Learning Communities

- About 8 per group
- Meet with your TA twice per week for 1 hour each time
- **Fill out the survey posted on slack!**

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

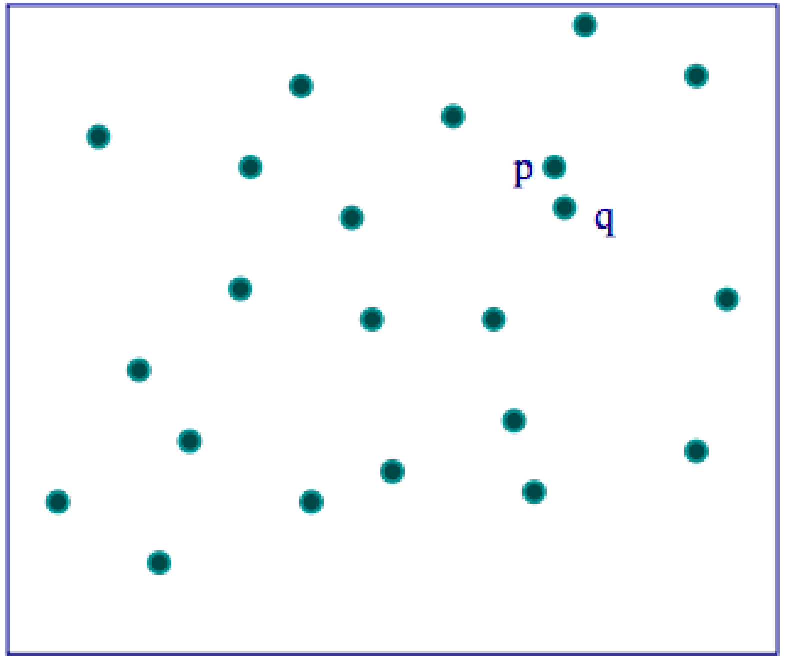
  (p_x, p_y) (q_x, q_y)

- **Output**: a pair of points (p, q) that are the "closest"
  - Distance is measured using Euclidean distance:

$$d(p, q) = sqrt((p_x - q_x)^2 + (p_y - q_y)^2)$$ $= T(n) = X$

- **Assumptions**: None

Closest Pair Problem

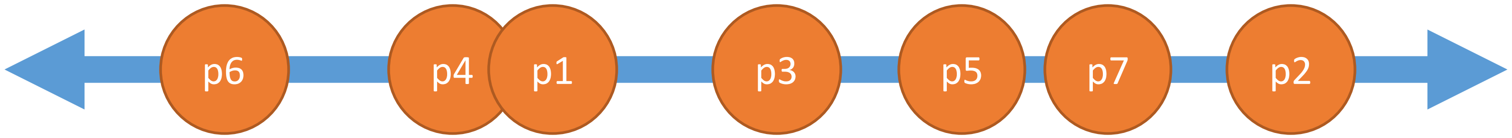


Can we do better than $O(n^2)$?

- 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?

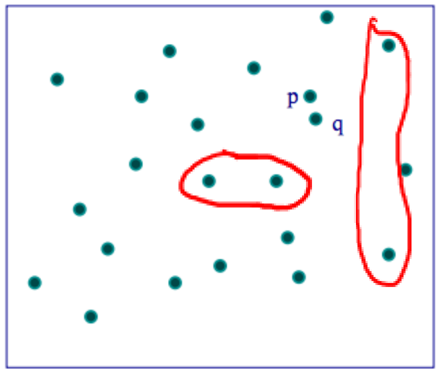
- Sort by position : O(n lg n)

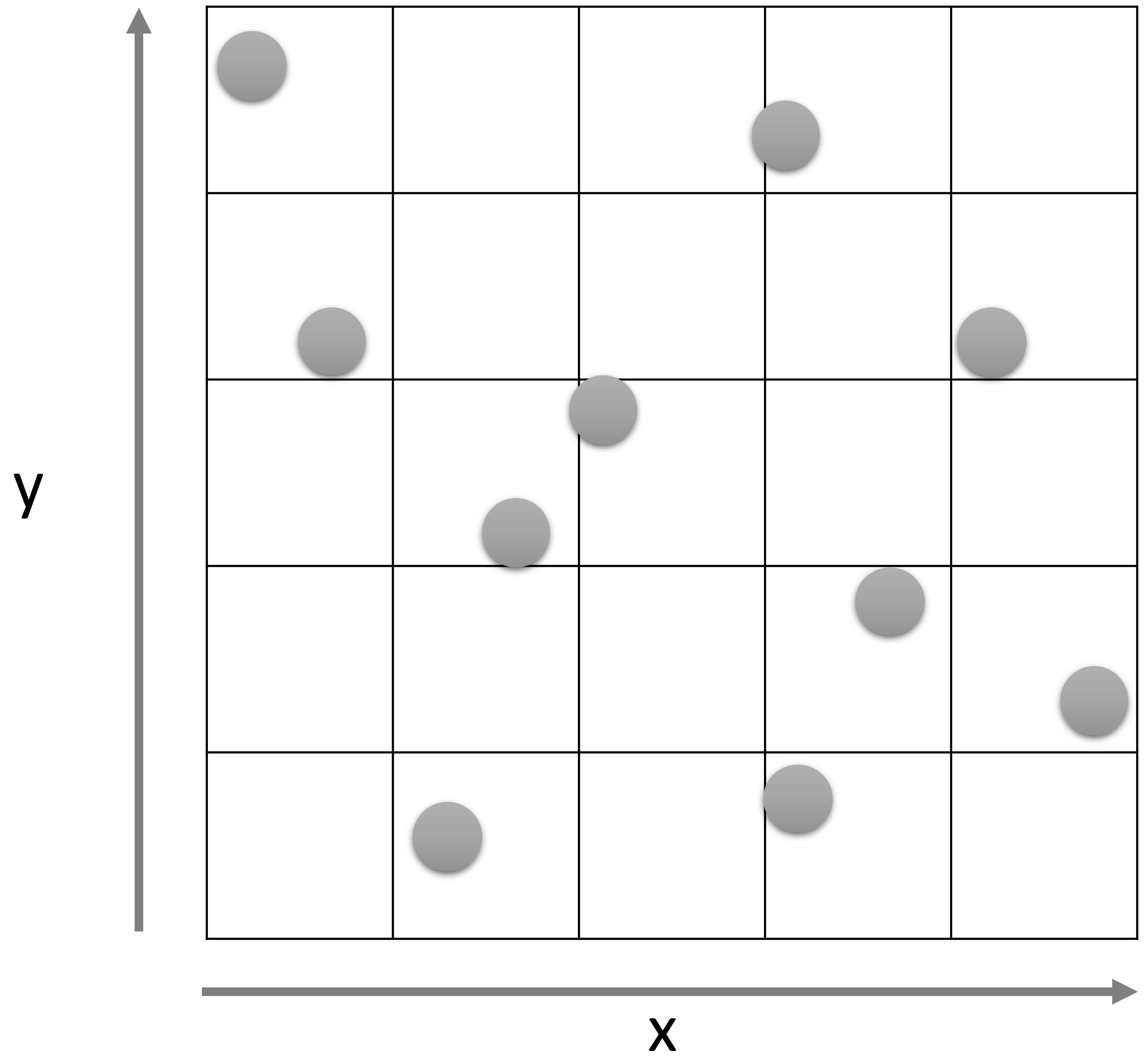
p6	p4	p1	p3	p5	p7	p2

- Return the closest two using a linear scan : O(n)
- Total time : O(n lg n) + O(n) = O(n lg n)

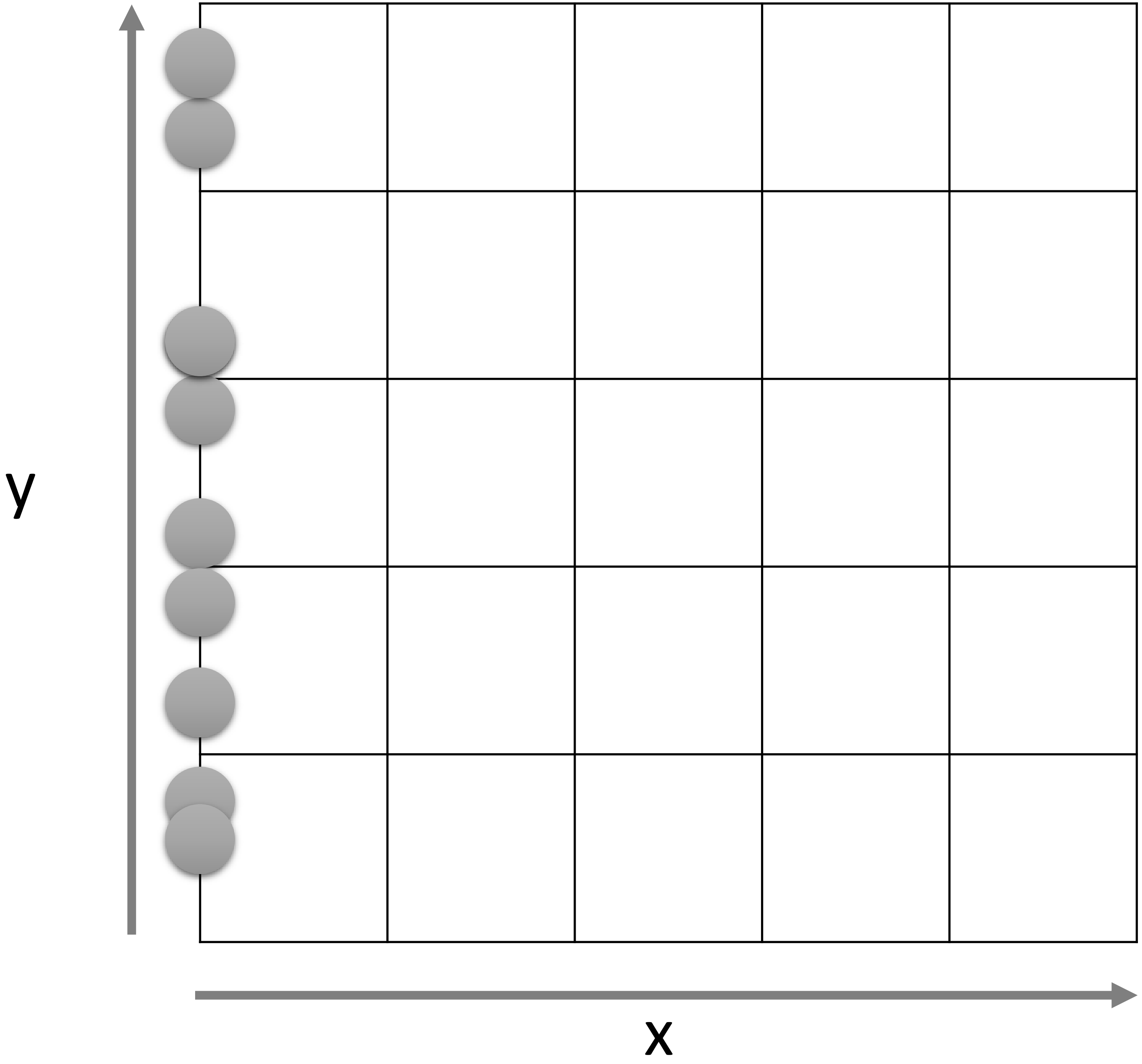
Any problems using this approach for the two-dimensional case?

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

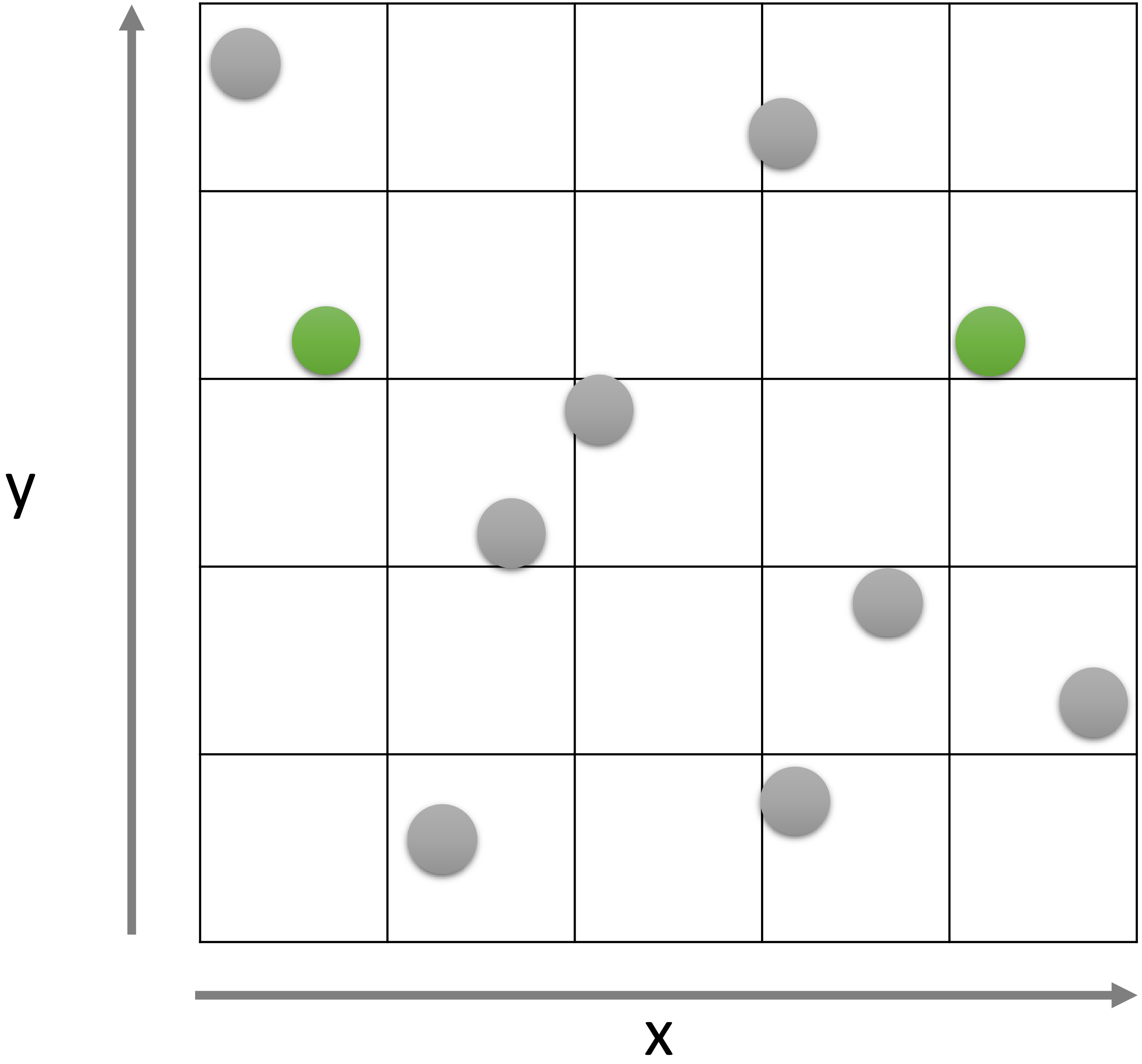




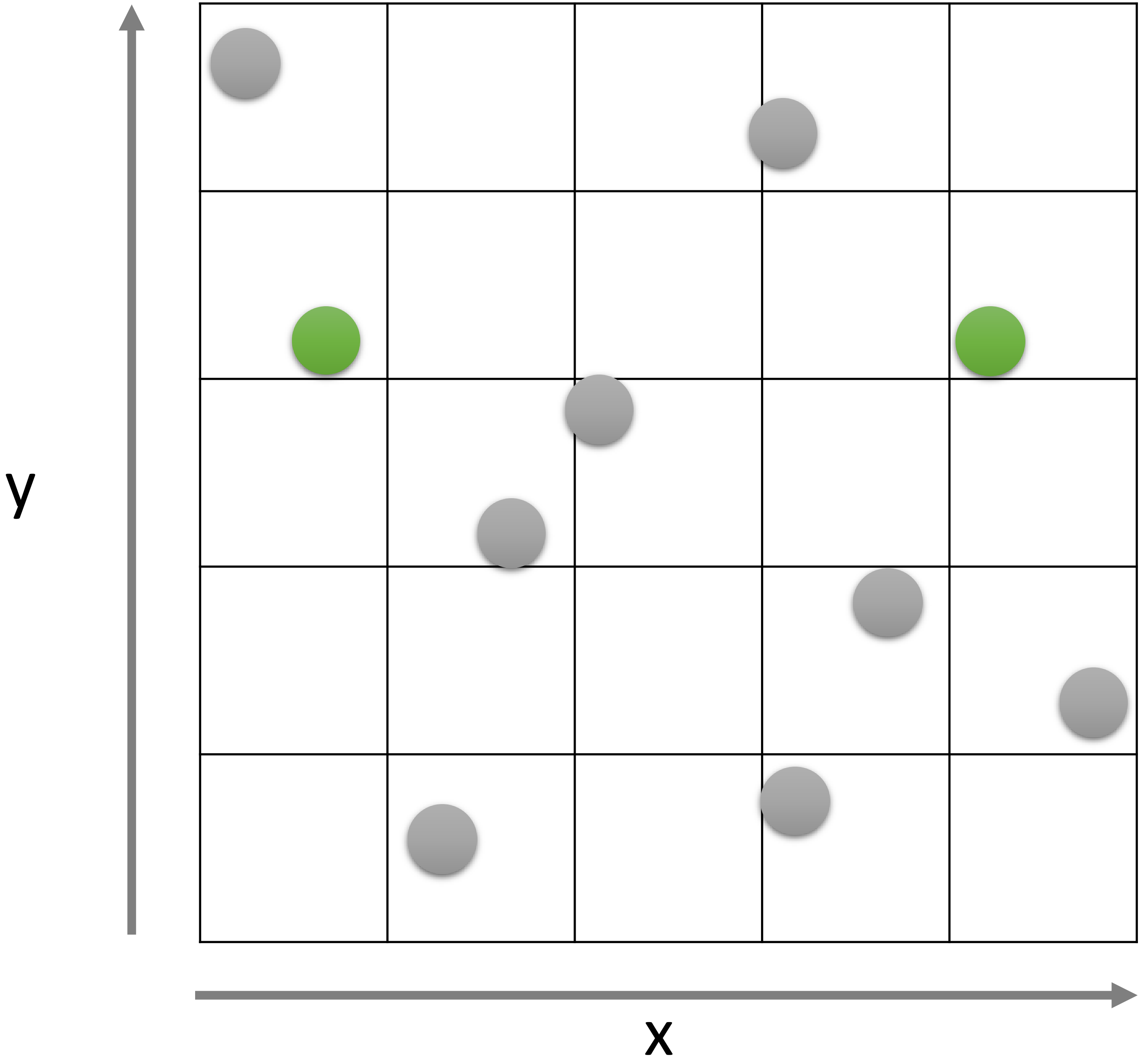
1. Which two are closest on the y-axis?



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

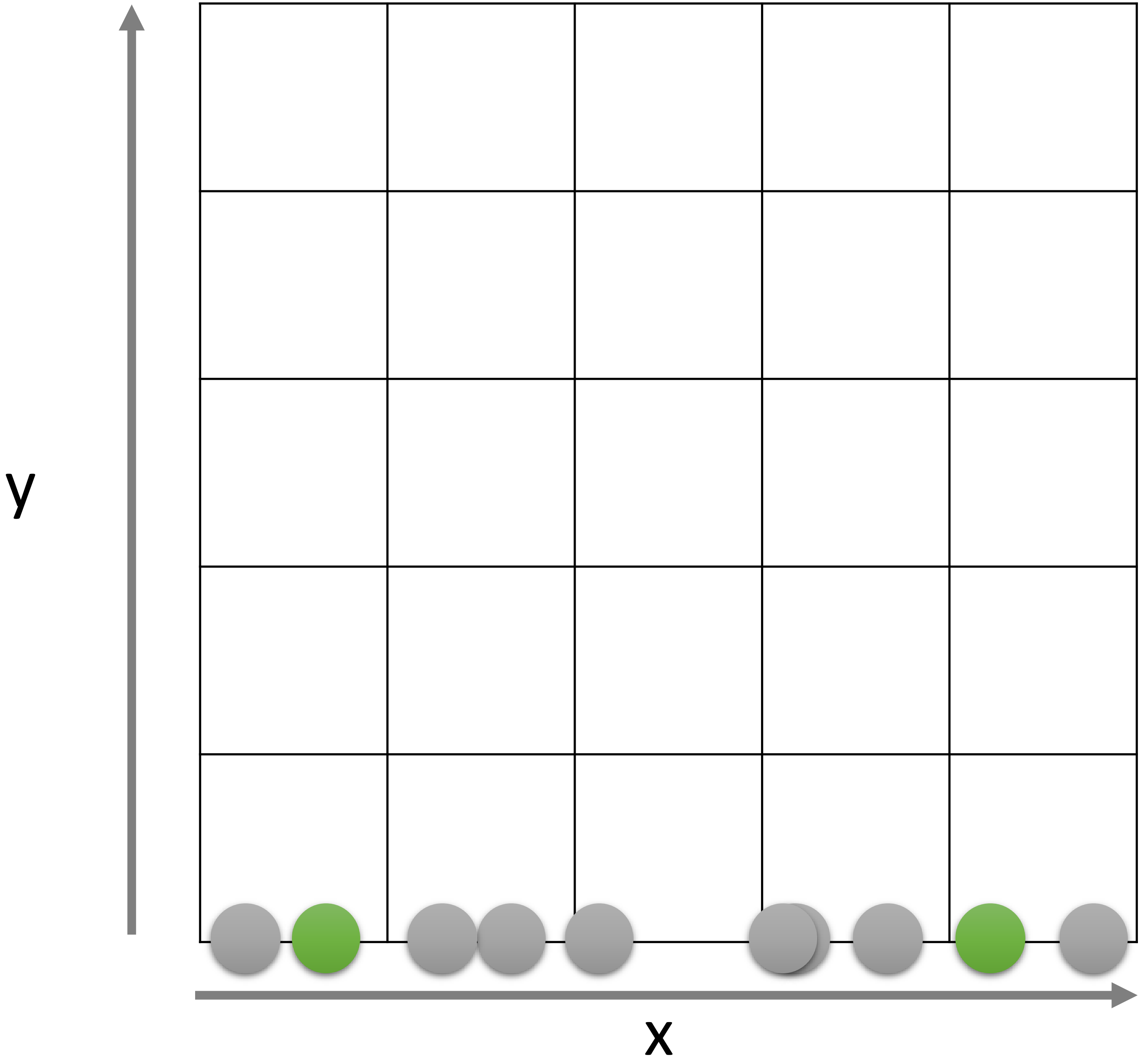


1. Which two are closest on the y-axis?



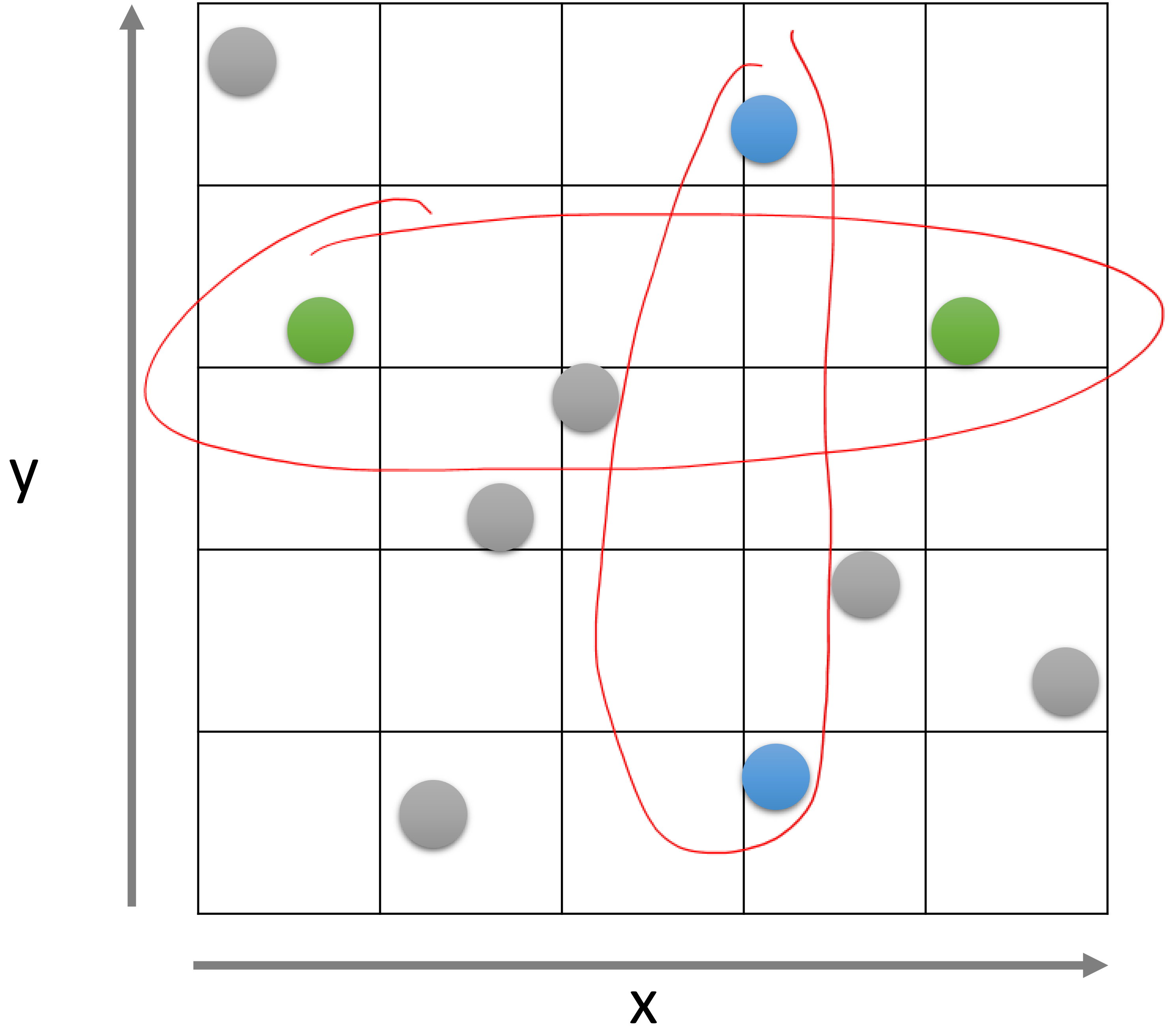
1. Which two are closest on the y-axis?

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



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

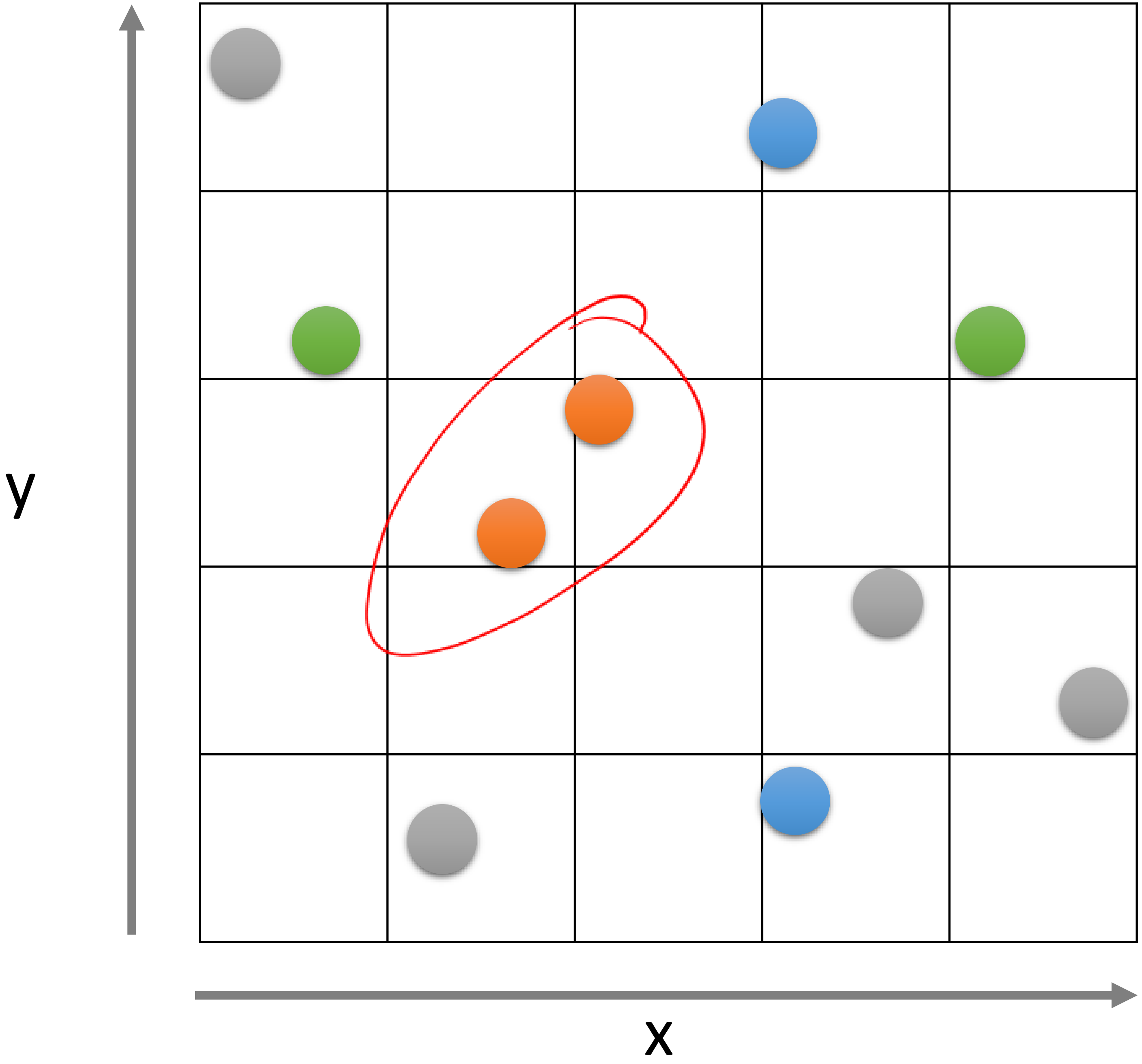
2. Which two are closest on the x-axis?



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?

Closet 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)

$$O(n \lg n) \leq T(n) \leq O(n^2)$$

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

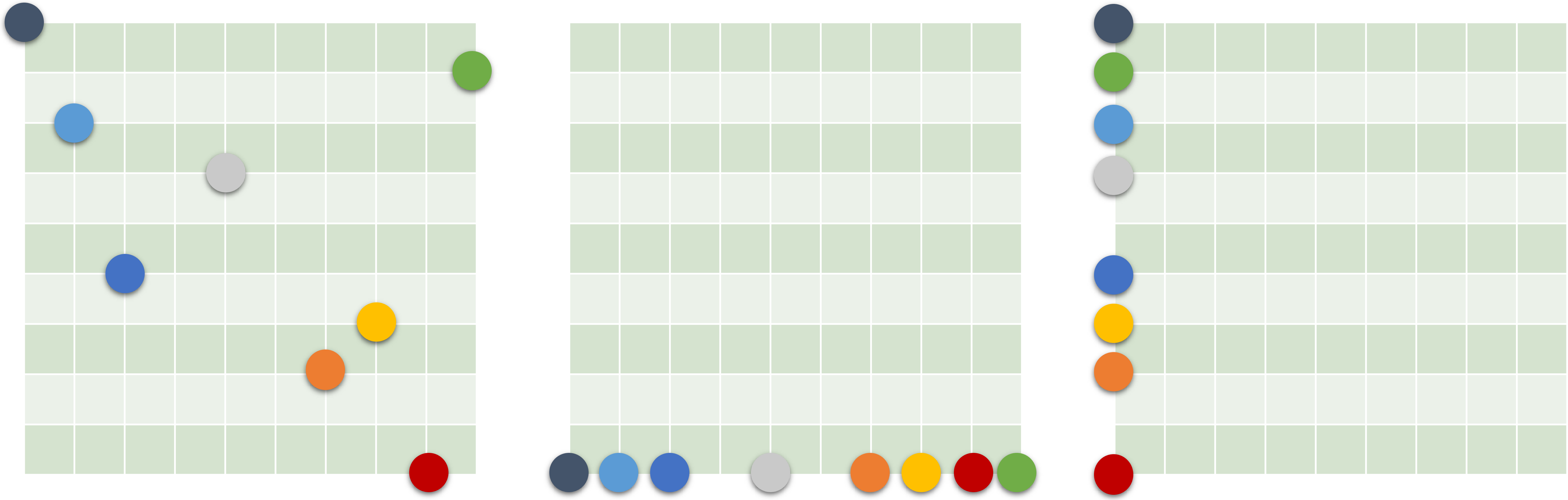
```
P  : [p0(1,10), p1(2,8), p2(7,3), p3(5,7), p4(8,4), p5(3,5), p6(10,9), p7(9,1)]
```

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)]
```



Closet 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?

API

$O(n \lg n)$

```
FUNCTION FindClosestPair(points)
    points_x = copy_and_sort_by_x(points)
    points_y = copy_and_sort_by_y(points)
    RETURN ClosestPair(points_x, points_y)
```

Preprocessing steps

Recursive procedure

Divide & Conquer

Closet 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)

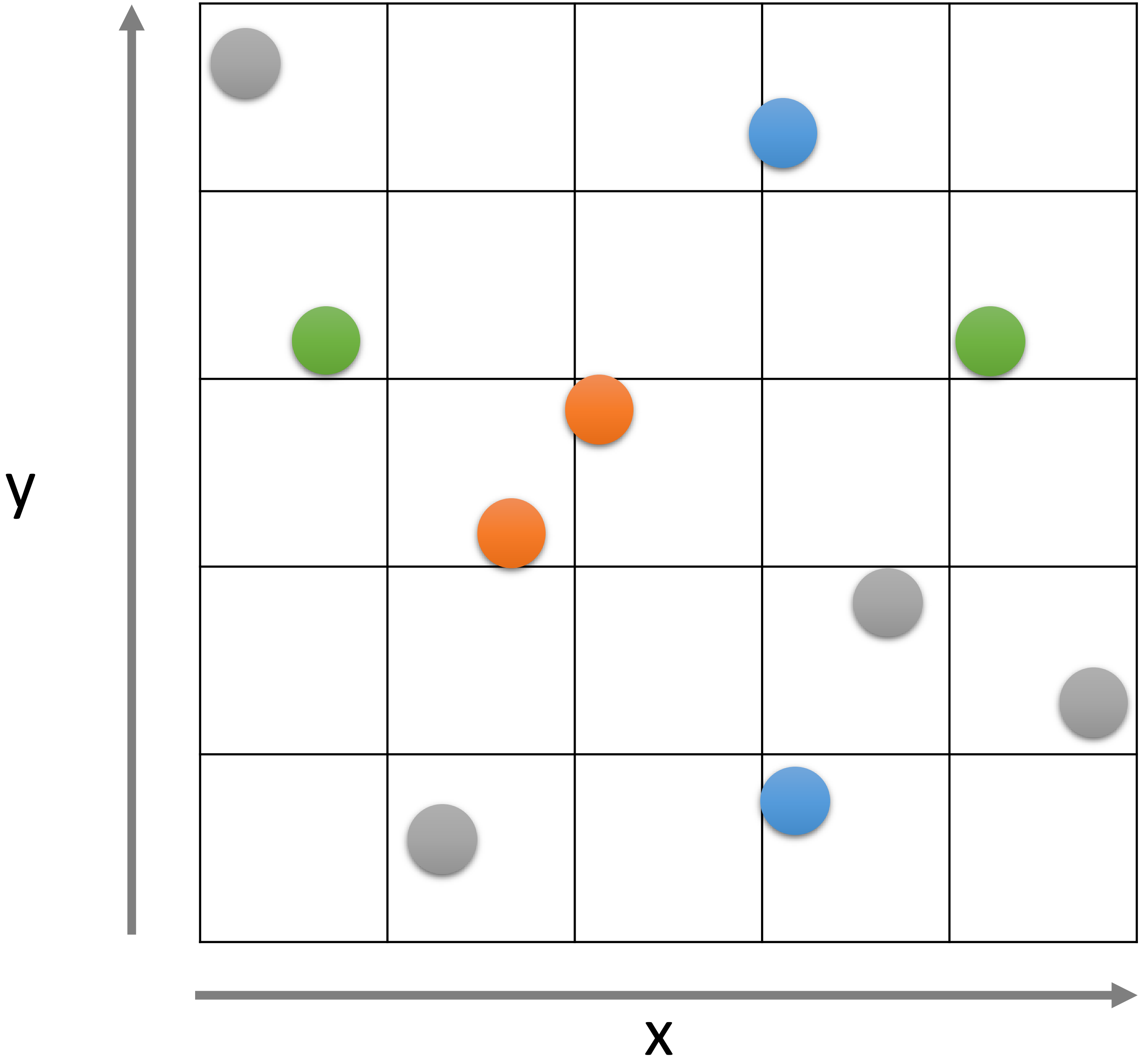
- Can we still end up with a O(n lg n) algorithm for finding the closest pair?
- Does the closeness of two points on one axis matter?

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?

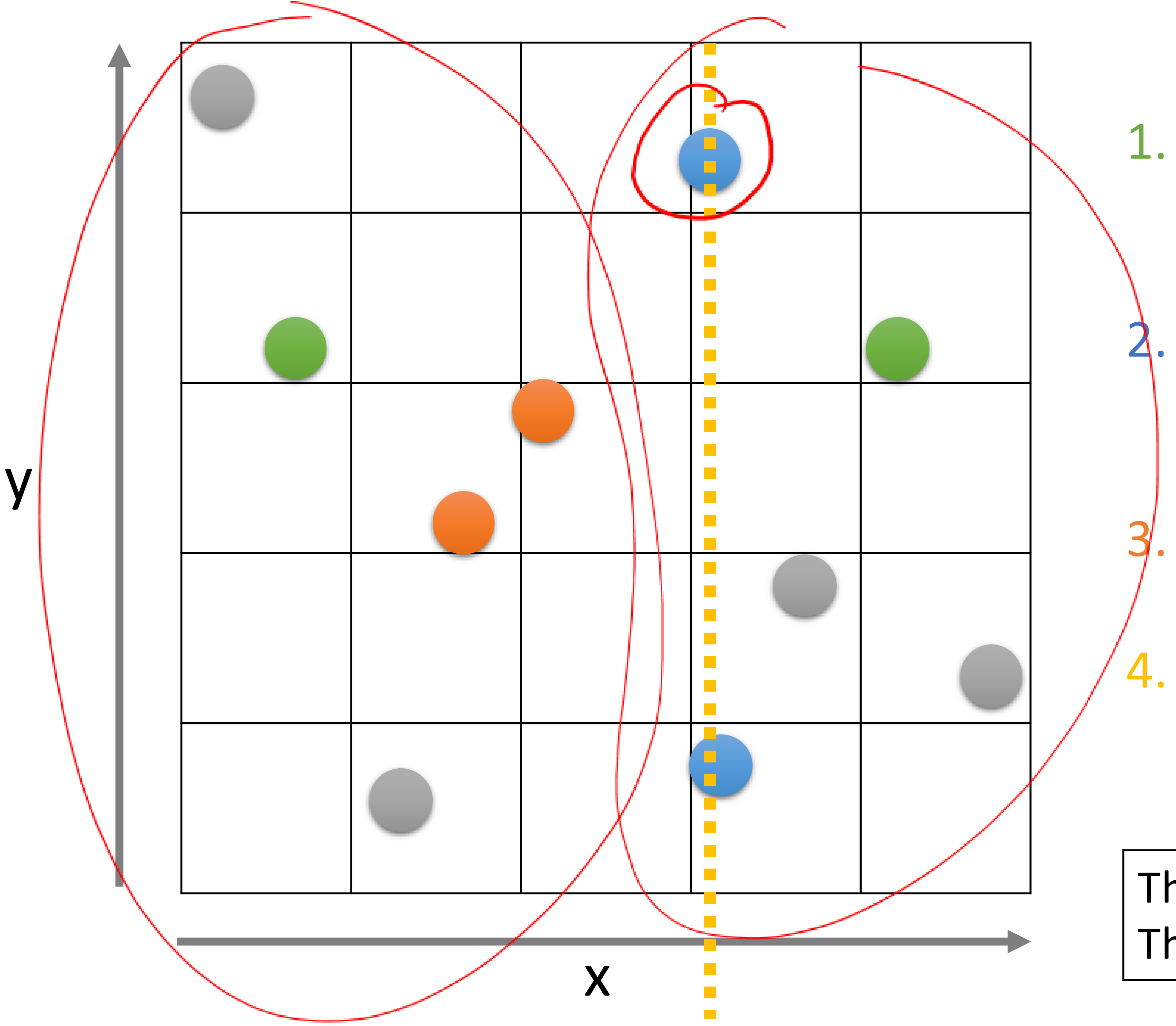


1. Which two are closest on the y-axis?

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.)



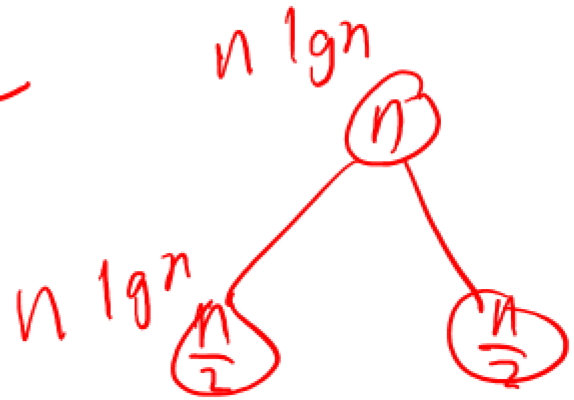
1. Which two are closest on the y-axis?
2. Which two are closest on the x-axis?
3. Which two are closest?
4. How would you divide the search space?

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

```
FUNCTION ClosestPair(px, py)
  n = px.length
  # What is the base case?
  IF n == 2
    RETURN px[0], px[1], dist(px[0], px[1])



  # What are the recursive cases?
  pl, ql, dl = ClosestPair(left_px, left_py)



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

```
FUNCTION FindClosestPair(points)
  points_x = copy_and_sort_by_x(points)
  points_y = copy_and_sort_by_y(points)
  RETURN ClosestPair(points_x, points_y)
```

API



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)]

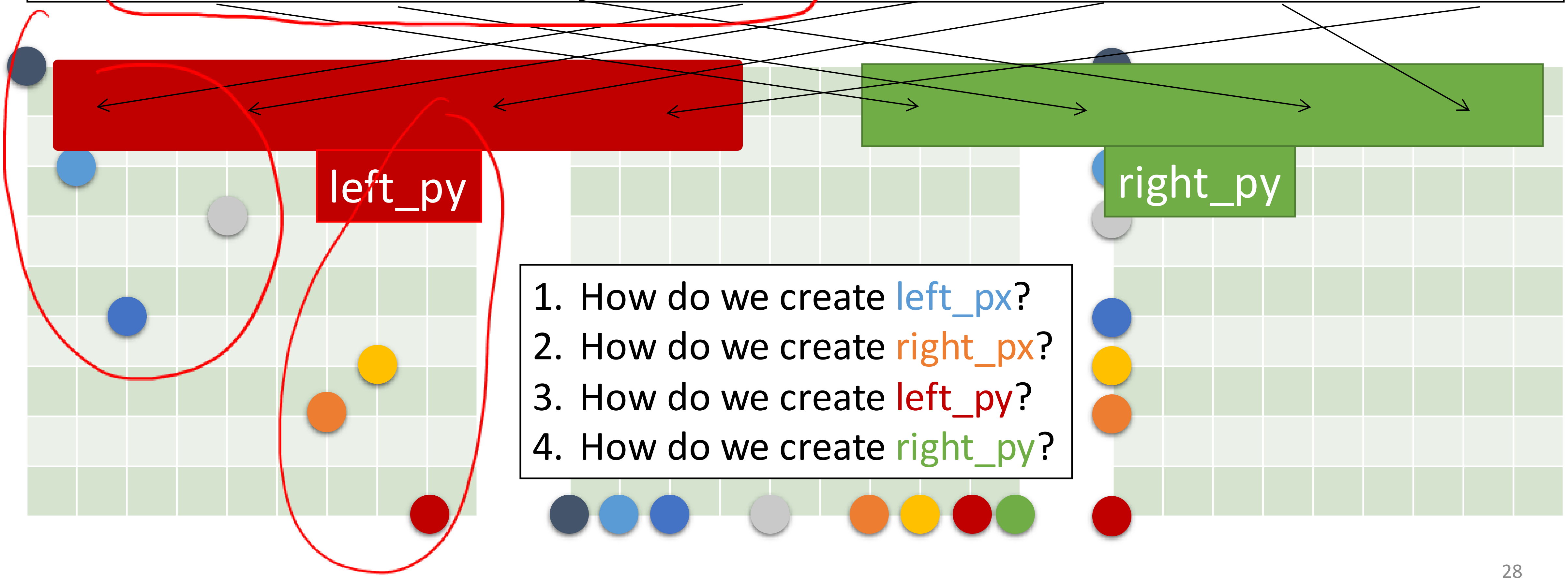
left_px   right_px

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)]

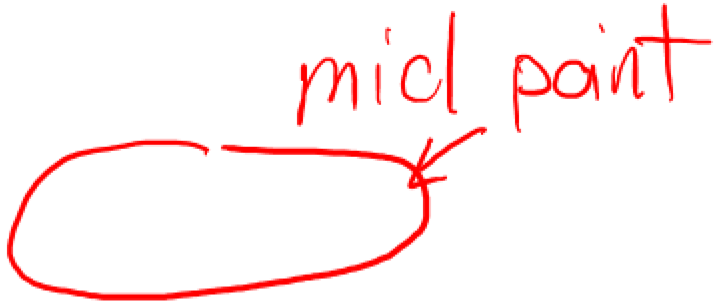
Sorted by y coordinate

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)]



1. How do we create left_px?
2. How do we create right_px?
3. How do we create left_py?
4. How do we create right_py?

```
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)
```

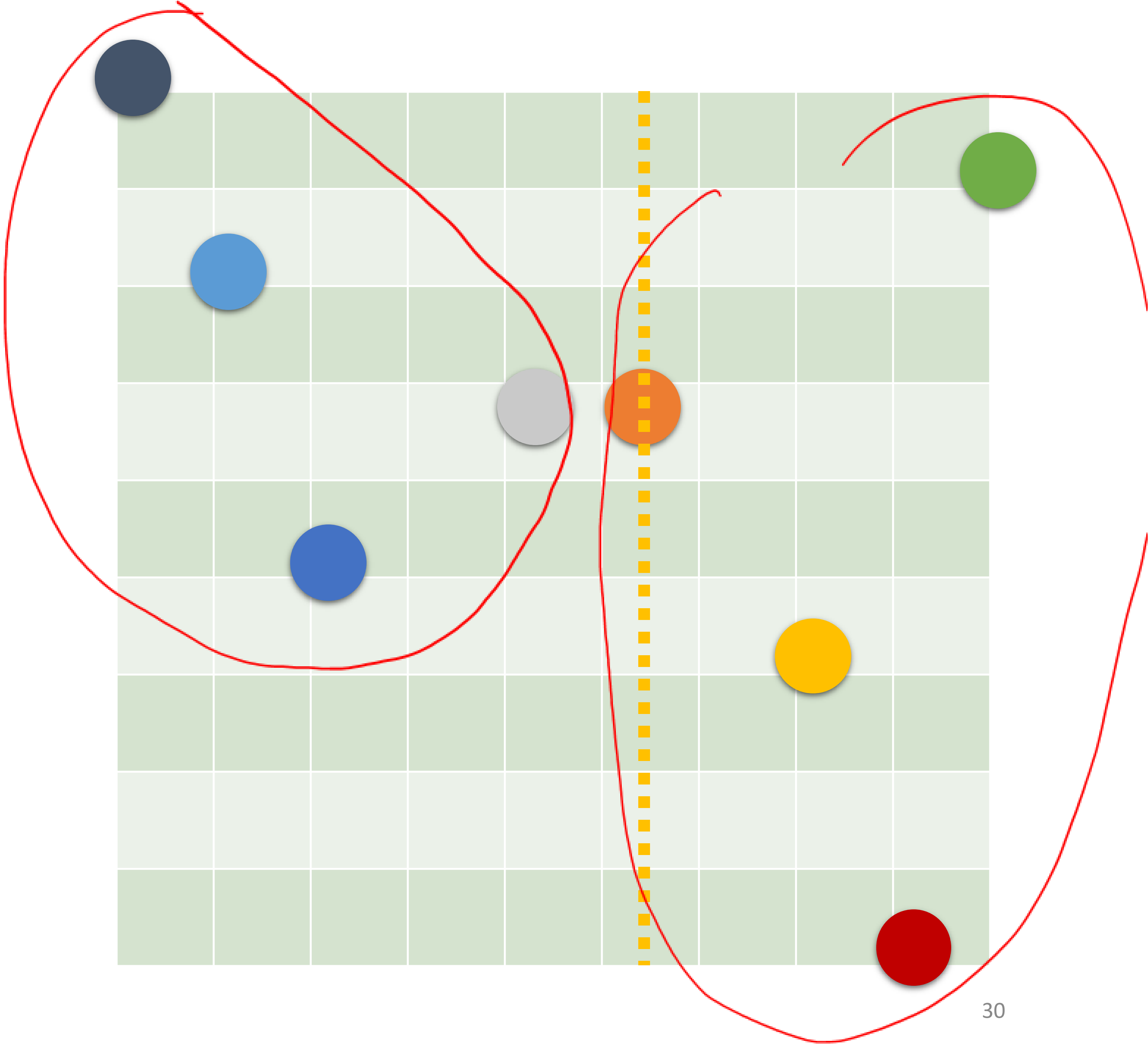


Median x value

```
left_py = []
for p in py:
if p.x < mid_point.x : left_py.append(p)
```

What is the running time of these operations?

Any problems with our current approach?



```
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)
```

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)
```

O(n lg n)

O(lg n)

O(1)

O(n lg n), O(n)

Merge Sort and It's Recurrence

Func MS(array) ← n

Base case

sort left $\frac{n}{2}$

sort right $\frac{n}{2}$

merge (left, right)

return

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = O(n \lg n)$$

```
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
```

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

```
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)
```

O(n)

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)    t_px = px[0 ..< n//2]
O(n)    t_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)    ht_px = px[n//2 ..< n]
O(n)    ht_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)    TURN Closest(pl, ql, dl, pr, qr, dr, ps, qs, ds)
```

$$T(n) = 2\,T(n/2) + O(n) = O(n \lg n)$$

```
T(n)      FUNCTION MergeSort(array)
  O(1) = array.length
  O(1)  n == 1
  O(1) RETURN array

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

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

  O(1) ETURN array_sorted
```

```
T(n) = 2 T(n/2) + O(n)
     = O(n lg n)
```

$$T(n) = 2\,T(n/2) + O(n)$$
$$= O(n \lg n)$$

Recurrence

```
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) ne = RecursiveFunction(some_input.first_half)
T(n/2) vo = RecursiveFunction(some_input.second_half)

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

O(1)   ETURN 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(pl, ql), d(pr, qr)]
```

```
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)?

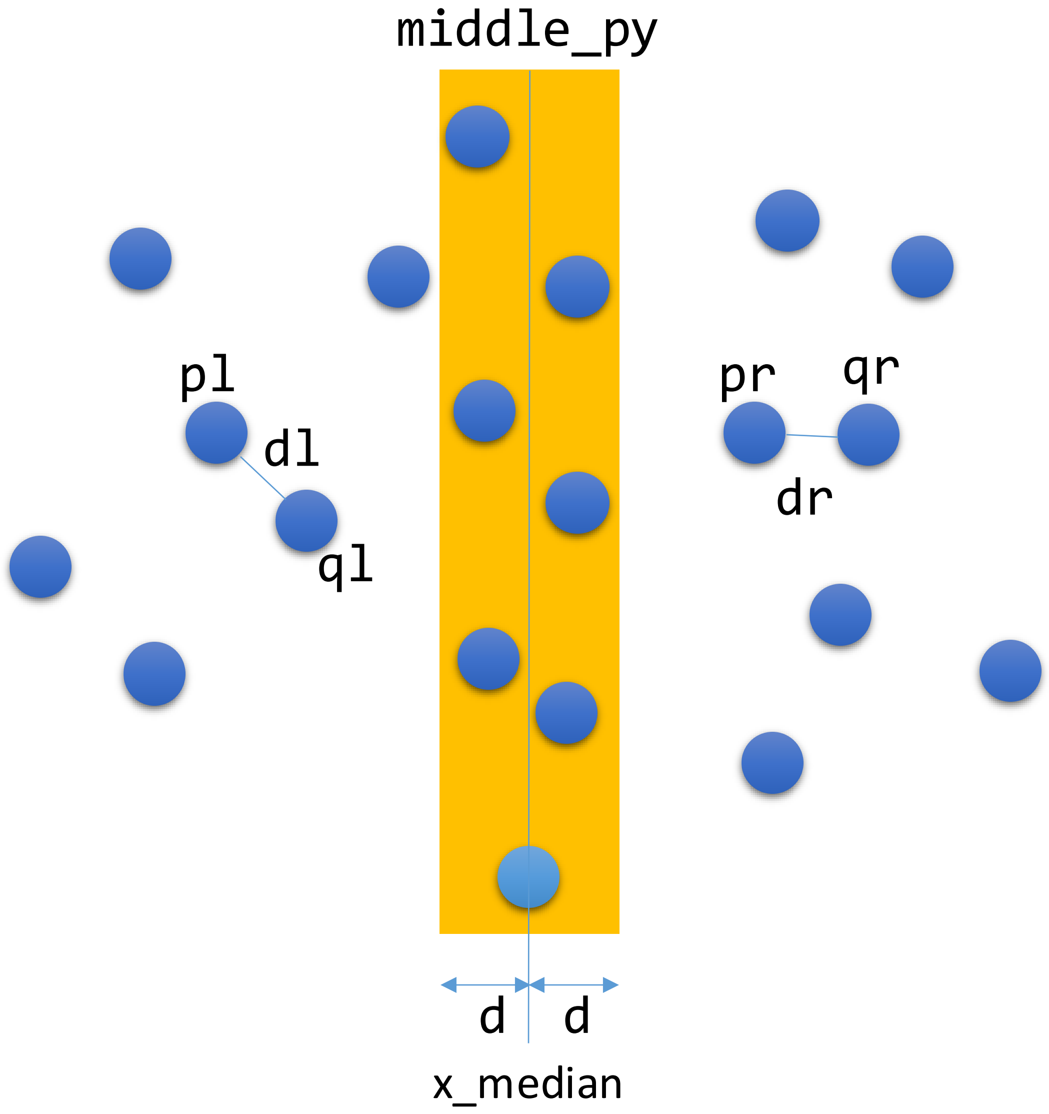
```
FUNCTION ClosestSplitPair(px, py, d)
   n = px.length
   x_median = px[n//2].x
   middle_py = [p FOR p IN py IF x_median - d < p.x < x_median + d]

   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

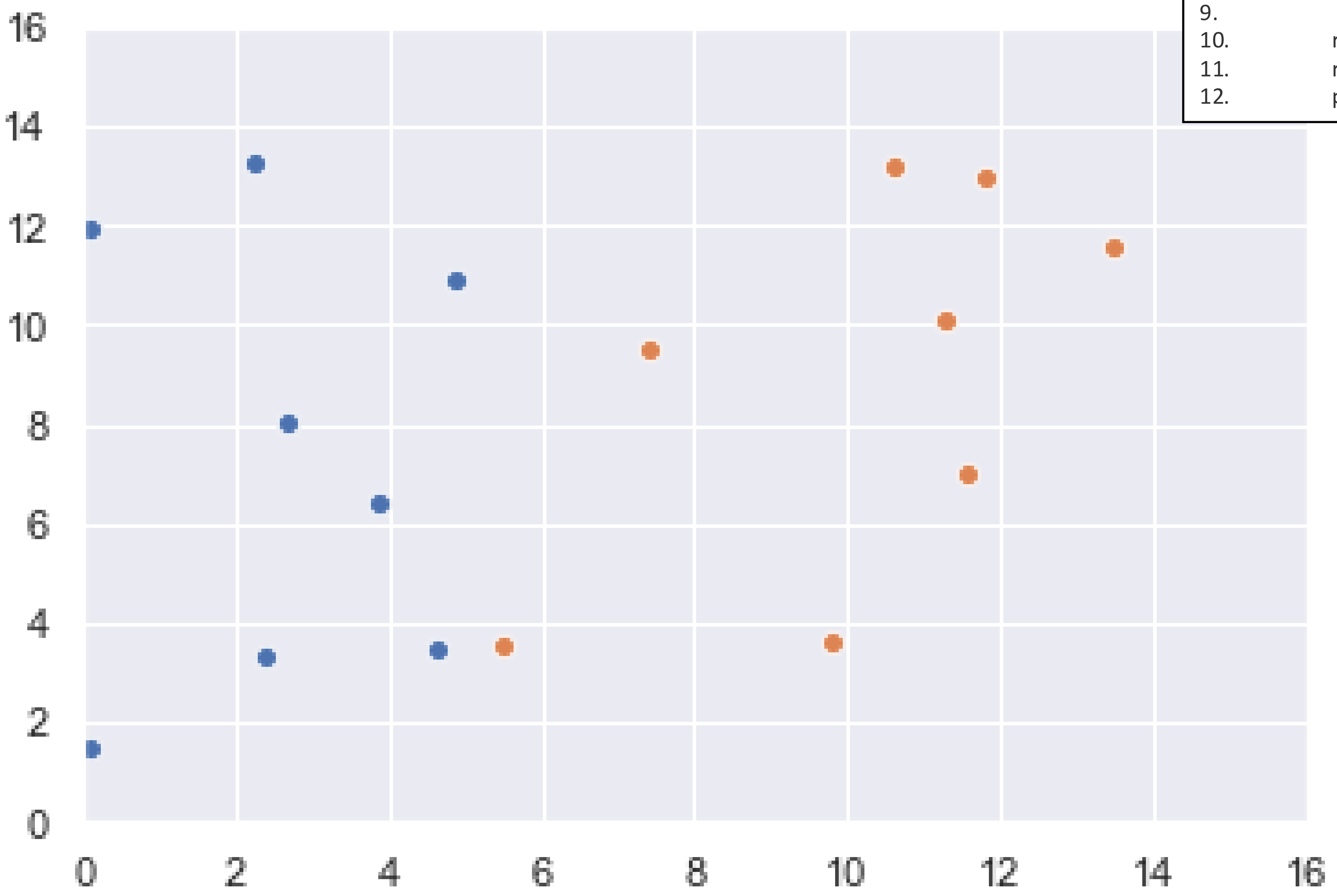
```
FOR j IN [1 ..= min(7, middle_py.length - i)]
   p = middle_py[i], q = middle_py[i + j]
   IF dist(p, q) < closest_d
      closest_d = dist(p, q)
      closest_p = p, closest_q = q
```

```
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]

…
```

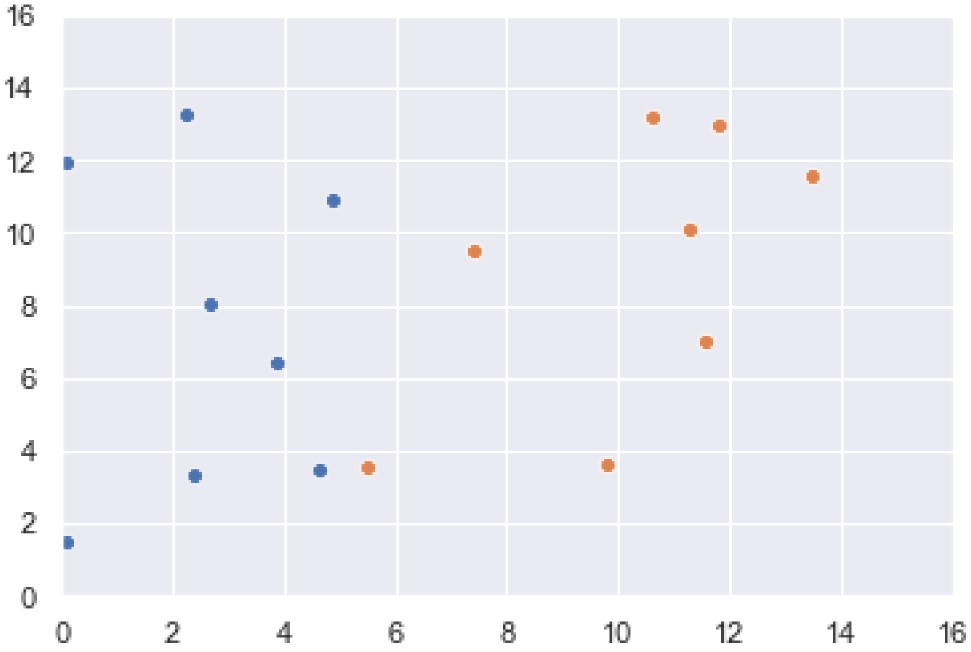
```
FUNCTION ClosestSplitPair(px, py, d)
  n = px.length
  x_median = px[n//2].x
  middle_py = [p FOR p IN py
          IF x_median - d < p.x < x_median + d]

  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
```



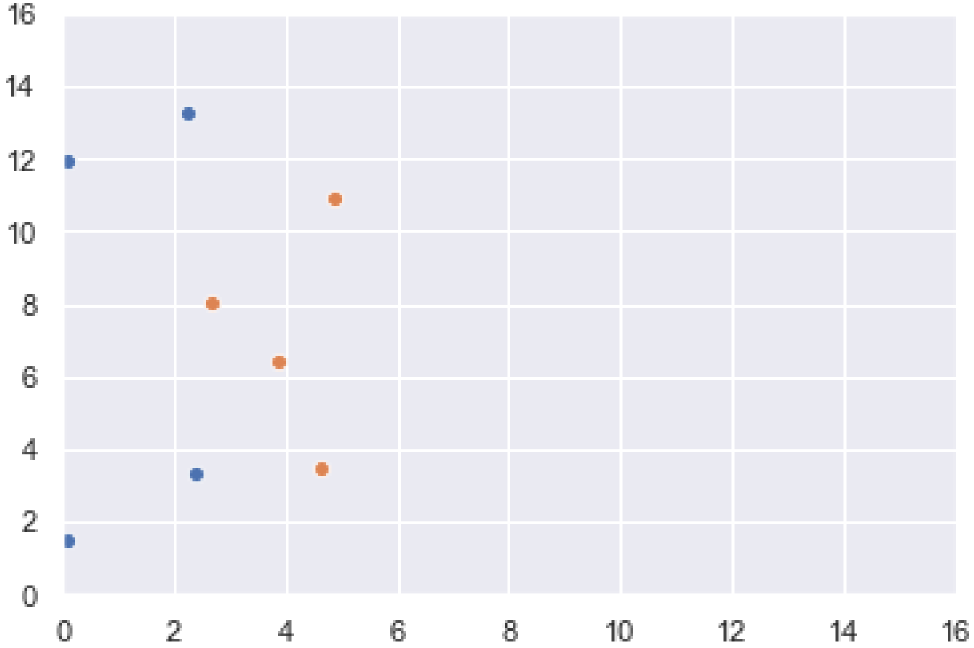
Example execution.



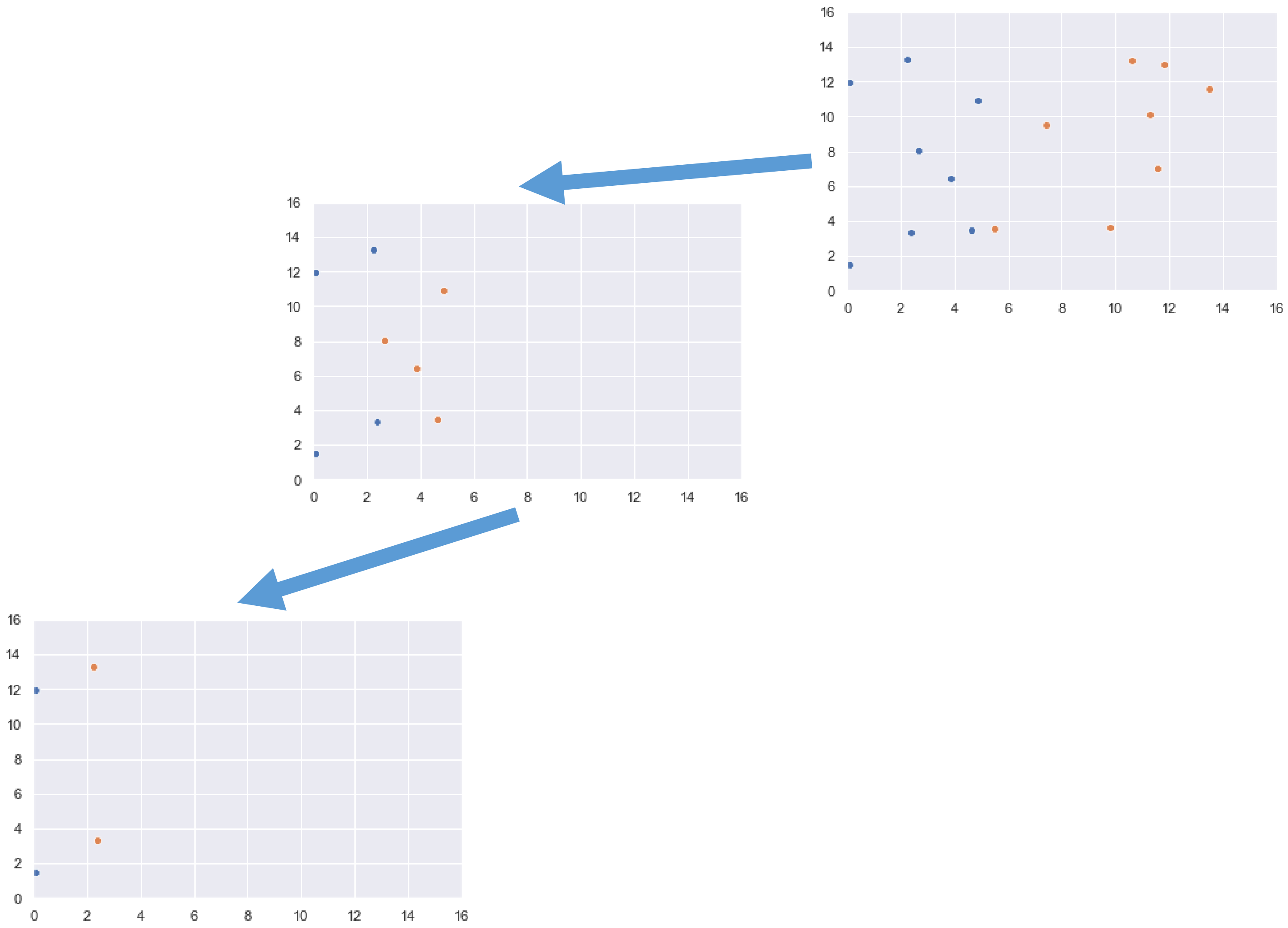

```
FUNCTION ClosestPair(px, py)
  n = px.length
  IF n == 2
    RETURN px[0], px[1], dist(px[0], px[1])

  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)
```



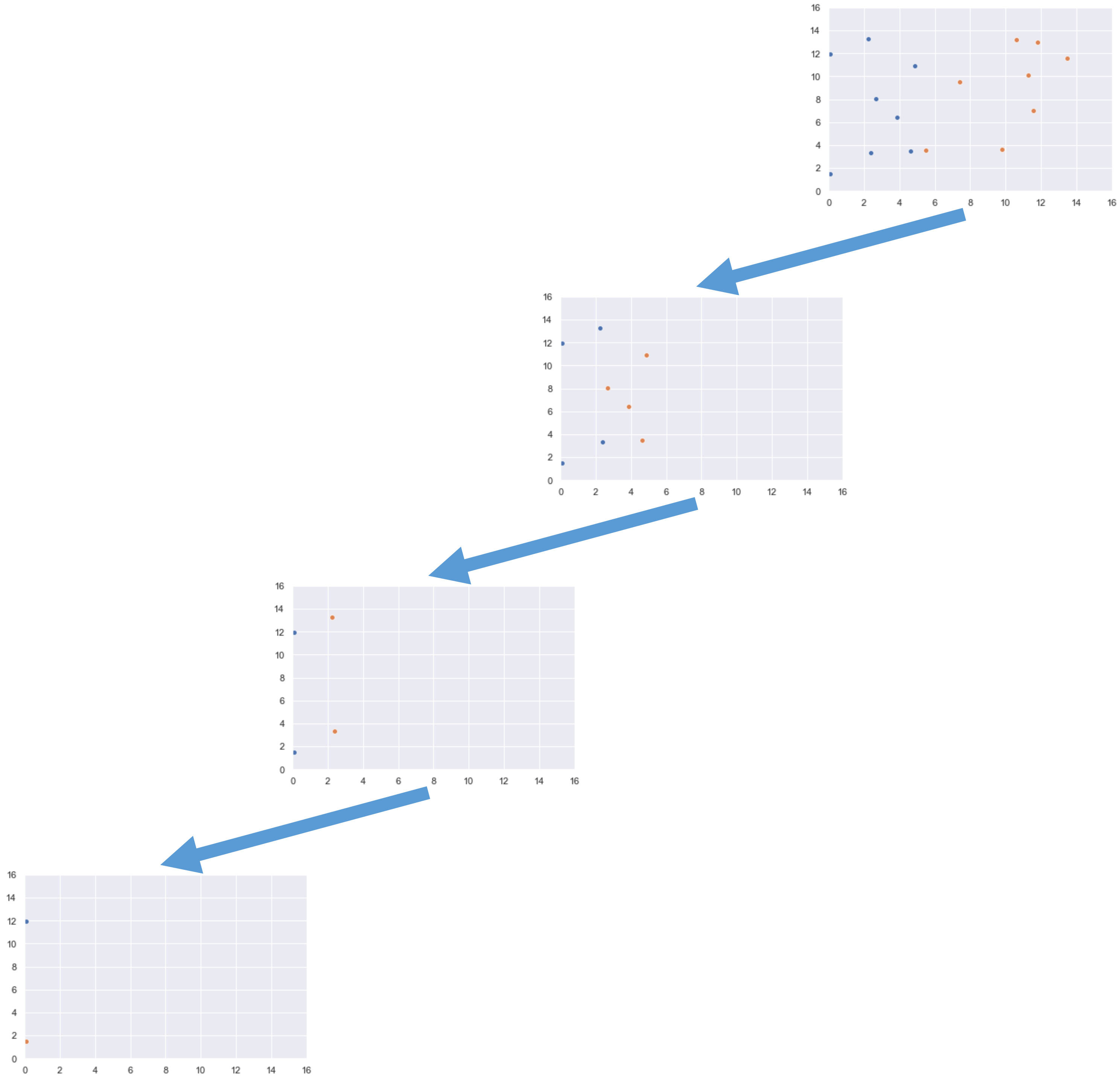
16
14
12
10
8
6
4
2
0
0 2 4 6 8 10 12 14 16

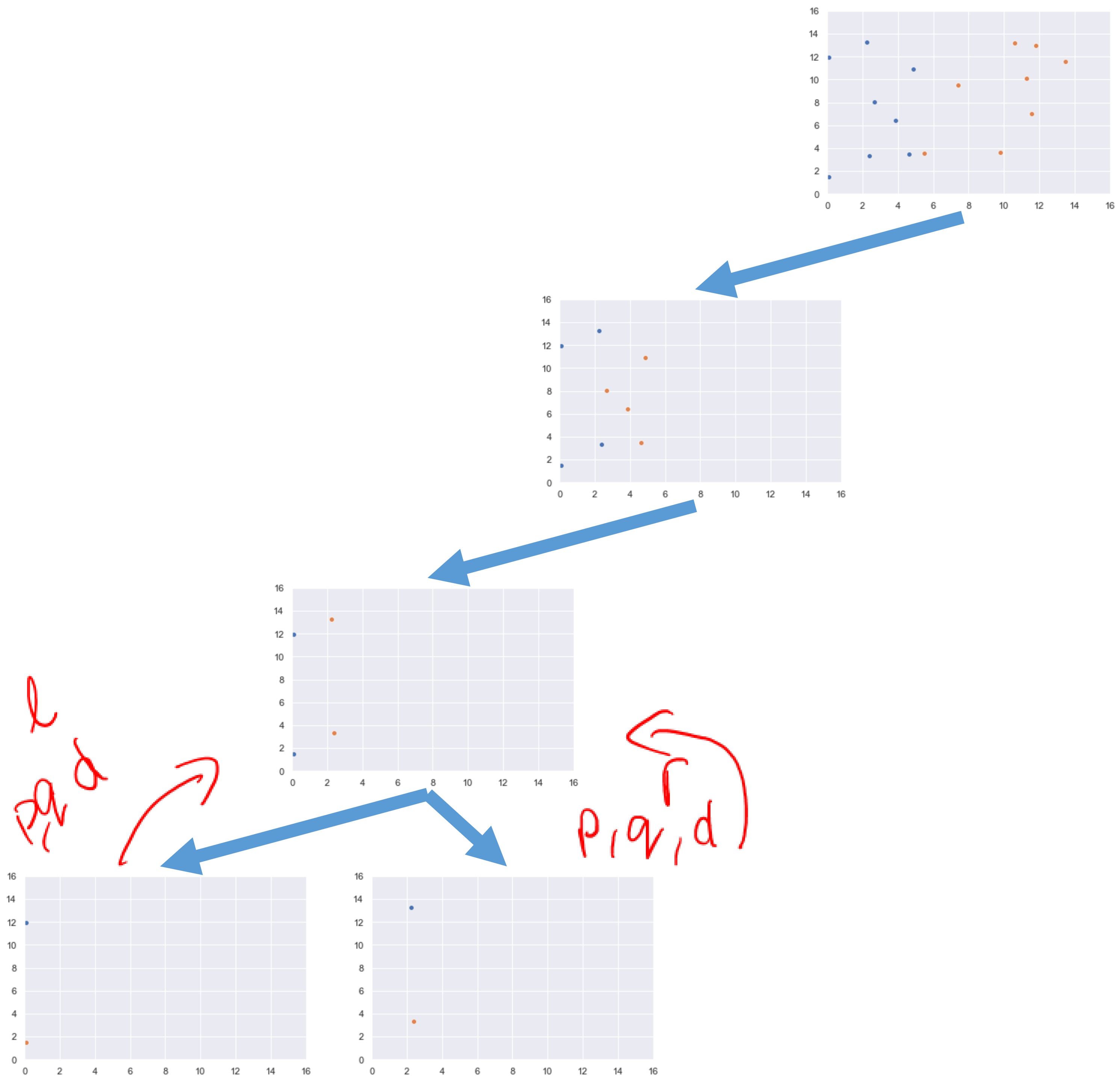


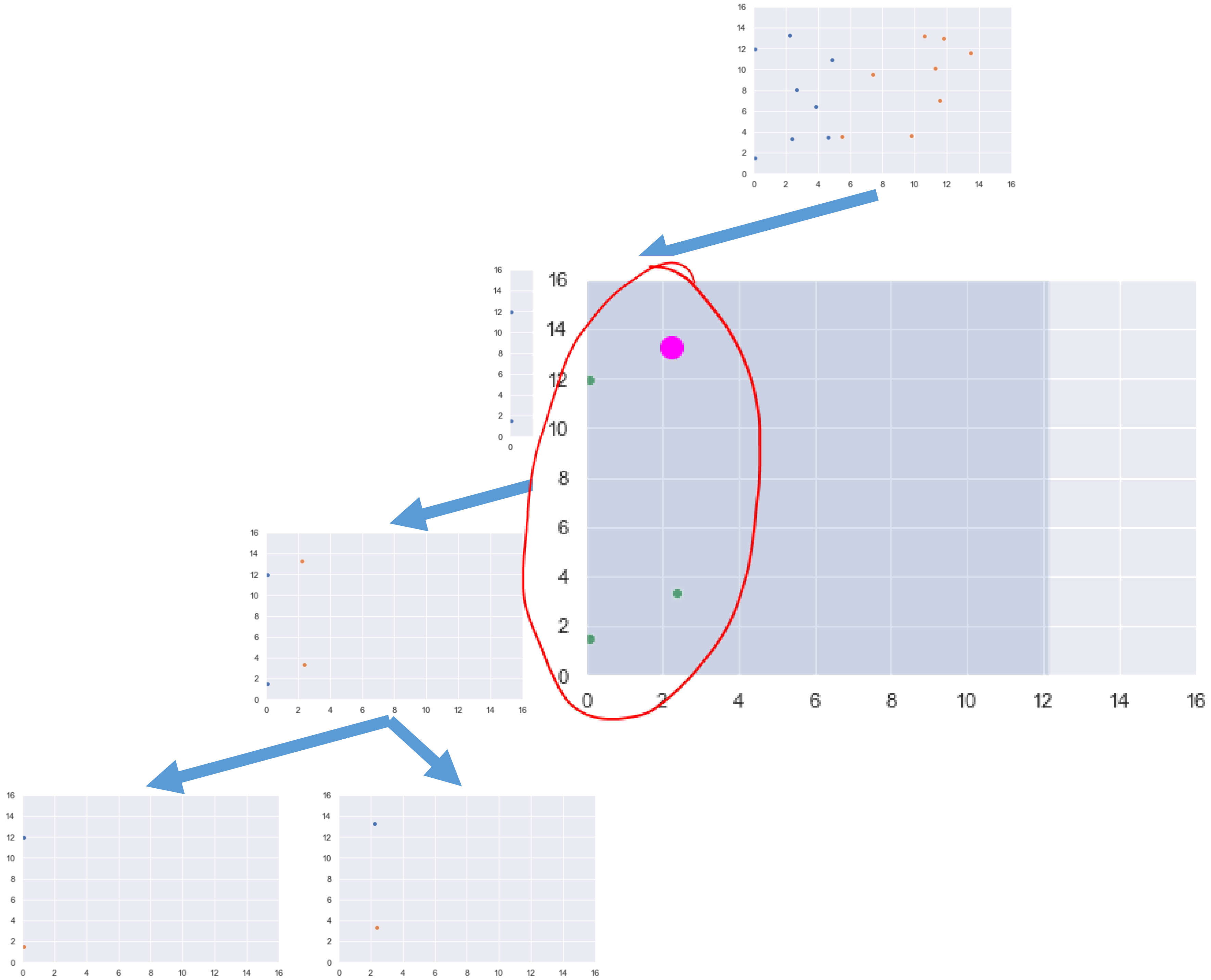
16
14
12
10
8
6
4
2
0
0 2 4 6 8 10 12 14 16



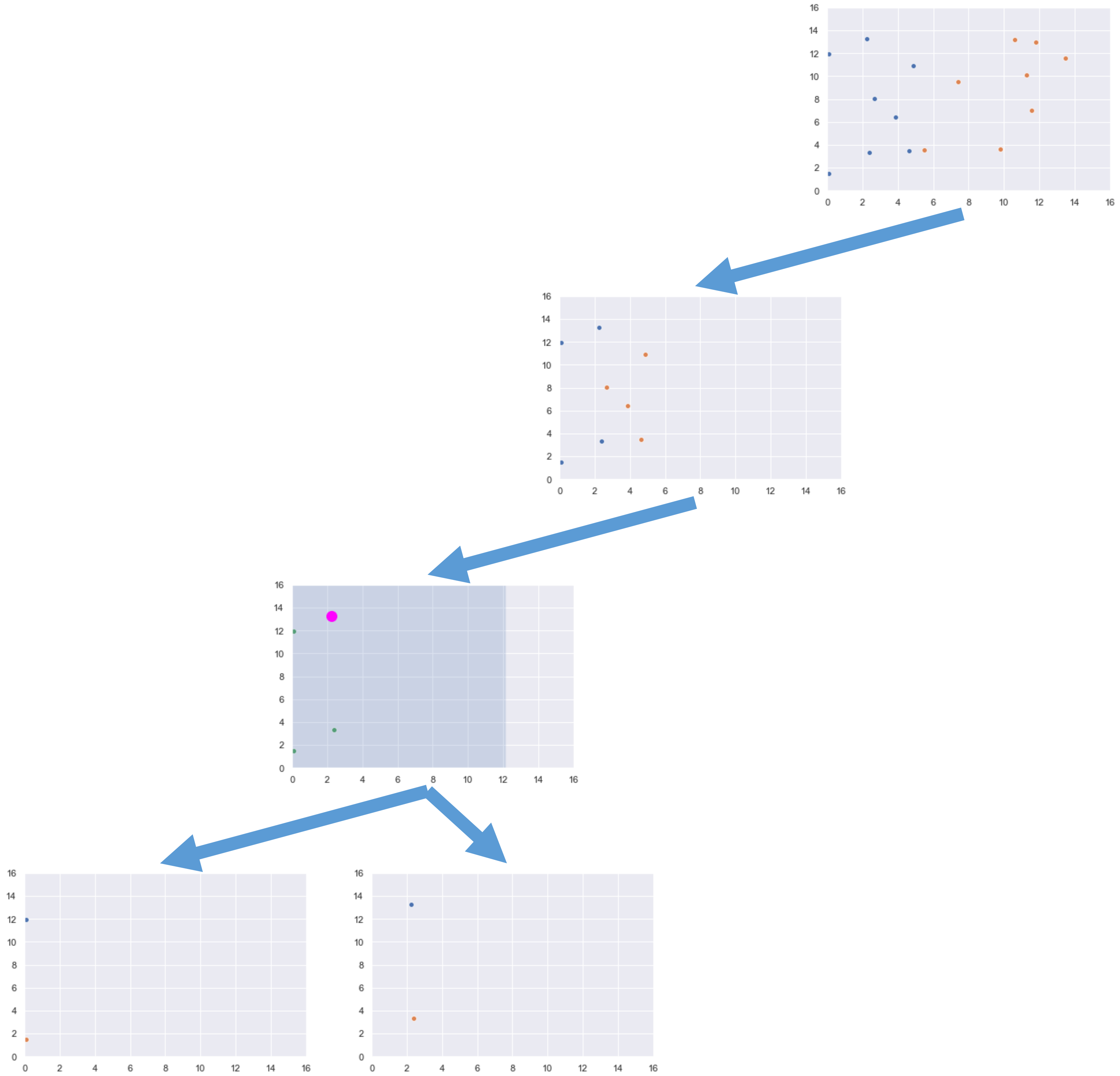
0
2
4
6
8
10
12
14
16

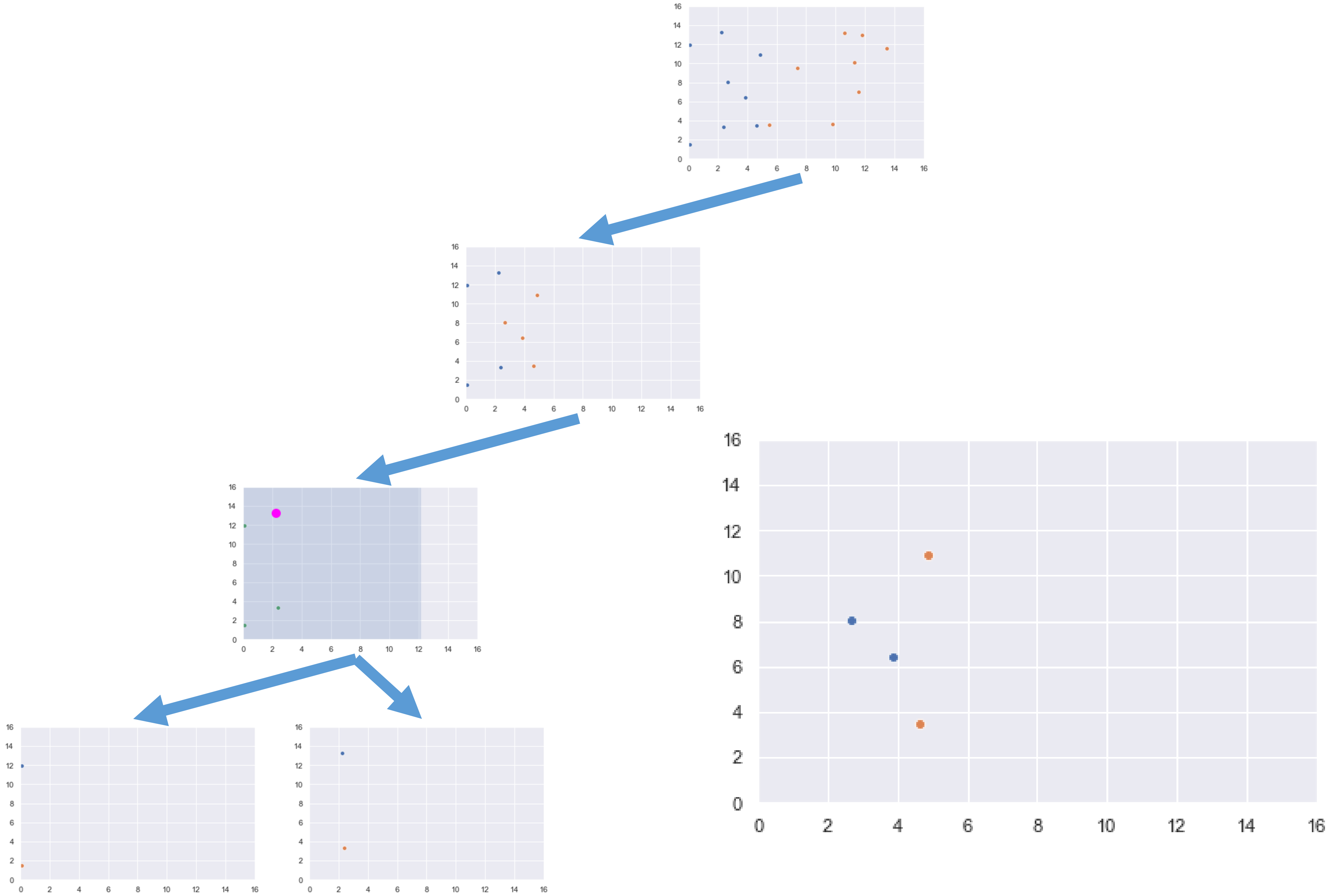


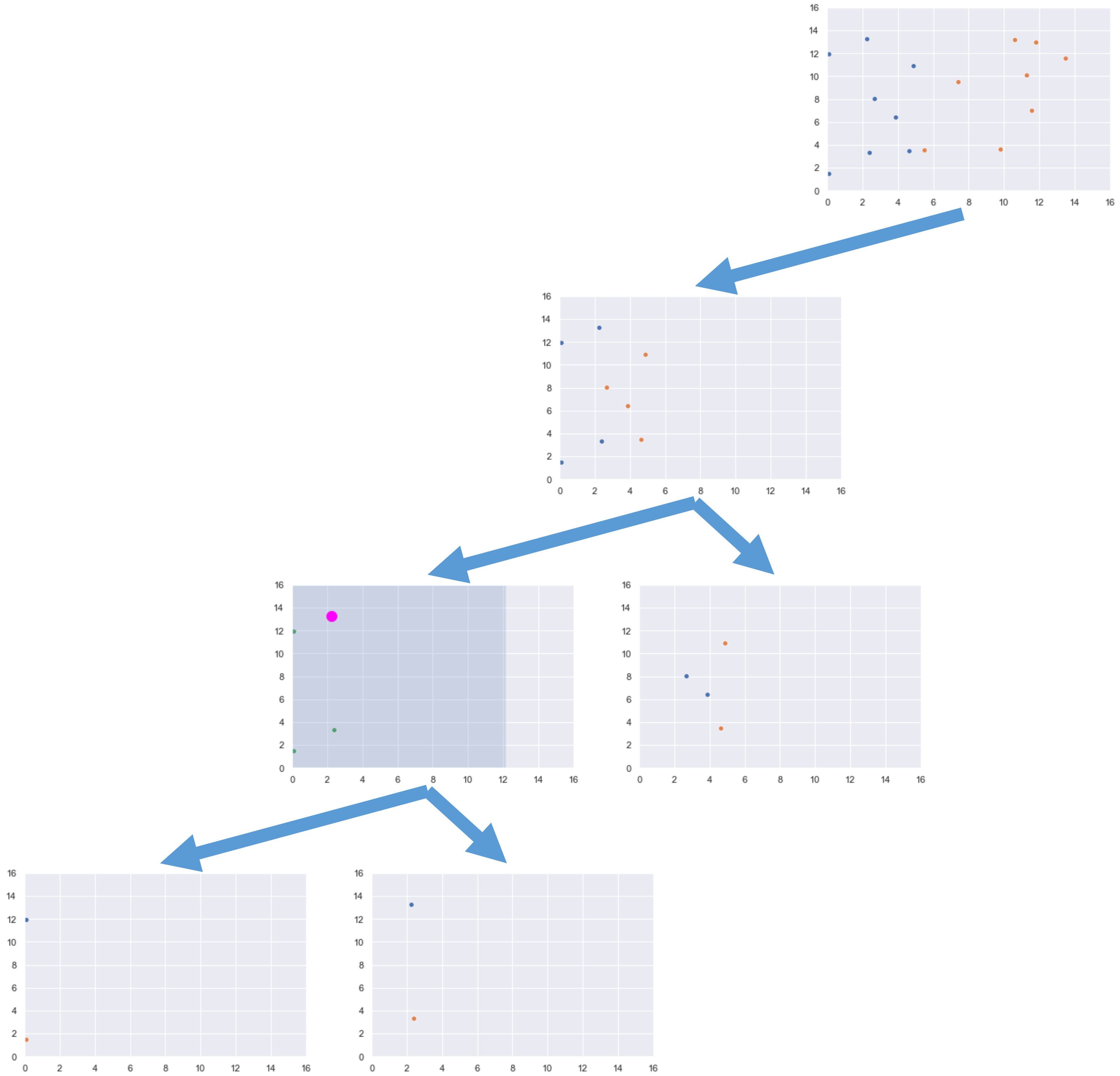

l
p, q, d
r
p, q, d

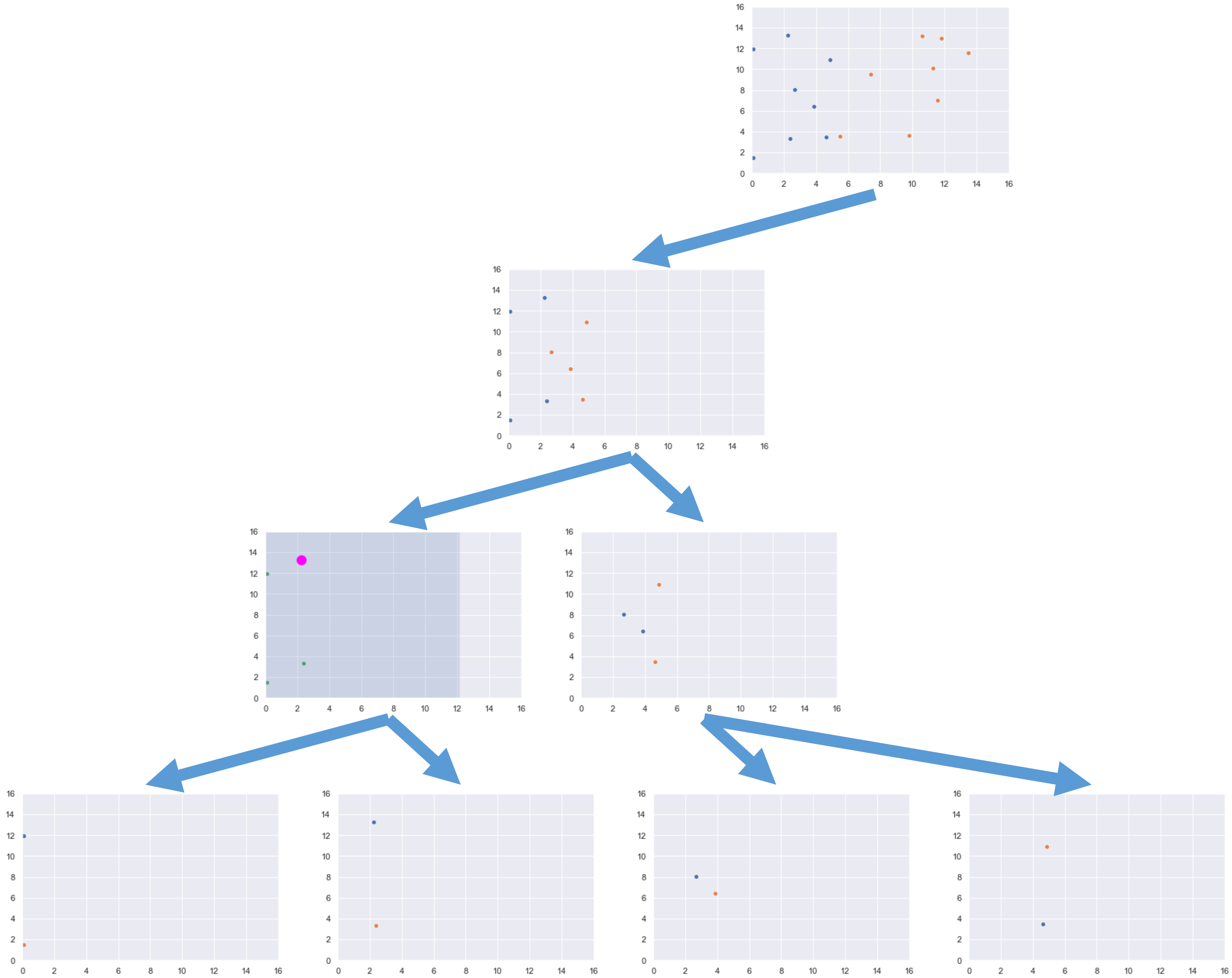


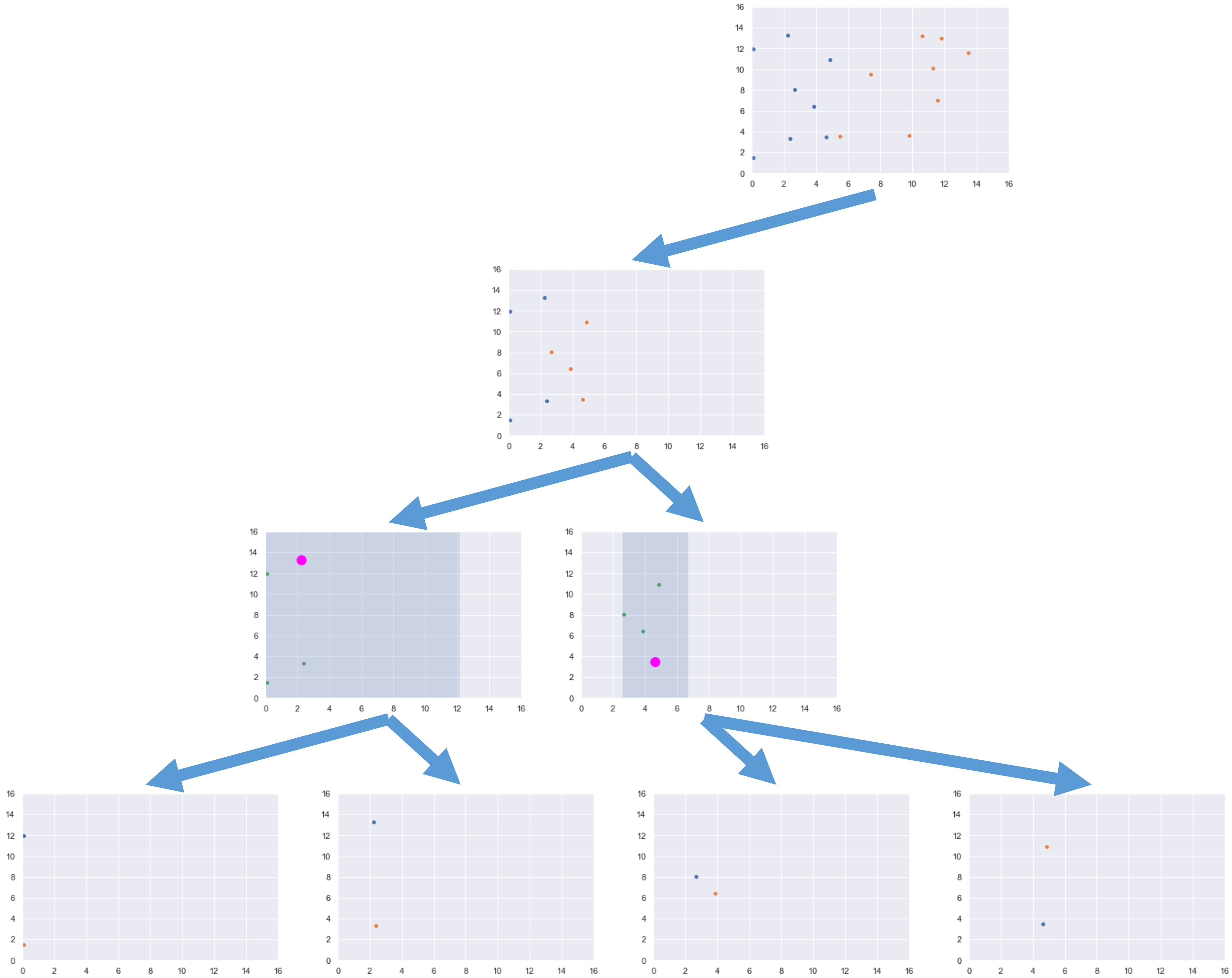
Closest Split Pair

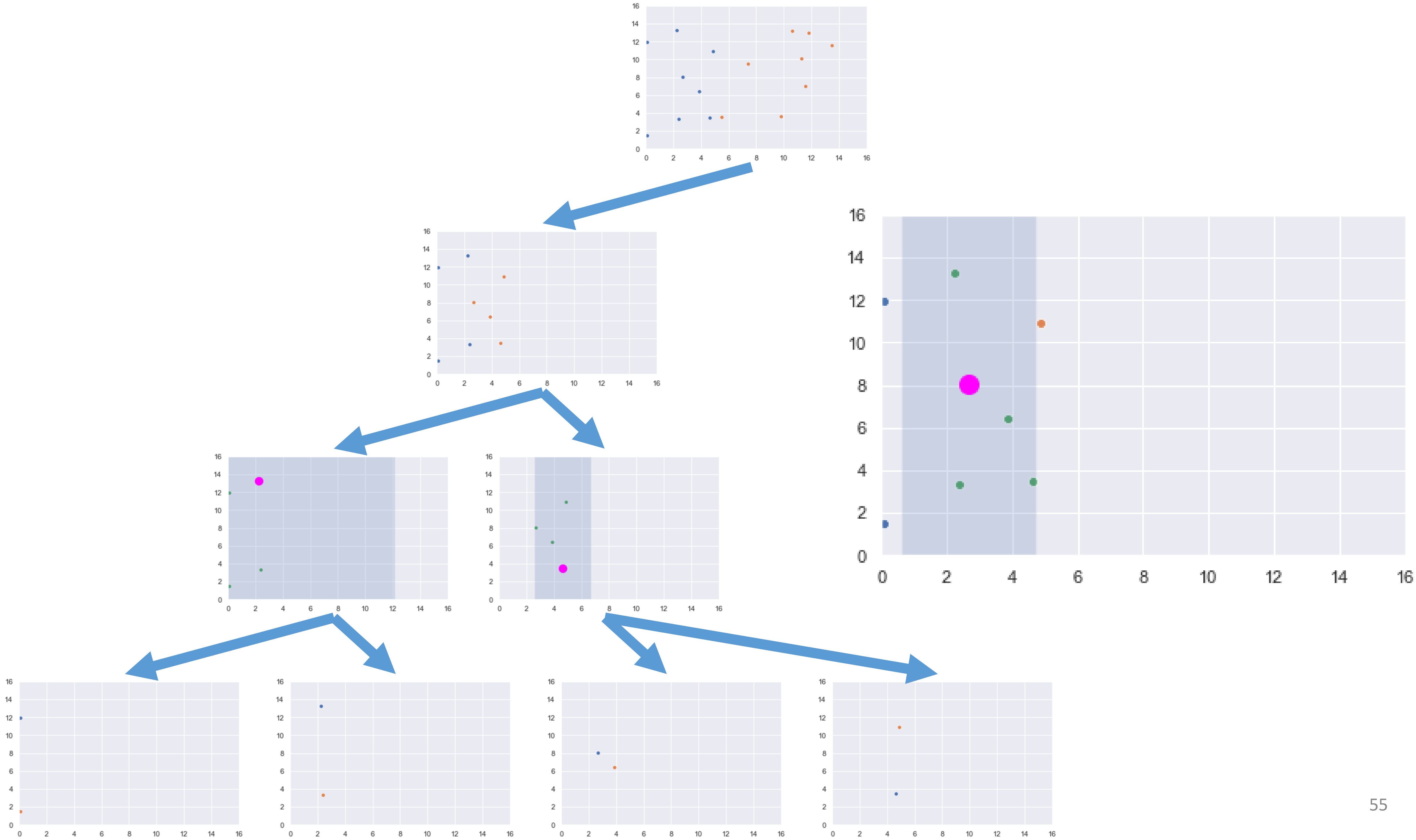


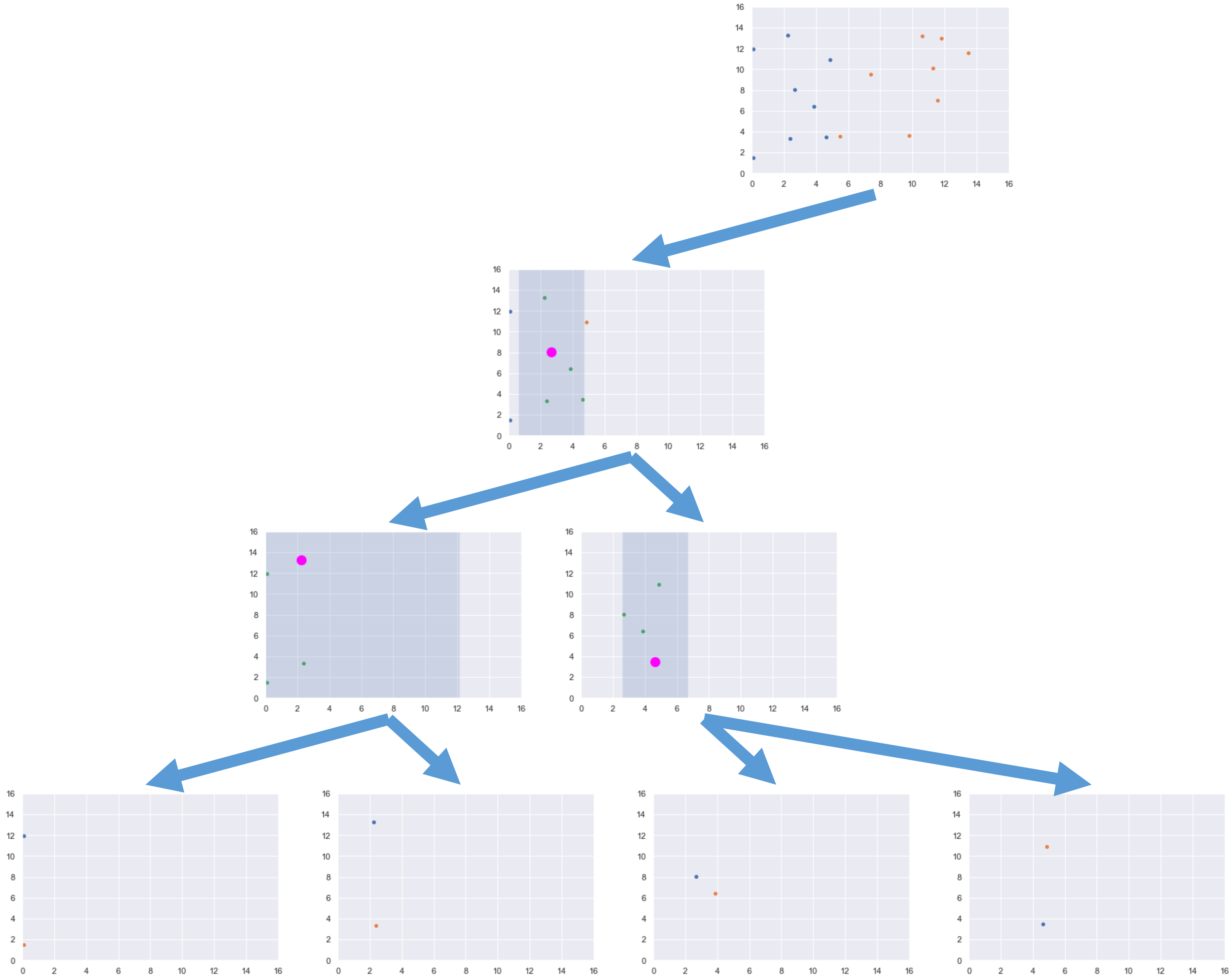


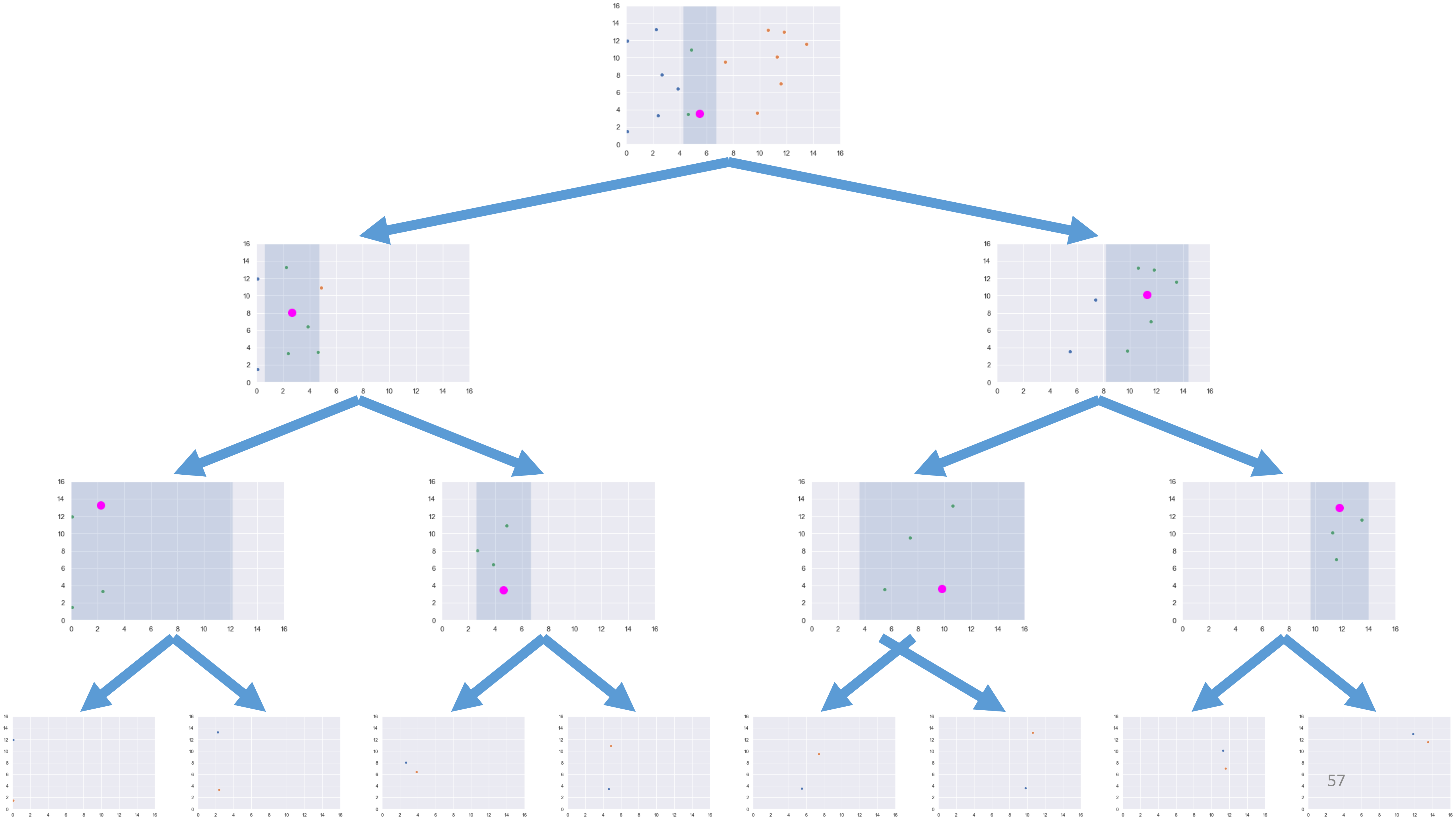












Theorem for correctness of ClosestPair

Theorem:

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

ClosestPair finds the closest pair

Let p ∈ `left`, q ∈ `right` be a split pair with `d(p, q) < d`

Then

A. p and q ∈ `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 $\in$ `left`, q $\in$ `right` be a split pair with `d(p, q) < d`

Then

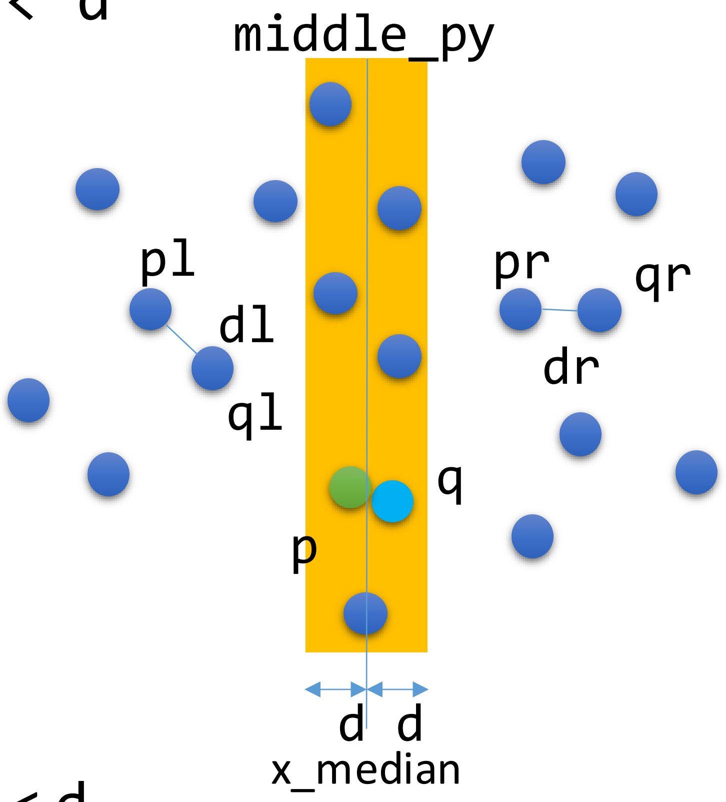
A. p and q ∈ `middle_py`, and

If p = (x1,y1) ∈ left AND q = (x2,y2) ∈ right AND d(p,q) < d

Then

```
x_median - d < x1 ≤ x_median and
x_median       ≤ x2 < x_median + d
```

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



Proof—Part A

Let p ∈ `left`, q ∈ `right` be a split pair with `d(p, q) < d`

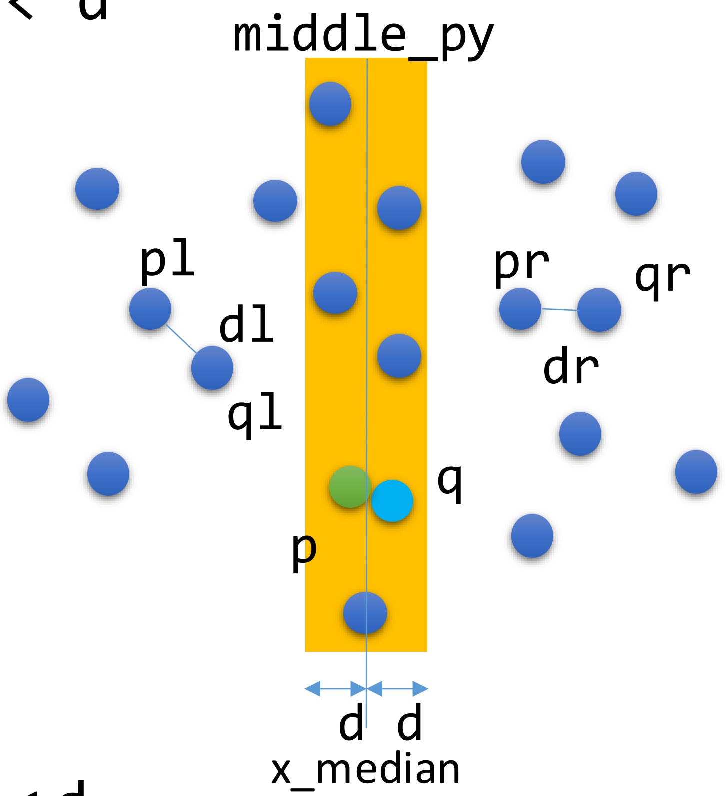
Then

> A. p and q ∈ `middle_py`, and

If p = (x1,y1) ∈ left <u>AND</u> q = (x2,y2) ∈ right <u>AND</u> d(p,q) < d

Then

```
x_median - d < x1 ≤ x_median and
x_median     ≤ x2 < x_median + d
```



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

ClosestPair finds the closest pair

Let p `∈ left, q ∈ right` be a split pair with `d(p, q) < d`

Then

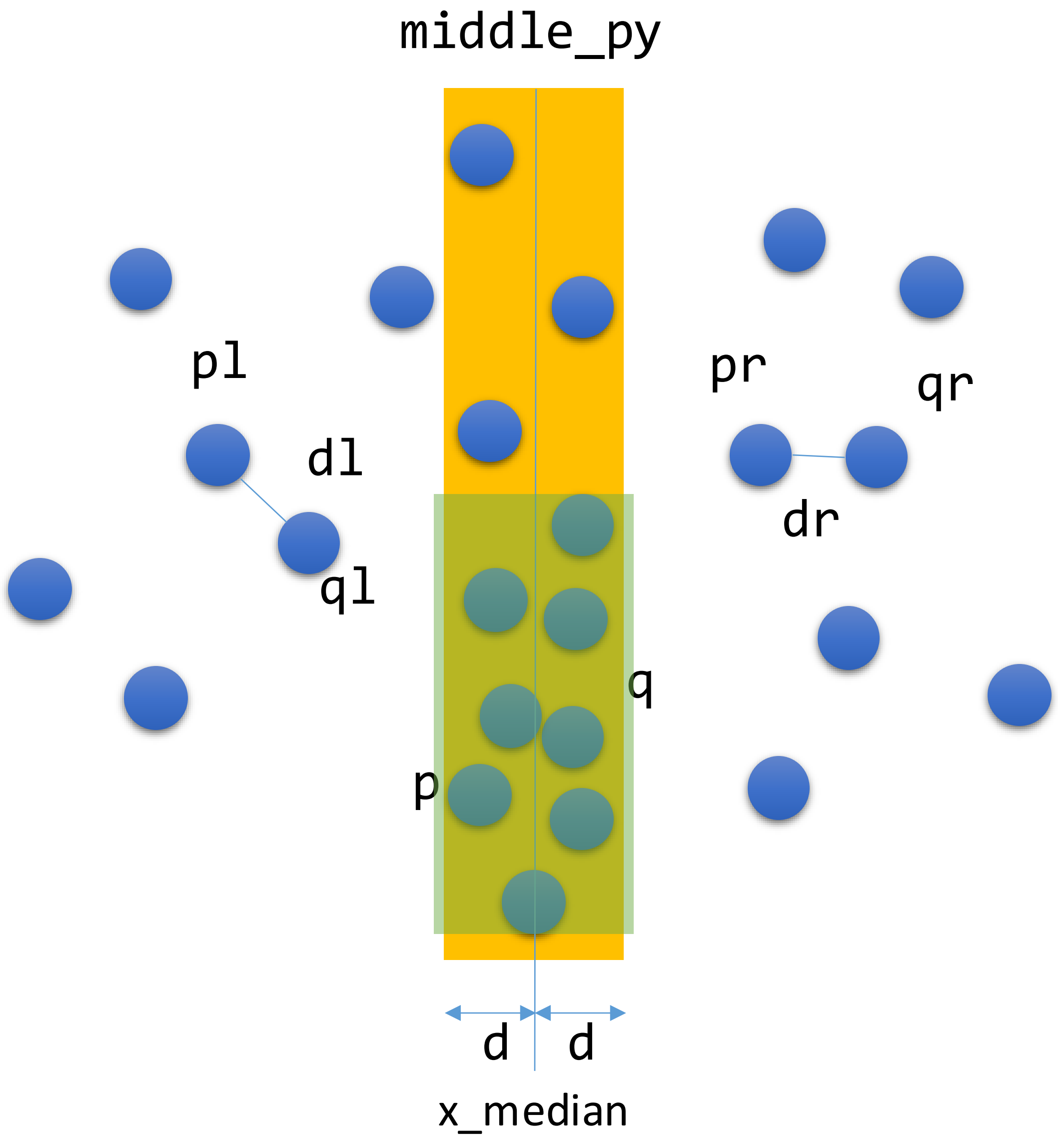
A. p and q ∈ `middle_py`, and

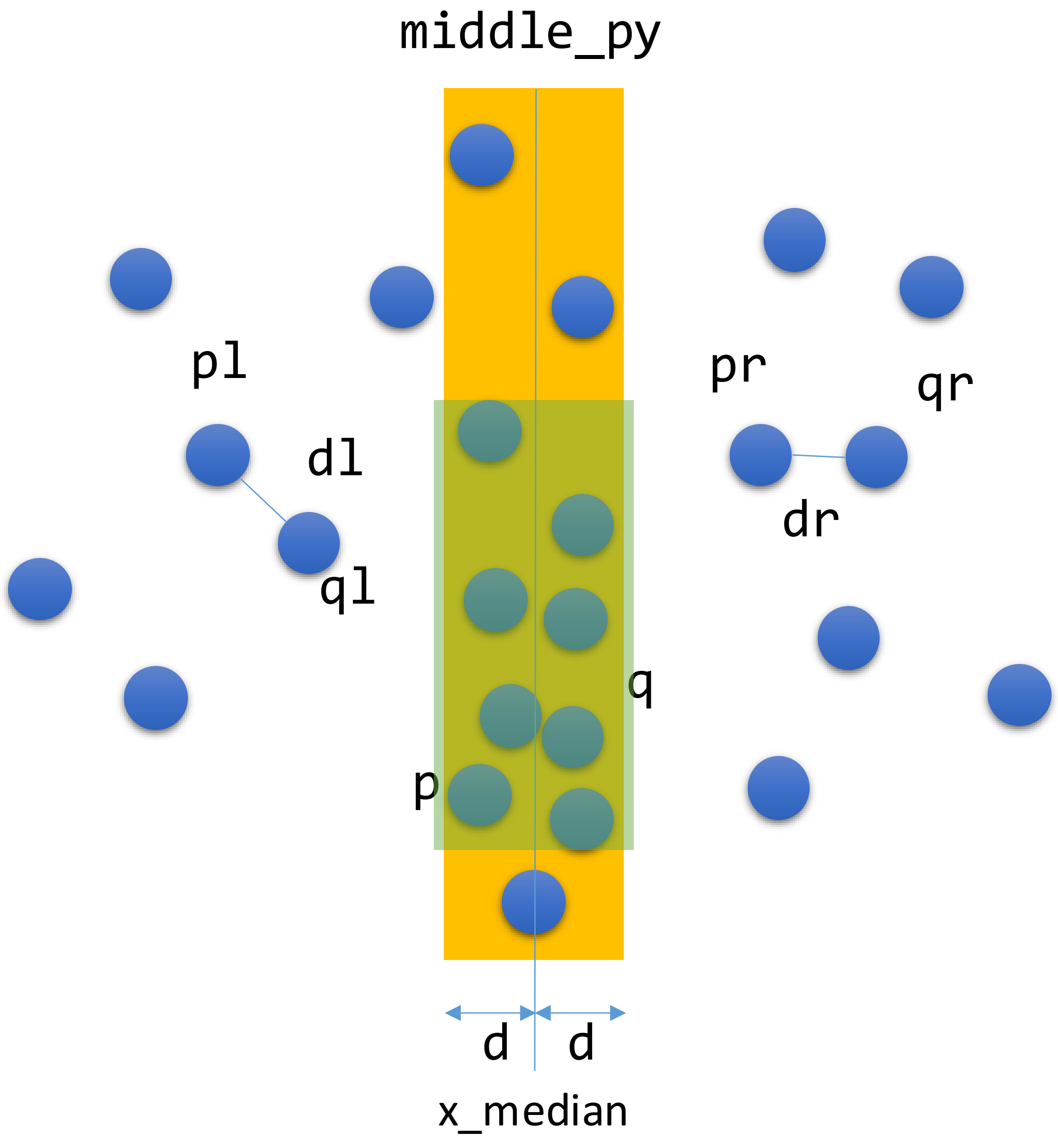
B. p and q are at most **7** positions apart in `middle_py`

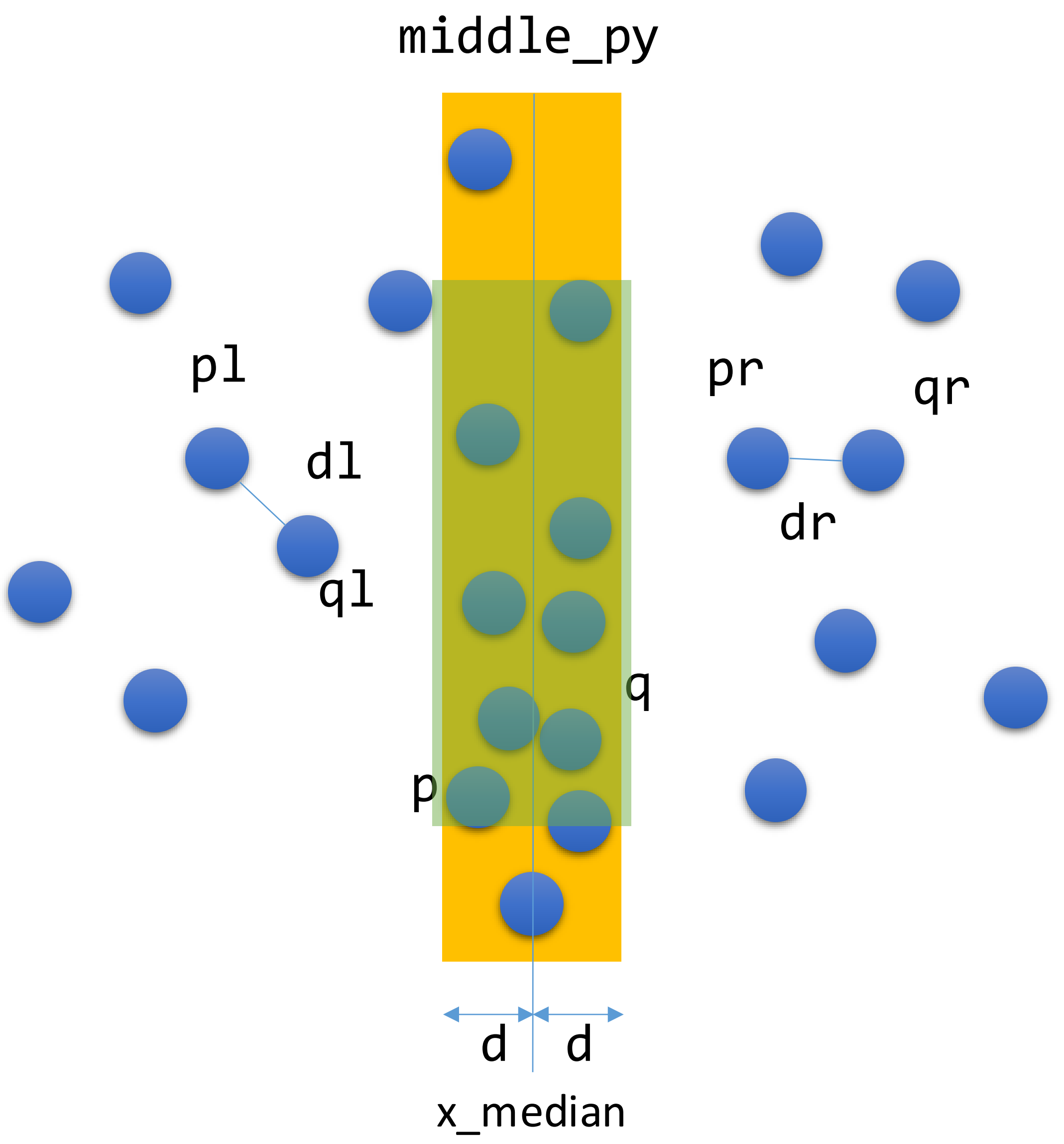
If the claim is true:

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

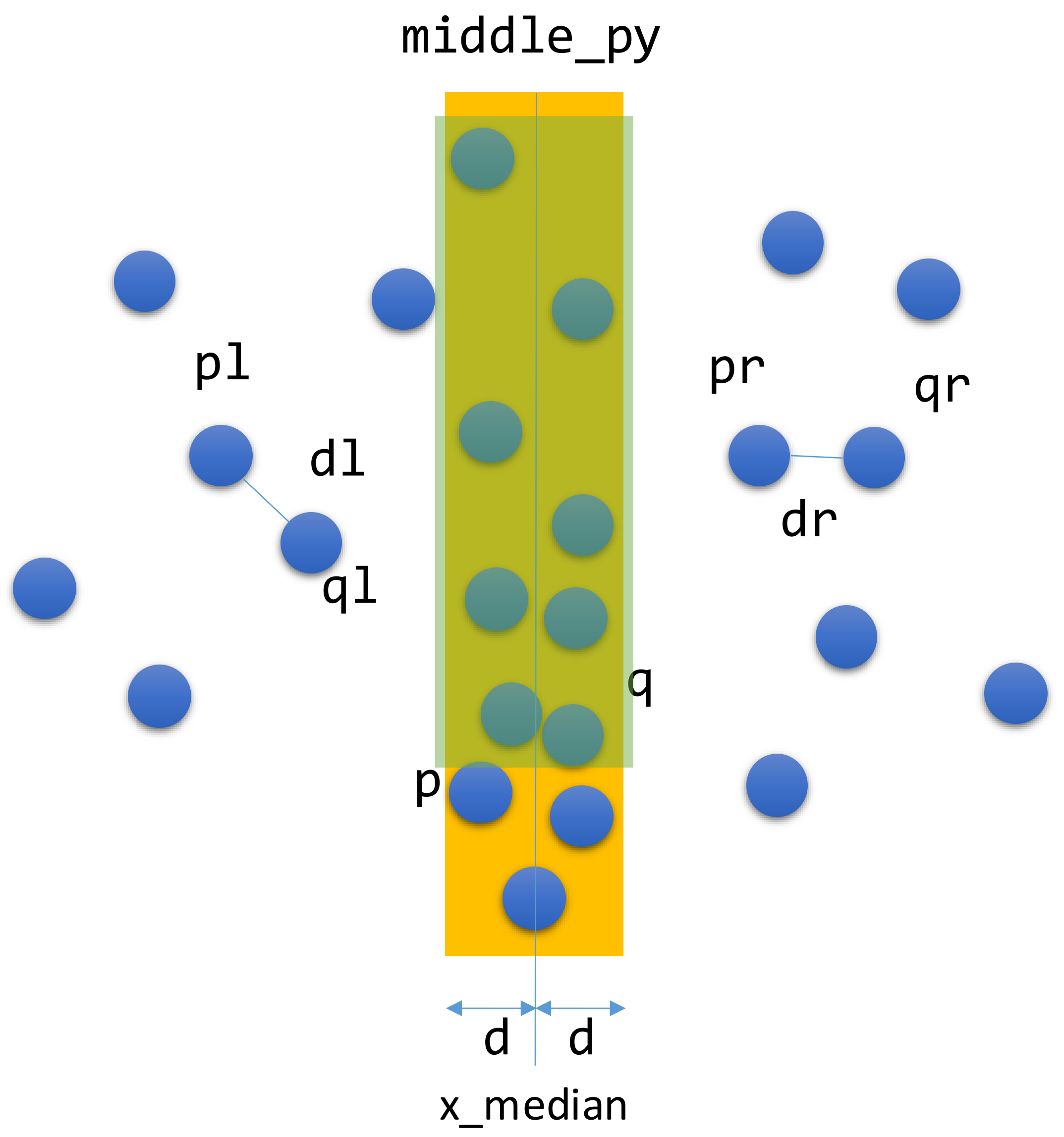
<u>Corollary 2</u>: ClosestPair is correct and runs in O(n lg n) since it has the same recursion tree as merge sort

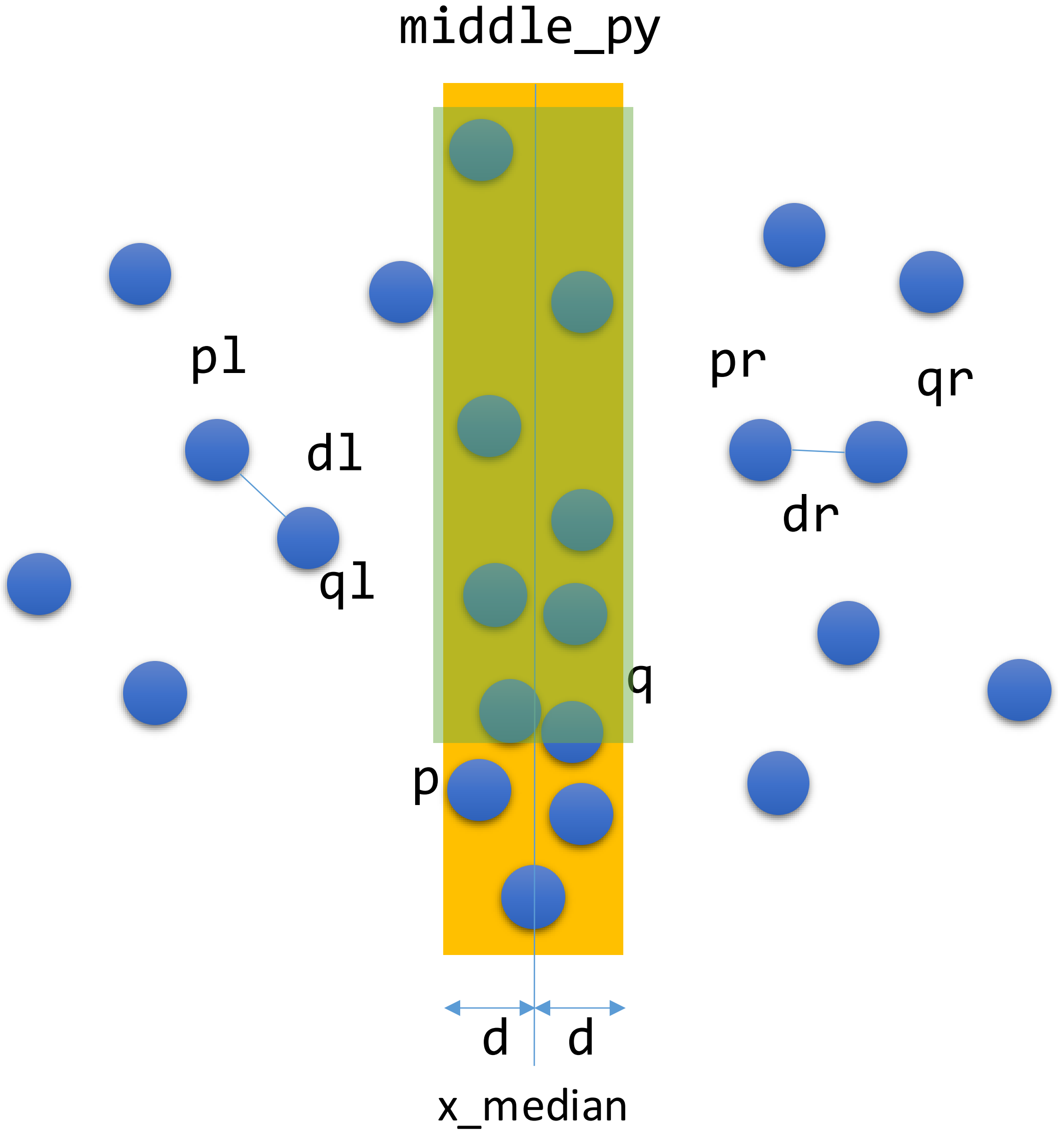

middle_py
pl
dl
ql
pr
qr
dr
q
p
d
d
x_median

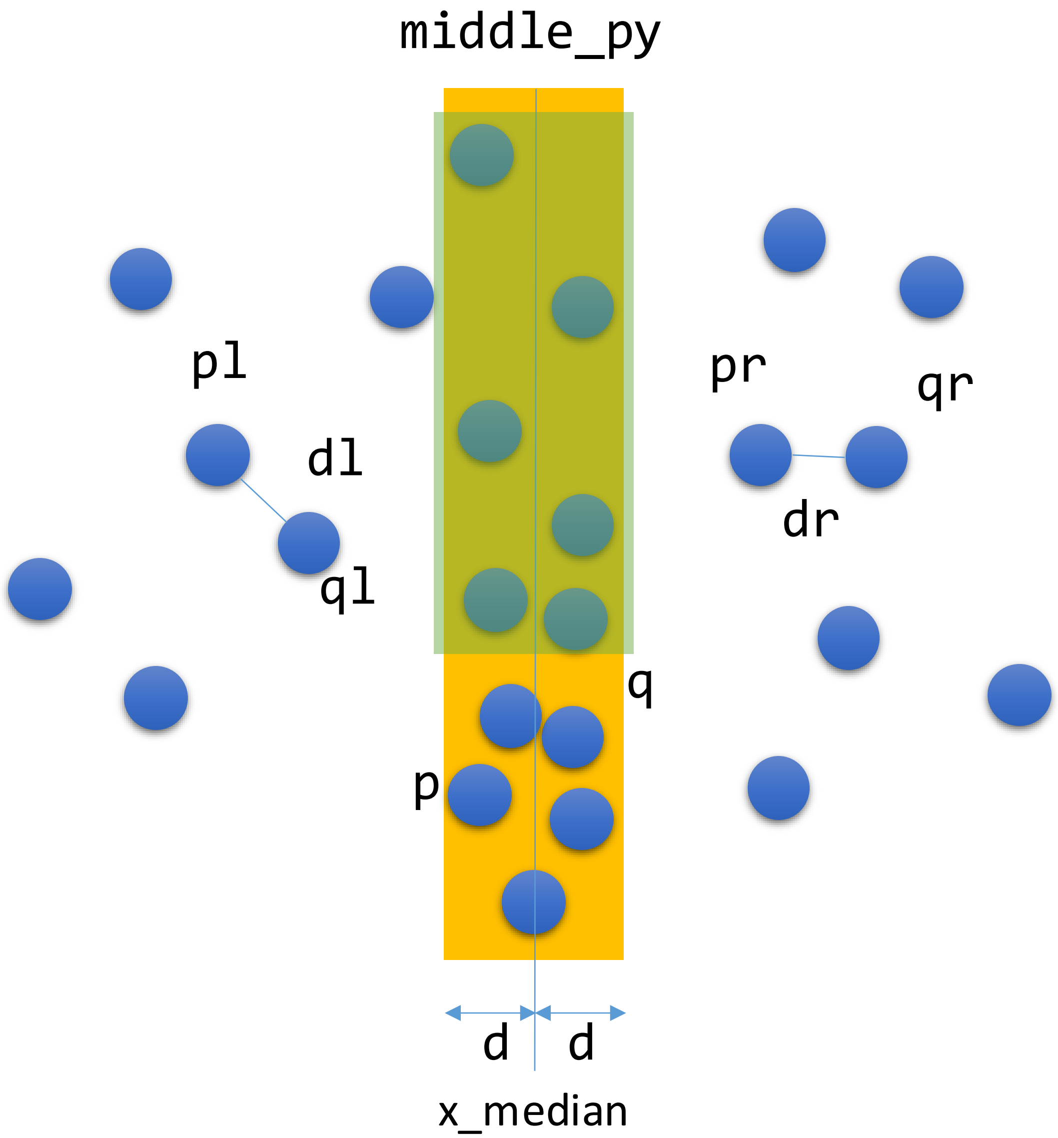

middle_py
pl
dl
ql
pr
qr
dr
q
p
d
d
x_median

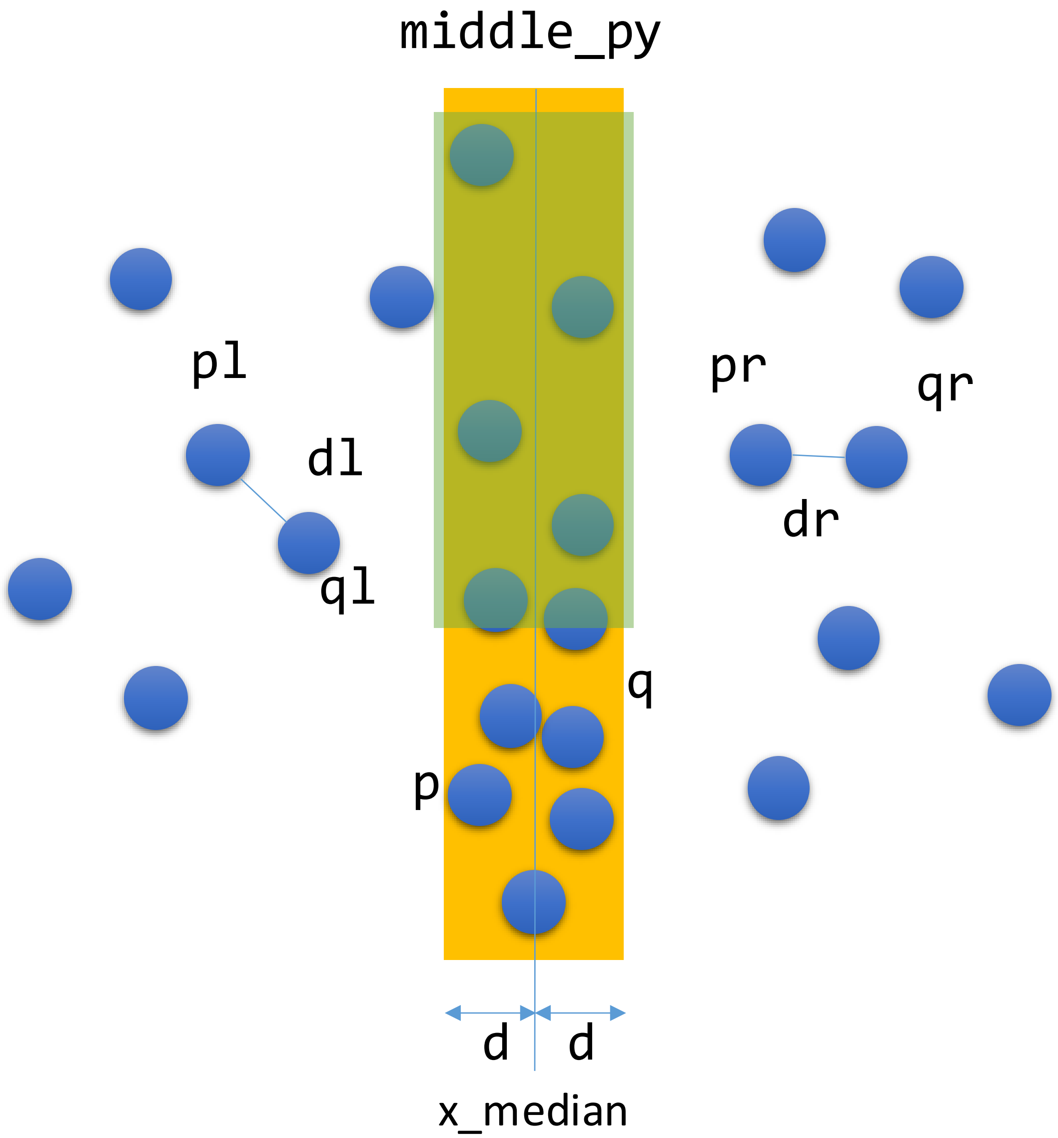


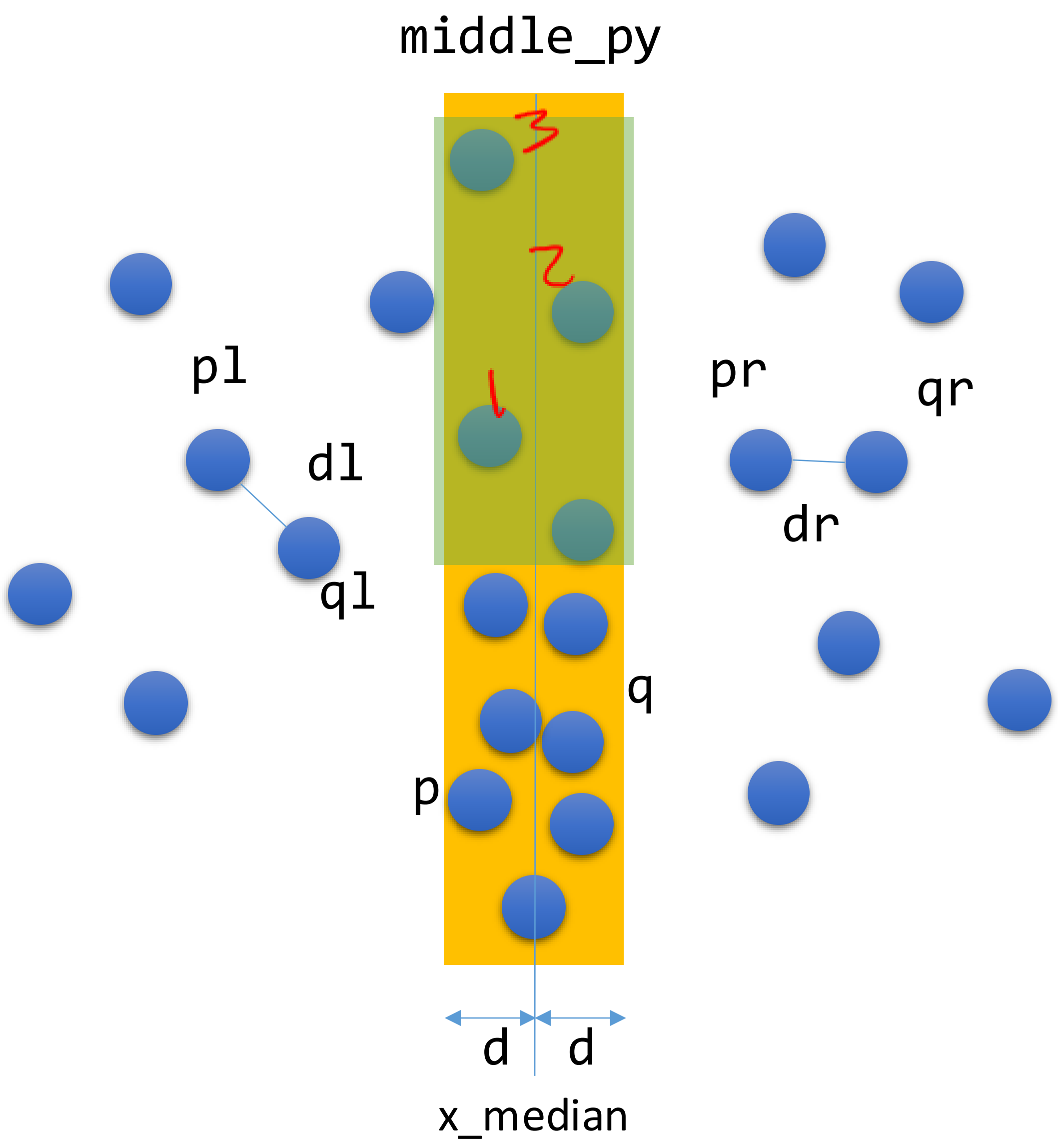
middle_py
pl
dl
ql
pr
qr
dr
q
p
d
d
x_median


middle_py
pl
dl
ql
pr
qr
dr
q
p
d
d
x_median

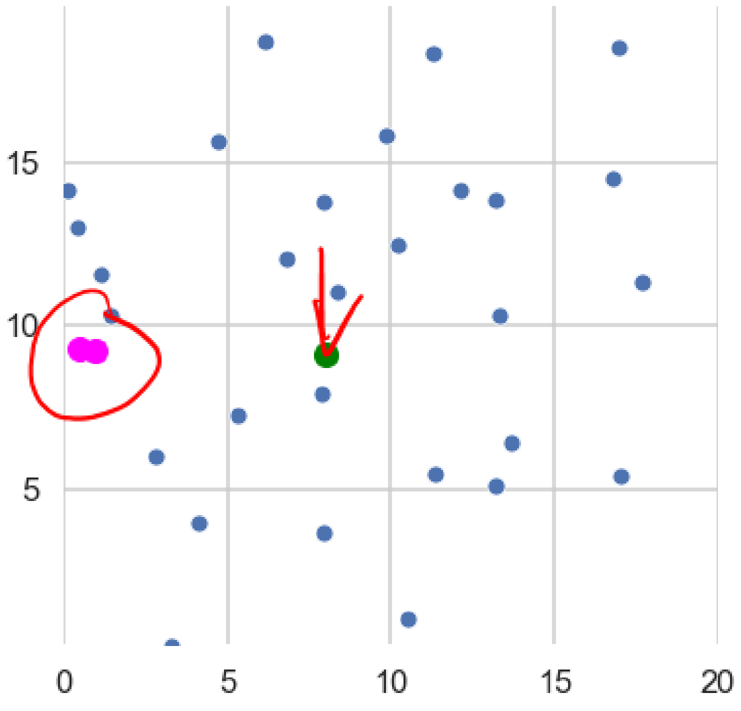

middle_py
pl
dl
ql
pr
qr
dr
q
p
d
d
x_median


middle_py
pl
dl
ql
pr
qr
dr
q
p
d
d
x_median

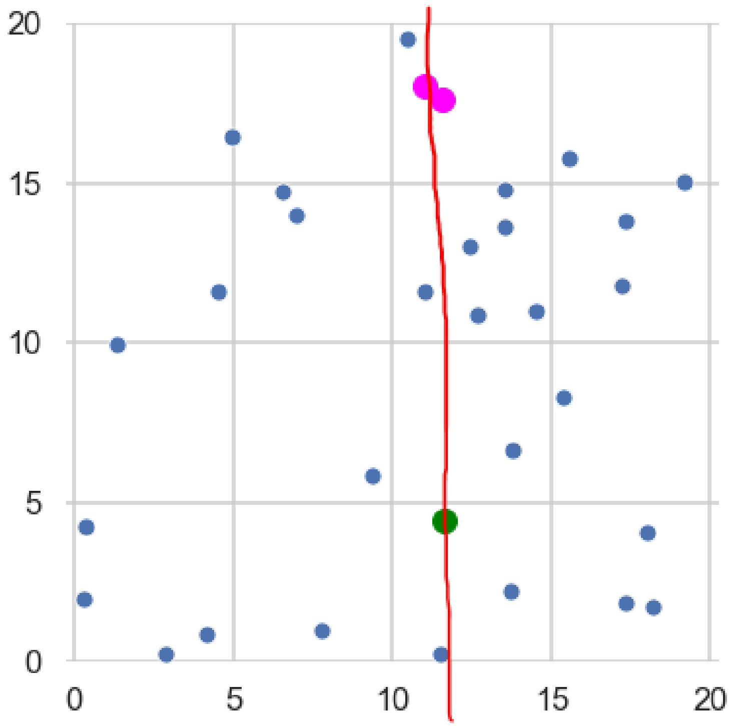

middle_py
pl
dl
ql
pr
qr
dr
q
p
d
d
x_median


middle_py
3
2
1
pl
dl
ql
pr
qr
dr
q
p
d
d
x_median

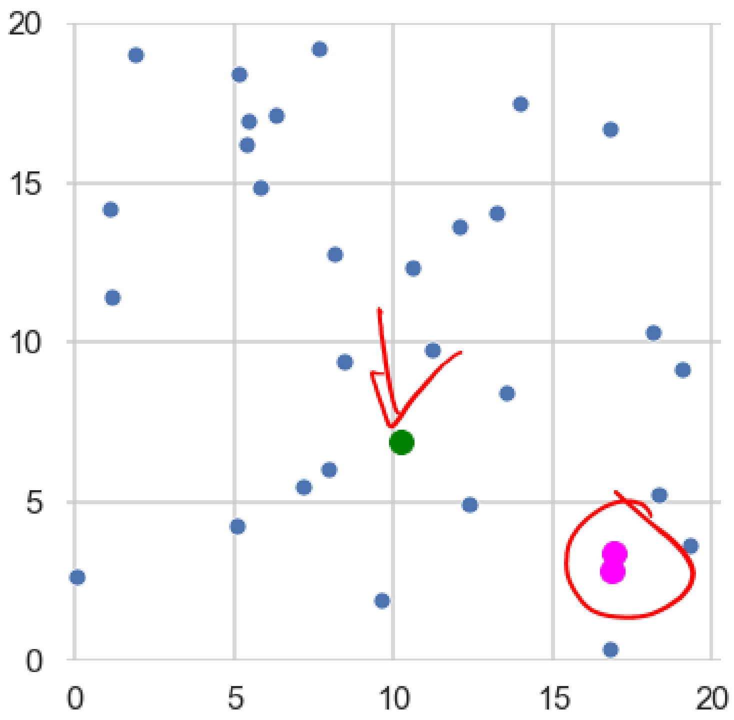
Closest on Left



Closest is Split

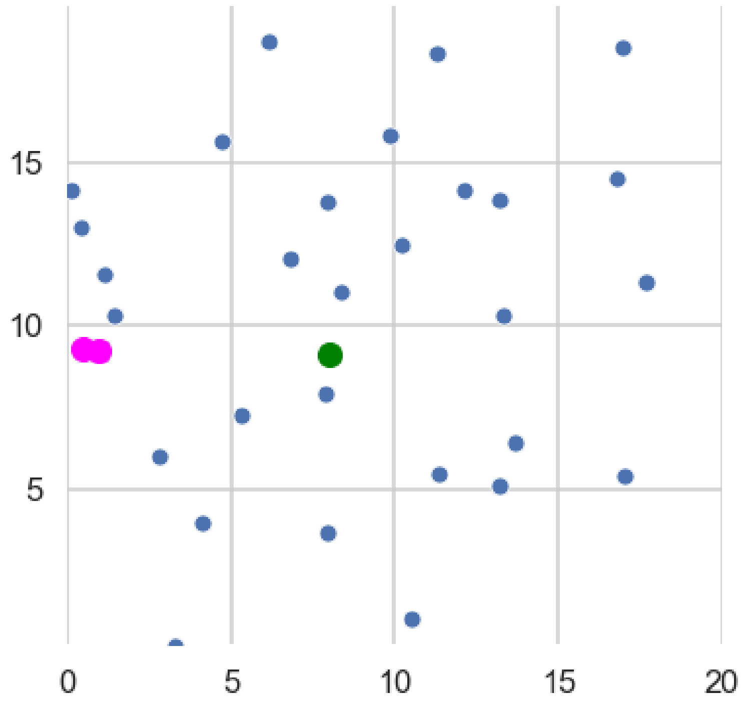


Closest on Right

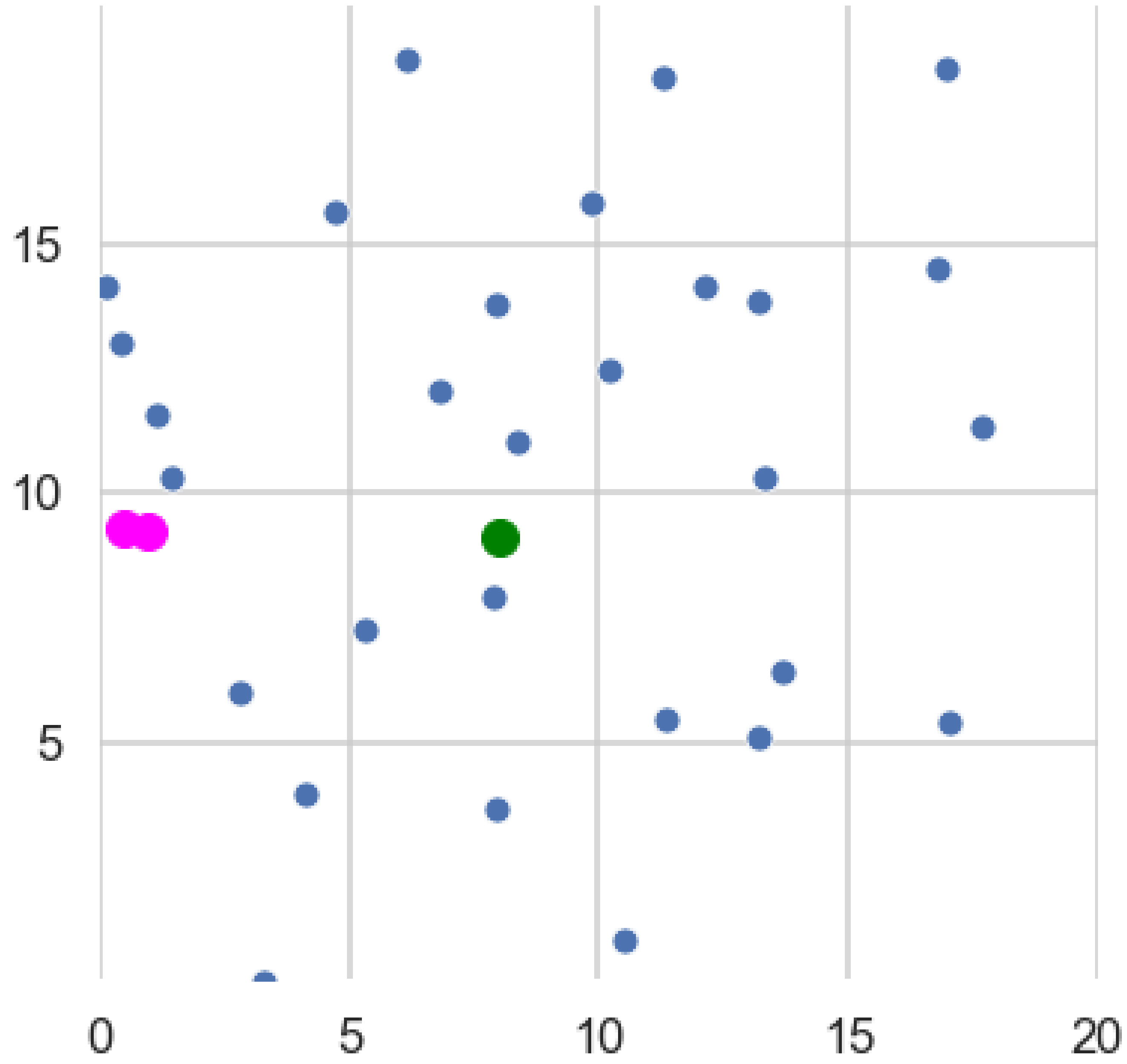


These are three different examples with different sets of points

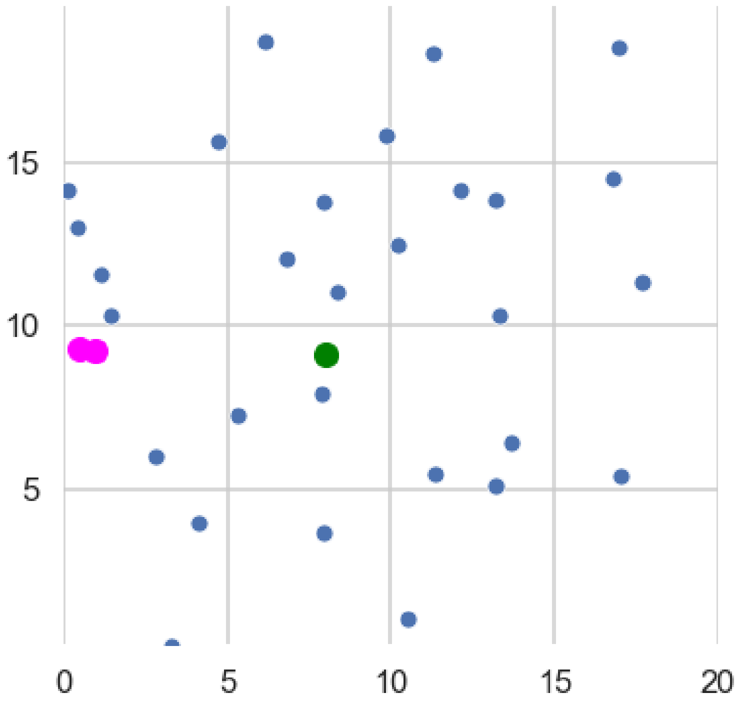
Closest on Left



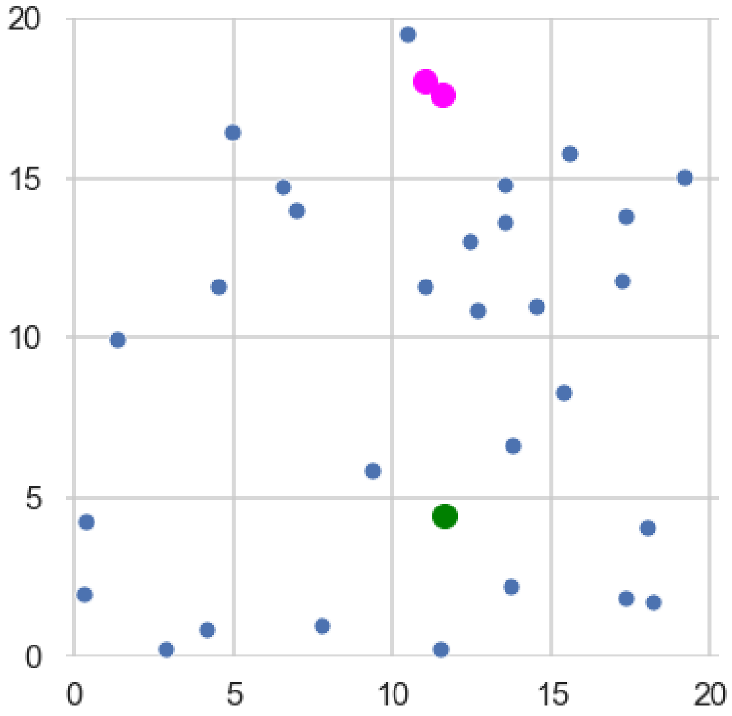
Closest on Left



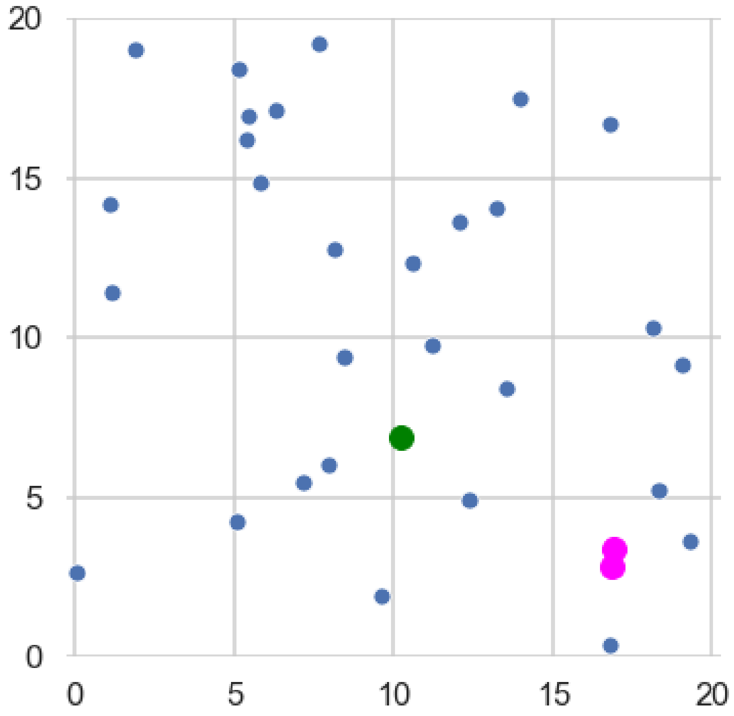
Closest on Left



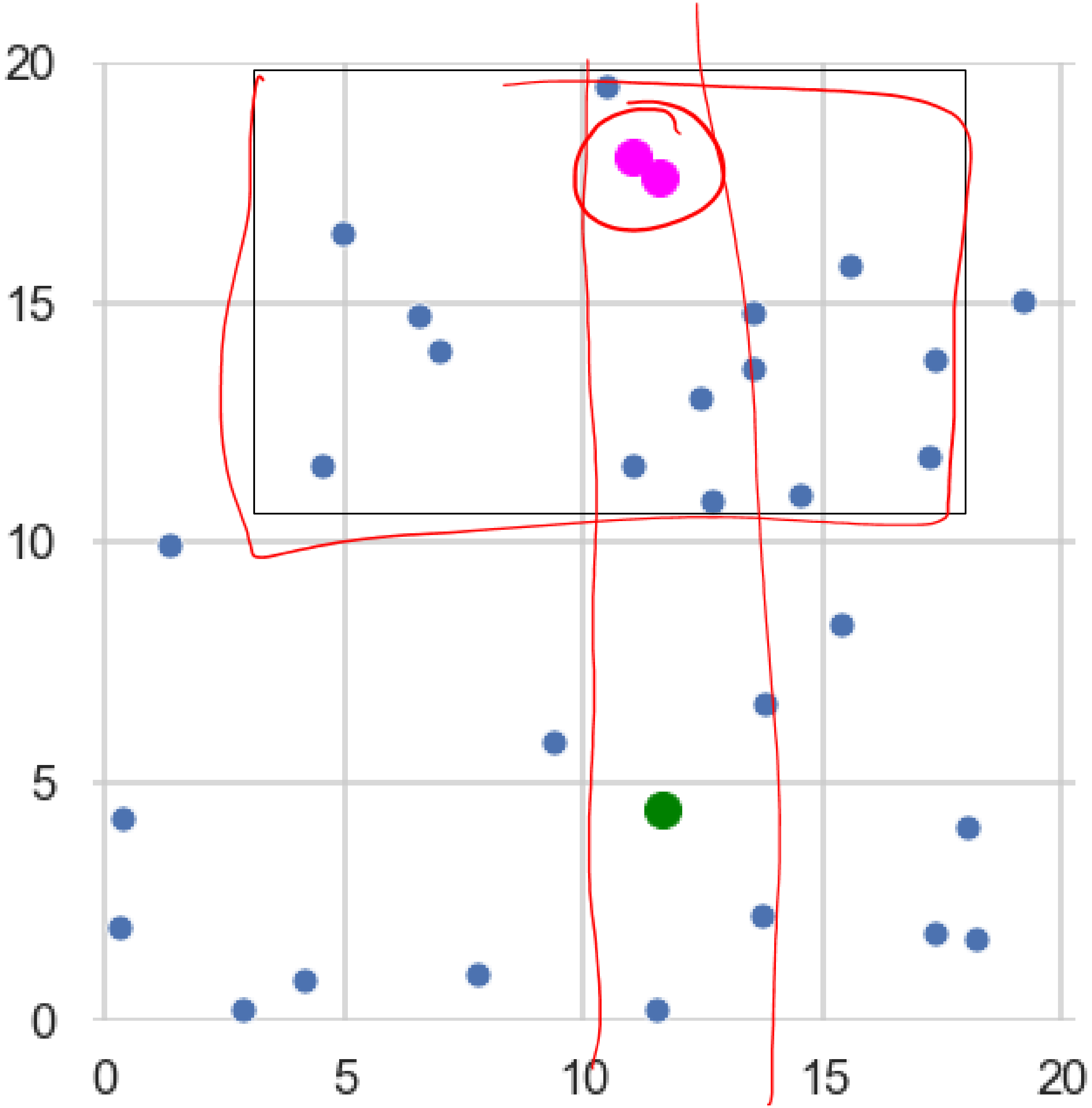
Closest is Split



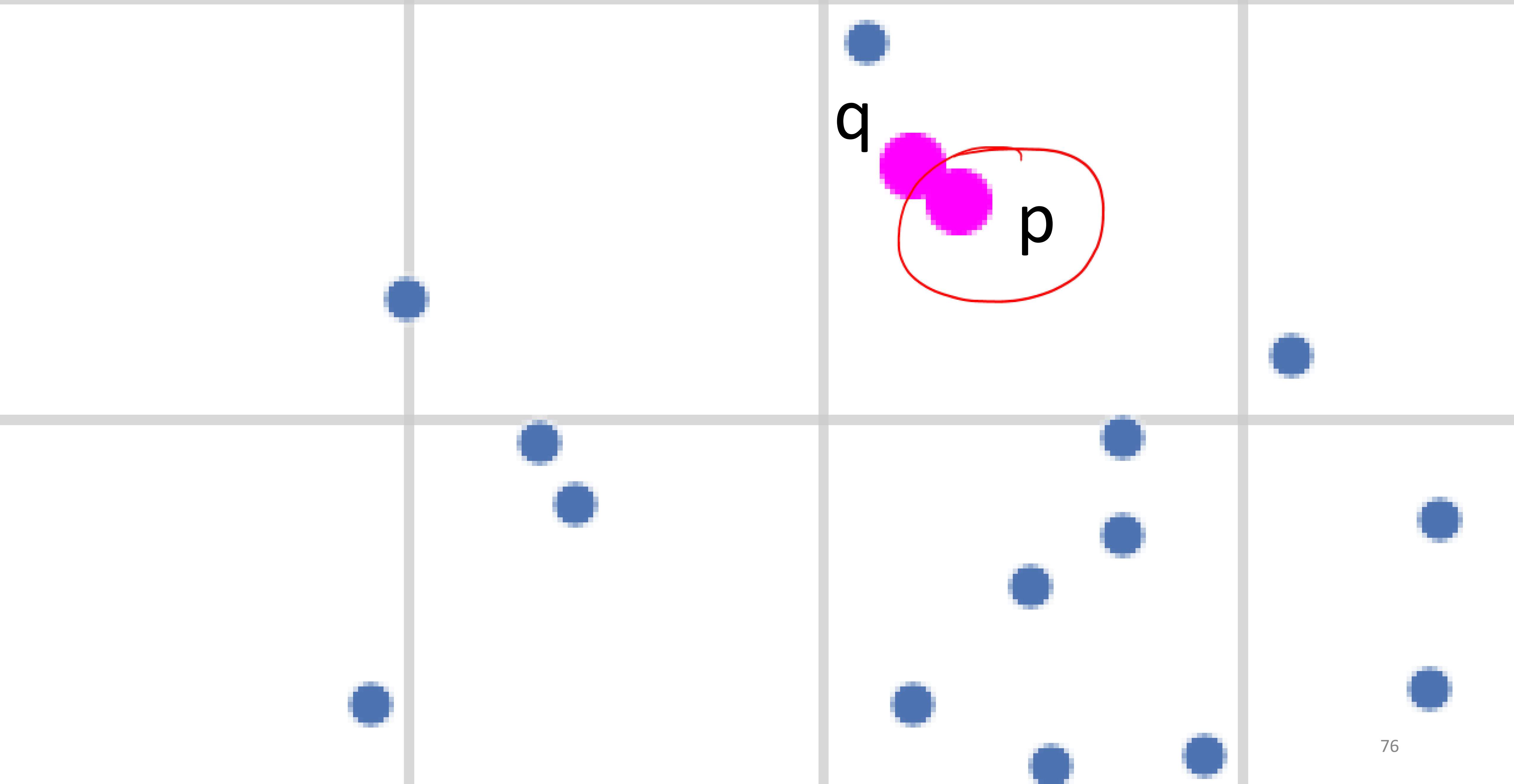
Closest on Right



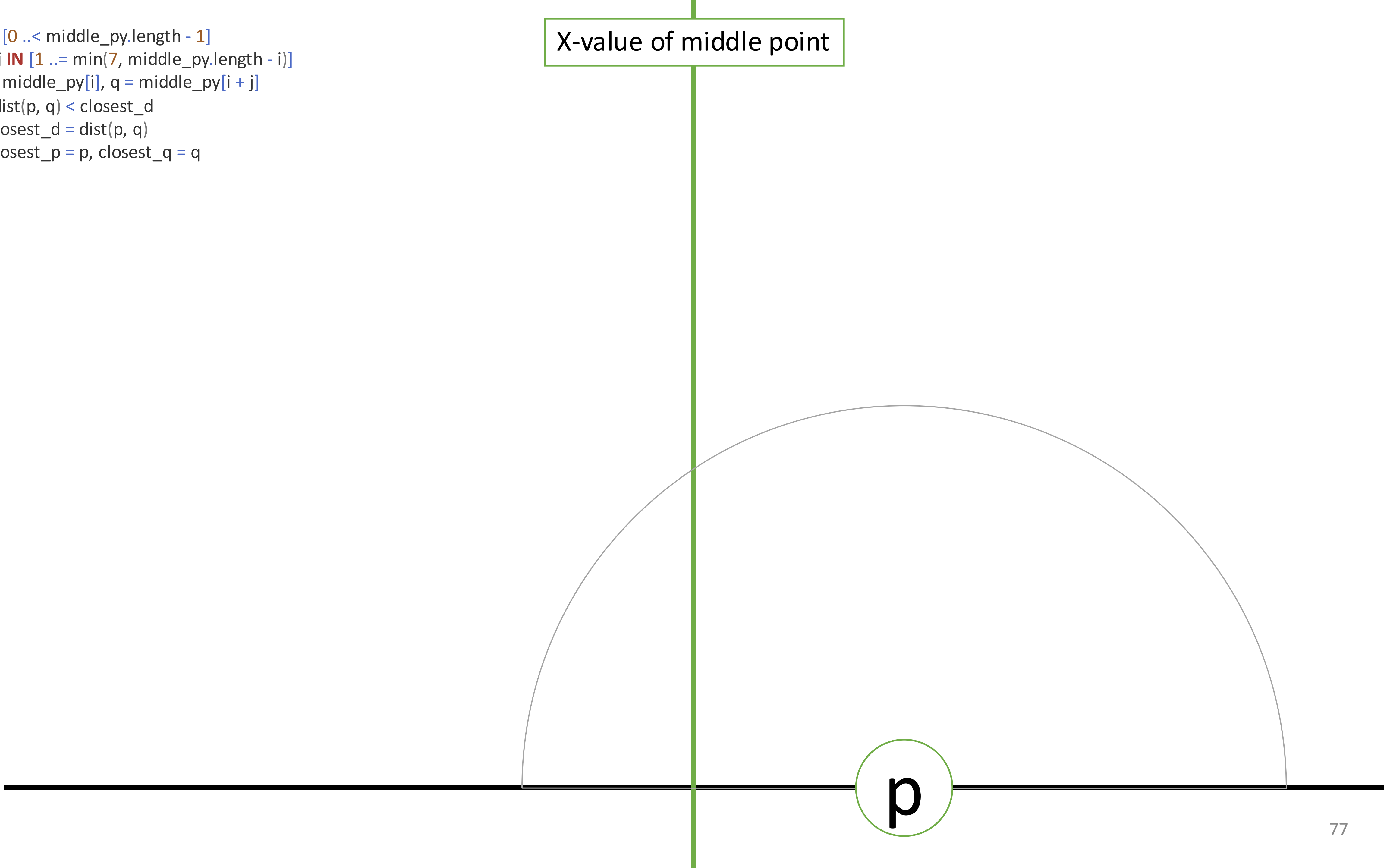
Closest is Split

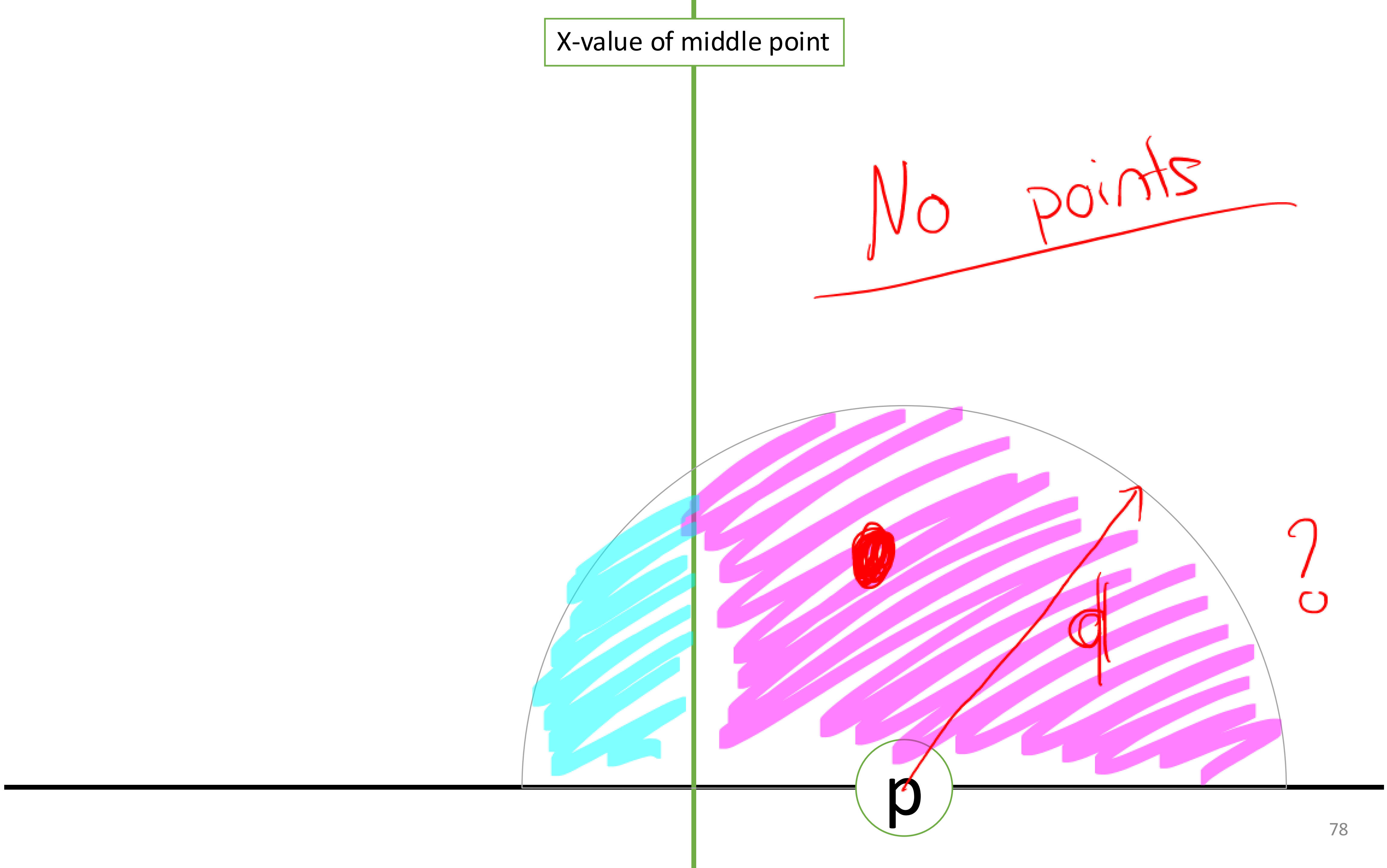


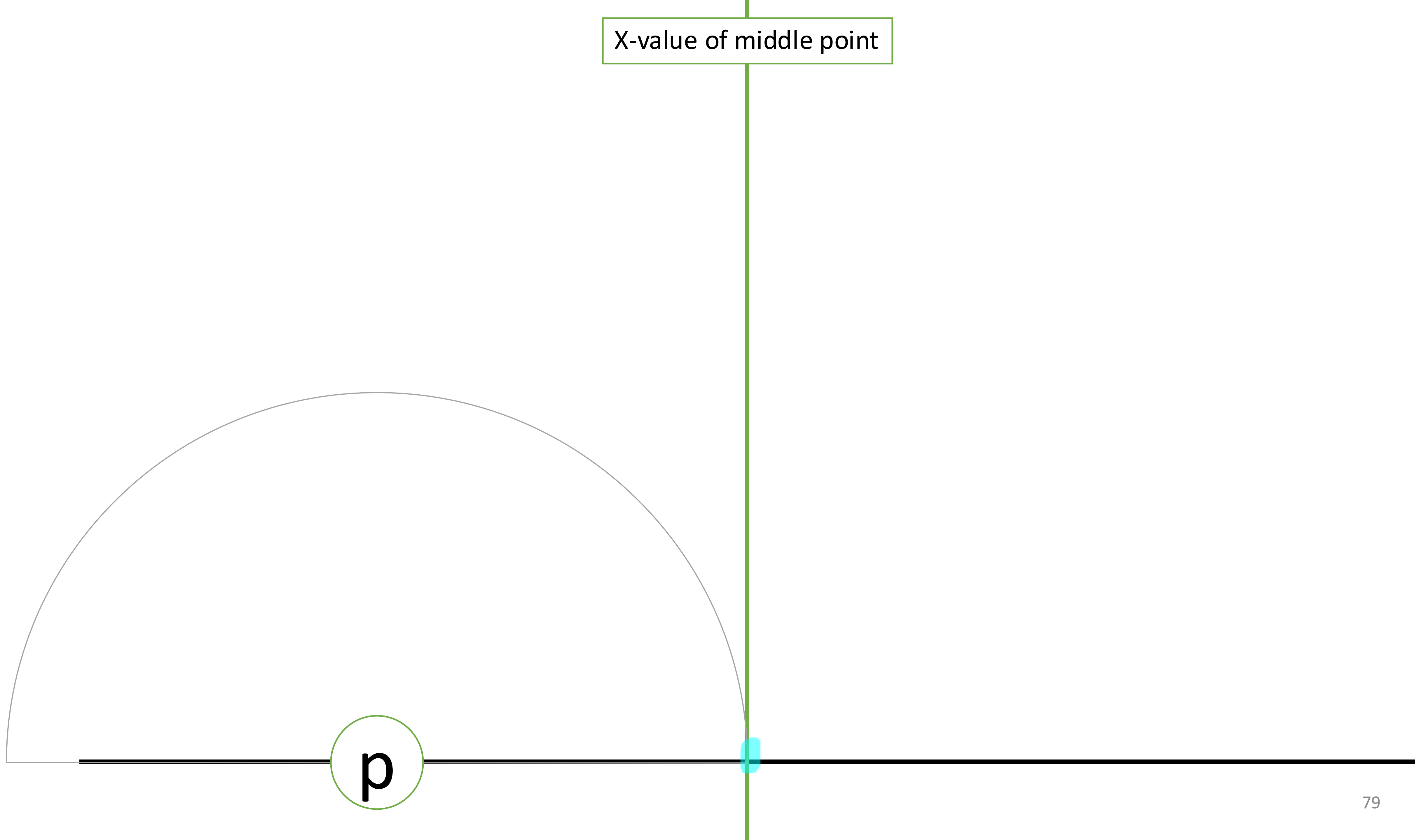
Closest is Split

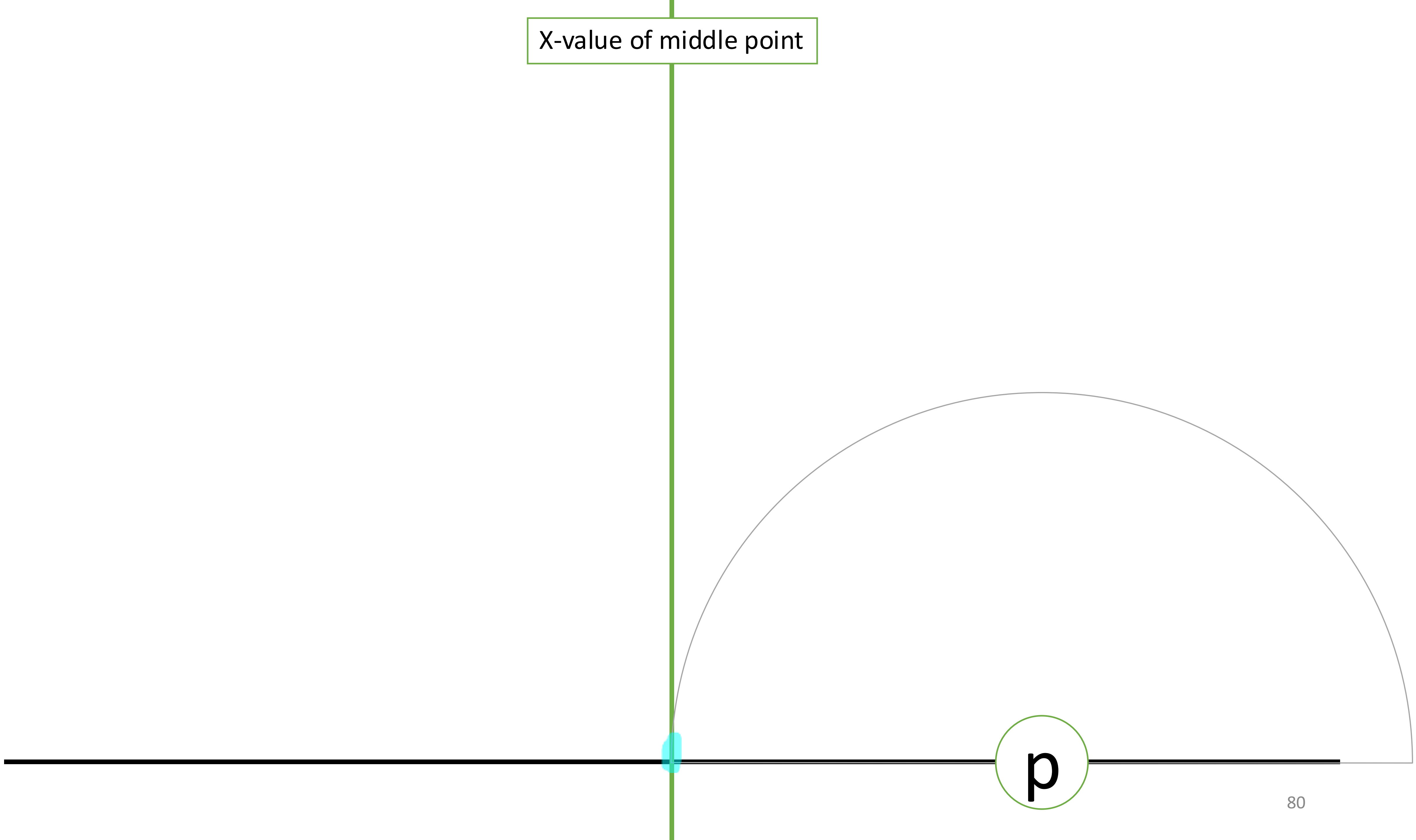


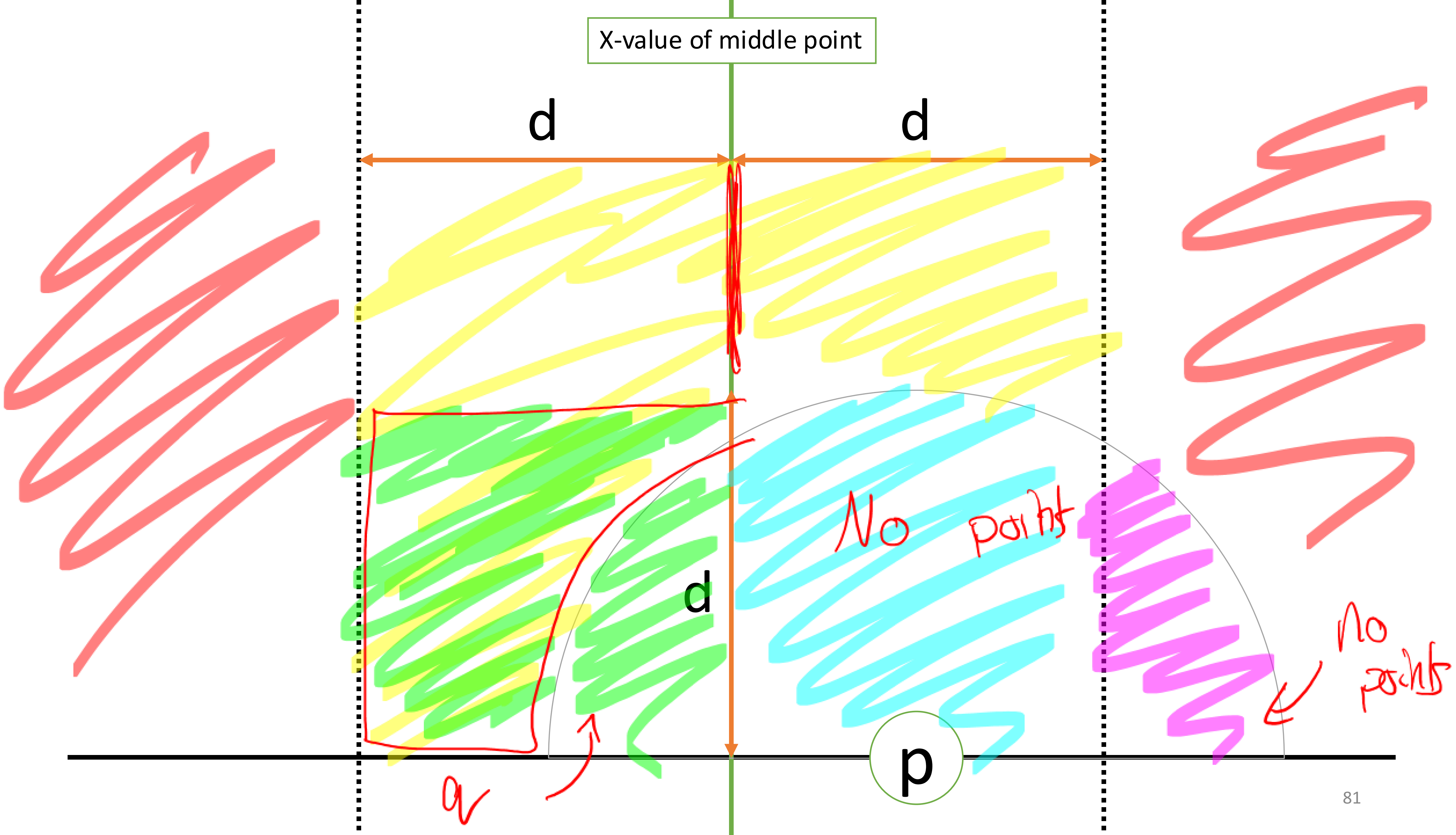
```
FOR i IN [0 ..< middle_py.length - 1]
    FOR j IN [1 ..= min(7, middle_py.length - i)]
      p = middle_py[i], q = middle_py[i + j]
      IF dist(p, q) < closest_d
        closest_d = dist(p, q)
        closest_p = p, closest_q = q
```

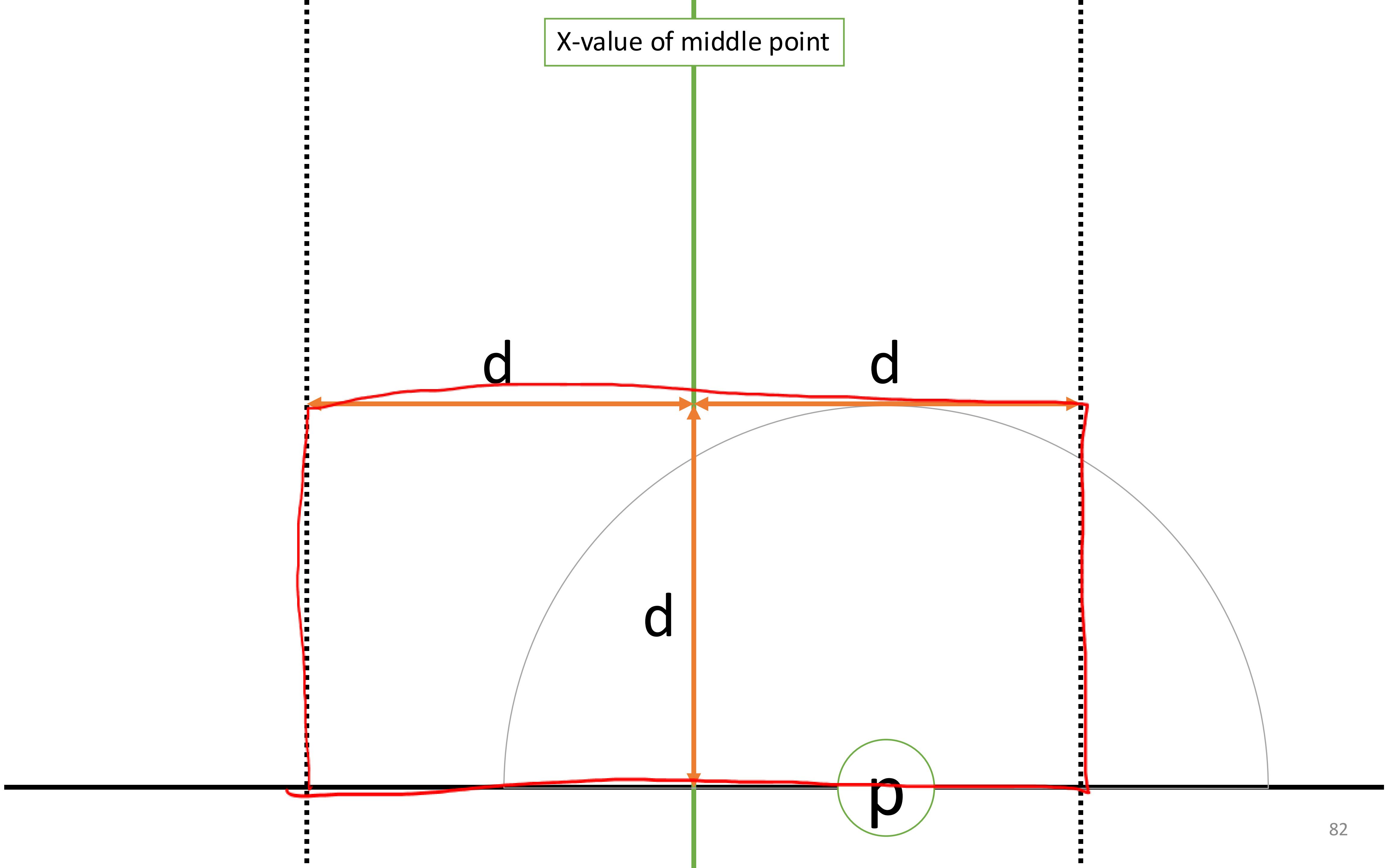


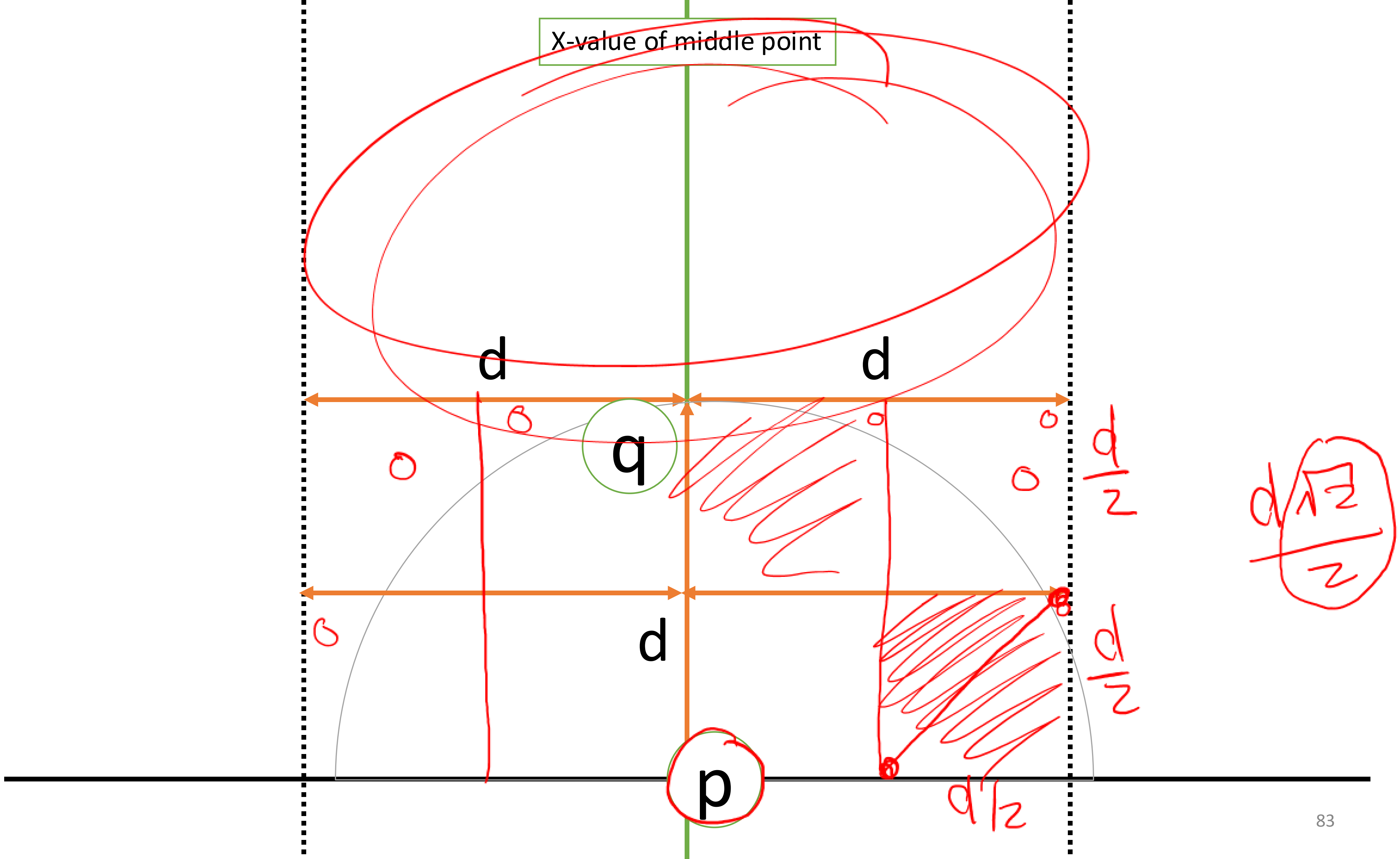

X-value of middle point
No points
d
?
p


X-value of middle point
p


X-value of middle point
p

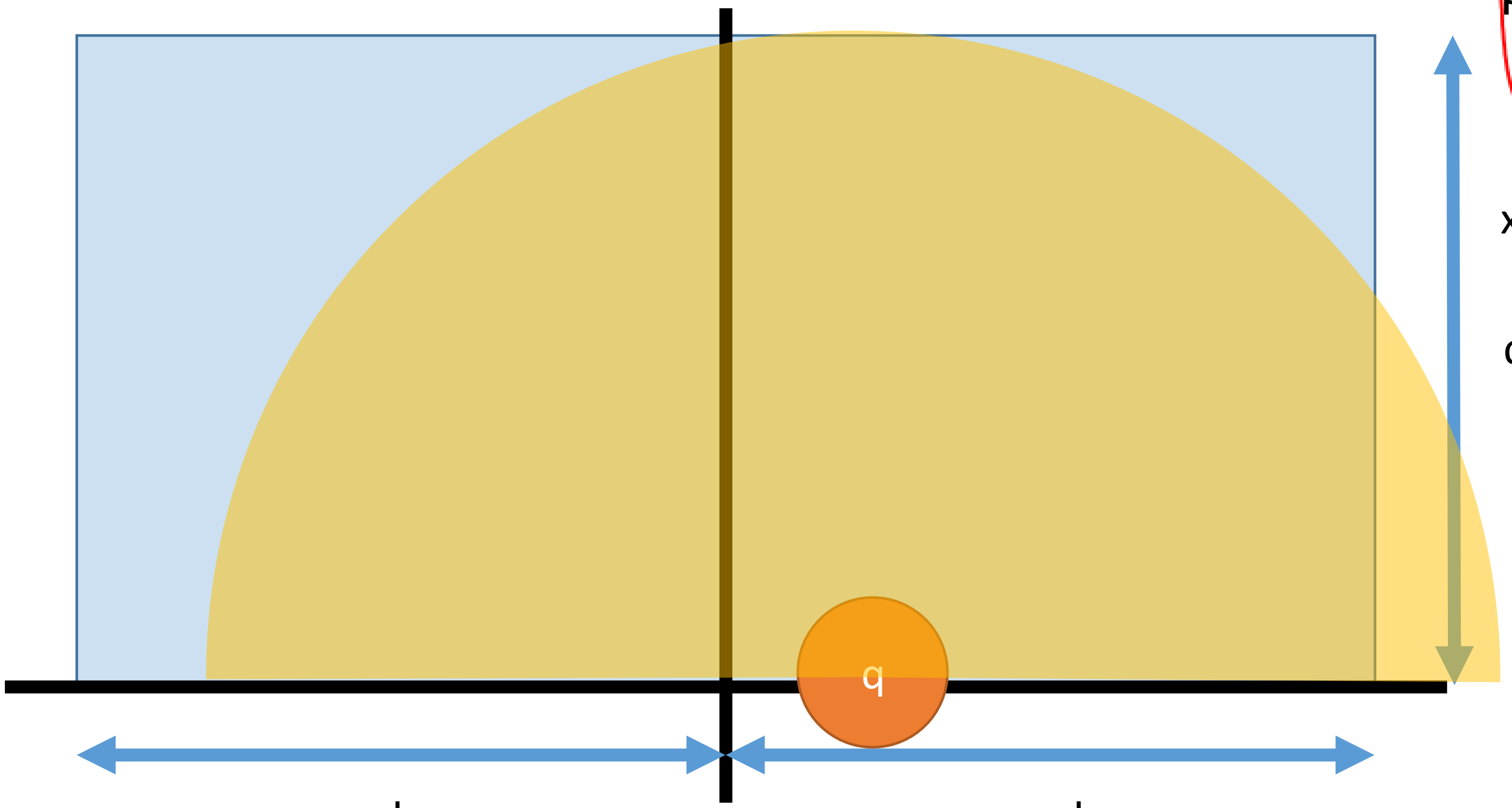

X-value of middle point
d
d
d
No point
no points
a
p

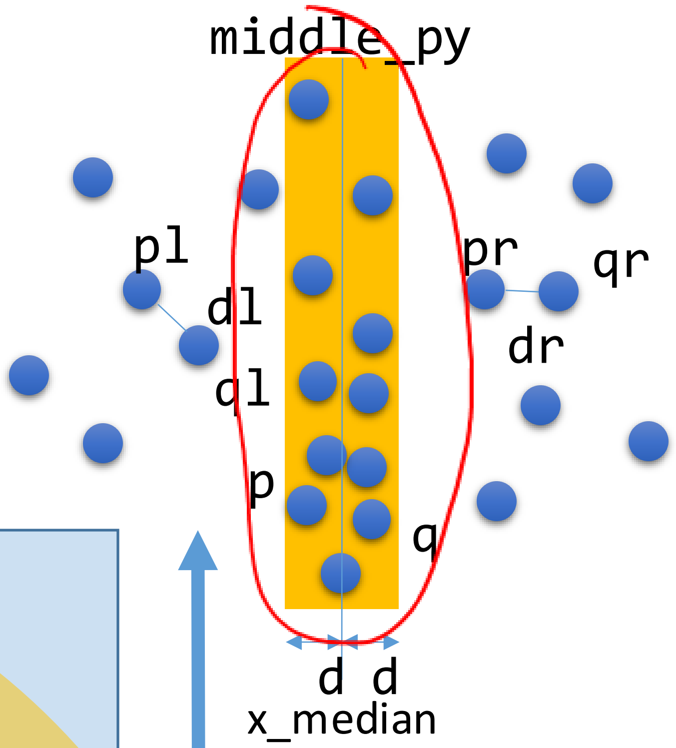

X-value of middle point
d
d
d
p


X-value of middle point
d
d
q
d
p

Proof—Part B

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

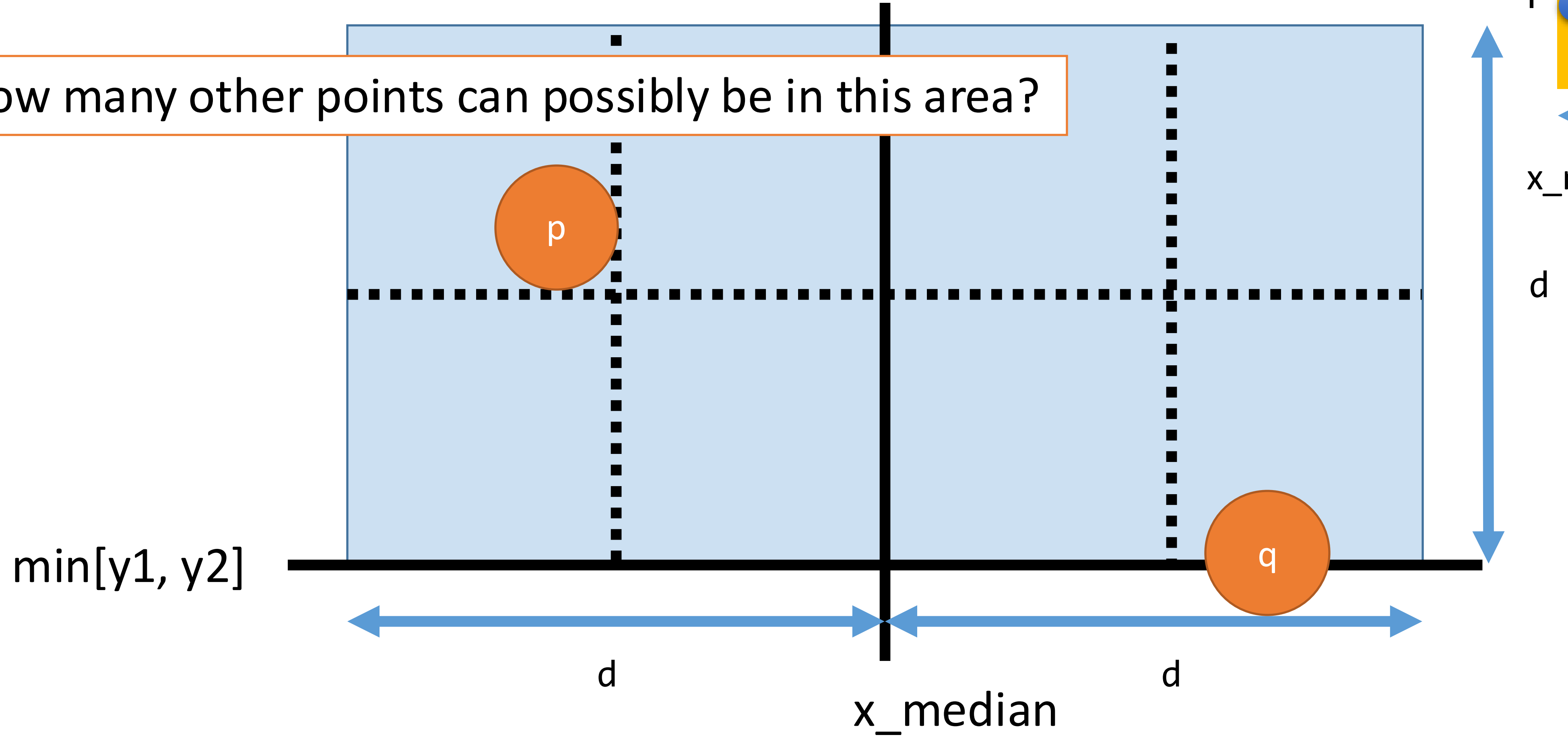


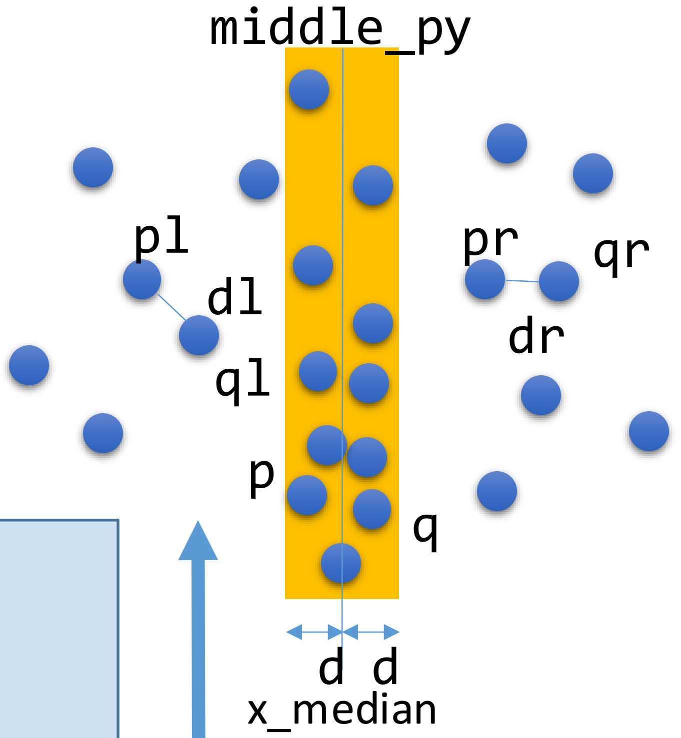


Proof—Part B

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

How many other points can possibly be in this area?

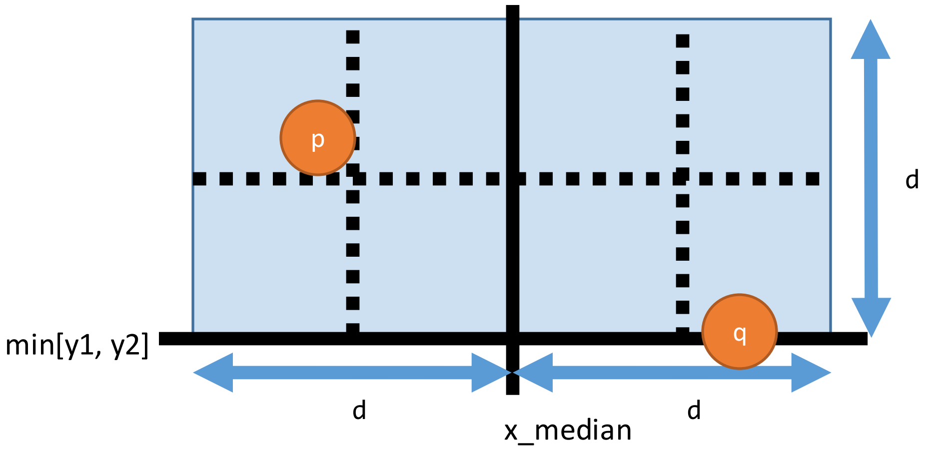




Proof—Part B

p and q are at most 7 positions apart

in `middle_py`



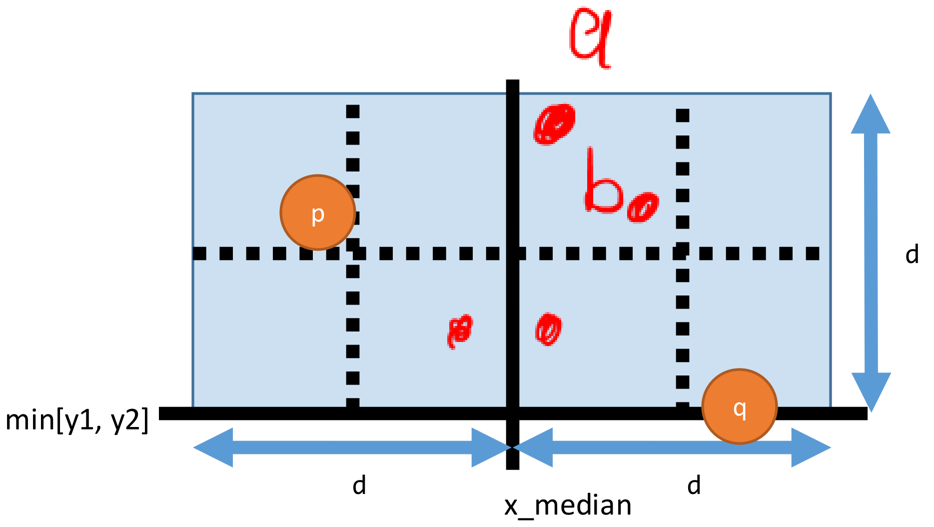
Lemma 1: All points of middle_py with a y-coordinate between those of p and q lie within those 8 boxes.

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.

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.

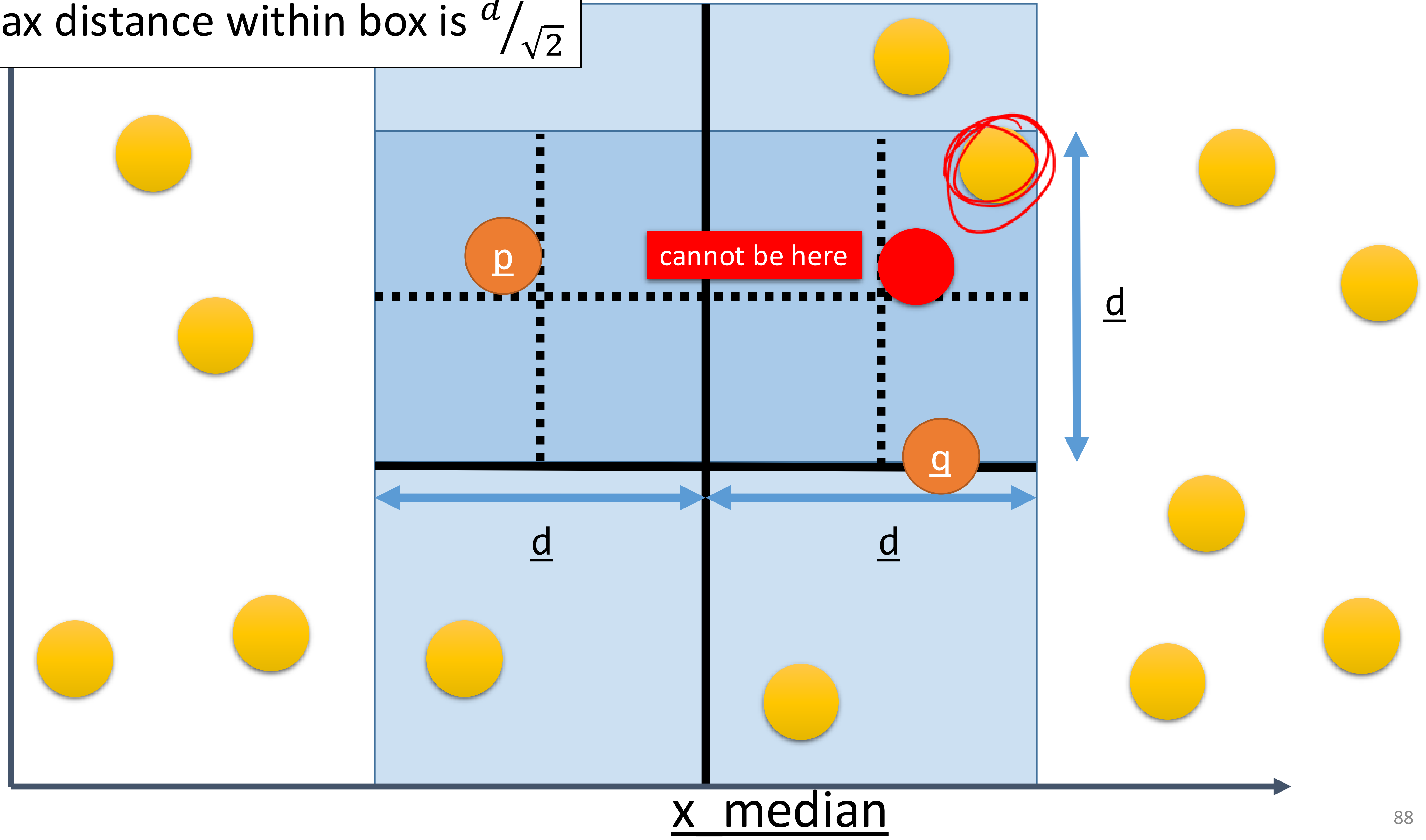
Lemma 2: At most one point of P can be in each box.

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) <= `d/2 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 ∈ `left`, q ∈ `right` be a split pair with `d(p, q) < d`

Then

A. p and q ∈ `middle_py`, and

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

If the claim is true:

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

<u>Corollary 2</u>: ClosestPair is correct and runs in O(n lg n) since it has the same recursion tree as merge sort

Summary Closest Pair

pre process

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