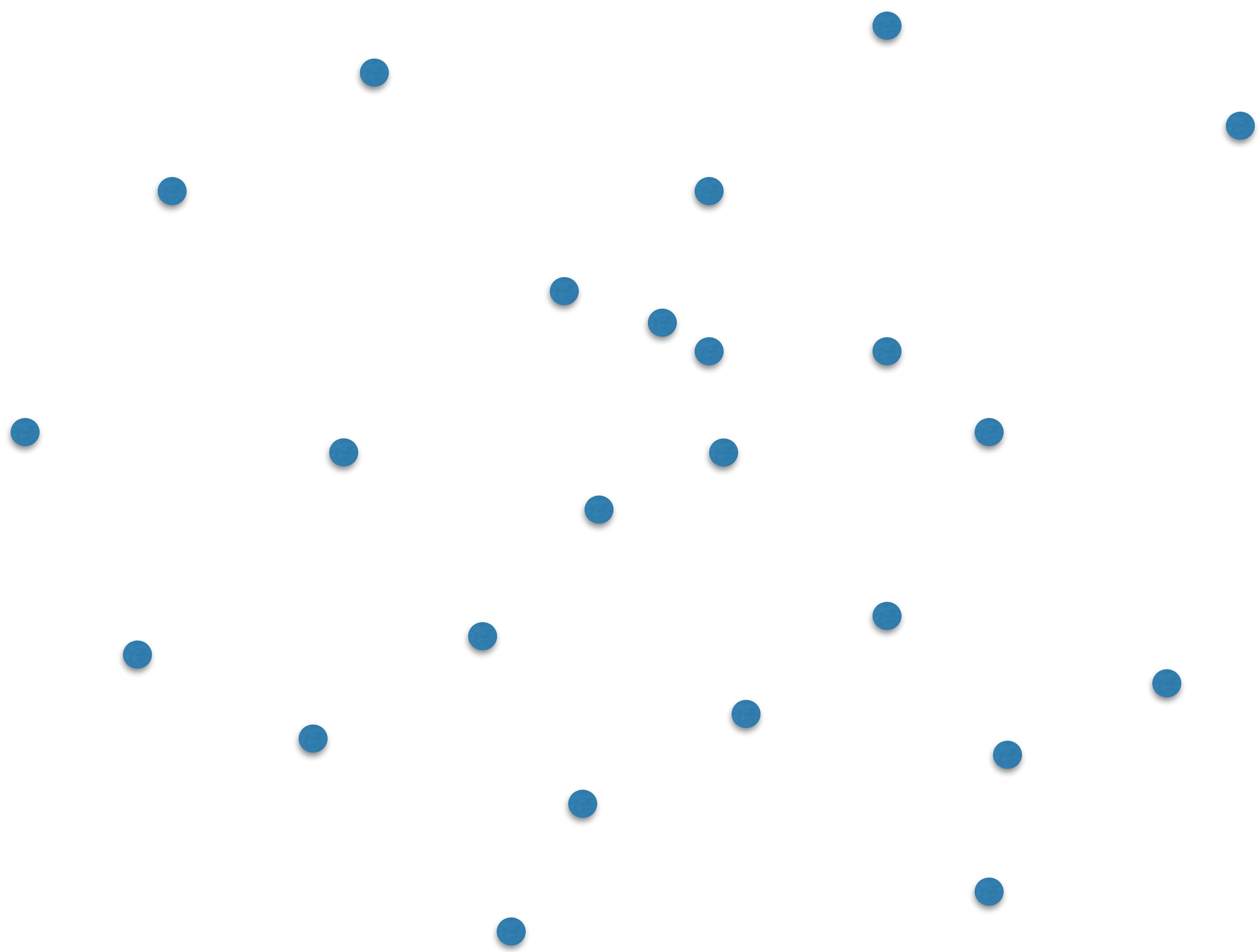
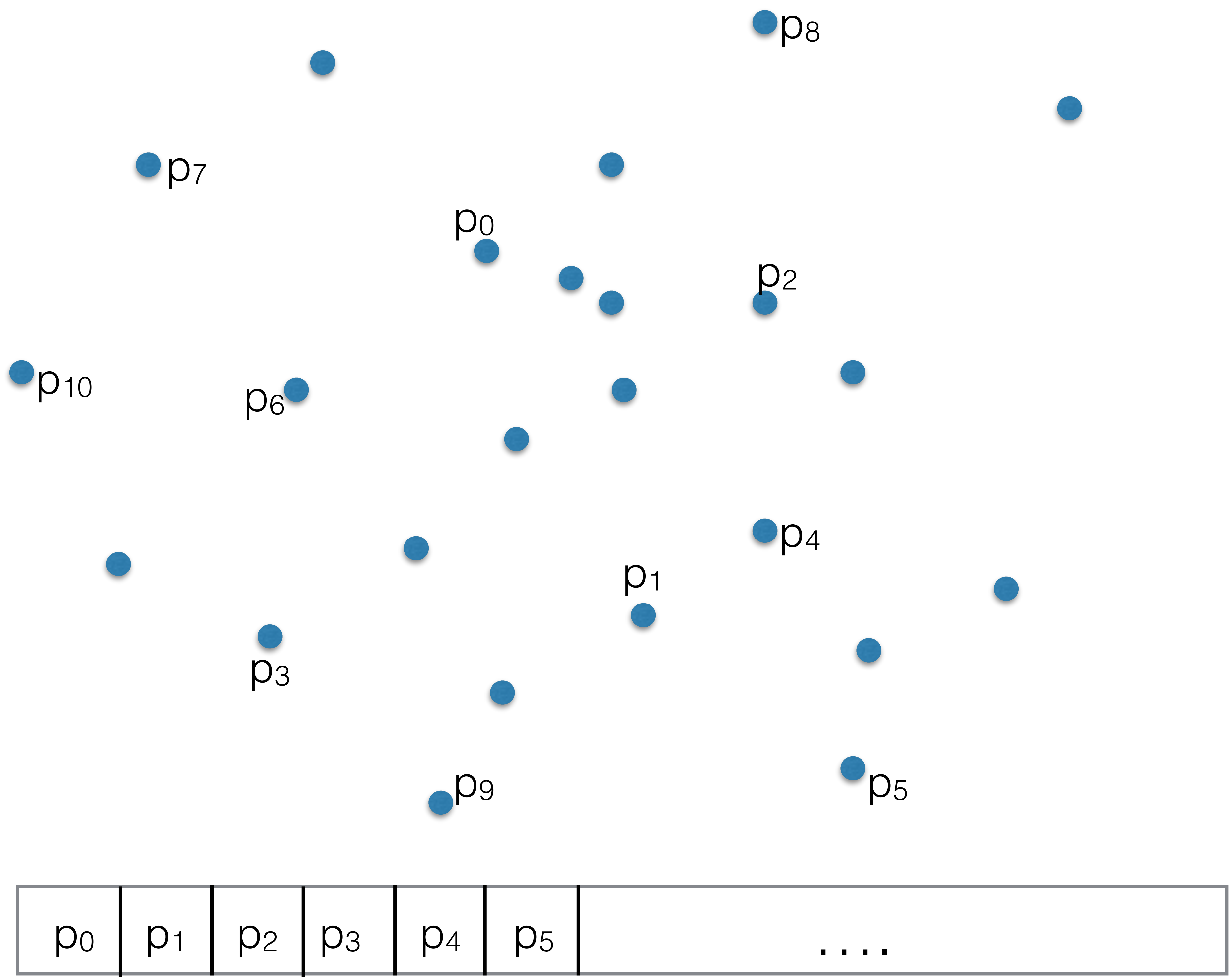
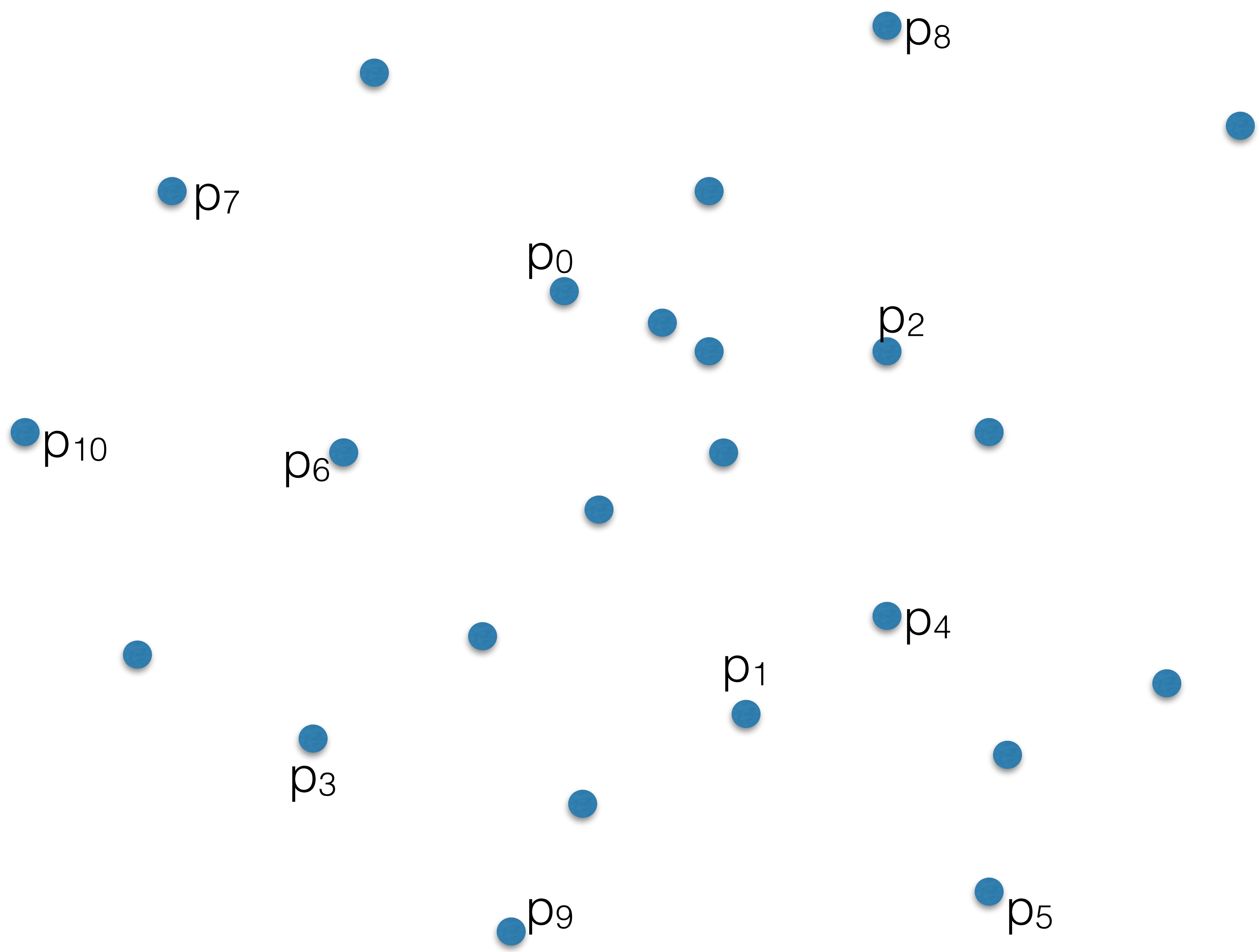
Finding closest pair

Computational Geometry [csci 3250]
Laura Toma
Bowdoin College



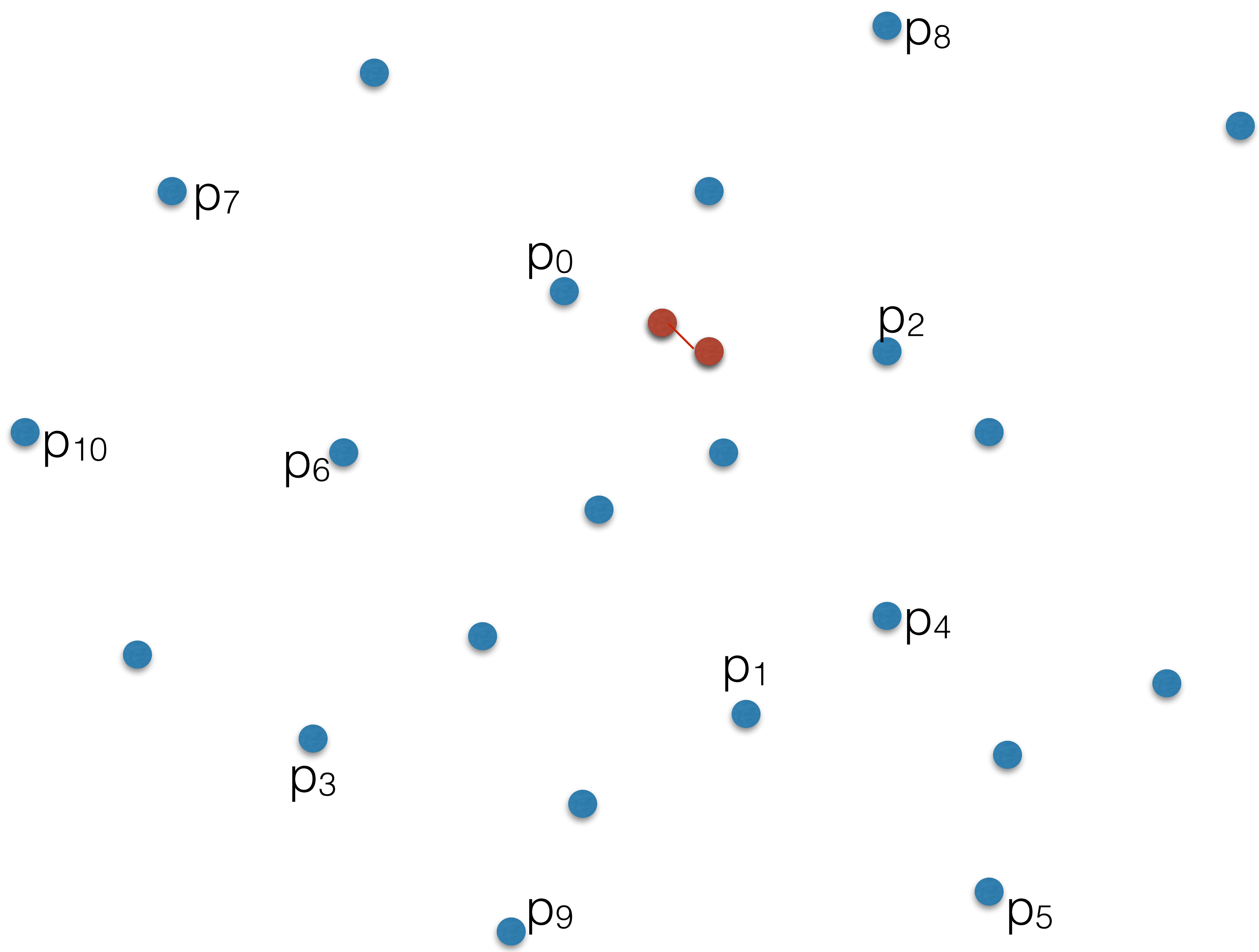

p8
p7
p0
p2
p10
p6
p4
p1
p3
p9
p5
p0 p1 p2 p3 p4 p5 ....

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



p_0	p_1	p_2	p_3	p_4	p_5	....

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



p_0	p_1	p_2	p_3	p_4	p_5	....

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

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_first_half = **mergesort**(array A1)
  - sorted_second_half = **mergesort**(array A2)

  //merge

  - result = merge_sorted_arrays(sorted_first_half, sorted_second_half)
  - 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_first_half = **mergesort**(array A1)
  - sorted_second_half = **mergesort**(array A2)

  //merge

  - result = merge_sorted_arrays(sorted_first_half, sorted_second_half)
  - return result

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

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

In general

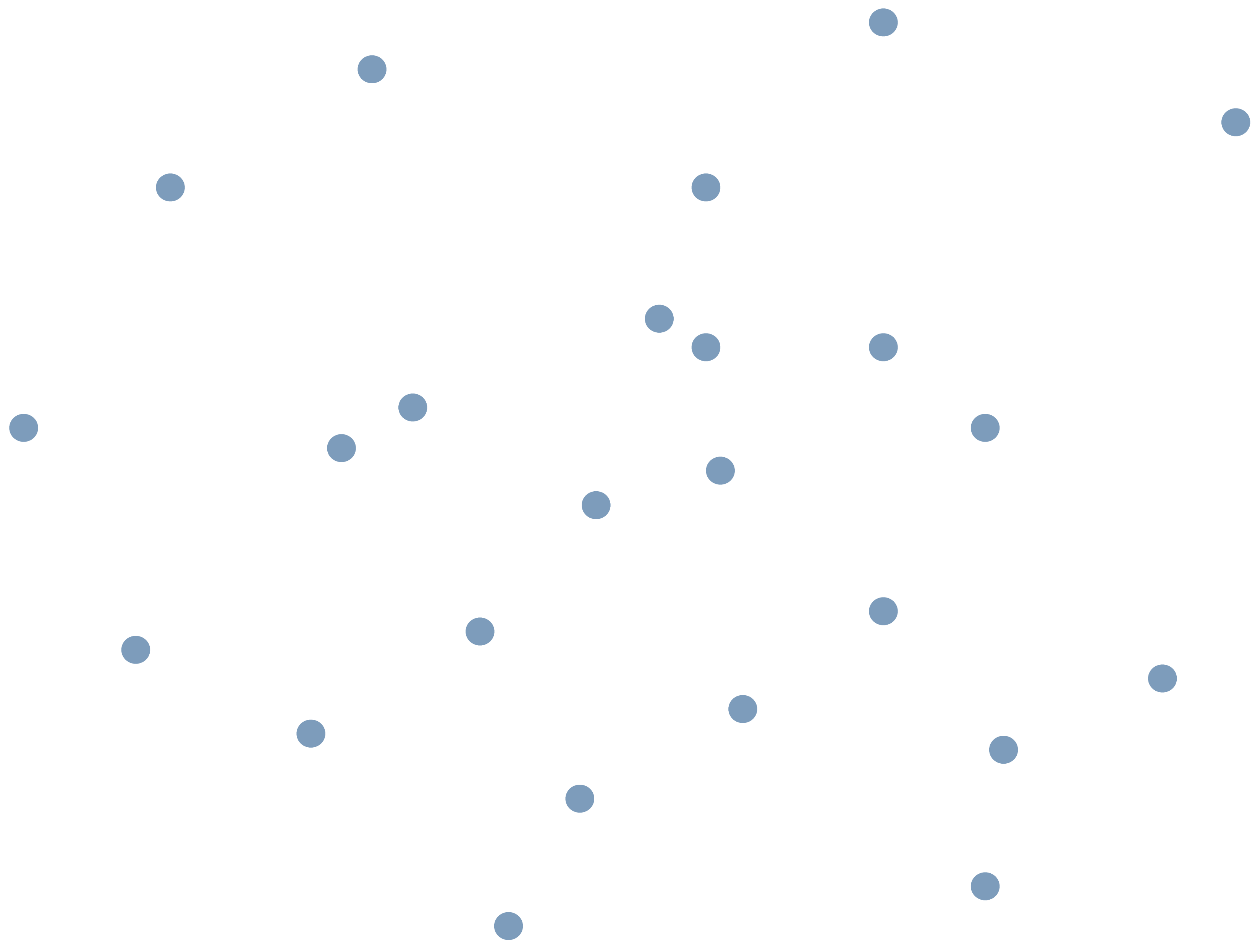
```
DC(input P)
  if P is small, solve and return
  else
    //divide
    divide input P into two halves, P1 and P2
    //recurse
    result1 = DC(P1)
    result2 = DC(P2)
    //merge
    do_something_to_figure_out_result_for_P

    return result
```

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

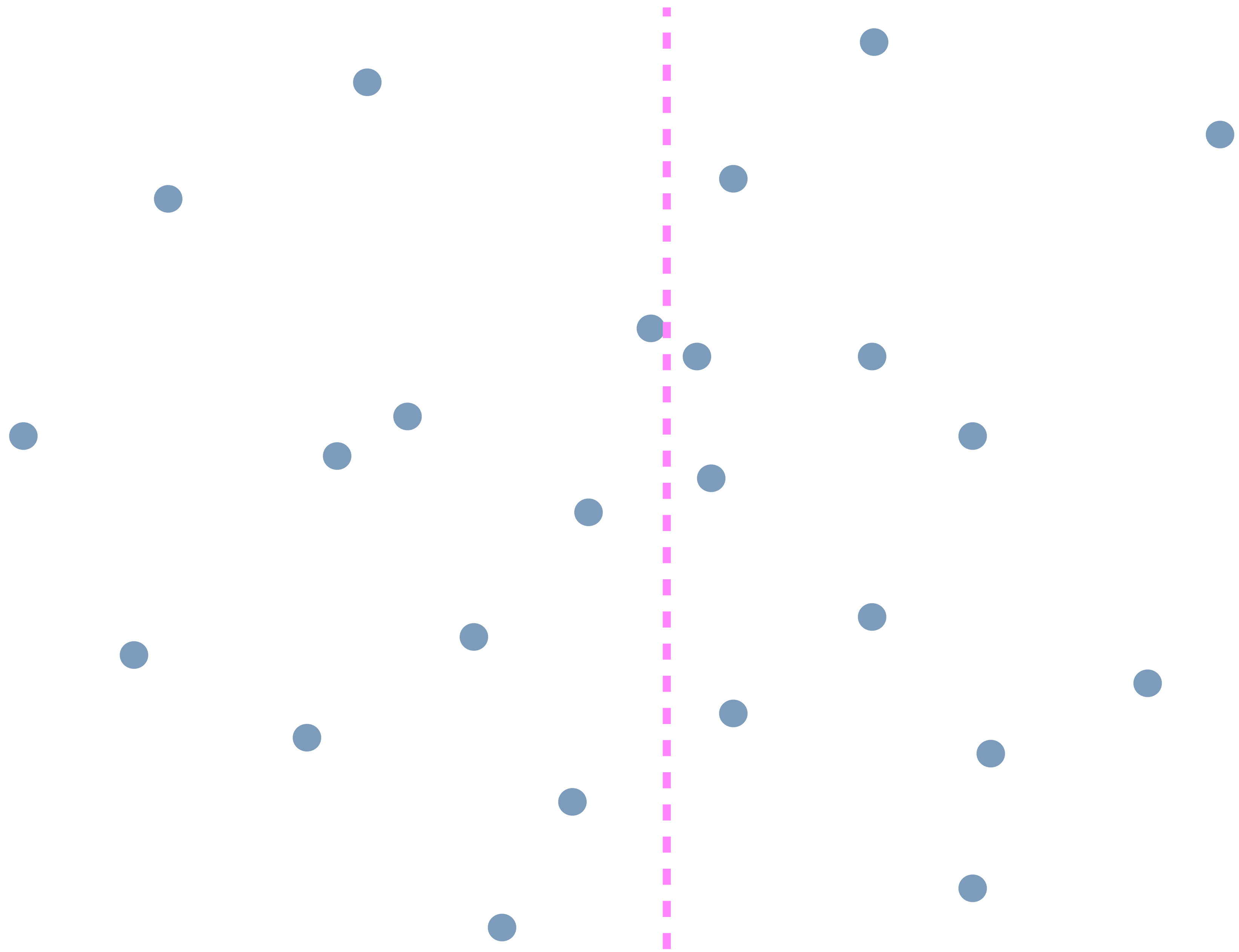
- if merge phase is **O(n)**: $T(n) = 2T(n/2) + O(n) \Rightarrow O(n \lg n)$
- if merge phase is **O(n lg n)**: $T(n) = 2T(n/2) + O(n \lg n) \Rightarrow O(n \lg^2 n)$

Divide-and-conquer for closest pair



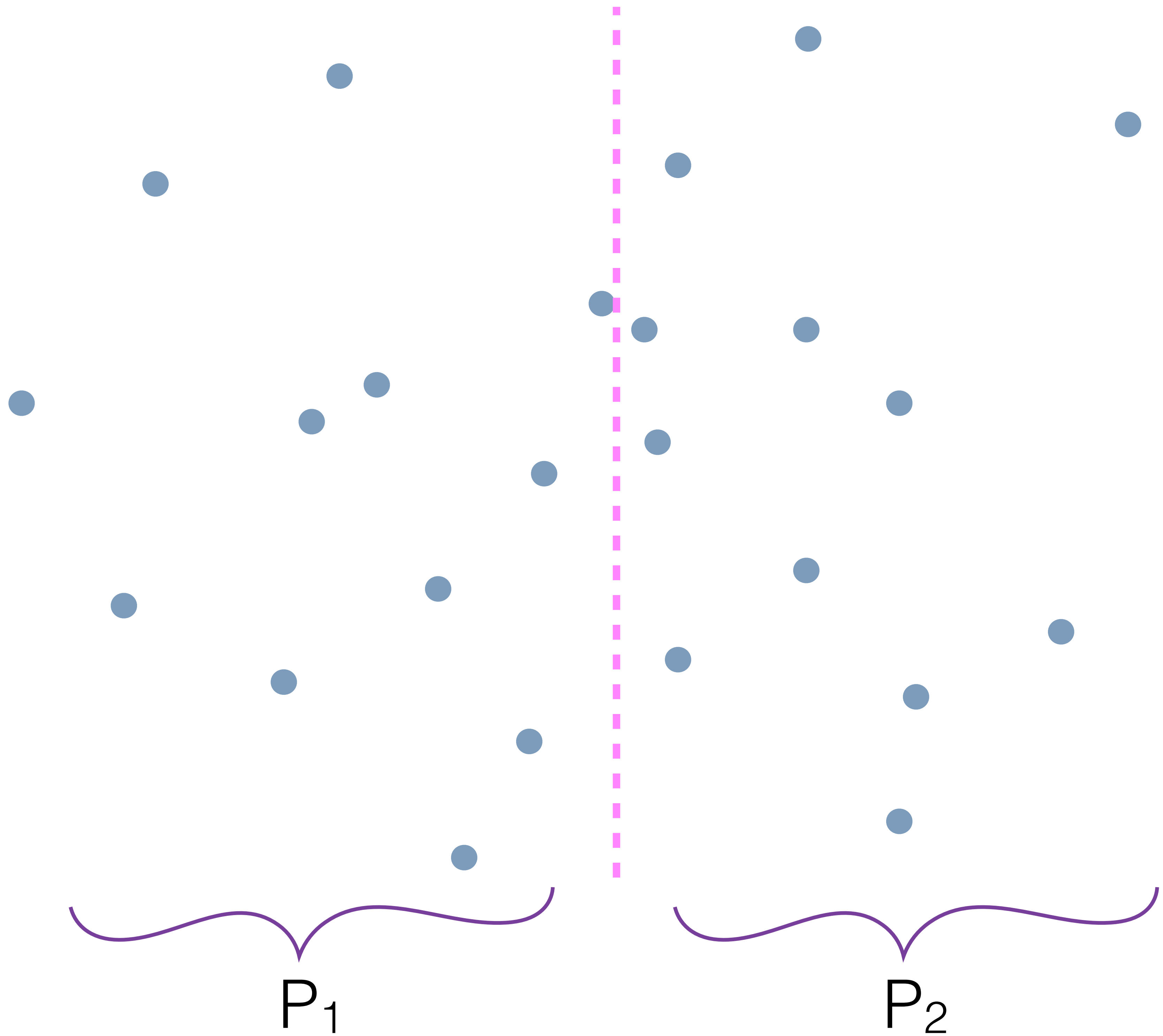
Divide-and-conquer for closest pair

- find vertical line that splits P in half



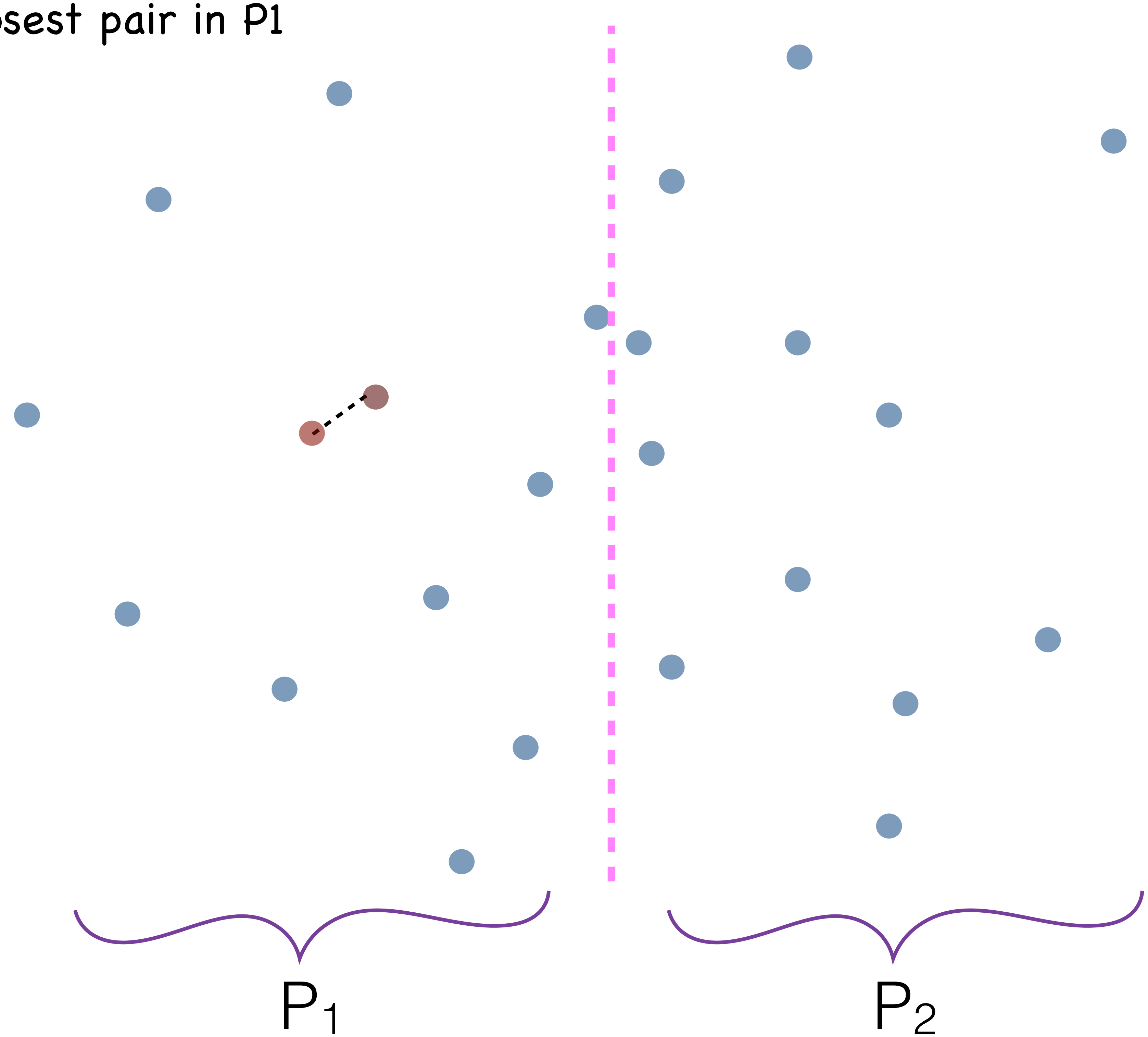
Divide-and-conquer for closest pair

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



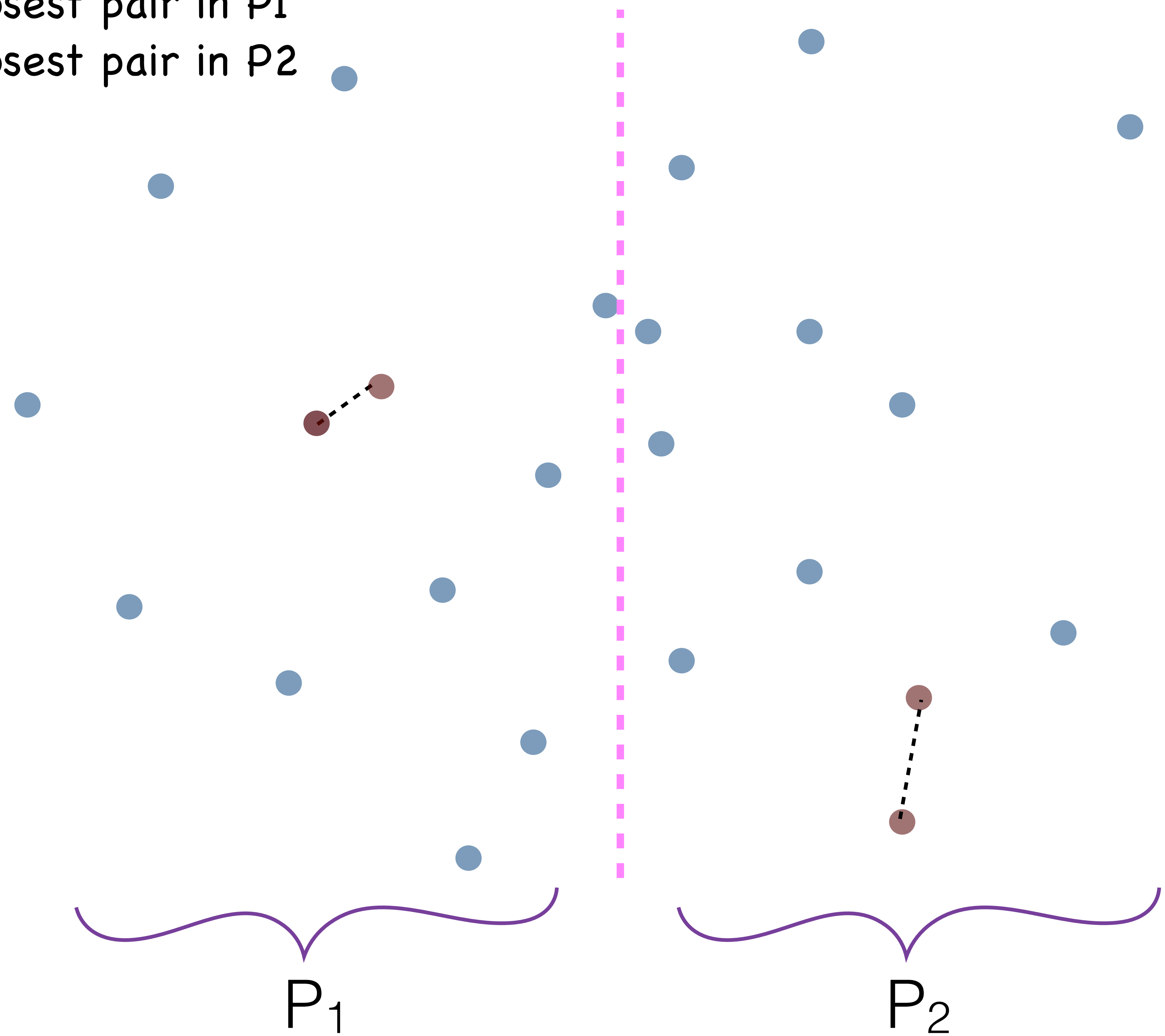
Divide-and-conquer for closest pair

- 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



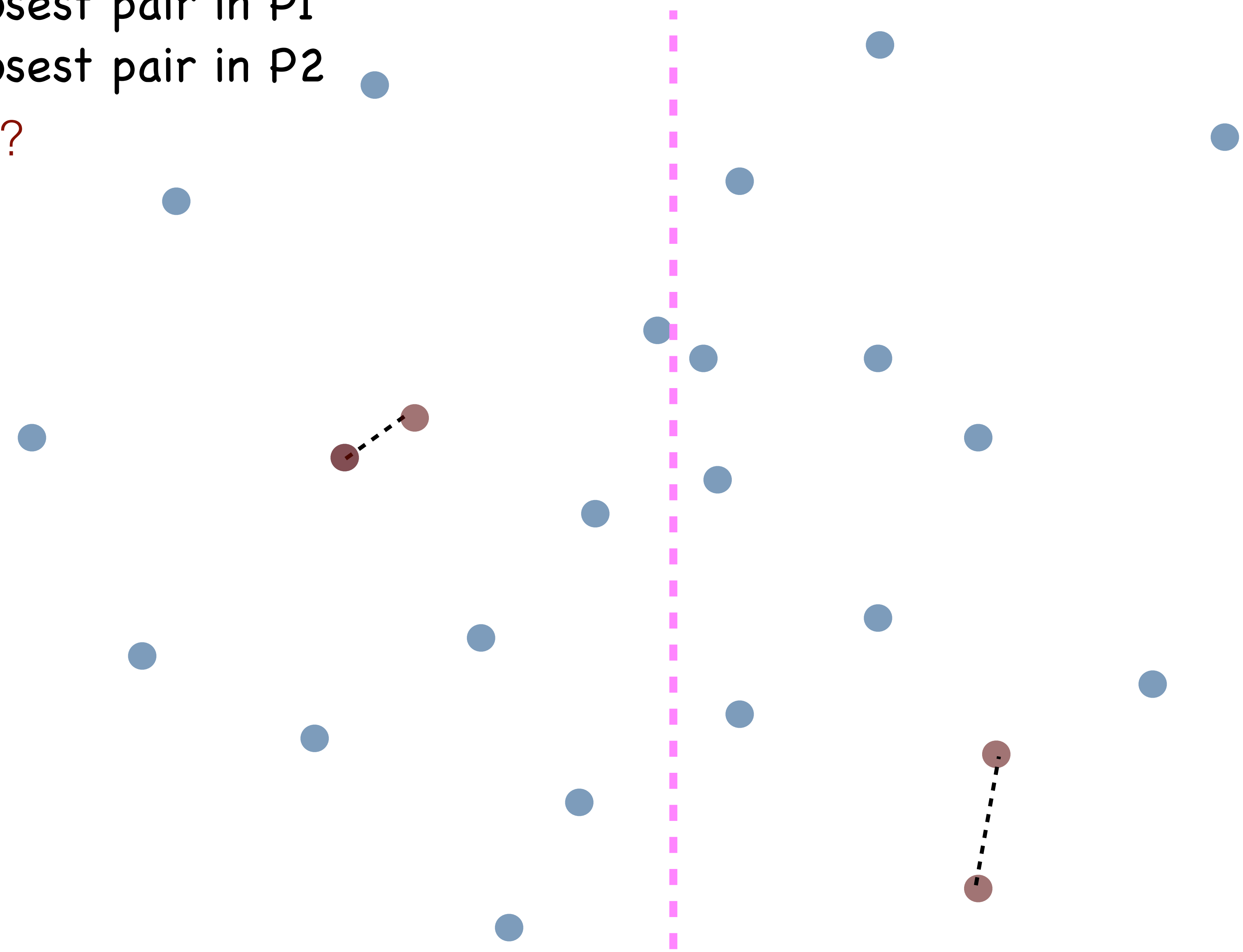
Divide-and-conquer for closest pair

- 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



Divide-and-conquer for closest pair

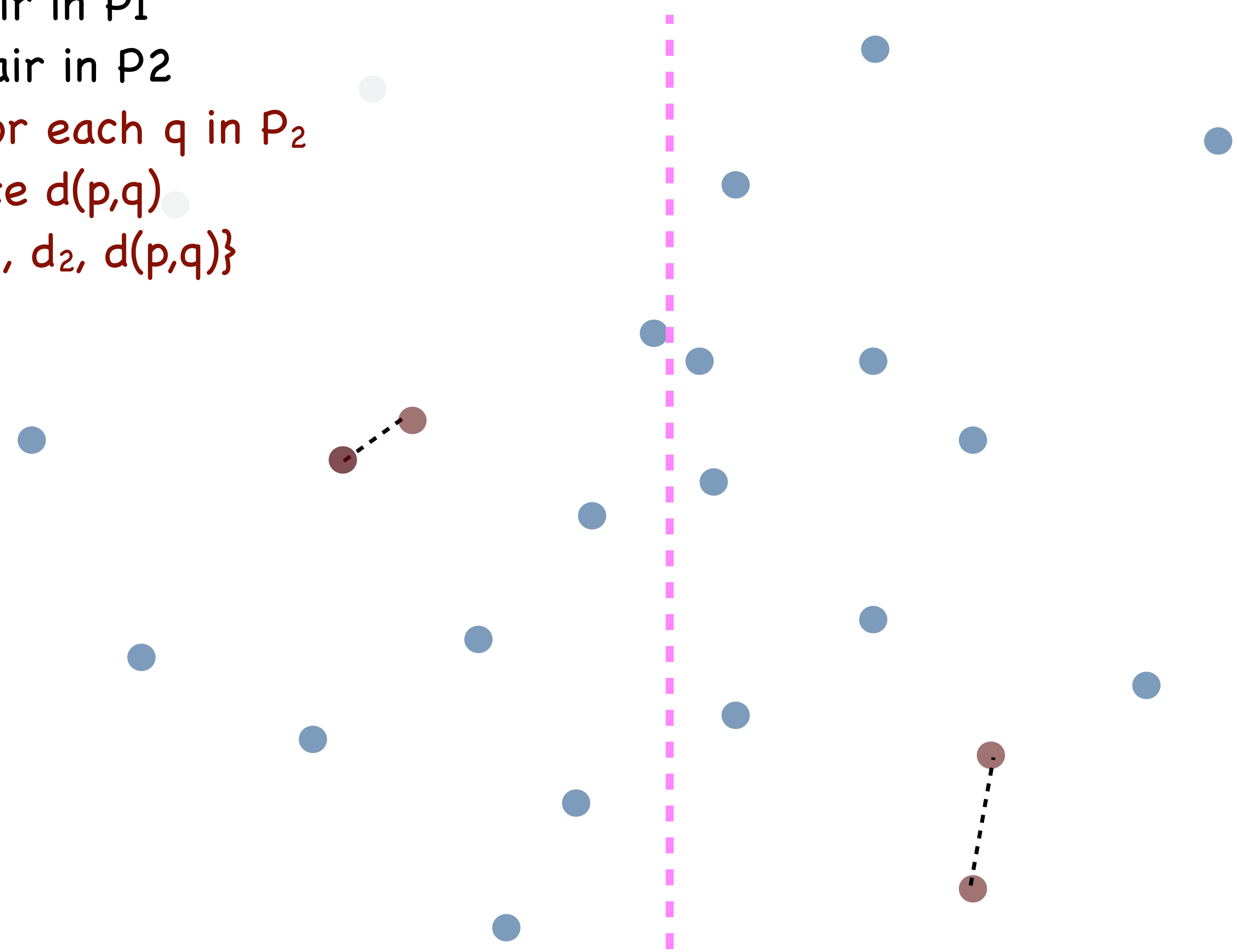
- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find closest pair in P1
- recursively find closest pair in P2
- …… //NOW WHAT?



How can you find closest pair in P?

Divide-and-conquer for closest pair

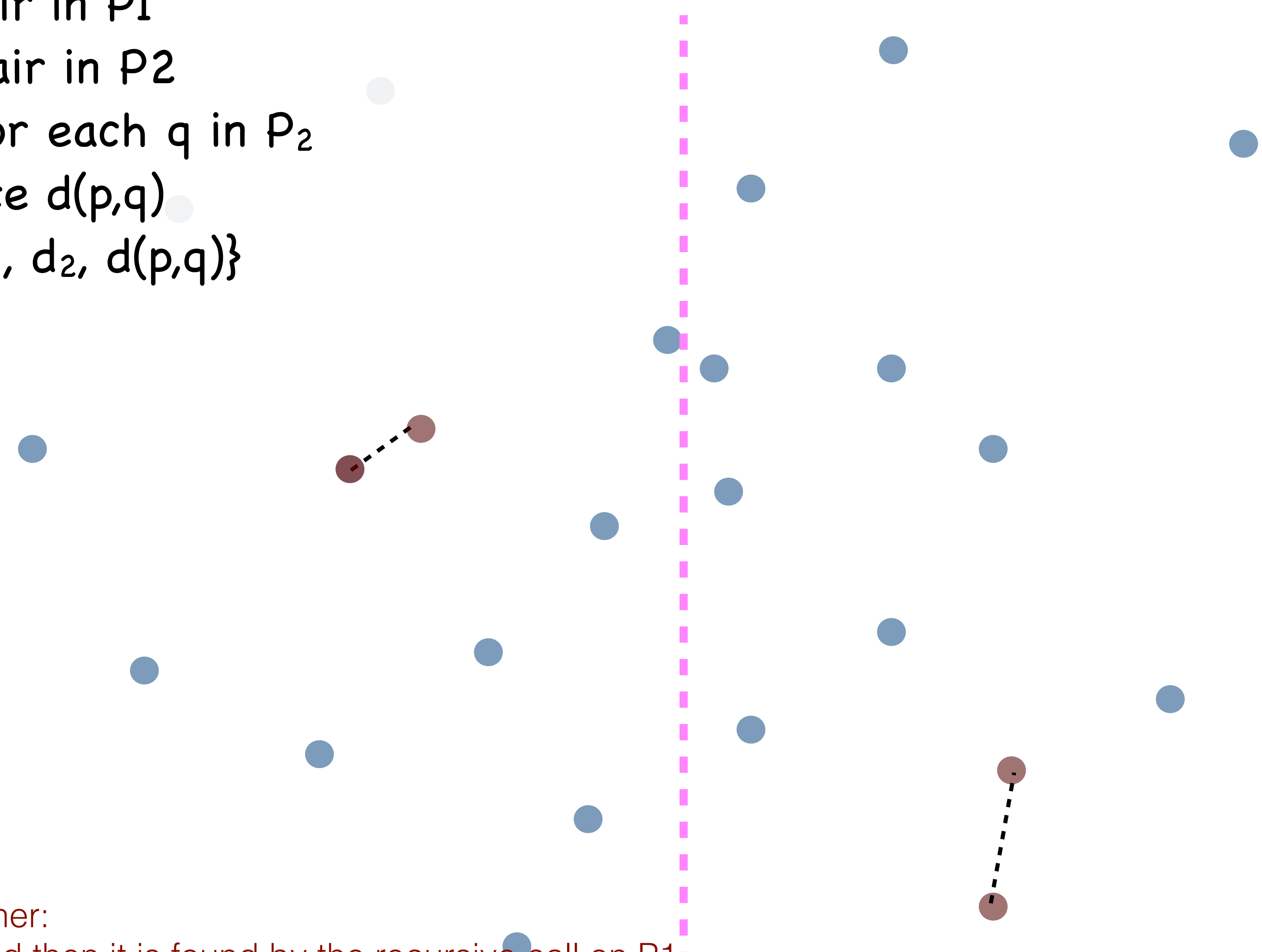
- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- d_1 = find closest pair in P1
- d_2 = find closest pair in P2
- for each p in P_1, for each q in P_2
  - compute distance d(p,q)
  - mindist = min{d_1, d_2, d(p,q)}



1. Is this correct?
2. Running time?

Divide-and-conquer for closest pair

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- d_1 = find closest pair in P1
- d_2 = find closest pair in P2
- 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)}



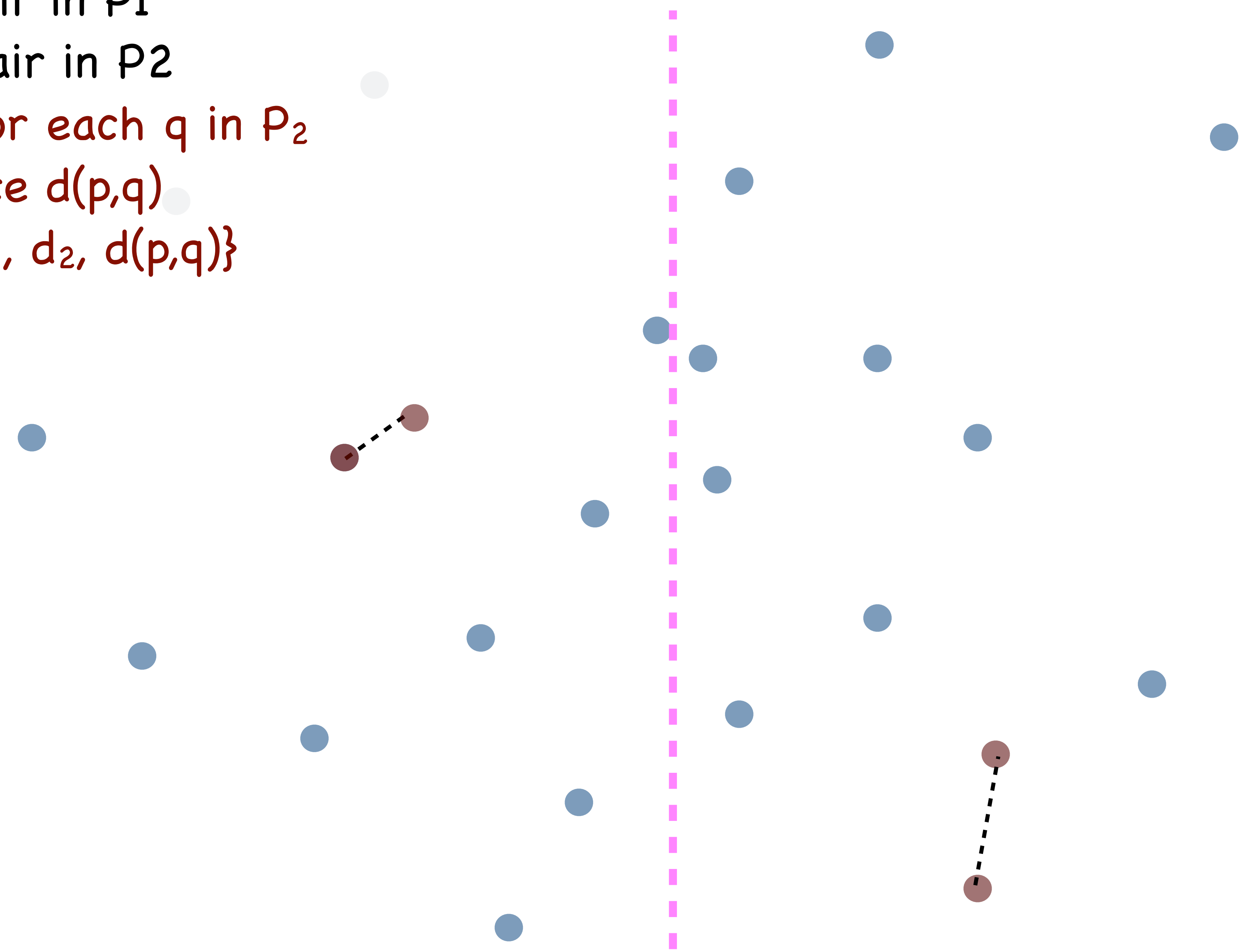
Is this correct?

YES. The closest pair is either:

- 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 consider all such pairs

Divide-and-conquer for closest pair

- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- d_1 = find closest pair in P1
- d_2 = find closest pair in P2
- for each p in P_1, for each q in P_2
  - compute distance d(p,q)
  - mindist = $\min\{d_1, d_2, d(p,q)\}$

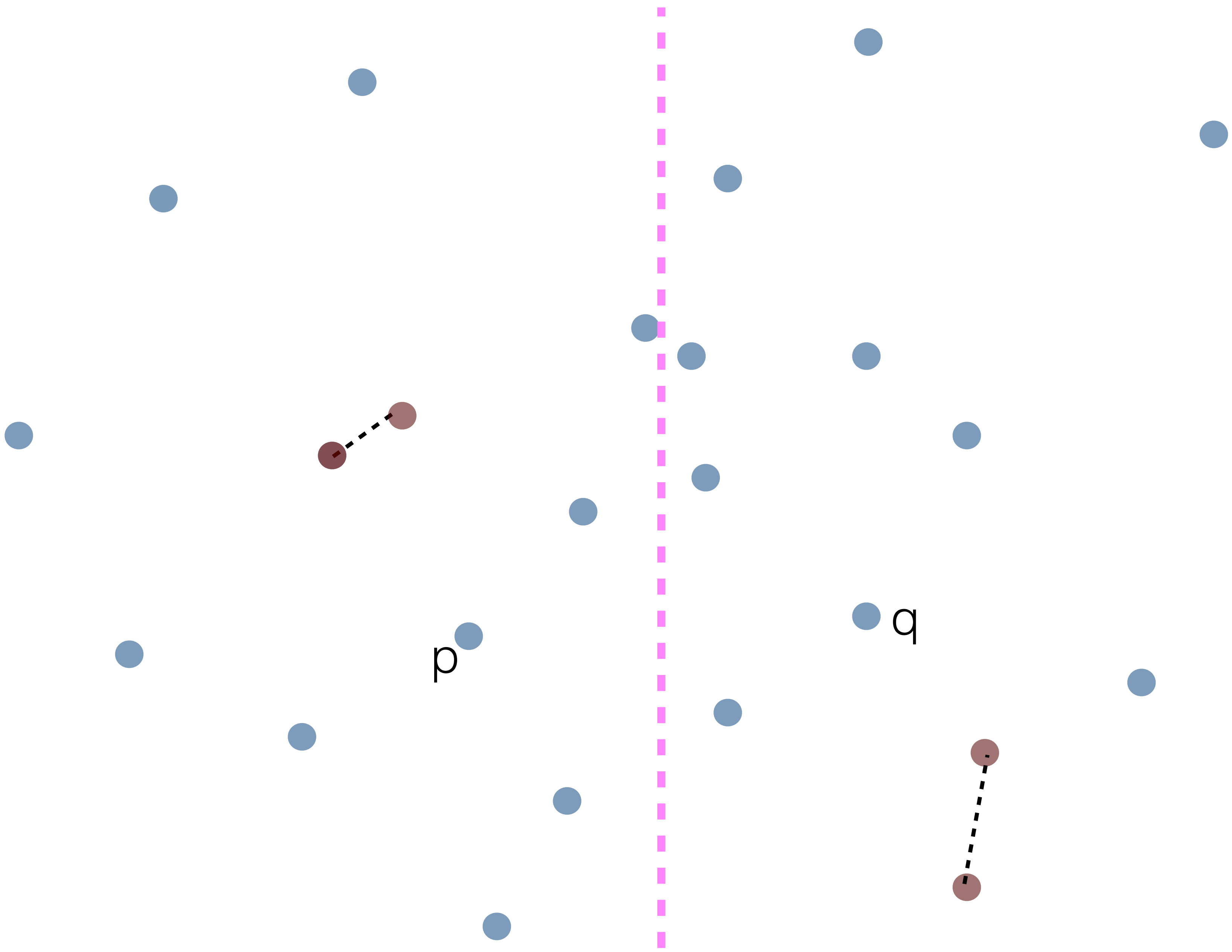


Running time?

- $T(n) = 2T(n/2) + O(n^2)$ => solves to $O(n^2)$

Refining the merge

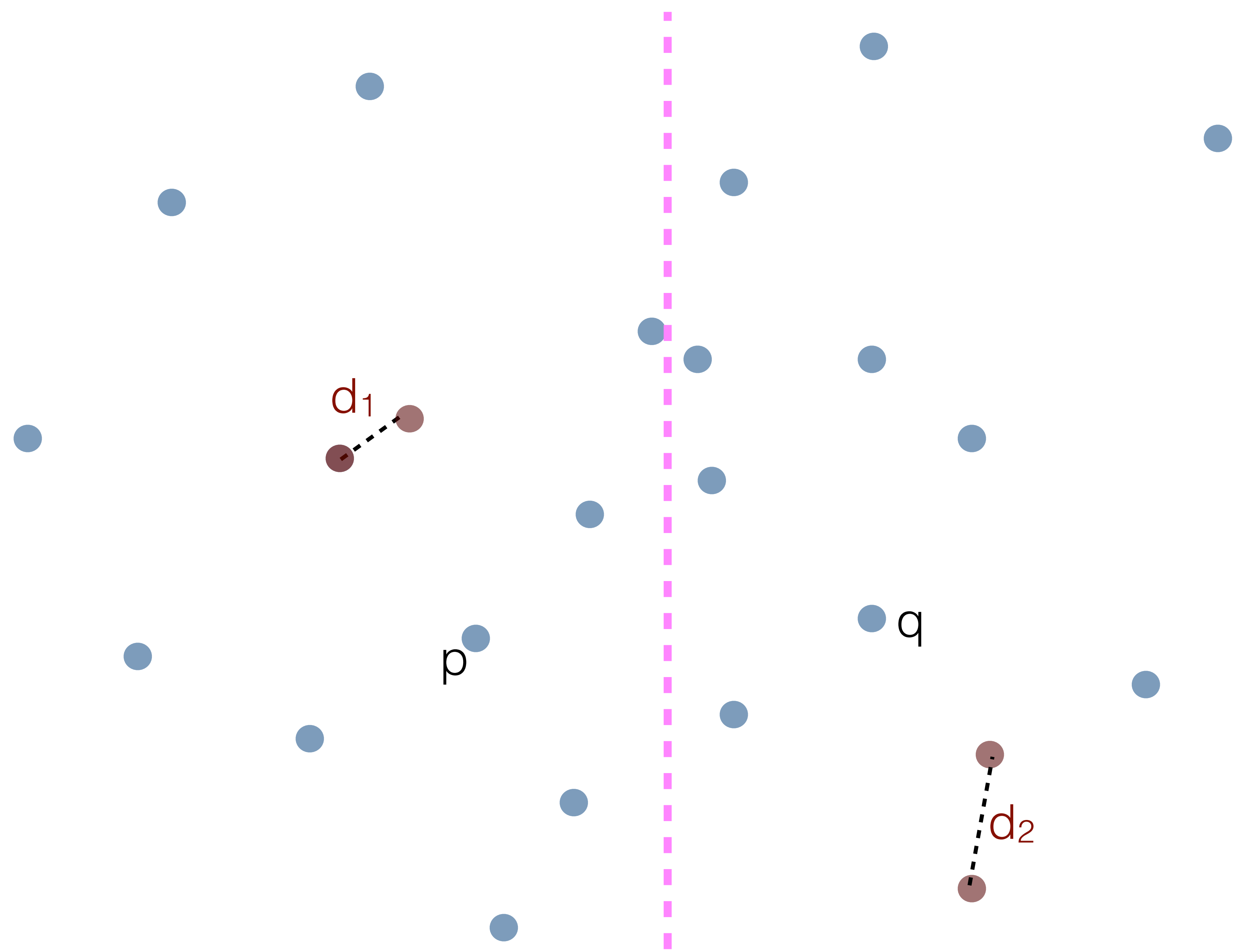
Do we need to examine all pairs (p,q), with p in P_1, q in P_2?



Can (p,q) be the closest pair?

Refining the merge

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

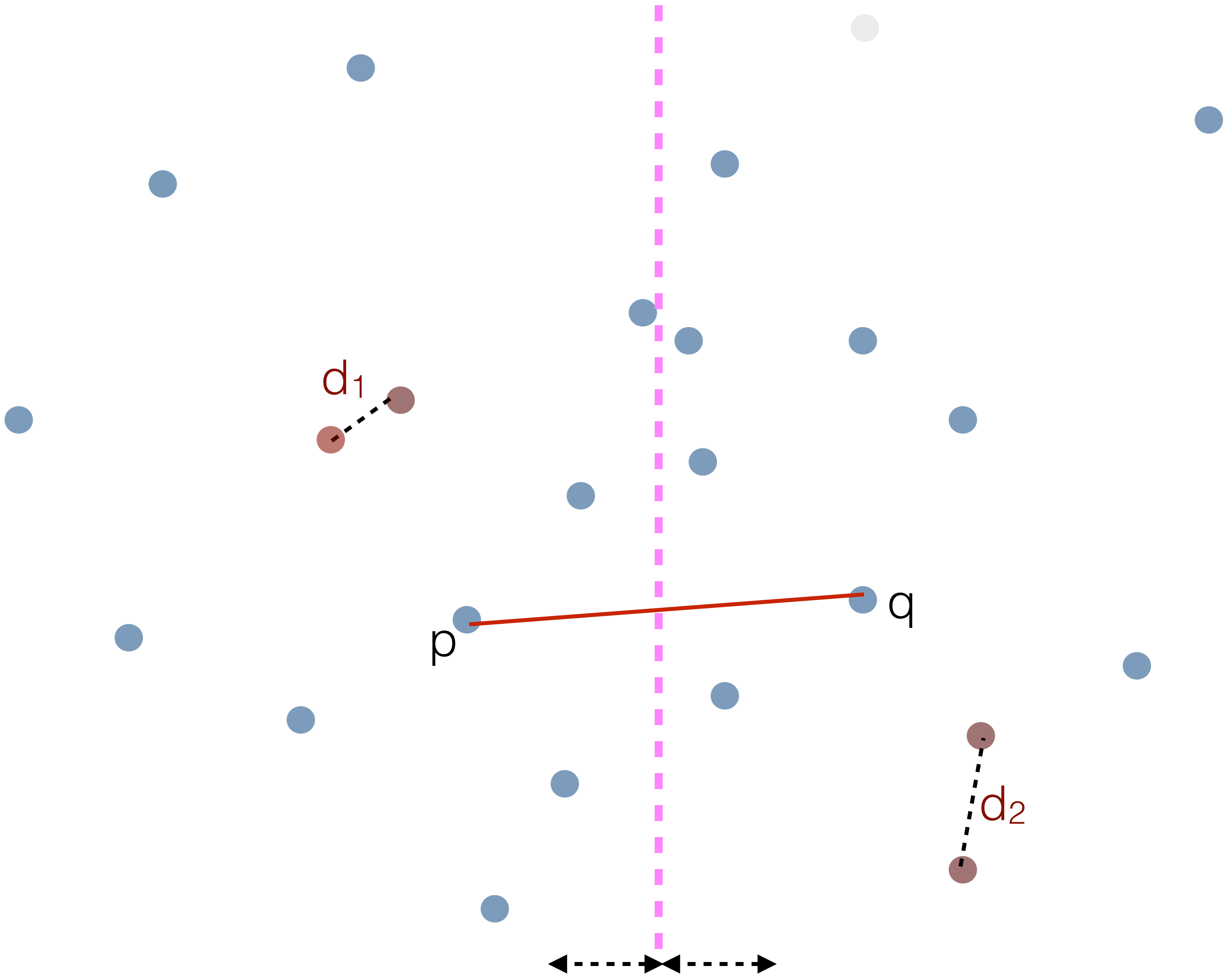


Can (p,q) be the closest pair?

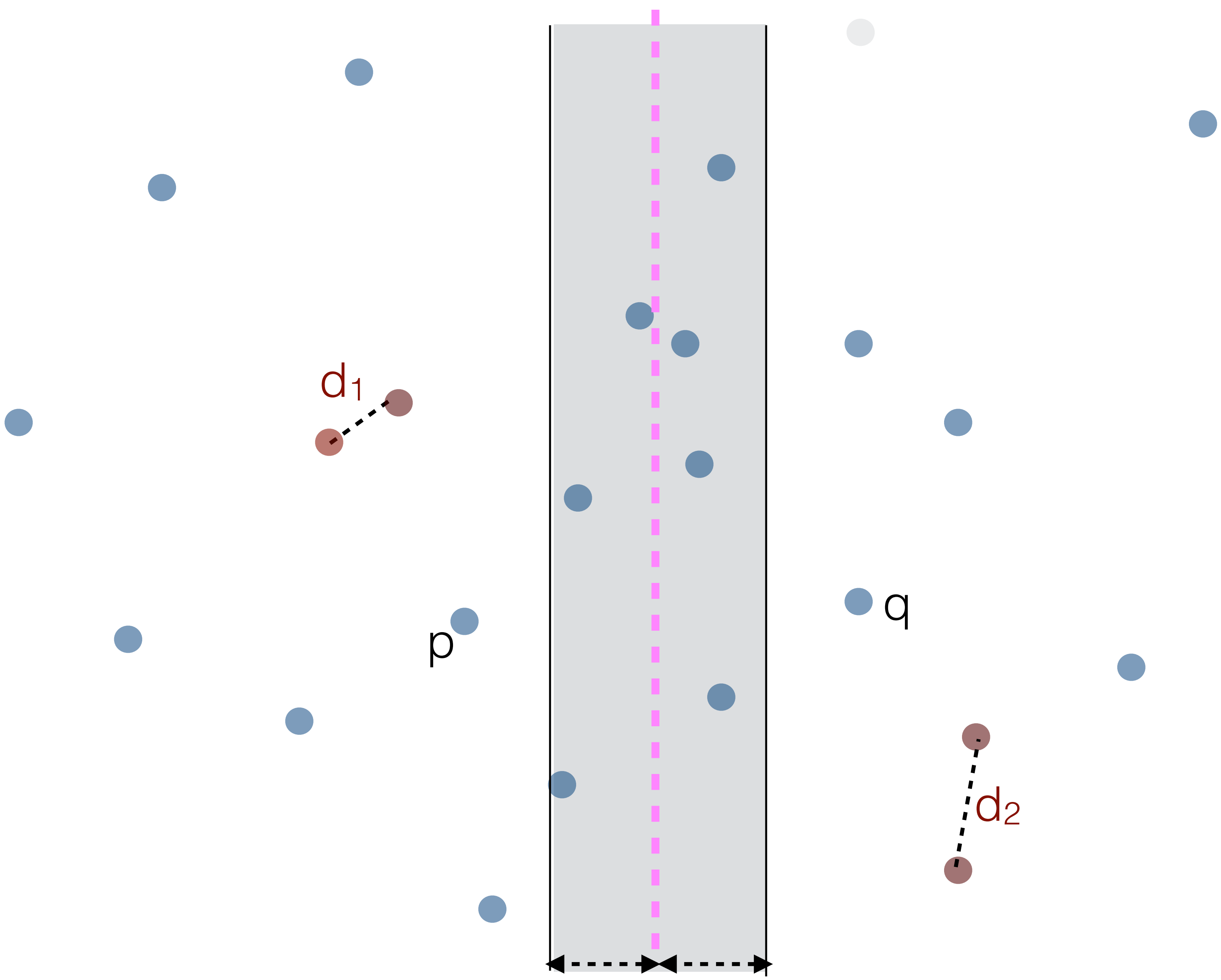
Why not? Where do p,q need to lie in order to be the closest pair?

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

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



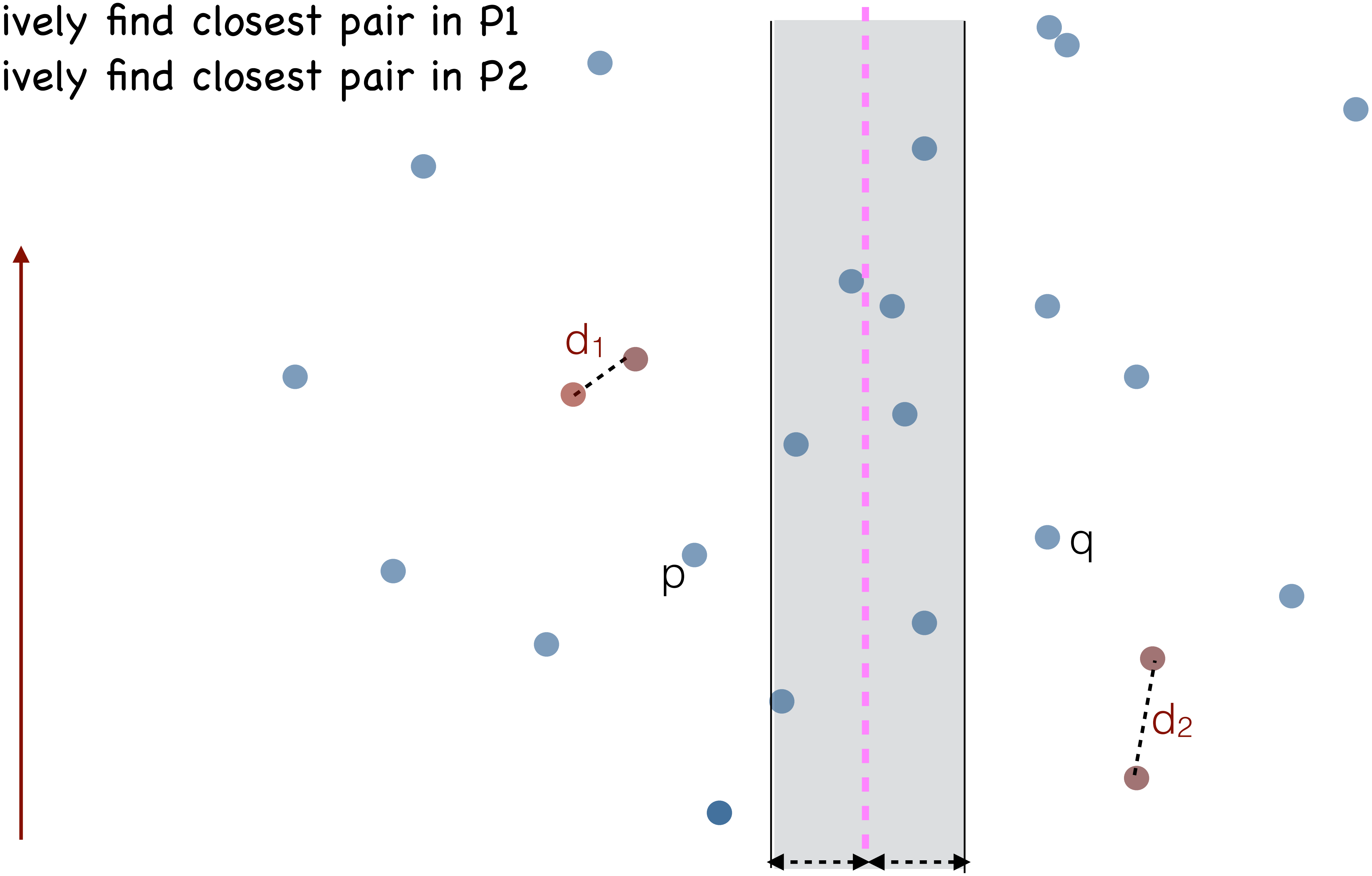
Claim: In order to be candidates for closest pair, points p, q must lie in the d-by-d strip centered at the median.



Proof:

Refining the merge

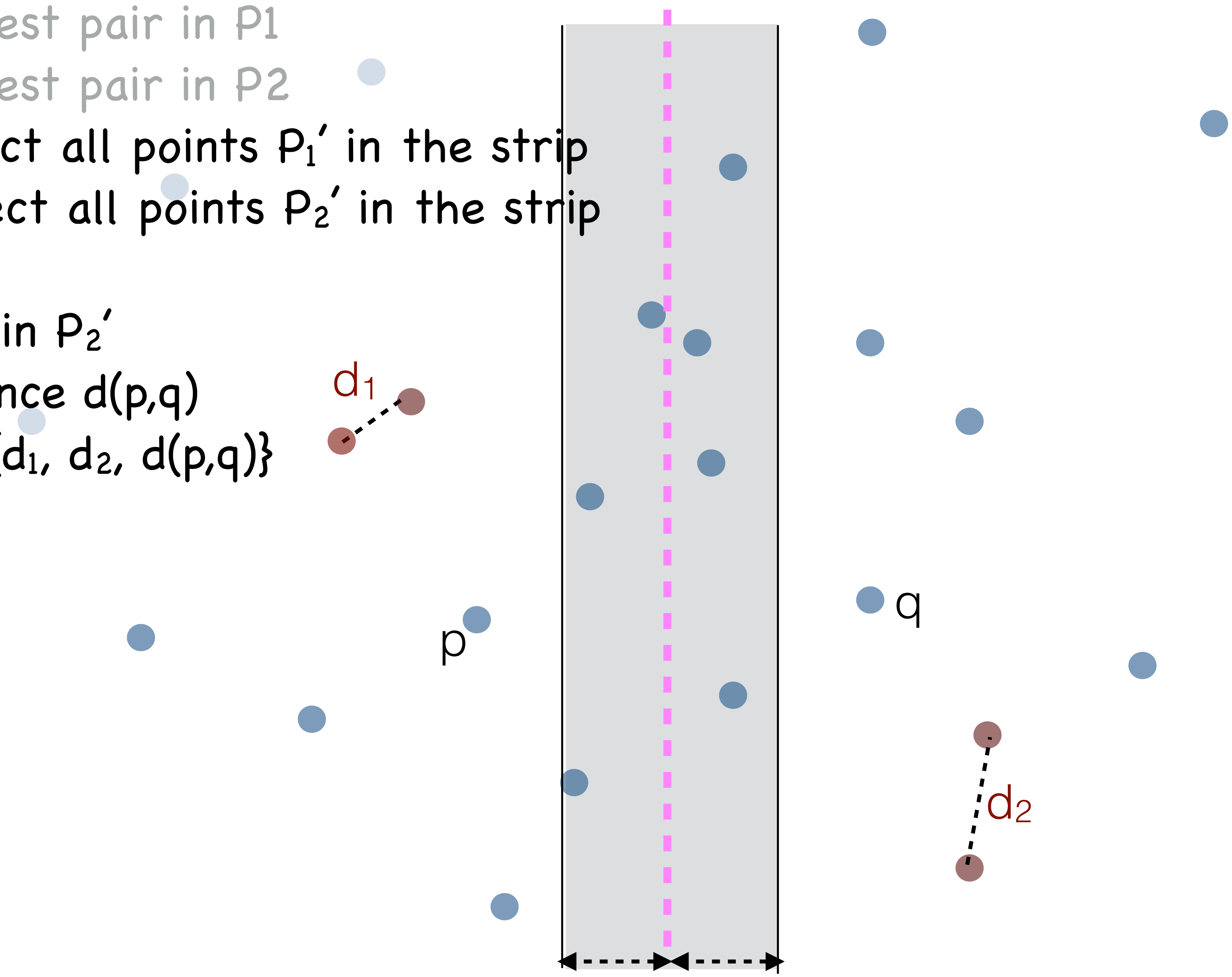
- find vertical line that splits P in half
- let P1, P2 = set of points to the left/right of line
- recursively find closest pair in P1
- recursively find closest pair in P2
- .....



Fill in the details of the new algorithm's merge phase and analyze it.

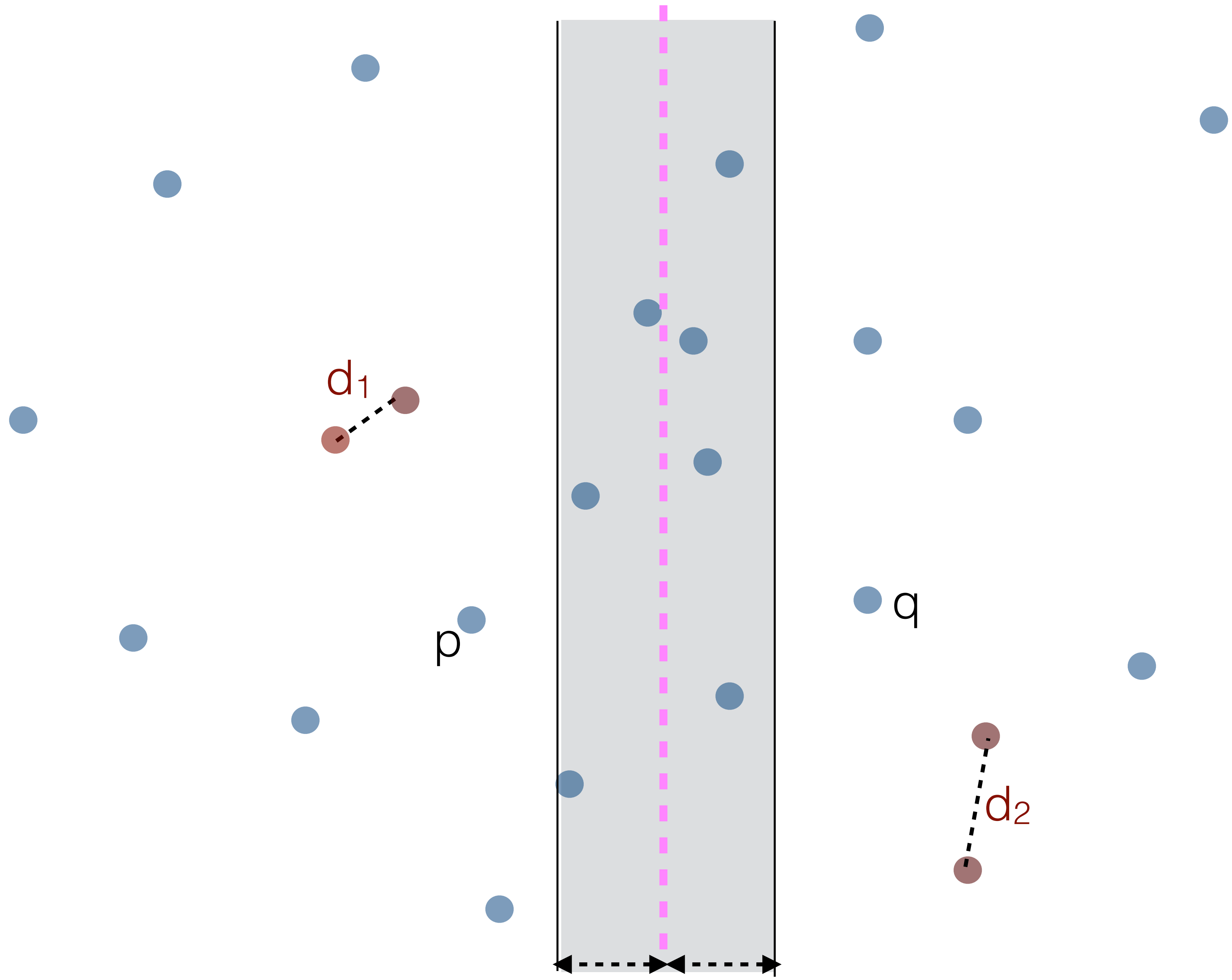
Refining the merge

- 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
- 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)}



Refining the merge

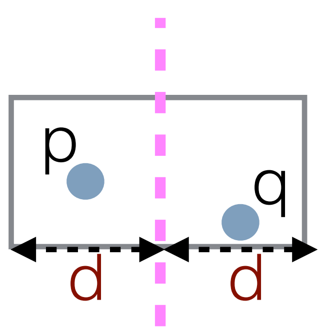
- Show an example where the strip may contain Omega(n) points.

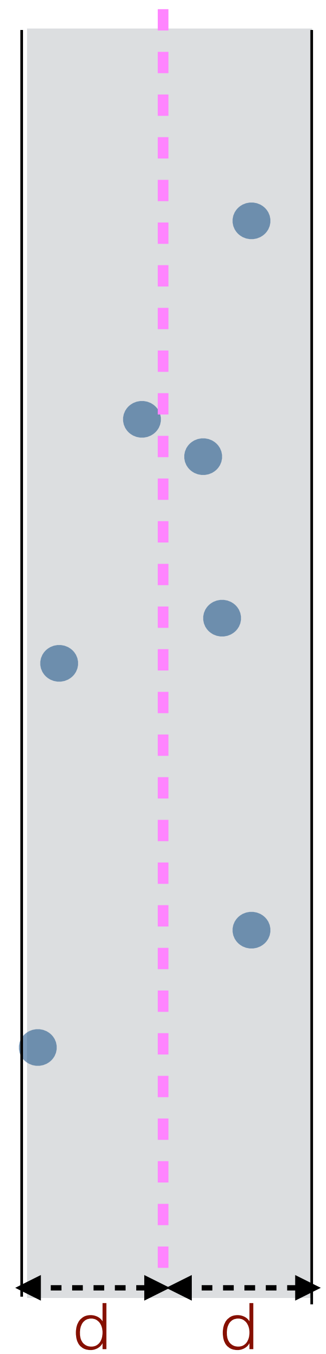


- What does this imply for the running time?

Refining the merge

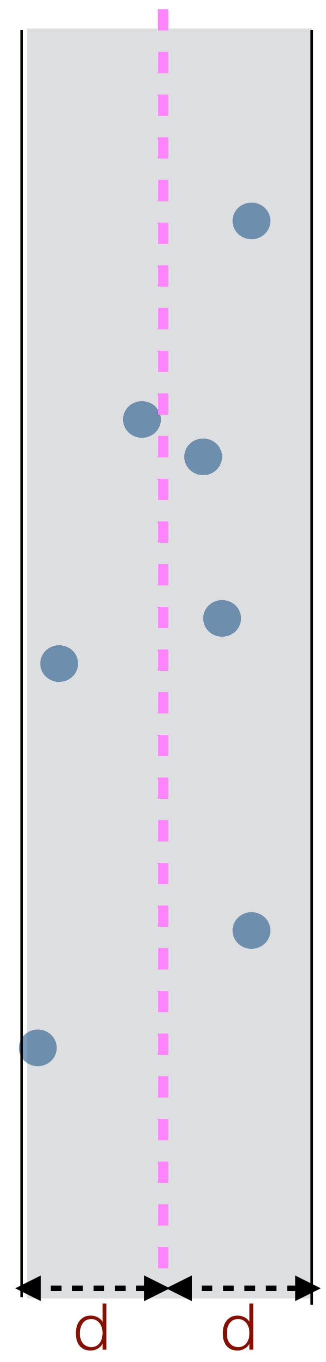
- Ok, so this is not yet enough
- But … we also know that the vertical distance between p and q cannot be greater than d.





Refining the merge

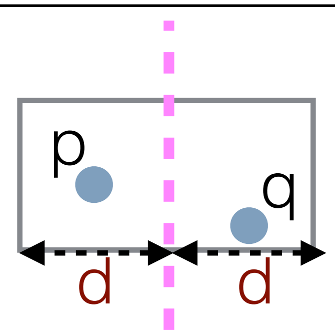
- Consider a point p in the stripe. How many points below it, at most, could be candidates for the closest pair (p,q)?



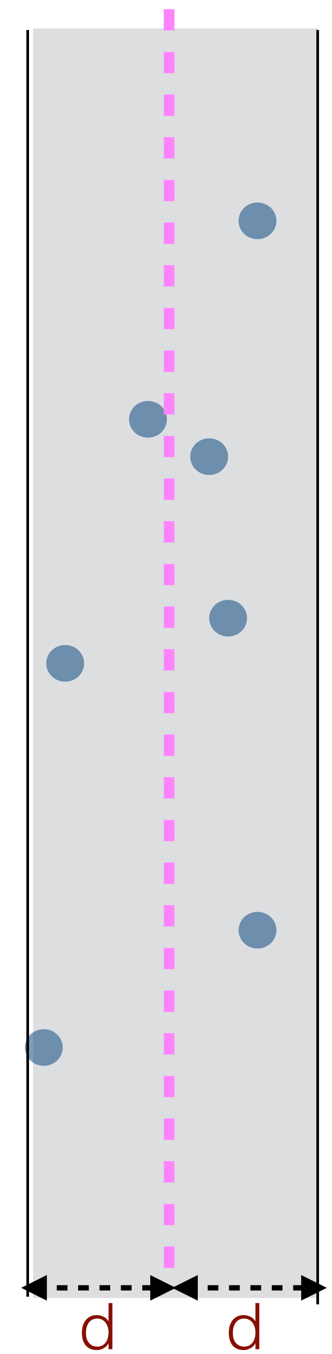
Refining the merge

- Consider a point p in the stripe. How many points below it, at most, could be candidates for the closest pair (p,q)?

p,q must lie in 2d-by-d rectangle



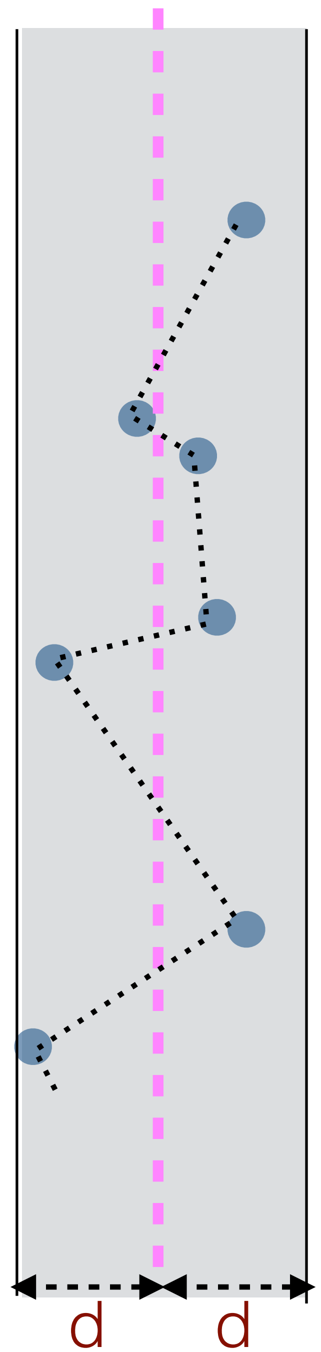
any pair of points in the left side must be at least d away



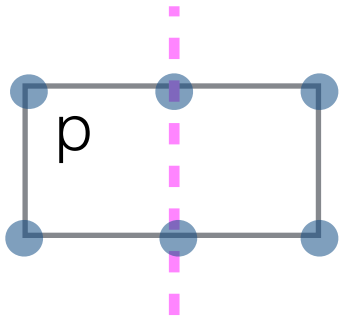
any pair of points in the right side must be at least d away

Refining the merge

Claim:
A point p needs to check at most 5 points following p in y-order.



Note: Assume no duplicate points.



Refining the merge

- Put all these together and write down the algorithm.
- Analyze the running time.

Refining the merge

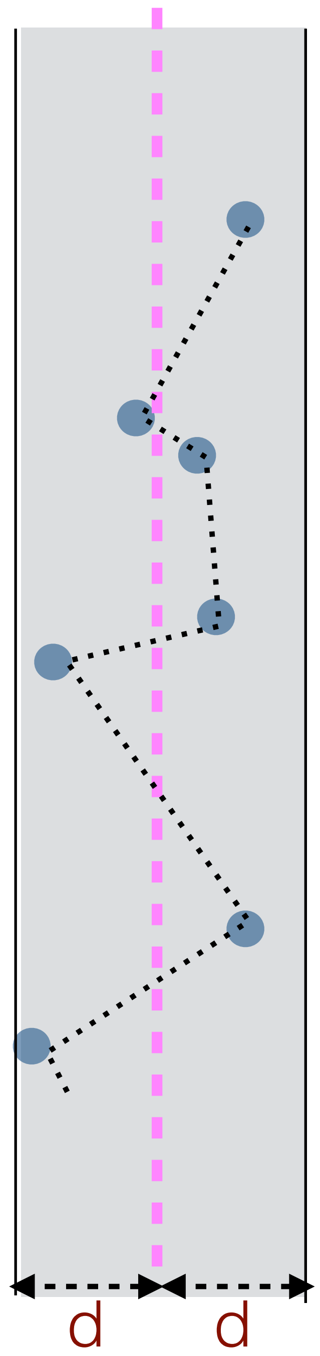
closestPair(P)

//divide

- find vertical line that splits P in half
- let P_1, P_2 = set of points to the left/right of line
- call closestPair(P_1); let d_1 be the returned closest distance
- call closestPair(P_2); let d_2 be the returned closest distance

//merge

- let d = min{d_1, d_2}
- Strip= empty
- for all p in P_1: if x_p > x_vertical - d: add p to Strip
- for all p in P_2: if x_p < x_vertical + 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

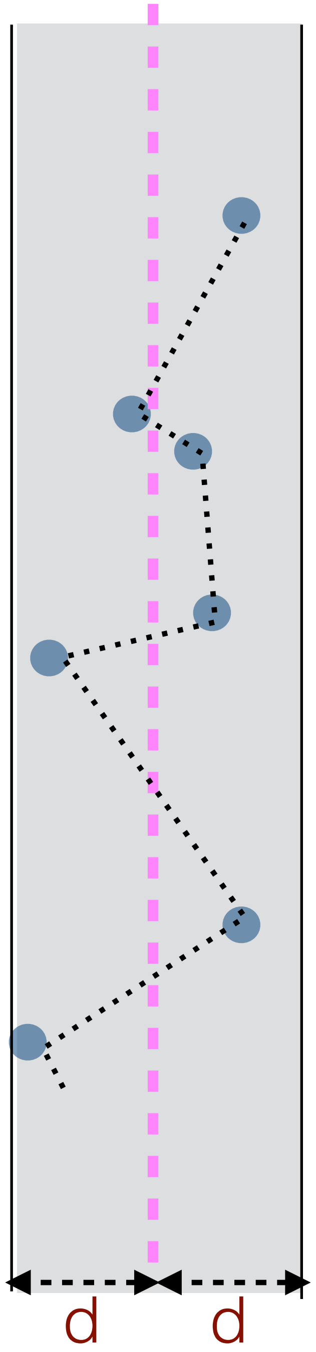
closestPair(P)

//divide

- find vertical line that splits P in half
- let P_1, P_2 = set of points to the left/right of line
- call closestPair(P_1); let d_1 be the returned closest distance
- call closestPair(P_2); let d_2 be the returned closest distance

//merge

- let d = min{d_1, d_2)
- Strip= empty
- for all p in P_1: if x_p > x_vertical - d: add p to Strip
- for all p in P_2: if x_p < x_vertical + 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: ?

Refining the merge

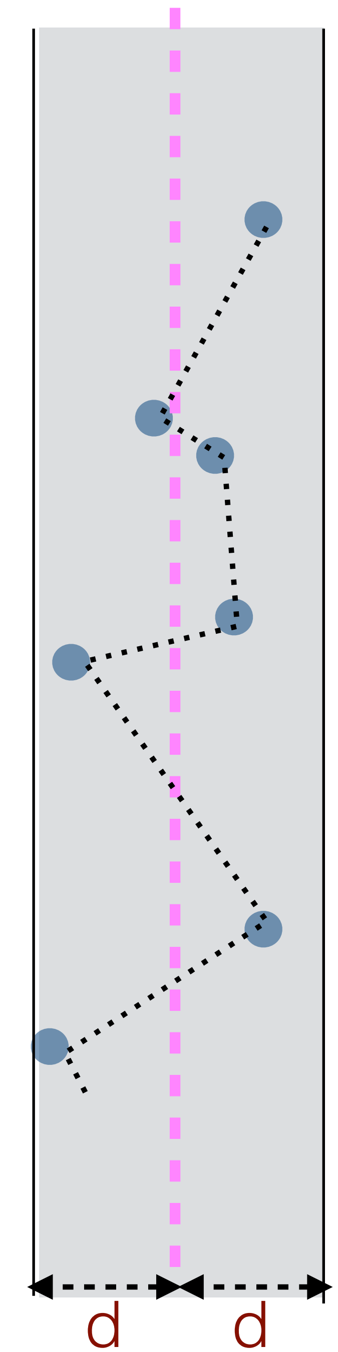
closestPair(P)

//divide

- find vertical line that splits P in half
- let P_1, P_2 = set of points to the left/right of line
- call closestPair(P_1); let d_1 be the returned closest distance
- call closestPair(P_2); let d_2 be the returned closest distance

//merge

- let d = min{d_1, d_2)
- Strip= empty
- for all p in P_1: if x_p > x_vertical - d: add p to Strip
- for all p in P_2: if x_p < x_vertical + 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: $T(n) = 2T(n/2) + O(n \lg n) \Rightarrow O(n \lg^2 n)$

Divide-and-conquer for closest pair

- Avoiding the sort

Divide-and-conquer for closest pair

- Describe in full detail how to avoid sorting at every level, and give the detailed pseudocode. Include an explanation for how to find the vertical line that splits P in half.