Closest Pair Problem

Given a set of points Q in plane, find out a pair whose distance is the smallest among all pairs of points.

- An obvious solution: compare each pair and determine the closest pair. This takes $O(n^2)$ time for n points

- We develop an $O(n\log n)$ time algorithm. It builds on divide-and-conquer approach.

A generic strategy: The algorithm works recursively. At each recursive step, we take a subset $P \subseteq Q$ and two arrays X and Y.

X: stores points of P sorted in x-coordinates

Y: stores points of P sorted in Y-coordinates

Divide: • find a vertical line l that bisects P into P_L and P_R with $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$. Points in P_L are on or to the left of l, all points in P_R are on or to the right of l.

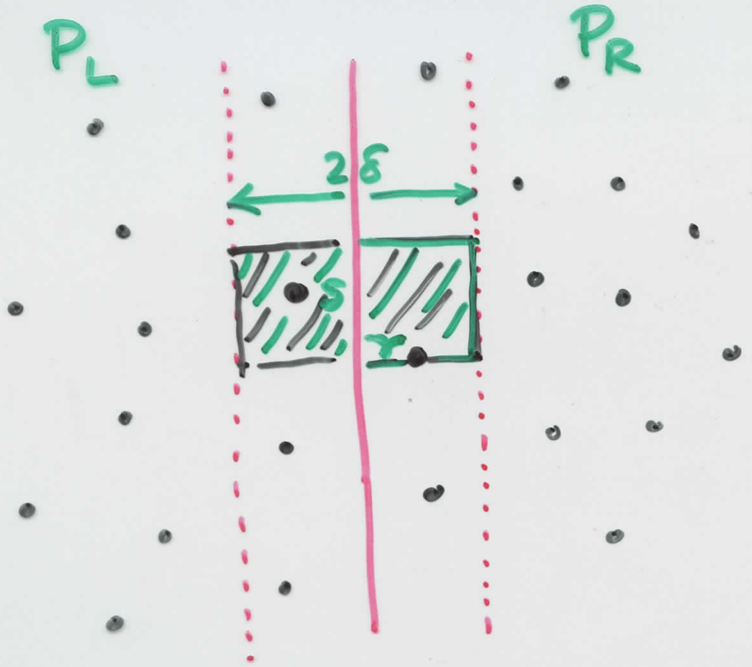
• Split X into X_L and X_R that contain points in P_L and P_R respectively sorted by x-coordinates

• Similarly split Y into Y_L and Y_R.

Conquer: • Recursively compute closest pairs in P_L and in P_R. Suppose the closest pair distances be δ_L and δ_R. Set $\delta = \min\{\delta_L, \delta_R\}$.

Combine: The closest pair in P is either the pair found in P_L or P_R, or there is a pair $r \in P_R$, $s \in P_L$ so that $d(s,r) < \delta$.

Combine contd...: The pair (s, r) must lie within the vertical strip. ③



Algorithm for computing (s, r):

1. Create array Y' from Y by removing all points that are not in 2δ strip. Y' is sorted in y-coordinates as Y is.

2. For each p in array Y', we find points within δ distance that are in Y'.
   This can be achieved by considering only 7 points following p in array Y'.

3. Suppose δ' is the distance of closest pair computed in step 2. If $\delta' < \delta$, this pair and δ' are returned. Otherwise δ and the corresponding pair is returned.

The complexity: All steps in recursion are linear-time implementable. Specifically, Combine can work in linear time because of the 7 checks. One can create a sorted subset of ~~an~~ a sorted array in linear time ((P_L, P_R from P which are all x-sorted), similarly X_L, X_R, Y_L, Y_R ~~can~~ from X and Y).

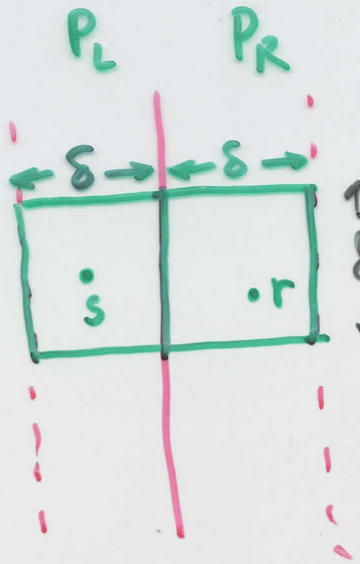
```
Split (Y)
length(Y_L] := Length(Y_R) := 0;
for i := 1 to length(Y) do
    if Y[i] ∈ P_L
        then length(Y_L) := length(Y_L) + 1
             Y_L[length[Y_L]] := Y[i]
    else length[Y_R] := length[Y_R] + 1
         Y_R[length[Y_R]] := Y[i]
endfor
```

Time complexity:= $T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 3 \\ O(1) & \text{if } n \leq 3. \end{cases}$

$T(n) = O(n \log n)$ assuming we stop recursion when $n \leq 3$ and apply the brute-force method. Also, we pre-sort the arrays for Q, X, Y. at the beginning of sorting.

Space complexity: One needs to be careful that new arrays are not created at each recursive step. Then, space complexity will be $\Theta(n \log n)$. We need to reuse the old arrays.

Why 7 points?



Observe that s and r must be within a $\delta \times 2\delta$ rectangle as shown in figure. This is because s and r are within the strip(2δ), and they cannot be more than δ apart vertically to have $d(s,r) < \delta$.

All points in P_L are at least δ apart from each other. How many points can the $\delta \times \delta$ square may have with this constraint. At most 4 (at the four corners). Similarly, $\delta \times \delta$ square on right for P_R can have at most 4 points. $\Rightarrow$ for s, we need to check at most 7 points.
These 7 points are consecutive in Y' around s.
But, only downward checking is sufficient... why?