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*

Computational Geometry: 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.

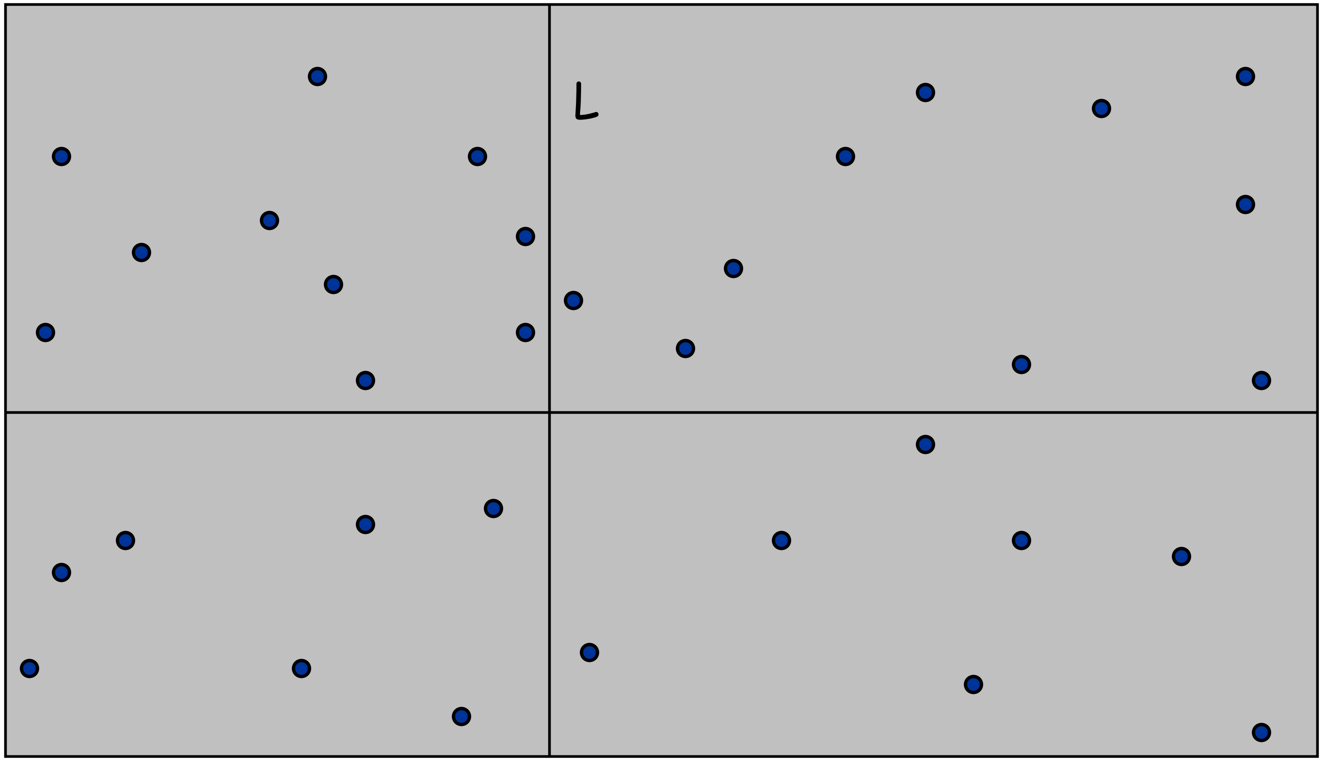
Assumption. No two points have same x coordinate (or y coordinate)

↑ Otherwise: rotate or enhance a bit our algorithm

to make presentation cleaner

Closest Pair of Points: First Attempt

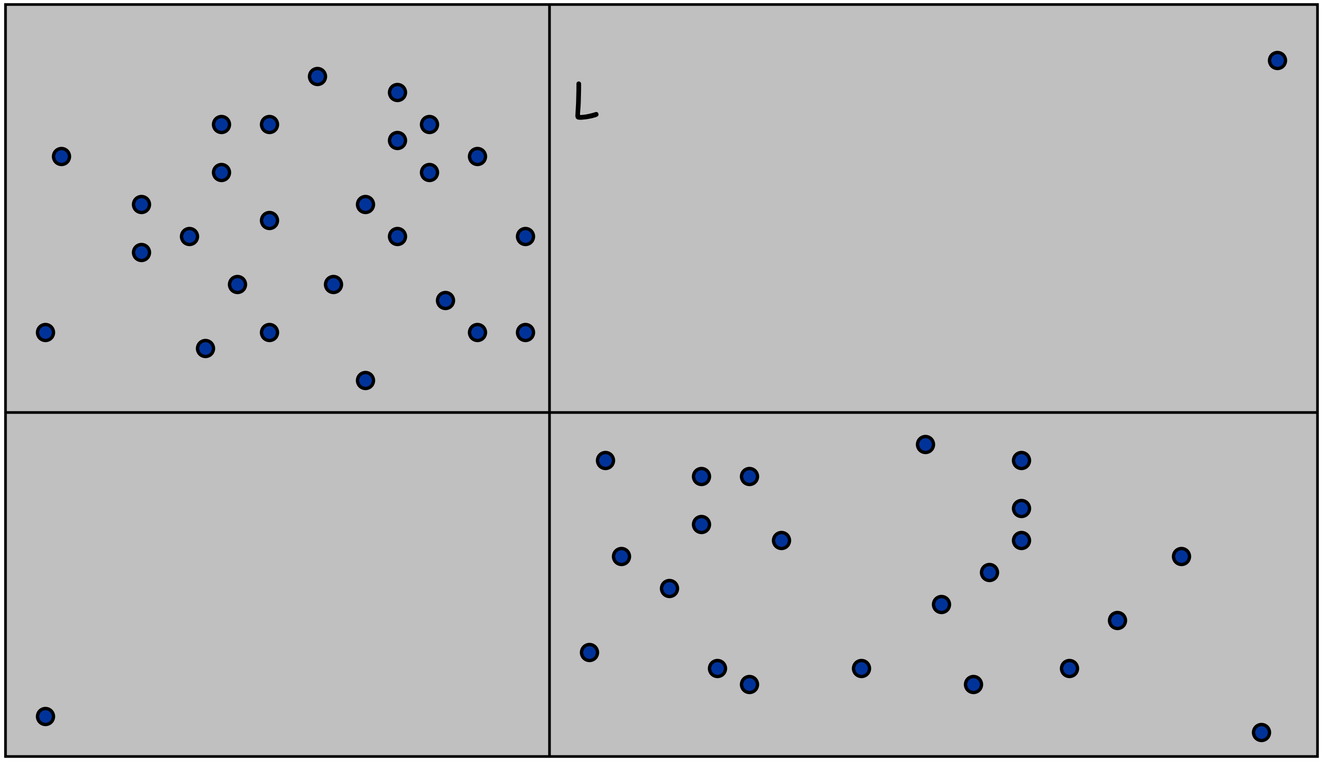
Divide. Sub-divide region into 4 quadrants.



Closest Pair of Points: First Attempt

Divide. Sub-divide region into 4 quadrants.

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



Designing the algorithm

Setting up the recursion. It would be useful if every recursive call on a set $P' \subseteq P$, beings with two lists: a list P'_x in which all the points in P' have been sorted by increasing x-coord, and a list P'_y in which all the points in P' have been sorted by increasing y-coord

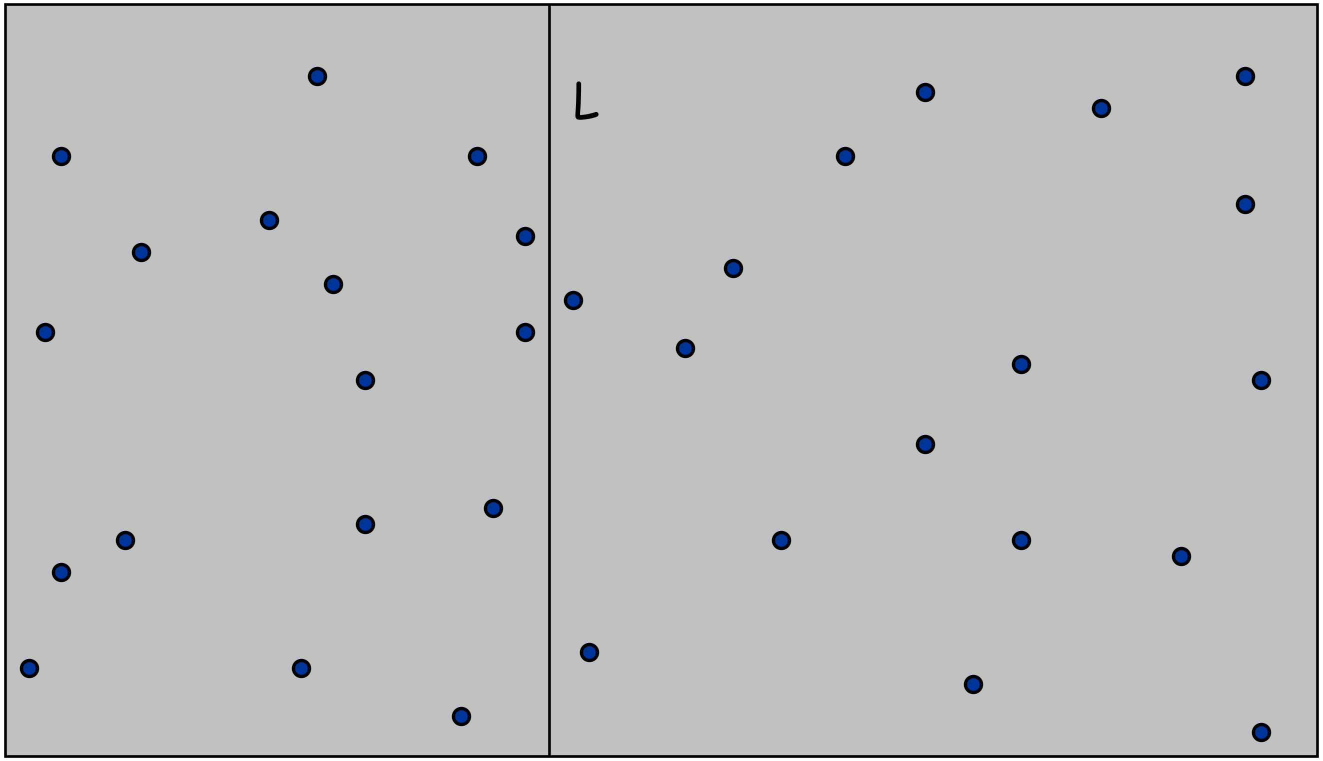
We can ensure that this remains true throughout the algorithms as follows:

- [1] Before any recursion begins, we sort all the points in P by x-coord and again by y-coord, producing lists P_x and P_y. Attached to each entry in each lists is a record of the position of that point in both lists
- [2] The first level of recursion works as follows (with all further levels working completely analogously): We define Q to be the set of points in the first *ceil*(n/2) positions of the list P_x (the "left" half) and R to be the set of points in the final *floor*(n/2) positions of the list P_x (the "right" half)

Closest Pair of Points

Algorithm.

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



Designing the algorithm

Setting up the recursion (cont'ed). By a single pass through each P_x and P_y in O(n) time, we can create the following four lists:

- [i] Q_x consisting of the points in Q sorted by increasing x-coord
- [ii] Q_y consisting of the points in Q sorted by increasing y-coord
- [iii] R_x consisting of the points in R sorted by increasing x-coord
- [iv] R_x consisting of the points in R sorted by increasing y-coord

For each entry of each of these lists, as before, we record the position of the point in both lists it belongs to

We recursively determine a closest pair of points in Q (with access to the lists Q_x and Q_y)

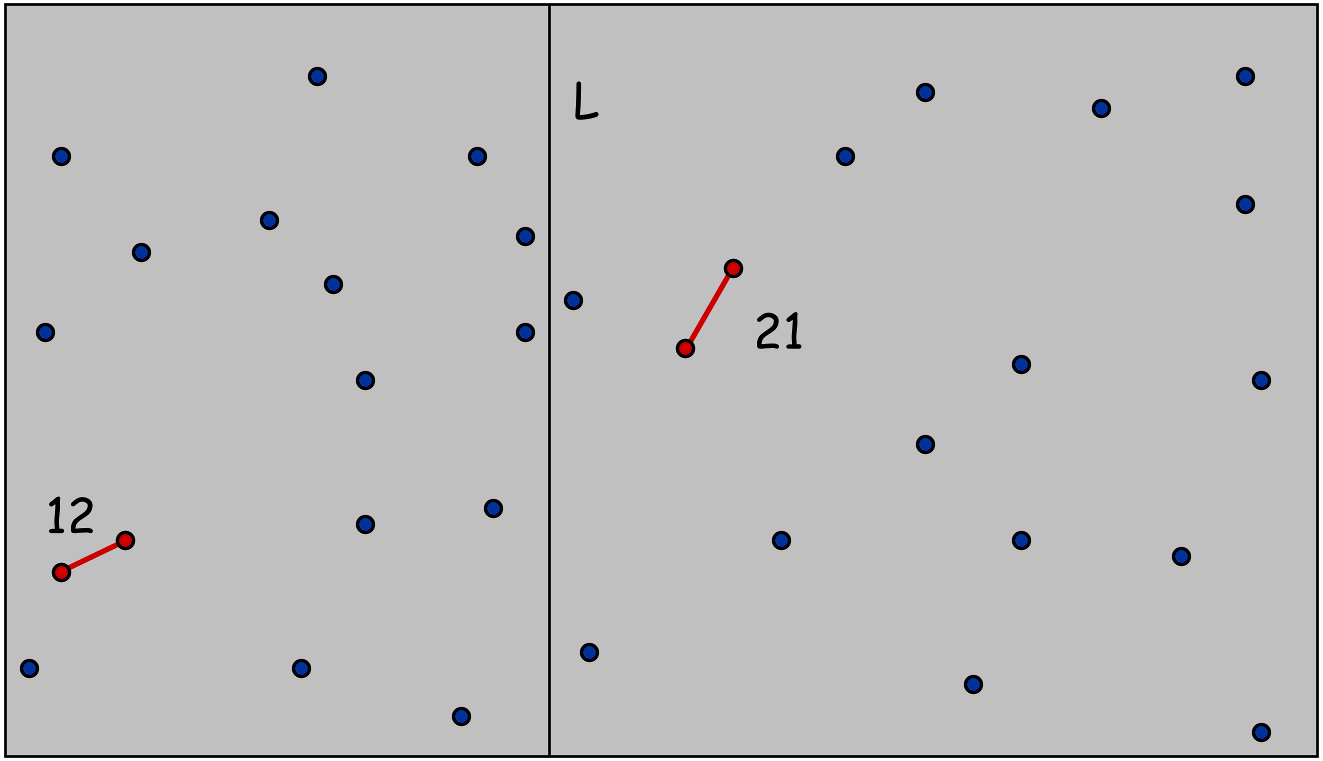
Suppose that q^*_0 and q^*_1 are (correctly) returned as a closest pair of points in Q

Similarly, we determine a closest pair of points in R, obtaining r^*_0 and r^*_1

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.



Designing the algorithm

Combining the solutions. Let δ be the minimum of $d(q^*_0, q^*_1)$ and $d(r^*_0, r^*_1)$

The real question is: Are there points $q \in Q$ and $r \in R$ for which $d(q,r)<\delta$?

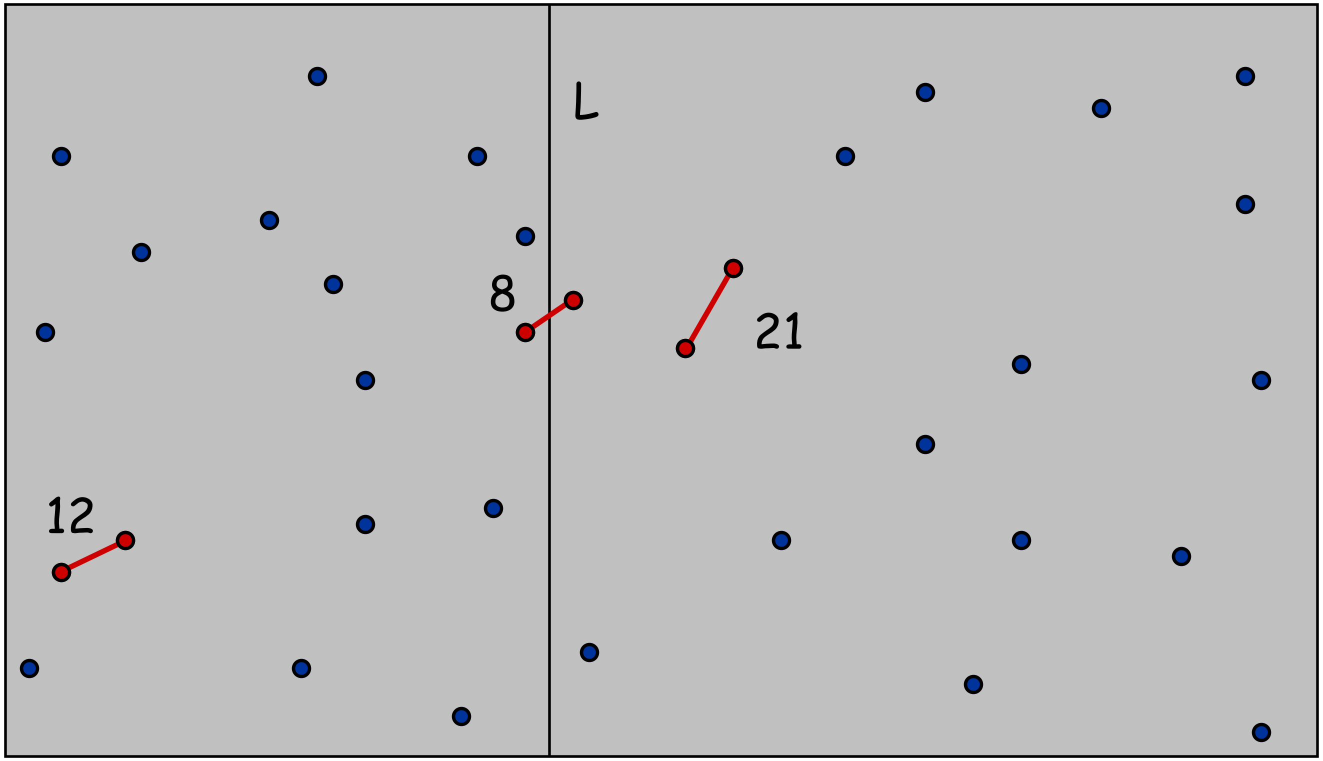
If not, then we have already found the closest pair in one of our recursive calls

But if there are, then the closest such q and r form the closest pair in P

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.



Designing the algorithm

Combining the solutions.

Let x^* denote the x-coord of the rightmost point in Q, and let L denote the vertical line described by the equation $x= x^*$

This line “separates” Q from R. Here is a simple fact:

Corollary. If there exists $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$, then each of q and r lies within a distance δ of L

So, if we want to find a close q and r, we can restrict our search to the narrow band consisting only of points in P within δ of L

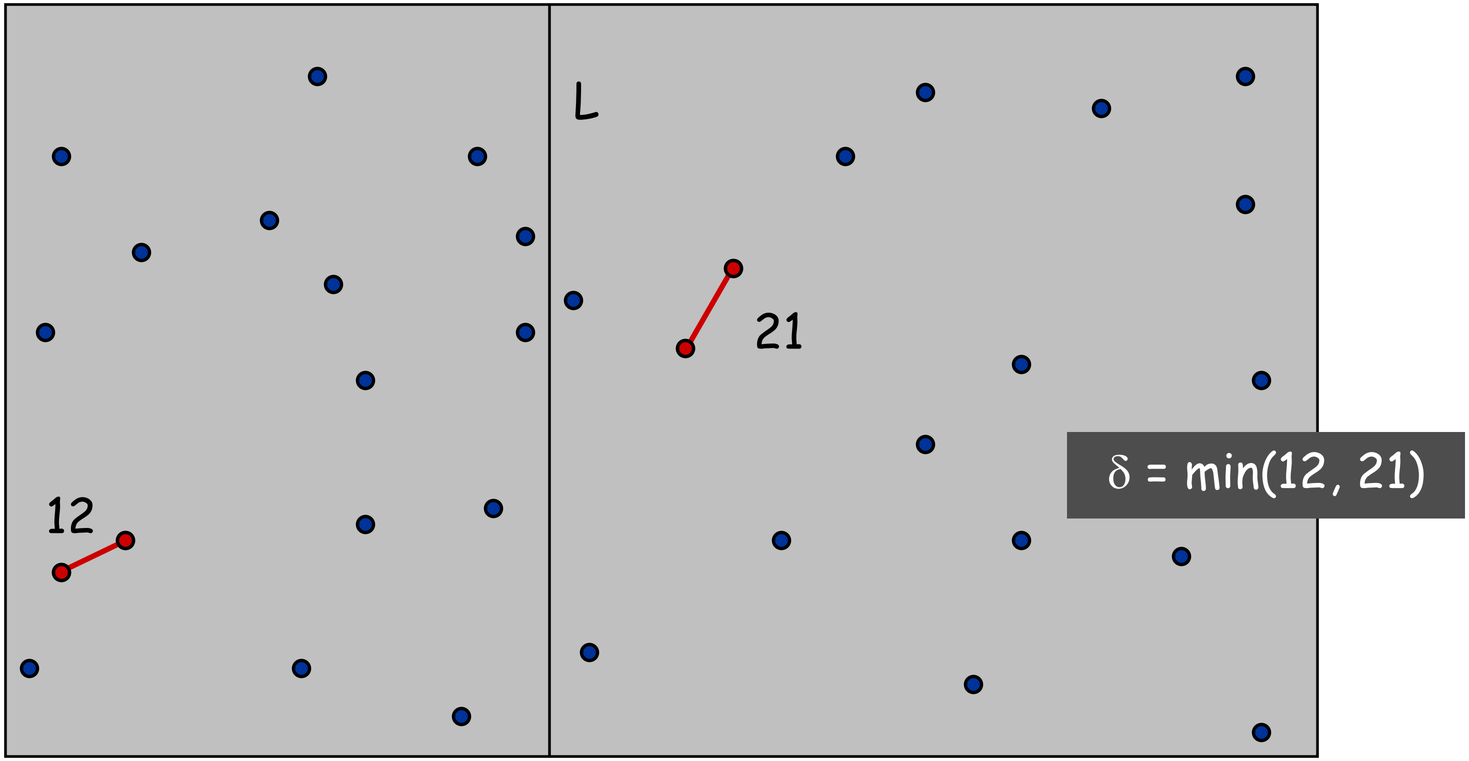
Let $S \subseteq P$ denote this set, and let S_y denote the list consisting of the points in S sorted by increasing y-coord

By a single pass through the list P_y, we can construct S_y in $O(n)$ time

Corollary (restated). There exist $q \in Q$ and $r \in R$ for which $d(q,r) < \delta$, if and only if there exist $s,s' \in S$ for which $d(s,s') < \delta$

Closest Pair of Points

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

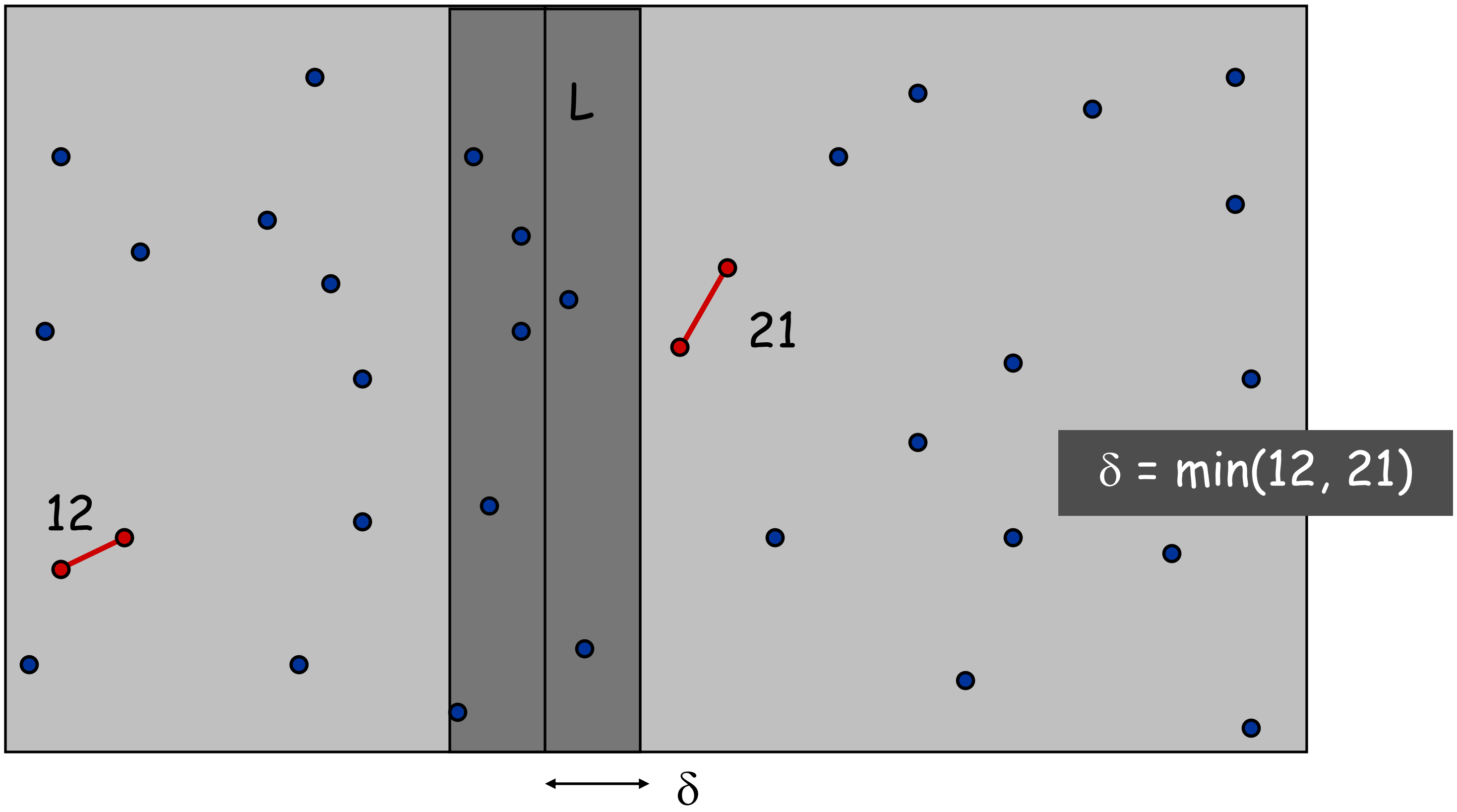


δ = min(12, 21)

Closest Pair of Points

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

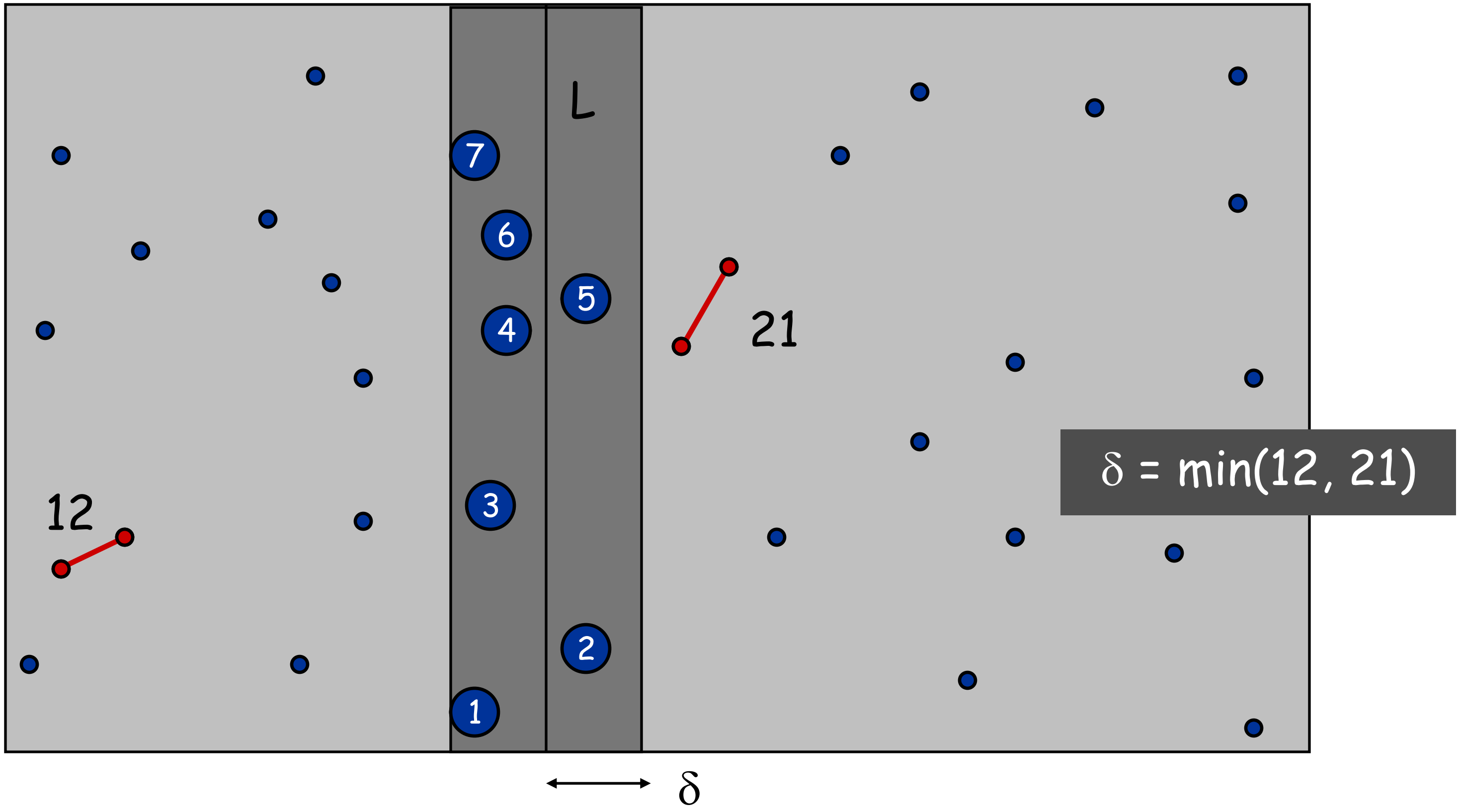
- Observation: only need to consider points within δ of line L.



Closest Pair of Points

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

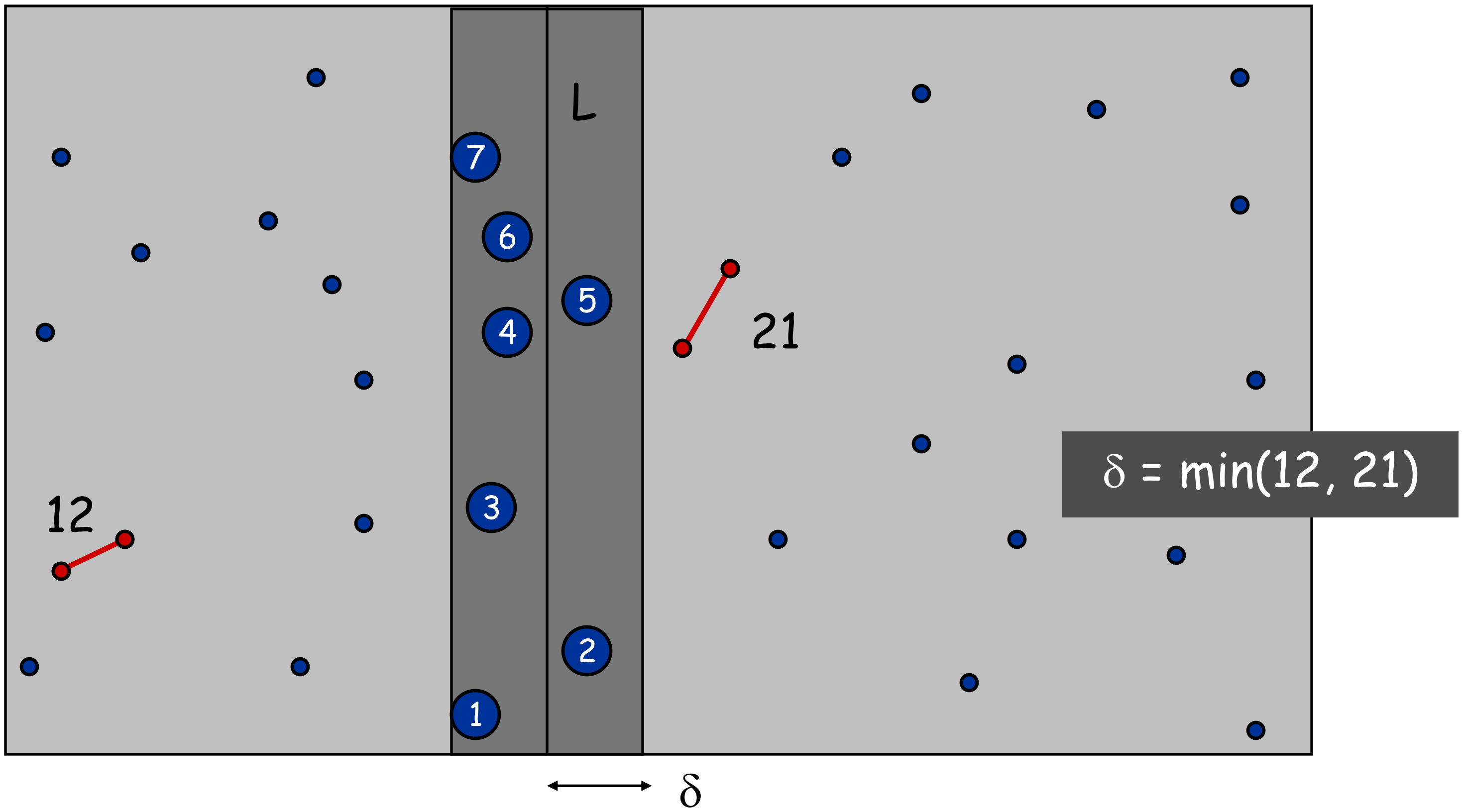
- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.



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!



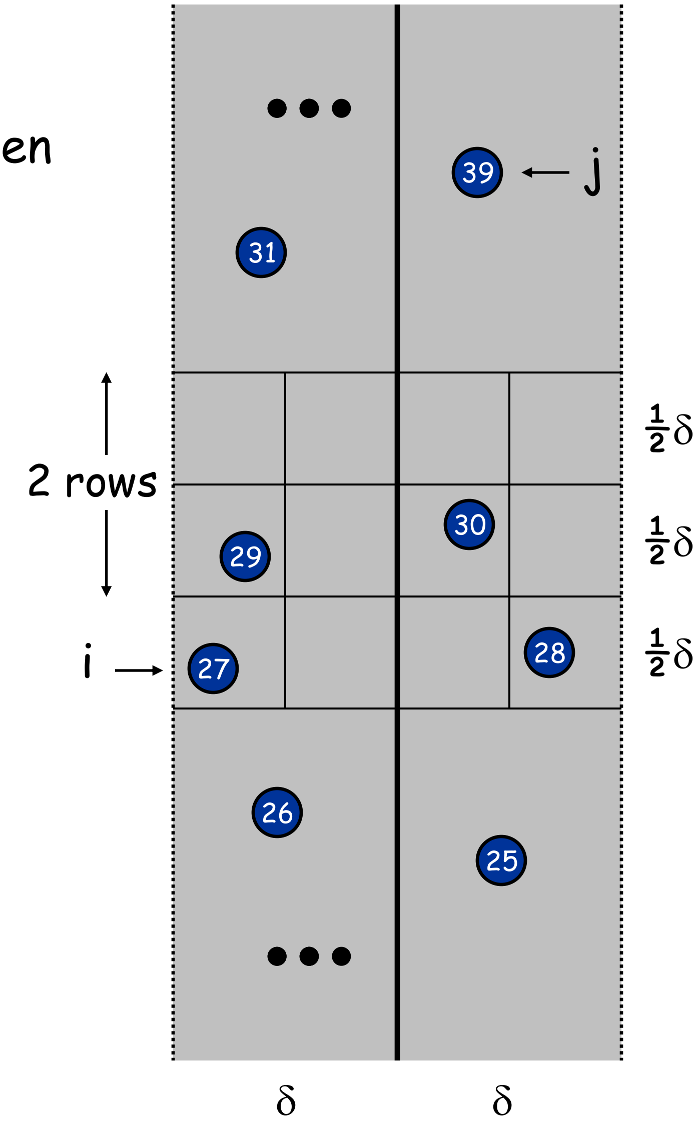
Closest Pair of Points

Def. Let s_i be the point in the 2δ-strip, with the 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 7.

Designing the algorithm

Finalizing the algorithm.

We make one pass through S_y, and for each $s \in S$, we compute the distance to each of the next 11 points in S.

The Restated Corollary implies that in doing so, we will have computed the distance of each pair of points in S (if any) that are at distance less than δ from each other.

So having done this, we can compare the smallest such distance to δ, and we can report one of two things:

- [i] the closest pair of points in S, if their distance is less than δ, or
- [ii] the (correct) conclusion that no pairs of points in S are within δ of each other

In case [i], this pair is closest pair in P,

In case [ii], the closest pair found by our recursive calls is the closest pair in P

Closest Pair Algorithm

```
Closest-Pair($p_1$, …, $p_n$)
    Compute separation line L such that half the points
    are on one side and half on the other side, i.e.,           $O(n \log n)$
    construct $P_x$ and $P_y$
    ($p_0^*$,$P_1^*$)= Closest-Pair-Rec($P_x$, $P_y$)

Closest-Pair-Rec($P_x$, $P_y$){
    if |P| $\le$ 3 then check pairwise distance, return;
    Construct $Q_x$, $Q_y$, $R_x$, $R_y$                         $O(n)$
    $\delta_1$ = Closest-Pair-Rec(left half)                     $2T(n / 2)$
    $\delta_2$ = Closest-Pair-Rec(right half)
    $\delta$  = min($\delta_1$, $\delta_2$)
    $x^*$= max x-coord of a point in set Q
    L= {(x,y) : x=$x^*$}

    S= points in P within distance $\delta$ of L                 $O(n)$

    Construct $S_y$, i.e., sort remaining points by y-coord      $O(n)$

    Scan points in y-order and compare distance between
    each point and next 11 neighbors. If any of these            $O(n)$
    distances is less than $\delta$, update $\delta$.

    return $\delta$.}
```

Closest Pair of Points: *Analysis*

Running time.

$$T(n) \le 2T\big(n/2\big) + O(n \log n) \;\Rightarrow\; T(n) = O(n \log^2 n)$$

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.

$$T(n) \le 2T\big(n/2\big) + O(n) \;\Rightarrow\; T(n) = O(n \log n)$$