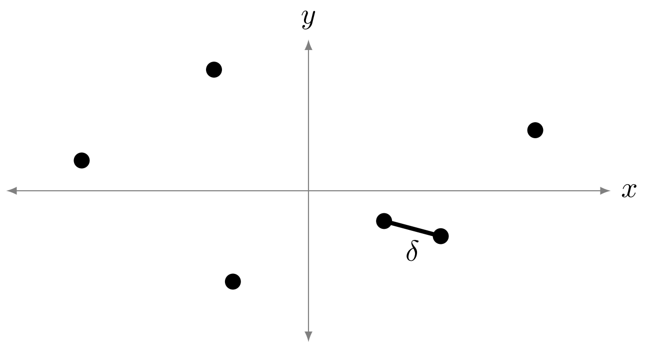
CS 341: ALGORITHMS (S18) — LECTURE 5
CLOSEST PAIR OF POINTS

ERIC BLAIS

You have already seen how Divide & Conquer can be used to sort lists (with MergeSort) and to do fast multiplication. Today, we explore how the same technique can also be used to solve problems in the completely different setting of computational geometry. One of the most fundamental problems in the area is the following.

Definition 5.1 (Closest pair)**.** An instance of the *Closest Pair problem* is a set of n points $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R} \times \mathbb{R}$ on the plane. A valid solution to such an instance is the distance $\delta = \min_{1 \le i < j \le n} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ between the closest pair of points in the set.



Algorithms for the closest pair problem are used as a building block for many other problems, in computational geometry and beyond. But the naïve brute-force algorithm for the problem, which computes the distance between every pair of points in the set, has time complexity $\Theta(n^2)$. Our goal today is to find a more efficient algorithm for the same problem.

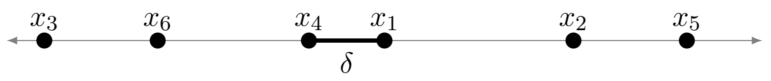
Remark 5.2. In many cases, what we want to compute is not quite the minimum distance between the closest pair of points, but rather to *identify* the pair of points that are closest to each other. By the end of the lecture, you should be able to easily modify the algorithms that we obtain to solve this variant of the problem as well.

It's not immediately obvious how we can do better than the brute-force algorithm. In situations like this, it is often extremely helpful to consider a simpler version of the problem first to build up some intuition for the problem.

1. Closest points on the line

We can easily obtain a natural simplification of the Closest Pair problem that is worth studying: instead of considering sets of points on the plane, let's consider sets of points on the (one-dimensional) line instead.

Definition 5.3 (Closest pair on the line)**.** An instance of the *Closest Pair on the Line (CPL) problem* is a set of n points $x_1, \ldots, x_n \in \mathbb{R}$ on the line. A valid solution to such an instance is the distance $\delta = \min_{1 \le i < j \le n} |x_i - x_j|$ between the closest pair of points in the set.



Note that the points $x_1, \ldots, x_n$ are not necessarily sorted. The simplest brute-force algorithm for this problem involves checking the distance between each pair of points and returning the minimum distance. This algorithm has time complexity $\Theta(n^2)$. Can we design a more efficient algorithm for the problem?

First algorithm. We can naïvely apply the Divide & Conquer method to the problem.

(1) *Divide* the points into the two smaller sets $X_1 = \{x_1, \ldots, x_{\frac{n}{2}}\}$ and $X_2 = \{x_{\frac{n}{2}+1}, \ldots, x_n\}$,
(2) *conquer* the smaller instances by using recursive calls with inputs X_1 and X_2 to find the smaller distances of pairs of points within each subset, and
(3) *combine* by returning the smallest value of (i) the minimum distance between pairs of points in X_1, (ii) the minimum distance between pairs of points in X_2, and (iii) the minimum distance between a point in X_1 and a point in X_2.

Implementing this idea leads to the following algorithm.

```
Algorithm 1: FirstCPL($x_1, \ldots, x_n$)
$\delta_1 \leftarrow$ FirstCPL($x_1, \ldots, x_{\frac{n}{2}}$);
$\delta_2 \leftarrow$ FirstCPL($x_{\frac{n}{2}+1}, \ldots, x_n$);
$\delta \leftarrow \min\{\delta_1, \delta_2\}$;
for $i = 1, \ldots, \frac{n}{2}$ do
    for $j = \frac{n}{2}+1, \ldots, n$ do
        $\delta \leftarrow \min\{\delta, |x_i - x_j|\}$;
return $\delta$;
```

We can convince ourselves without too much effort that this algorithm is correct, but is it any better than the brute-force algorithm? Unfortunately, no: the time complexity of this algorithm satisfies the recursion

$$T(n) = 2\,T(\tfrac{n}{2}) + \Theta(n^2)$$

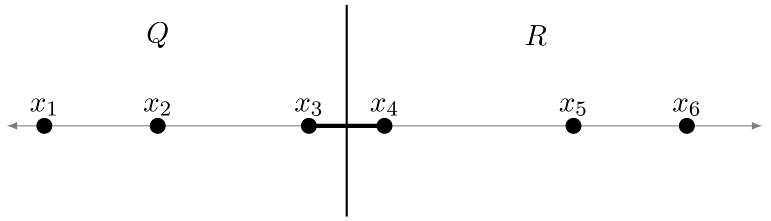
which, by the Master Theorem, is $T(n) = \Theta(n^2)$.

Better algorithm. The main problem with the naïve divide & conquer algorithm is that it takes too much work to do the *combine* step, since it still requires the computation of $\Theta(n^2)$ distances. And since the sets X_1 and X_2 have no structure, we can't hope to do better unless we find a better way to *divide* the input. And there is a better way: divide the points according to their position in space. This can be done efficiently if we first sort the list so that $x_1 \le x_2 \le \cdots \le x_n$. Then we have the following approach:

(1) (Initially sort the list of points by position,)
(2) *divide* the sorted list of points into the two smaller sets $Q = \{x_1, \ldots, x_{\frac{n}{2}}\}$ and $R = \{x_{\frac{n}{2}+1}, \ldots, x_n\}$,

(3) *conquer* the smaller instances by using recursive calls with inputs Q and R to find the smaller distances of pairs of points within each subset, and
(4) *combine* by returning the smallest value of (i) the minimum distance between pairs of points in Q, (ii) the minimum distance between pairs of points in R, and (iii) the distance $|x_{\frac{n}{2}+1} - x_{\frac{n}{2}}|$.

The approach can be visualized as follows.



It is implemented by the following simple algorithm (which assumes that the points have already been sorted).

Algorithm 2: SortedCPL($x_1, \ldots, x_n$)

```
$\delta_Q \leftarrow$ SortedCPL($x_1, \ldots, x_{\frac{n}{2}}$);
$\delta_R \leftarrow$ SortedCPL($x_{\frac{n}{2}+1}, \ldots, x_n$);
$\delta \leftarrow \min\{\delta_Q, \delta_R, |x_{\frac{n}{2}+1} - x_{\frac{n}{2}}|\}$;
return $\delta$;
```

The time complexity of the SortedCPL algorithm now satisfies the recurrence

$$T(n) = 2\,T(\tfrac{n}{2}) + O(1),$$

which is $T(n) = \Theta(n)$. So, by initially sorting the points in time $\Theta(n \log n)$, we obtain an algorithm for the Closest Pair on the Line problem with total time complexity $\Theta(n \log n)$.

Best algorithm. As it turns out, there's another algorithm that is even simpler and also has time complexity $\Theta(n)$ when the points are already sorted: since the closest pair of points in the input must be next to each other in the sorted list, we only need to consider adjacent pairs to find the minimum distance.

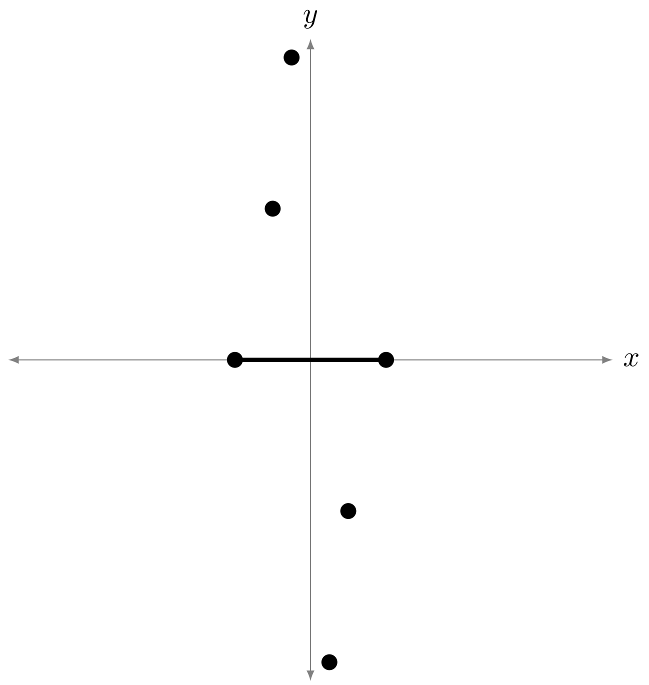
Algorithm 3: OptimalSortedCPL($x_1, \ldots, x_n$)

```
$\delta \leftarrow |x_1 - x_2|$;
for $i = 3, \ldots, n$ do
    $\delta \leftarrow \min\{\delta, |x_i - x_{i-1}|\}$;
return $\delta$;
```

2. Closest pair

We're now ready to return to the original Closest Pair problem.

As a first step, we can try to generalize the optimal algorithm to this setting. A simple way to do this is to sort the points by x position, then again check the distance of adjacent pairs of points in the sorted list. Unfortunately, that algorithm is not correct, as the following example shows.



(Note that in the example, the closest pair of points is (x_1, y_1) and (x_n, y_n)!)

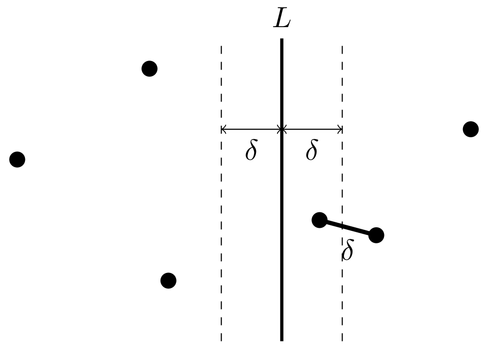
Divide & Conquer algorithm. We can try to "fix" the previous algorithm by considering other ways to sort the points, but this line of attack quickly leads to multiple dead ends and/or algorithms that are very hard to analyze. A better idea is to go back to the divide & conquer algorithm that we obtained for points on the line, and see if *that* algorithm generalizes nicely. We can indeed do so as follows.

(1) (Initially sort the list of points by x position,)
(2) *divide* the sorted list of points into the two smaller sets $Q = \{(x_1, y_1), \ldots, (x_{\frac{n}{2}}, y_{\frac{n}{2}}\}$ and $R = \{(x_{\frac{n}{2}+1}, y_{\frac{n}{2}+1}), \ldots, (x_n, y_n)\}$,
(3) *conquer* the smaller instances by using recursive calls with inputs Q and R to find the smaller distances of pairs of points within each subset, and
(4) *combine* by returning the smallest value of (i) the minimum distance between pairs of points in Q, (ii) the minimum distance between pairs of points in R, and (iii) the minimum distance between a point in Q and a point in R.

This looks promising, but we have to be careful: we can't afford to compute the distance between every pair of points $q \in Q$ and $r \in R$ (otherwise we are back to a time complexity $\Theta(n^2)$) so we have to find a more clever way to compute the *combine* step.

Restricting the area of focus. There's a neat observation that will prove to be extremely useful: to implement the *combine* step, we don't necessarily have to compute the minimum distance between a point in Q and a point in R: if we let δ be the minimum distance between 2 points in Q or 2 points in R, we only have to worry about the case where there's a point in Q and a point in R that are at distance less than δ from each other.

Let L denotes a vertical line separating the points in Q from the points in R and S denotes the set of points in X that are at distance at most δ from L, then the above observation means that we only need to consider the points in S.



Proposition 5.4. *Any pair of points $(x_i, y_i) \in Q$ and $(x_j, y_j) \in R$ at distance*

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leq \delta$$

from each other must satisfy $(x_i, y_i) \in S$ and $(x_j, y_j) \in S$.

Proof. If $(x_i, y_i) \notin S$, then the distance between the two points is

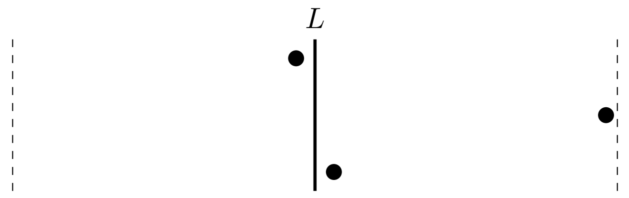
$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \geq \sqrt{(x_i - x_j)^2} = |x_i - x_j| > \delta$$

and the same lower bound applies if $(x_j, y_j) \notin S$ as well. □

Intuitively, this should be very helpful: we may expect that for many "well-behaved" inputs, only a few points are in S and we can compute the distance between every pair of these points to complete the *combine* step. Unfortunately, that approach does not work in general, because *all* n points might be contained in S. (This is the case, for instance, in the example we constructed at the beginning of the section.)

Nonetheless, if we look at this image carefully, we can see that we have made some progress in that the problem of finding the closest pair of points in S is starting to look a lot more like the one-dimensional version of the problem!

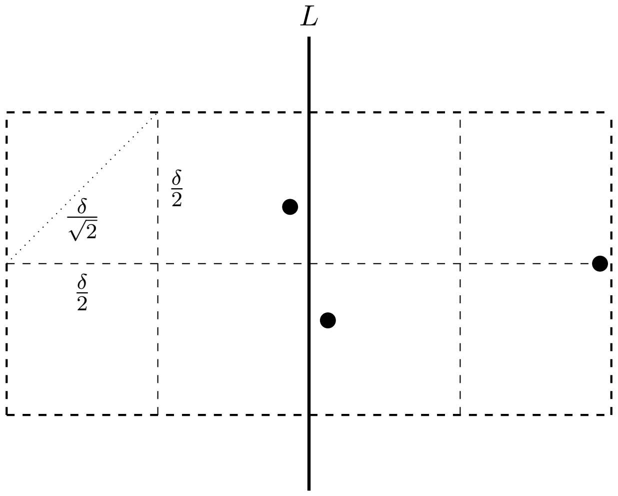
Key observation. If the problem of finding the closest pair of points in S is *exactly* like the one-dimensional version of the problem, we could then sort the points in S by their y value and then compute the distance between every adjacent pair of points. This will not quite work, because of examples like the following.



However, we will be able to show the next-best thing: that the closest pair of points in S will be *nearly* adjacent to each other when we sort by y value. This is because of the following key observation.

Proposition 5.5. *For any point $(x^*, y^*) \in S$, there can be at most 8 points $(x', y') \in S$ where $y^* \leq y' \leq y^* + \delta$.*

Proof. The space of all points that are at distance at most δ from L and with y coordinate bounded by $[y^*, y^* + \delta]$ can be partitioned into 8 squares with side length $\frac{\delta}{2}$. But there can be at most 1 point in S contained in each of those squares, because two points in the same square would have distance at most $\sqrt{(\frac{\delta}{2})^2 + (\frac{\delta}{2})^2} = \frac{\sqrt{2}}{2}\delta < \delta$ from each other and would both be in Q or in R, contradicting the fact that δ is the minimum distance between any two points in Q or two points in R. □



The proposition guarantees that when we do the *combine* step, we only need to compute the distances of points that are at most 8 positions apart in the list that is sorted by y coordinate.

We therefore obtain the following divide & conquer algorithm for the Closest Pair problem, which we run after sorting the points $(x_1, y_1), \ldots, (x_n, y_n)$ so that $x_1 \leq x_2 \leq \cdots \leq x_n$.

Algorithm 4: $\textsc{ClosestPair}((x_1, y_1), \ldots, (x_n, y_n))$

```
$\delta_Q \leftarrow \textsc{ClosestPair}\big((x_1, y_1), \ldots, (x_{\frac{n}{2}}, y_{\frac{n}{2}})\big)$;
$\delta_R \leftarrow \textsc{ClosestPair}\big((x_{\frac{n}{2}+1}, y_{\frac{n}{2}+1}), \ldots, (x_n, y_n)\big)$;
$\delta \leftarrow \min\{\delta_Q, \delta_R\}$;
$S \leftarrow \emptyset$;
for $i = 1, \ldots, n$ do
    if $x_{\frac{n}{2}} - \delta \leq x_i \leq x_{\frac{n}{2}} + \delta$ then
        $S \leftarrow S \cup \{(x_i, y_i)\}$;
$\textsc{SortByY}(S)$;
for $1 \leq i < j \leq |S|,\ j - i \leq 7$ do
    $\delta = \min\{\delta, \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}\}$;
return $\delta$;
```

What is the time complexity of this algorithm? Sorting the set S takes time $\Theta(n \log n)$ in the worst-case[1] and the rest of the work in the algorithm takes linear time, so that the time complexity of the algorithm on inputs that contain n points satisfies the recurrence

$$T(n) = 2\,T(\tfrac{n}{2}) + \Theta(n \log n)$$

[1] Recall that $|S|$ can be as large as n.

. . . which you can solve if you have completed Assignment 1! (Your solution should show that the time complexity of this algorithm is much better than $\Theta(n^2)$.)

An optimization. We can improve the time complexity of the algorithm by avoiding the step of sorting S at every iteration of the algorithm. The main idea is that we can sort the initial set of points by y positions (as well as by x positions; we keep both lists). Then when we build S we can use the list sorted by y position to create it in sorted order directly.

With this optimization, the time complexity of the algorithm now satisfies the recurrence

$$T(n) = 2\,T(\tfrac{n}{2}) + \Theta(n)$$

which, by the Master Theorem, is satisfied by $T(n) = \Theta(n \log n)$. The initial sorts by x and by y both have the same time complexity, which means that we obtain an algorithm that solves the Closest Pair problem in time $\Theta(n \log n)$. This is much better than the $\Theta(n^2)$ bound of the brute-force algorithm!