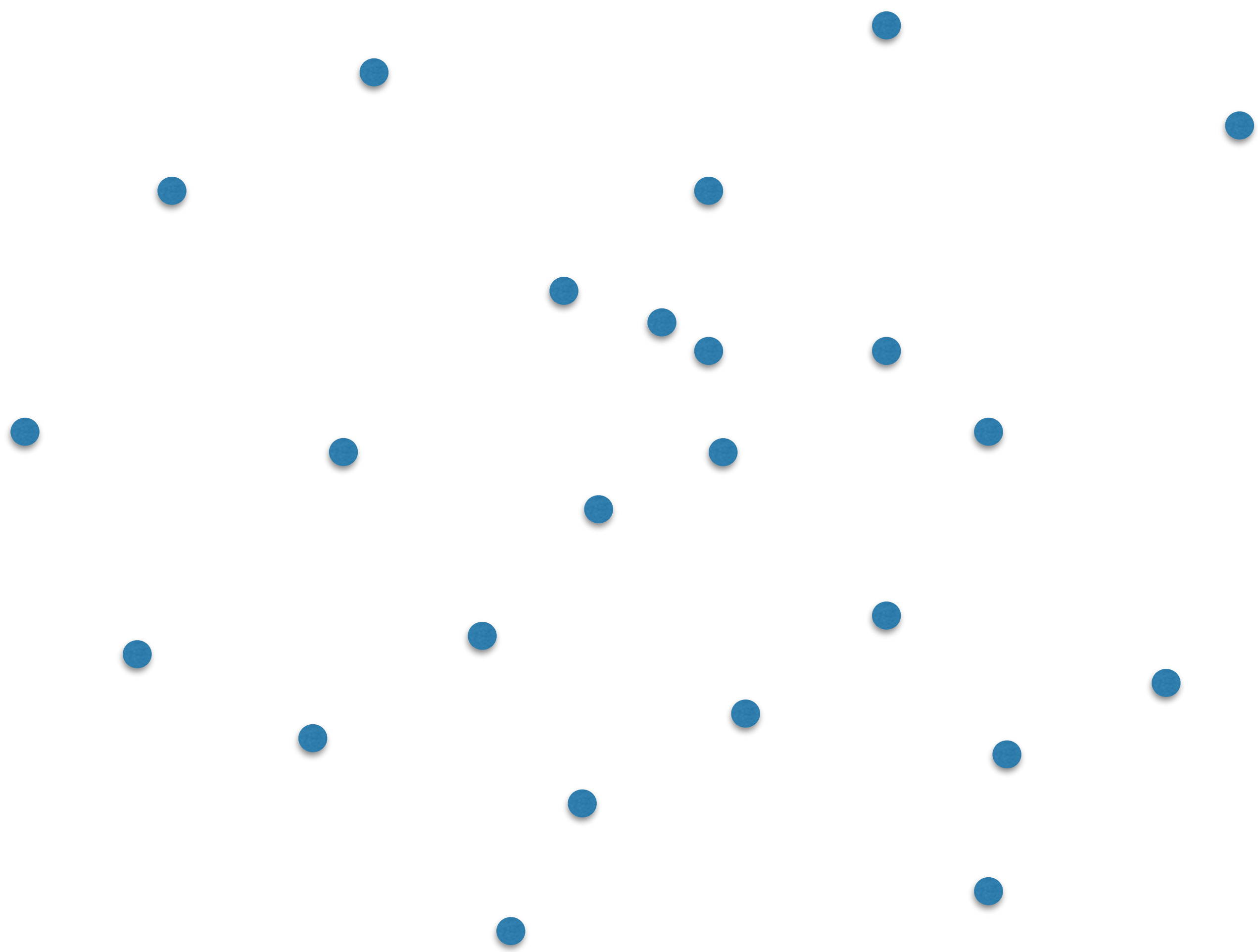
Finding the closest pair

Computational Geometry [csci 3250]
Laura Toma
Bowdoin College

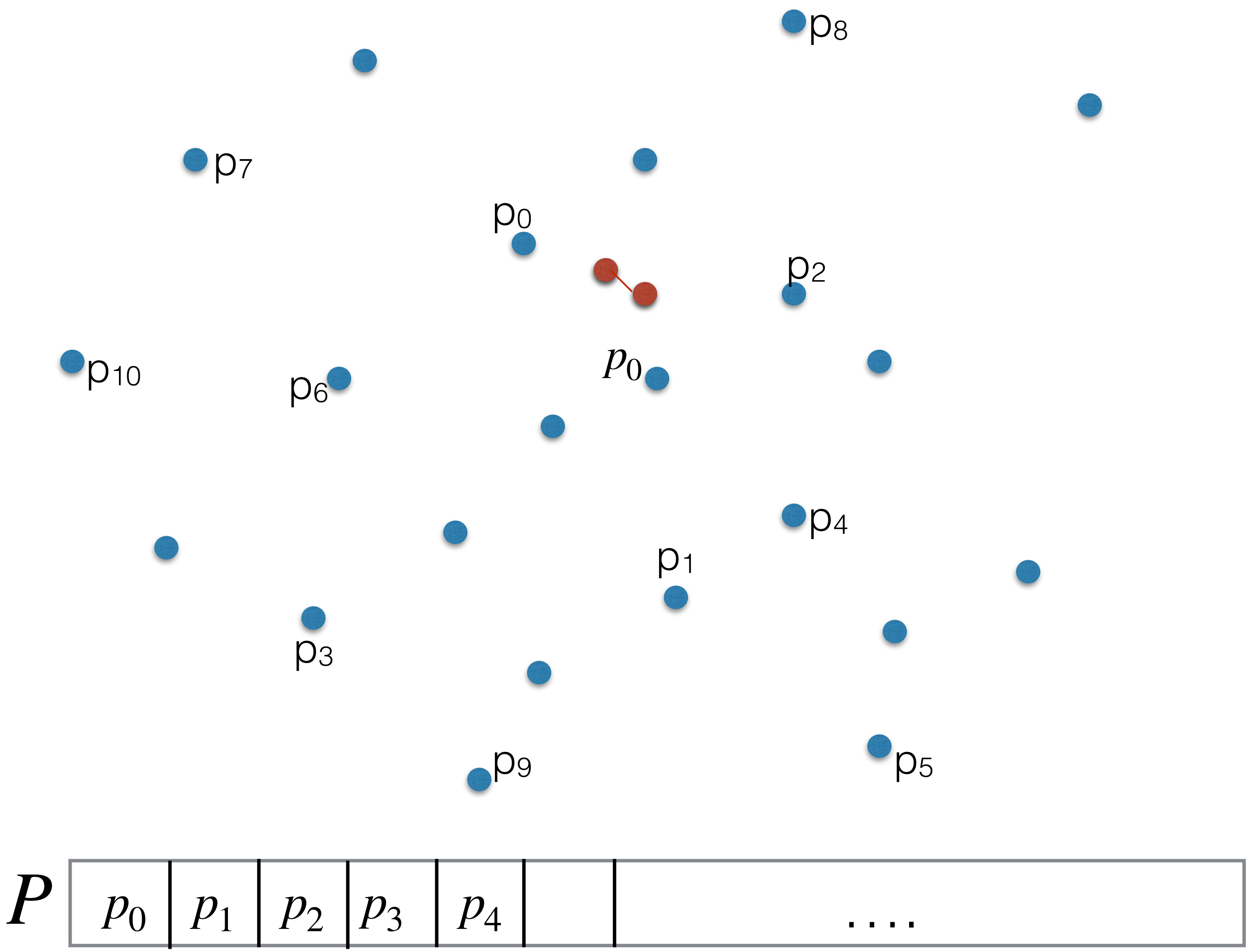
Given an array of points in 2D, find the **closest** pair.

In terms of the Euclidian distance



Given an array of points in 2D, find the **closest** pair.

In terms of the Euclidian distance



Given an array of points in 2D, find the **closest** pair.

Brute force:

- `mindist = VERY_LARGE_VALUE`
- `for all distinct pairs of points` p_i, p_j
    - `d = distance` (p_i, p_j) ← $d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
    - `if (d < mindist): mindist=d`

- Analysis:
    - $O(n^2)$ pairs ==> $O(n^2)$ time

Can we do better than $O(n^2)$?

Divide-and-conquer refresher

Divide-and-conquer

mergesort(array A)

- if A has 1 element, there's nothing to sort, so just return it
- else

  //divide input A into two halves, A1 and A2

  - A1 = first half of A
  - A2 = second half of A

  //sort recursively each half

  - sorted_A1 = **mergesort**(array A1)
  - sorted_A2 = **mergesort**(array A2)

  //merge

  - result = merge_sorted_arrays(sorted_A1, sorted_A2)
  - return result

Analysis: $T(n) = 2T(n/2) + O(n) \Rightarrow O(n \lg n)$

D&C, in general

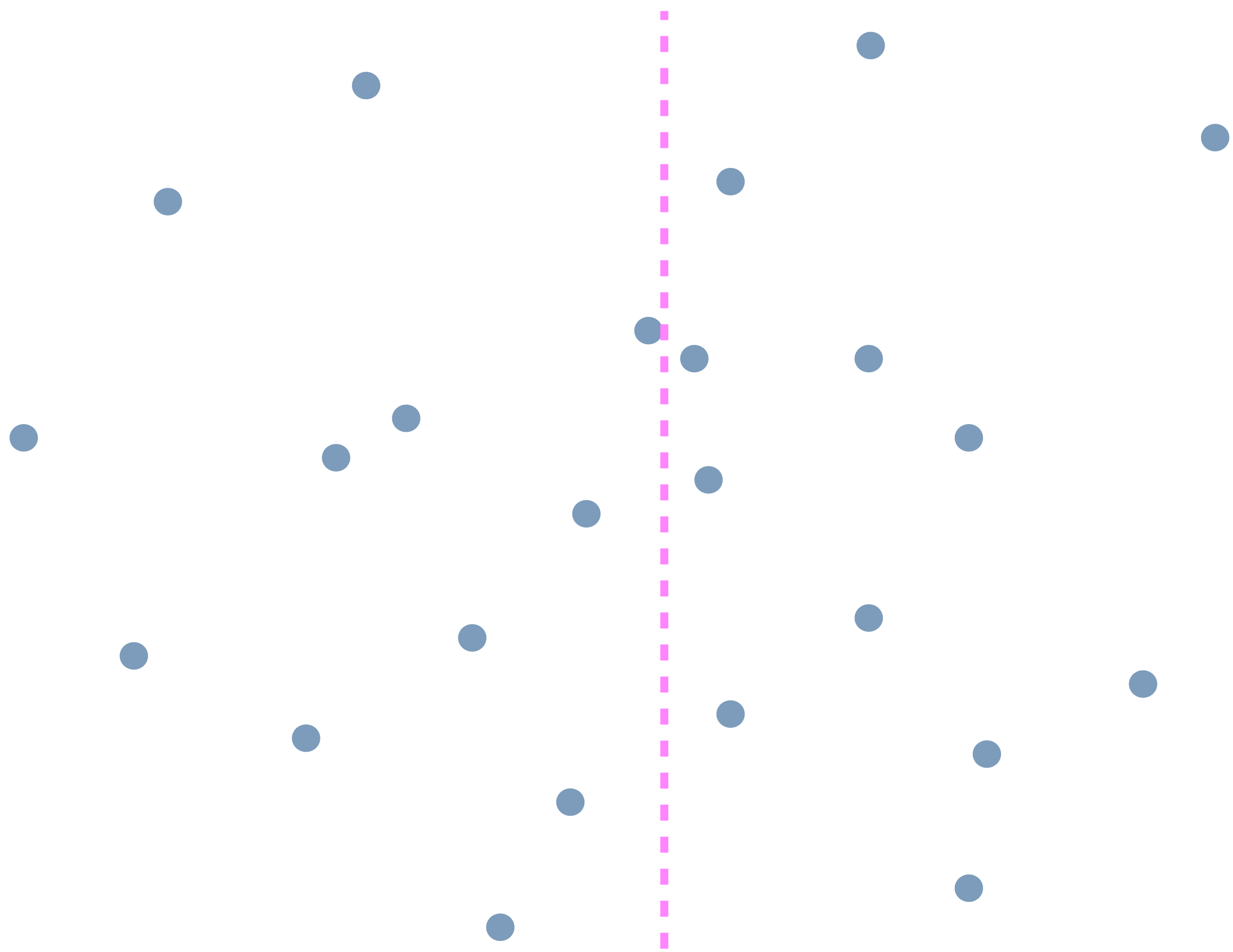
```
DC(input P)
  if P is small, solve and return
  else
    //divide
    divide input P into two halves, P1 and P2
    //recurse
    result1 = DC(P1)
    result2 = DC(P2)
    //merge
    do_something_to_figure_out_result_for_P

    return result
```

Analysis: $T(n) = 2T(n/2) + O(\text{merge phase})$

- if merge phase is **O(n)**: $T(n) = 2T(n/2) + O(n) \Rightarrow O(n \lg n)$
- if merge phase is **O(n lg n)**: $T(n) = 2T(n/2) + O(n \lg n) \Rightarrow O(n \lg^2 n)$

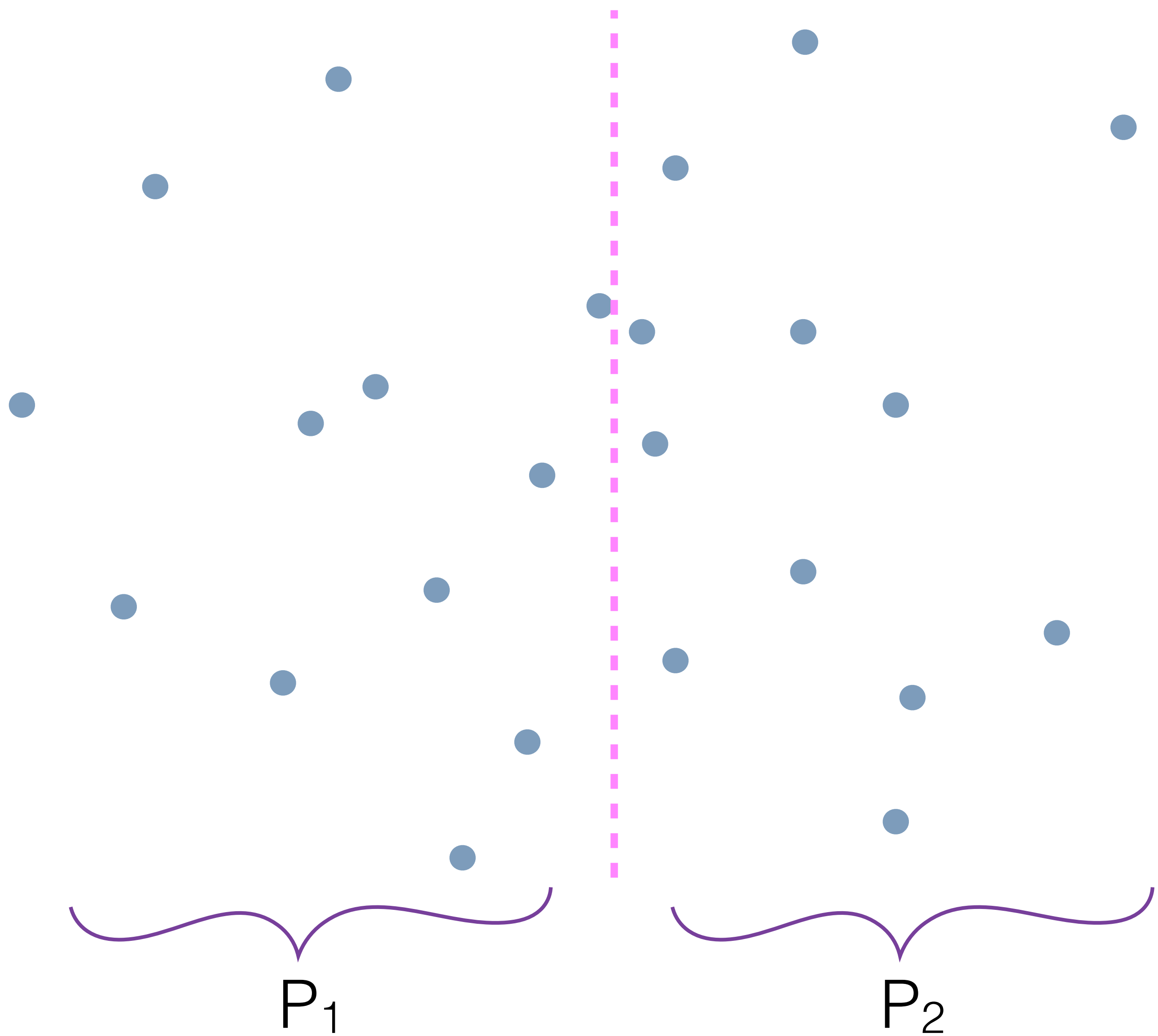
Closest pair, divide-and-conquer

- find vertical line that splits P in half



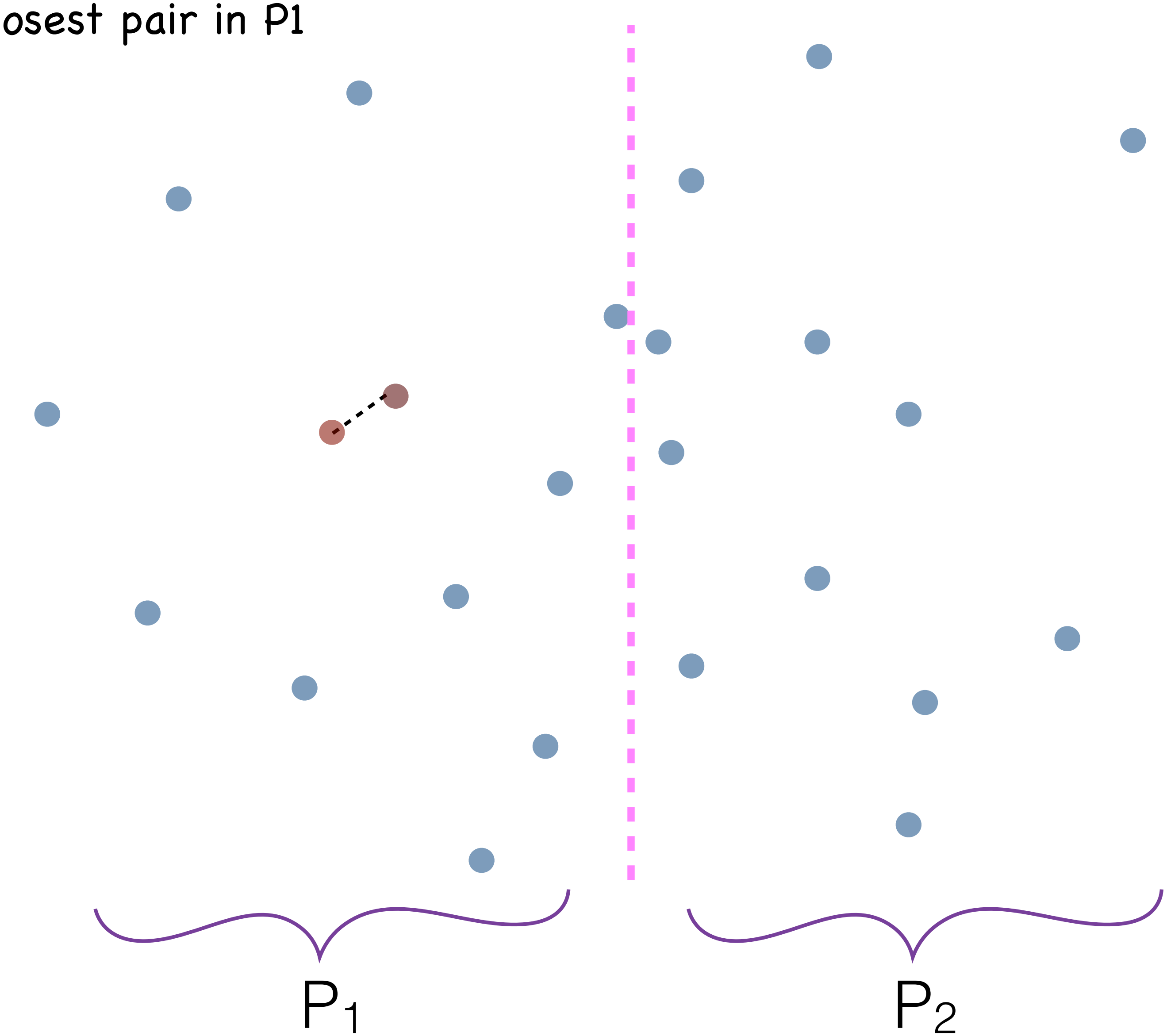
Closest pair, divide-and-conquer

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line



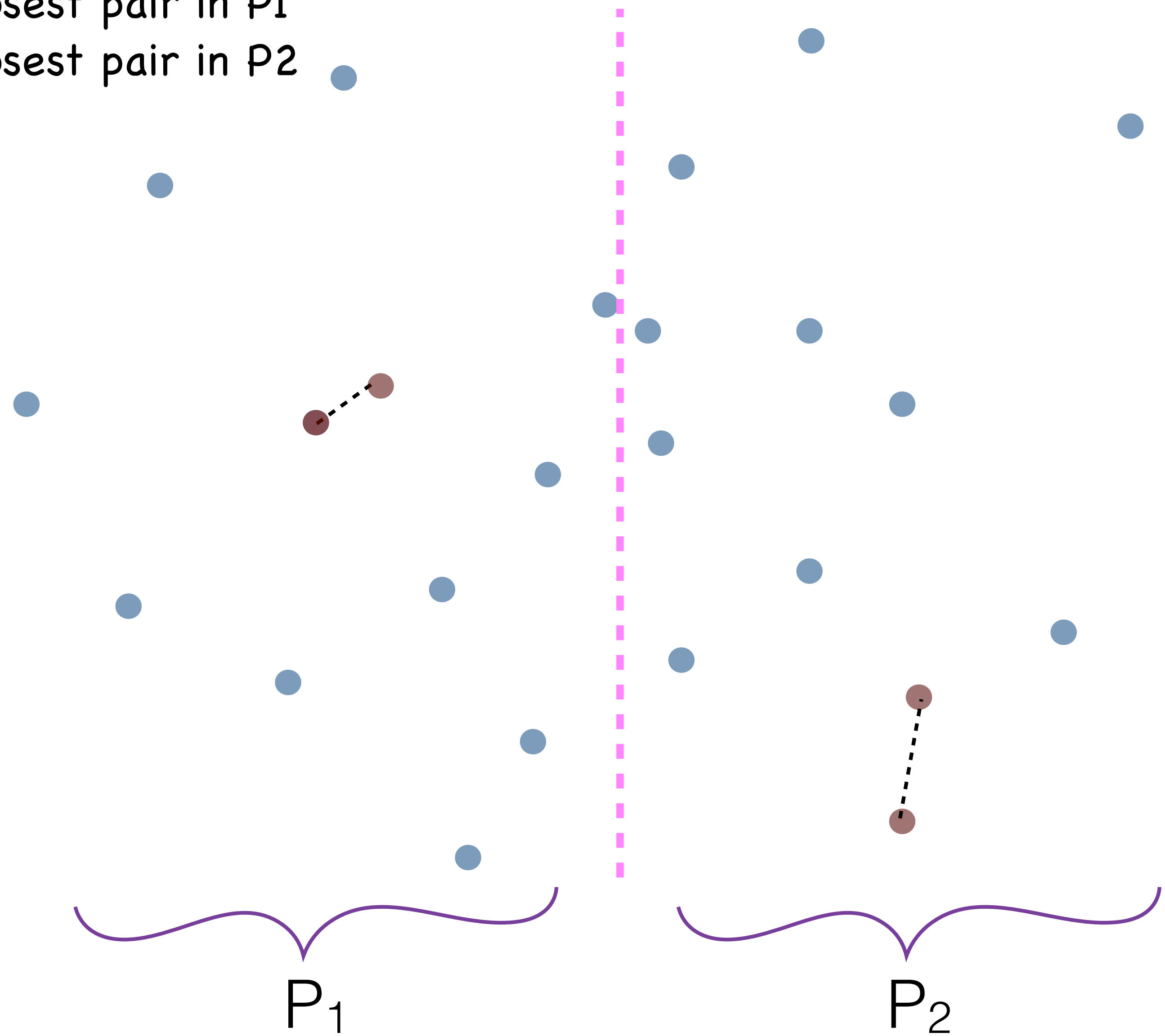
Closest pair, divide-and-conquer

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find closest pair in P1



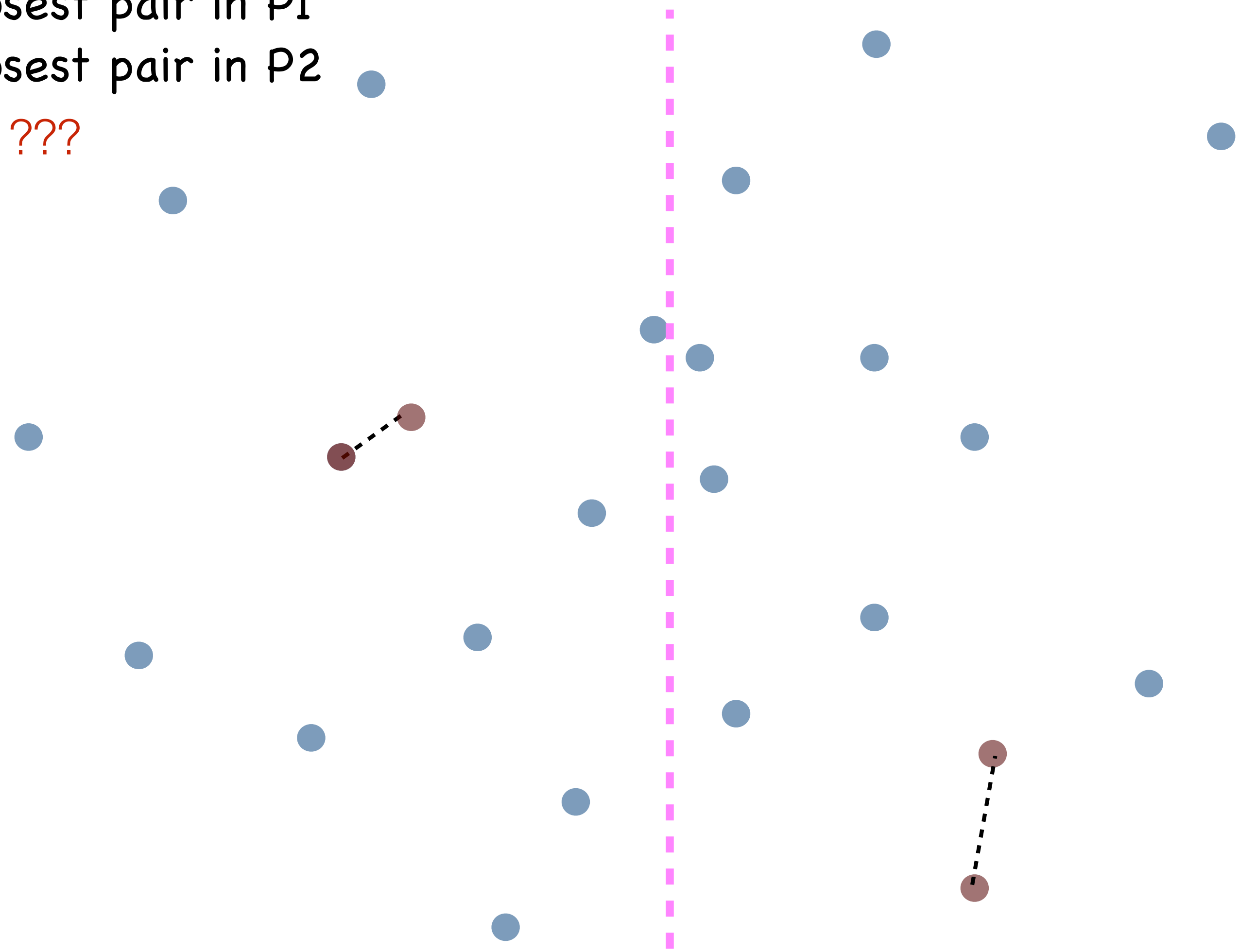
Closest pair, divide-and-conquer

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find closest pair in P1
- recursively find closest pair in P2



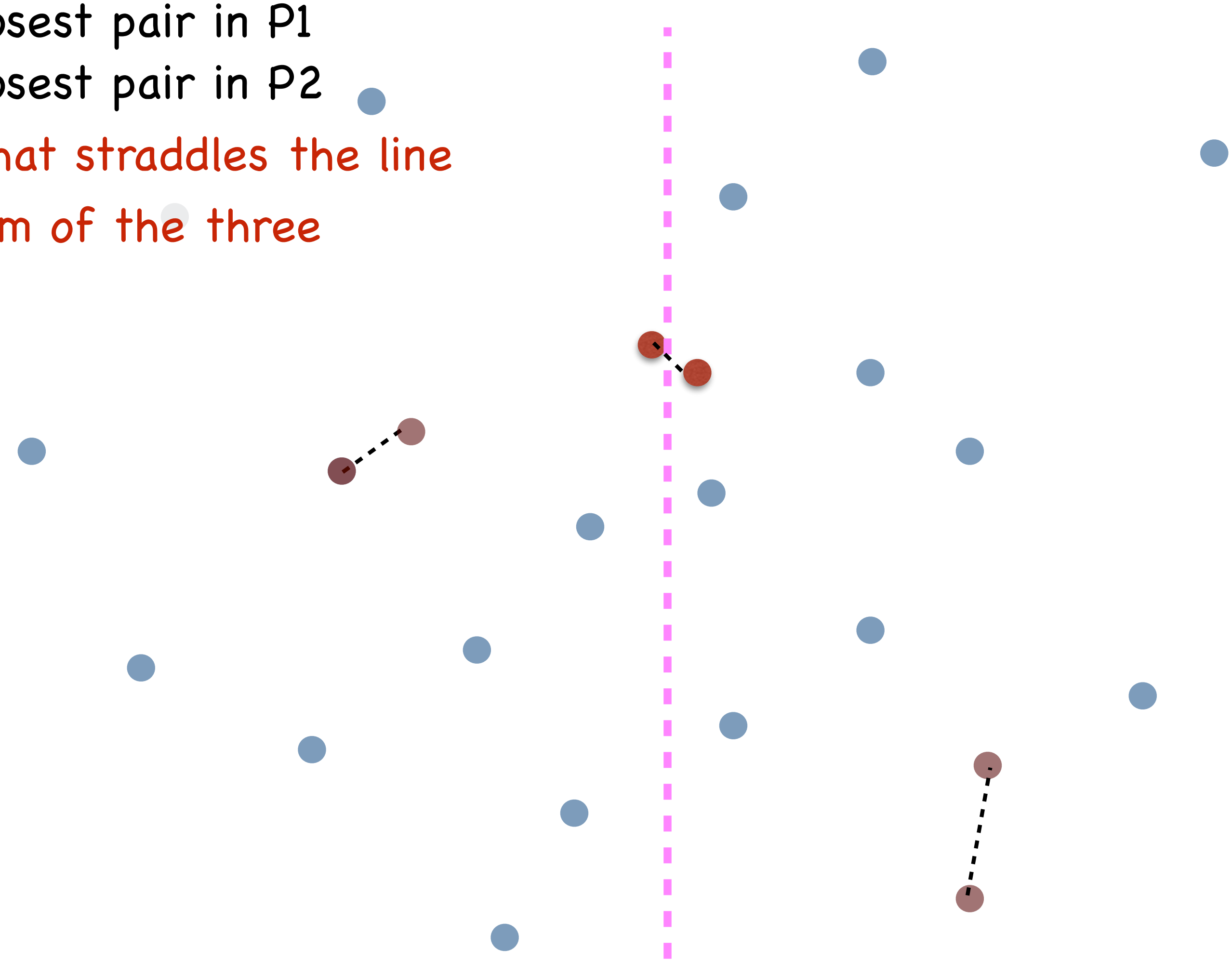
Closest pair, divide-and-conquer

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find closest pair in P1
- recursively find closest pair in P2
- //...... NOW WHAT ???



Closest pair, divide-and-conquer

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find closest pair in P1
- recursively find closest pair in P2
- find closest pair that straddles the line
- return the minimum of the three



Closest pair, divide-and-conquer

FindClosestPair(P)

//basecase

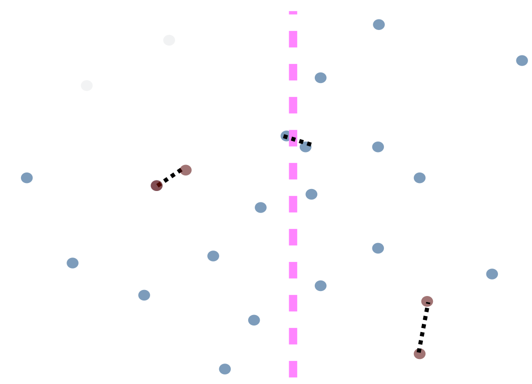
- if P has 1 point, return infinity
- if P has 2 points, return their distance
- else
  - find vertical line that splits P in half
  - let P1, P2 = set of points to the left/right of line
  - d_1 = FindClosestPair(P1)
  - d_2 = FindClosestPair(P2)

  //compute closest pair across

  - mindist=infinity
  - for each p in P_1, for each q in P_2
    - compute distance d(p,q)
    - mindist = min{mindist, d(p,q)}

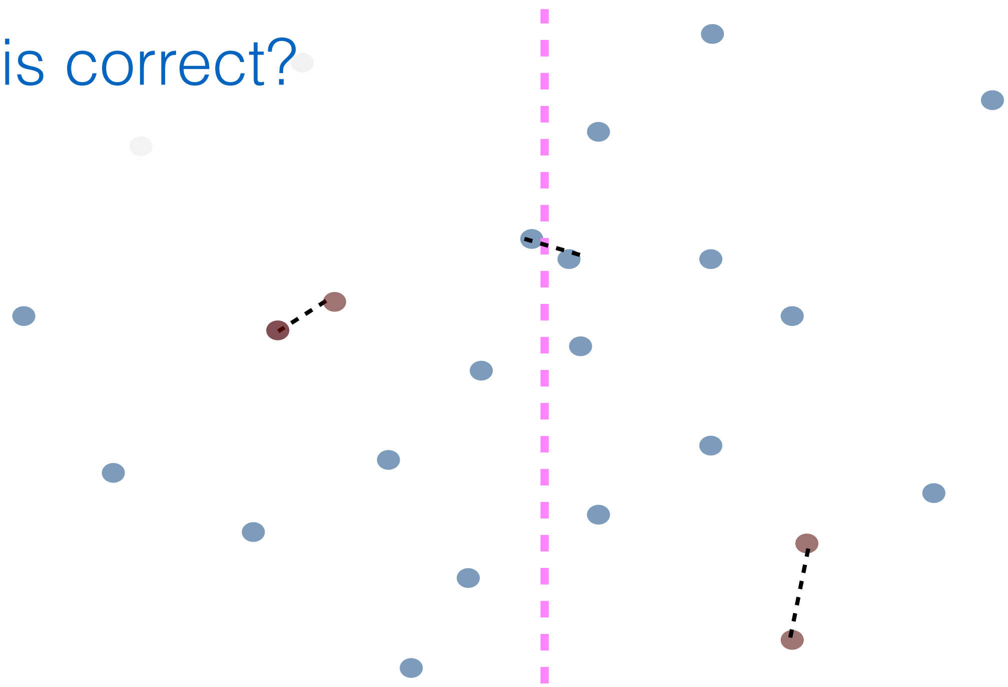
  //return smallest of the three

  - return min {d_1, d_2, mindist}



1. Is this correct?
2. Running time?

1.Why is this correct?



The closest pair in P falls in one of three cases:

- **Both points are in P1:** then it is found by the recursive call on P1
- **Both points are in P2:** then it is found by the recursive call on P2
- **One point is in P1 and one in P2:** then it is found in the merge phase, because the merge phase considers all such pairs

2. Running time

FindClosestPair(P)

//basecase

- if P has 1 point, return infinity
- if P has 2 points, return their distance
- else
  - find vertical line that splits P in half
  - let P1, P2 = set of points to the left/right of line
  - d_1 = FindClosestPair(P1)
  - d_2 = FindClosestPair(P2)

  //compute closest pair across
  - mindist=infinity
  - for each p in P_1, for each q in P_2
    - compute distance d(p,q)
    - mindist = min{mindist, d(p,q)}

  //return smallest of the three
  - return min {d_1, d_2, mindist}

this merge is too slow

$T(n) = 2T(n/2) + O(n^2)$

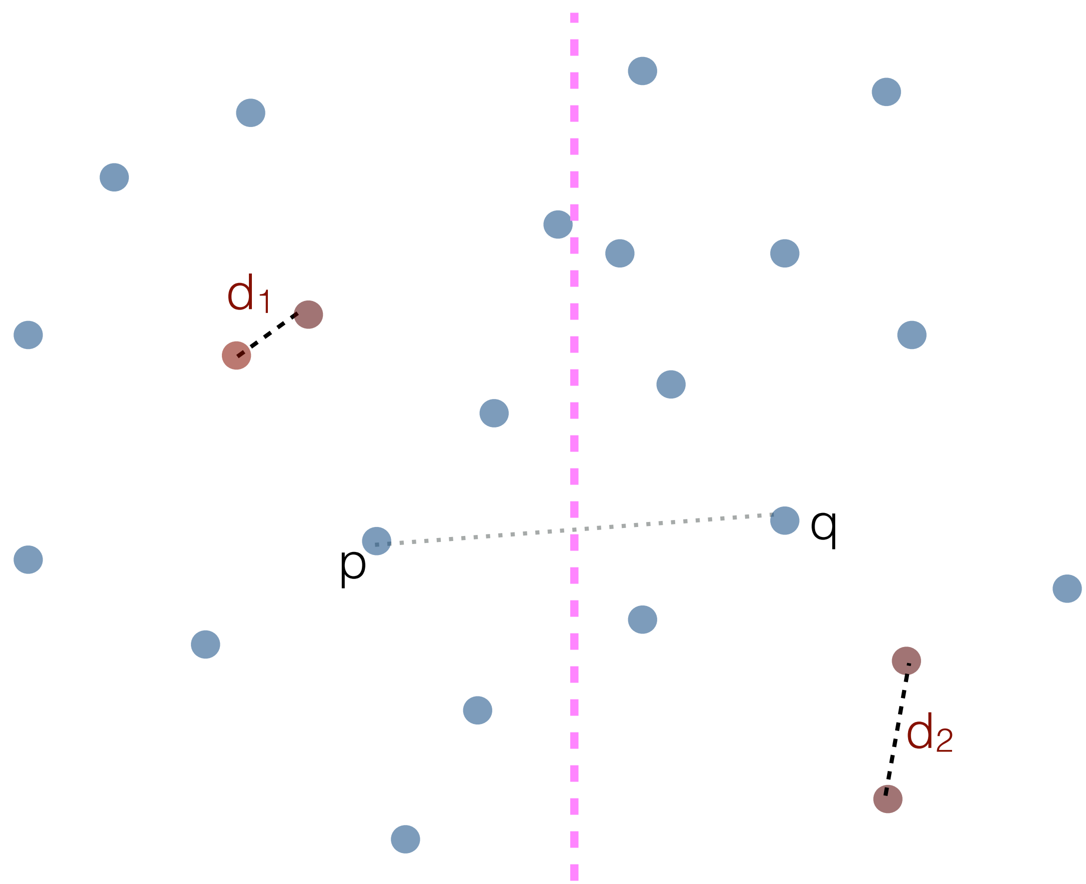
solves to $O(n^2)$

Can we do better?

Refining the merge

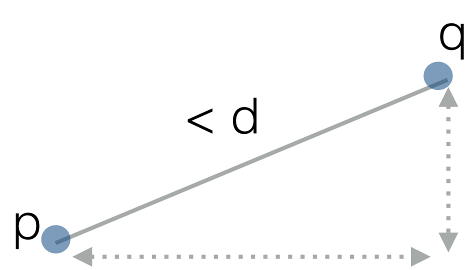
Do we need to examine all pairs p,q, with p in P_1, q in P_2?

Which pairs {p,q} can be discarded?

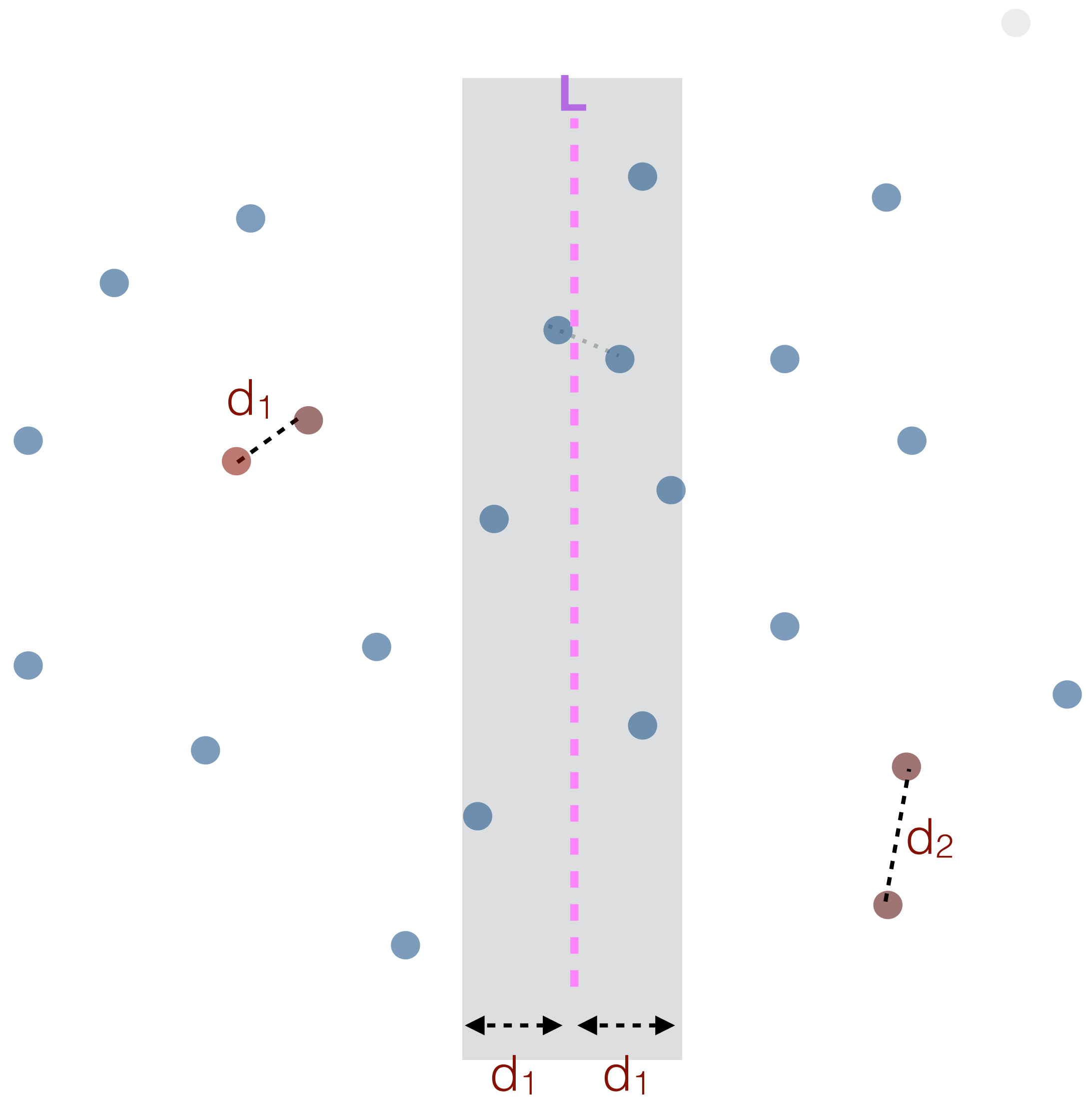


Here's a simple observation

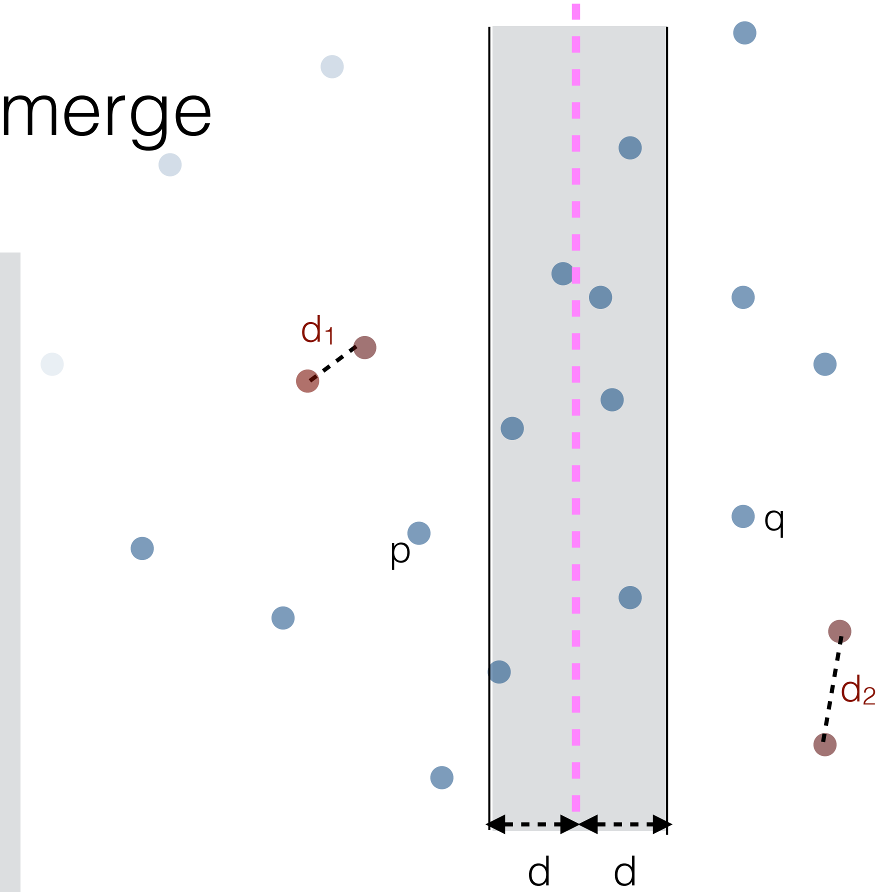
- Notation: $d = \min\{d_1, d_2\}$
- Observation: We are looking for points that are closer than d. If there is a pair of points p, q with $d(p, q) < d$, then both the horizontal and vertical distance between p and q must be smaller than d.



- Notation: $d = \min\{d_1, d_2\}$
- Furthermore, if there is a pair of points p, q with $d(p, q) < d$, then both p and q must be within distance d from line L.



Refining the merge



FindClosestPair(P)

//basecase

- if P has 1 point, return infinity
- if P has 2 points, return their distance
- else
  - find vertical line that splits P in half
  - let P1, P2 = set of points to the left/right of line
  - d_1 = FindClosestPair(P1)
  - d_2 = FindClosestPair(P2)

  //compute closest pair across
  - mindist=infinity
  - ~~for each p in P_1, for each q in P_2~~
    - ~~compute distance d(p,q)~~
    - ~~mindist = min{mindist, d(p,q)}~~

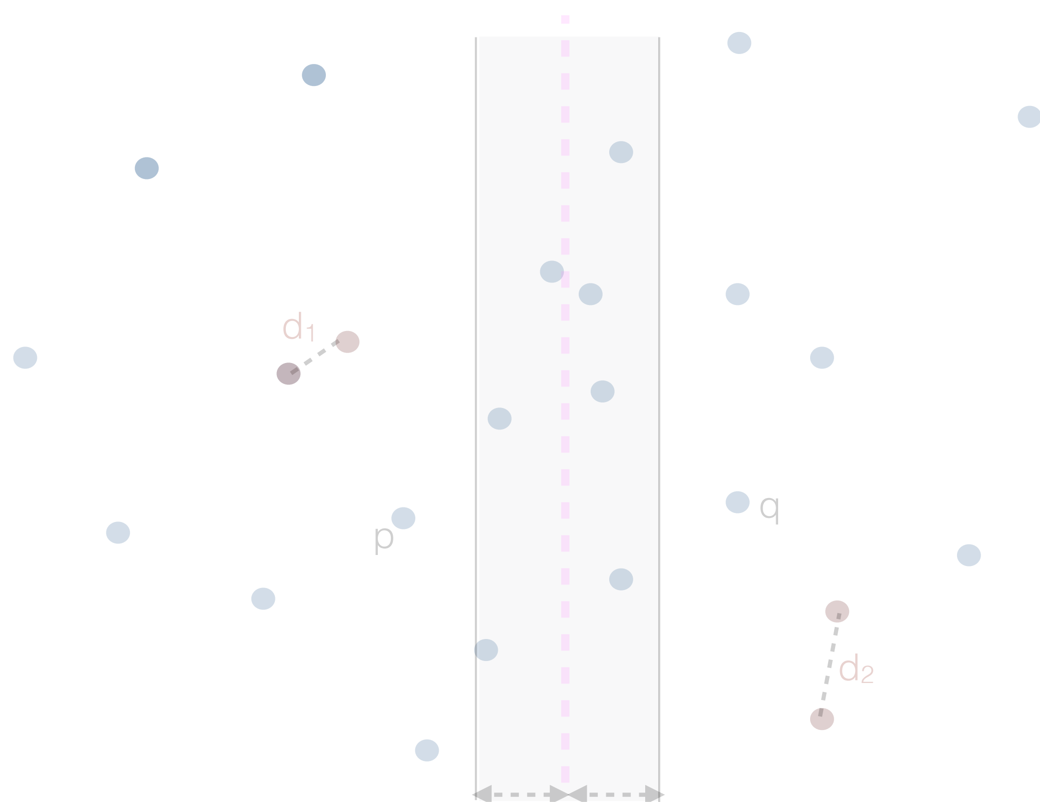
  //return smallest of the three
  - return min {d_1, d_2, mindist}

- traverse P1 and select all points P1' in the strip
- traverse P2 and select all points P2' in the strip
- for each p in P1', for each q in P2'
  - compute distance d(p,q)
  - mindist = min{mindist, d(p,q)}

Running time?

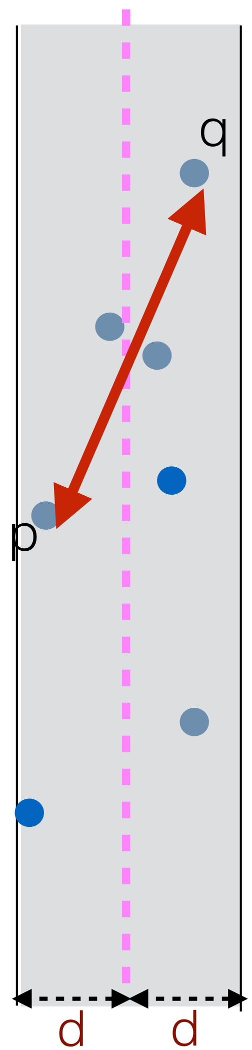
Running time

- How many points can there be in the strip?
- What does this imply for the running time?



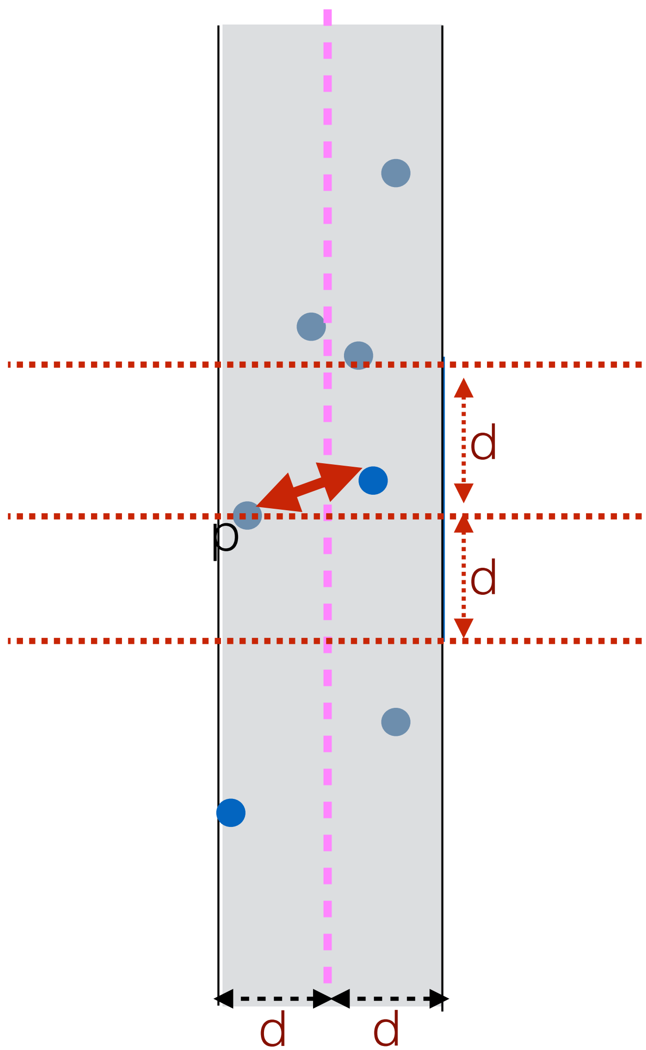
Refining some more

- Using the points in the strip is not enough, there can still be $\Omega(n)$ of them
- Note that the strip contains candidate pairs that could be within distance d of each other horizontally
- We haven't used yet that candidate pairs have to be within distance d of each other vertically



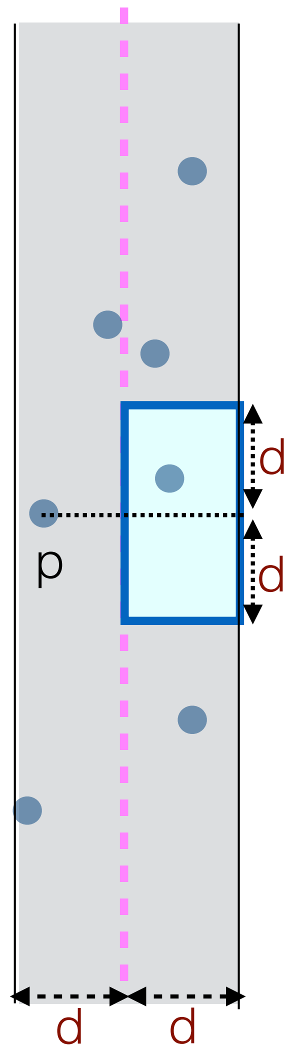
(p, q) not a candidate pair because their vertical distance $> d$

Refining some more



We are interested in the points q of P₂' whose distance to p is $< d$

These points are vertically above or below p by at most d



- For a point p in P_1': We only need to check the points on the other side that are vertically at most d above/below p

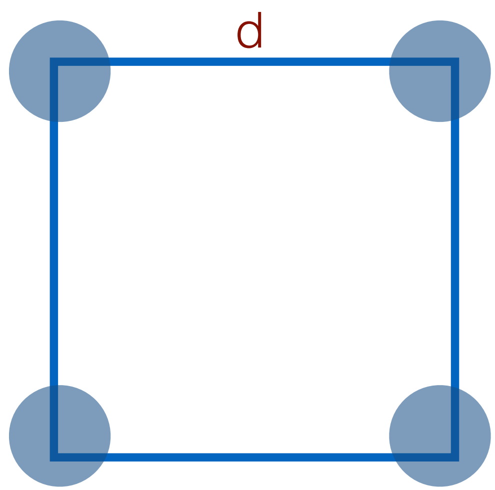
- How many such points can there be?

Let P be a set of points such that any two points are at least d away from each other.

Claim: Then any square with side d contains at most _____ points of P.

Let P be a set of points such that any two points are at least d away from each other.

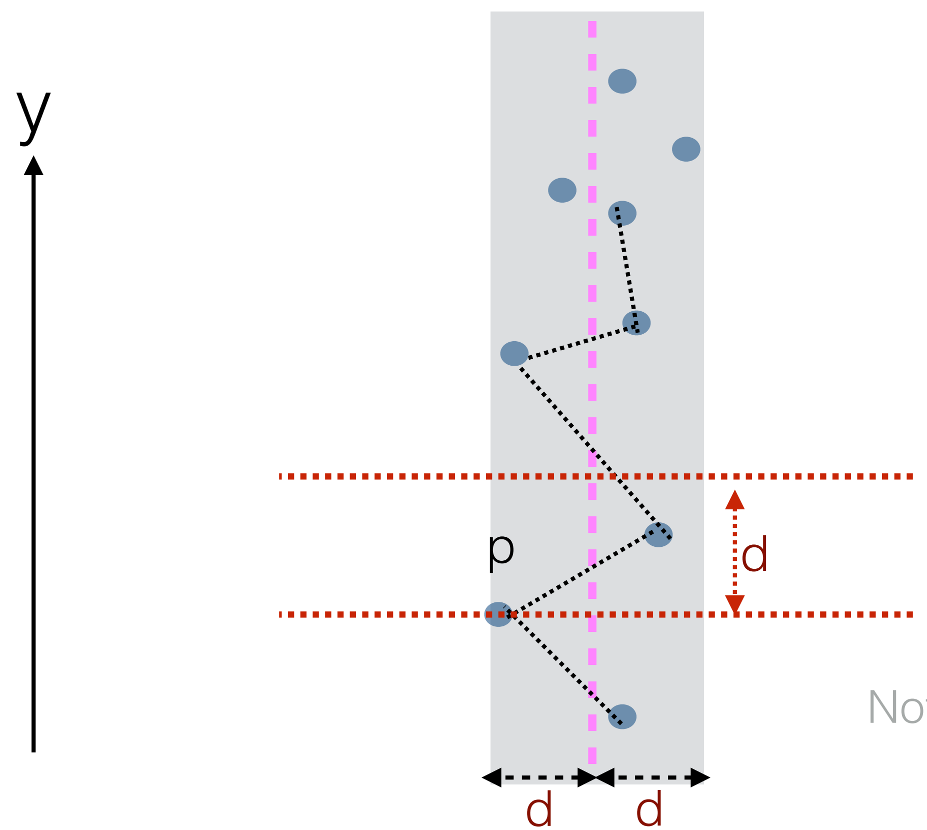
Claim: Then any square with side d contains at most **4** points of P.

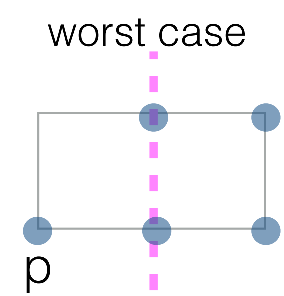


The new merge

- Traverse the points in P_1' and P_2' in increasing order of their y-coordinate
- Mimic the process of merging P_1' and P_2' in y-order
- Consider the next point p in y-order and let's say it comes from P_1'
    - p will check only the points in P_2' above it (following it in y-order) that are within d

    //There can be at most 4 subsequent points in P_2' that are within d from p.





Note: Assume no duplicate points.

closestPair(P)

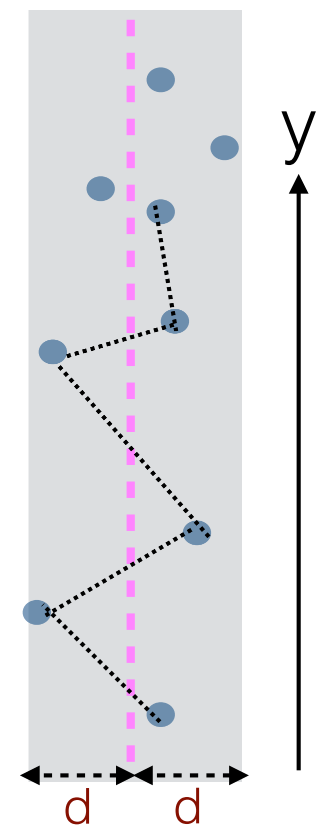
//divide

- find vertical line l that splits P in half
- let P_1, P_2 = set of points to the left/right of line
- d_1 = closestPair(P_1)
- d_2 = closestPair(P_2)

//refined merge

- let d = min{d_1, d_2}
- for all p in P_1: if $x_p > x_l - d$: add p to Strip1
- for all p in P_2: if $x_p < x_l + d$: add p to Strip2
- sort Strip1, Strip2 by y-coord
- initialize mindist=d
- merge Strip1, Strip2: for next point p,
    - compute its distance to the 4 points that come after it on the other side of the strip
    - if any of these is smaller than mindist, update mindist

- return min{d1, d2, mindist}



Analysis: $T(n) = 2T(n/2) + O(n \lg n) \Rightarrow O(n \lg^2 n)$

Can we do better?

We'd love to get rid of the extra lg n

Refining the refined merge

- Instead of **sorting inside every merge**, we'll pre-sort P **once** at the beginning
  - sort by x-coord: PX note: sorting by x is not necessary but practical
  - sort by y-coord: PY..

closestPair(PX,   PY)

- These sorted list will be maintained through the recursion

Refining the refined merge

closestPair(PX, PY)

//divide

- find vertical line L that splits P in half
- let P_1, P_2 = set of points to the left/right of line <— We need to get P1X, P1Y, P2X, P2Y
- d_1 = ~~closestPair(P_1)~~ closestPair(P1X, P1Y)
- d_2 = ~~closestPair(P_2)~~ closestPair(P2X, P2Y)

//merge

- let d = min{d_1, d_2} Traverse P1Y: if $x_p > x_L$-d: add p to Strip1
- ~~for all p in P_1: if $x_p > x_l - d$: add p to Strip1~~
- ~~for all p in P_2: if $x_p < x_l + d$: add p to Strip2~~
- ~~sort Strip1, Strip2 by y-coord~~ //Strip1, Strip2 are y-sorted!
- initialize mindist=d
- merge Strip1, Strip2: for next point p,
  - compute its distance to the 5 points that come after it on the other side of the strip
  - if any of these is smaller than mindist, update mindist
- return min{d1, d2, mindist}

Analysis: $T(n) = 2T(n/2) + O(n) \Longrightarrow O(n \lg n)$

Hooray!

Almost there..

- A few more details to think about
  - We have PX, PY
  - We need to:
    - Find the vertical line that splits P in half.
    - Get P1X, P2X.
    - Get P1Y, P2Y.