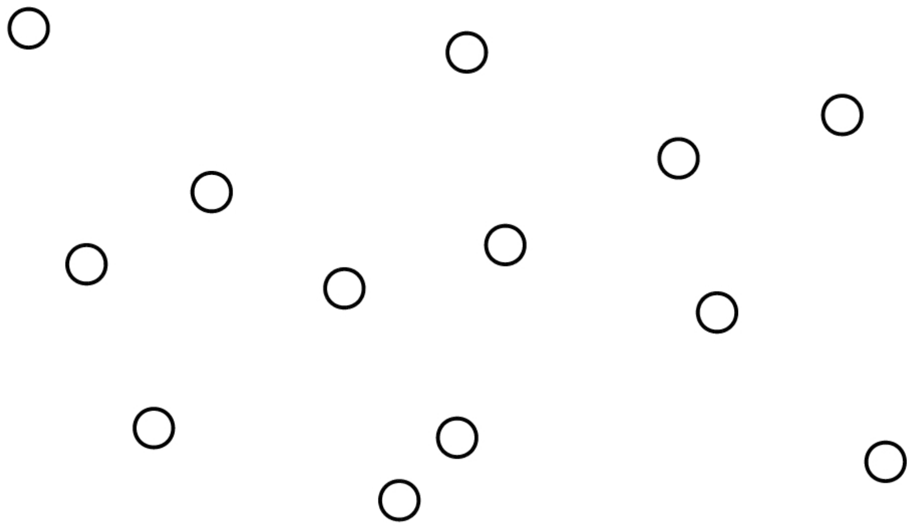
20190917

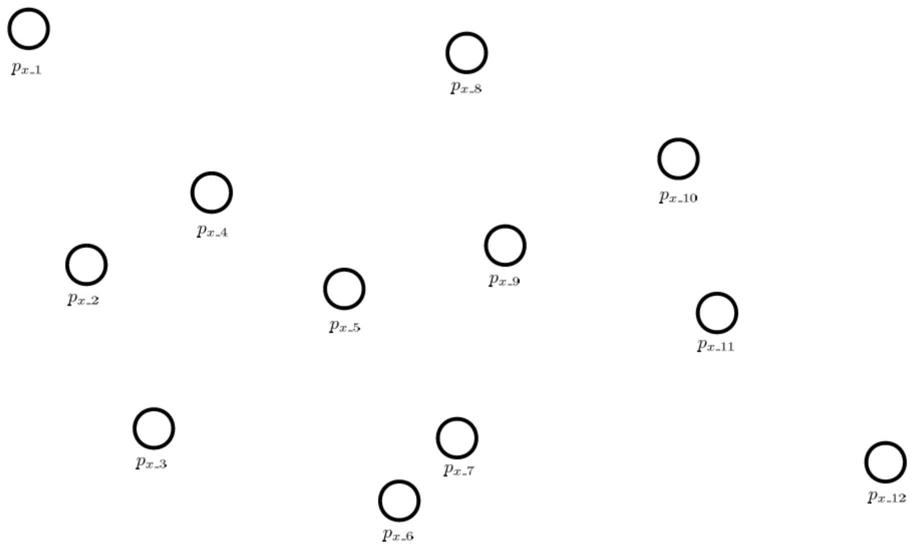
# Closest Pair of Points

Another surprising and satisfying application of the D&C method is the following algorithm for finding the closest pair of points in a set of points in the x-y plane.



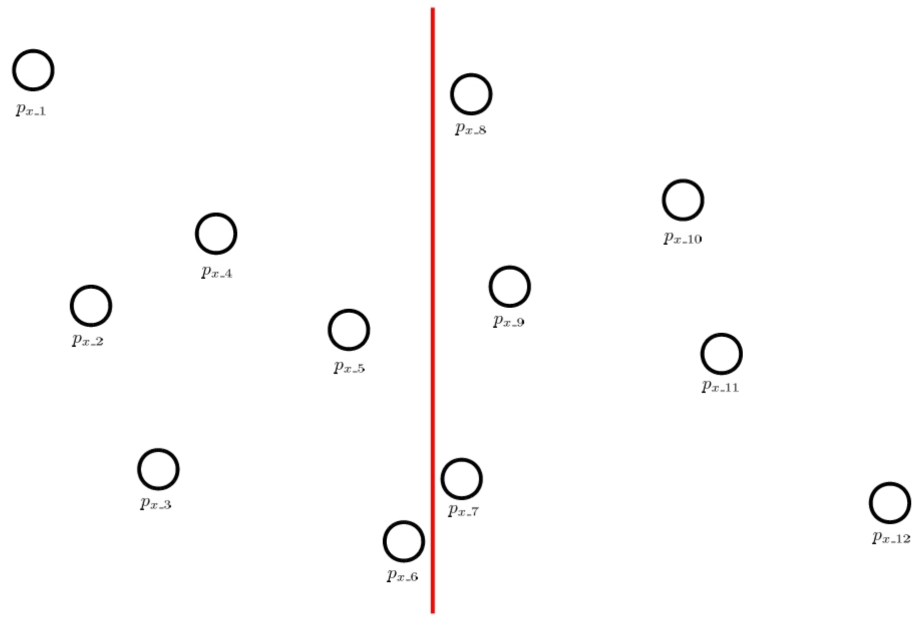
Assume we have n points, each identified by its x and y coordinates. Our goal is to find the two points with the minimum distance between them. Obviously we can solve this problem in O $(n^2)$ time by computing the distance between all pairs of points. It may seem unlikely that we can reduce this, but by clever design we can eliminate enough of the pairwise distance calculations that we can achieve a lower complexity.

For reasons that will become clear later, we start by creating a list or array of all the points sorted by their x-coordinate and another list of all the points sorted by their y-coordinate. This is a pre-processing phase that sets up the rest of the algorithm.

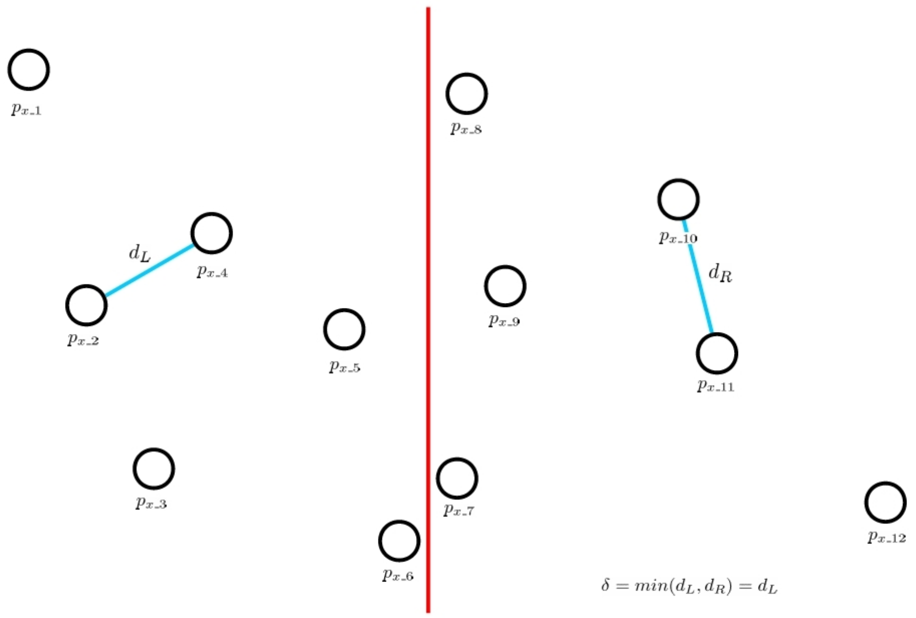


In this figure we see the points labelled in left-to-right order. This makes it very easy to split the set into a Left half and a Right half, which is exactly what we do. We then recursively solve the problem on the left side and the right side. As always with a D&C algorithm, when the problem size is $\leq$ some specific number we solve the problem directly. In this case we might decide that when the number of points is $\leq 3$ (or 5, or 10 – it really doesn't matter what cut-off value we pick so long as we stick with it), we will compute all the distances between the points. This takes O(1) time, which means we can treat it as a constant.

However, after we split the point set into a left side and a right side and solve the problem recursively on both sides, we are not done. We still have to deal with the possibility that the two points that are closest together are on opposite sides of the dividing line. If we compute all the distances between points on the left and points on the right, we are back with O($n^2$) complexity.

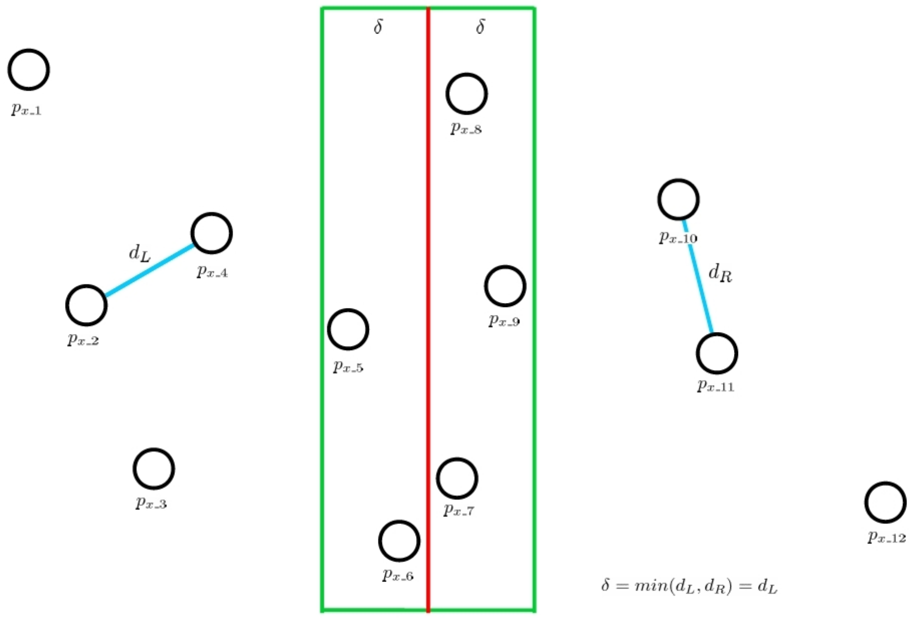


Fortunately we are able to avoid most of the potential computations. Let $d_L$ be the minimum distance on the left side, and $d_R$ be the minimum distance on the right side - we get these values from the recursive applications of the algorithm to the left and right halves of the set. Let $\delta$ = min($d_L$, $d_R$).



We need to determine if there are two points, one on each side of the dividing line, that have distance less than $\delta$ from each other. We can eliminate all points that have distance more than $\delta$ from the dividing line, since they cannot be less than $\delta$ from any point on the other side of the line.

Imagine a vertical panel or strip, $2 * \delta$ wide, centred on the dividing line between the left and right sides. The only points we need to consider in this stage of the algorithm are within this panel.

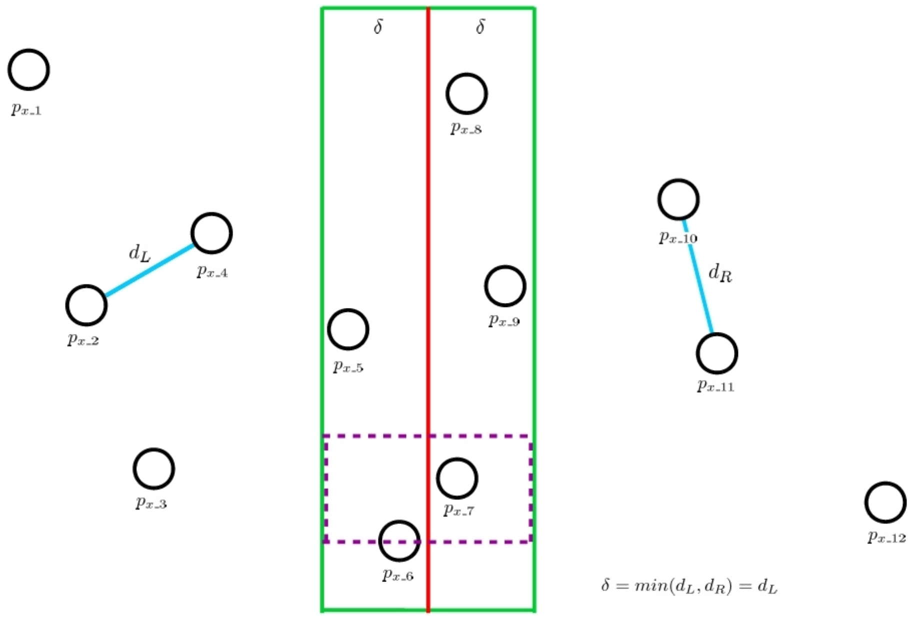


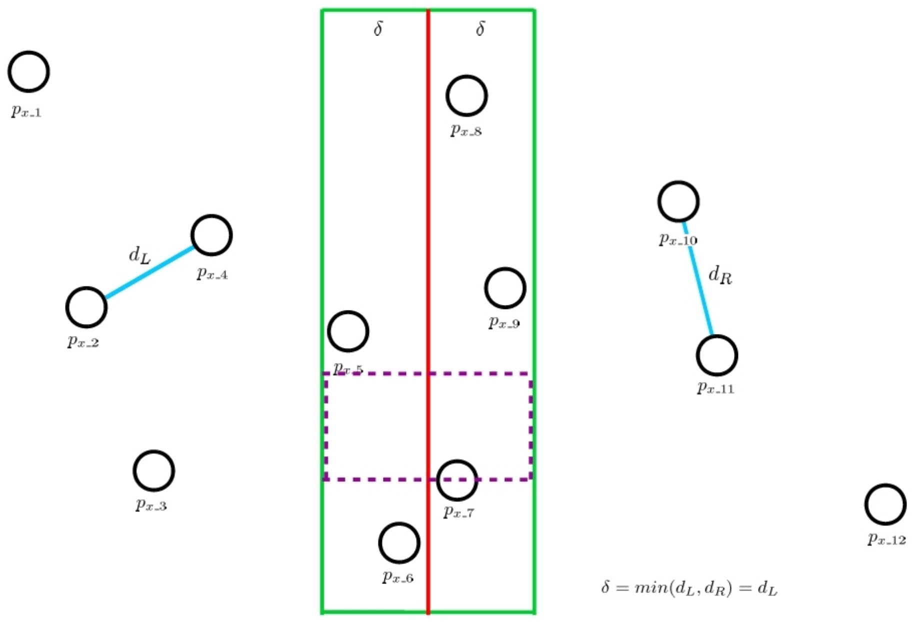
Take these points in ascending order by y-coordinate (this is why we sorted the points on their y-coordinates before we started!). For each point p, compute its distance to the points in the panel **above it** that might possibly be less than $\delta$ away.

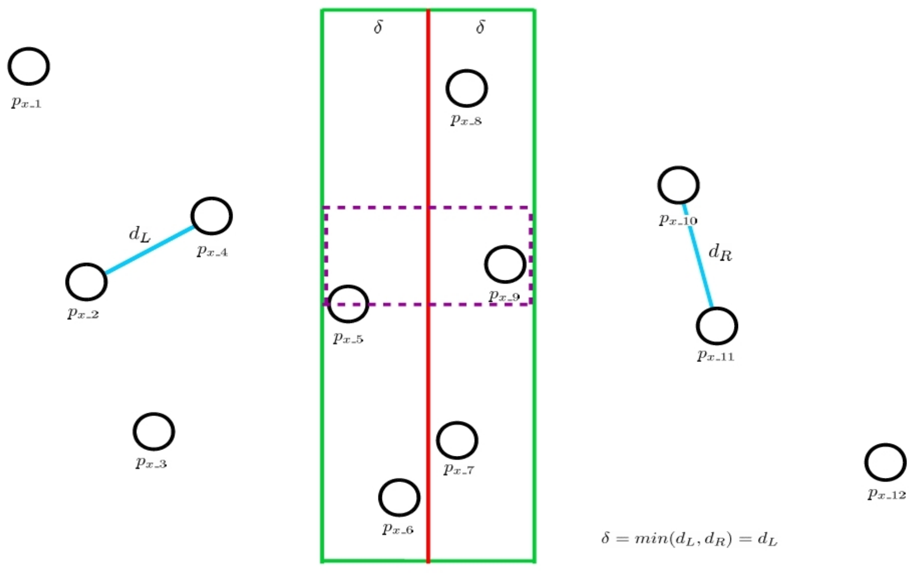
But wait a minute here. It's entirely possible that **all** of the points in the set are inside this vertical strip. If that happens, and we compute the distance from each point to all the points above it in the vertical strip, we will be right back to $O(n^2)$ complexity. But this is where the algorithm gets really smart. It turns out that no matter how many of the points are in the vertical strip, we can limit the number of pairs for which we have to compute the distance.

Let p be the lowest point in the vertical strip. We can imagine a box $2 * \delta$ wide and $\delta$ high with p on its bottom edge. Any point above p in the vertical strip that could be $< \delta$ away from p must be in this box.

The following 3 figures show the magical box of holding for each of the lowest three points in the vertical panel in our example.




$\delta$
$\delta$
$p_{x\_1}$
$p_{x\_2}$
$p_{x\_3}$
$p_{x\_4}$
$p_{x\_5}$
$p_{x\_6}$
$p_{x\_7}$
$p_{x\_8}$
$p_{x\_9}$
$p_{x\_10}$
$p_{x\_11}$
$p_{x\_12}$
$d_L$
$d_R$
$\delta = min(d_L, d_R) = d_L$


$\delta$
$\delta$
$p_{x\_1}$
$p_{x\_2}$
$p_{x\_3}$
$p_{x\_4}$
$p_{x\_5}$
$p_{x\_6}$
$p_{x\_7}$
$p_{x\_8}$
$p_{x\_9}$
$p_{x\_10}$
$p_{x\_11}$
$p_{x\_12}$
$d_L$
$d_R$
$\delta = min(d_L, d_R) = d_L$

By a simple geometric argument there cannot be more than 7 points other than p in this box (see my note [1] below) – and any point that is above the box is > $\delta$ away from p. Thus for each point in the vertical panel, we need to compute no more than 7 distances to other points in the panel – and that takes O(1) time. Even if all n points are in the panel, the complexity of computing the necessary distances within the panel is in O(n).

Thus the complexity of the algorithm is given by the recurrence relation

$$\begin{aligned} T(n) &= 2*T(n/2) + c*n && \text{when } n > 3 \\ T(n) &= \text{constant} && \text{when } n \leq 3 \end{aligned}$$

(Remember, the "3" is our arbitrarily chosen cut-off value for the recursion. It can be replaced with any other constant without changing the complexity)

We know this recurrence relation - it is exactly the same one that describes the complexity of Mergesort. We know that it works out to O(n*log n). This is the same as the complexity of the pre-processing step, which means that the pre-processing step is effectively free.

It is worth noting that if we had to re-sort the points at the beginning of each recursive call, the complexity would be higher. Fortunately we don't - each reduced set of points is just a subset of the set at the previous level and the relative order of the points does not change.

## Next stop – Convex Hulls!

1Note: I have followed the text-book here by saying that there are no more than 7 points to consider. In class I presented an argument that there can be no more than 5 such points. The argument for 7 such points depends on allowing identical points in the set, and having the identical pairs on "opposite sides" of the dividing line. I think this situation can be resolved as a special case. Either way, the number of point-combinations that must be considered is strictly linear.