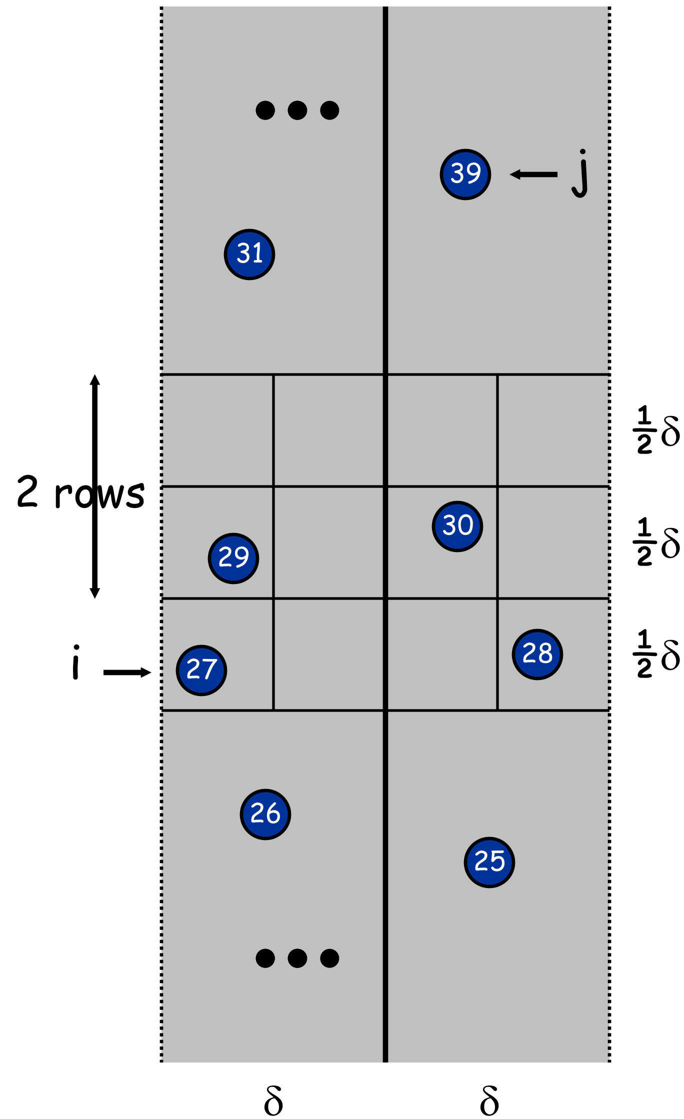
Closest Pair of Points

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.

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 6.

E.g. For pt. 28, need to check with three **squares** on the other side of the median line,

or for pt. 30, six **squares**,.

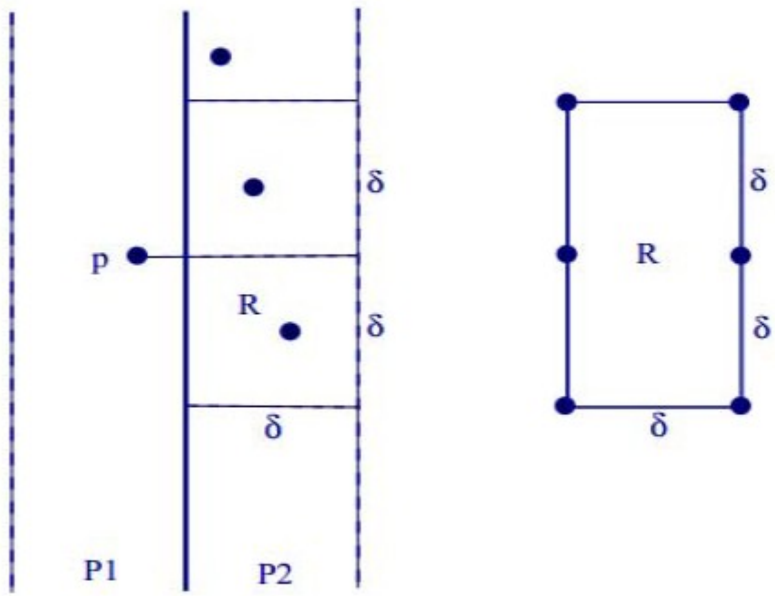
E.g., For pt. 27, check 5 **points** in 2 rows.

–(Constant time for each point x O(n)

Using y-coordinate sorted points in the 2δ-strip)

Alternate Explanation

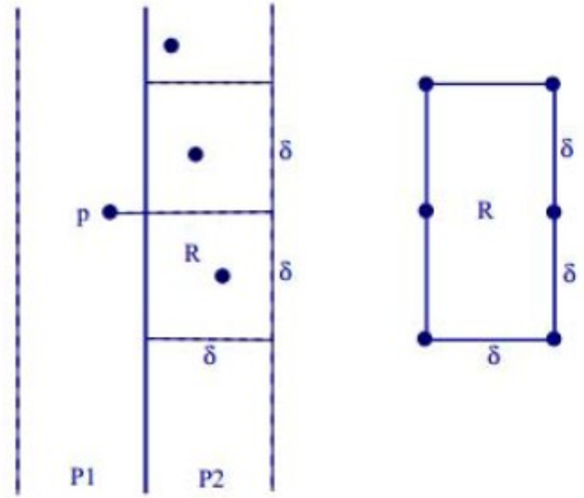
- Consider a point $p \in S_1$. All points of S_2 within distance δ of p must lie in a $\delta \times 2\delta$ rectangle R.



- How many points can be inside R if each pair is at least δ apart?
- In 2D, this number is at most 6!
- So, we only need to perform $6 \times n/2$ distance comparisons!

Alternate Explanation

- In order to determine at most 6 potential mates of p, project p and all points of P_2 onto line ℓ.



- Pick out points whose projection is within δ of p; at most six.
- We can do this for all p, by walking sorted lists of P_1 and P_2, in total $O(n)$ time.
- The sorted lists for P_1, P_2 can be obtained from pre-sorting of S_1, S_2.

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.

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

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.

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