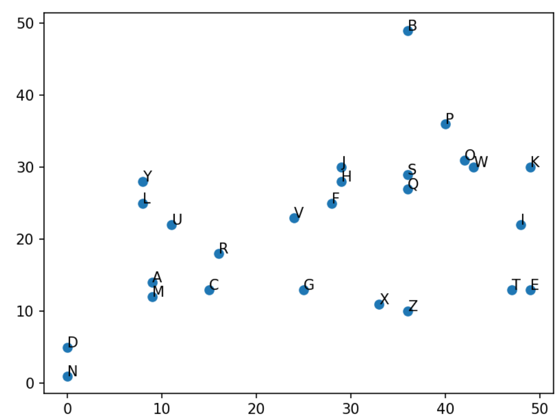
Closest Pairs

In 2 dimensions:

we have n points in a plane, our objective is to find the closest couple of points between all couples in the fastst time possible.

Notice that a naive approach whould be to test all possible couples, sort them by distance and then take the first element, resulting in a complexity of $O(n^2)$.we will show that there is a better approach.

let's begin drawing our points on the plan:



$$
\begin{array}{ll}
a:(9,8), & b:(36,36),\\
c:(15,14), & d:(0,49),\\
e:(49,13), & f:(28,5),\\
g:(25,13), & h:(29,25),\\
i:(48,13), & j:(29,28),\\
k:(49,22), & l:(8,30),\\
m:(9,30), & n:(0,25),\\
o:(42,12), & p:(40,1),\\
q:(36,31), & r:(16,36),\\
s:(36,27), & t:(47,18),\\
u:(11,29), & v:(24,13),\\
w:(43,22), & x:(33,23),\\
y:(8,28), & z:(36,10)
\end{array}
$$

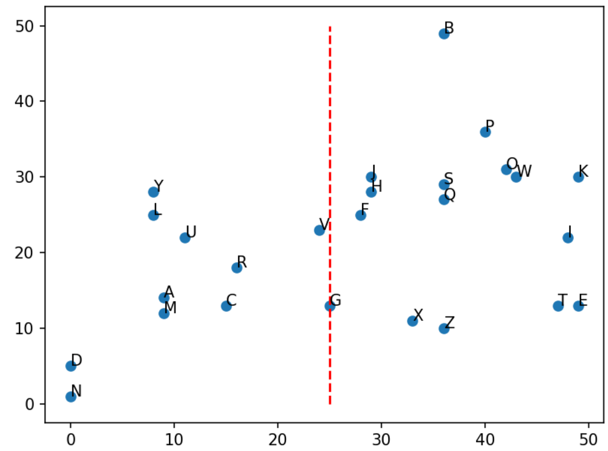
what we can do now is write all our points into one array $P=\{p_1 \dots p_n\}, p_x \in \mathbb{R}^2$, remember that we seek for $p_a, p_b : \|p_a - p_b\| \leq \|p_c - p_d\| \forall p_c, p_d \neq p_a, p_b; \;\; p_a, p_b, p_c, p_d \in P$, so the couple with minimum distance.

Anstatz

we will use a Divide Et Impera algorithm, where we split into 2 parts:

Divide

we sort the points according to the x coordinate int 2 halves



$P = \{d, n, l, y, a, m, u, c, r, v, g, f, h, j, x, b, q, s, z, p, o, w, t, i, e, k\}$

Conquer

we can consider 3 cases for the couple we are searching:

- the 2 points are both on the left
- the 2 points are both on the right
- one point is on the left, the other on the right

what we want to do is taking the smaller distance in the 3 cases

Algorithm

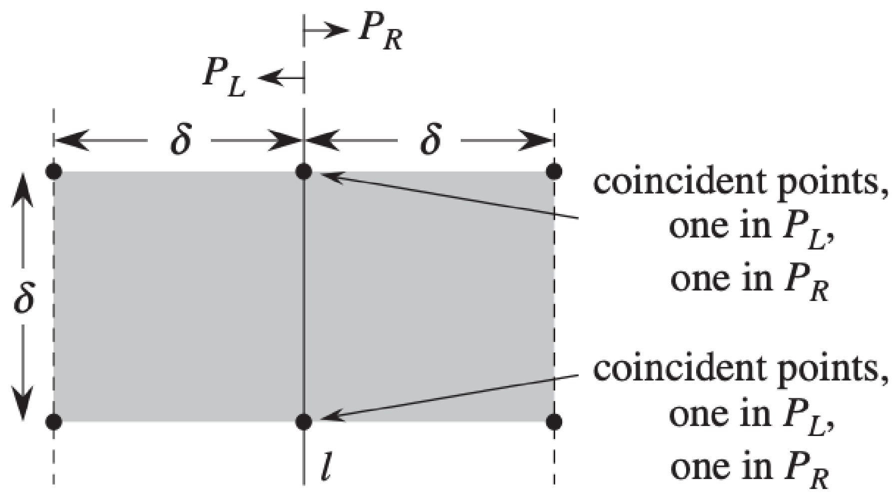
```
Function ClosestPair(P)
  Let n be the number of points in set P.
  If n <= 3, compute the distance between each pair and return the smallest.
  Sort the points in P according to their x-coordinates, let's call this array Px.
  Sort the points in P according to their y-coordinates, let's call this array Py.
  Call the auxiliary function ClosestPairAux(Px, Py).

Function ClosestPairAux(Px, Py)
  Let n be the number of points in set P.
  If n <= 3, compute the distance between each pair and return the smallest.
  Find the midpoint Q of Px, divide Px into two subsets:
    Qx (points to the left of the midpoint)
    Rx (points to the right of the midpoint)
  Divide the Py into two subsets:
    Qy (points in Qx, sorted by y-coordinate)
    Ry (points in Rx, sorted by y-coordinate)
  d1 = ClosestPairAux(Qx, Qy) // recursive call on the left subset
  d2 = ClosestPairAux(Rx, Ry) // recursive call on the right subset
  d = min(d1, d2) // find the minimum distance
  Sy = points of Py that are within distance d from the midpoint Q.
  d3 = minimum distance of pairs in Sy.
  return min(d, d3)
```

$T(n) = 2T(\frac{N}{2}) + O(nlogn) + O(nlogn) = 2T(\frac{N}{2}) + O(nlogn) = O(nlog^2(n))$

let’s consider the last part in closedSplitPairs: **why do we iterate only 6 times?**

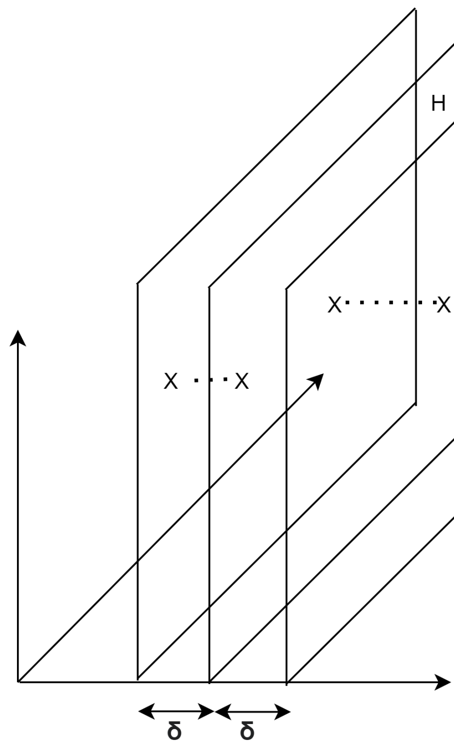
the answer lives in this **Conjecture:** $|Q| \leq 6$**,** i.e. given a point and a distance δ we can show that there a a maximum of 6 points with that distance form that point:



(b)

in higher dimensions

in the case of higher dimensions the splitting describes an hyperplane H with dimension $d-1$ and partition P according to the x coordinate of this hyperplane. This time the closest points are not contained in a square , but in an hyperspace distant δ in all dimensions from the cutting hyperplane. At this point we project all the points onto H and we get a $d-1$ dimensional closest pairs problem, then we can iterate until we reach our case in 2 dimensions




```
Function ClosestPair(P)
  Let n be the number of points in set P.
  For each dimension d in D, sort the points in P according to their d-th coordinate, let's call these arrays P1, P2, ..., PD.
  Call the auxiliary function ClosestPairAux(P1, P2, ..., PD, 1).

Function ClosestPairAux(P1, P2, ..., PD, current_dim)
  Let n be the number of points in set P.
  If n <= 3, compute the distance between each pair and return the smallest.
  Find the midpoint Q of P[current_dim], divide P[current_dim] into two subsets:
    Qx (points to the left of the midpoint)
    Rx (points to the right of the midpoint)
  For each dimension d in D, divide the sorted list P[d] into two subsets based on Q:
    Qy (points in Qx, sorted by d-th coordinate)
    Ry (points in Rx, sorted by d-th coordinate)
  d1 = ClosestPairAux(Q1, Q2, ..., QD, (current_dim % D) + 1) // recursive call on the left subset
  d2 = ClosestPairAux(R1, R2, ..., RD, (current_dim % D) + 1) // recursive call on the right subset
  d = min(d1, d2) // find the minimum distance
  Sy = points of P[current_dim] that are within distance d from the midpoint Q.
  d3 = minimum distance of pairs in Sy considering all dimensions in a circular manner starting from (current_dim % D) + 1.
  Return min(d, d3)
```

The new complexity becomes:

$T(n,d) = 2T(\frac{N}{2}, d) + O(n) + U(n, d-1)$,

where U is the complexity of finding the closest pairs in the $d-1$ sized problem

$U(n, d-1) = O(nlog^{d-2}n)$,

it follows that: $T(n,d) = 2T(\frac{N}{2}, d) + O(n) + O(nlog^{d-2}n) = O(nlog^{d-1}n)$

it can be proven that if we build an hyperbox around a point we will discover that that box will contain a number of points in the order of $O(4^d)$

proof:

imagine we have a ball of radius $\frac{\delta}{2}$ into a box of size 2δ

$vol(box) = (2\delta)^d$

$vol(ball) = const \cdot (\frac{\delta}{2})^d$

$\#ballsNonOverlapping \leq \frac{vol(box)}{vol(ball)} = 4^d$

Cool material

https://sites.cs.ucsb.edu/~suri/cs235/ClosestPair.pdf

https://itzsyboo.medium.com/algorithms-studynote-4-divide-and-conquer-closest-pair-49ba679ce3c7

https://stackoverflow.com/questions/15664962/explanation-of-these-seven-points-in-finding-the-closest-pair-of-points

https://www.cs.ucdavis.edu/~bai/ECS122A/Notes/Closestpair.pdf