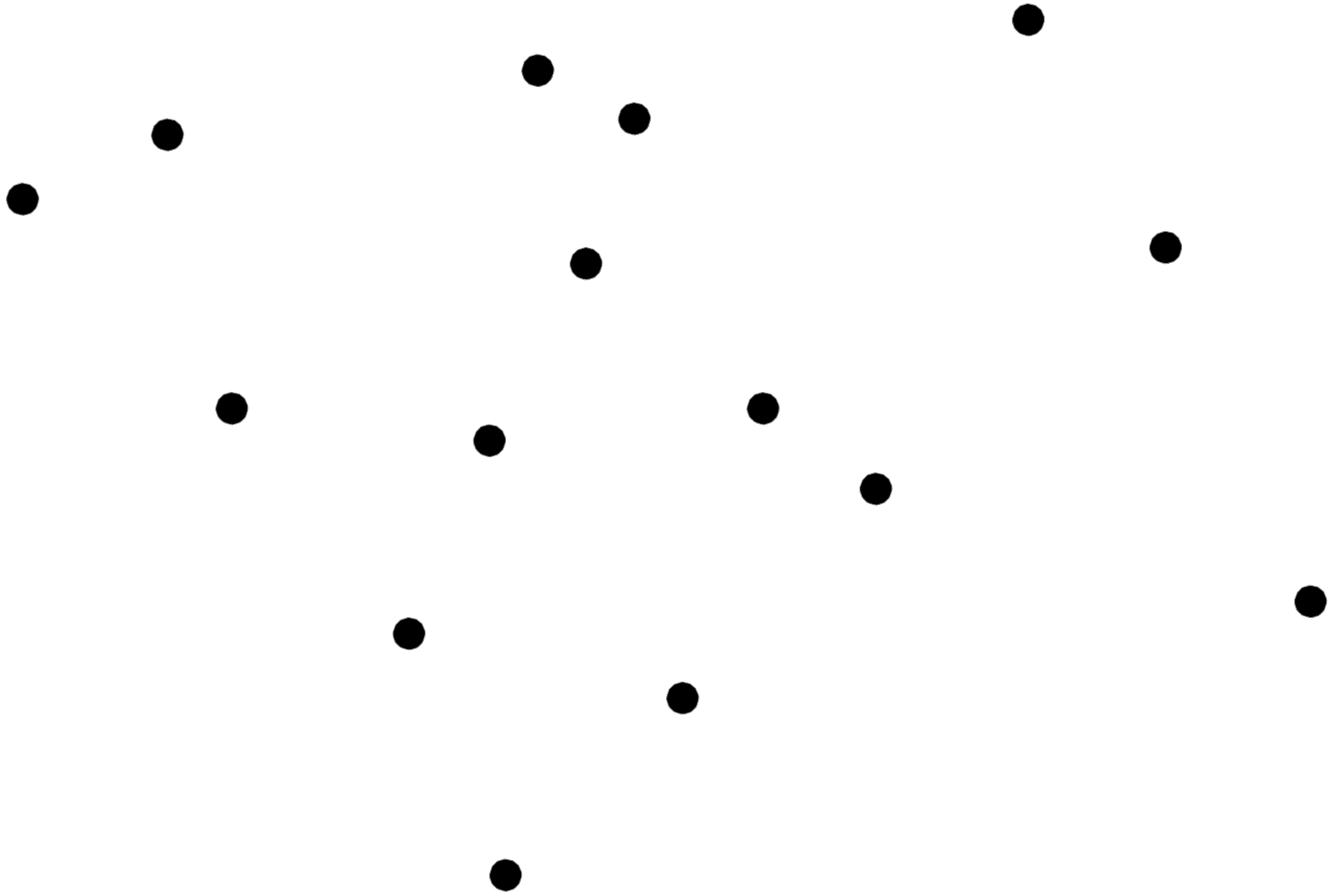
Find the closest pair. Brute force $\Theta(n^2)$



Reminder: Brute force closest pair

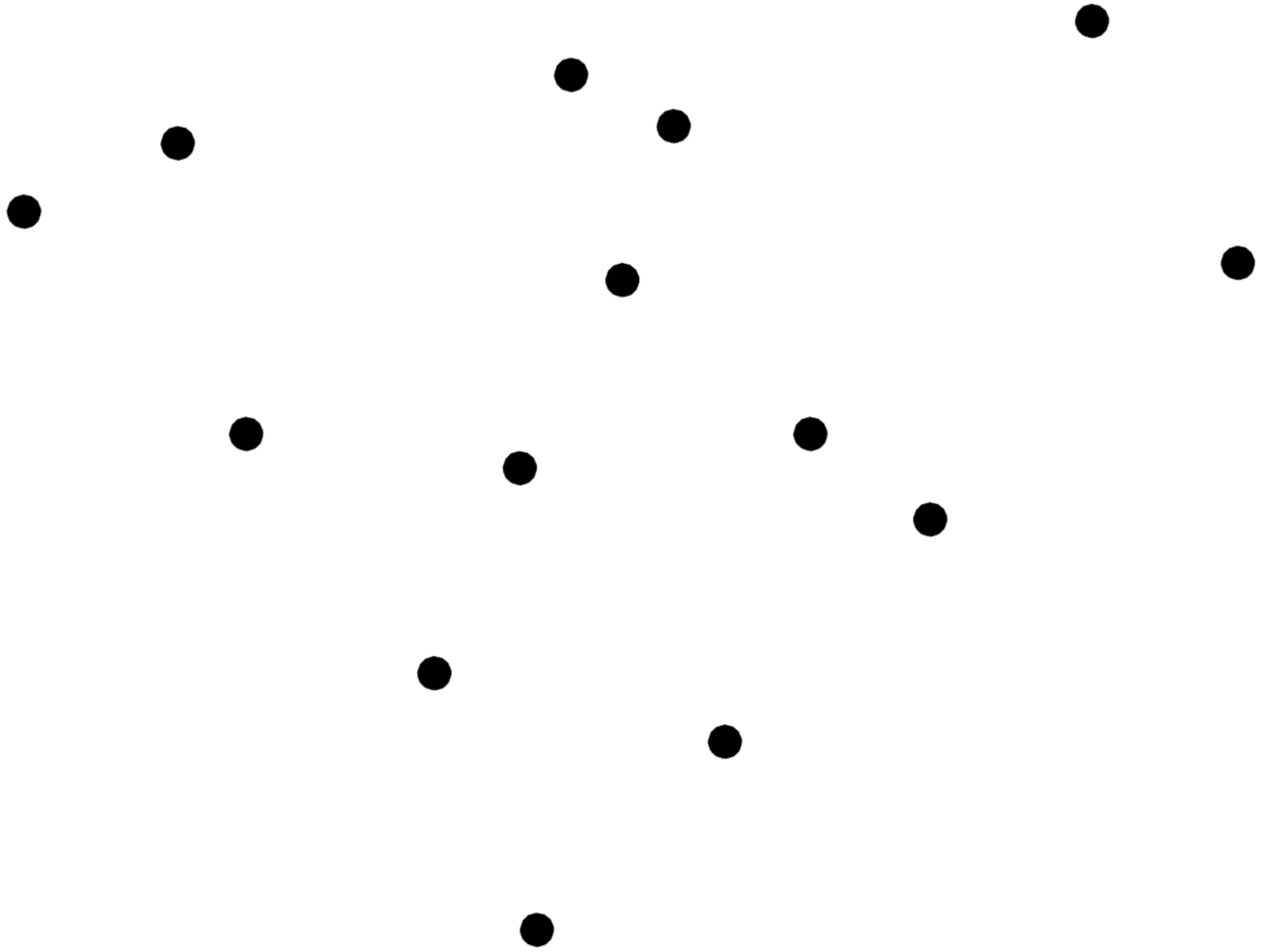
$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

```
BruteForce(Array P [0 to n – 1] of {x, y} )
    d = MAX_NUMBER_ON_SYSTEM
    for i = 0 to n – 2:
        for j = i + 1 to n-1:
            m =   (P[i].x – P[j].x) * (P[i].x – P[j].x)  +
                  (P[i].y – P[j].y) * (P[i].y – P[j].y)
            if m < d: d = m
    return d
```

Divide & Conquer closest pair

- Appears to have been invented by Michael Ian Shamos and Dan Hoey in 1975.
- M. I. Shamos and D. Hoey. "Closest-point problems." In Proc. 16th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 151—162, 1975 (DOI 10.1109/SFCS.1975.8
- http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=4567872

Divide & conquer closest pair (1)

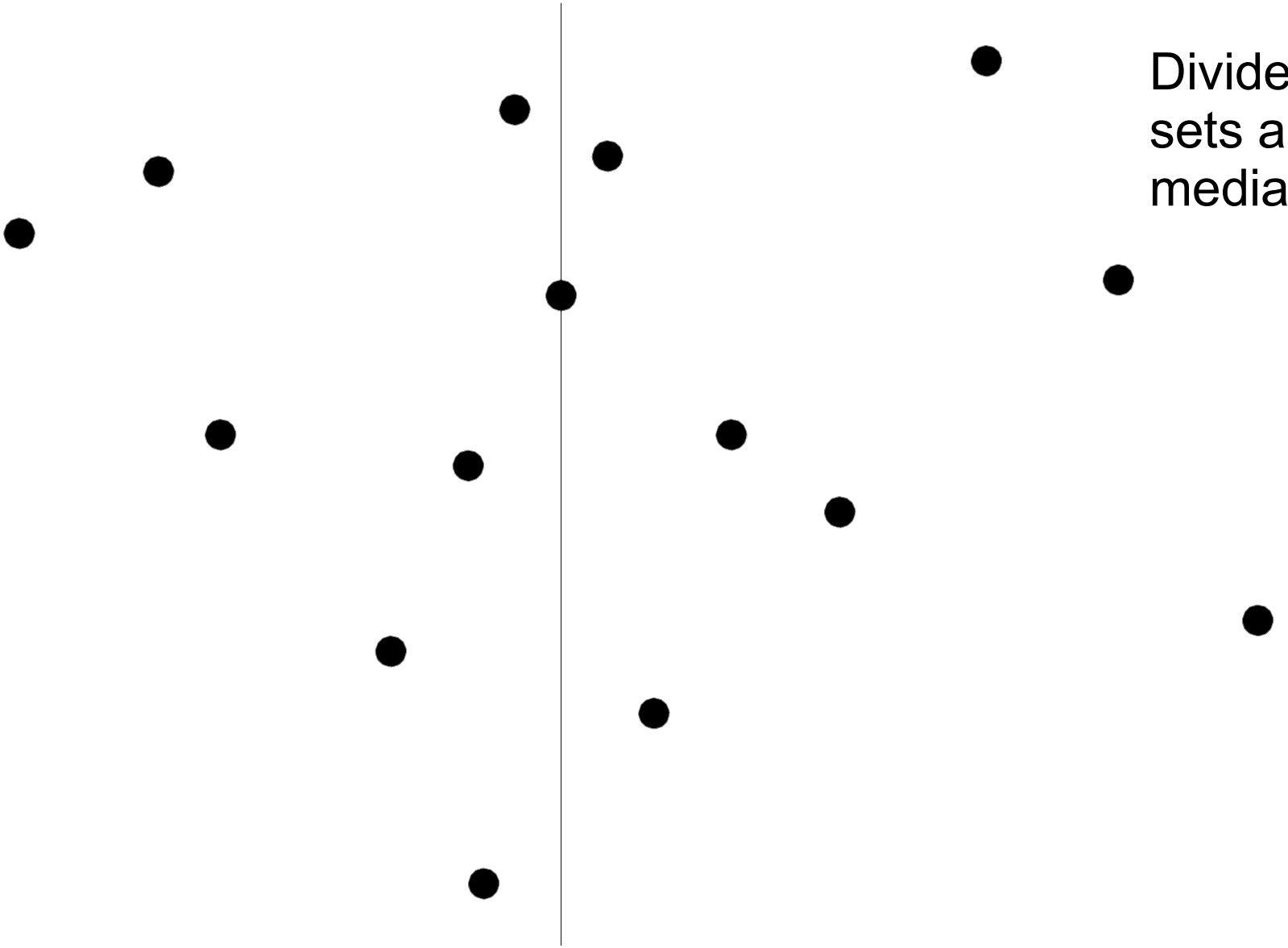


We have a set of distinct points on a Cartesian plane.

Sort them in x co-ordinate order.

For later, copy them and sort the copy in y co-ordinate order.

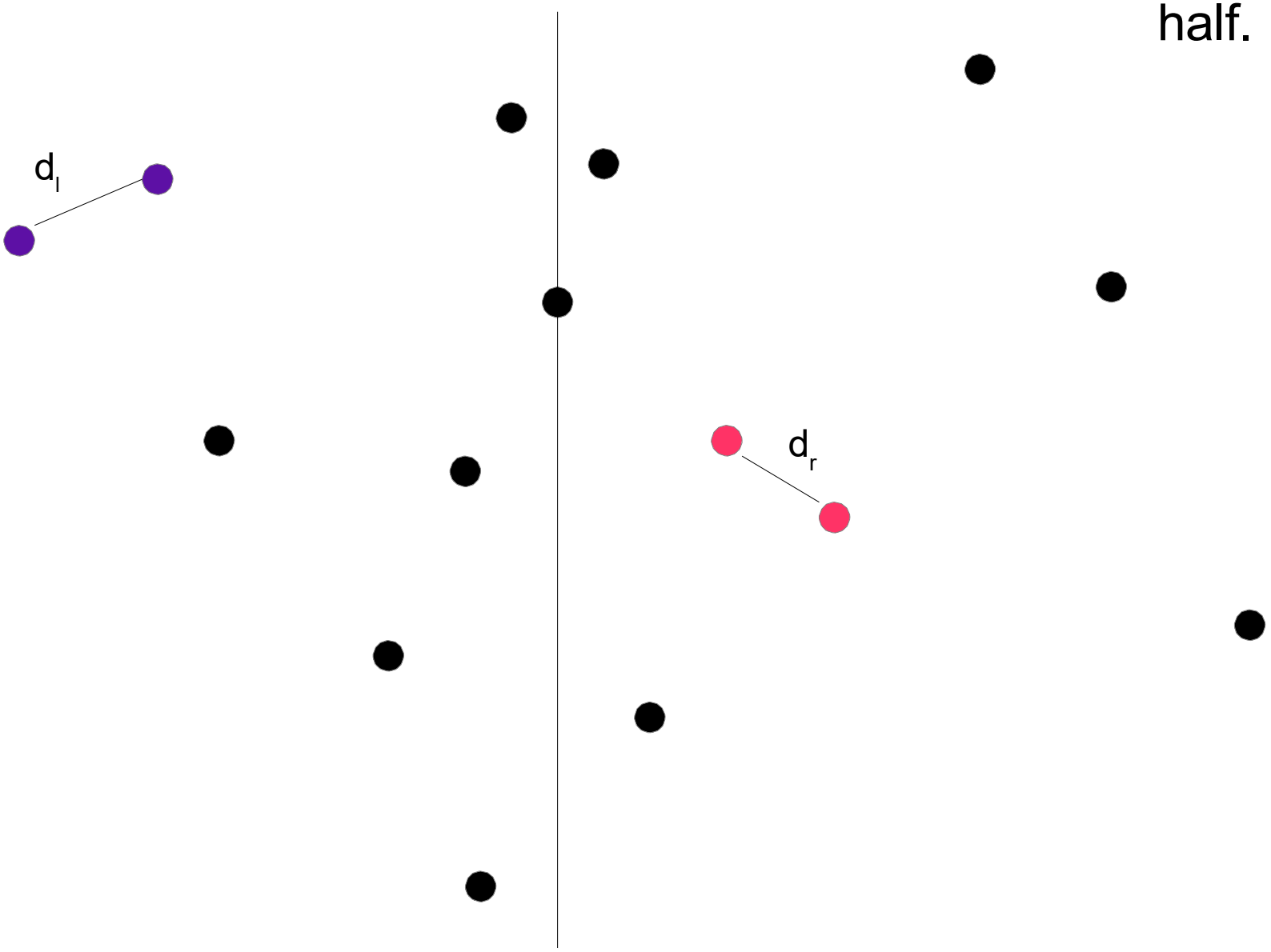
D & C closest pair (2)



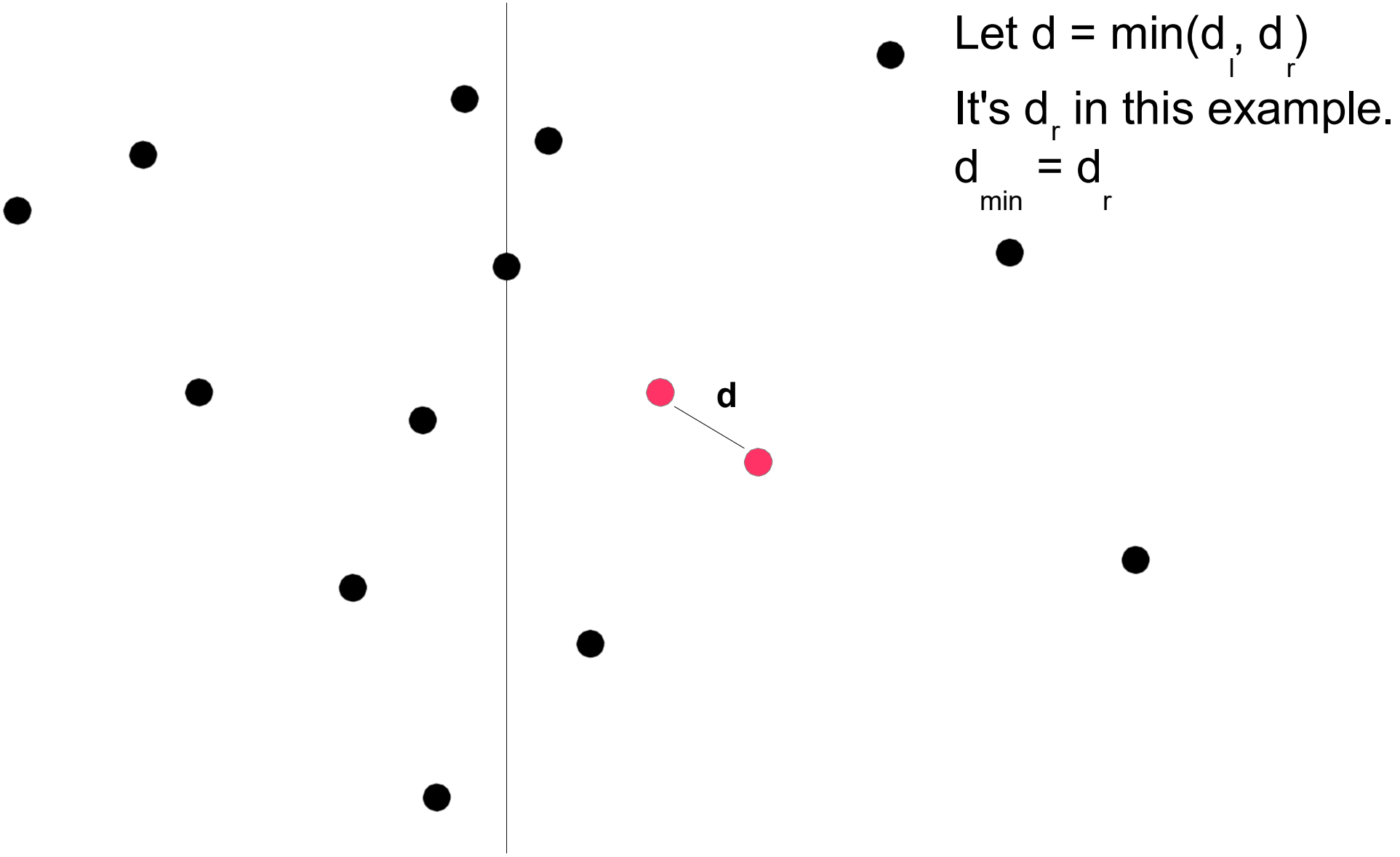
Divide into two sets along the x median.

D & C closest pair (3)

recursively found the shortest distance in each half.



D & C closest pair (4)

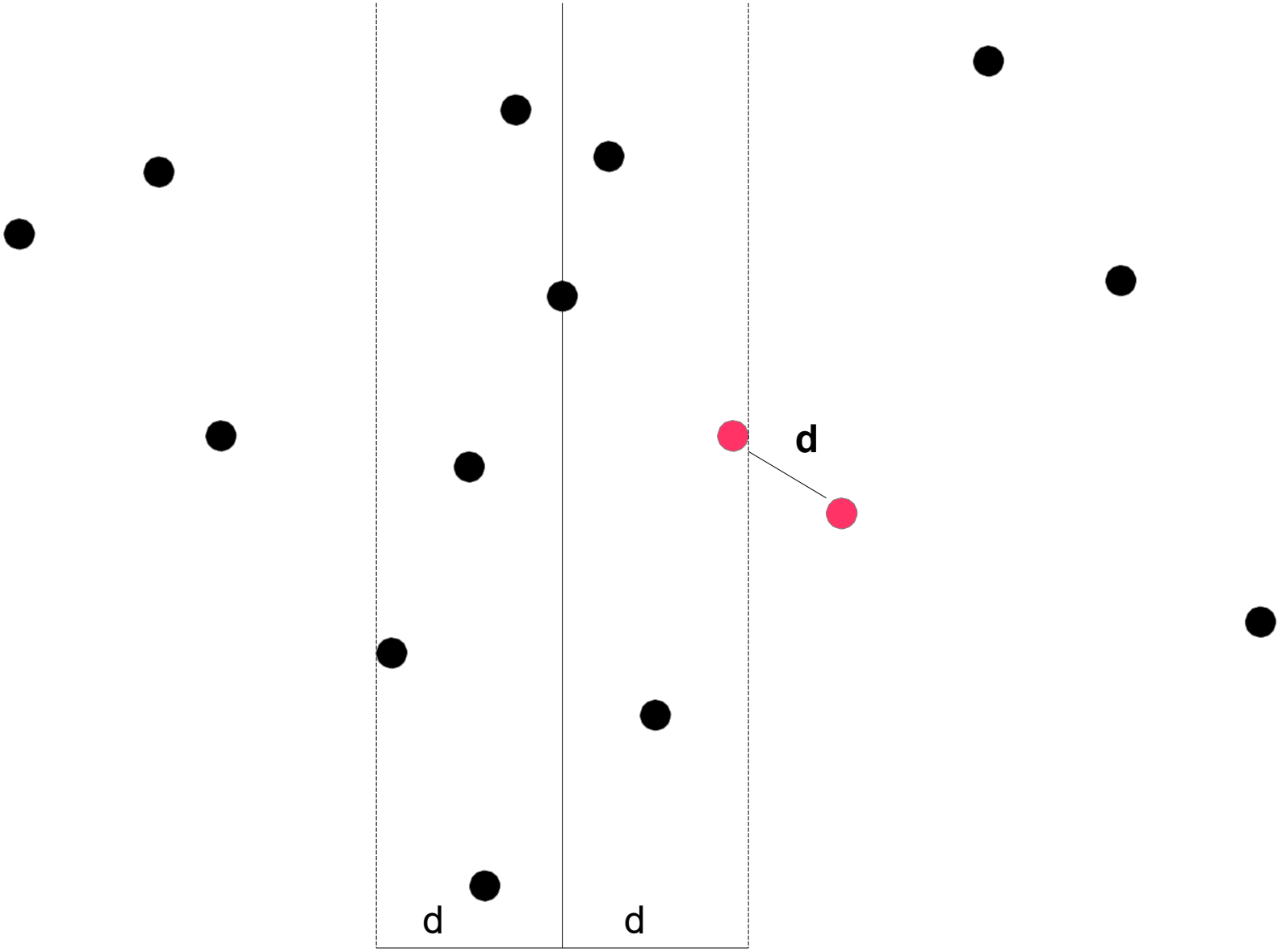


Let $d = \min(d_l, d_r)$

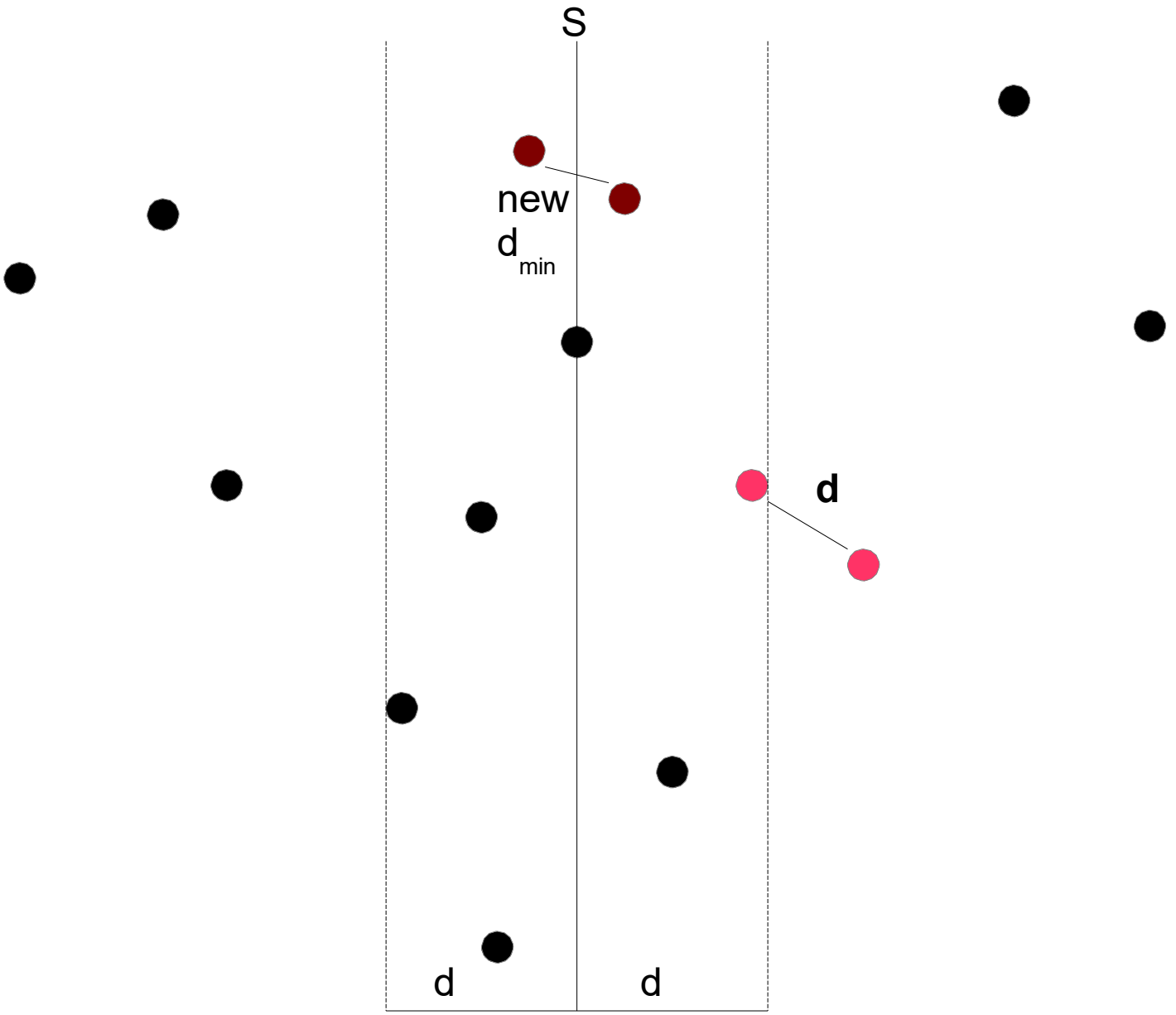
It's d_r in this example.

$d_{min} = d_r$

D & C closest pair (5)



D & C closest pair (6)



Let S = the points in the rectangular region.

We systematically go through S in order of y co-ordinates.

Along the way we update d_{min} as we find smaller distances.

D & C closest pair (7)

- It's hard to see how this can differ in efficiency from brute force. What if S has almost as many points in it as the non S part of the plain?
- If we use brute force, then for each point in S we compare it to **ALL** points in front of it:

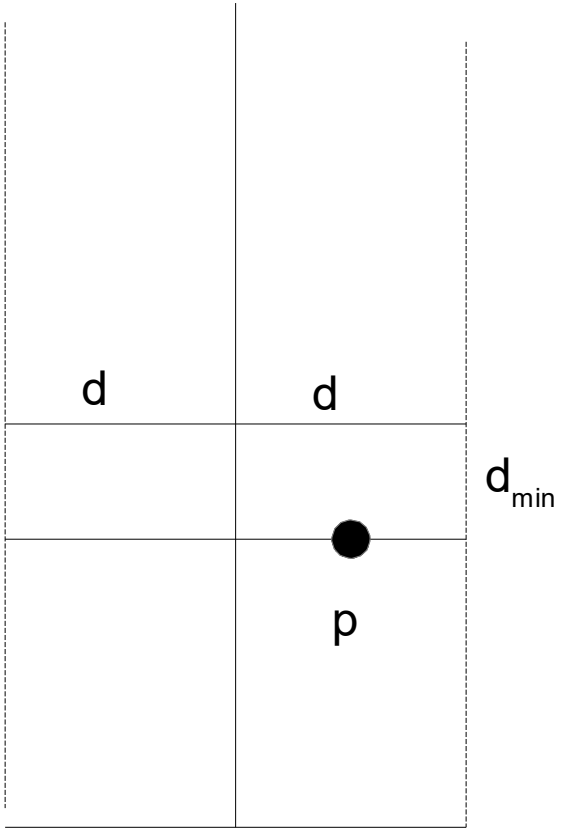
```
for i = 0 to n – 2:
    for j = i + 1 to n-1: etc
```

It seems that this is exactly what we are doing here, so why bother?

But there is actually a property of S that massively limits the number of points we have to look at.

D & C closest pair (8)

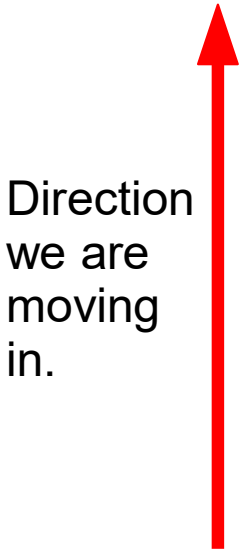
Direction in which we are going through the points in S.

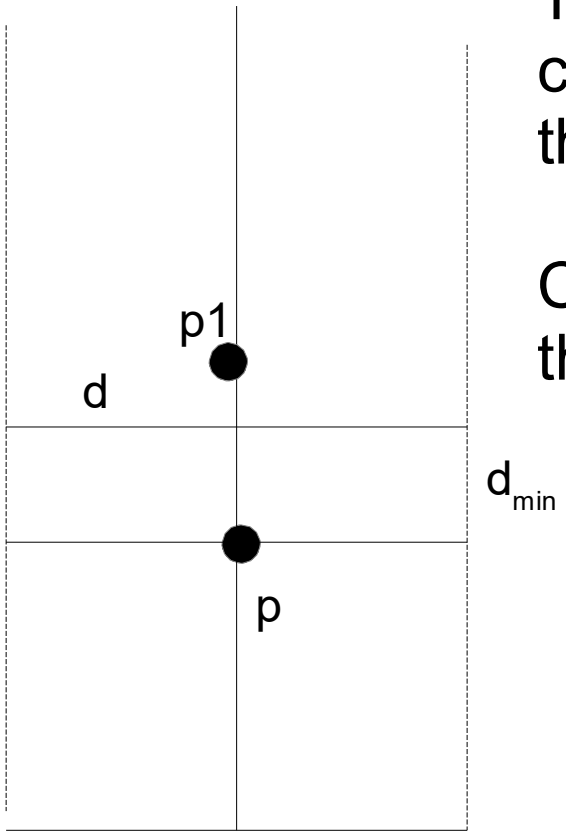


If there is a point closer to p and in front of p than distance d_{min} it has to be in the rectangle.

But why?

D & C closest pair (9)

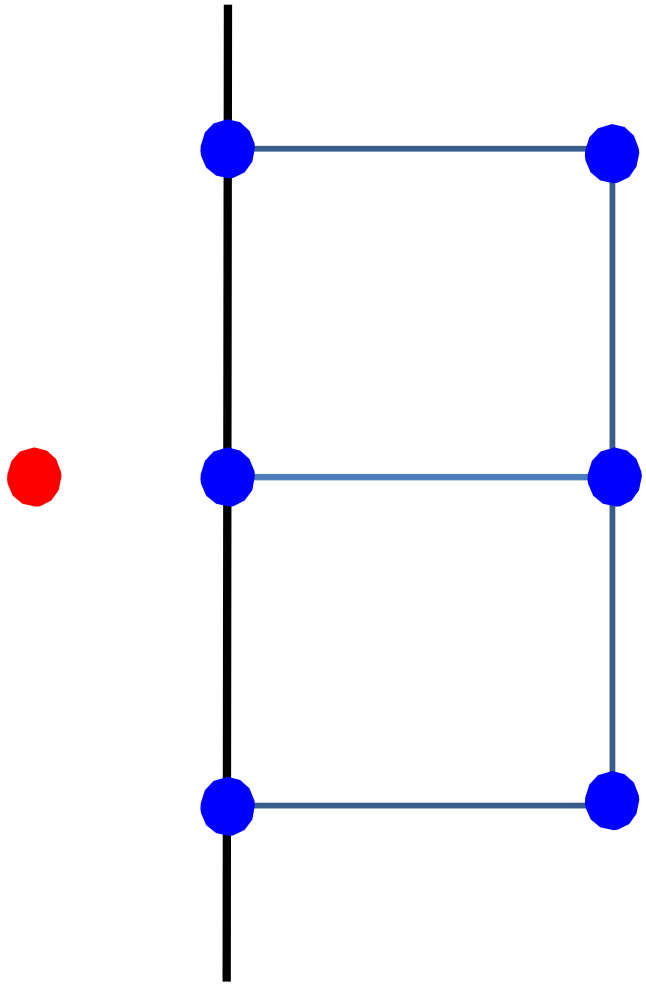




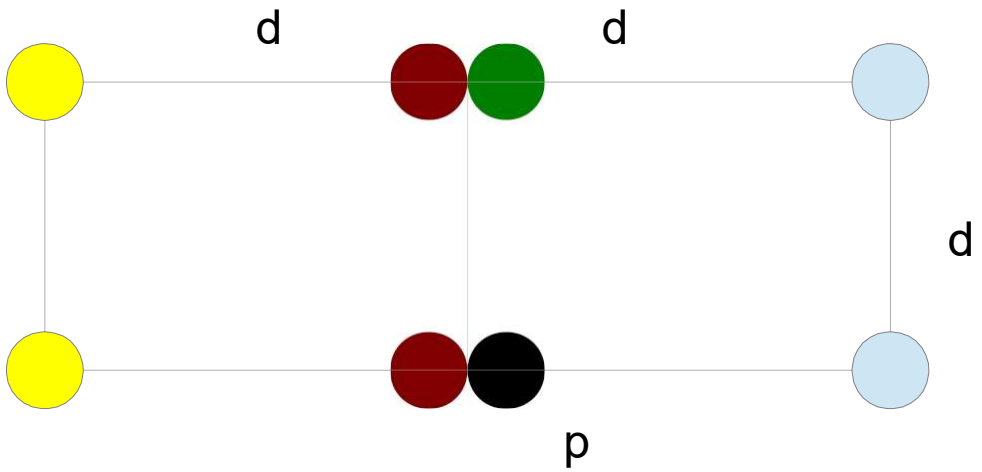
Take a look at point p1. It's a best case scenario for a point outside the rectangle ABOVE p.

Clearly its distance is greater than d_{min}.

D & C closest pair (10)



D & C closest pair (11)



This is the most densely packed the rectangle can be without contradicting the finding that d is the minimum space between any two points in each half. So there are at most 8 points.

But if you look at this carefully, you'll notice that even the red points cannot be there, because either there must be a tiny distance less than d between the red points and the yellow points or they must be exactly the same points as p and the green point.
Therefore max 6 points.

D & C closest pair (12)

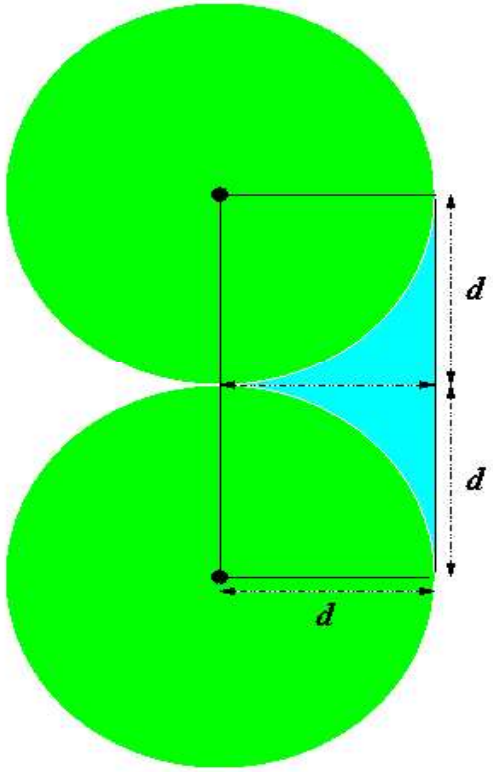
If you are unconvinced by the previous slide, here is an intuitive proof that the rectangle has 6 points.

If the previous slide was sufficient proof for you then skip the next two slides.

The source for all slides dealing with this proof is:
http://www.cs.mcgill.ca/~cs251/ClosestPair/proofbox.html

D & C closest pair (13)

Draw a circle of radius d around each point representing the area that we are not allowed to insert another point into. We can minimize the overlapping area of such a circle with the rectangle by placing the point on the corner of the rectangle.



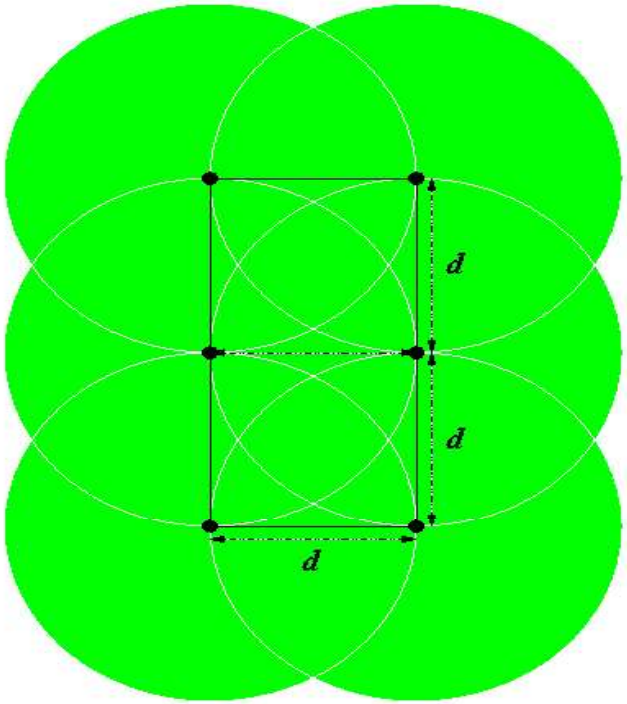
Source: http://www.cs.mcgill.ca/~cs251/ClosestPair/proofbox.html

D & C closest pair (14)

Once we've placed all the points, this is what we get.

Note no point is in any other point's circle. But the circles themselves obviously overlap.

We can't place any more points without violating circle space.



Source: http://www.cs.mcgill.ca/~cs251/ClosestPair/proofbox.html

D & C Closest Pair Overview

1. **Sort** points according to their **x-coordinates**.
2. **Split** the set of points **into two** equal-sized subsets by a vertical line $x=x_{median}$.
3. Solve the problem **recursively** in the left and right subsets, so that we get the left-side and right-side minimum distances d_l and d_r. $d_{min}=\min(d_l, d_r)$
4. Find the minimal distance in the set S of points of **width 2d around the vertical** line. Update d_{min} if necessary.

Adapted from: http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

D&C closest pair (16)
Pseudo-code

Sort points in order of x co-ordinates into array P[0..n-1].

Sort P in y co-ordinate order into array Q[0..n-1].

EfficientClosestPair(P, Q)

Pseudocode continues on next slide

D&C closest pair (17)

```
double EfficientClosestPair(P, Q):
   if n <= 3: return minimal distance found by brute force
   copy the first ceiling[n/2] points of P to Pl (l for left)
   copy the same points of Q to Ql
   copy the remaining floor[n/2] points of P to Pr
   copy the same points of Q to Qr
   dl = EfficientClosestPair(Pl, Ql)
   dr = EfficientClosestPair(Pr, Qr)
   d = min(dl, dr)
   m = P[ceiling(n/2)]-1].x
   copy all points of Q for which |x-m|<d into S[0..num-1]
   dminsq = d * d
   for i = 0 to num - 2:
      k = i + 1
      while k <= num - 1 and (S[k].y - S[i].y)^2 < dminsq:
         dminsq = min((S[k].x-S[i].x)^2+(S[k].y-S[i].y)^2,
                      dminsq)
         k = k + 1
   return sqrt(dminsq)
```

D&C closest pair

- How efficient is this algorithm?
  - The pre-sorting is $\Theta(n \log n)$
  - Leaving aside the recursion every other step in the algorithm is at worst $\Theta(n)$.
- So we can set up a recurrence relation in terms of a function T of whatever basic operation you choose.
- Recursion is executed twice. So we have

  $T(n) = 2\,T(n / 2) + f(n)$
- Now we have seen that f(n) is in $\Theta(n)$ because there are at most 6 points to consider each time.

D&C closest pair

master theorem:

If $T(n) = aT(n/b) + f(n)$ and $f(n) \in \Theta(n^d)$
then

$T(n) \in \Theta(n^d)$ if $a < b^d$

$T(n) \in \Theta(n^d \log n)$ if $a = b^d$

$T(n) \in \Theta(n^{\log_b a})$ if $a > b^d$

b = 2 and b = 2 and d = 1

so $a = b^d$

So $T(n) \in \Theta(n^d \log n)$

$T(n) \in \Theta(n \log n)$.

main algorithm is the same efficiency class as the pre-sorting part.