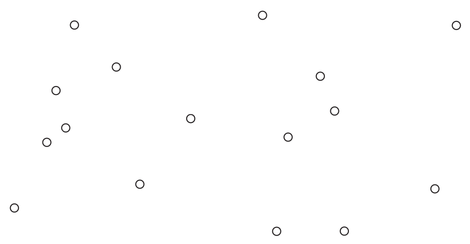
DS 320 Algorithms for Data Science **Lecture #11**
Spring 2026 Prof. Kira Goldner and Jeffrey Considine

Divide & Conquer II: Closest Pair of Points

The Problem

Your input for the *closest pair of points* problem is a set P of n points in $\mathbb{R}^2$. Assume that all of the x and y coordinates are distinct. The goal is to output a pair of points p_1 and p_2 minimizing the Euclidean L_2 distance $d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.



This problem extremely common: in graphics, computer vision, robotics, scientific simulation, etc.

As we mentioned last time, Divide & Conquer is usually the way to come up with a *more efficient* algorithm for a problem which already has a polynomial brute force solution. What's the **naïve algorithm** here and what is its **running time**?

Just compare all pairs of points: $O(n^2)$.

Step 1: Define your recursive subproblem.

Hint: An idea similar to mergesort works here.

In preprocessing, sort P by both x and y coordinate. Call these P_x and P_y. Divide the points in half by x coordinate, where the first (left) half is L and the second (right) half is R.

The dividing line h will go through the rightmost point in L.

Step 2: Combine the solutions to your subproblems.

Given the solutions (closest pair) from your subproblems (make sure the parameters make sense), how do you combine or compare these solutions to get the solution to our problem of size n?

The closest pair will be one of: (1) the closest pair from L, (2) the closest pair from R, or (3) the closest pair across the halves. The solution to our subproblems will return the first two, but how do we find the solution to (3) in order to compare these?

Computing the Closest Pair Across Halves

Idea: Narrow down the set of points we need to search.

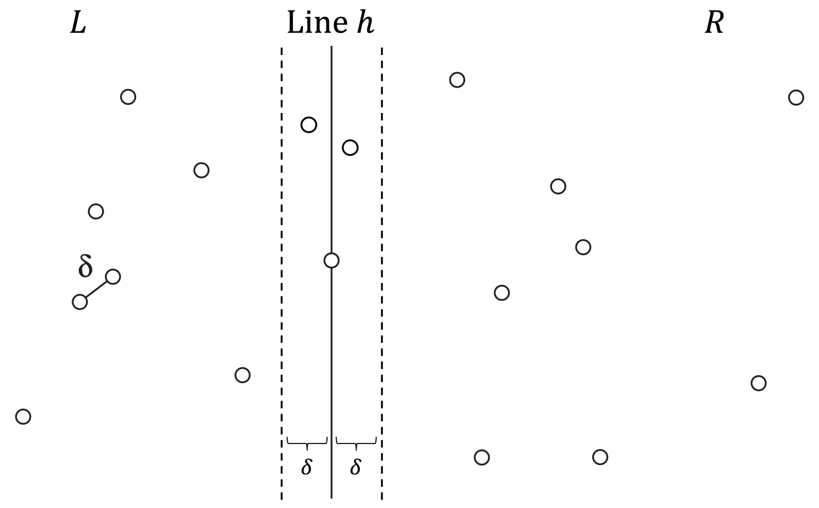
Lemma 1. *For subproblem solutions of closest pairs $\ell_0^*, \ell_1^* \in L$ and $r_0^*, r_1^* \in R$, let*

$$\delta = \min\{ \quad d(\ell_0^*, \ell_1^*) \quad , \quad d(r_0^*, r_1^*) \quad \}.$$

If $\ell \in L$ and $r \in R$ and $d(\ell, r) \leq \delta$, then ℓ and r are within δ of h.

Proof.

Suppose some $\ell \in L$ and $r \in R$ and $d(\ell, r) \leq \delta$. Then the distance in the x-coordinates $|x_\ell - x_r|$ must also be less than δ. If we call x_h the x-coordinate of the line, then by definition of h, $x_\ell \leq x_h < x_r$, hence it must also be the case that these distances $|x_\ell - x_h|$ and $|x_h - x_r|$ are at most δ, that is, that r and ℓ fall within δ of the line h.



Definition 1. Let $S = \{p \in P : d(p, h) < \delta\}$ be the set of points within distance δ of the line h.

Note: We can compute S in linear time.

Lemma 1 implies that we only need to search S. Problem: Brute-force over S is $O(n^2)$ still.

Idea: Brute force more intelligently.

Lemma 2. *Let S_y be a list of the points in S sorted by y-coordinate and let (s, s') be closest pair of points in P. If $s, s' \in S$, then they cannot lie more than 15 positions apart in S_y.*

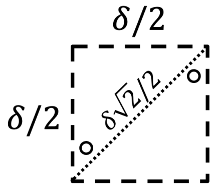
(There are less than $15n$ such pairs, hence linear time to brute force!)

Proof.

a. Divide S into boxes of side-length $\delta/2$.

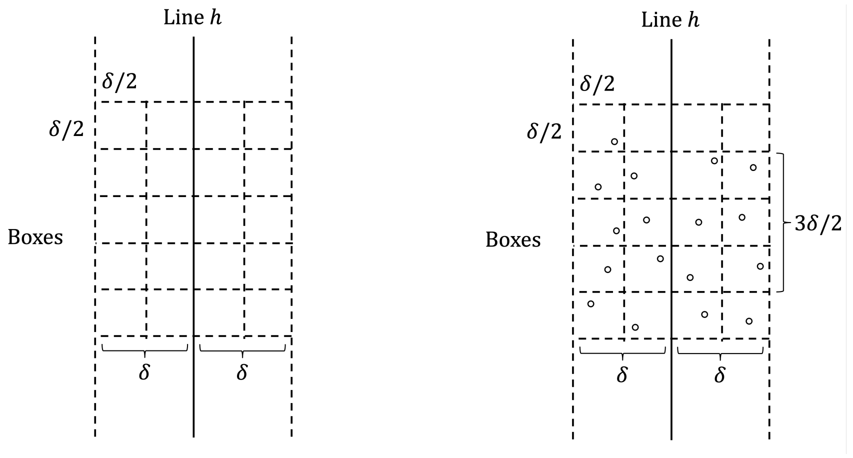
b. Show: No two points can share a box.

If they did, they're on the same side and they're closer than δ apart, which is a contradiction to what we know about the closest pairs in L and R.



c. Show: If s and s' are at least 16 positions apart in S_y, then they're not the closest pair of points.

If s and s' are at least 16 positions apart in the list S_y (which is the list of points in S, the points within δ-distance of h, sorted by y-coordinate), then they must be at least 3 rows of boxes apart. By part (b) we know that there cannot be more than one point in a box, so even if the points are packed as closely as possible, they must each lie in their own box, and thus two points that are 15 positions apart by y-coordinate, even when most densely packed, must have at least three full rows of boxes between them. Then the distance between these points $d(s, s')$ must be at least $3\delta/2 > \delta$, and hence these points are not closer than the closest pair in one of the halves.



Then we know that for a pair of points across halves (in S) to possibly be the closest pair, they must be within 15 positions of each other when sorted by y-coordinate. We simply check for closest pairs within these proximity, and there are only $15 \cdot |S| \leq 15n$ such pairs, not n^2, so it takes linear time to brute force to check if any are closest.

The Algorithm

Preprocessing: Generate P_x and P_y.

Algorithm 1 closest(P_x, P_y).

```
Input: Array $P_x$ of points in $P$ sorted by $x$ coordinate; array $P_y$ sorted by $y$ coordinate.
if $|P| \leq 3$ then                              // Base Case
    return closest pair by brute force
end if
Construct $L_x$, $L_y$, $R_x$, $R_y$              // Subproblem: Recursive calls on halves
$(\ell_0^*, \ell_1^*)$ = closest($L_x$,$L_y$)
$(r_0^*, r_1^*)$ = closest($R_x$,$R_y$)
Construct $S_y$
$(s_0^*, s_1^*)$ = closest pair in $S_y$ within 15 spots of each other
return closest of $(\ell_0^*, \ell_1^*)$, $(r_0^*, r_1^*)$, $(s_0^*, s_1^*)$
```

Runtime

$T(n) = a\,T(n/b) + O(f(n))$ $\qquad a =$ $\qquad b =$ $\qquad f(n) =$

$\Rightarrow T(n) =$

$T(n) = 2\,T(n/2) + O(n)$

$\Rightarrow T(n) = n \log n.$

Proof of Correctness

Proof. We prove the correctness by induction on the size of P.

The base case of $|P| \leq 3$ is clear by the algorithm.

IH: For sizes smaller than $|P|$, the closest pairs are computed correctly by recursion.

IS: By Lemma 1, the remainder of the algorithm correctly determines whether any pair of points in S is at distance less than δ, and if so, returns the closest such pair. Now the closest pair in P either has both elements in one of L or R, or it has one element in each. In the former case, the closest pair is correctly found by the recursive call (IH). In the latter case, this pair is at distance less than δ, and it is correctly found by the remainder of the algorithm. $\square$