

Closest Pair Algorithm

<https://cs.pomona.edu/classes/cs140/>

Outline

Topics and Learning Objectives

- Learn more about Divide and Conquer paradigm
- Learn about the closest-pair problem and its $O(n \lg n)$ algorithm
 - Gain experience analyzing the run time of algorithms
 - Gain experience proving the correctness of algorithms

Exercise

- Closest Pair

Extra Resources

- Algorithms Illuminated: Part 1: Chapter 3

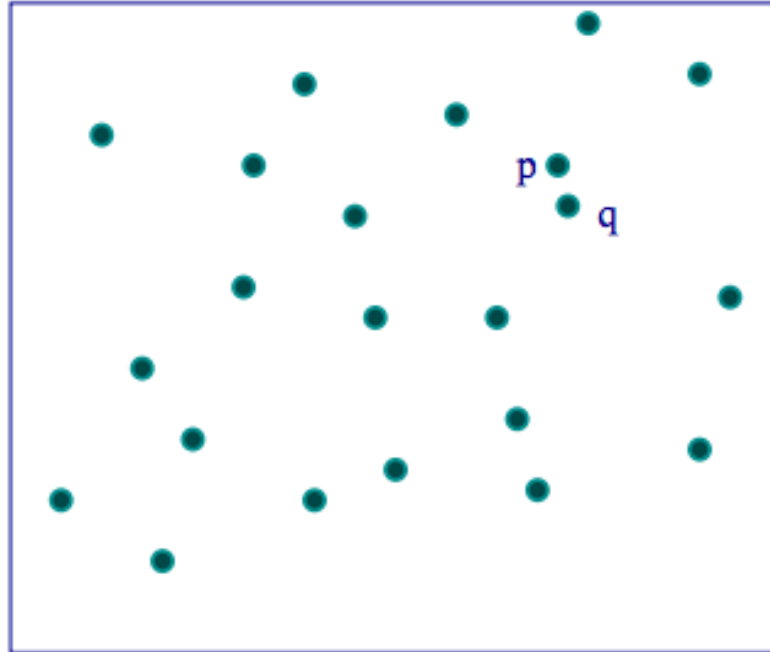
Closest Pair Problem

- **Input**: P , a set of n points that lie in a (two-dimensional) plane
- **Output**: a pair of points (p, q) that are the “closest”
 - Distance is measured using Euclidean distance:

$$d(p, q) = \text{sqrt}((p_x - q_x)^2 + (p_y - q_y)^2)$$

- **Assumptions**: None

Closest Pair Problem



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

- What is the brute force method for this search?
- What is the asymptotic running time of the brute force method?

Input

p1

p2

p3

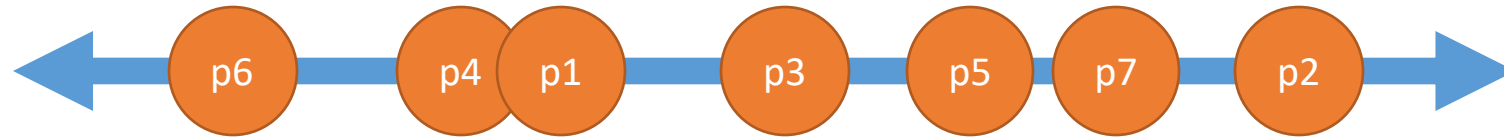
p4

p5

p6

p7

One-dimensional closest pair

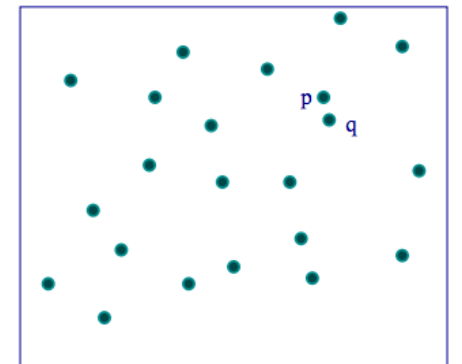


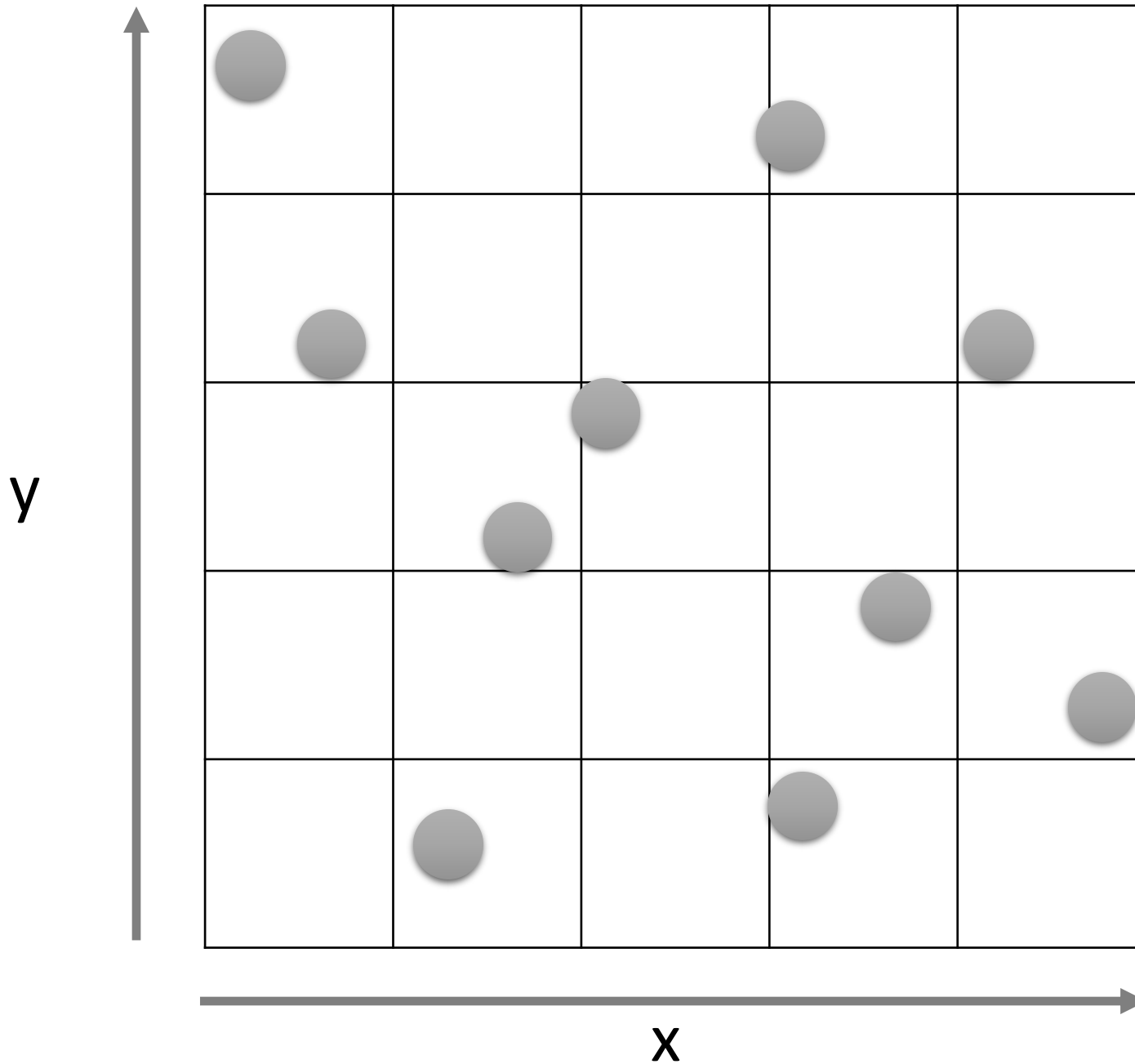
How would you find the closest two points?

- Sort by position : $O(n \lg n)$
- Return the closest two using a linear scan : $O(n)$
- Total time : $O(n \lg n) + O(n) = O(n \lg n)$

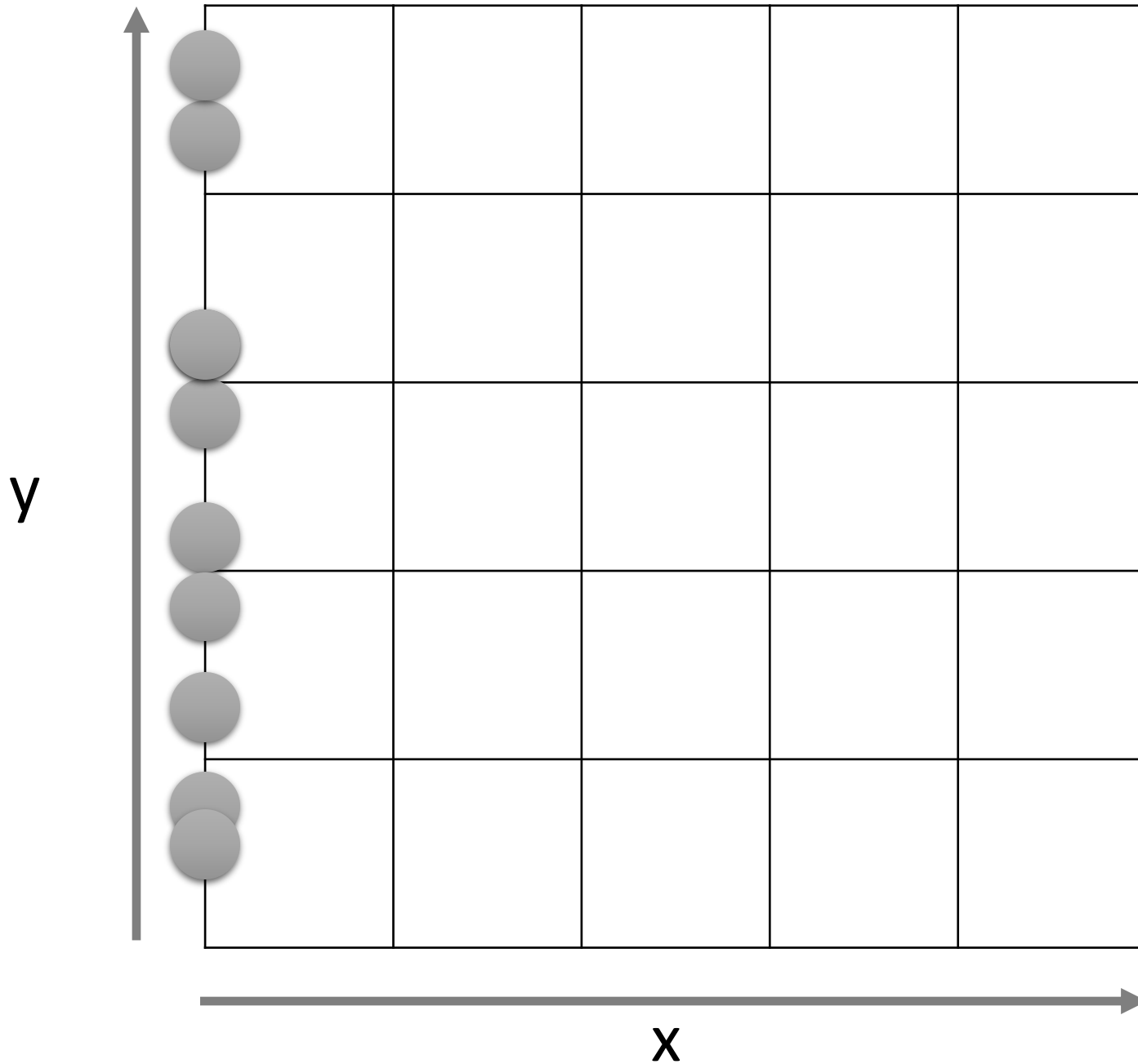
Any problems using this approach for the two-dimensional case?

- Sorting does not generalize to higher dimensions!
- How do you sort the points?

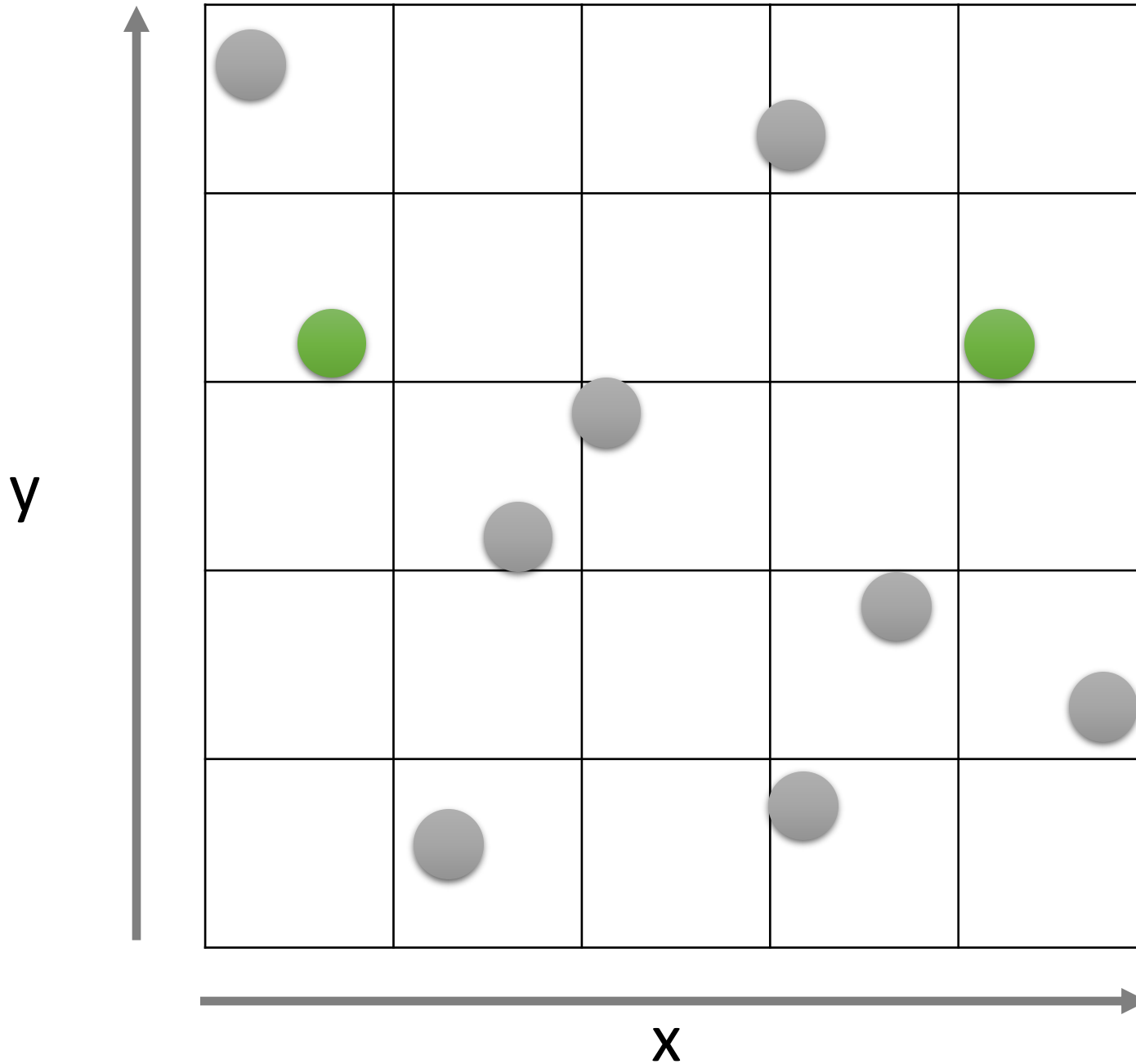




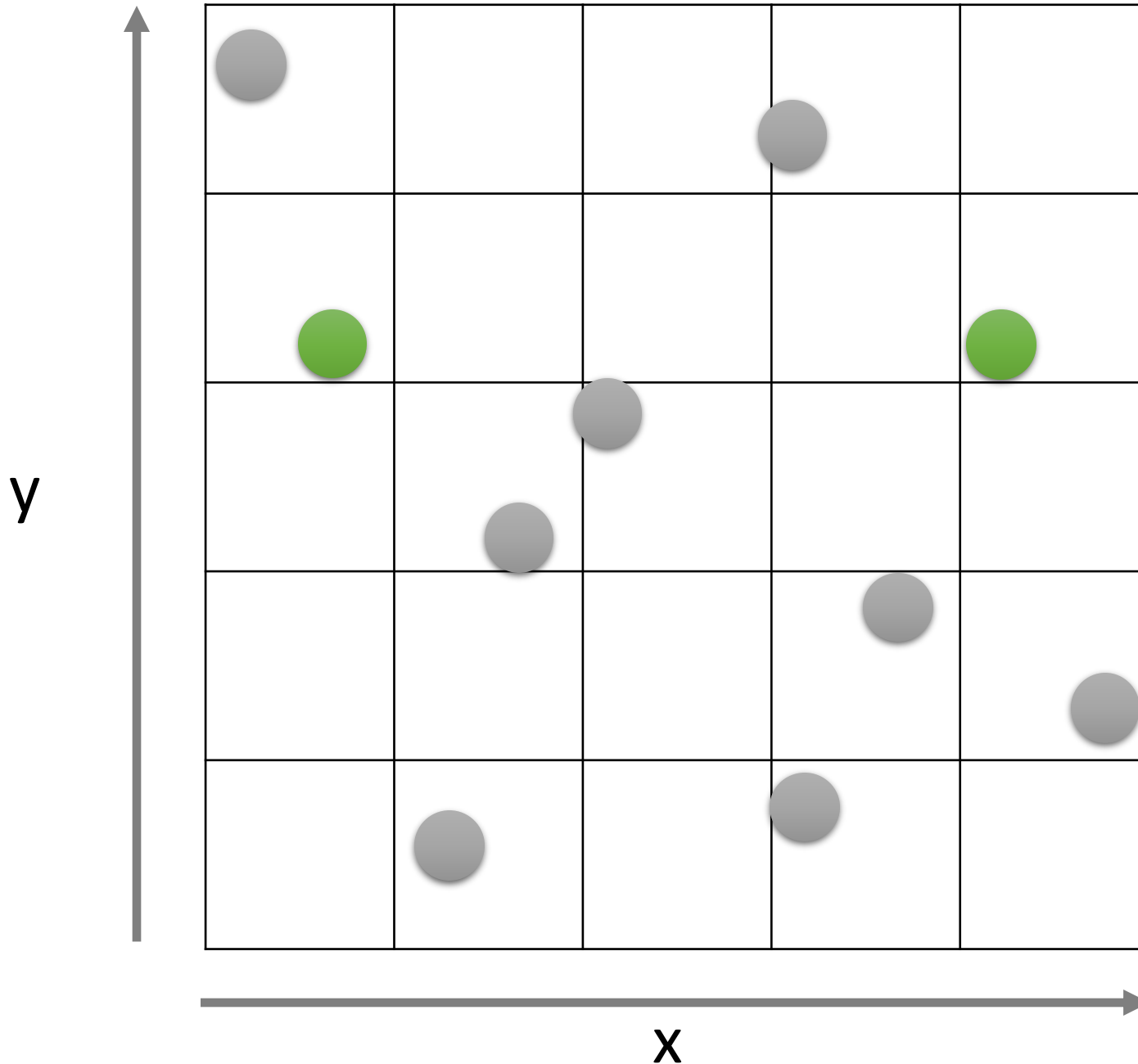
1. Which two are closest on the y-axis?



1. Which two are closest on the y-axis?

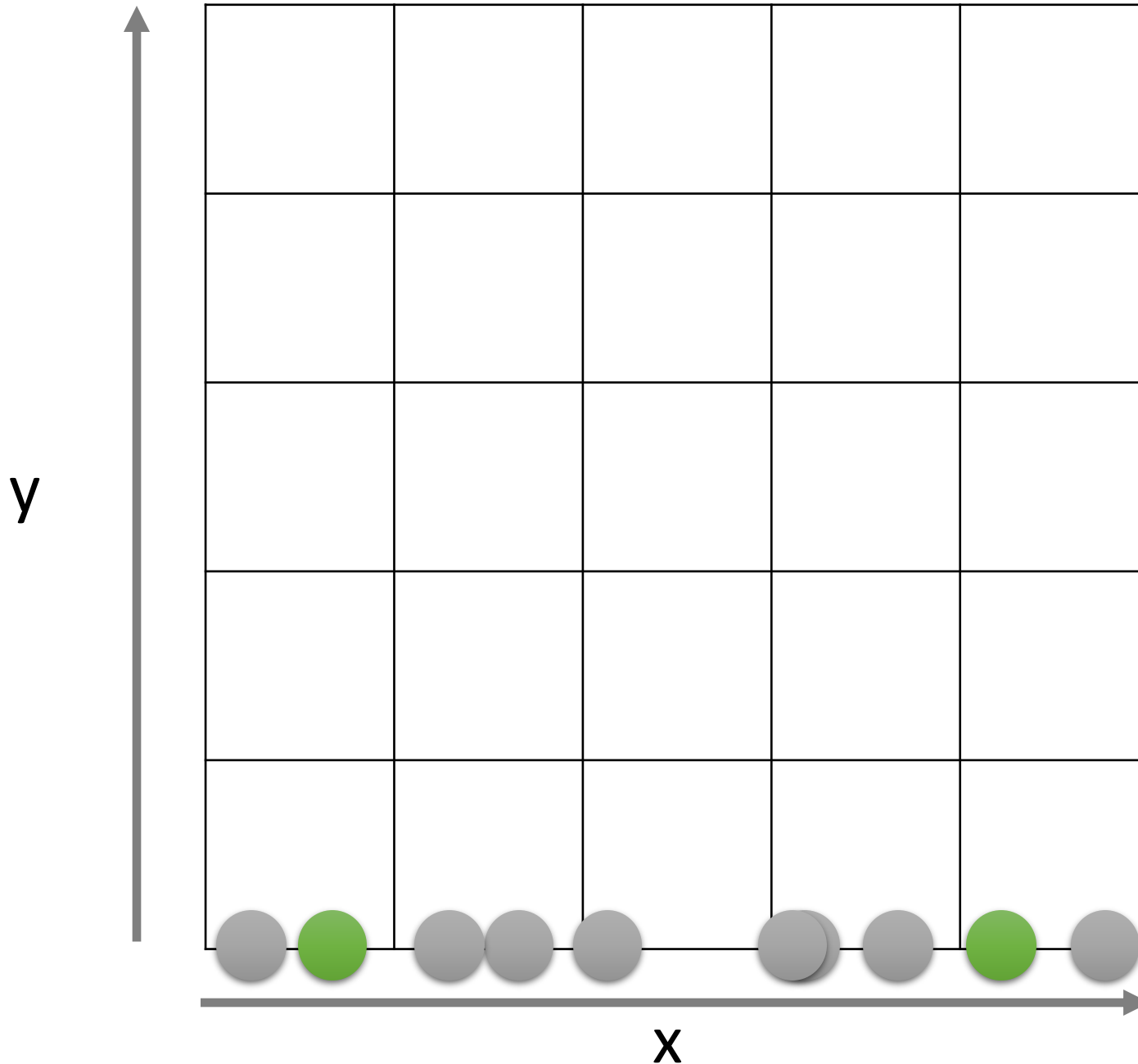


1. Which two are closest on the y-axis?

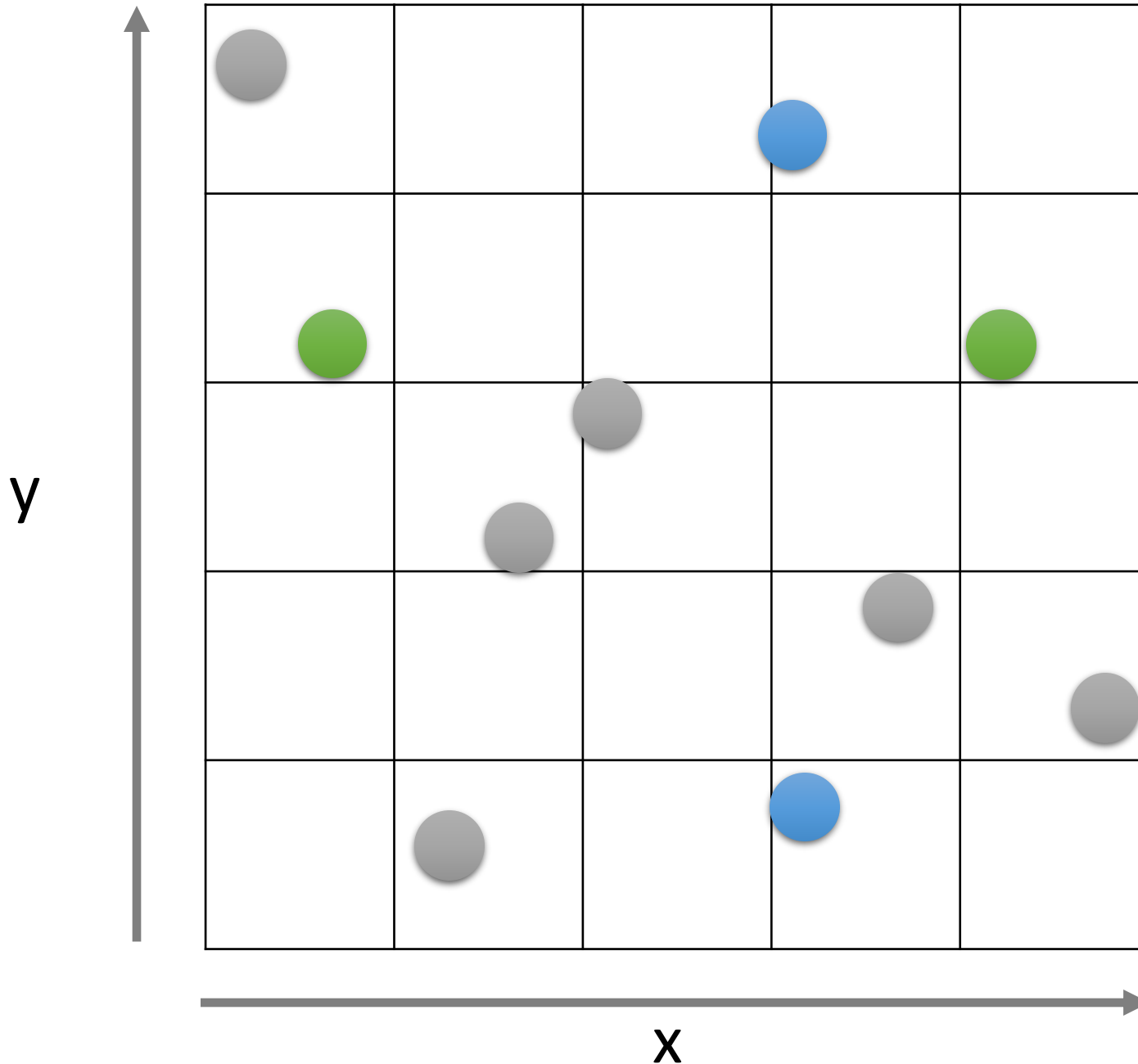


1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?



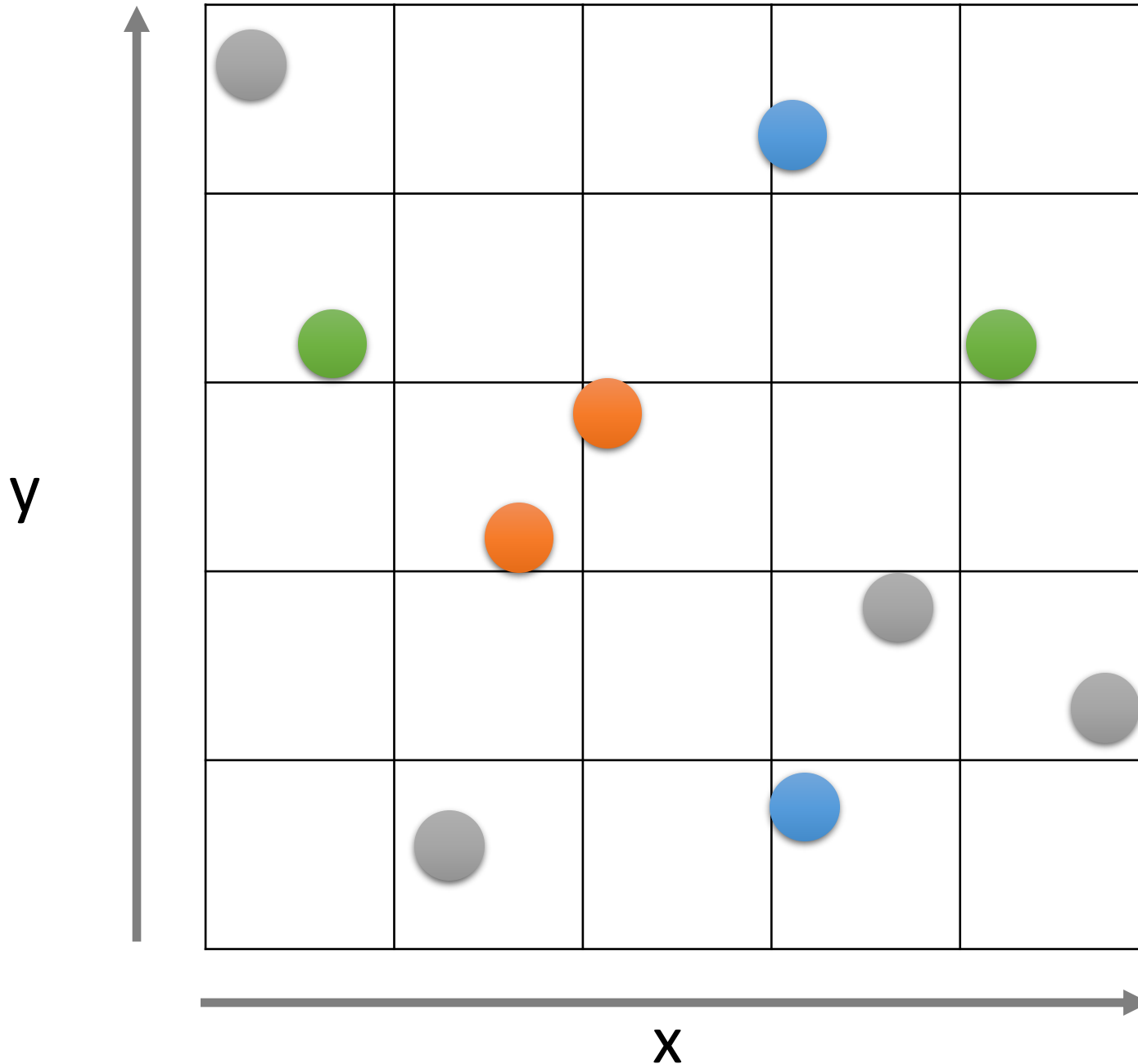
1. Which two are closest on the y-axis?
2. Which two are closest on the x-axis?



1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?



1. Which two are closest on the y-axis?
2. Which two are closest on the x-axis?
3. Which two are closest?

Closest Pair—Two-Dimensions

1. Create a **copy** of the points (we now have two separate copies of P)
 1. Sort by x-coordinate
 2. Sort other by y-coordinate

$O(n \lg n)$

Now we know we can't do better than $O(n \lg n)$

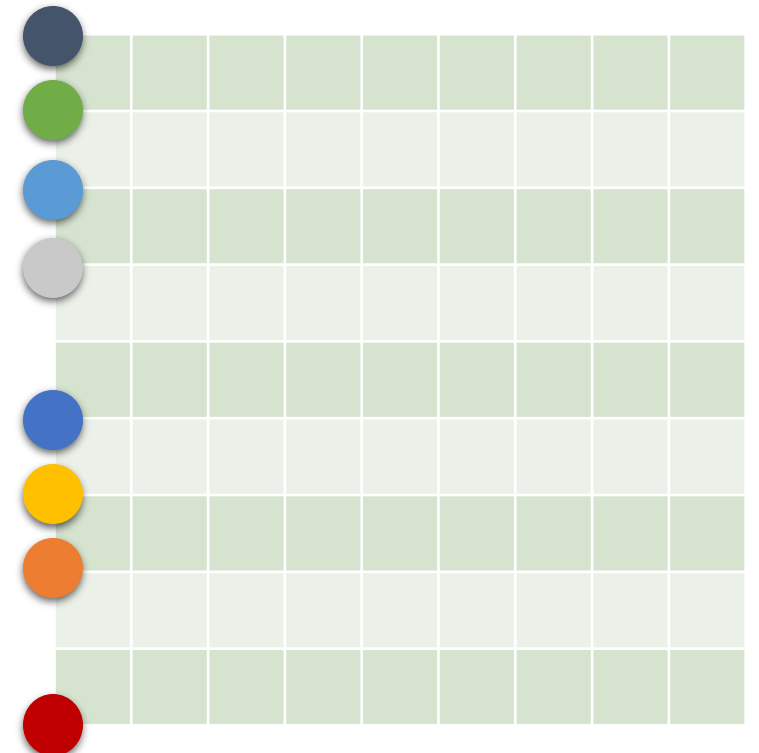
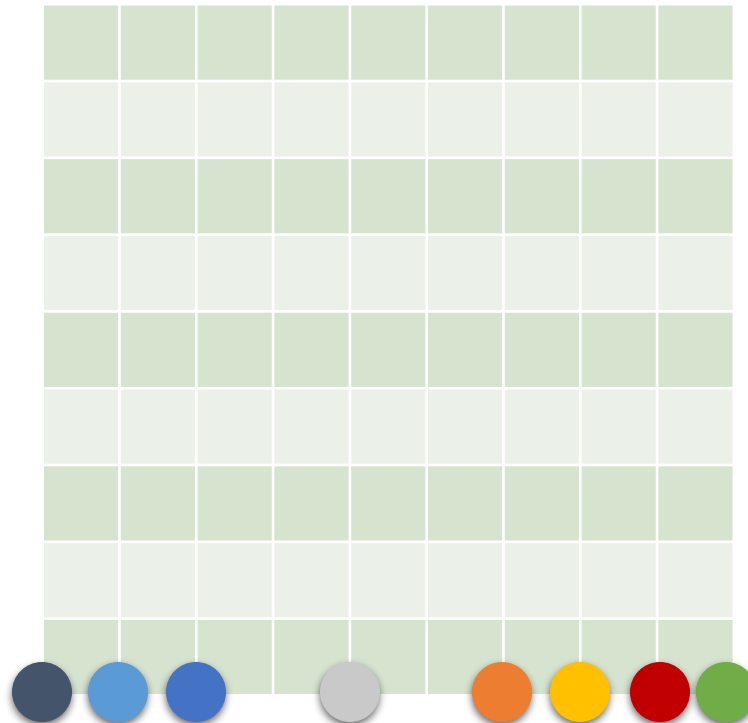
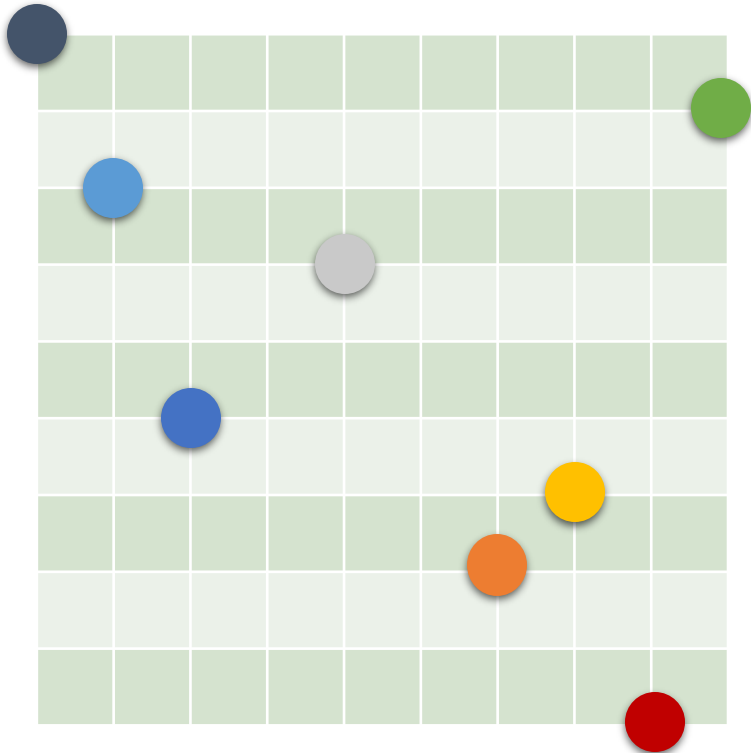
P : [p0(1,10), p1(2,8), p2(7,3), p3(5,7), p4(8,4), p5(3,5), p6(10,9), p7(9,1)]

Sorted by x coordinate

Px : [p0(1,10), p1(2,8), p5(3,5), p3(5,7), p2(7,3), p4(8,4), p7(9,1), p6(10,9)]

Sorted by y coordinate

Py : [p7(9,1), p2(7,3), p4(8,4), p5(3,5), p3(5,7), p1(2,8), p6(10,9), p0(1,10)]



Closest Pair—Two-Dimensions

1. Create a copy of the points (we now have two separate copies of P)

1. Sort by x-coordinate

2. Sort other by y-coordinate

$O(n \lg n)$

- Can we still end up with a $O(n \lg n)$ algorithm for finding the closest pair?
- Does the closeness of two points on one axis matter?

1. **FUNCTION** FindClosestPair(points)
2. points_x = copy_and_sort_by_x(points)
3. points_y = copy_and_sort_by_y(points)
4. **RETURN** ClosestPair(points_x, points_y)

Preprocessing steps

Recursive procedure

Closest Pair—Two-Dimensions

1. Create a copy of the points (we now have two separate copies of P)

1. Sort by x-coordinate
2. Sort other by y-coordinate

$O(n \lg n)$

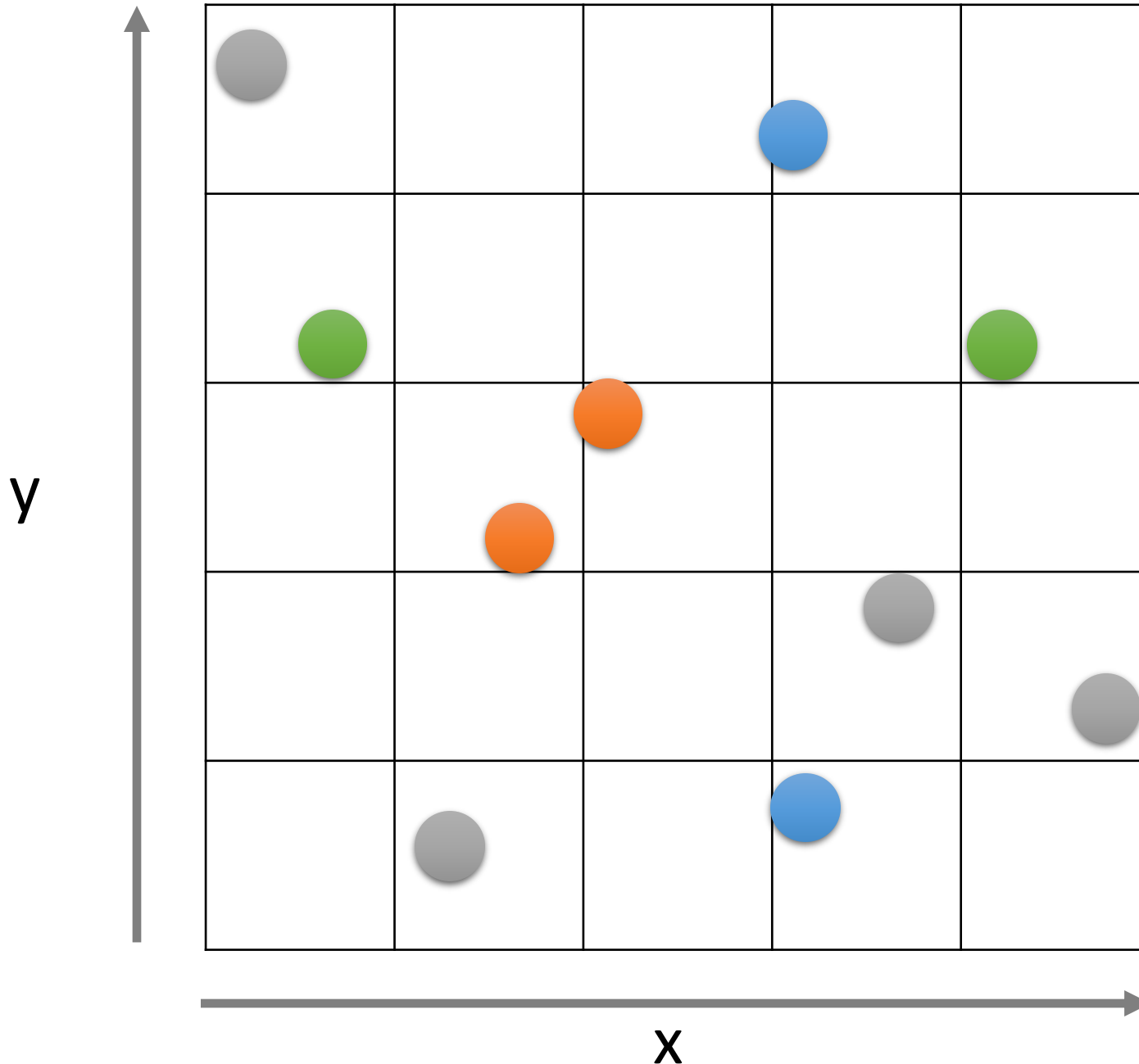
- Can we still end up with a $O(n \lg n)$ algorithm for finding the closest pair?
- Does the closeness of two points on one axis matter?

2. Apply the Divide-and-Conquer method

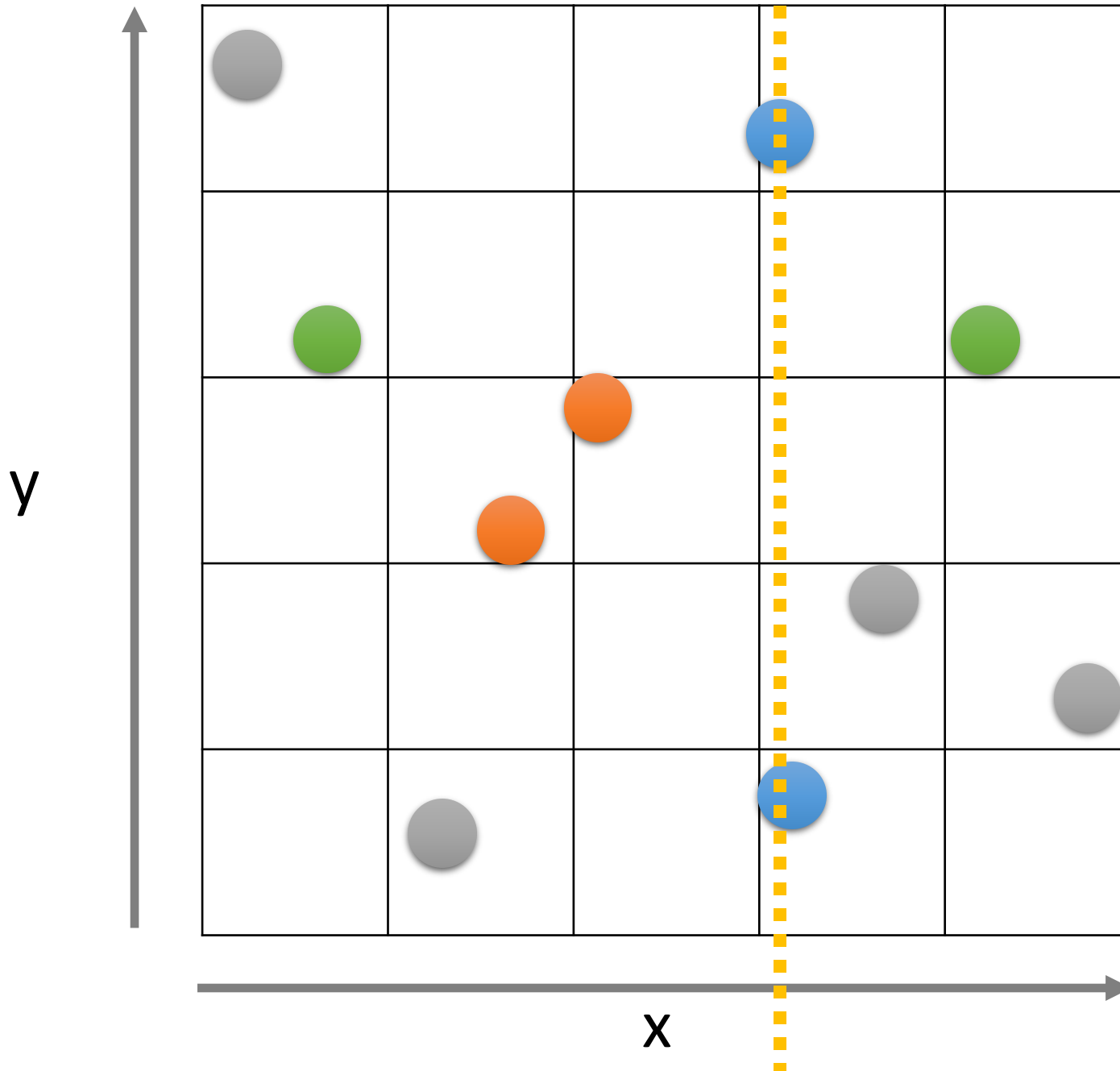
Divide-and-Conquer

1. **DIVIDE** into smaller subproblems
2. **CONQUER** (solve) the subproblems via recursive calls
3. **COMBINE** solutions from the subproblems

- How would you divide the problems for the closest pair problem?



1. Which two are closest on the y-axis?
2. Which two are closest on the x-axis?
3. Which two are closest?
4. How would you divide the search space?
(Give me a simple heuristic.)



1. Which two are closest on the y-axis?

2. Which two are closest on the x-axis?

3. Which two are closest?

4. How would you divide the search space?

This is the median x-value
This is not the average x-value

```
1. FUNCTION ClosestPair(px, py)
2.   n = px.length
3.   # What is the base case?
4.   IF n == 2
5.     RETURN px[0], px[1], dist(px[0], px[1])
6.
7.
8.
9.   # What are the recursive cases?
10.  pl, ql, dl = ClosestPair(left_px, left_py)
11.
12.
13.
14.  pr, qr, dr = ClosestPair(right_px, right_py)
```

```
1. FUNCTION FindClosestPair(points)
2.   points_x = copy_and_sort_by_x(points)
3.   points_y = copy_and_sort_by_y(points)
4.   RETURN ClosestPair(points_x, points_y)
```

How do we create these arrays?

```
1.  FUNCTION ClosestPair(px, py)
2.      n = px.length
3.      # What is the base case?
4.      IF n == 2
5.          RETURN px[0], px[1], dist(px[0], px[1])
6.
7.
8.
9.      # What are the recursive cases?
10.     pl, ql, dl = ClosestPair(left_px, left_py)
11.
12.
13.
14.     pr, qr, dr = ClosestPair(right_px, right_py)
```

```
1.  FUNCTION FindClosestPair(points)
2.      points_x = copy_and_sort_by_x(points)
3.      points_y = copy_and_sort_by_y(points)
4.      RETURN ClosestPair(points_x, points_y)
```

Draw the recursion tree!


```
1. FUNCTION ClosestPair(px, py)
2.   n = px.length
3.   # What is the base case?
4.   IF n == 2
5.     RETURN px[0], px[1], dist(px[0], px[1])
6.
7.
8.
9.   # What are the recursive cases?
10.  pl, ql, dl = ClosestPair(left_px, left_py)
11.
12.
13.
14.  pr, qr, dr = ClosestPair(right_px, right_py)
```

```
1. FUNCTION FindClosestPair(points)
2.   points_x = copy_and_sort_by_x(points)
3.   points_y = copy_and_sort_by_y(points)
4.   RETURN ClosestPair(points_x, points_y)
```

How do we create these arrays?

P : [p0(1,10), p1(2,8), p2(7,3), p3(5,7), p4(8,4), p5(3,5), p6(10,9), p7(9,1)]

Sorted by x coordinate

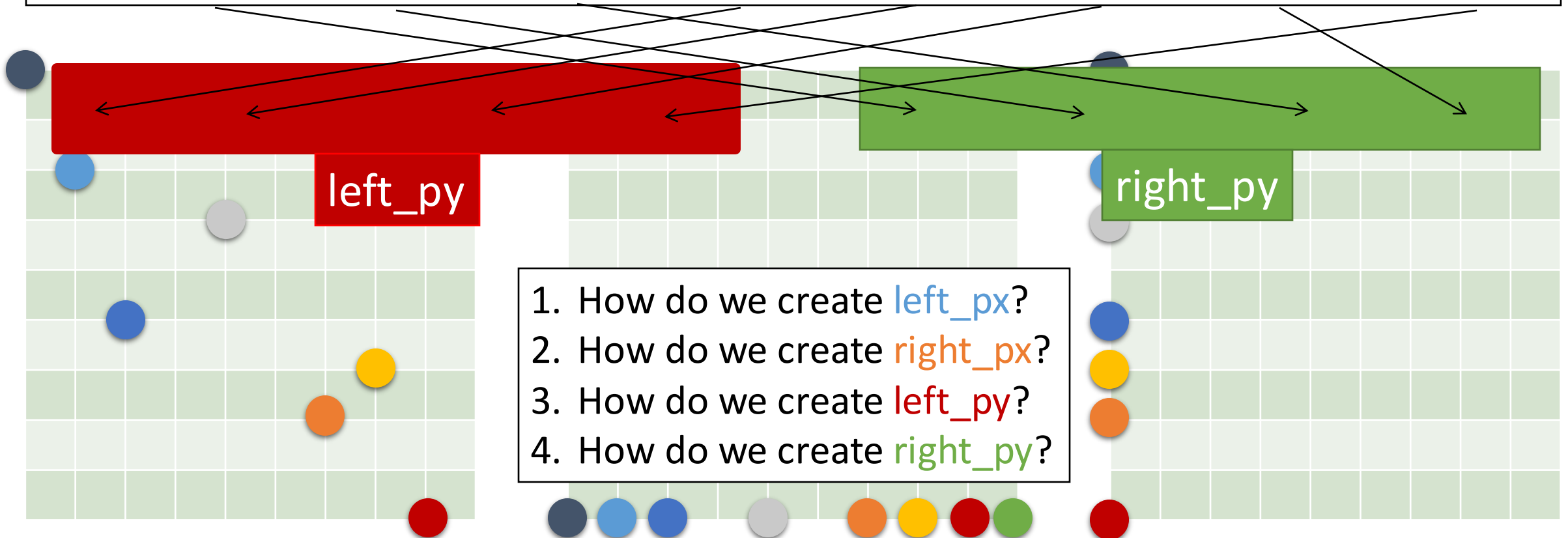
left_px

right_px

Px : [p0(1,10), p1(2,8), p5(3,5), p3(5,7), p2(7,3), p4(8,4), p7(9,1), p6(10,9)]

Sorted by y coordinate

Py : [p7(9,1), p2(7,3), p4(8,4), p5(3,5), p3(5,7), p1(2,8), p6(10,9), p0(1,10)]

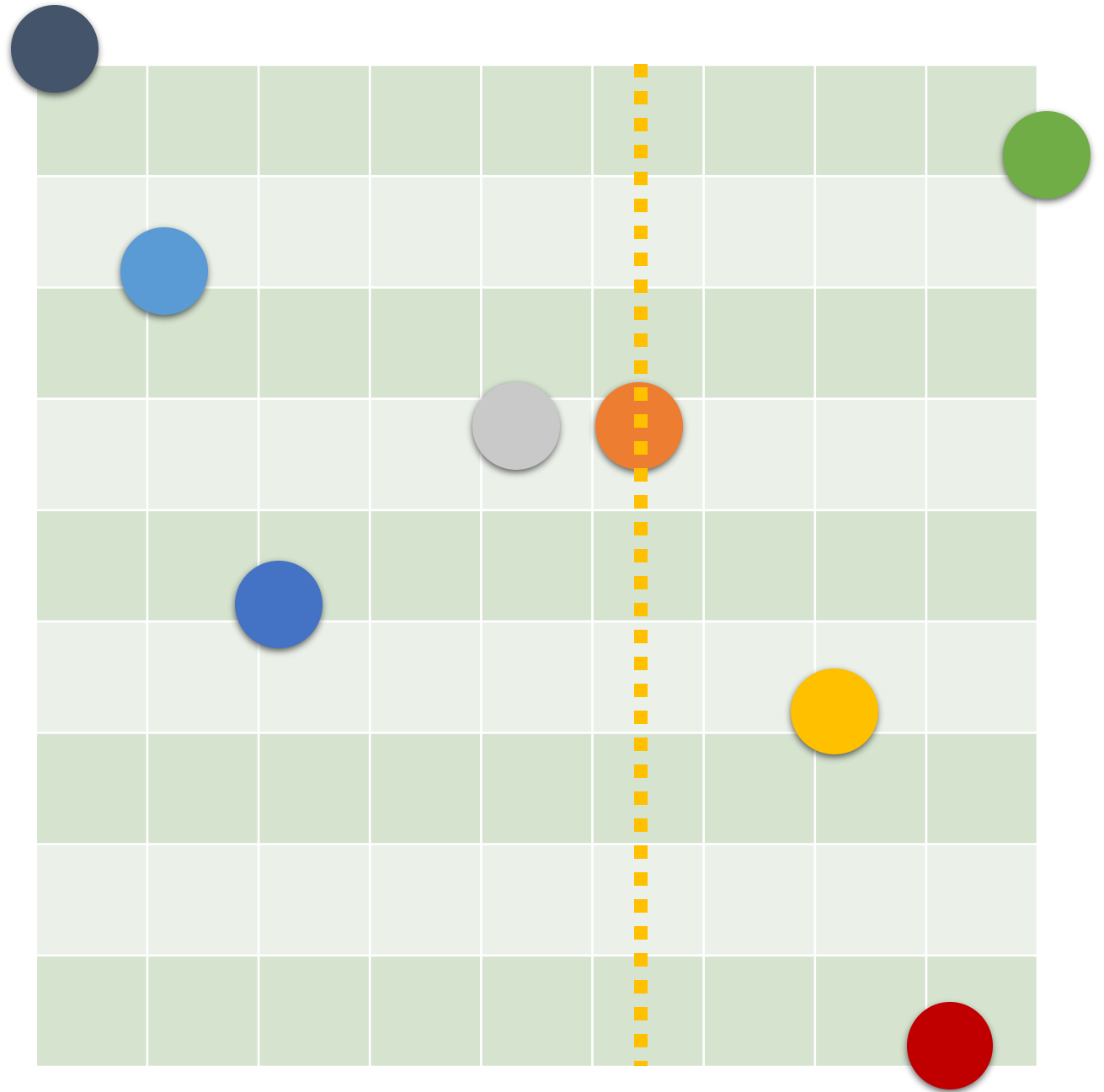


```
1.  FUNCTION ClosestPair(px, py)
2.      n = px.length
3.      IF n == 2
4.          RETURN px[0], px[1], dist(px[0], px[1])
5.
6.      left_px = px[0 ..< n//2]
7.      left_py = [p FOR p IN py IF p.x < px[n//2].x]
8.      pl, ql, dl = ClosestPair(left_px, left_py)
9.
10.     right_px = px[n//2 ..< n]
11.     right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]
12.     pr, qr, dr = ClosestPair(right_px, right_py)
```

Median x value

What is the running time of these operations?

Any problems
with our current
approach?



```
1.  FUNCTION ClosestPair(px, py)
2.      n = px.length
3.      IF n == 2
4.          RETURN px[0], px[1], dist(px[0], px[1])
5.
6.      left_px = px[0 ..< n//2]
7.      left_py = [p FOR p IN py IF p.x < px[n//2].x]
8.      pl, ql, dl = ClosestPair(left_px, left_py)
9.
10.     right_px = px[n//2 ..< n]
11.     right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]
12.     pr, qr, dr = ClosestPair(right_px, right_py)
13.
14.     d = min(dl, dr)
15.     ps, qs, ds = ClosestSplitPair(px, py, d)
16.
17.     RETURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)
```

What time complexity does this process need such that the overall algorithm runs in $O(n \lg n)$?
Hint: think about Merge Sort.

Exercise Question 1

1. What must be the running time of `ClosestSplitPair` if the `ClosestPair` algorithm is to have a running time of $O(n \lg n)$?

```
FUNCTION ClosestPair(px, py)
    n = px.length
    IF n == 2
        RETURN px[0], px[1], dist(px[0], px[1])

    left_px = px[0 ..< n//2]
    left_py = [p FOR p IN py IF p.x < px[n//2].x]
    pl, ql, dl = ClosestPair(left_px, left_py)

    right_px = px[n//2 ..< n]
    right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]
    pr, qr, dr = ClosestPair(right_px, right_py)

    d = min(dl, dr)
    ps, qs, ds = ClosestSplitPair(px, py, d)

    RETURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)
```

Merge Sort and Its Recurrence

```
FUNCTION RecursiveFunction(some_input)
  IF base_case:
    # Usually O(1)
    RETURN base_case_work(some_input)

  # Two recursive calls, each with half the data
  one = RecursiveFunction(some_input.first_half)
  two = RecursiveFunction(some_input.second_half)

  # Combine results from recursive calls (usually O(n))
  one_and_two = Combine(one, two)

RETURN one_and_two
```



```
1.  FUNCTION ClosestPair(px, py)
2.      n = px.length
3.      IF n == 2
4.          RETURN px[0], px[1], dist(px[0], px[1])
5.
6.      left_px = px[0 ..< n//2]
7.      left_py = [p FOR p IN py IF p.x < px[n//2].x]
8.      pl, ql, dl = ClosestPair(left_px, left_py)
9.
10.     right_px = px[n//2 ..< n]
11.     right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]
12.     pr, qr, dr = ClosestPair(right_px, right_py)
13.
14.     d = min(dl, dr)
15.     ps, qs, ds = ClosestSplitPair(px, py, d)
16.
17.     RETURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)
```

How do we find the
closest pair that splits the
two sides?

$T(n)$ **FUNCTION** ClosestPair(px, py)

$O(1)$ = px.length

$O(1)$ n == 2

$O(1)$ **RETURN** px[0], px[1], dist(px[0], px[1])

$O(n)$ left_px = px[0 ..< n//2]

$O(n)$ left_py = [p **FOR** p **IN** py **IF** p.x < px[n//2].x]

$T(n/2)$ ql, dl = ClosestPair(left_px, left_py)

$O(n)$ right_px = px[n//2 ..< n]

$O(n)$ right_py = [p **FOR** p **IN** py **IF** p.x ≥ px[n//2].x]

$T(n/2)$ qr, dr = ClosestPair(right_px, right_py)

$O(1)$ = min(dl, dr)

$O(n)$, qs, ds = ClosestSplitPair(px, py, d)

$O(1)$ **RETURN** Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)

$$\begin{aligned} T(n) &= 2 T(n/2) + O(n) \\ &= O(n \lg n) \end{aligned}$$

T(n)

FUNCTION MergeSort(array)

O(1) = array.length

O(1) n == 1

O(1) **RETURN** array

T(n/2) left_sorted = MergeSort(array[0 ..< n//2])

T(n/2) right_sorted = MergeSort(array[n//2 ..< n])

O(n) array_sorted = Merge(left_sorted, right_sorted)

O(1) **RETURN** array_sorted

$$\begin{aligned} T(n) &= 2 T(n/2) + O(n) \\ &= O(n \lg n) \end{aligned}$$

$$\begin{aligned} T(n) &= 2 T(n/2) + O(n) \\ &= O(n \lg n) \end{aligned}$$

T(n) **FUNCTION** RecursiveFunction(some_input)

O(1) **base_case:**

Usually O(1)

O(1) **RETURN** base_case_work(some_input)

Two recursive calls, each with half the data

T(n/2) one = RecursiveFunction(some_input.first_half)

T(n/2) two = RecursiveFunction(some_input.second_half)

Combine results from recursive calls (usually O(n))

O(n) one_and_two = Combine(one, two)

O(1) **RETURN** one_and_two

Key Idea

- **In ClosestSplitPair we only need to check for pairs that are closer than those found in the recursive calls to ClosestPair**
- This is easier (**faster**) than trying to find the closest split pair without any extra information!

$$d = \min[d(p_l, q_l), d(p_r, q_r)]$$

```

FUNCTION ClosestSplitPair(px, py, d)
    n = px.length
    x_median = px[n//2].x
    middle_py = [p FOR p IN py IF x_median - d < p.x < x_median + d]

    closest_d = INFINITY, closest_p = closest_q = NONE
    FOR i IN [0 ..< middle_py.length - 1]
        FOR j IN [1 ..= min(7, middle_py.length - i)]
            p = middle_py[i], q = middle_py[i + j]
            IF dist(p, q) < closest_d
                closest_d = dist(p, q)
                closest_p = p, closest_q = q

    RETURN closest_p, closest_q, closest_d

```

At most 6 points vertically “between” the two closest points.

Exercise Question 2

2. What is the running time of the nested for-loop (looping over j)?

```
FUNCTION ClosestSplitPair(px, py, d)
    n = px.length
    x_median = px[n//2].x
    middle_py = [p FOR p IN py IF x_median - d < p.x < x_median + d]

    closest_d = INFINITY, closest_p = closest_q = NONE
    FOR i IN [0 ..< middle_py.length - 1]
        FOR j IN [1 ..= min(7, middle_py.length - i)]
            p = middle_py[i], q = middle_py[i + j]
            IF dist(p, q) < closest_d
                closest_d = dist(p, q)
                closest_p = p, closest_q = q

    RETURN closest_p, closest_q, closest_d
```

Loop Unrolling

```
FOR j IN [1 ..= min(7, middle_py.length - i)]  
    p = middle_py[i], q = middle_py[i + j]  
    IF dist(p, q) < closest_d  
        closest_d = dist(p, q)  
        closest_p = p, closest_q = q
```

```
IF dist(middle_py[i], middle_py[i + 1]) < closest_d  
    closest_d = dist(middle_py[i], middle_py[i + 1])  
    closest_p = middle_py[i]  
    closest_q = middle_py[i + 1]
```

```
IF dist(middle_py[i], middle_py[i + 2]) < closest_d  
    closest_d = dist(middle_py[i], middle_py[i + 2])  
    closest_p = middle_py[i]  
    closest_q = middle_py[i + 2]
```

...

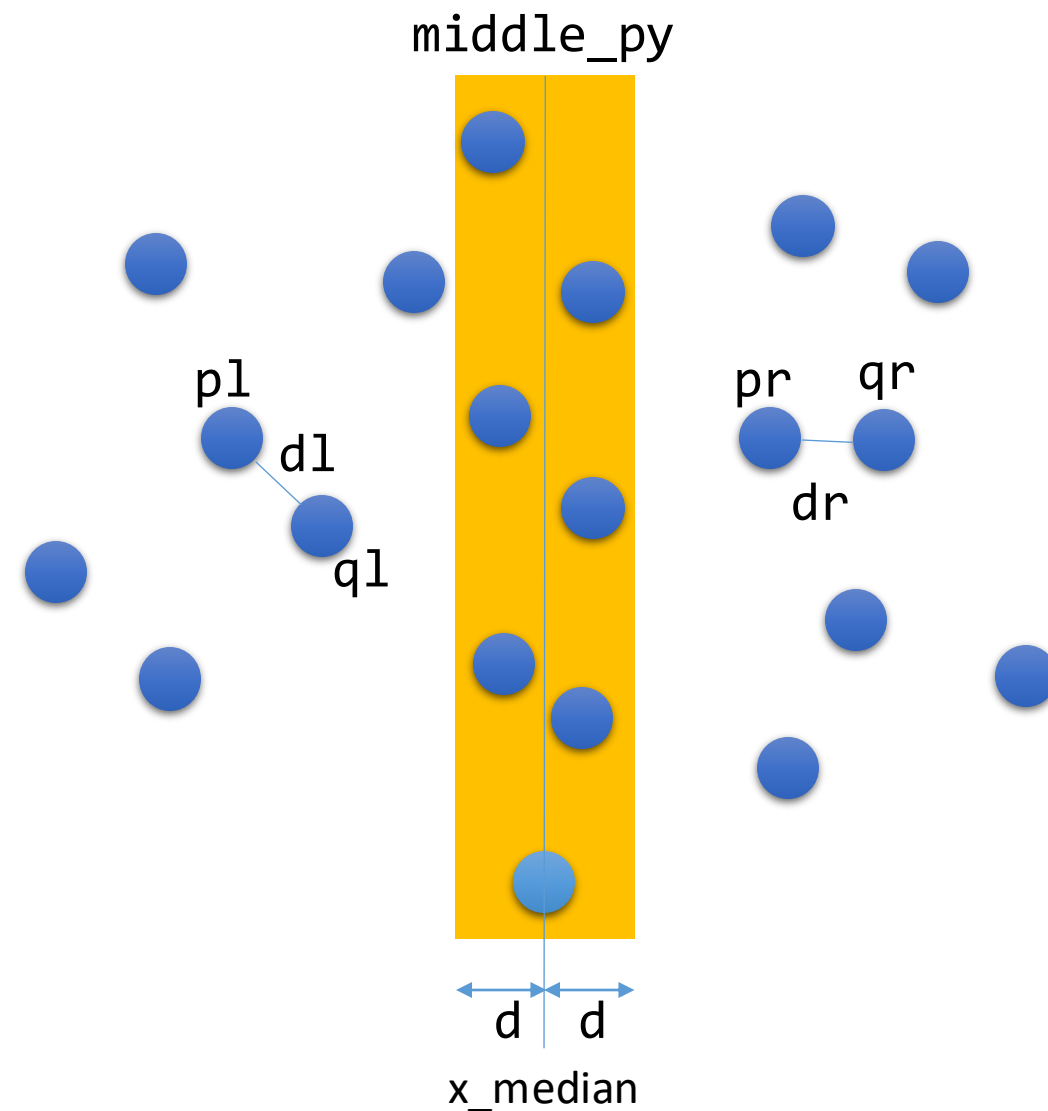

```

FUNCTION ClosestSplitPair(px, py, d)
  n = px.length
  x_median = px[n//2].x
  middle_py = [p FOR p IN py
               IF x_median - d < p.x < x_median + d]

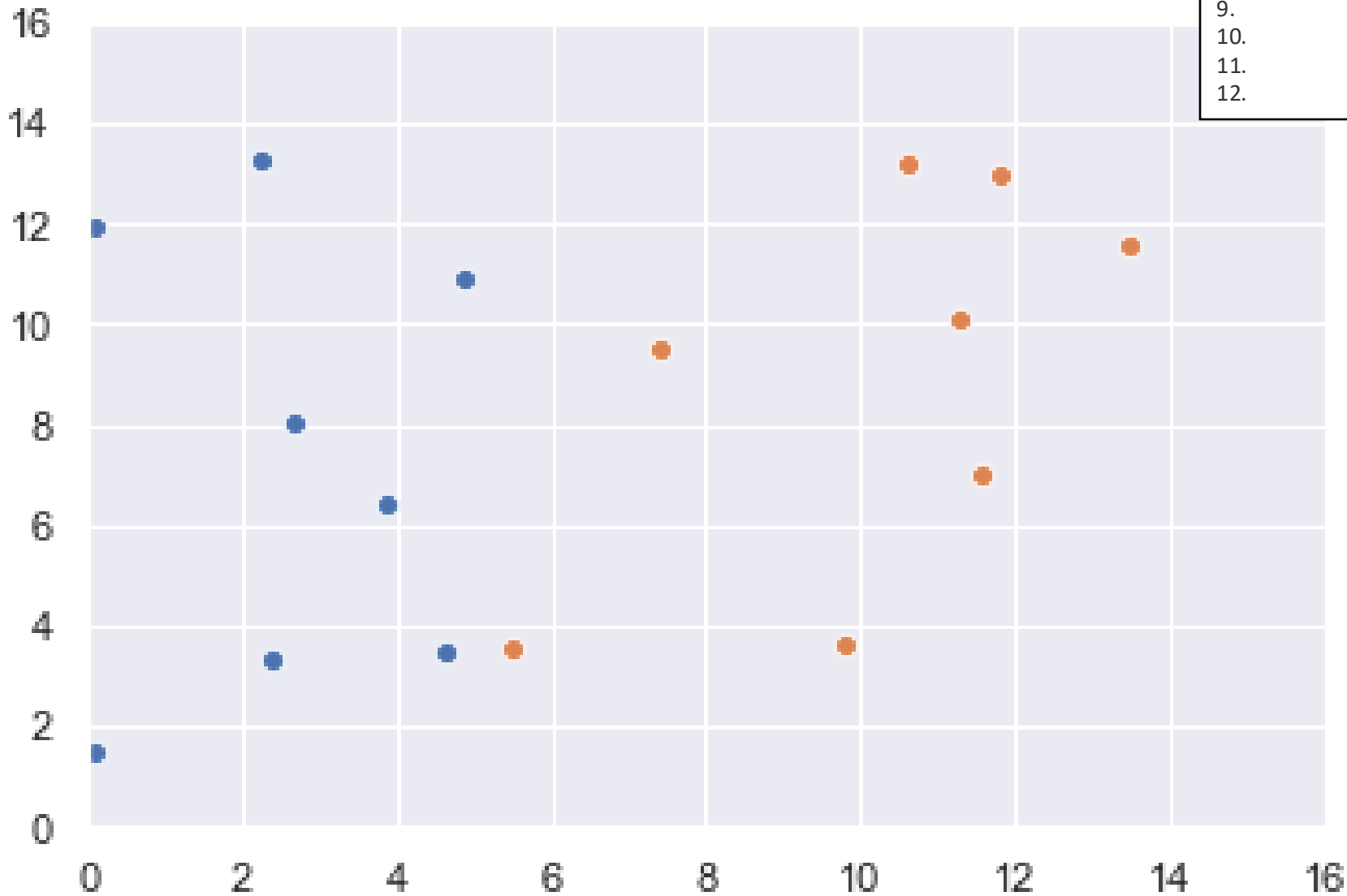
  closest_d = INFINITY, closest_p = closest_q = NONE
  FOR i IN [0 ..< middle_py.length - 1]
    FOR j IN [1 ..= min(7, middle_py.length - i)]
      p = middle_py[i], q = middle_py[i + j]
      IF dist(p, q) < closest_d
        closest_d = dist(p, q)
        closest_p = p, closest_q = q

  RETURN closest_p, closest_q, closest_d

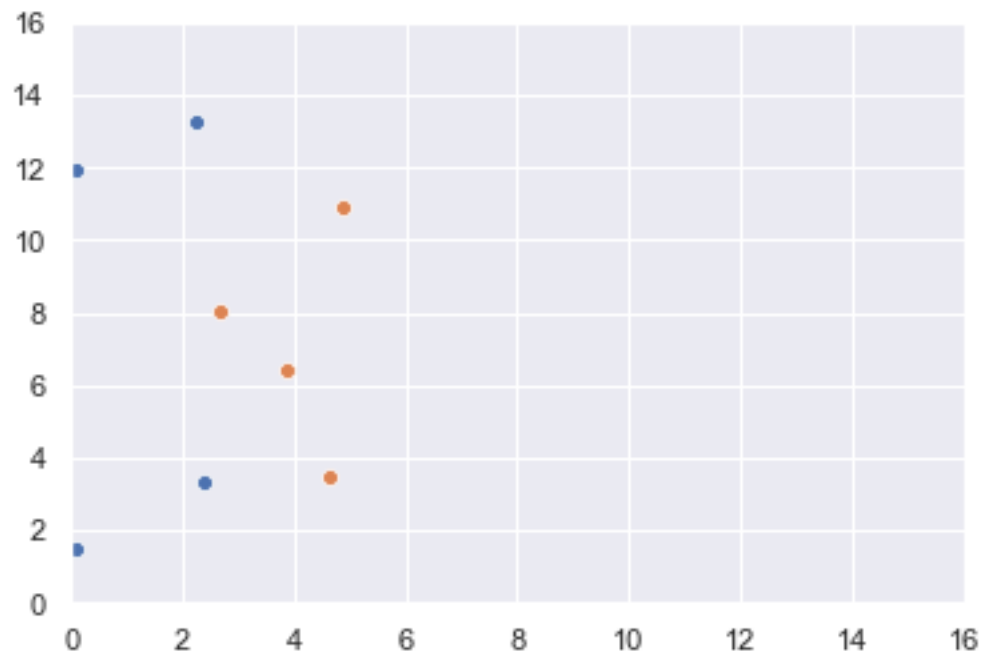
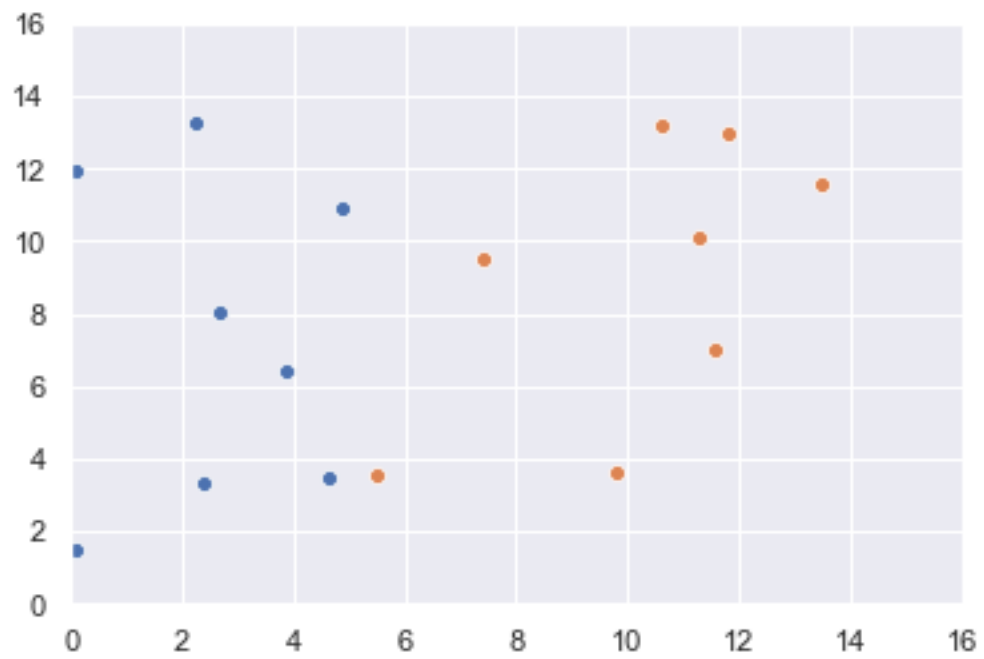
```

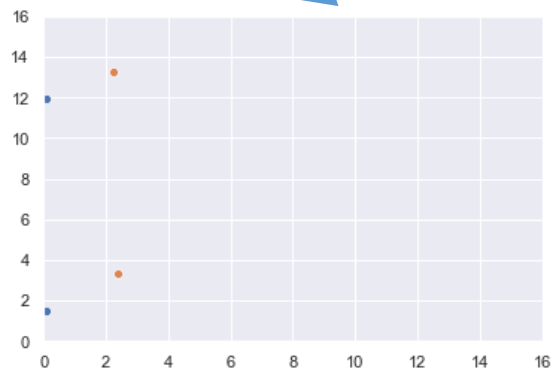
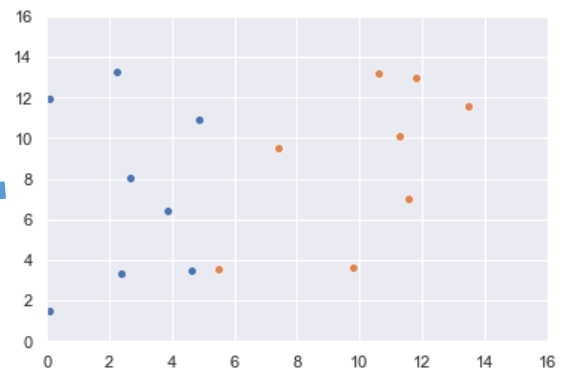
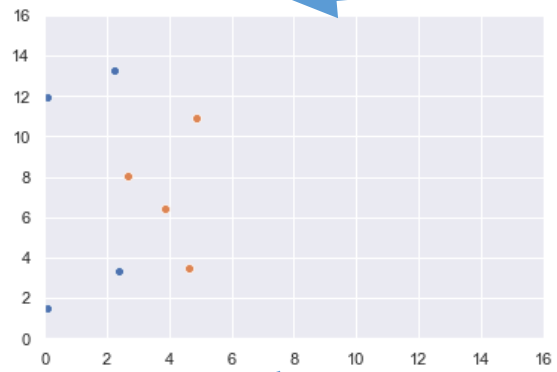


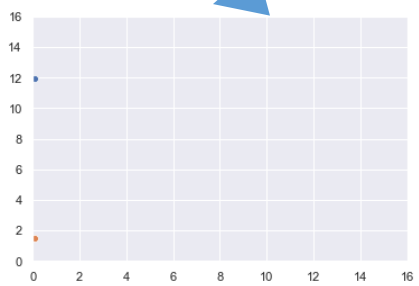
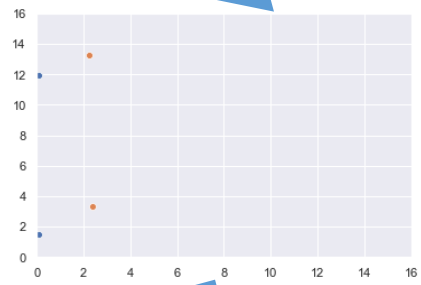
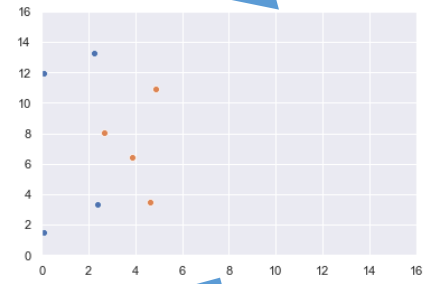
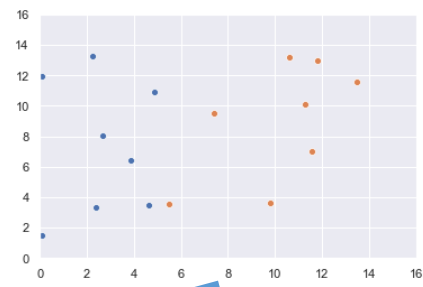
Example execution.

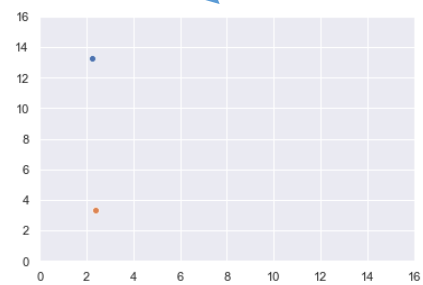
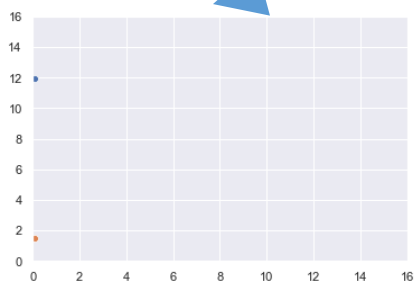
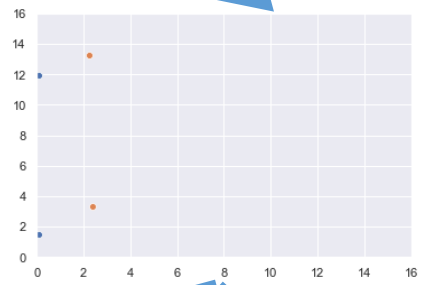
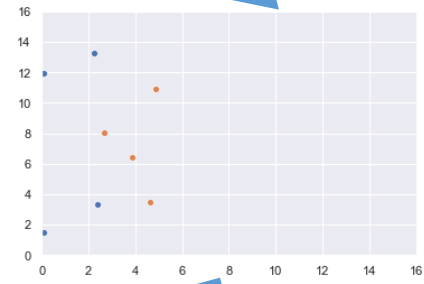
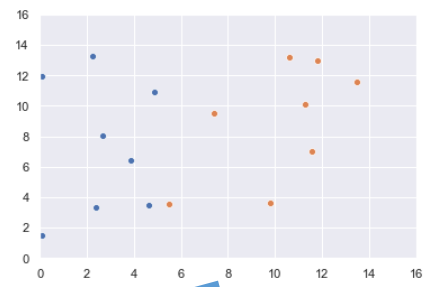


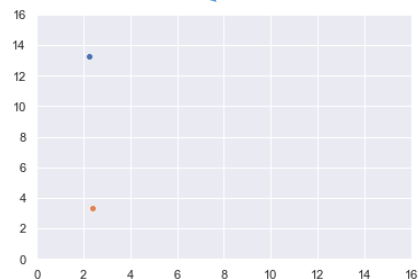
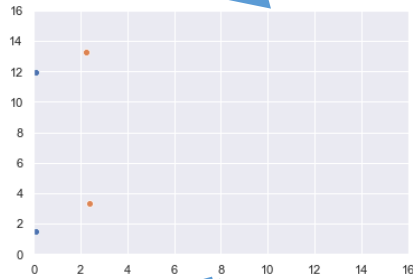
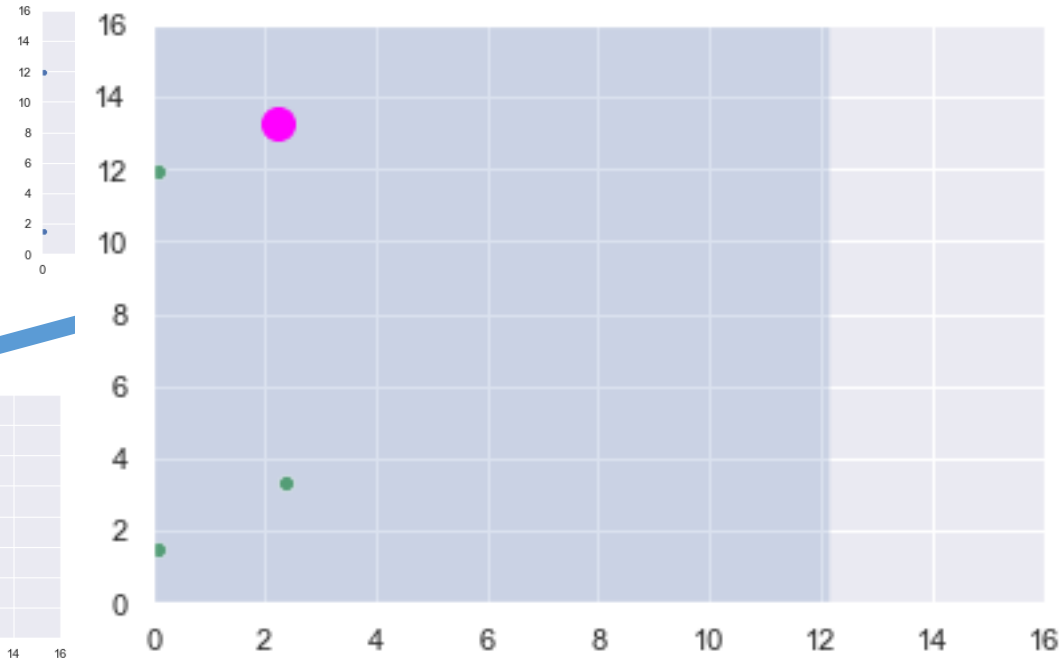
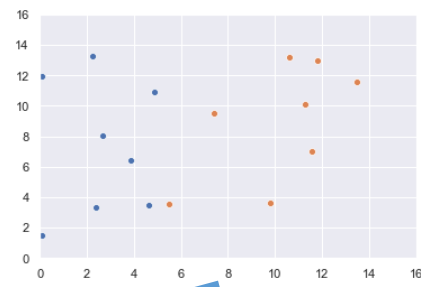
```
1.  FUNCTION ClosestPair(px, py)
2.      n = px.length
3.      IF n == 2
4.          RETURN px[0], px[1], dist(px[0], px[1])
5.
6.      left_px = px[0 ..< n//2]
7.      left_py = [p FOR p IN py IF p.x < px[n//2].x]
8.      pl, ql, dl = ClosestPair(left_px, left_py)
9.
10.     right_px = px[n//2 ..< n]
11.     right_py = [p FOR p IN py IF p.x ≥ px[n//2].x]
12.     pr, qr, dr = ClosestPair(right_px, right_py)
```



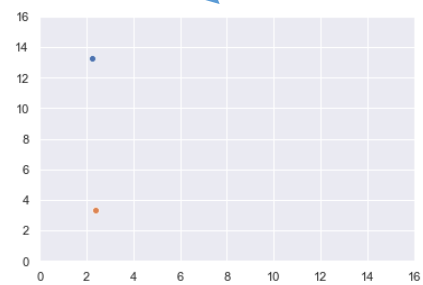
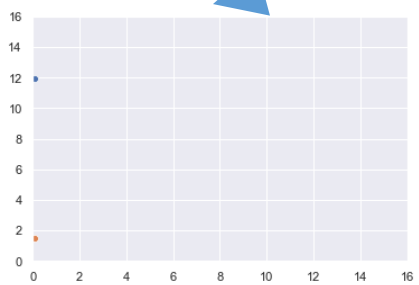
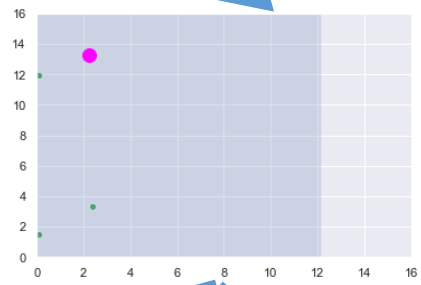
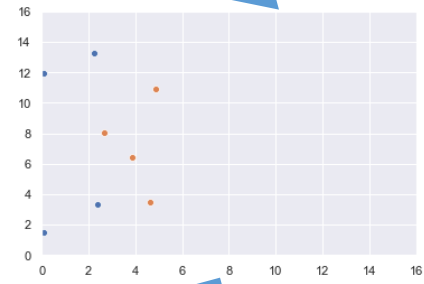
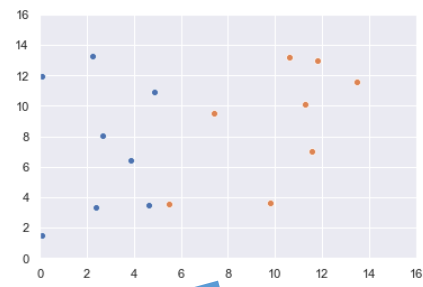


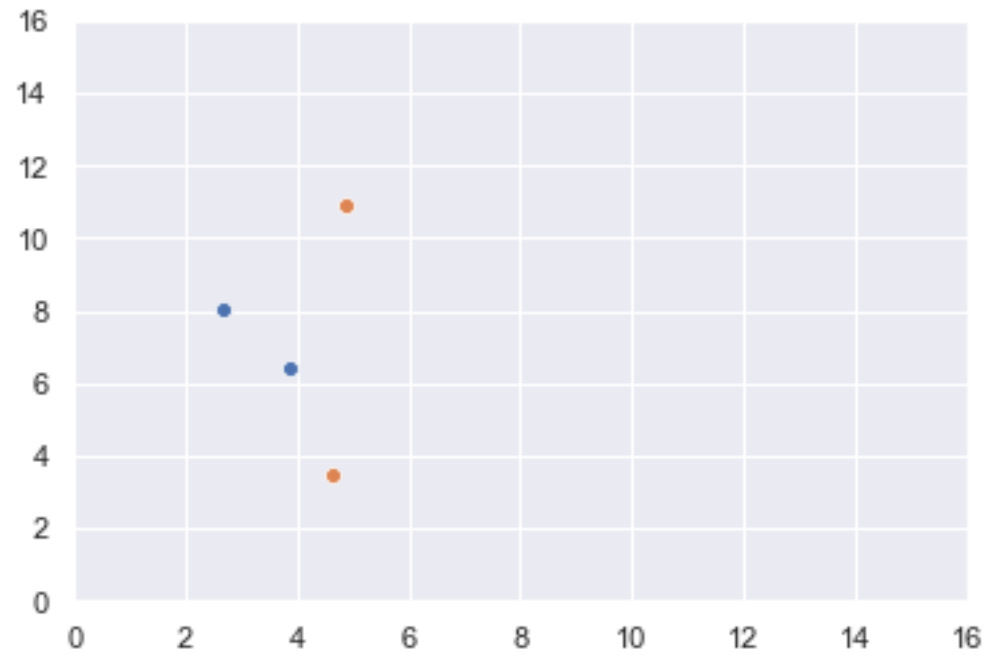
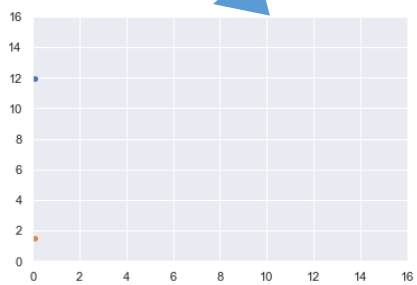
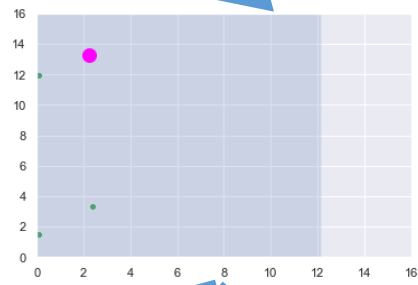
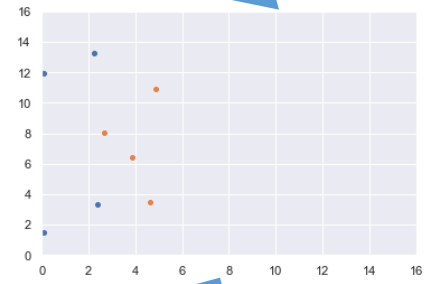
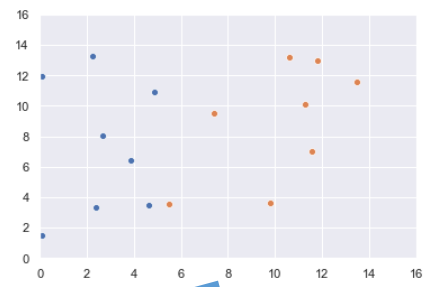


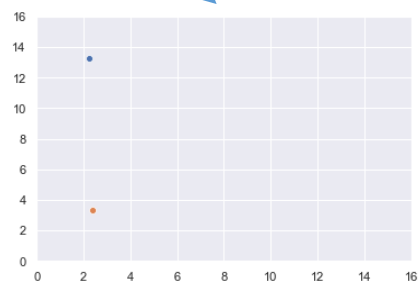
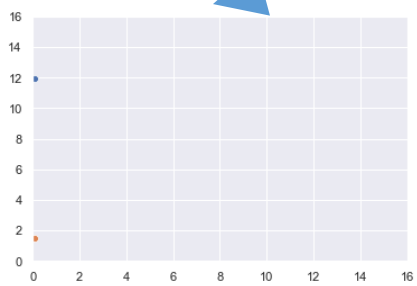
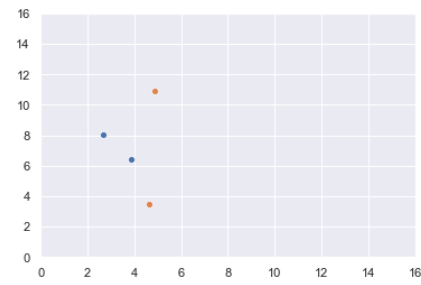
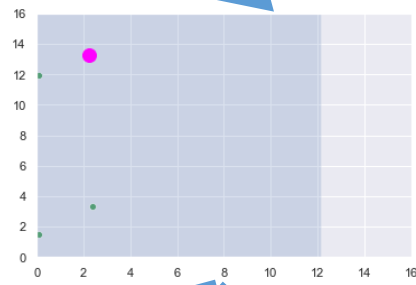
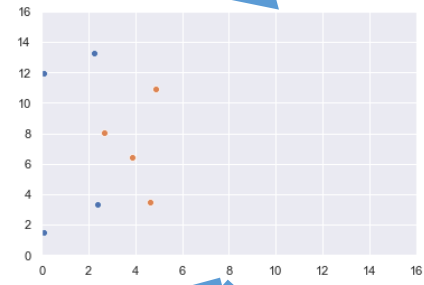
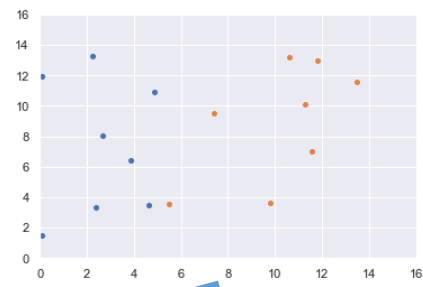


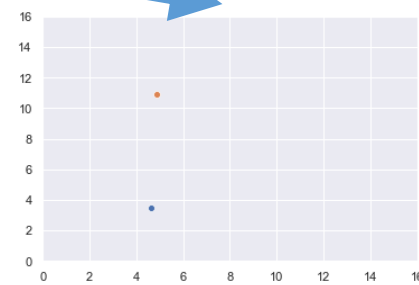
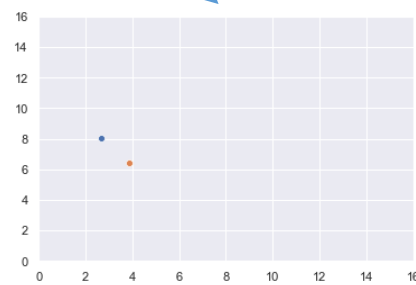
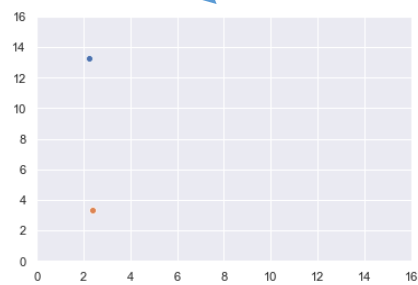
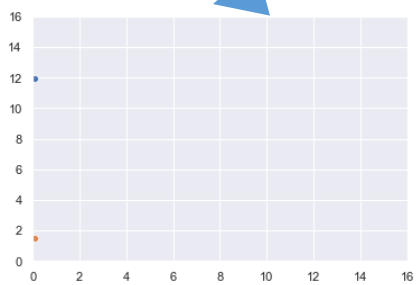
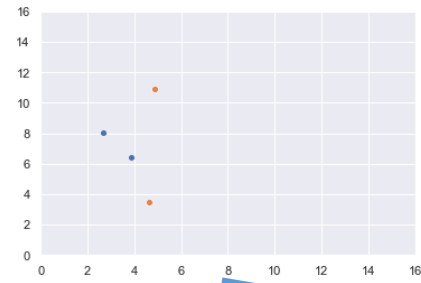
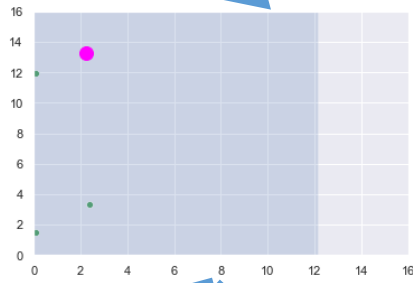
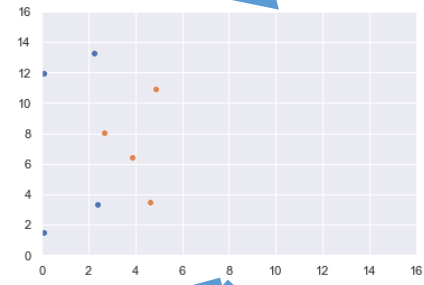
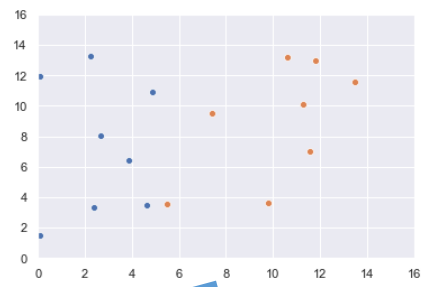


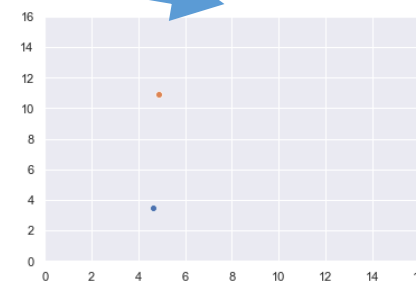
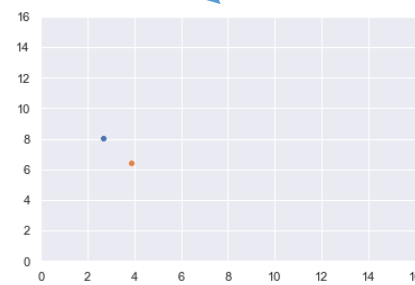
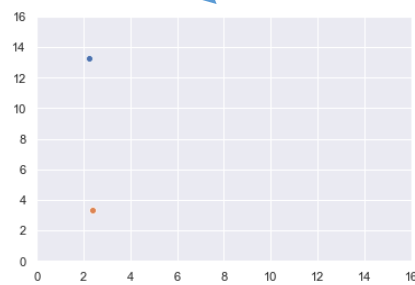
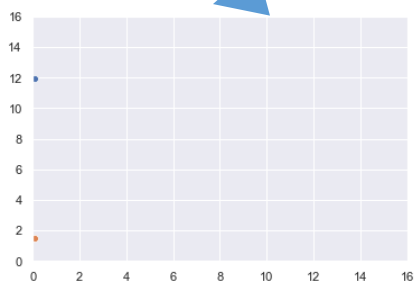
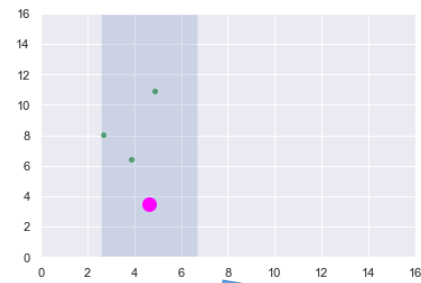
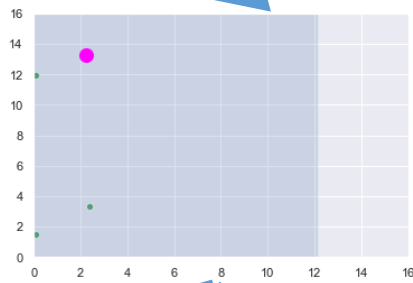
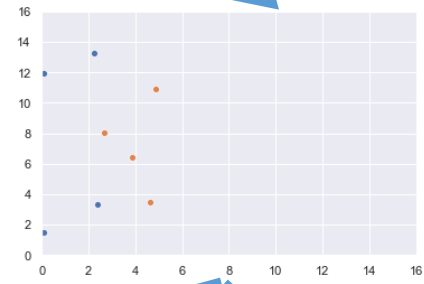
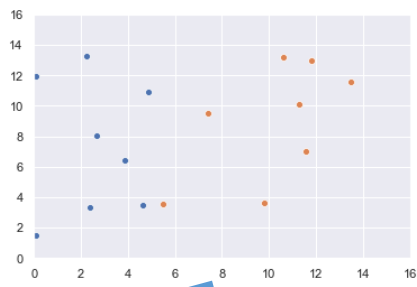
Closest Split Pair

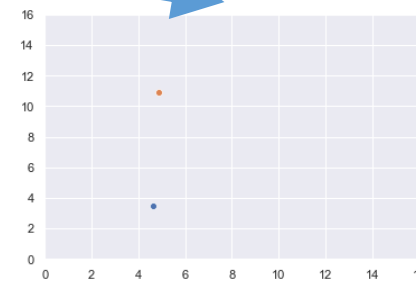
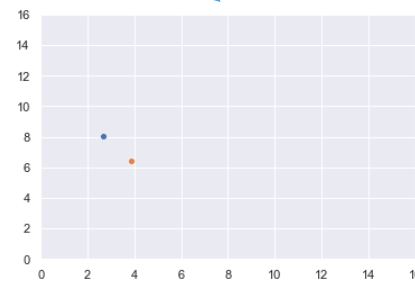
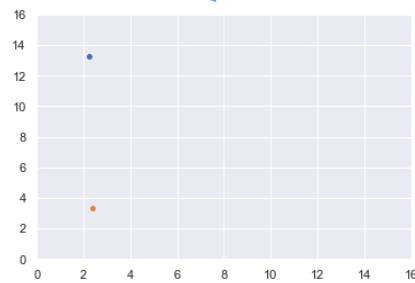
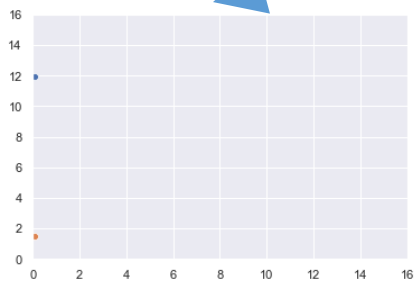
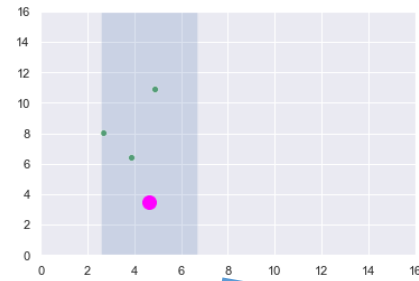
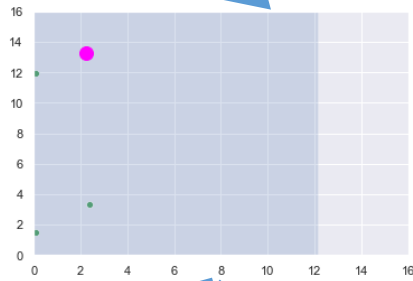
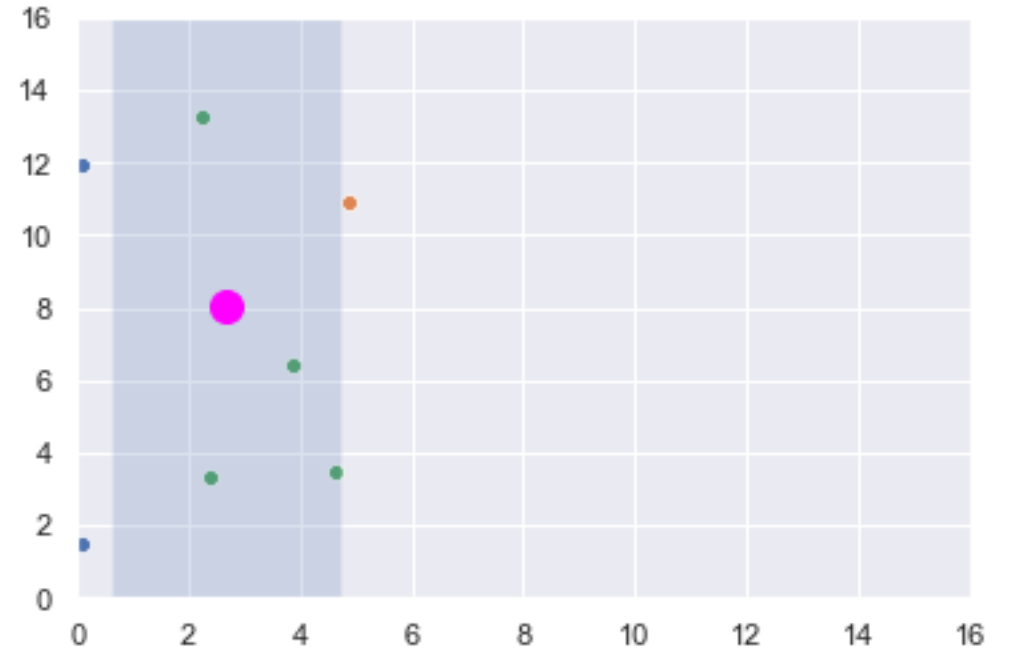
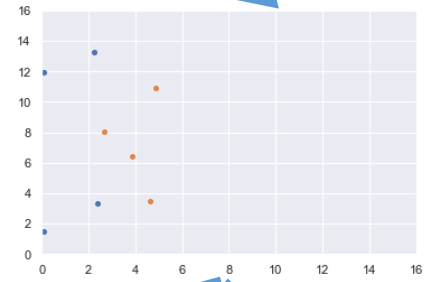
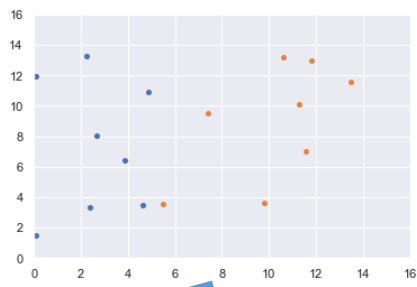


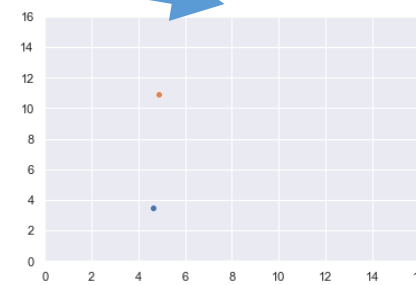
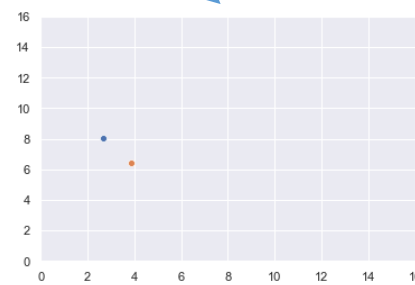
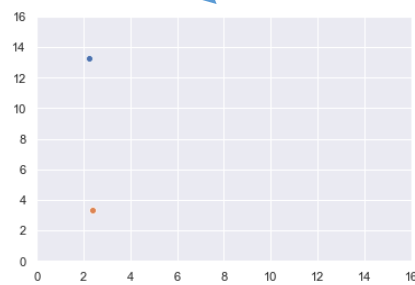
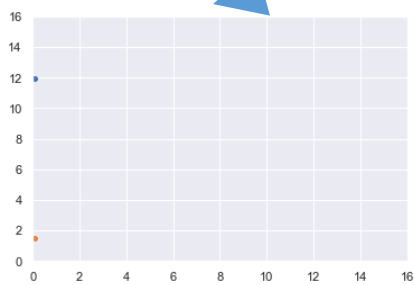
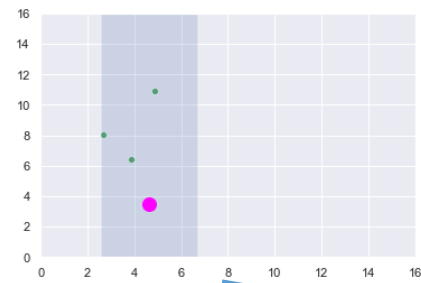
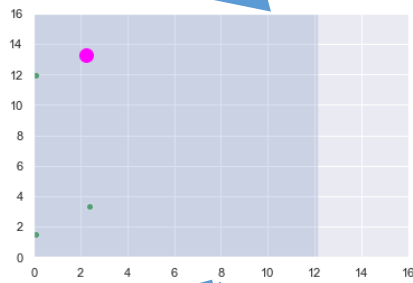
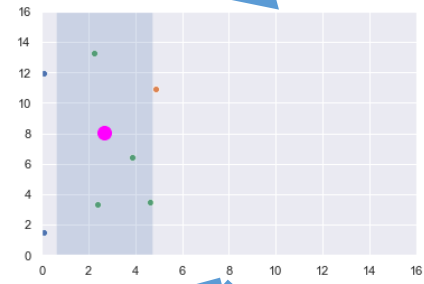
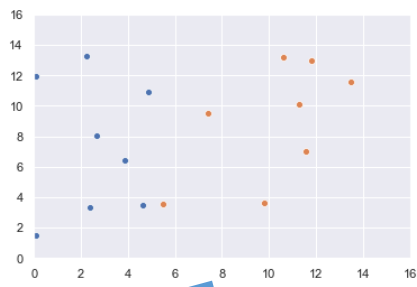


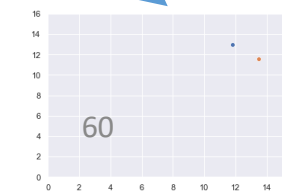
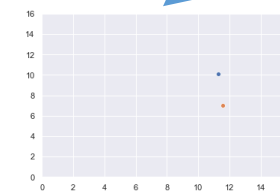
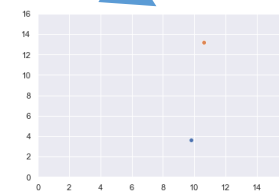
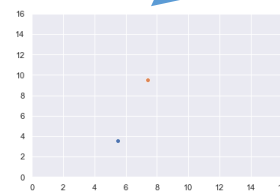
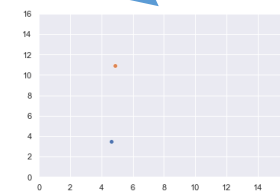
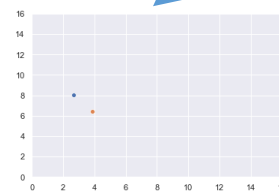
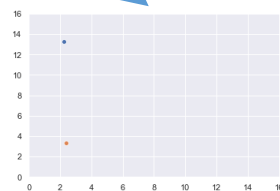
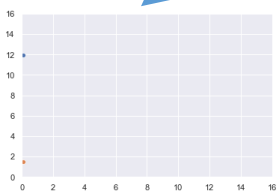
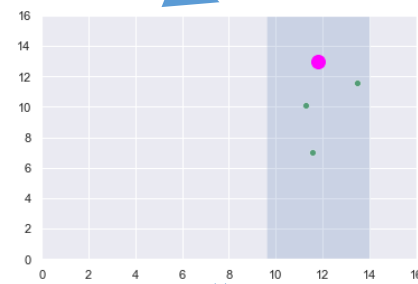
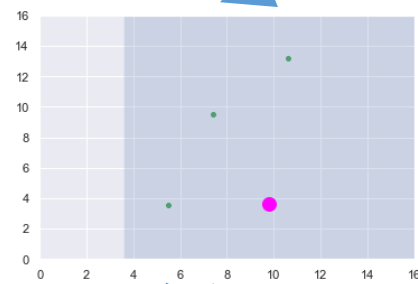
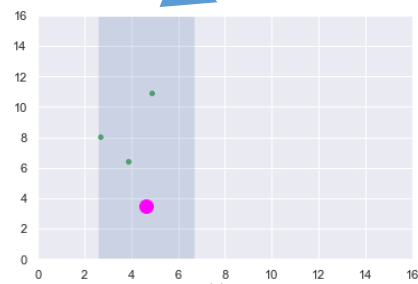
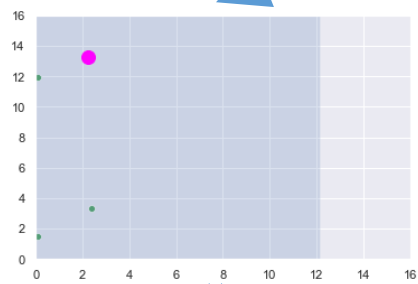
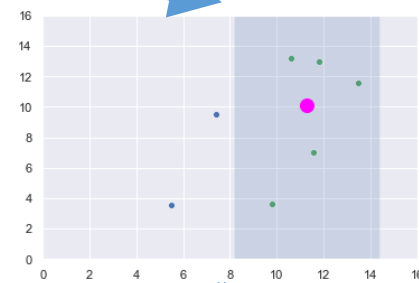
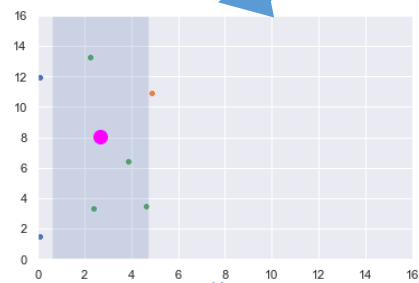
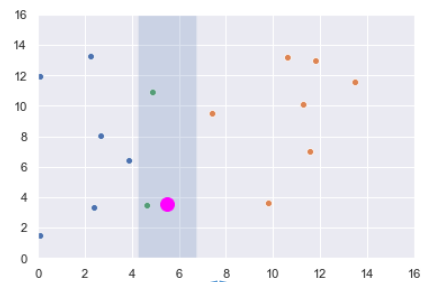












Theorem for correctness of ClosestPair

Theorem:

Provided a set of n points called P , the **ClosestPair** algorithm find the closest pair of points according to their pairwise Euclidean distances.

Input: P , a set of n points that lie in a (two-dimensional) plane

Output: a pair of points (p, q) that are the “closest”

Distance is measured using Euclidean distance:

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Assumptions: None

FUNCTION ClosestPair(px, py)

n = px.length

IF n == 2

RETURN px[0], px[1], dist(px[0], px[1])

left_px = px[0 ..< n//2]

left_py = [p **FOR** p **IN** py **IF** p.x < px[n//2].x]

pl, ql, dl = ClosestPair(left_px, left_py)

right_px = px[n//2 ..< n]

right_py = [p **FOR** p **IN** py **IF** p.x ≥ px[n//2].x]

pr, qr, dr = ClosestPair(right_px, right_py)

d = min(dl, dr)

ps, qs, ds = ClosestSplitPair(px, py, d)

RETURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)

FUNCTION ClosestPair(px, py)

n = px.length

IF n == 2

RETURN px[0], px[1], dist(px[0], px[1])

left_px = px[0 ..< n//2]

left_py = [p **FOR** p **IN** py **IF** p.x < px[n//2].x]

pl, ql, dl = ClosestPair(left_px, left_py)

right_px = px[n//2 ..< n]

right_py = [p **FOR** p **IN** py **IF** p.x ≥ px[n//2].x]

pr, qr, dr = ClosestPair(right_px, right_py)

d = min(dl, dr)

ps, qs, ds = ClosestSplitPair(px, py, d)

RETURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)

ClosestPair finds the closest pair

Let $p \in \text{left}$, $q \in \text{right}$ be a split pair with $d(p, q) < d$

Then

- A. p and $q \in \text{middle_py}$, and
- B. p and q are at most 7 positions apart in middle_py

If the claim is true:

Corollary 1: If the closest pair of P is in a split pair, then our **ClosestSplitPair** procedure finds it.

Corollary 2: **ClosestPair** is correct and runs in $O(n \lg n)$ since it has the same recursion tree as merge sort

Proof—Part A

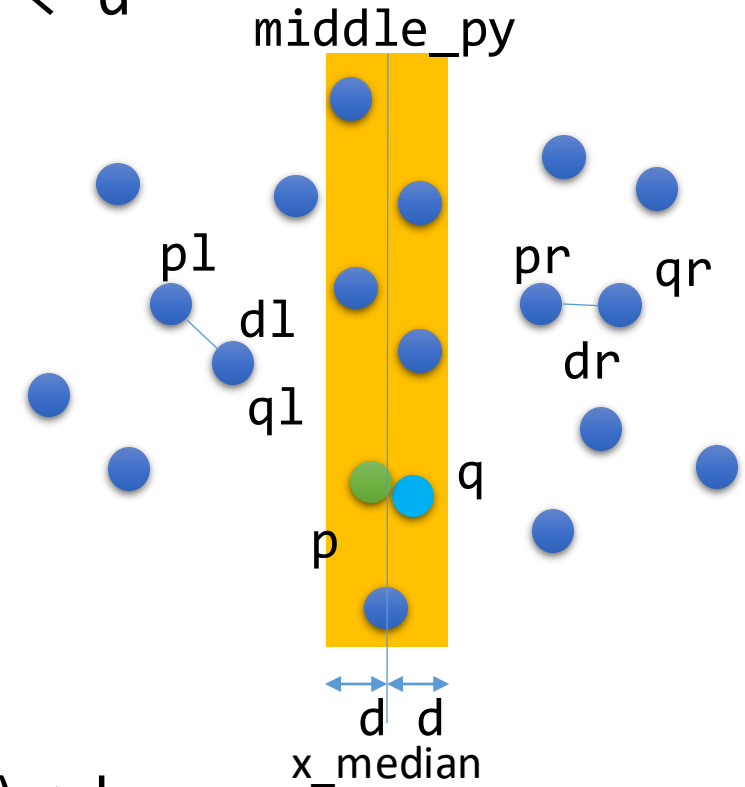
Let $p \in \text{left}$, $q \in \text{right}$ be a split pair with $d(p, q) < d$
Then

A. p and $q \in \text{middle_py}$, and

If $p = (x_1, y_1) \in \text{left}$ AND $q = (x_2, y_2) \in \text{right}$ AND $d(p, q) < d$
Then

$$\begin{aligned} x_{\text{median}} - d &< x_1 \leq x_{\text{median}} \quad \textbf{and} \\ x_{\text{median}} &\leq x_2 < x_{\text{median}} + d \end{aligned}$$

Otherwise, p and q would not be the closest pair with $d(p, q) < d$



Proof—Part A

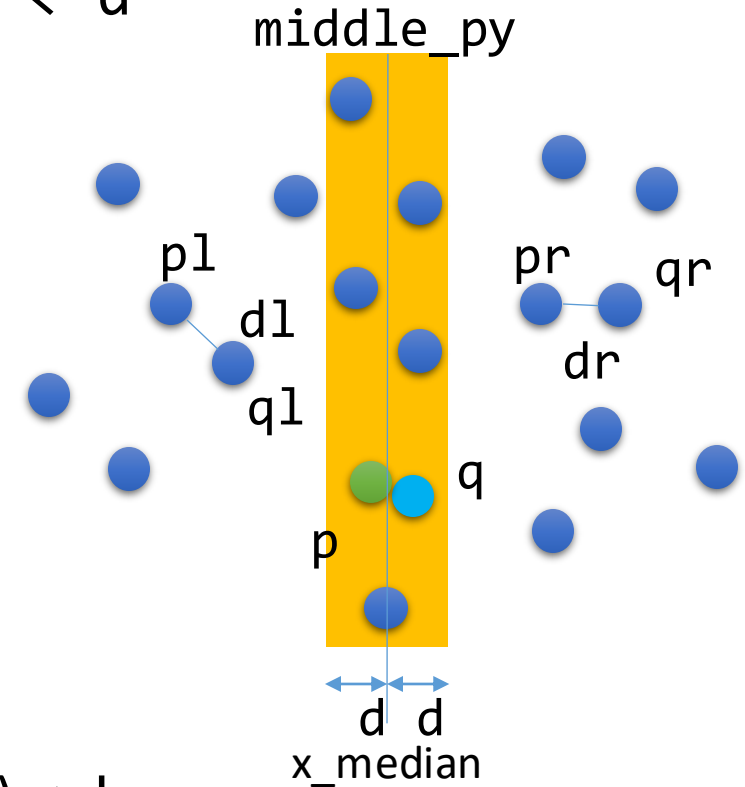
Let $p \in \text{left}$, $q \in \text{right}$ be a split pair with $d(p, q) < d$
Then

A. p and $q \in \text{middle_py}$, and

If $p = (x_1, y_1) \in \text{left}$ AND $q = (x_2, y_2) \in \text{right}$ AND $d(p, q) < d$
Then

$$\begin{aligned} x_{\text{median}} - d < x_1 &\leq x_{\text{median}} \quad \textbf{and} \\ x_{\text{median}} &\leq x_2 < x_{\text{median}} + d \end{aligned}$$

Otherwise, p and q would not be the closest pair with $d(p, q) < d$



ClosestPair finds the closest pair

Let $p \in \text{left}$, $q \in \text{right}$ be a split pair with $d(p, q) < d$

Then

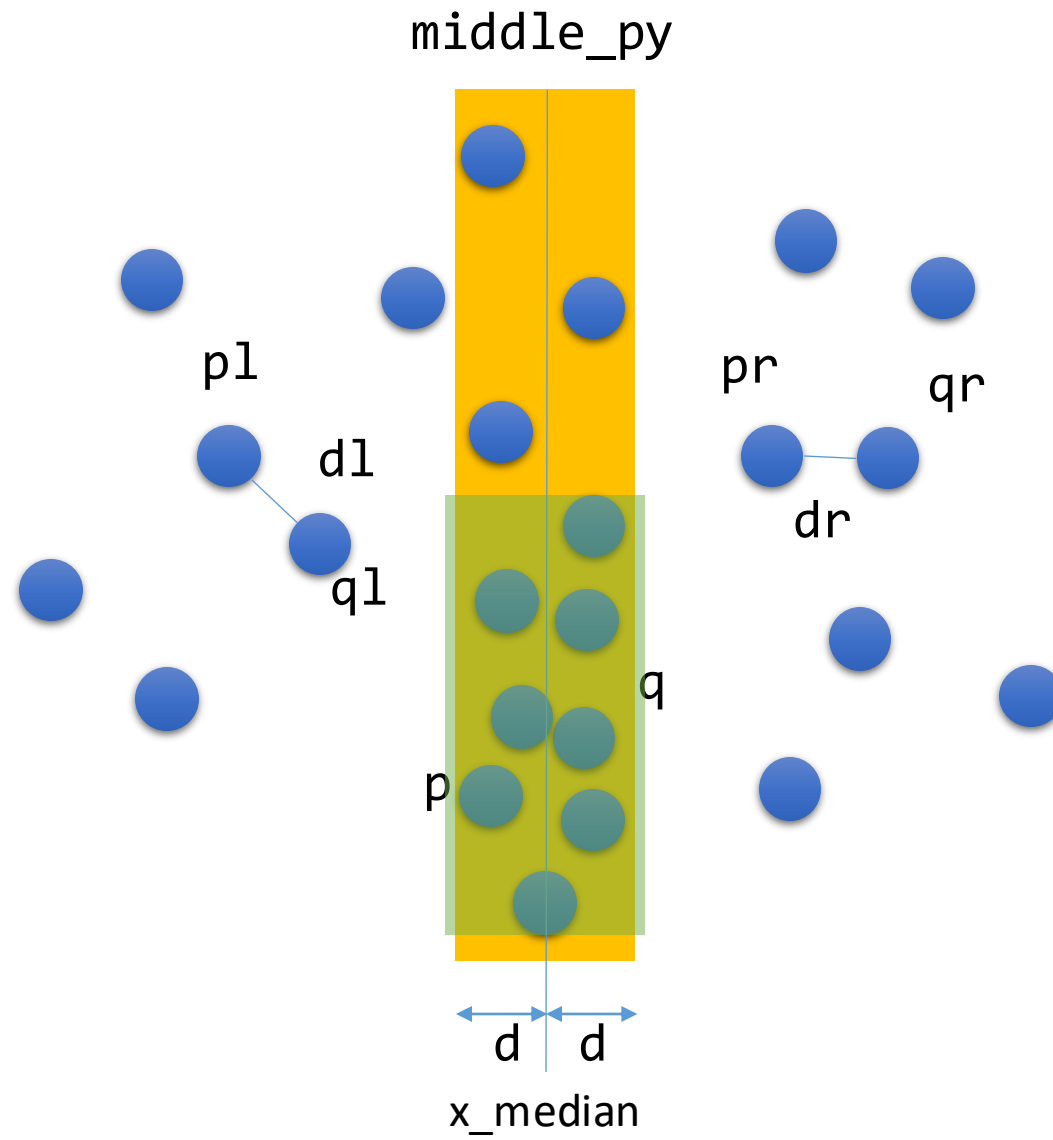
A. p and $q \in \text{middle py}$, and

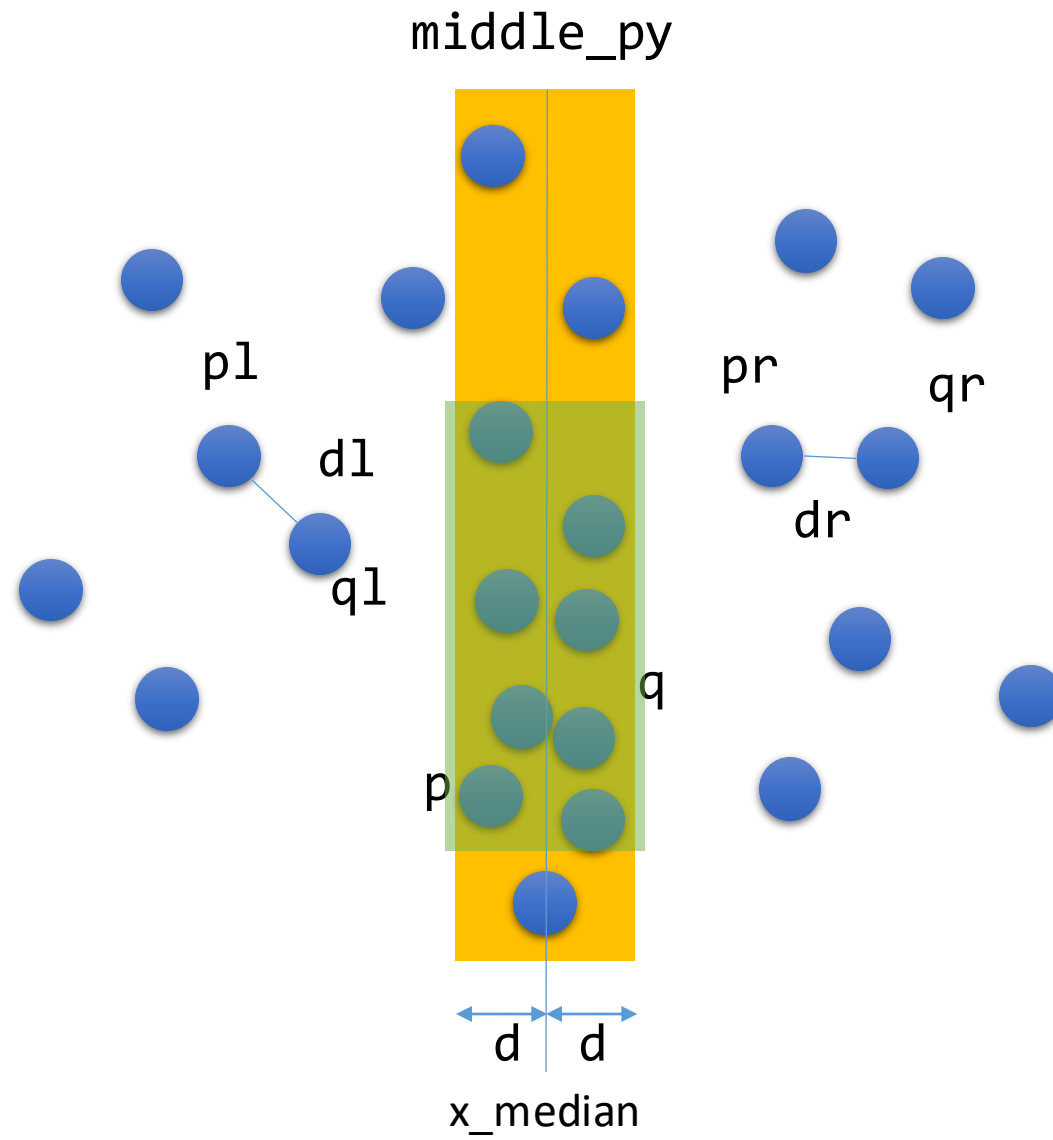
B. p and q are at most 7 positions apart in middle py

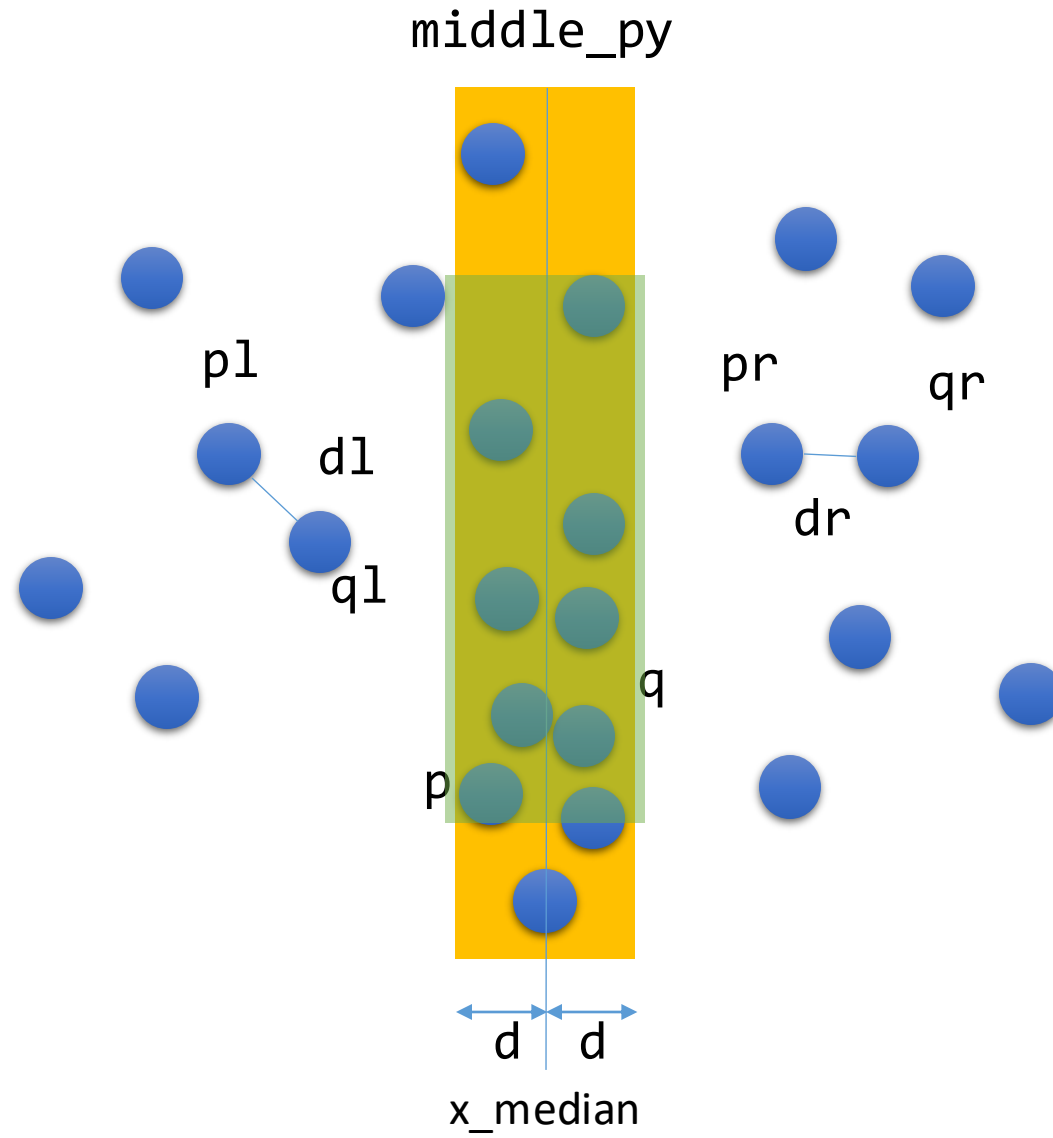
If the claim is true:

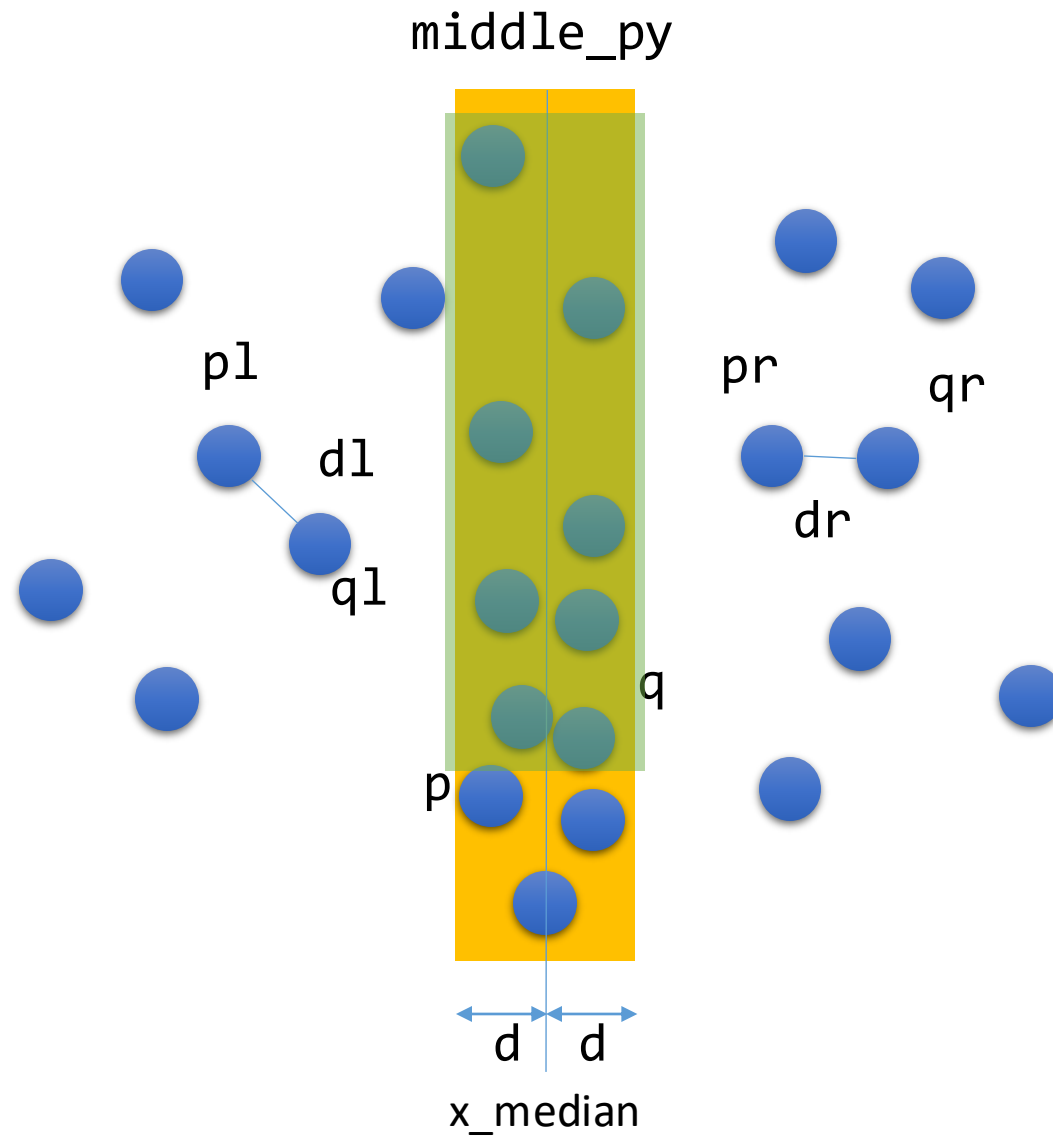
Corollary 1: If the closest pair of P is in a split pair, then our **ClosestSplitPair** procedure finds it.

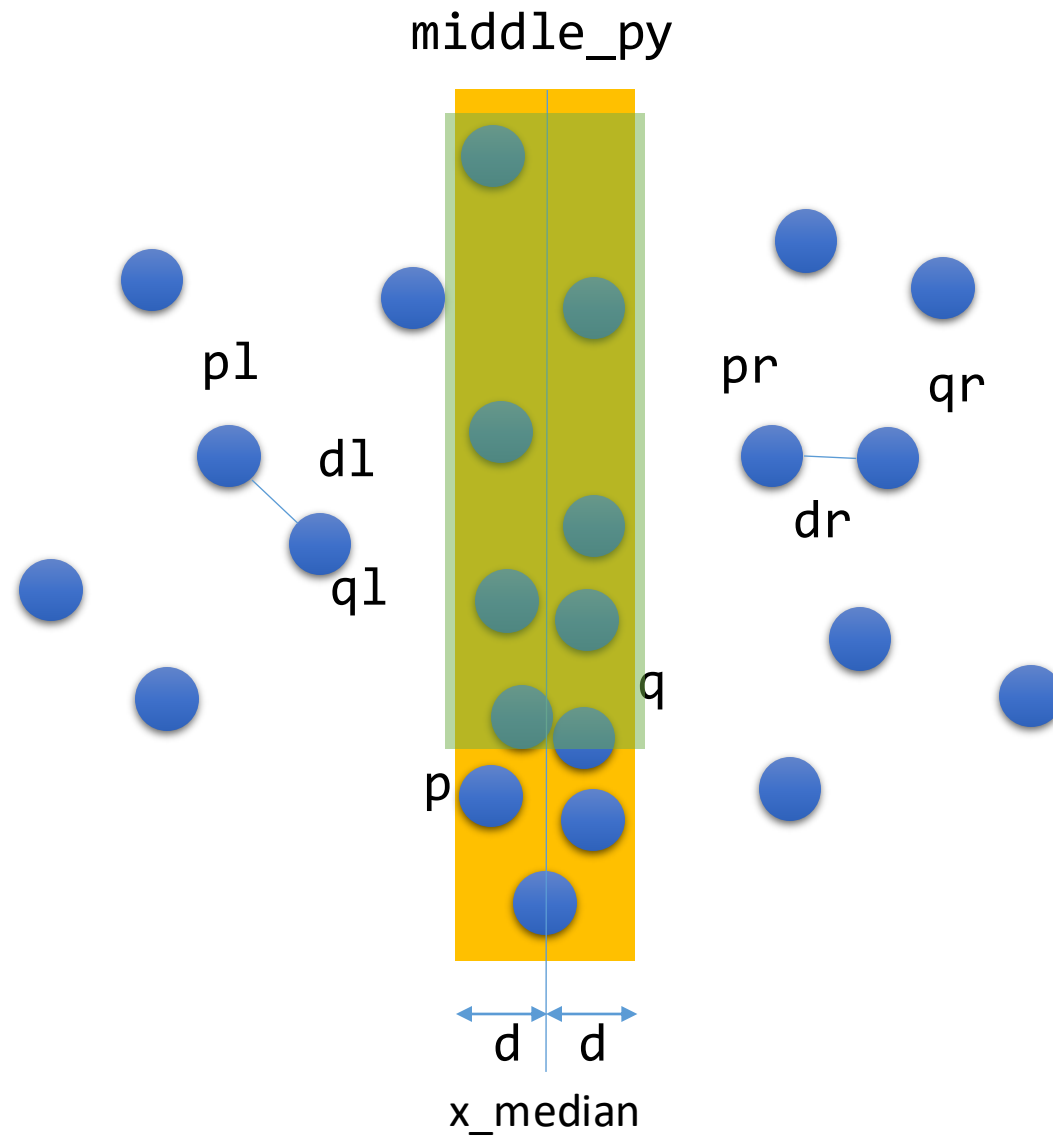
Corollary 2: **ClosestPair** is correct and runs in $O(n \lg n)$ since it has the same recursion tree as merge sort

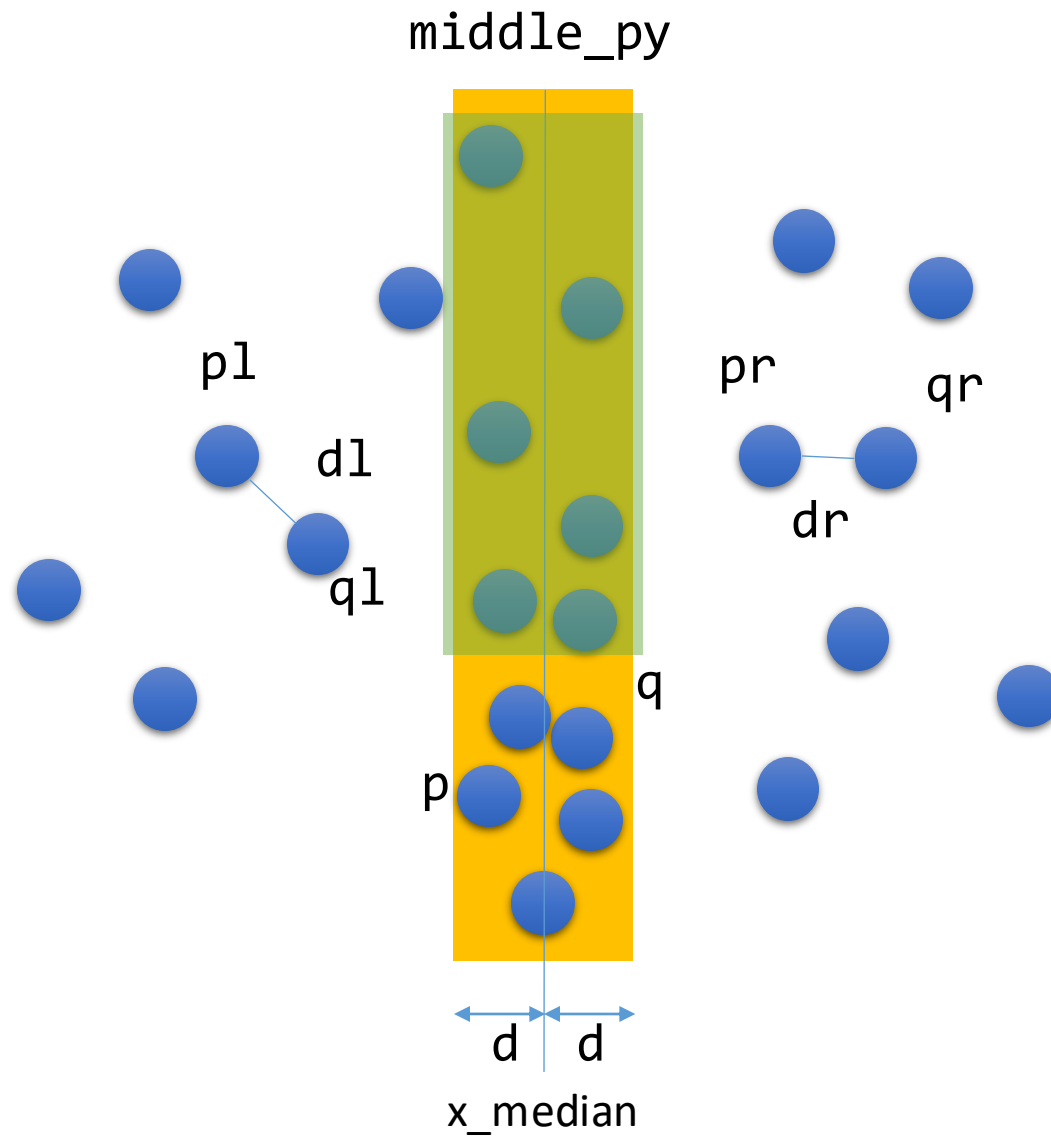


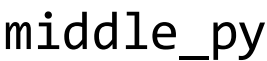


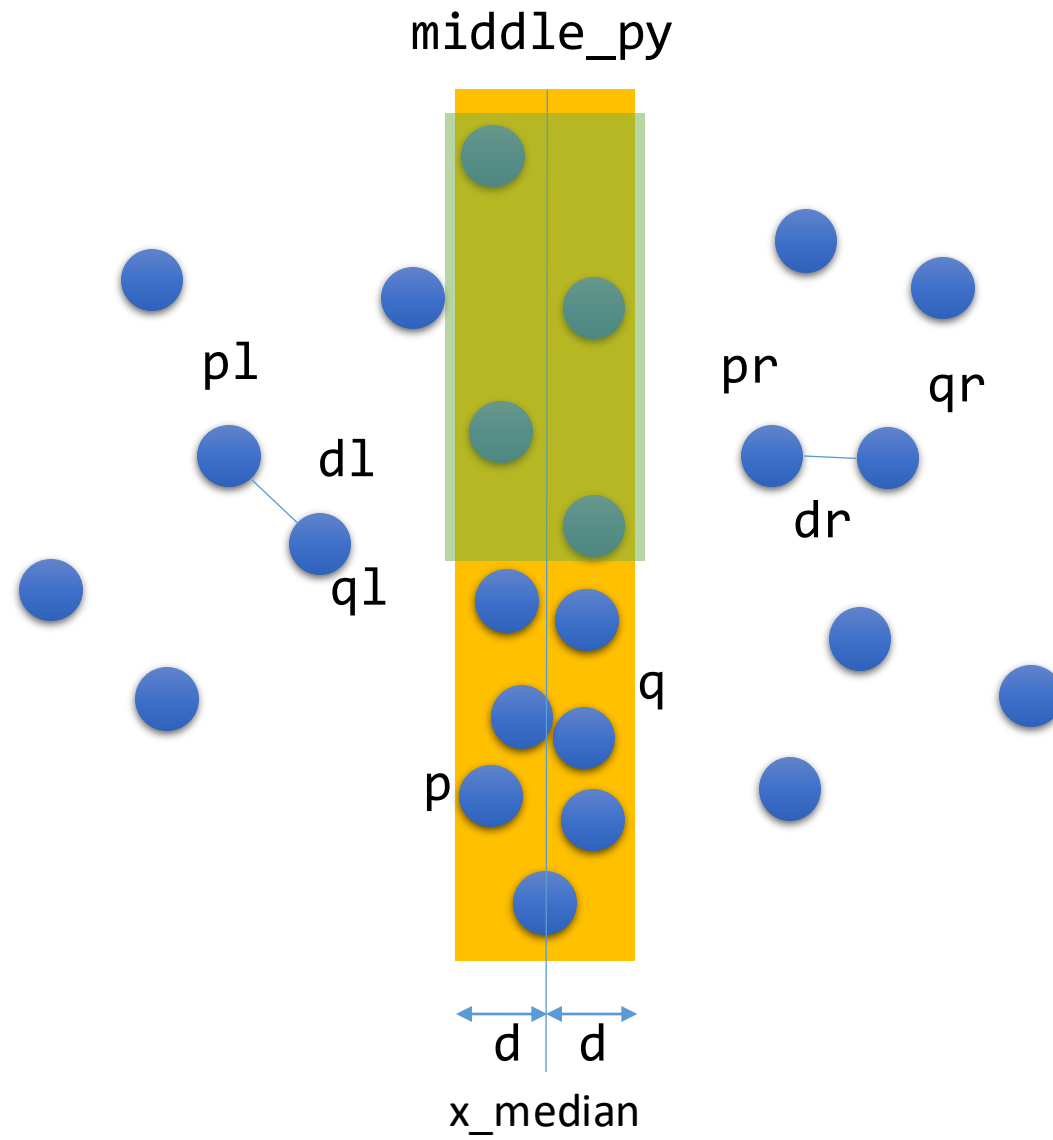




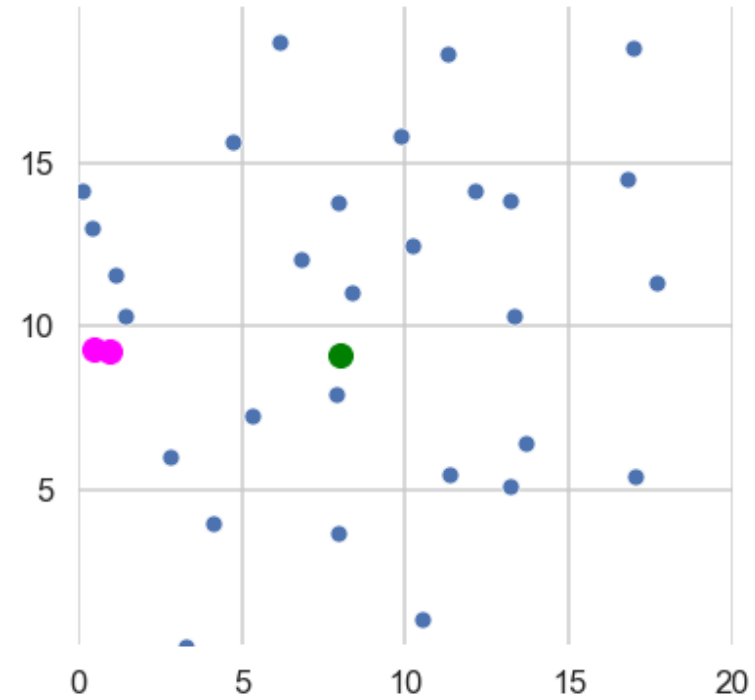




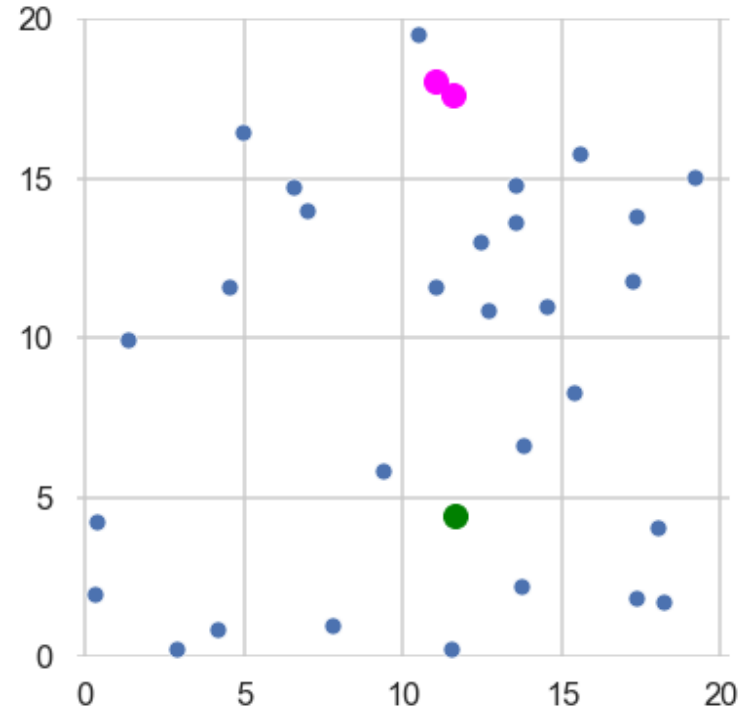




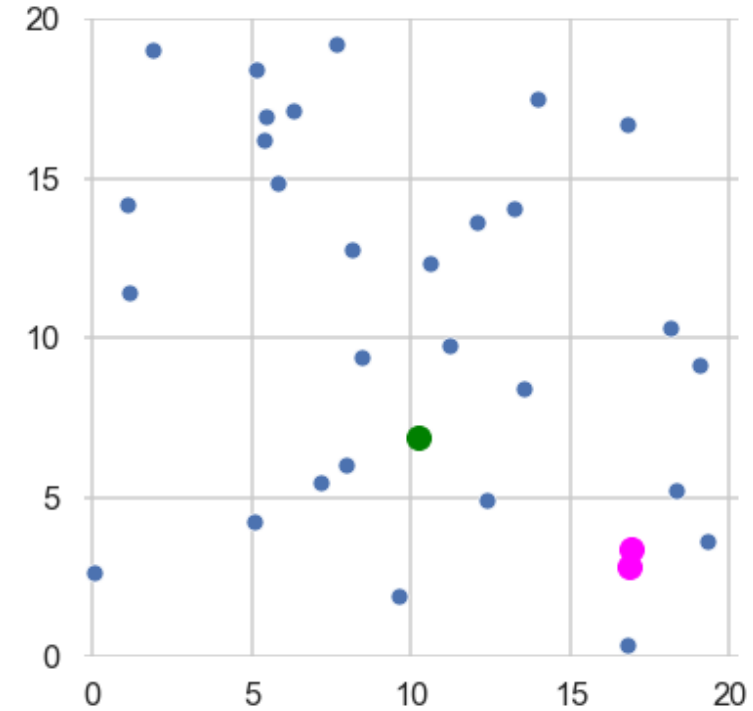
Closest on Left



Closest is Split

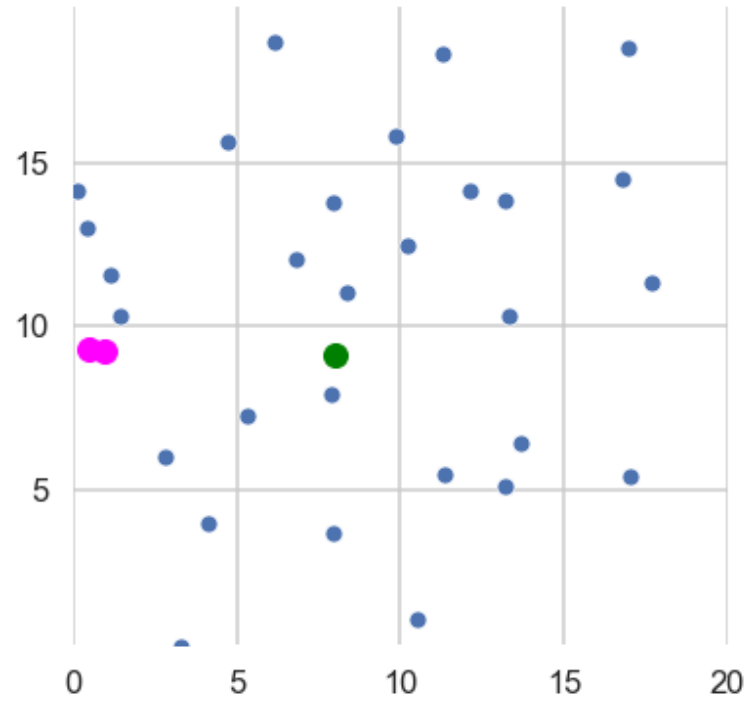


Closest on Right

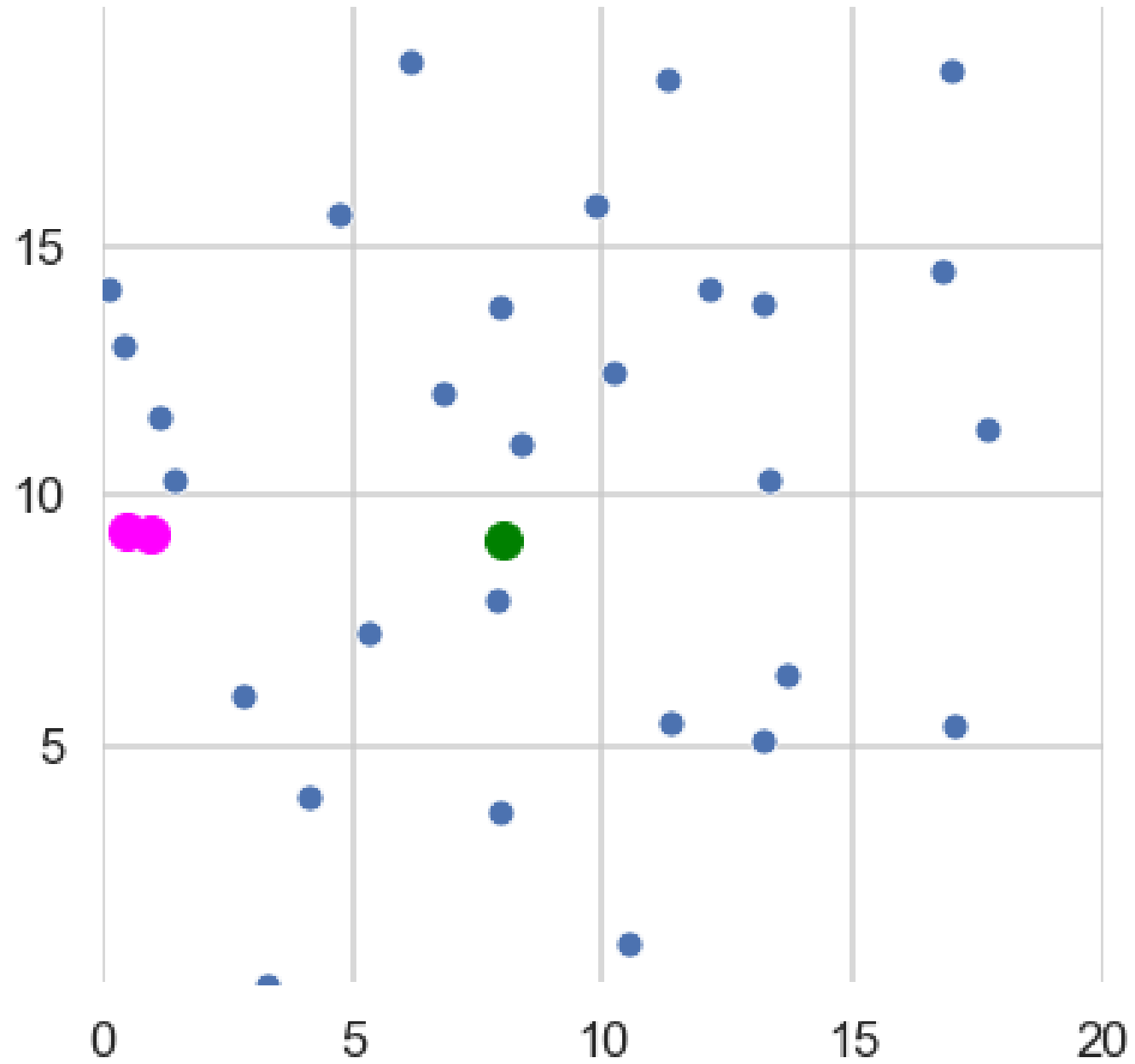


These are three different examples with different sets of points

Closest on Left

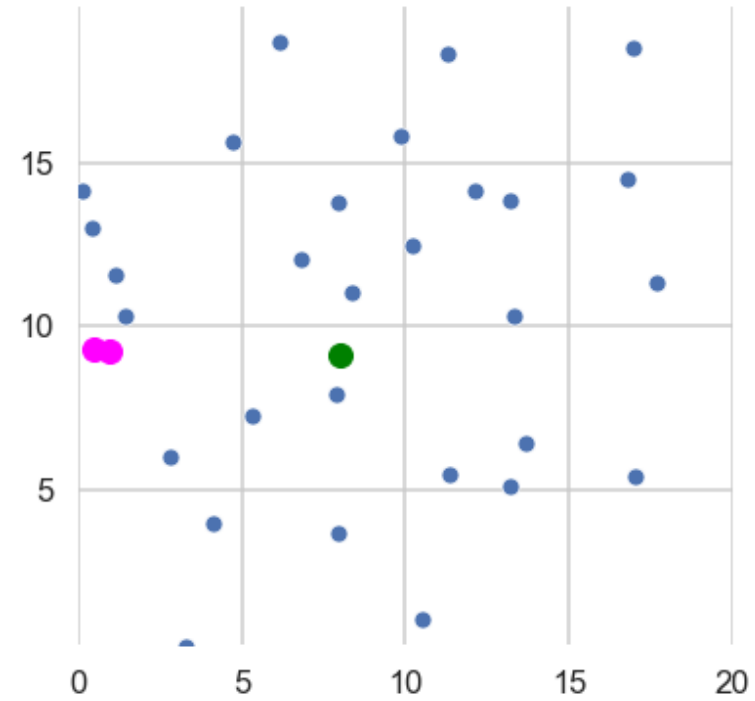


Closest on Left

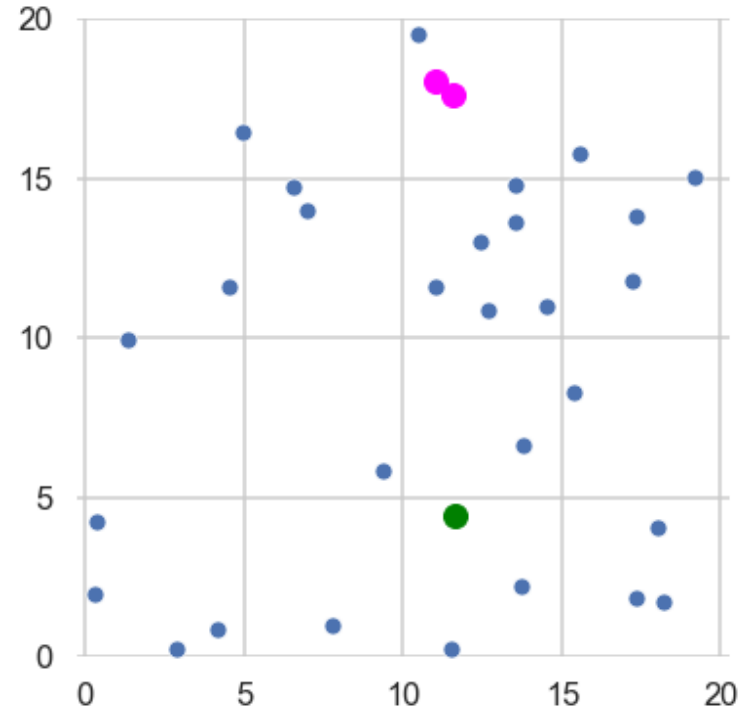


Handled by recursion
and the base case
(when viewing from
the top-level call)

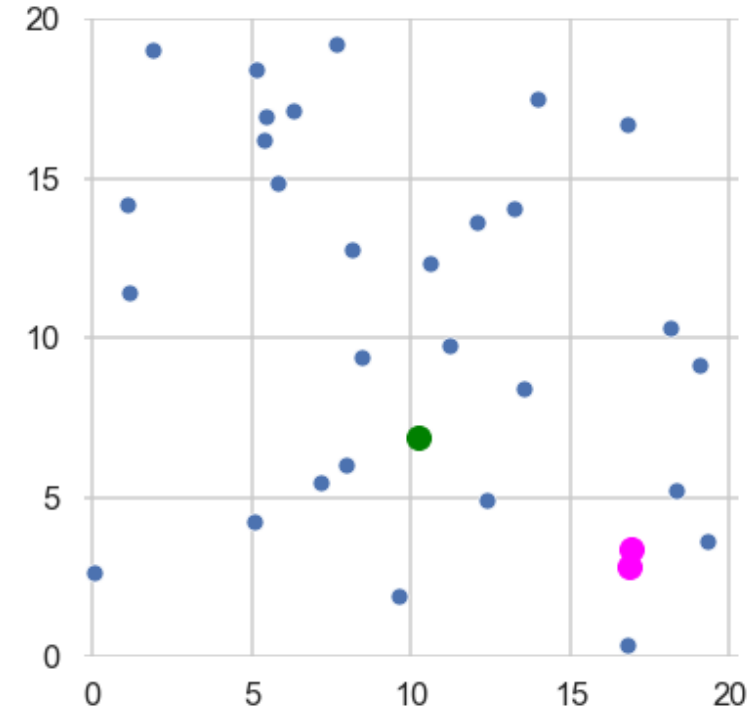
Closest on Left



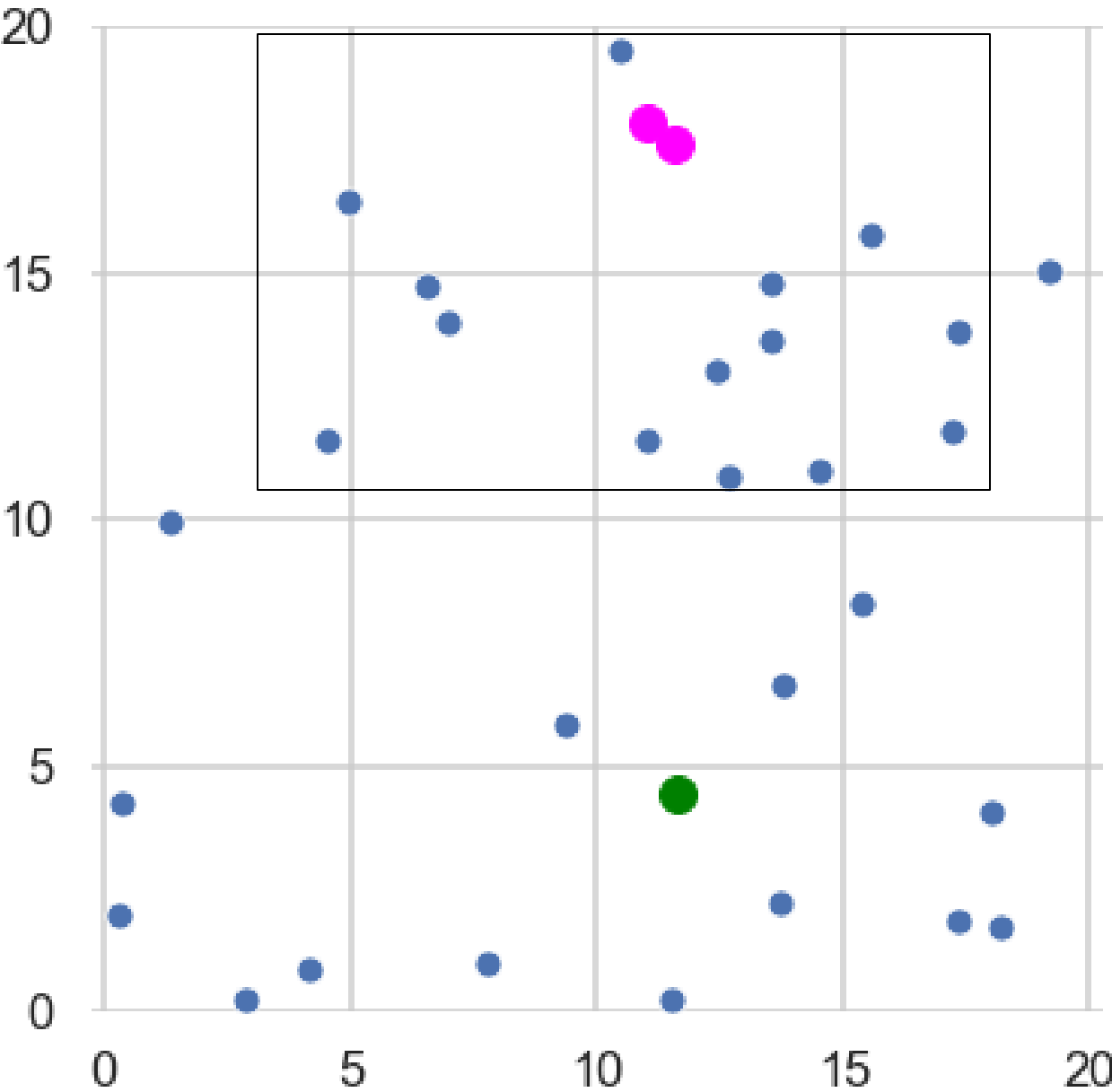
Closest is Split

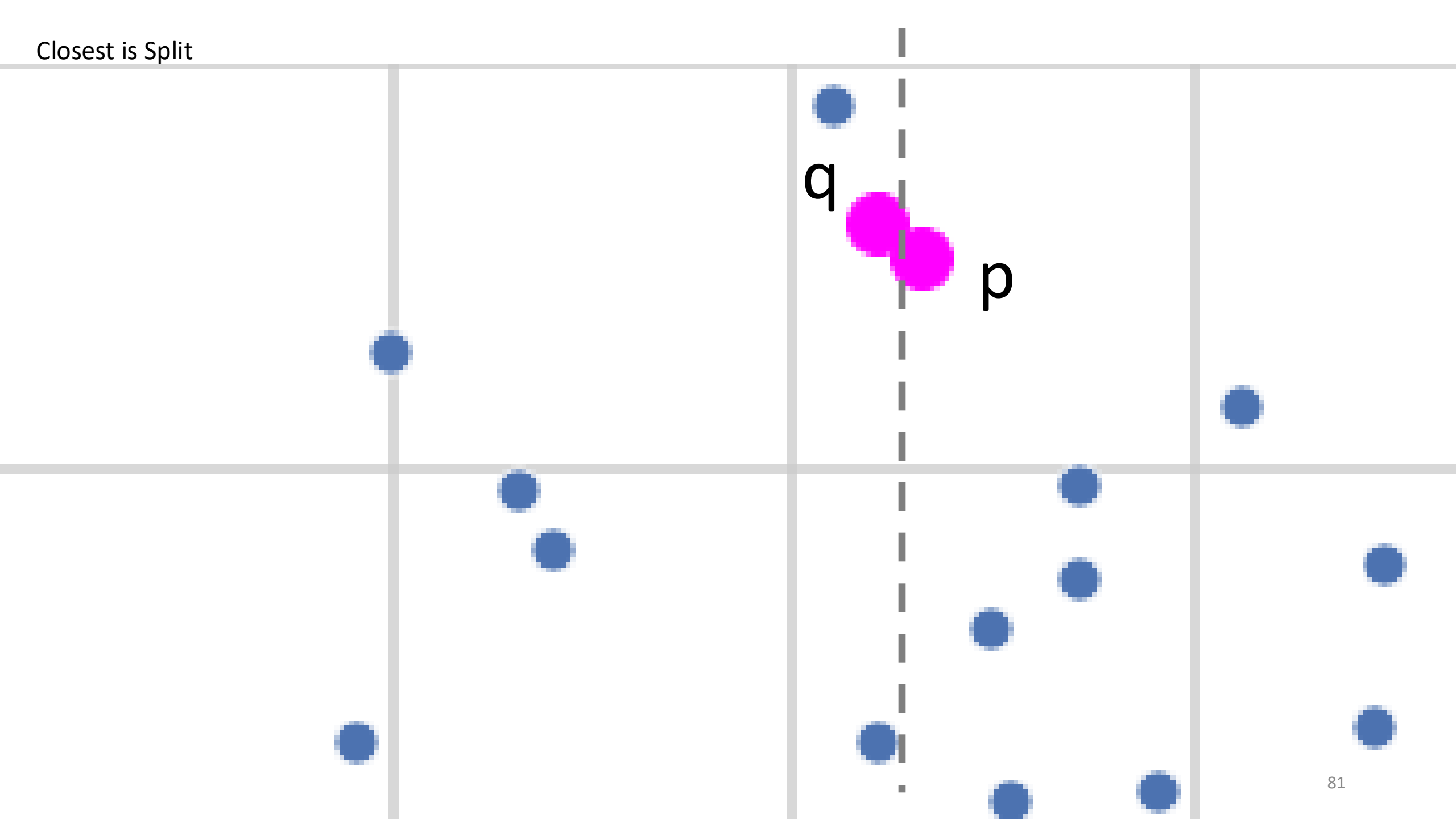


Closest on Right



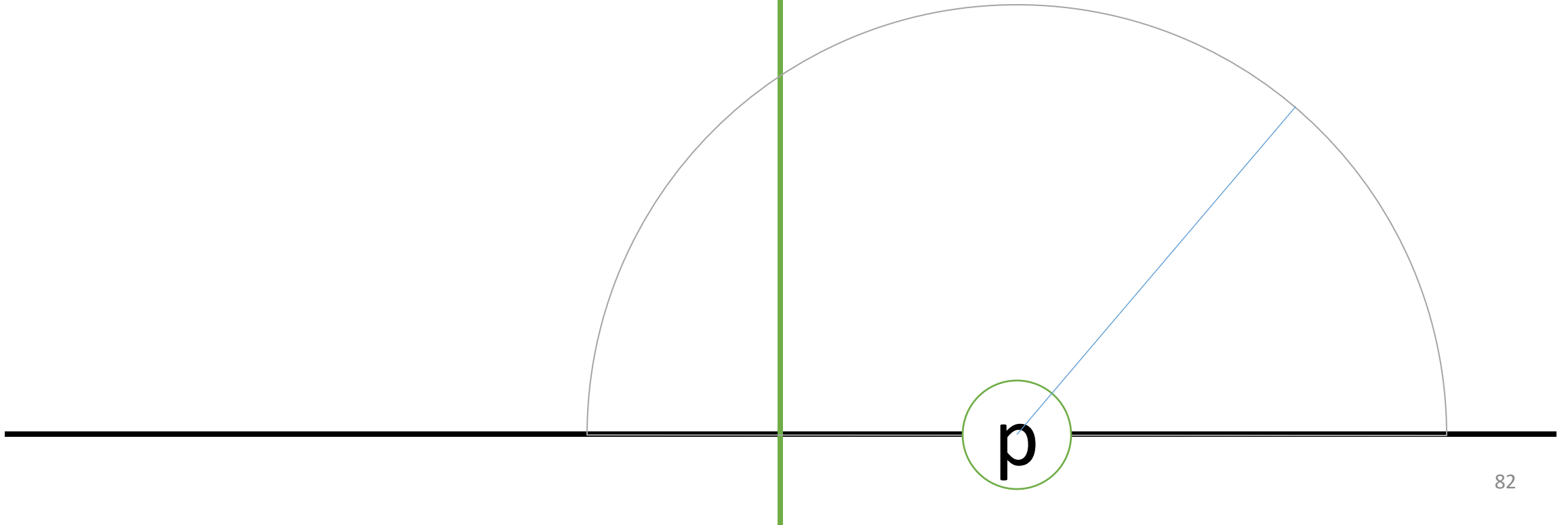
Closest is Split



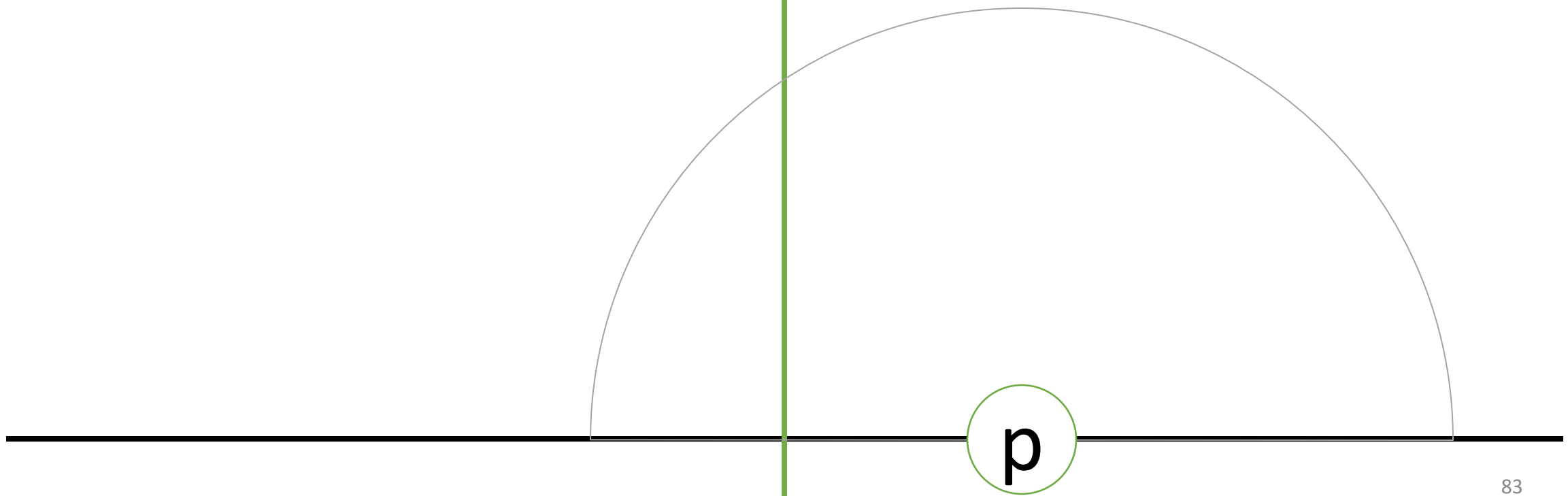


```
FOR i IN [0 ..< middle_py.length - 1]
  FOR j IN [1 ..= min(7, middle_py.length - i)]
    p = middle_py[i], q = middle_py[i + j]
    IF dist(p, q) < closest_d
      closest_d = dist(p, q)
      closest_p = p, closest_q = q
```

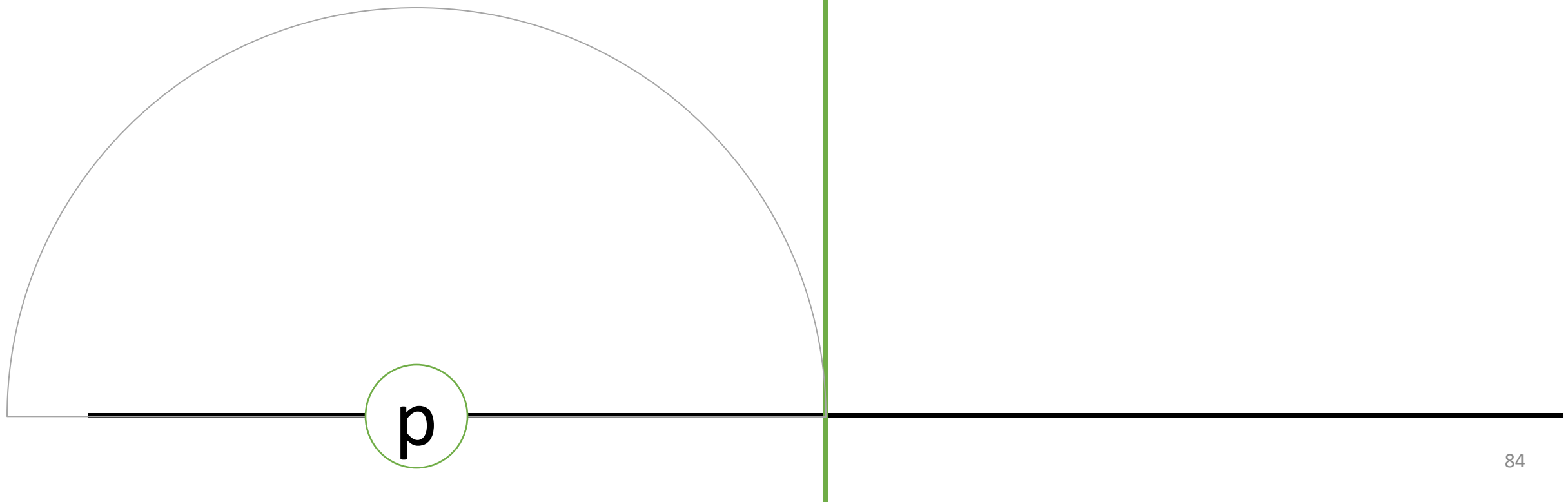
X-value of middle point



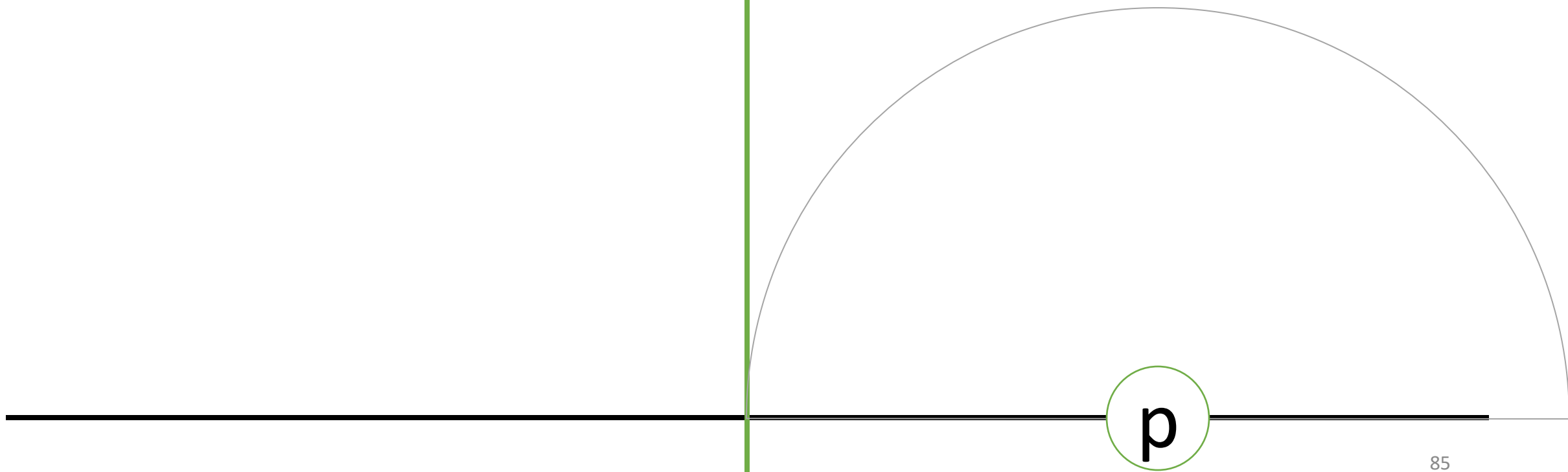
X-value of middle point

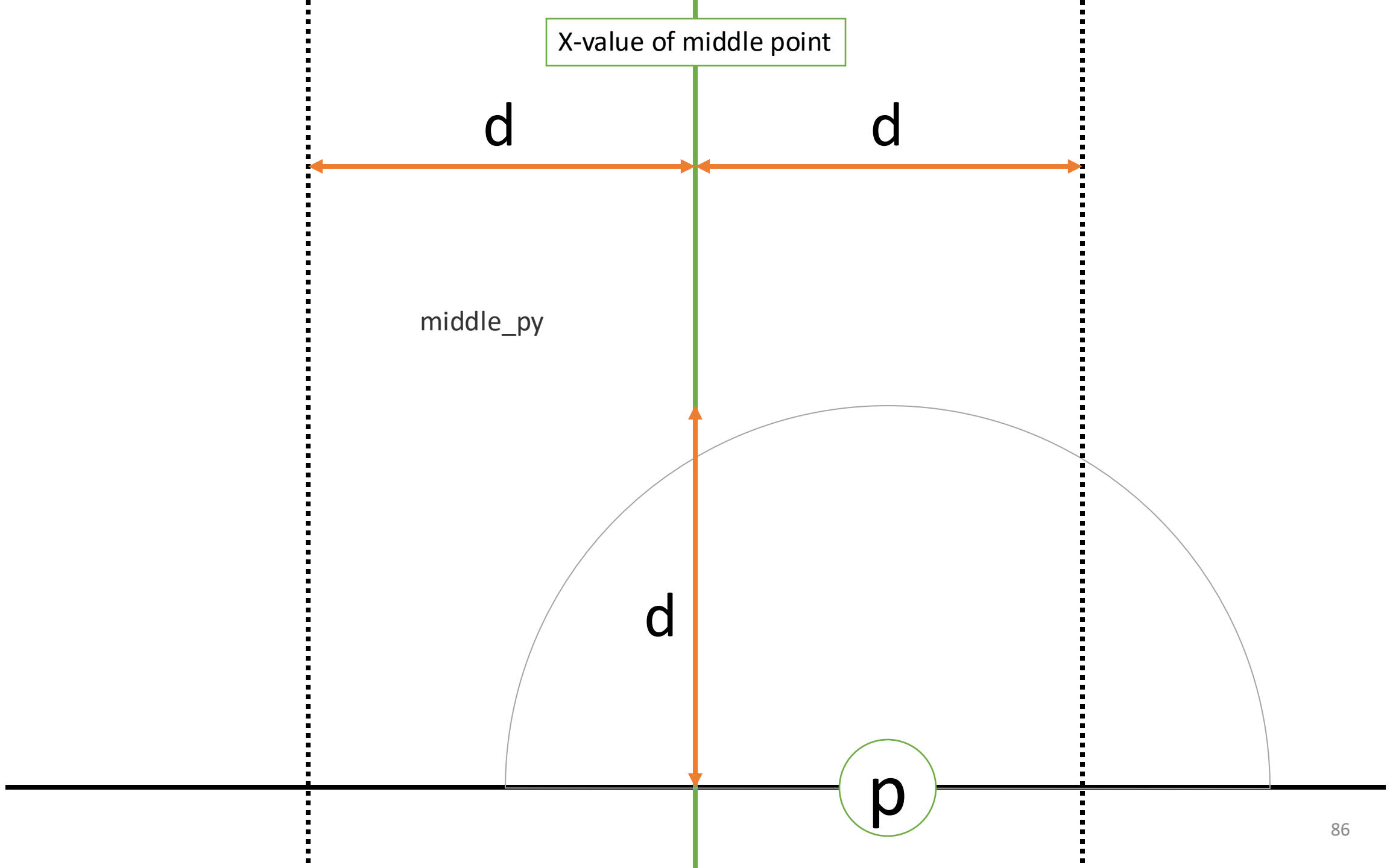


X-value of middle point

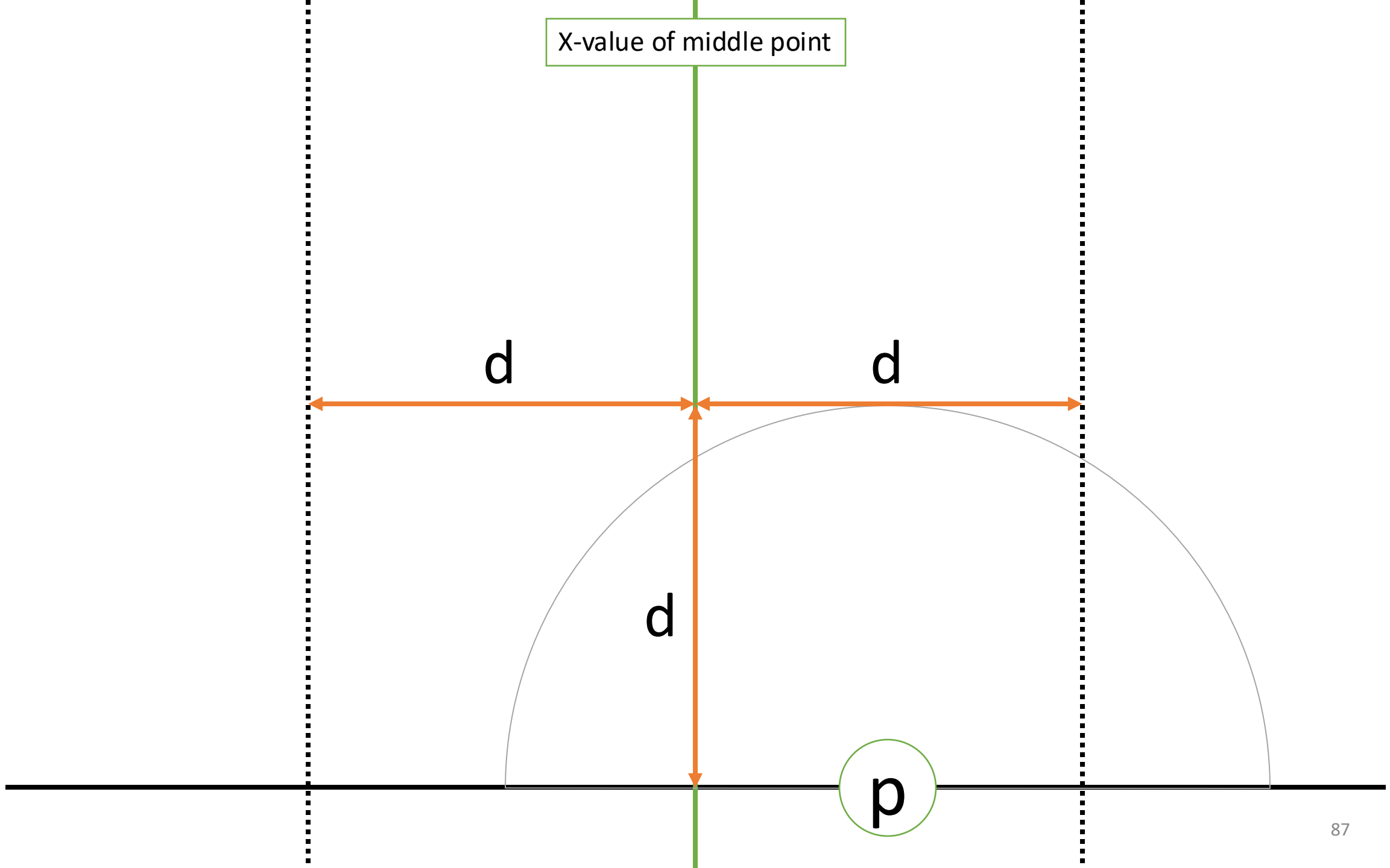


X-value of middle point

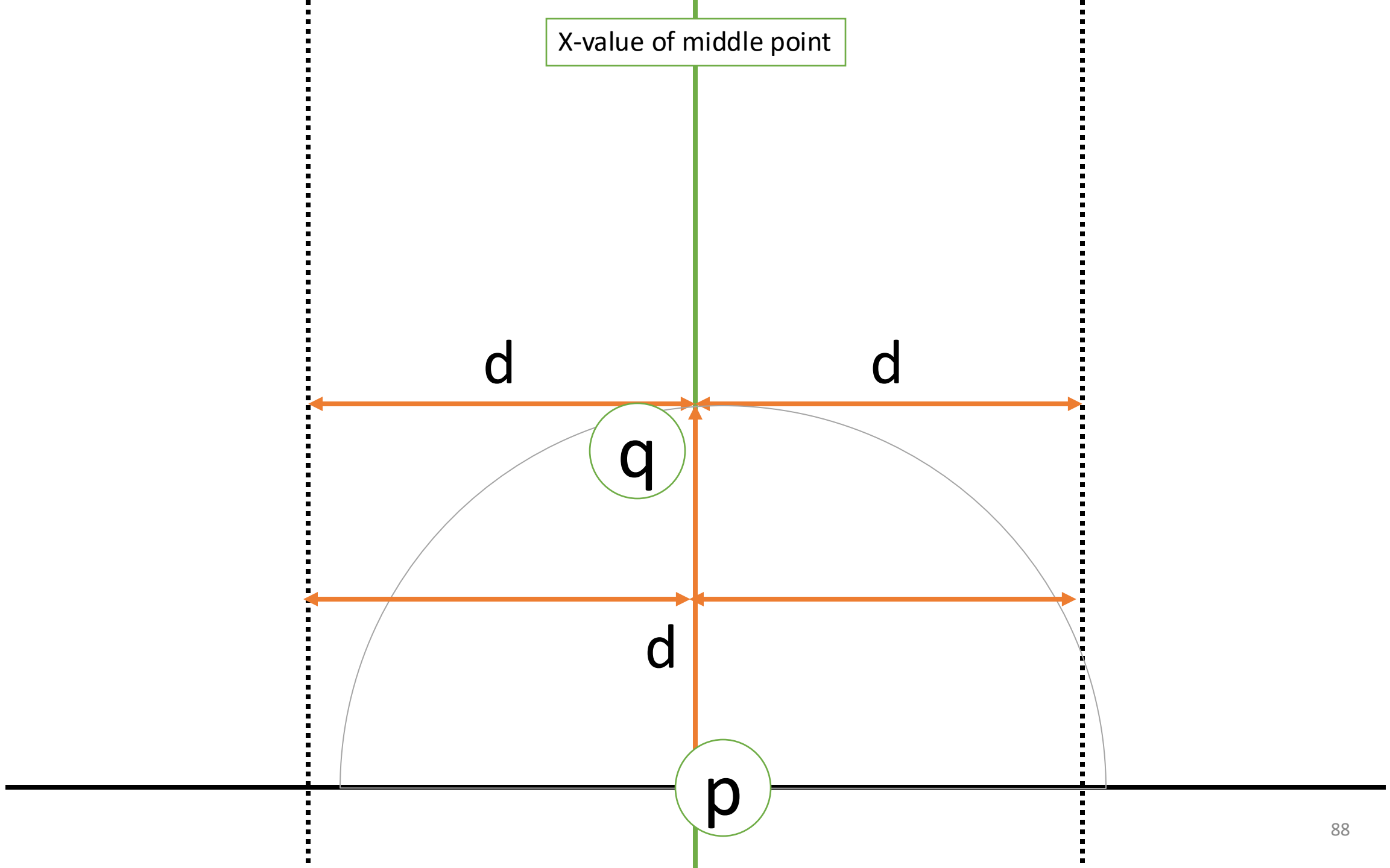




X-value of middle point

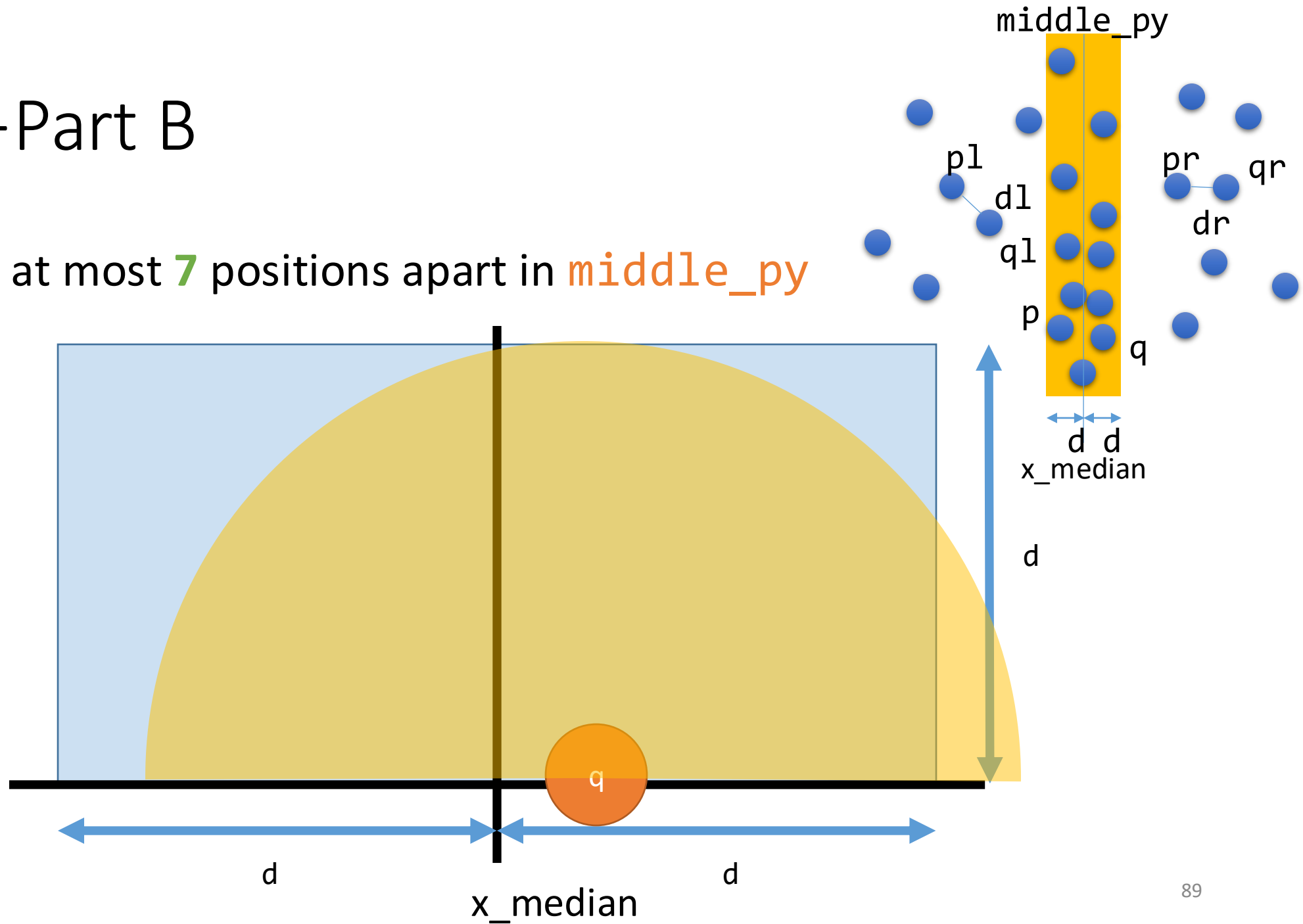


X-value of middle point



Proof—Part B

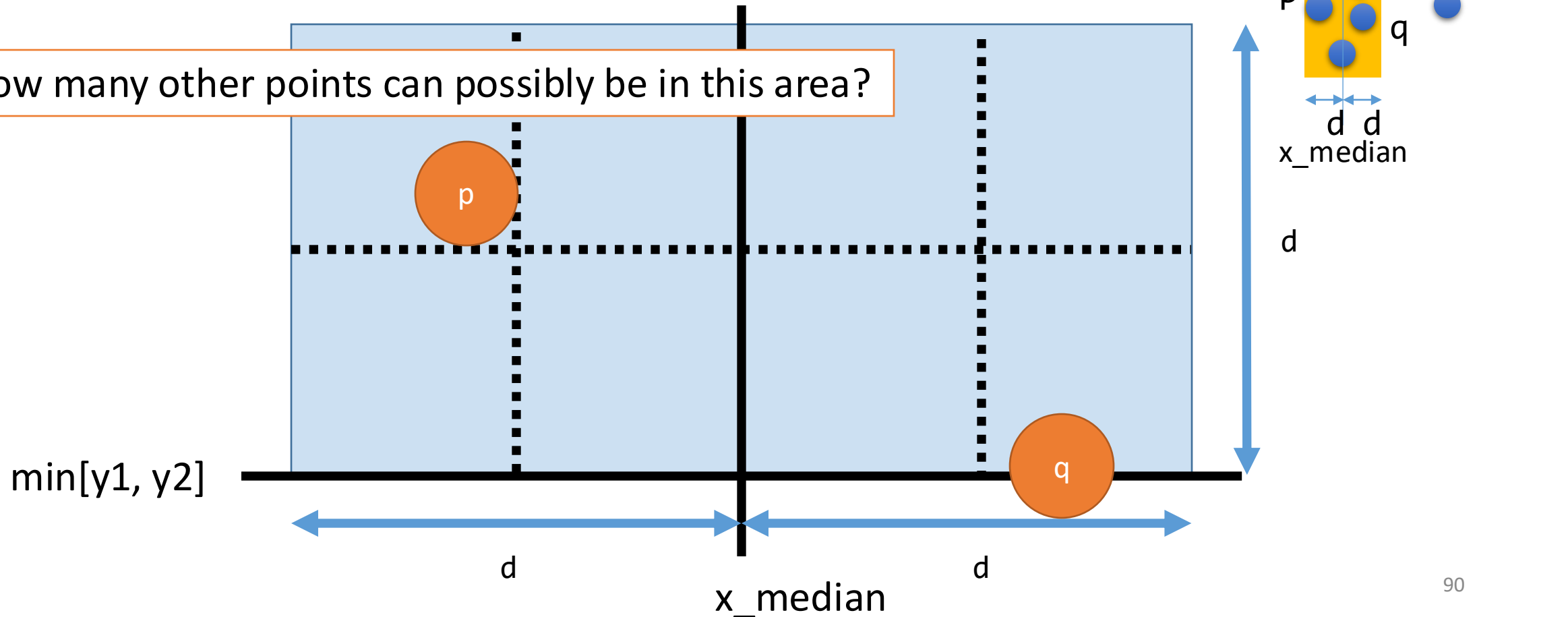
p and q are at most 7 positions apart in `middle_py`



Proof—Part B

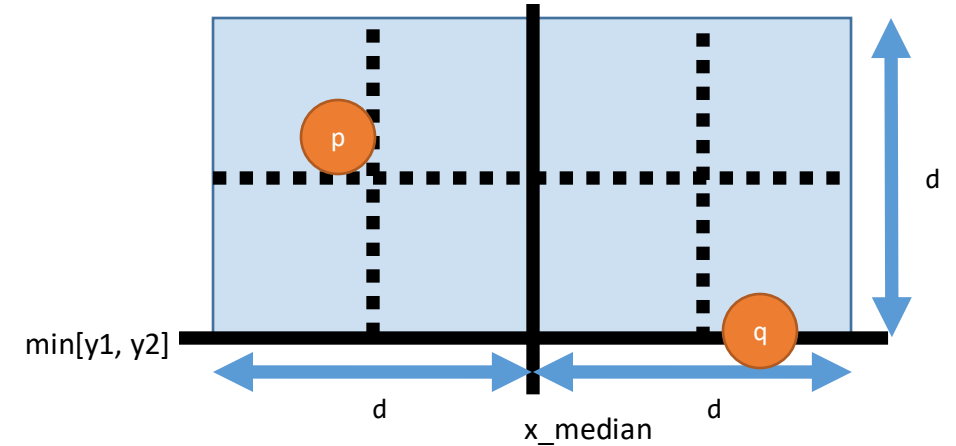
p and q are at most 7 positions apart in `middle_py`

How many other points can possibly be in this area?



Proof—Part B

p and q are at most 7 positions apart
in `middle_py`



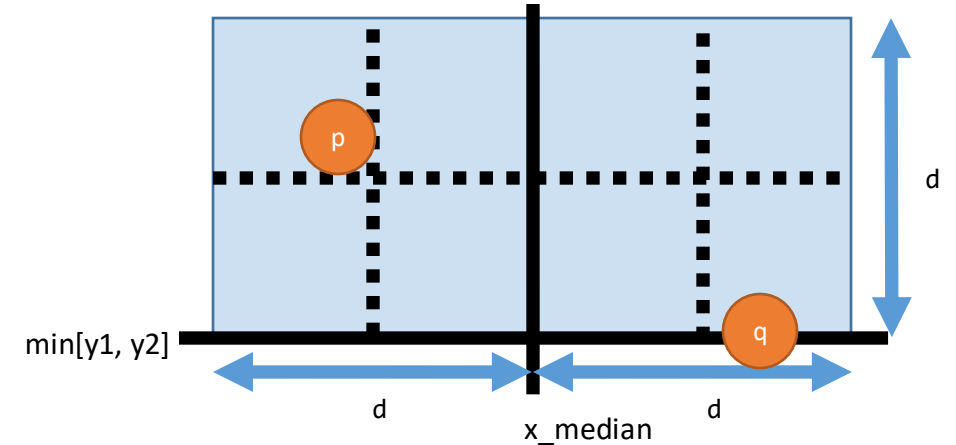
Lemma 1: All points of `middle_py` with a y-coordinate between those of p and q lie within those 8 boxes.

Proof:

1. First, recall that the y-coordinate of p, q differs by less than d.
2. Second, by definition of `middle_py`, all have an x-coordinate between $x_median - d$ and $x_median + d$.

Proof—Part B

p and q are at most 7 positions apart
in `middle_py`



Lemma 1: All points of `middle_py` with a y-coordinate between those of p and q lie within those 8 boxes.

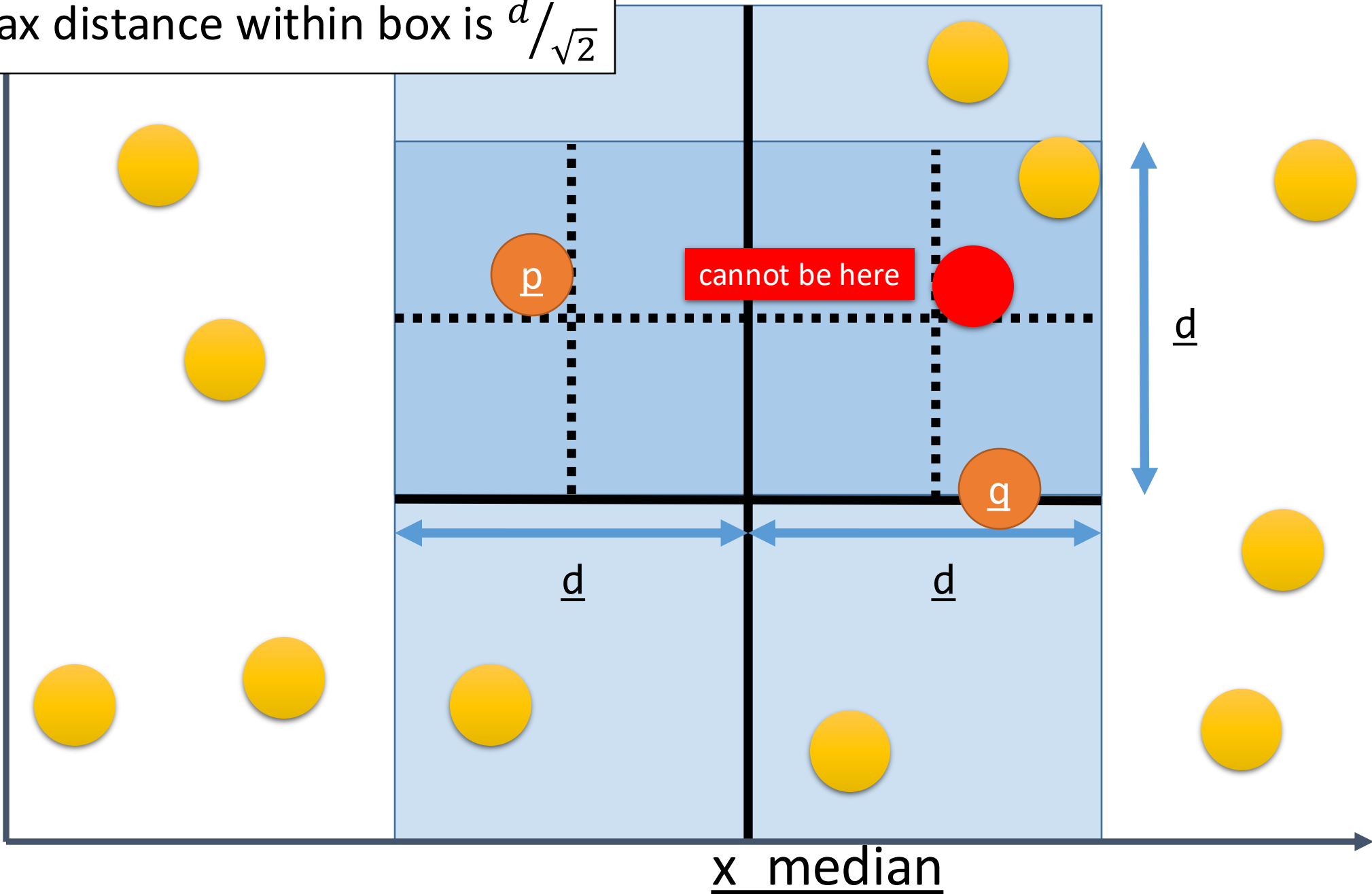
Lemma 2: At most one point of P can be in each box.

Proof: By contradiction. Suppose points a and b lie in the same box. Then

1. a and b are either both in L or both in R
2. $d(a, b) \leq d/2 \cdot \sqrt{2} < d$

This is a contradiction! How did we define d ?

Max distance within box is $d/\sqrt{2}$



ClosestPair finds the closest pair

Let $p \in \text{left}$, $q \in \text{right}$ be a split pair with $d(p, q) < d$

Then

- A. p and $q \in \text{middle_py}$, and
- B. p and q are at most 7 positions apart in middle_py

If the claim is true:

Corollary 1: If the closest pair of P is in a split pair, then our **ClosestSplitPair** procedure finds it.

Corollary 2: **ClosestPair** is correct and runs in $O(n \lg n)$ since it has the same recursion tree as merge sort

Theorem for correctness of ClosestPair

Theorem:

Provided a set of n points called P , the **ClosestPair** algorithm find the closest pair of points according to their pairwise Euclidean distances.

Input: P , a set of n points that lie in a (two-dimensional) plane

Output: a pair of points (p, q) that are the “closest”

Distance is measured using Euclidean distance:

$$d(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Assumptions: None

Summary Closest Pair

1. Copy P and sort one copy by x and the other copy by y in $O(n \lg n)$
2. Divide P into a left and right in $O(n)$
3. Conquer by recursively searching left and right
4. Look for the closest pair in middle_py in $O(n)$
 - Must filter by x
 - And scan through middle_py by looking at adjacent points