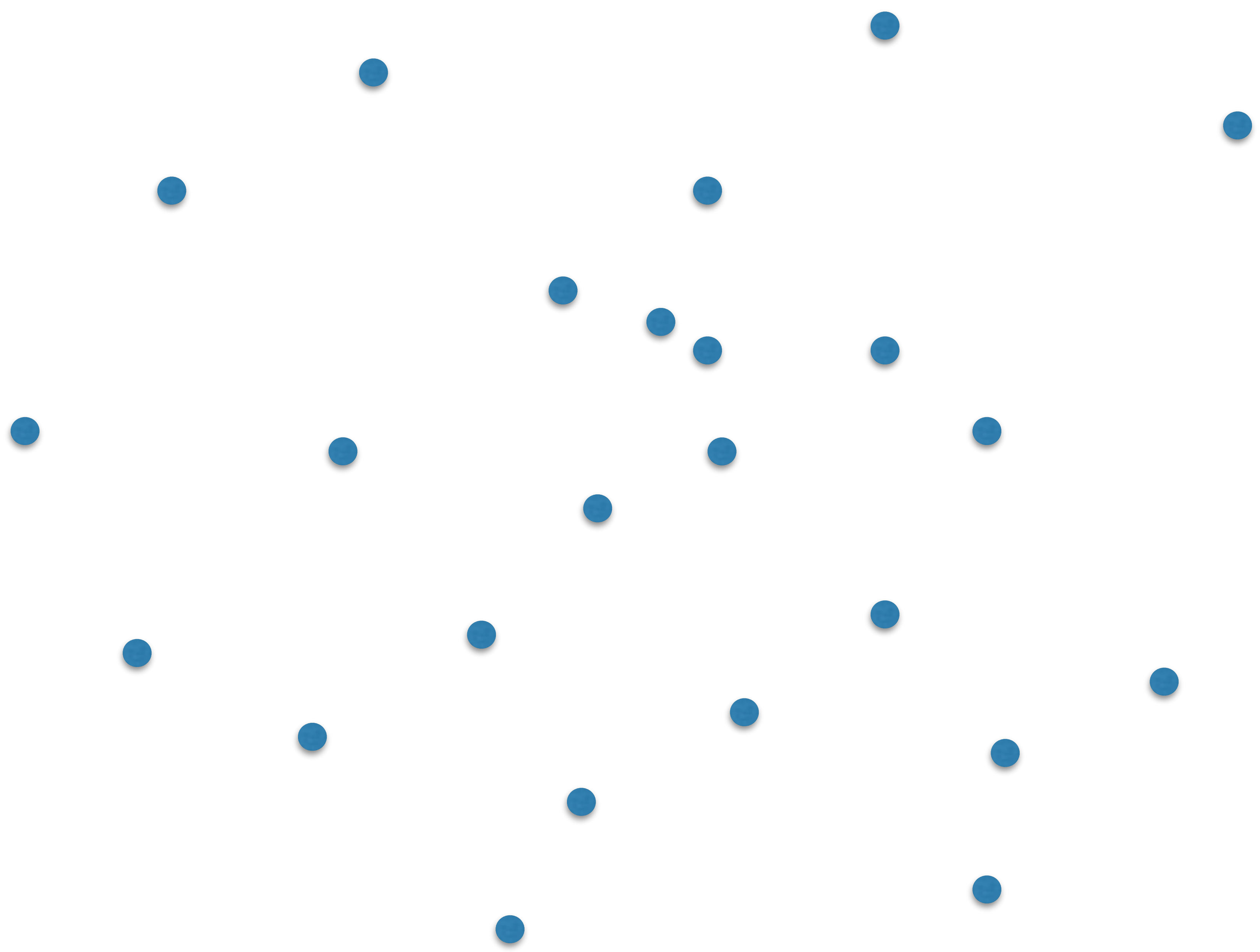
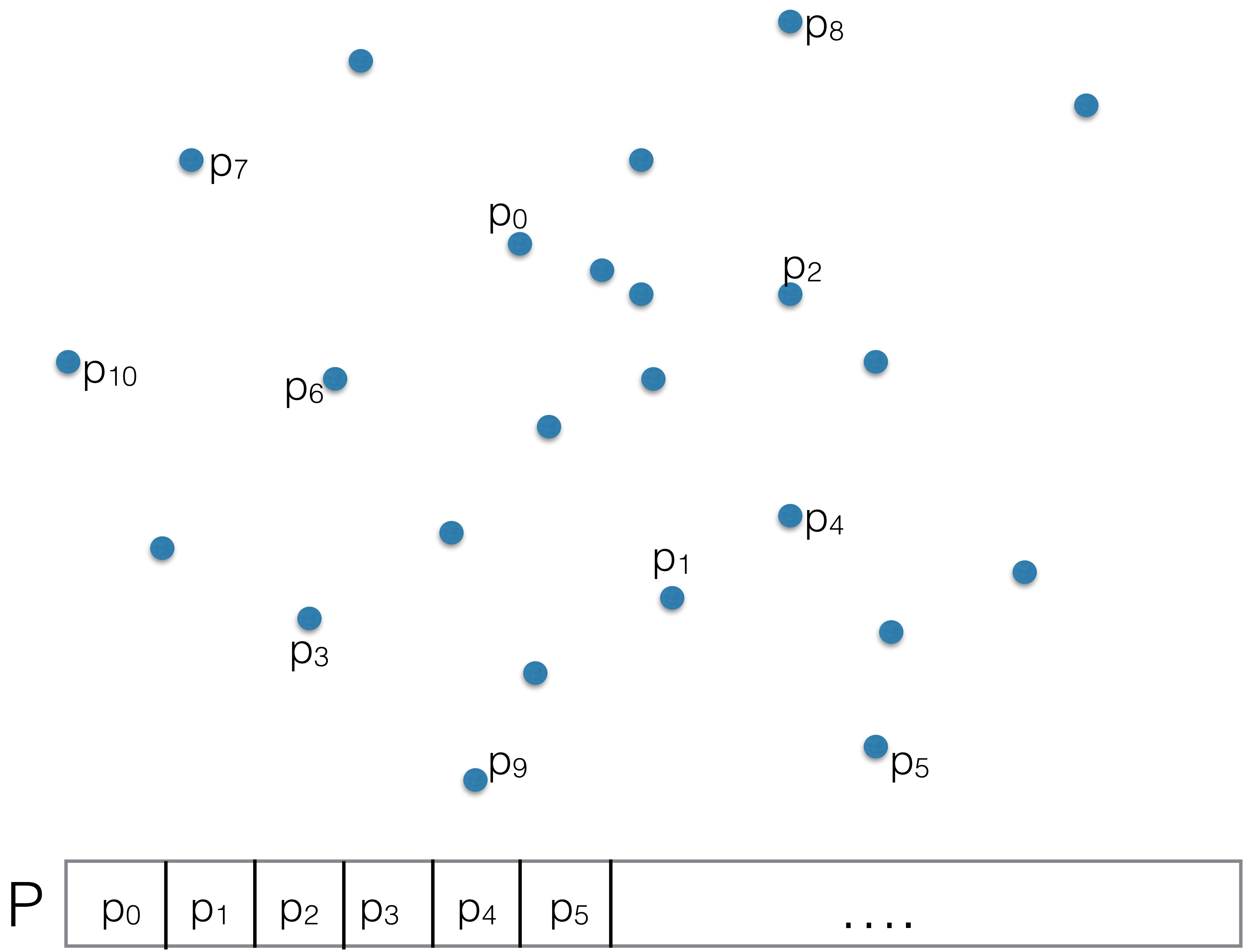
Finding the closest pair

Computational Geometry [csci 3250]
Laura Toma
Bowdoin College

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



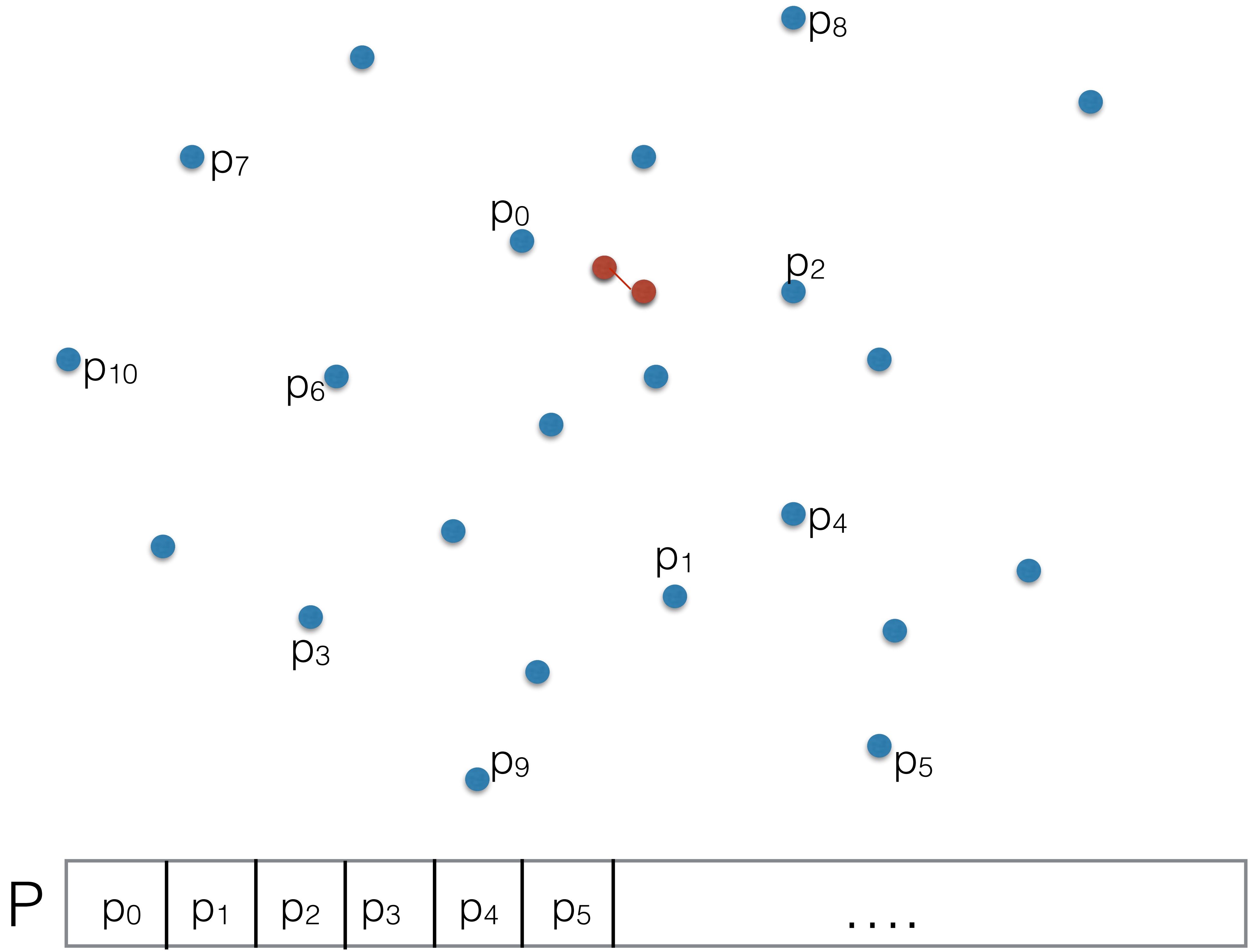
Given an array of points in 2D, find the closest pair.



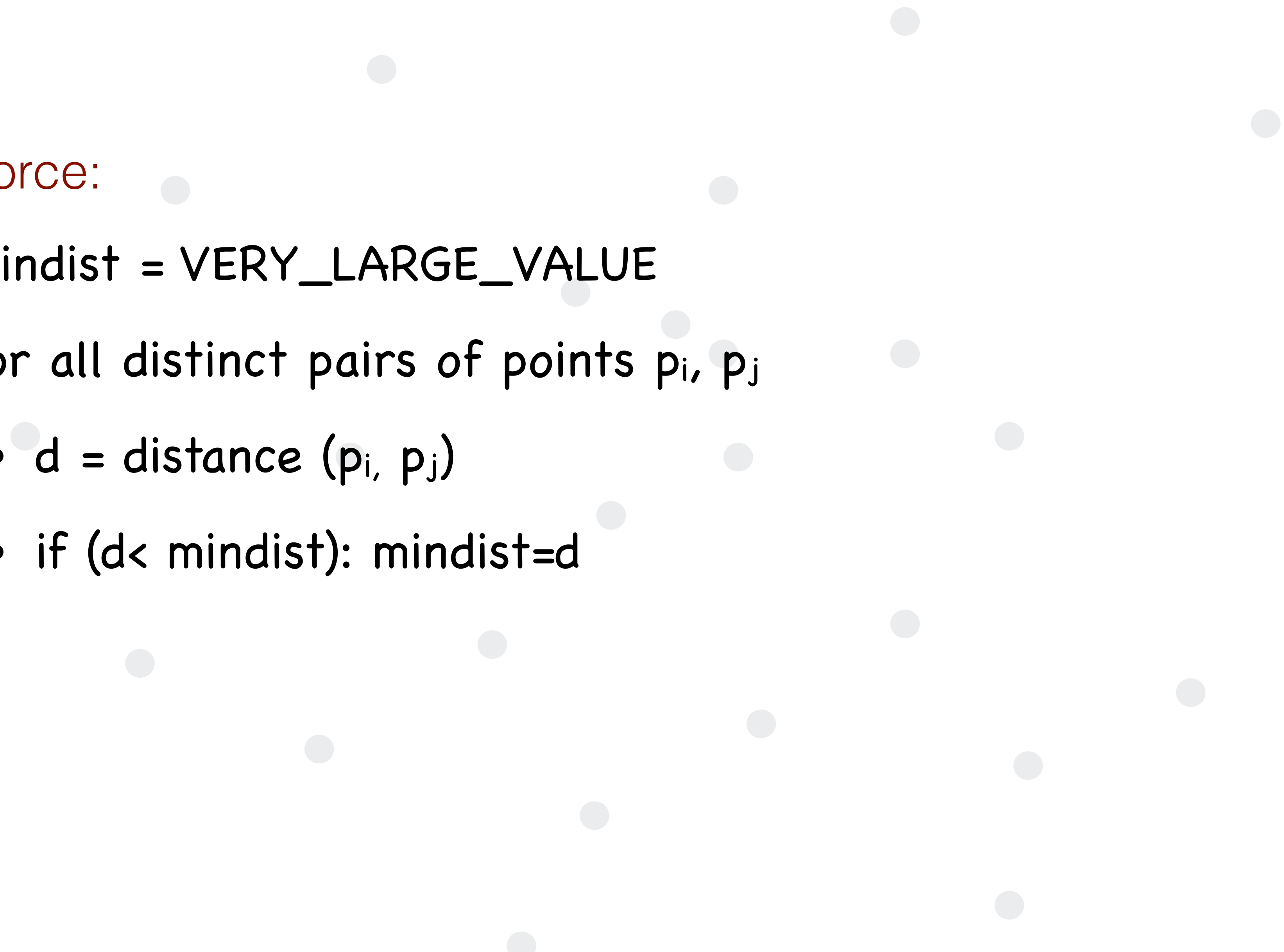
The distance between two points p and q is given by the Euclidian distance given by the formula:

$$d(p,q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$$

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



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)
  - if (d< mindist): mindist=d

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)
  - if (d< mindist): mindist=d

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

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)
    - if (d< mindist): mindist=d

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

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

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)
  - if (d< mindist): mindist=d

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

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

Hint: divide-and-conquer

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

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

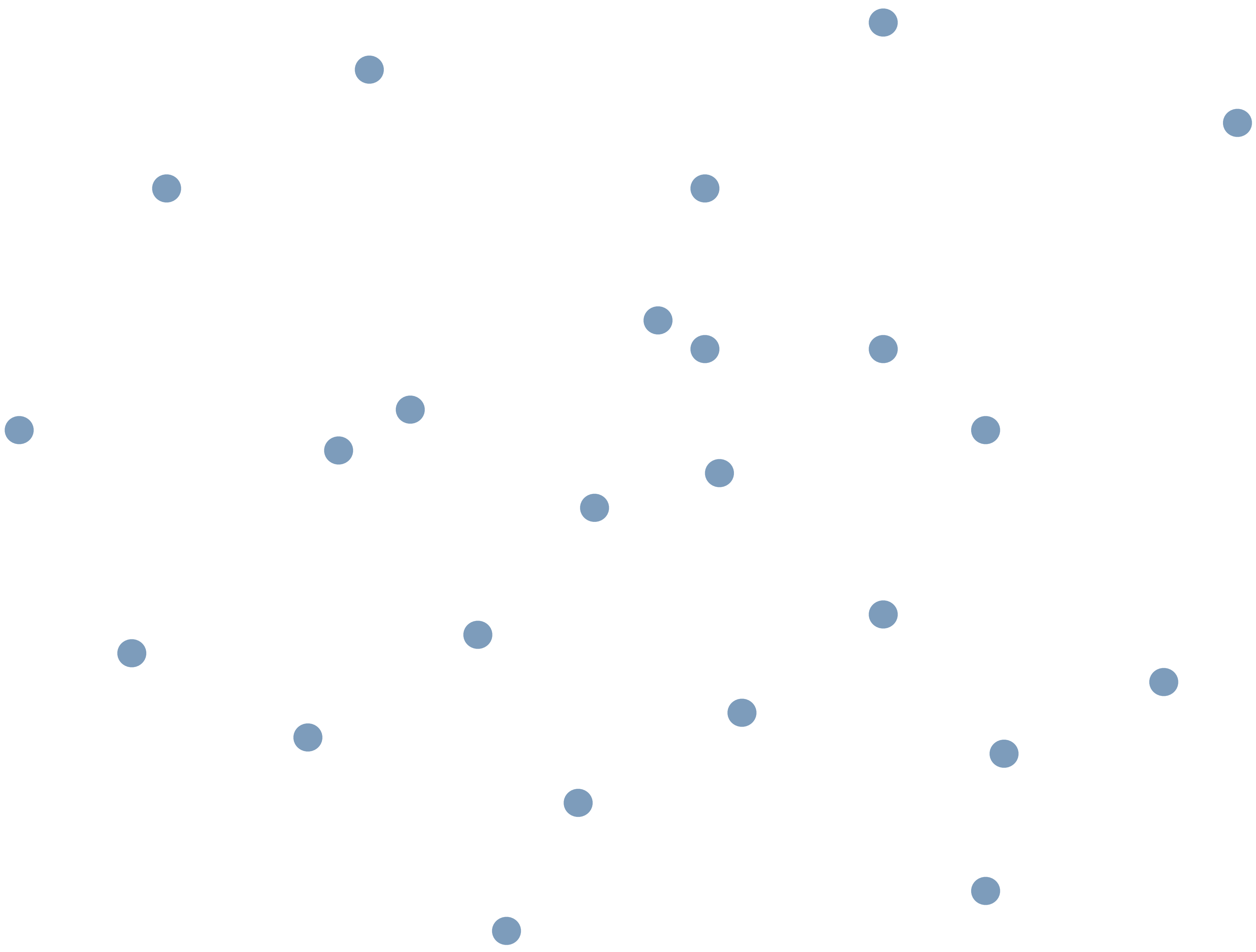
D&C, in general

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

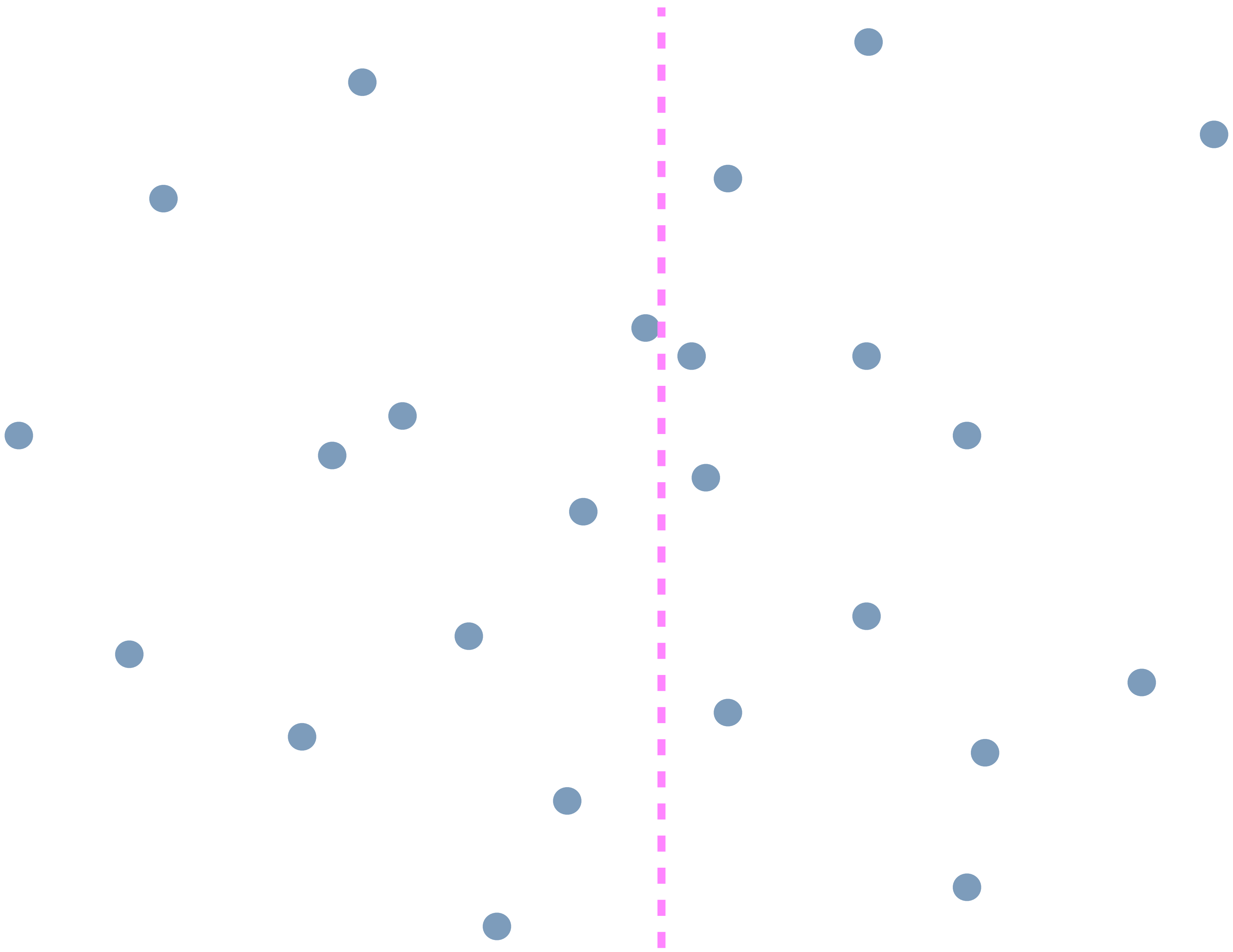
- if merge phase is **O(n):** $T(n) = 2T(n/2) + O(n) \Rightarrow O(n \lg n)$
- if merge phase is **O(n lg n)**: $T(n) = 2T(n/2) + O(n \lg n) \Rightarrow O(n \lg^2 n)$

Closest pair, divide-and-conquer



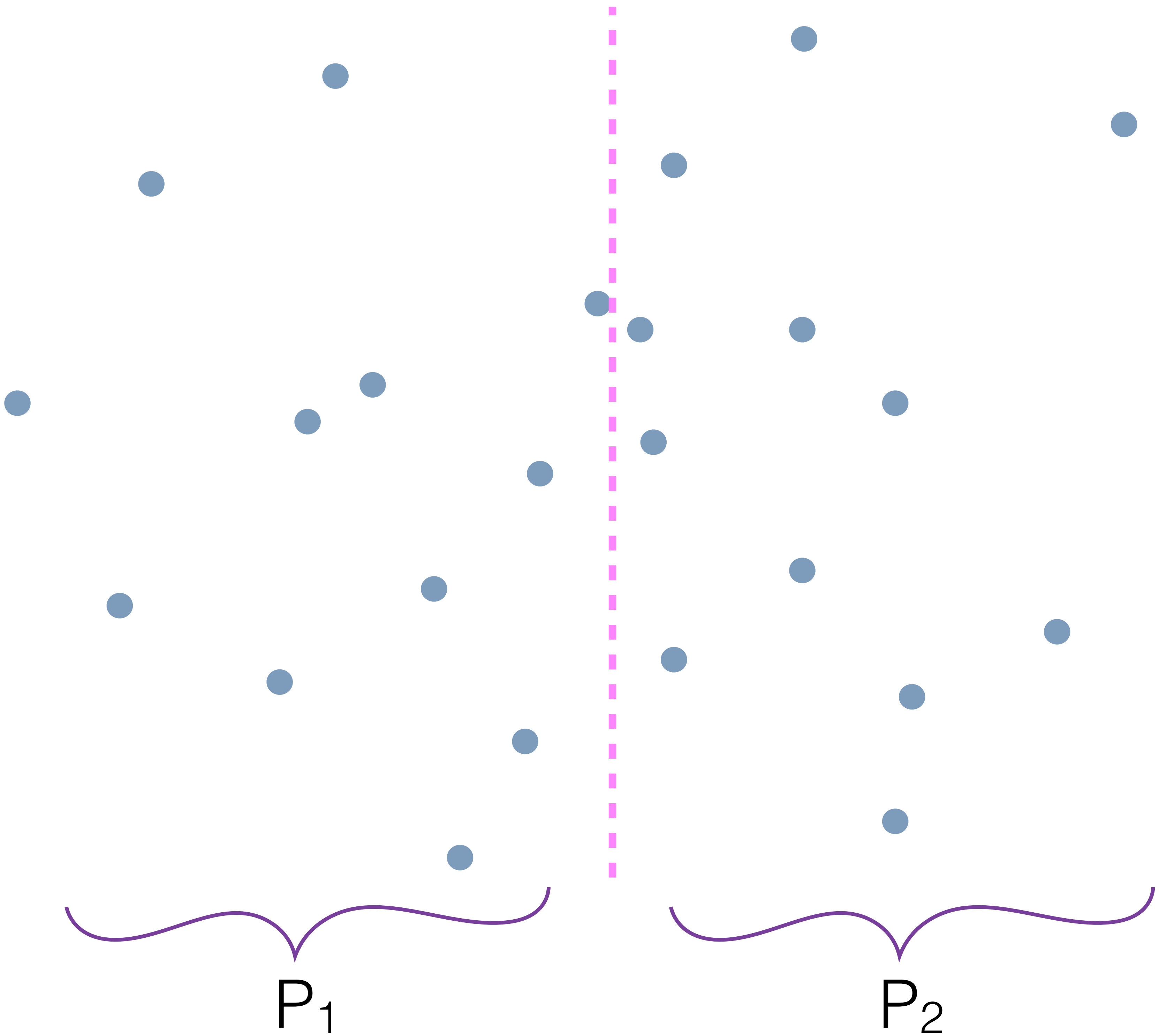
Closest pair, divide-and-conquer

- find vertical line that splits P in half



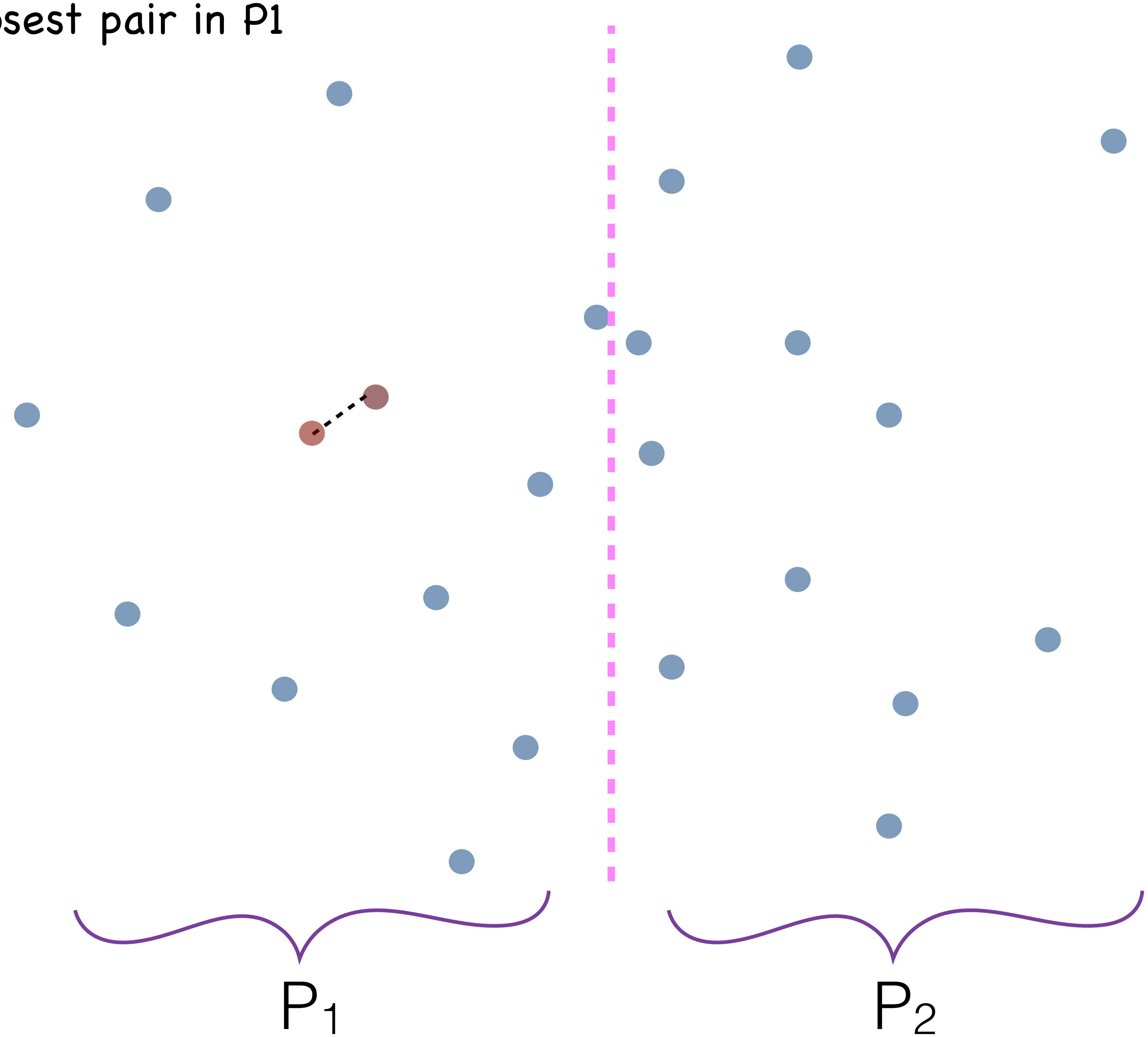
Closest pair, divide-and-conquer

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



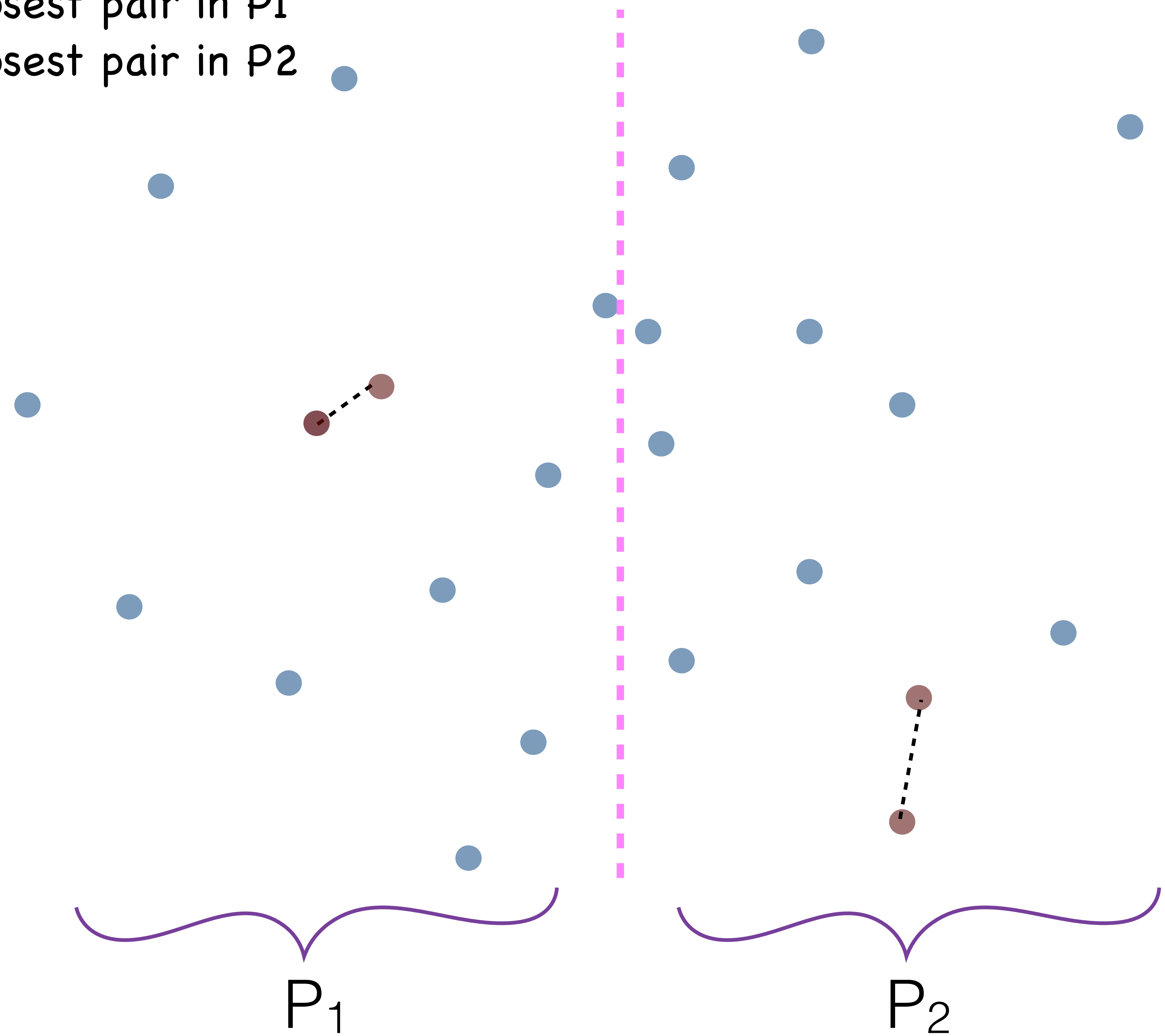
Closest pair, divide-and-conquer

- 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



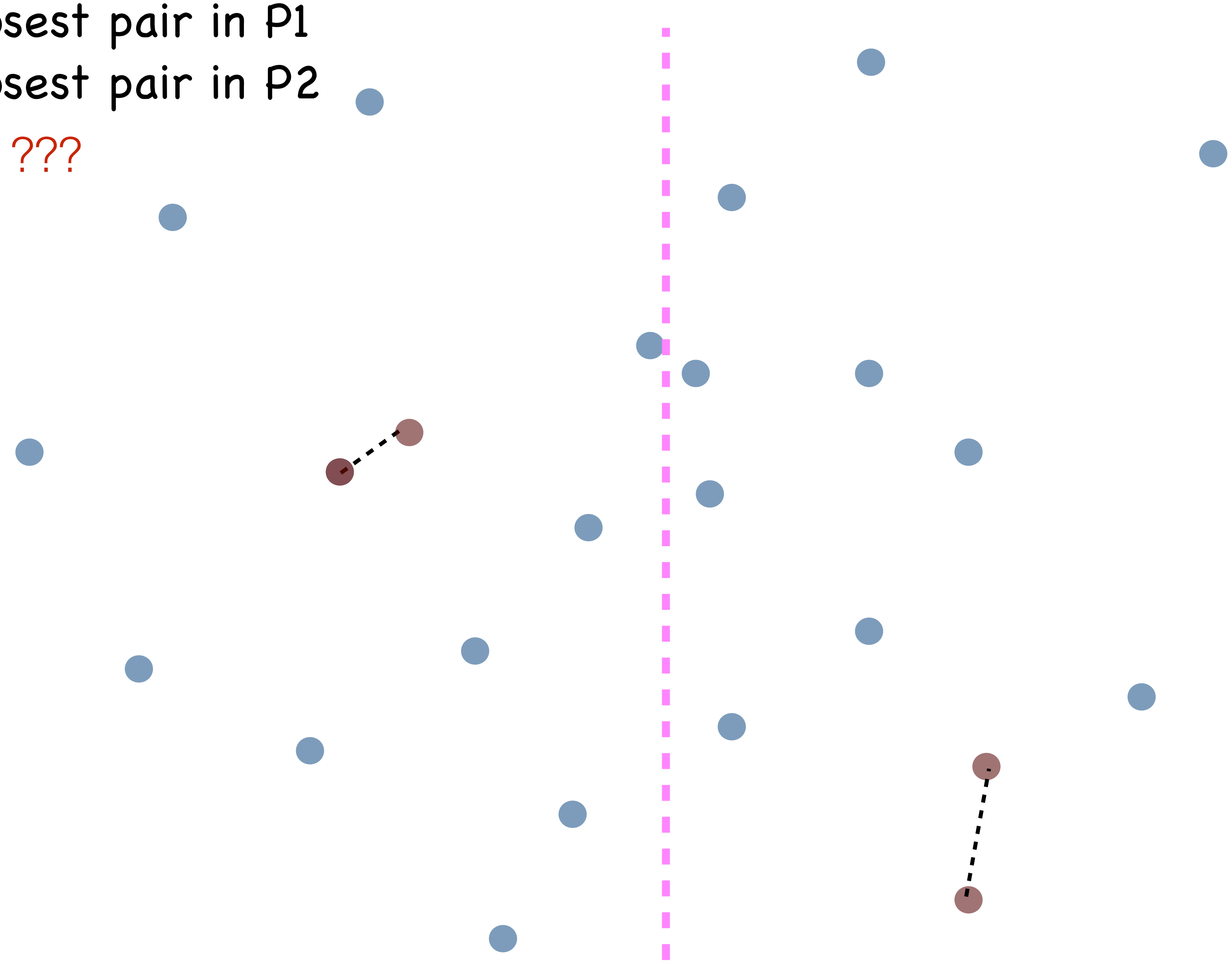
Closest pair, divide-and-conquer

- 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



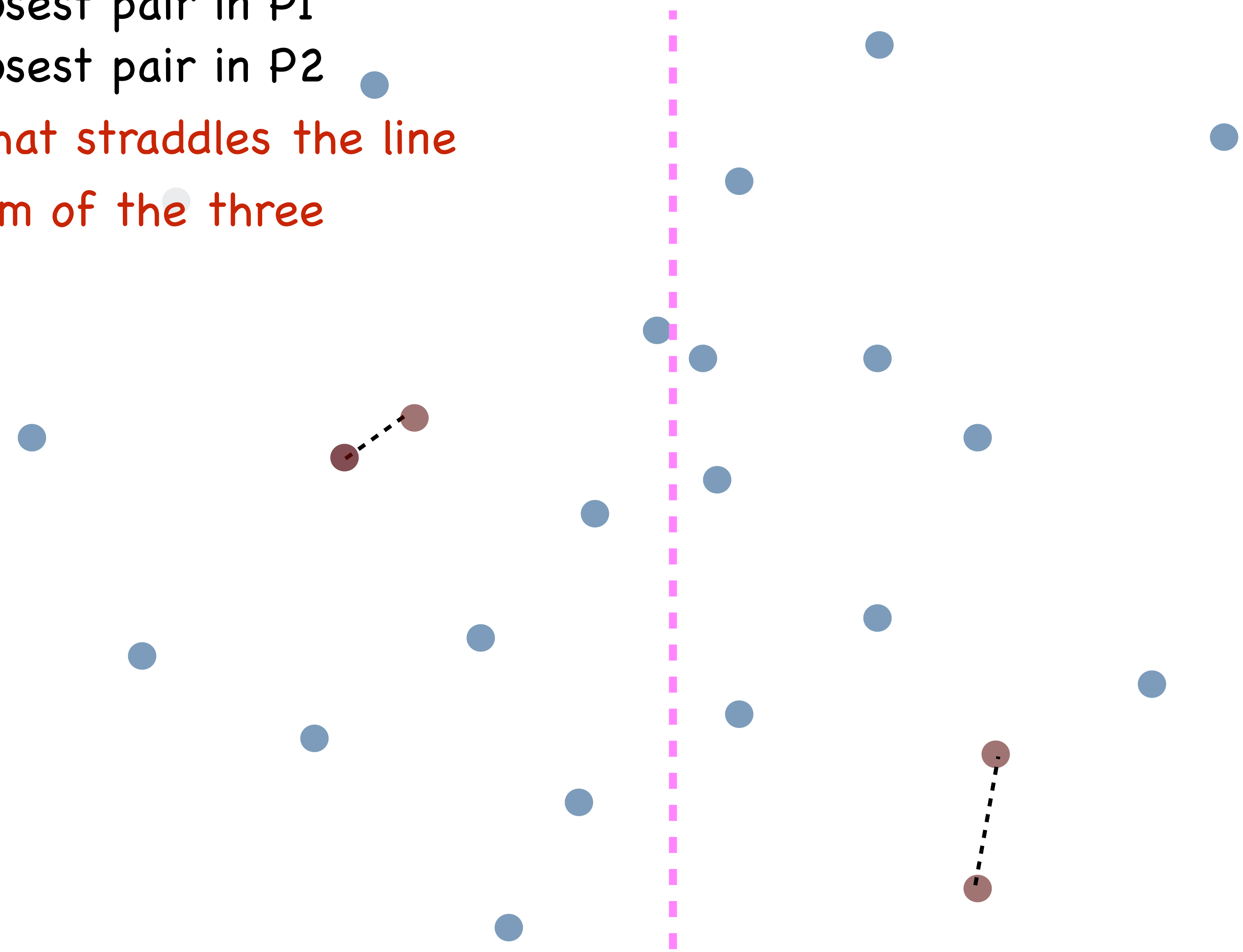
Closest pair, divide-and-conquer

- 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{d_1, d_2, d(p,q)}

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

Is this correct?

Closest pair, divide-and-conquer

Correct, because the closest pair in P must be one of:

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

Closest pair, divide-and-conquer

FindClosestPair(P)

//basecase

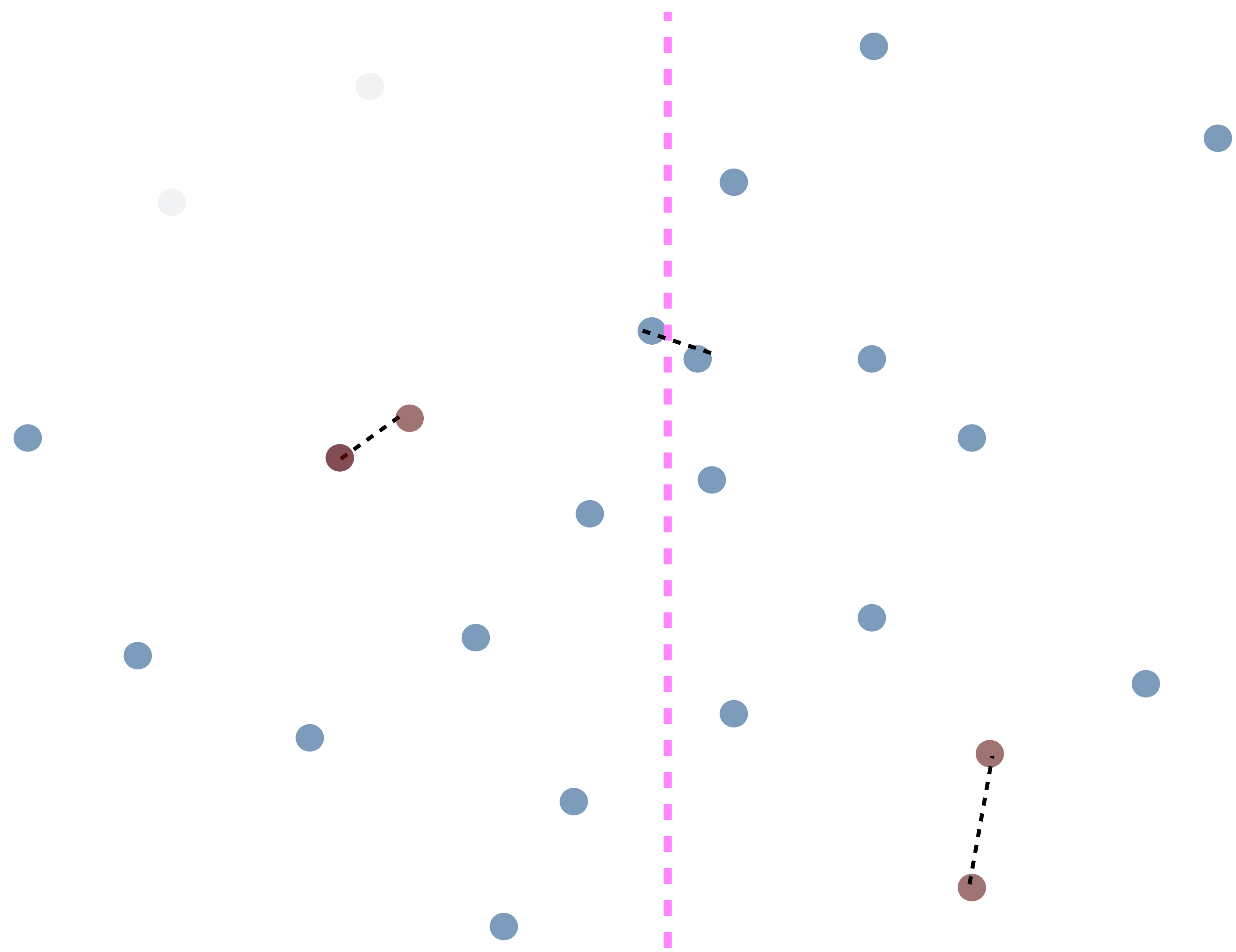
- if P has 1 point, return infinity
- if P has 2 points, return their distance
- else
  - find vertical line that splits P in half
  - let P1, P2 = set of points to the left/right of line
  - d_1 = FindClosestPair(P1)
  - d_2 = FindClosestPair(P2)

  //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{d_1, d_2, d(p,q)}

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

Running time?

Closest pair, divide-and-conquer



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{d_1, d_2, d(p,q)}

  //return smallest of the three

  - return min {d_1, d_2, mindist}

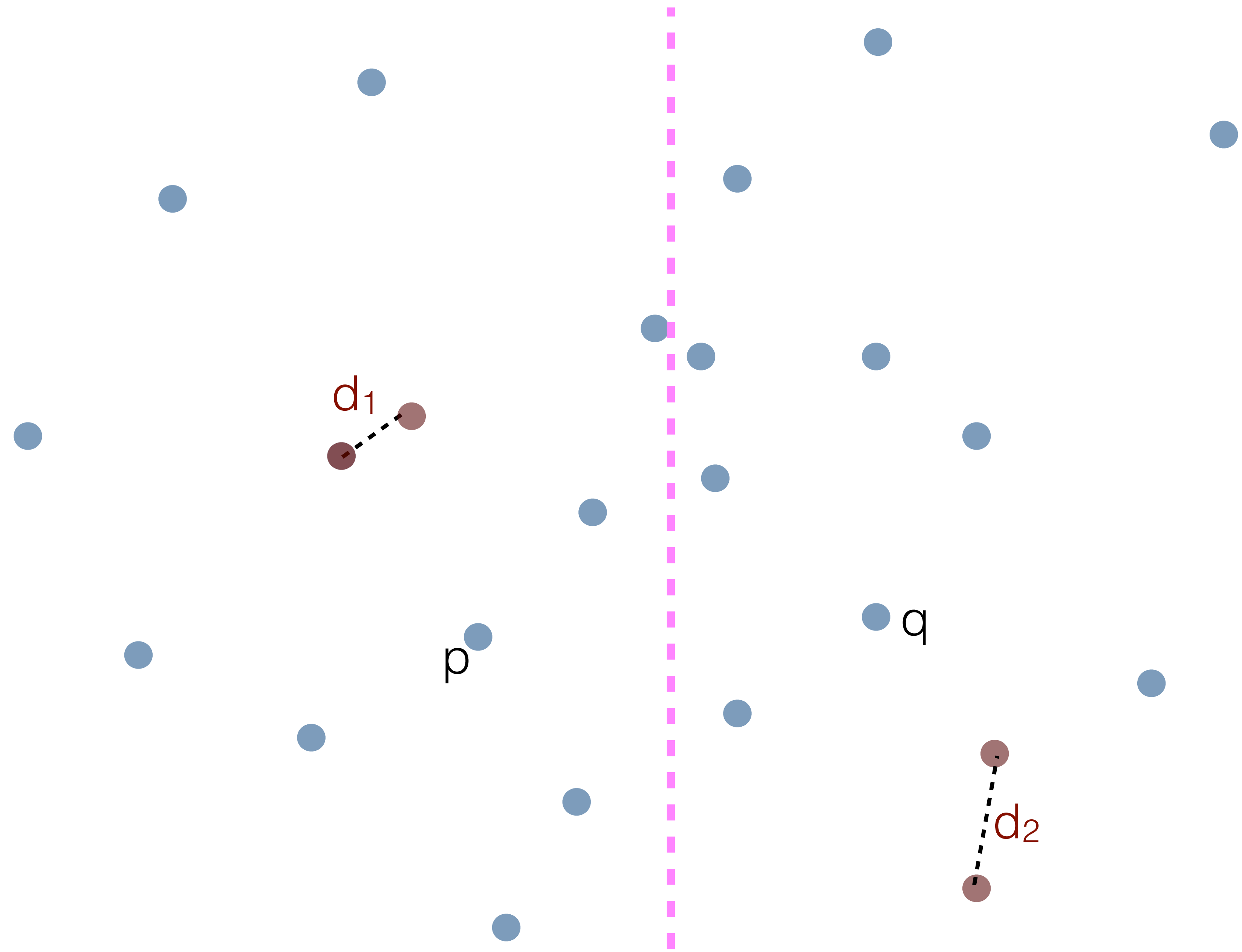
Running time?

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

solves to $O(n^2)$

Refining the merge

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

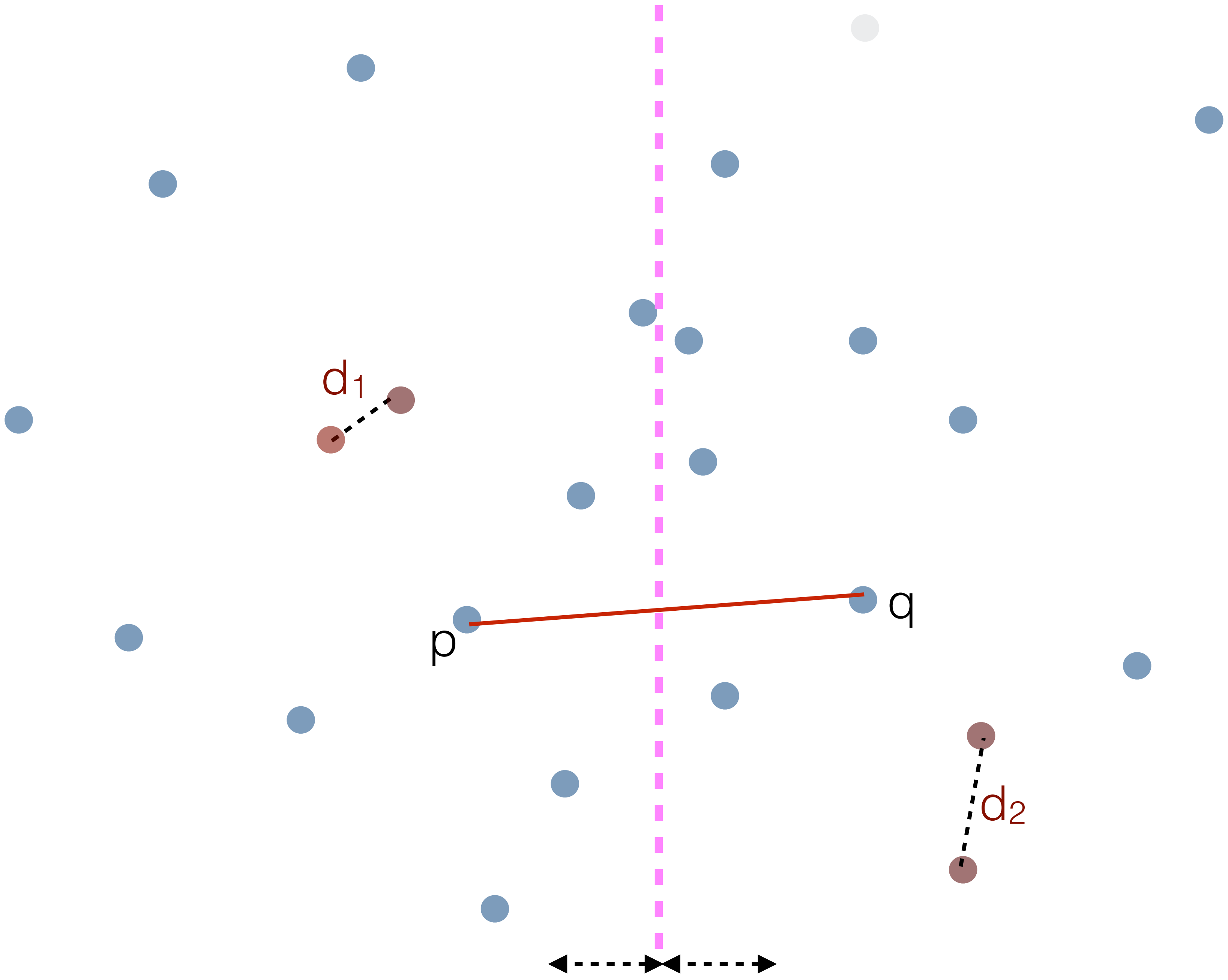


Can (p,q) be the closest pair?

Why not? Where do p,q need to lie in order to be the closest pair?

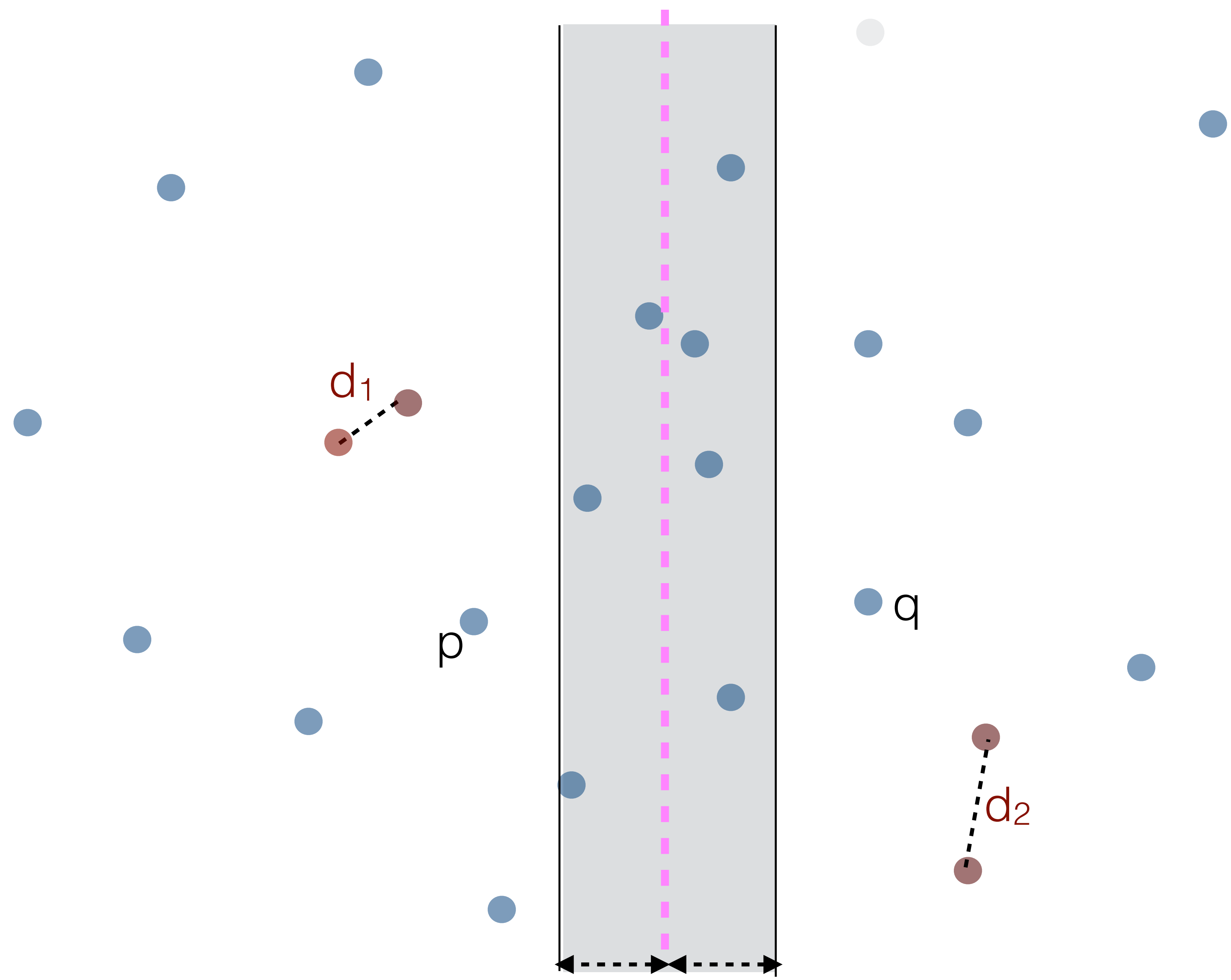
Notation: $d = \min\{d_1, d_2\}$

Claim: In order for dist(p,q) to be smaller than d, it must be that both the horizontal and vertical distance between p and q must be smaller than d.

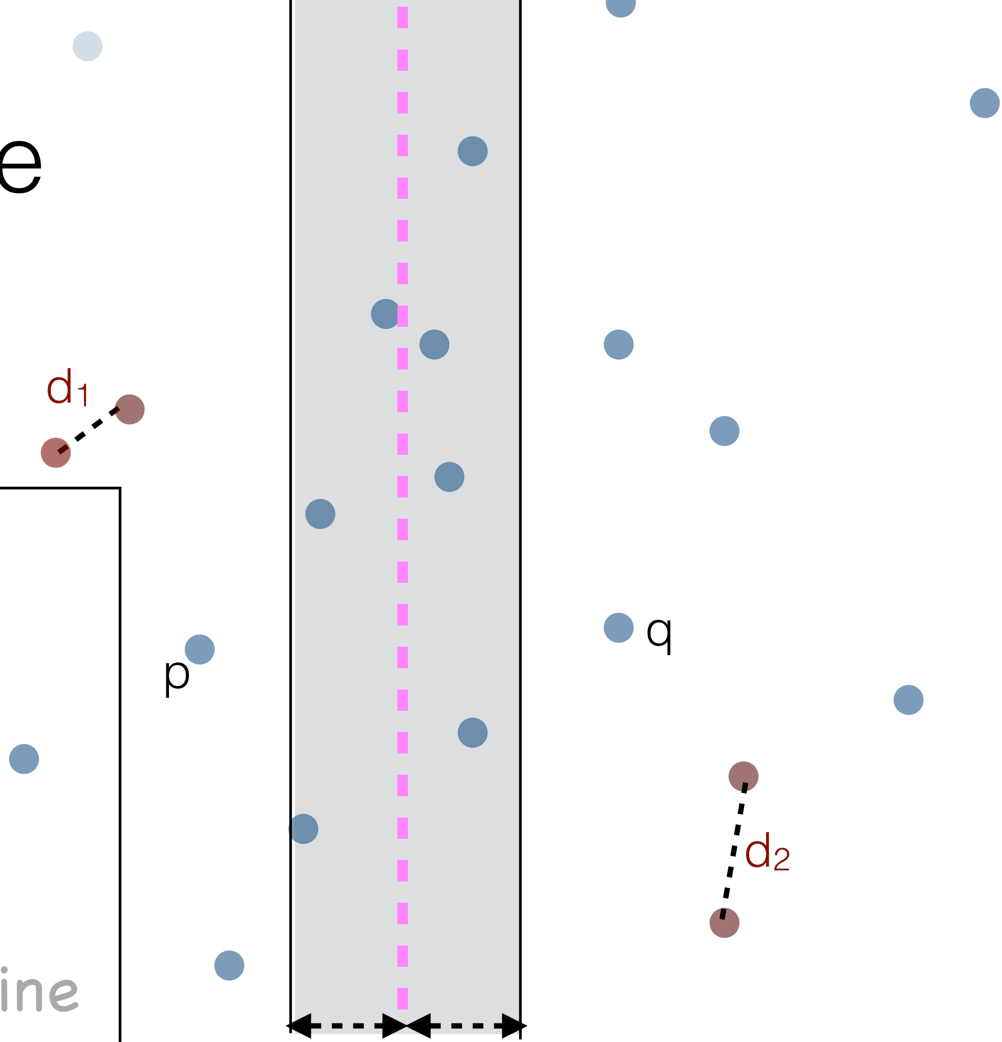


Notation: $d = \min\{d_1, d_2\}$

Claim: In order to be candidates for closest pair, points p, q must lie in the d-by-d strip centered at the median.



Refining the merge

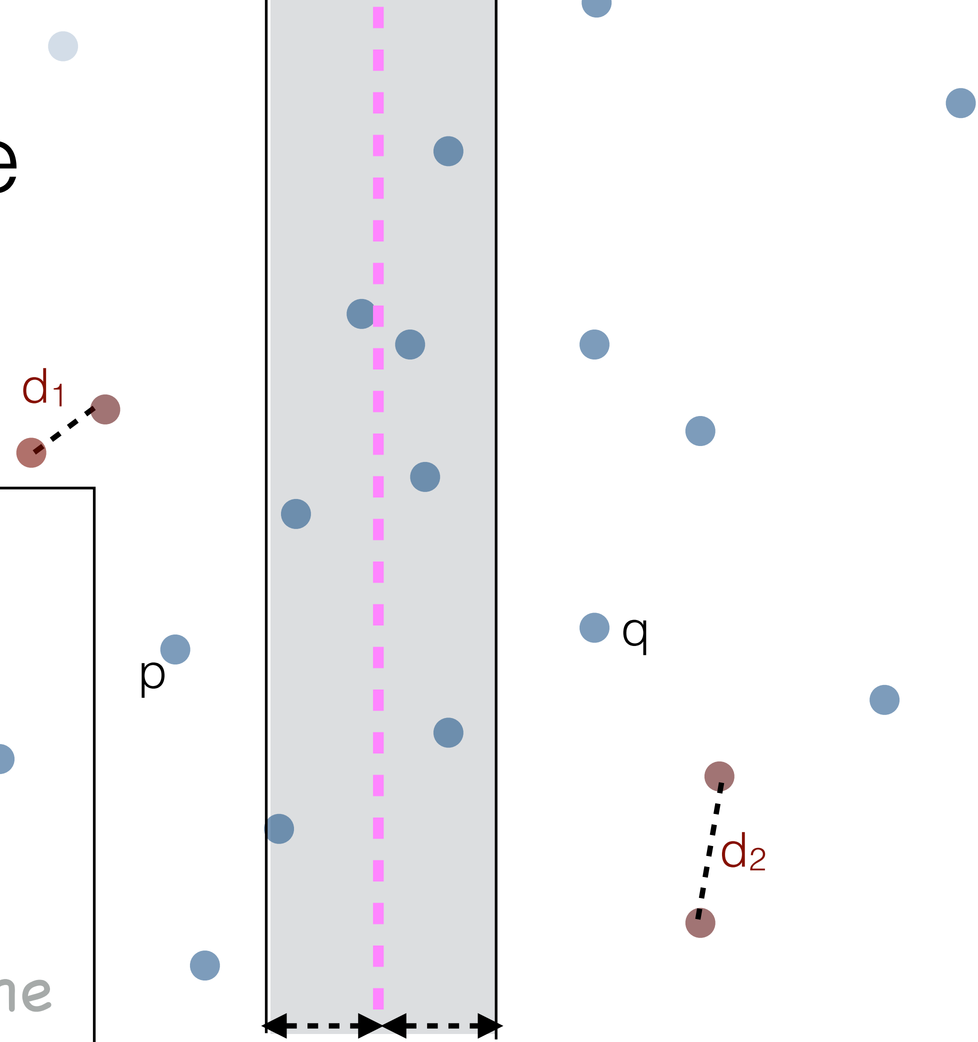


FindClosestPair(P)

- 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
  - d1=FindClosestPair(P1)
  - d2=FindClosetPair(P2)
  - traverse P_1 and select all points P_1' in the strip
  - traverse P_2 and select all points P_2' in the strip
  - for each p in P_1'
    - for each point q in P_2'
      - compute distance d(p,q)
      - mindist = min{d_1, d_2, d(p,q)}
- return min {d_1, d_2, mindist}

Running time?

Refining the merge



FindClosestPair(P)

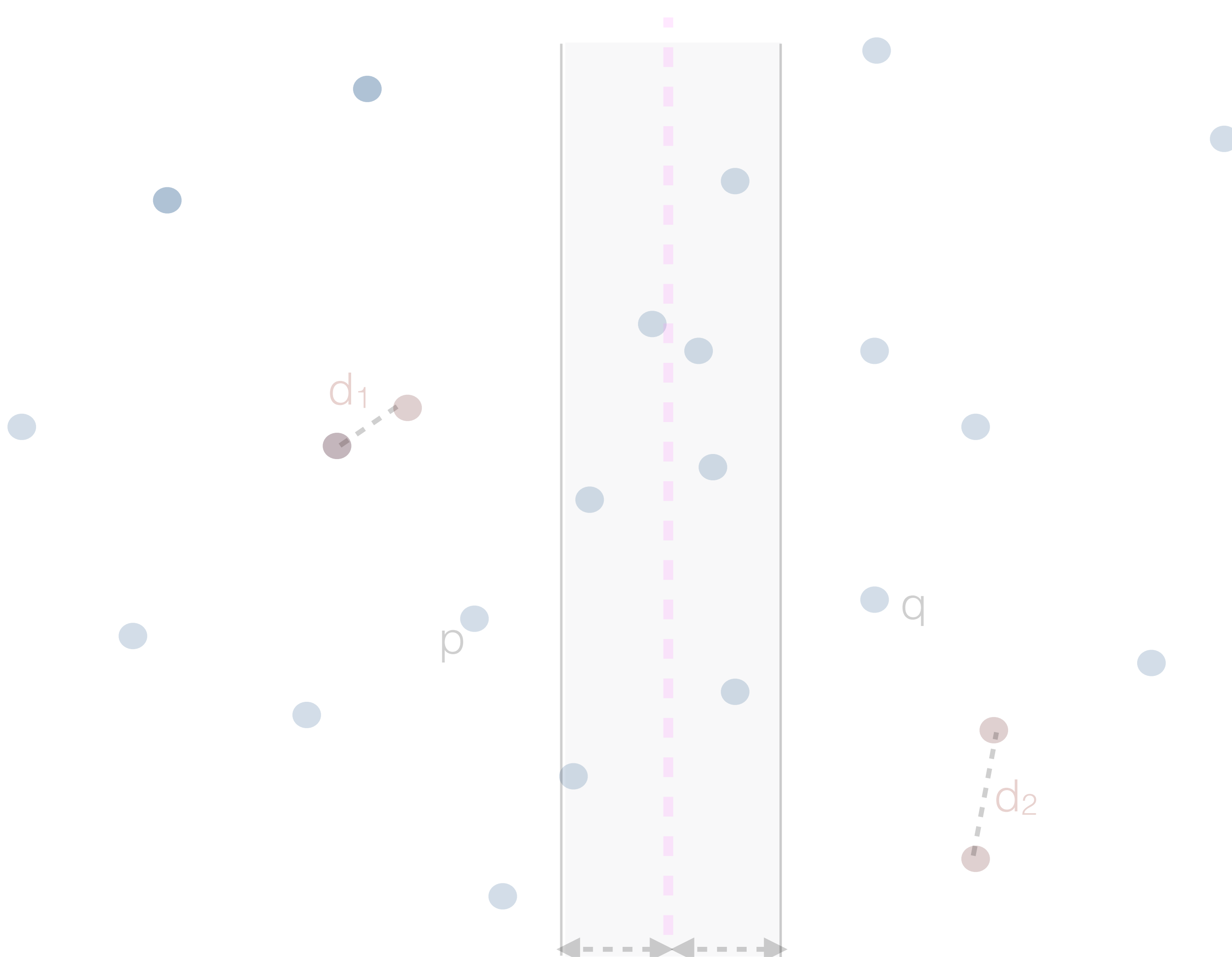
- if P has 1 point, return infinity
- if P has 2 points, return their distance
- else
  - find vertical line that splits P in half
  - let P1, P2 = set of points to the left/right of line
  - d1=FindClosestPair(P1)
  - d2=FindClosetPair(P2)
  - traverse P_1 and select all points P_1' in the strip
  - traverse P_2 and select all points P_2' in the strip
  - for each p in P_1'
    - for each point q in P_2'
      - compute distance d(p,q)
      - mindist = min{d_1, d_2, d(p,q)}
- return min {d_1, d_2, mindist}

Running time?

It's possible that all n/2 points on either side lie inside the strip

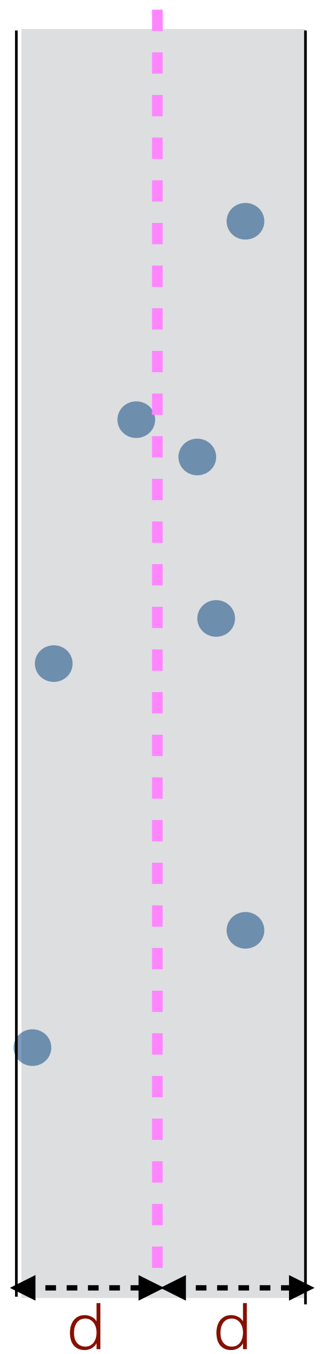
Refining the merge

- Show an example where the strip may contain Omega(n) points.
- What does this imply for the running time?



Refining the merge

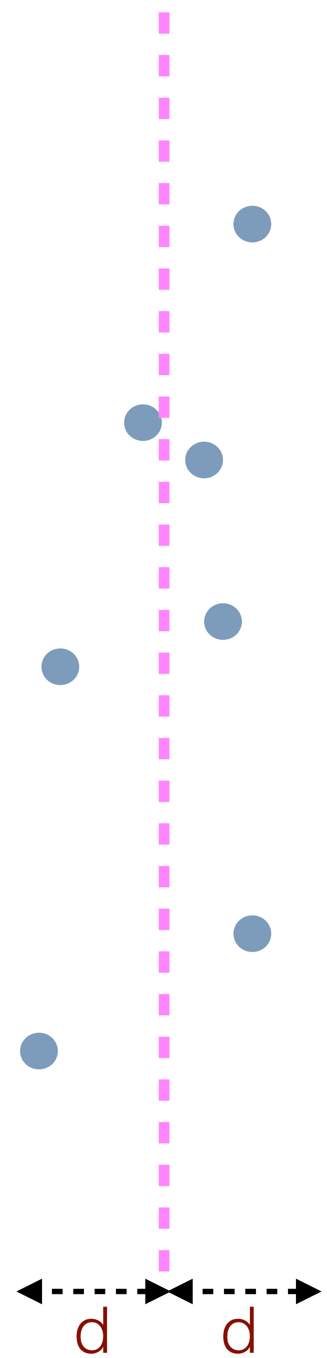
- Filtering the points in the strip is not enough
- But, we can show that the points in the strip have a special structure which will enable us to merge faster



Refining the merge

Points on both sides are “sparse”

Any pair of points in P_1 must be at least d away



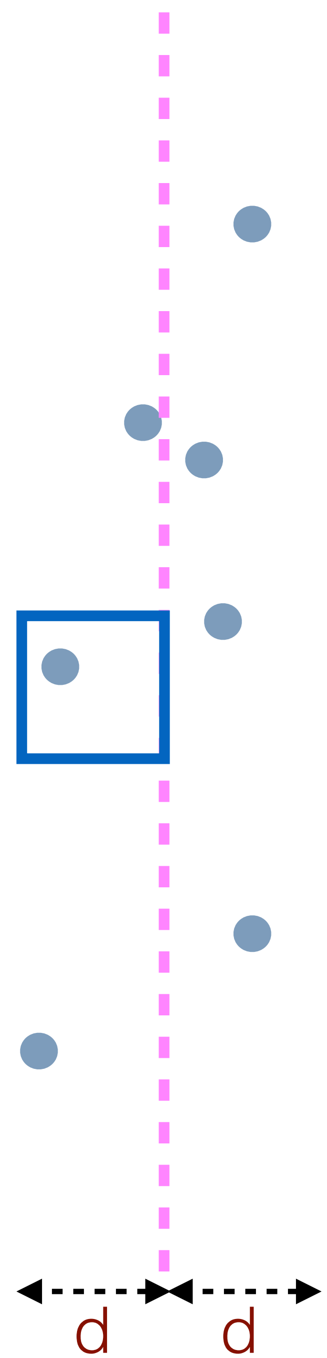
Any pair of points in P_2 must be at least d away

Refining the merge

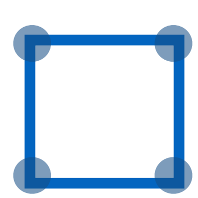
Points on both sides are "sparse"

Any pair of points in P_1 must be at least d away

Any pair of points in P_2 must be at least d away

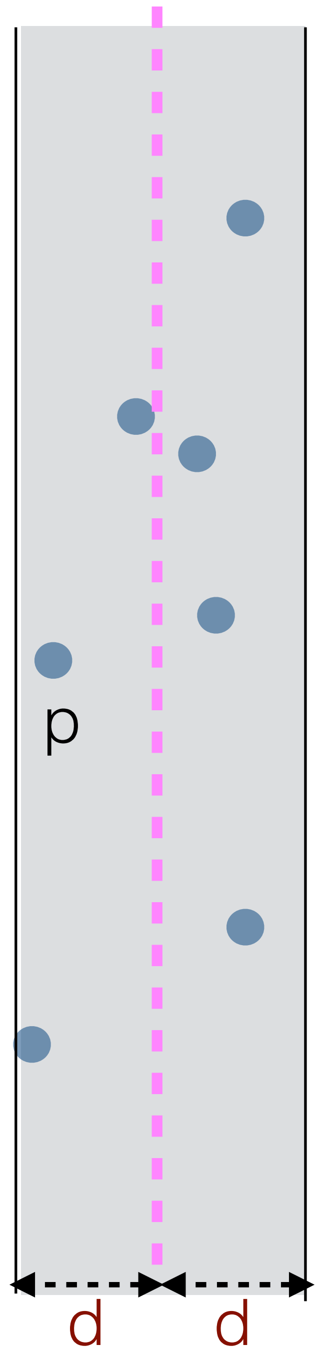


Any square with side d contains at most 4 points of P_1



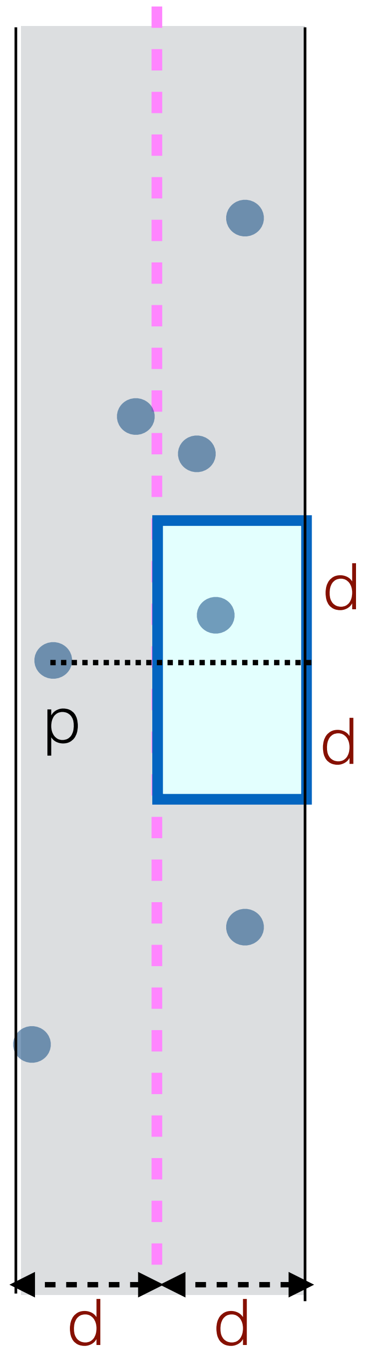
Refining the merge

- Furthermore, consider a point p in P_1'
- We don't need to compute the distances from p to all points in P_2'



Refining the merge

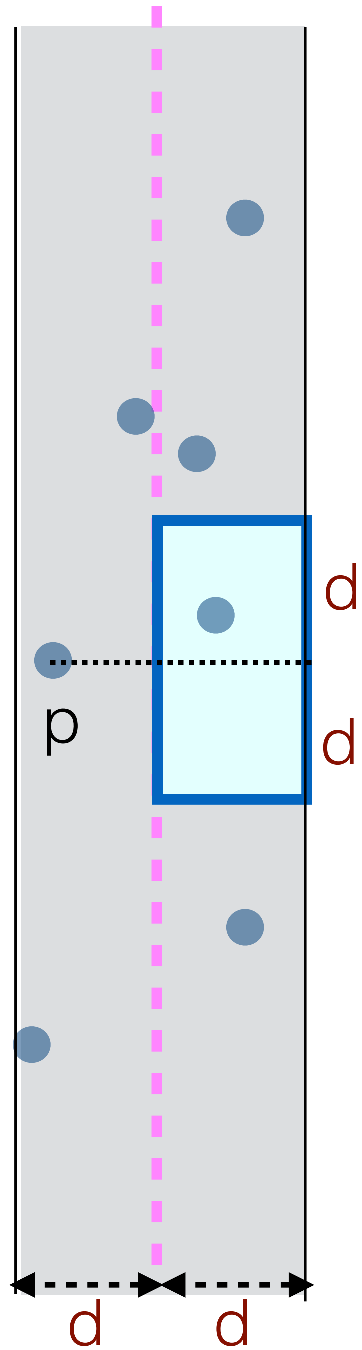
- Furthermore, consider a point p in P_1'
- We don't need to compute the distances from p to all points in P_2'



- CLAIM: All points of P_2' within distance d of p are vertically above or below p by at most d ==> they must lie in a rectangle d x 2d

Refining the merge

- Furthermore, consider a point p in P_1'
- We don't need to compute the distances from p to all points in P_2'

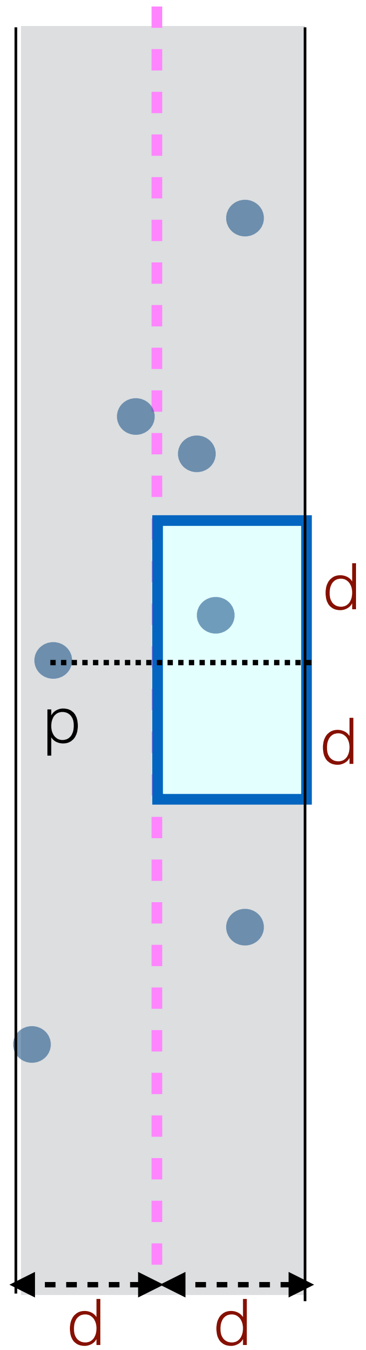


- CLAIM: All points of P_2' within distance d of p are vertically above or below p by at most d ==> they must lie in a rectangle d x 2d

- How many points q of P_2' can there be in a rectangle of size d x 2d? (knowing that any pair of points in P_2' must be at least d away).

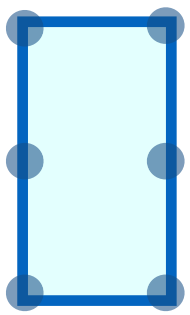
Refining the merge

- Furthermore, consider a point p in P_1'
- We don't need to compute the distances from p to all points in P_2'



- CLAIM: All points of P_2' within distance d of p are vertically above or below p by at most d ==> they must lie in a rectangle d x 2d

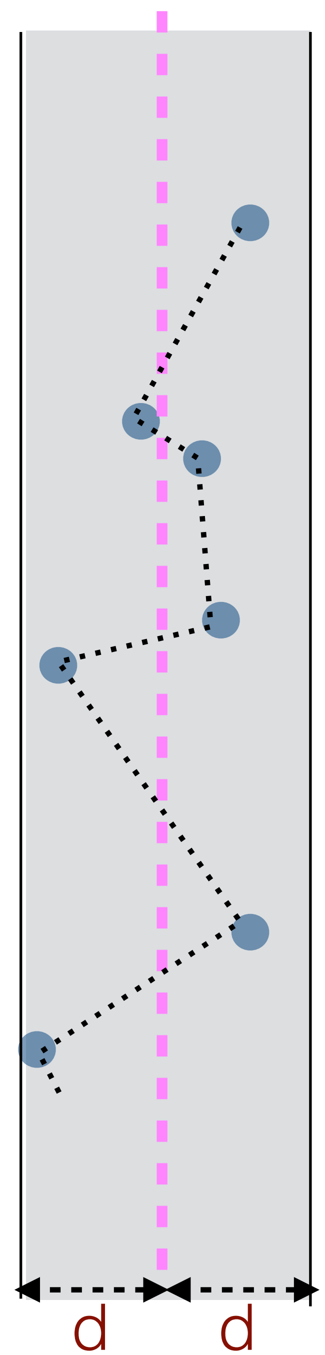
- How many points q of P_2' can there be in a rectangle of size d x 2d? (knowing that any pair of points in P_2' must be at least d away).



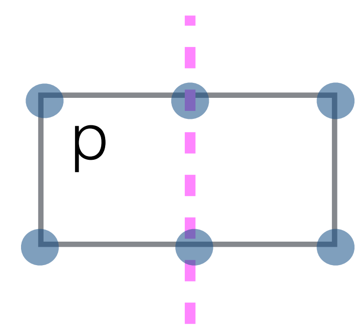
=> So for every p in P_1', we only need to check at most 6 points!

Refining the merge

- An elegant/simple way to do this is by traversing the points in P_1’ and P_2’ in y-order
- A point p needs to check only the points following it, and there can be at most 5 points following p in y-order that are within d from p.



Note: Assume no duplicate points.



Refining the merge

closestPair(P)

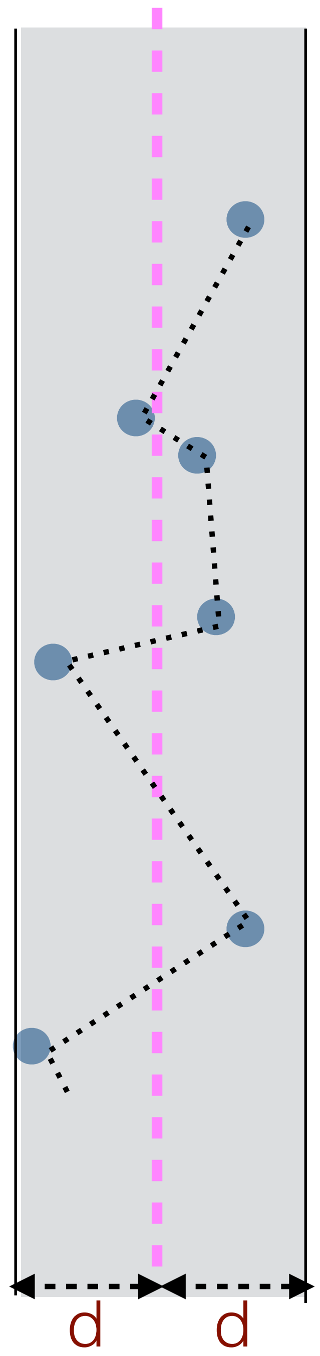
//divide

- 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)

//merge

- let d = min{d_1, d_2}
- Strip= empty
- for all p in P_1: if $x_p > x_l - d$: add p to Strip
- for all p in P_2: if $x_p < x_l + d$: add p to Strip
- sort Strip by y-coord
- initialize mindist=d
- for each p in Strip in sorted order
    - compute its distance to the 5 points that come after it in sorted order
    - if any of these is smaller than mindist, update mindist

- return mindist



Refining the merge

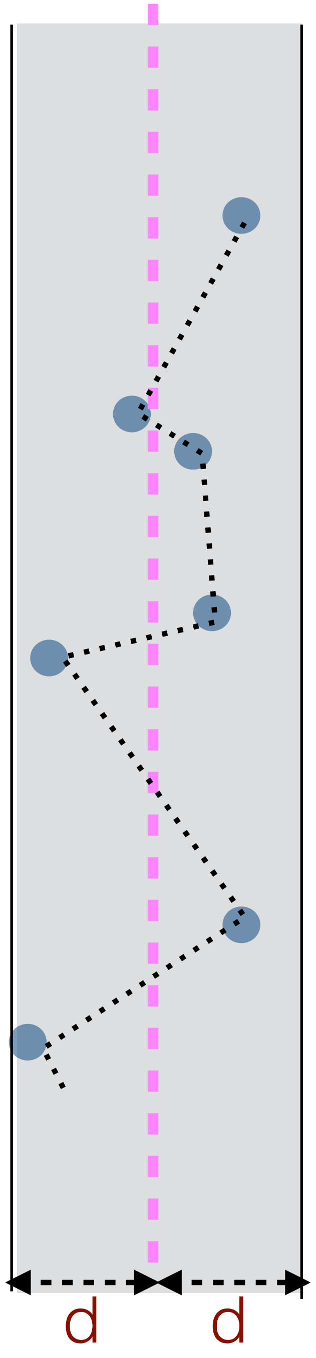
closestPair(P)

//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)

//merge

- let d = min{d_1, d_2}
- Strip= empty
- for all p in P_1: if $x_p > x_l - d$: add p to Strip
- for all p in P_2: if $x_p < x_l + d$: add p to Strip
- sort Strip by y-coord
- initialize mindist=d
- for each p in Strip in sorted order
  - compute its distance to the 5 points that come after it in sorted order
  - if any of these is smaller than mindist, update mindist
- return mindist



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

Closest pair, divide-and-conquer

- The sorted list of points in the strip can be obtained from pre-sorting P

closestPair(P_sorted_by_x,   P_sorted_by_y)

- Sort P at the beginning and maintain it through recursion
- Final recurrence is $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \lg n)$