COMP3251
Lecture 6: Closest Pair

Closest Pair

Input: A set of n points in a plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

Output: A pair of distinct points whose distance is smallest.

Closest Pair

Input: A set of n points in a plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

Output: A pair of distinct points whose distance is smallest.

A straight-forward closest algorithm:

1) Compute the distance of all $n(n-1)/2$ pairs of distinct points.
2) Output the pair whose distance is smallest.

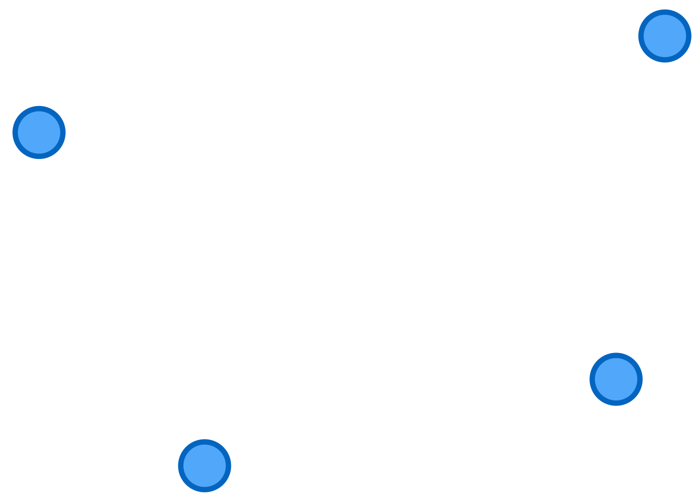
Closest Pair

Input: A set of *n* points in a plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

Output: A pair of distinct points whose distance is smallest.

A straight-forward closest algorithm:

1) Compute the distance of all $n(n-1)/2$ pairs of distinct points.
2) Output the pair whose distance is smallest.



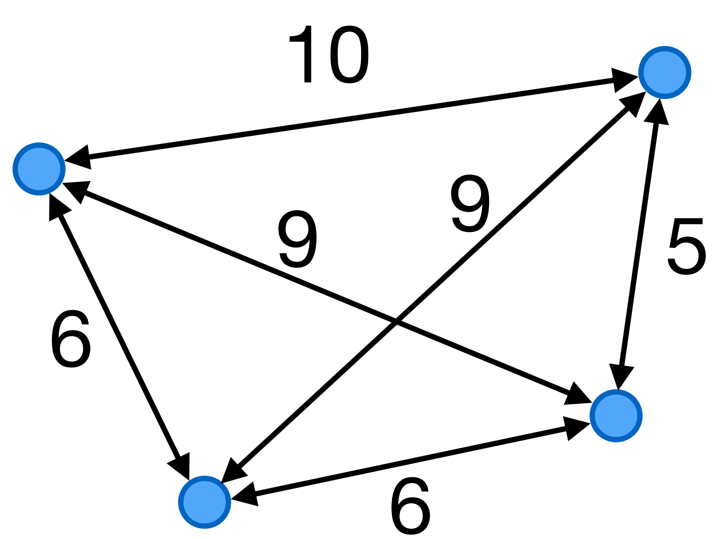
Closest Pair

Input: A set of n points in a plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

Output: A pair of distinct points whose distance is smallest.

A straight-forward closest algorithm:

1) Compute the distance of all $n(n-1)/2$ pairs of distinct points.
2) Output the pair whose distance is smallest.



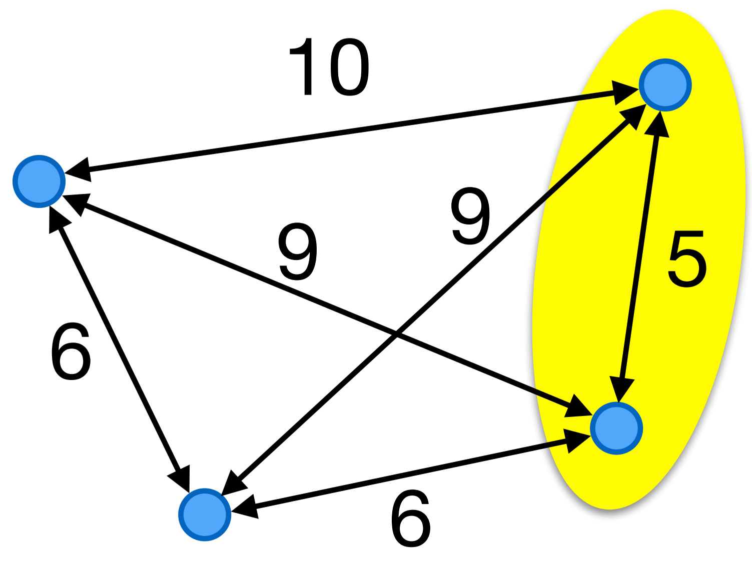
Closest Pair

Input: A set of n points in a plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

Output: A pair of distinct points whose distance is smallest.

A straight-forward closest algorithm:

1) Compute the distance of all $n(n-1)/2$ pairs of distinct points.
2) Output the pair whose distance is smallest.



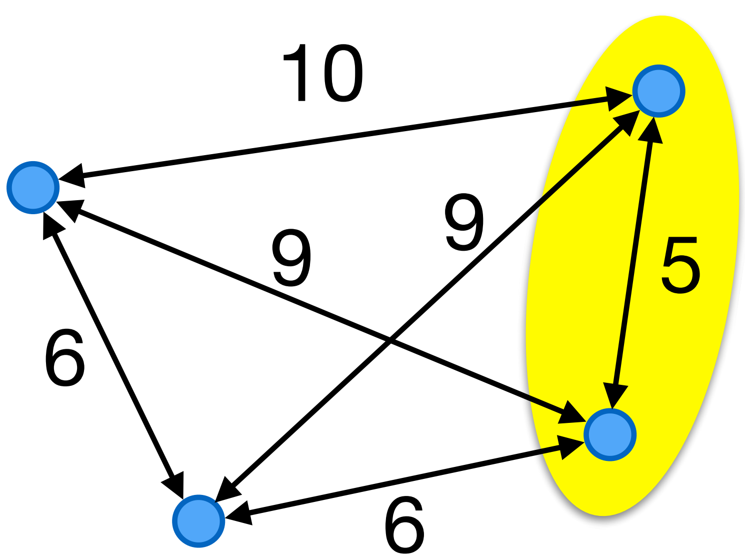
Closest Pair

Input: A set of n points in a plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.

Output: A pair of distinct points whose distance is smallest.

A straight-forward closest algorithm:

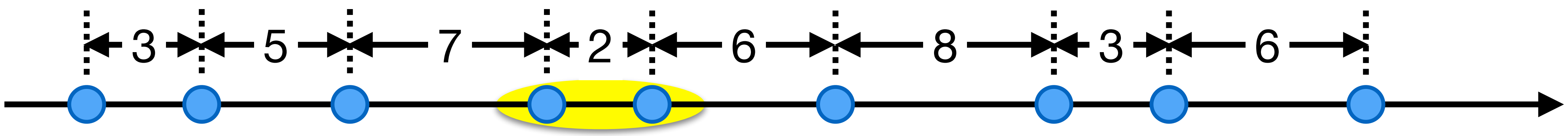
1) Compute the distance of all $n(n-1)/2$ pairs of distinct points.
2) Output the pair whose distance is smallest.



Running time: $O(n^2)$

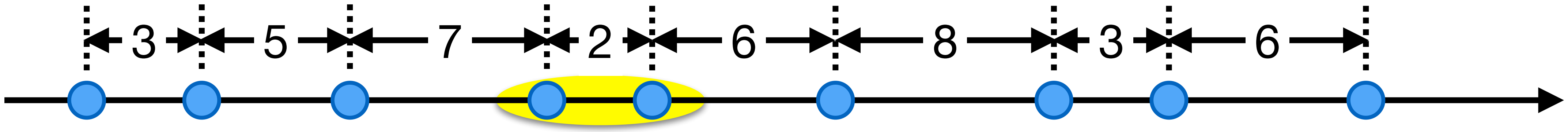
Is it possible to do any better?

Is it possible to do any better?



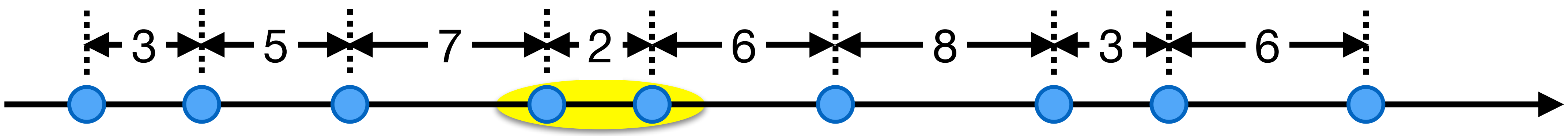
Is it possible to do any better?

- If all points are on the same line, we can first sort them and then check only the *n - 1* neighboring pairs.



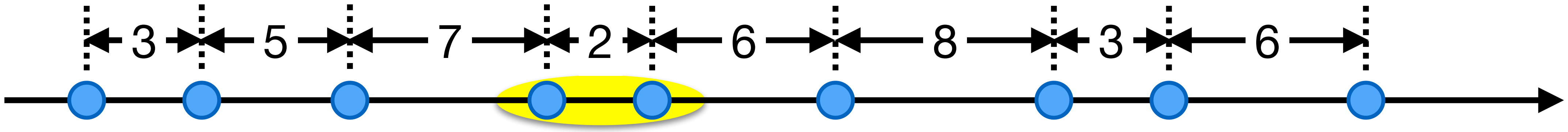
Is it possible to do any better?

- If all points are on the same line, we can first sort them and then check only the *n - 1* neighboring pairs.
- This takes *O(n log n)* time.



Is it possible to do any better?

- If all points are on the same line, we can first sort them and then check only the *n - 1* neighboring pairs.
- This takes *O(n log n)* time.

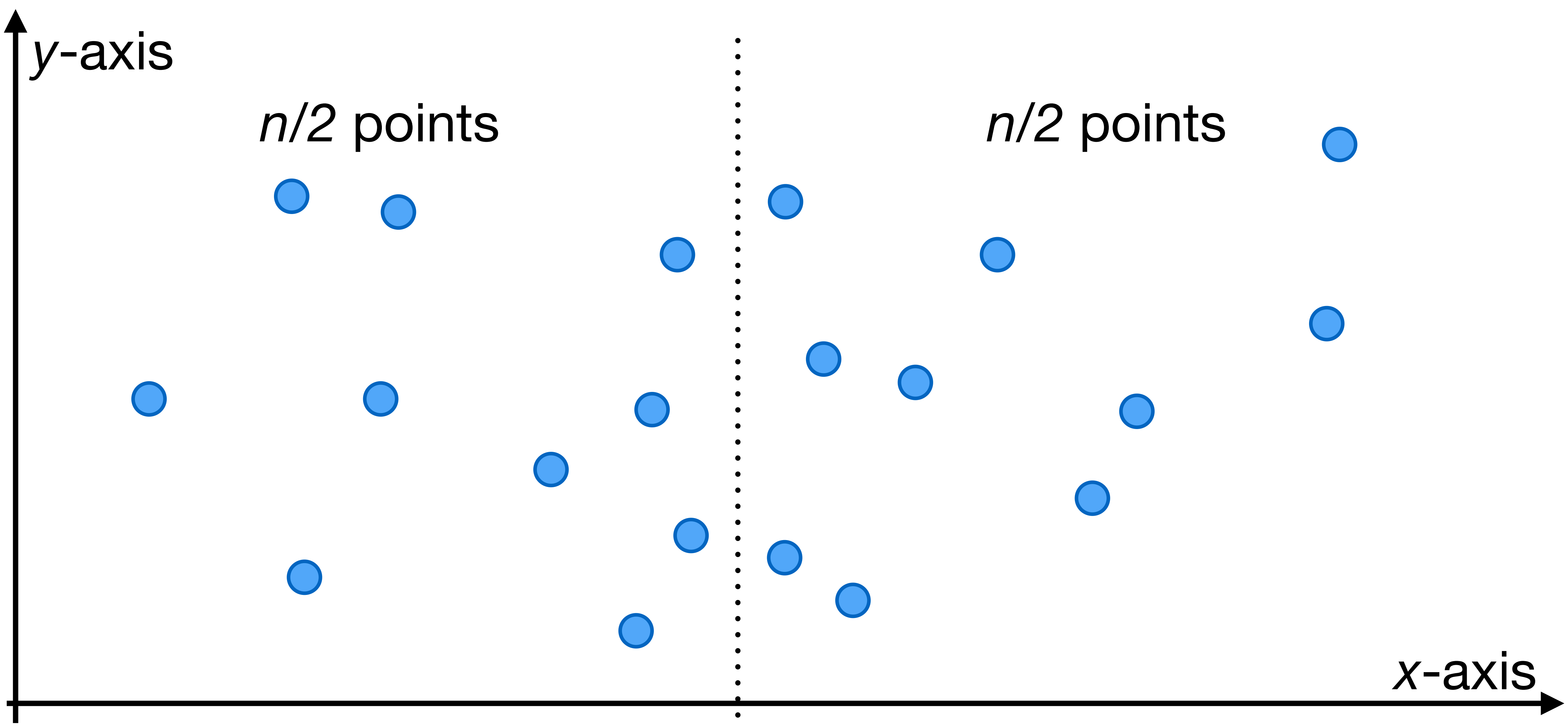


Idea: If some pairs of points are obviously too far, then we can simply ignore them.

A Divide and Conquer Algorithm for Closest Pair

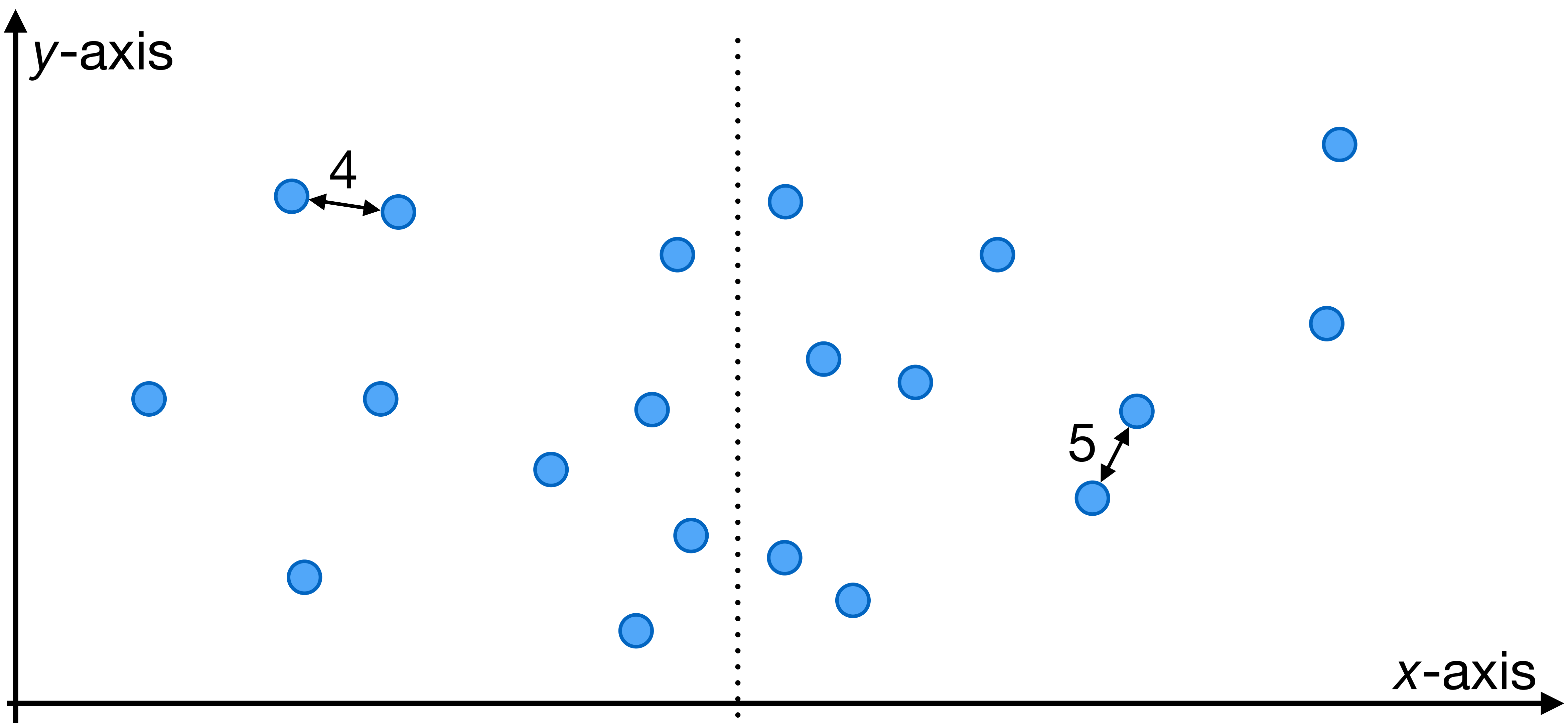
Divide: Sort the points by their x-coordinates.
Draw a vertical line L so that $n/2$ points on each side.

Assumption (for ease of discussion): No two points have same x-coordinate.



A Divide and Conquer Algorithm for Closest Pair

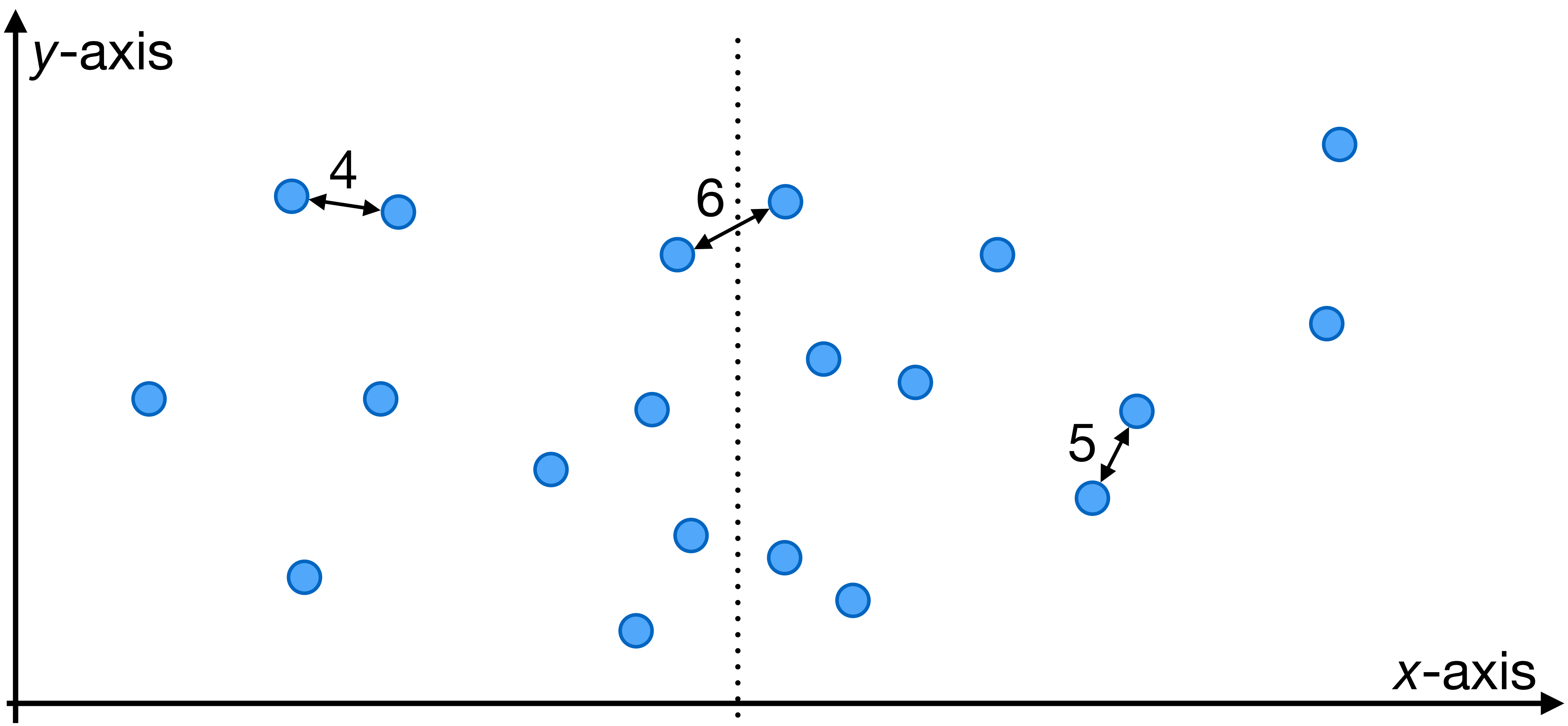
Recurse: Find the closest pair on each side.



A Divide and Conquer Algorithm for Closest Pair

Recurse: Find the closest pair on each side.

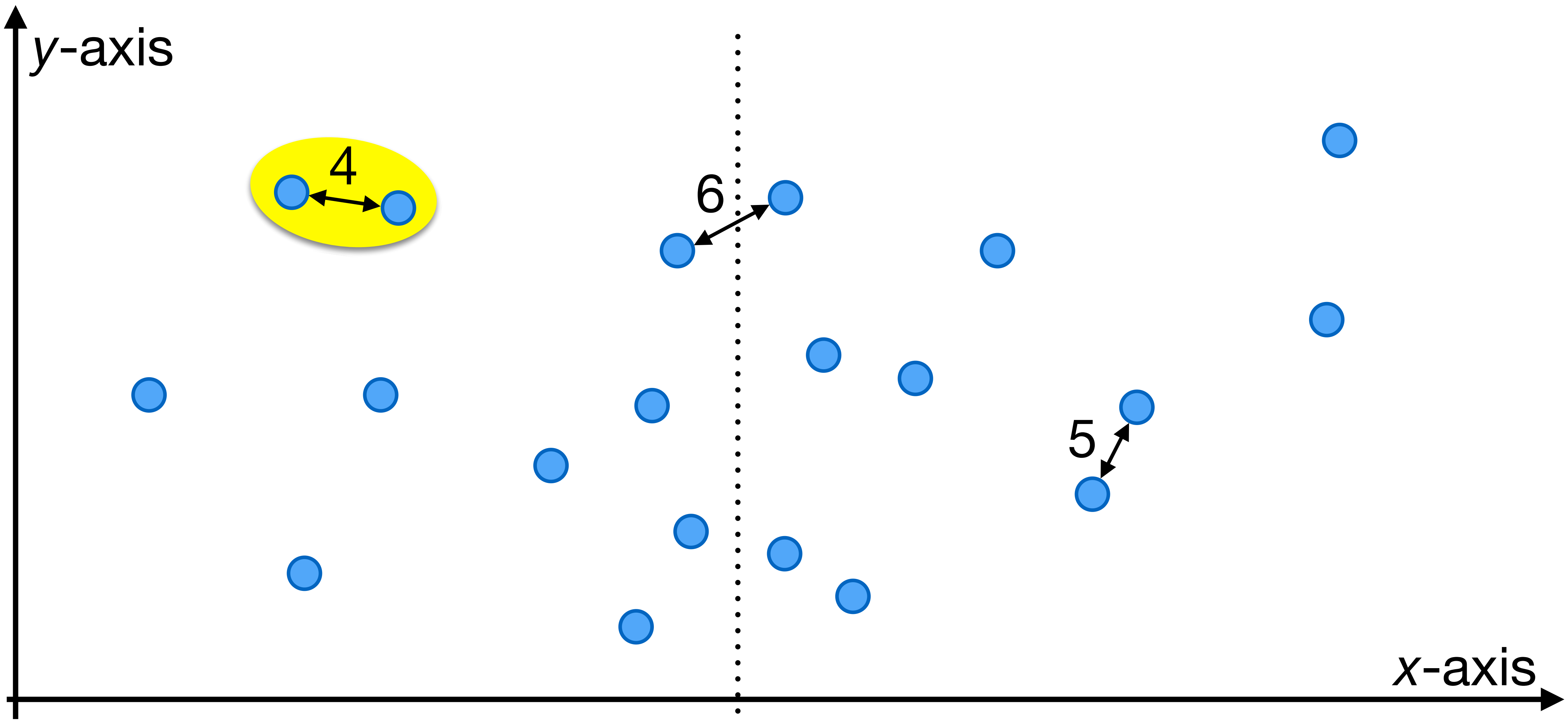
Combine: Find the closest pair with one point on each side.
Output the closest of the three pairs.



A Divide and Conquer Algorithm for Closest Pair

Recurse: Find the closest pair on each side.

Combine: Find the closest pair with one point on each side.
Output the closest of the three pairs.

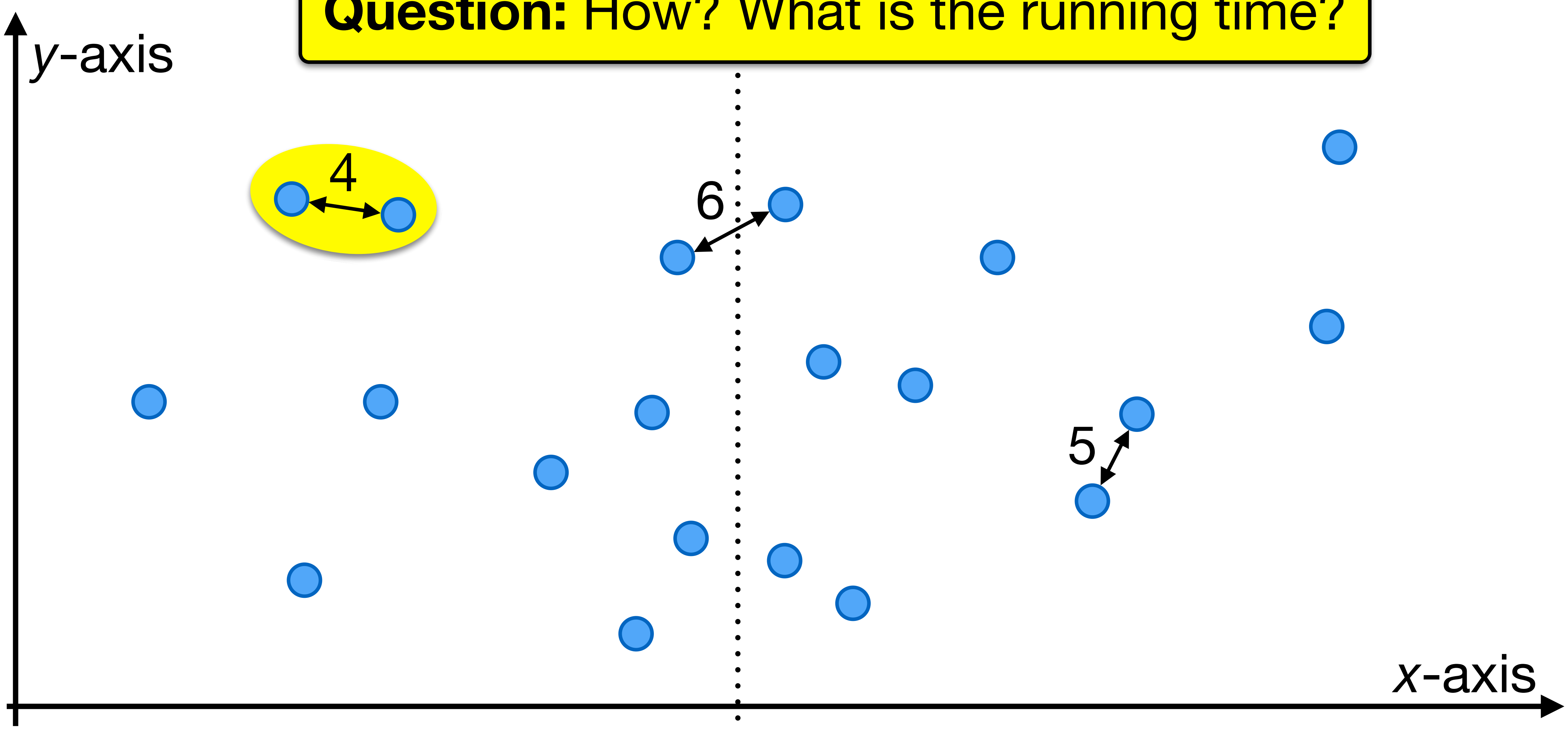


A Divide and Conquer Algorithm for Closest Pair

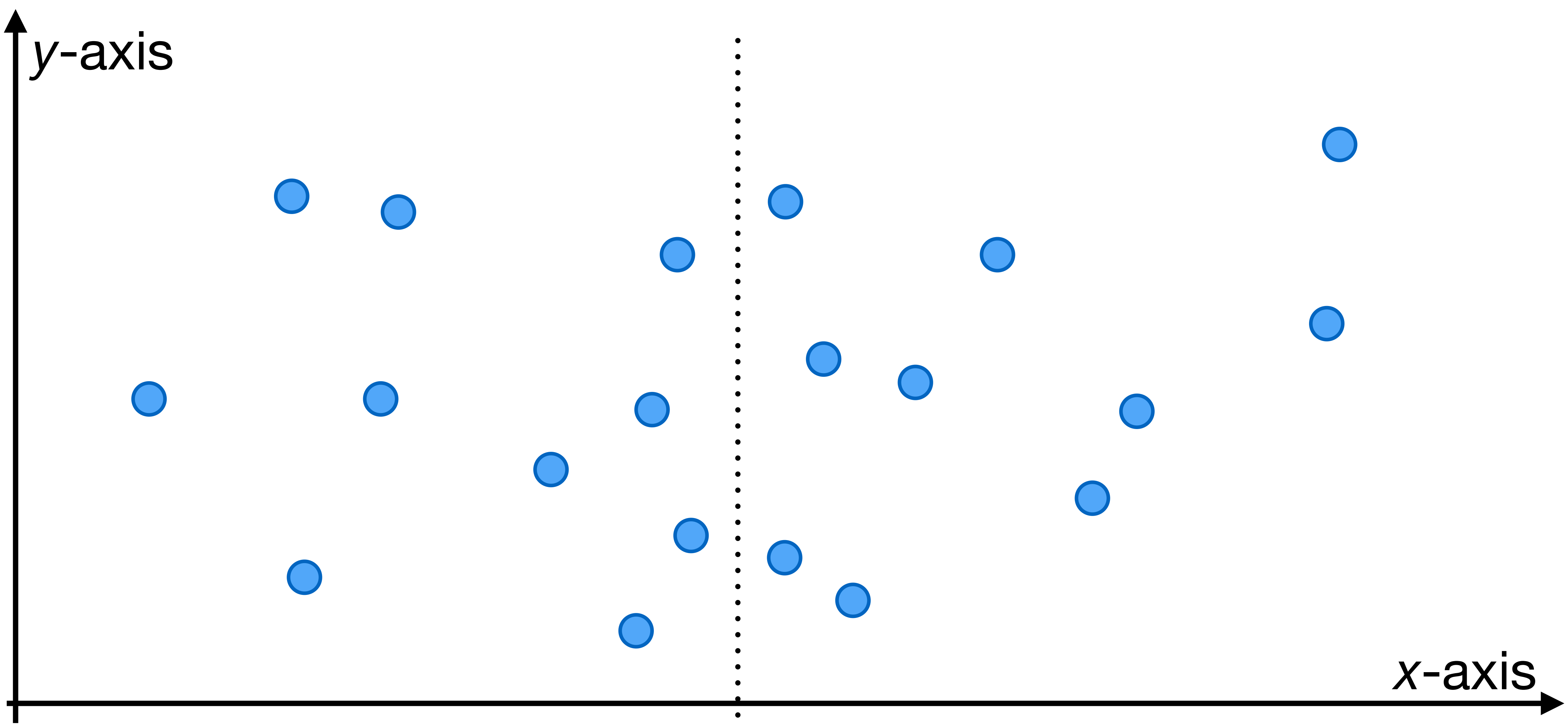
Recurse: Find the closest pair on each side.

Combine: Find the closest pair with one point on each side.
Output the closest of the three pairs.

Question: How? What is the running time?

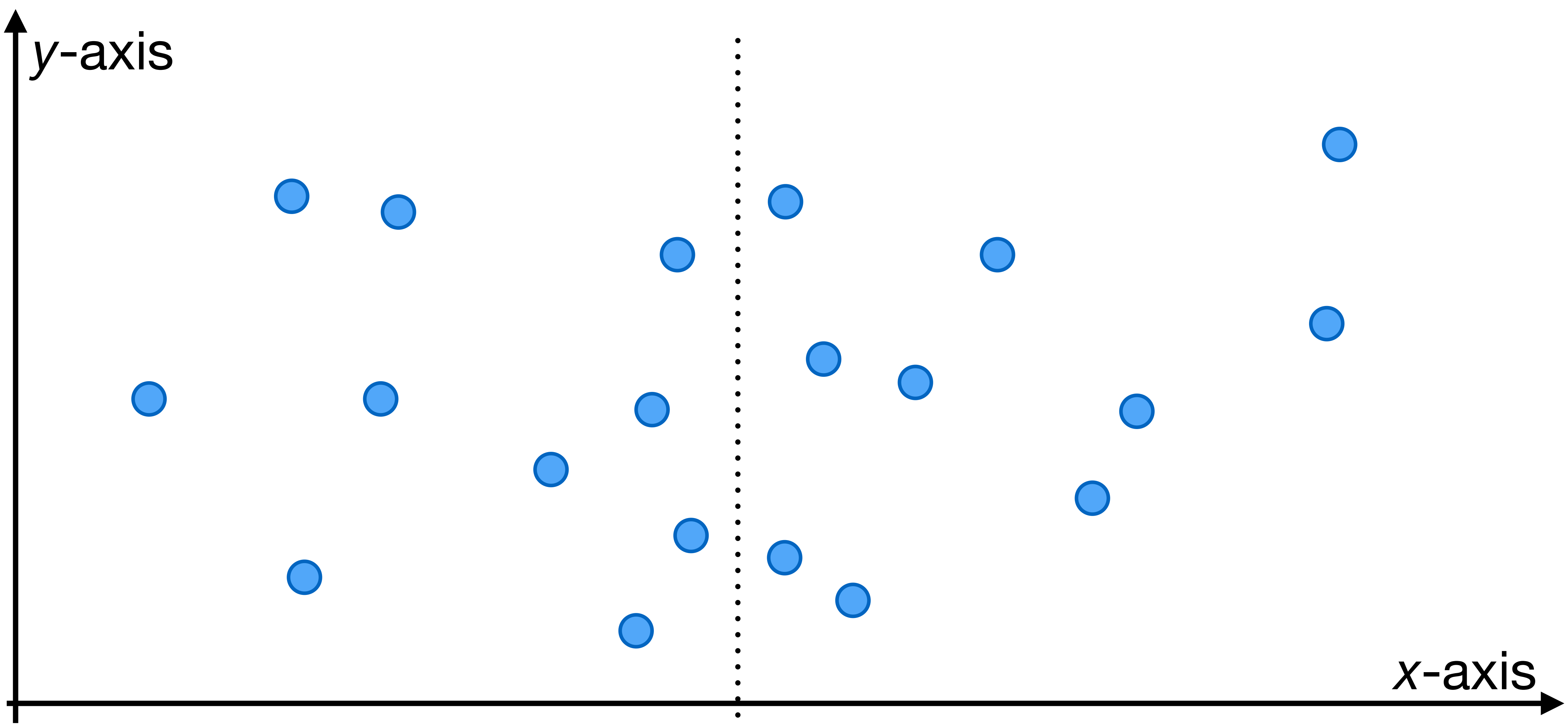


Closest pair with one point on each side



Closest pair with one point on each side

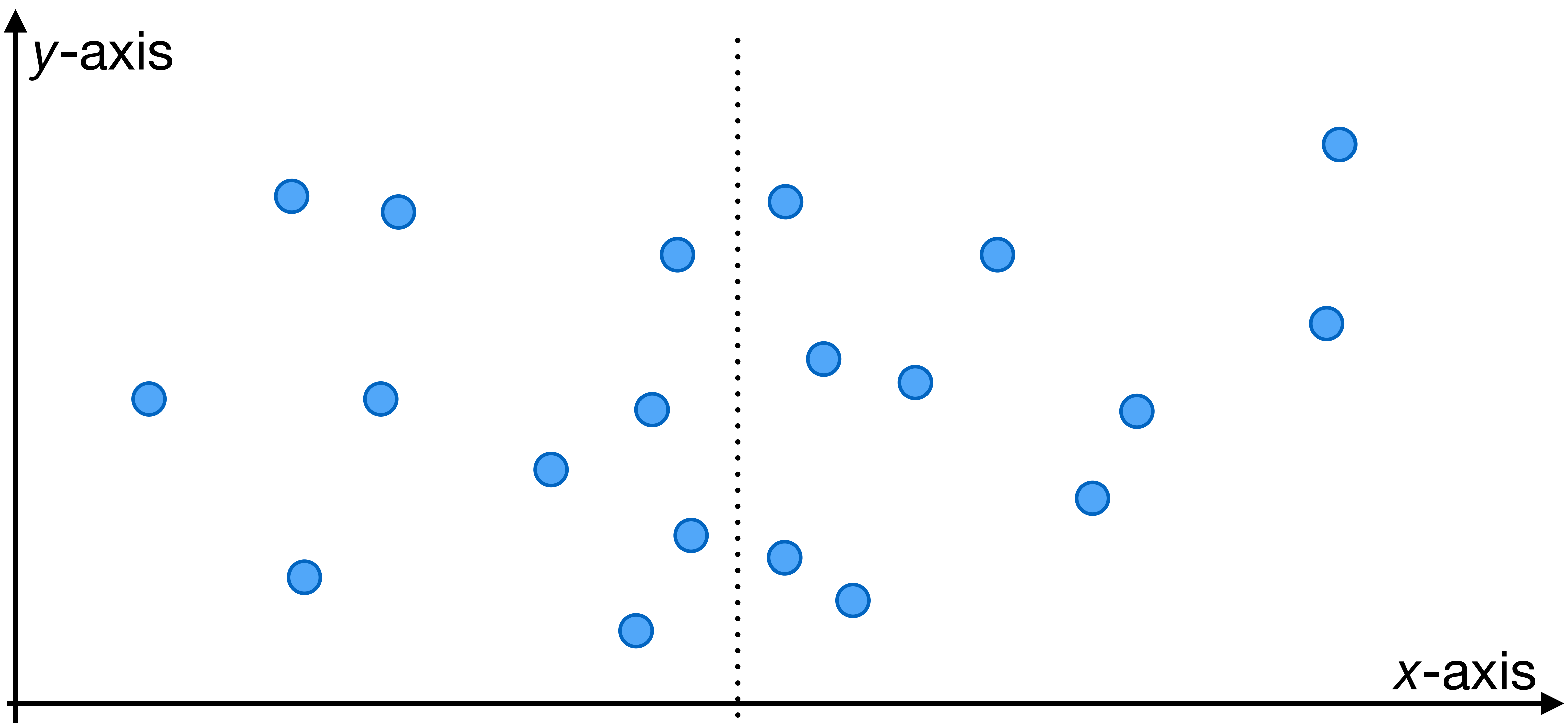
A straightforward brute-force approach: Compare all $(n/2)^2$ pairs with one point on each side, and return the smallest one.



Closest pair with one point on each side

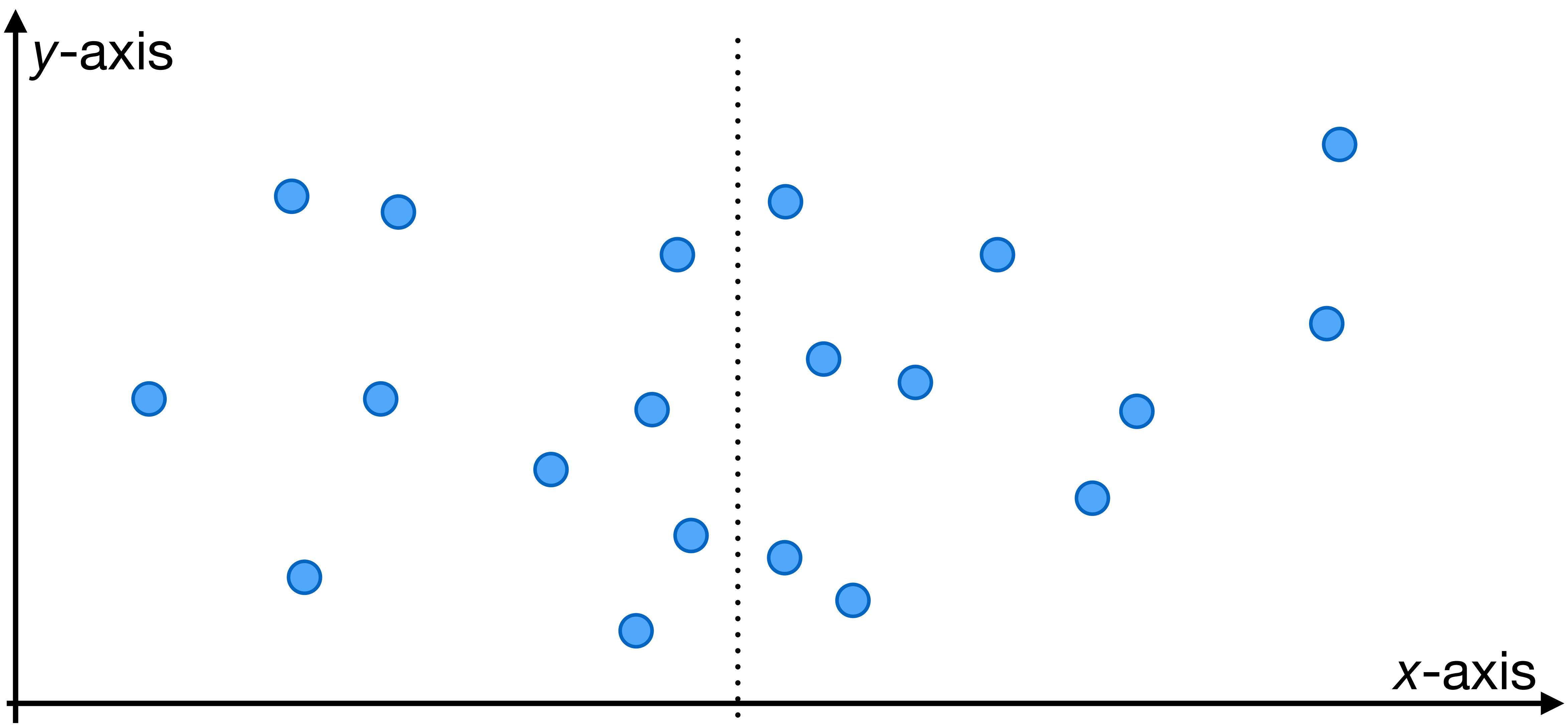
A straightforward brute-force approach: Compare all $(n/2)^2$ pairs with one point on each side, and return the smallest one.

1) **Divide** takes $O(n \log n)$ time; 2) **Recurse** takes $2\,T(n/2)$ time; 3) **Combine** takes $O(n^2)$ time. So $T(n) = 2\,T(n/2) + O(n^2) = O(n^2)$.



Closest pair with one point on each side

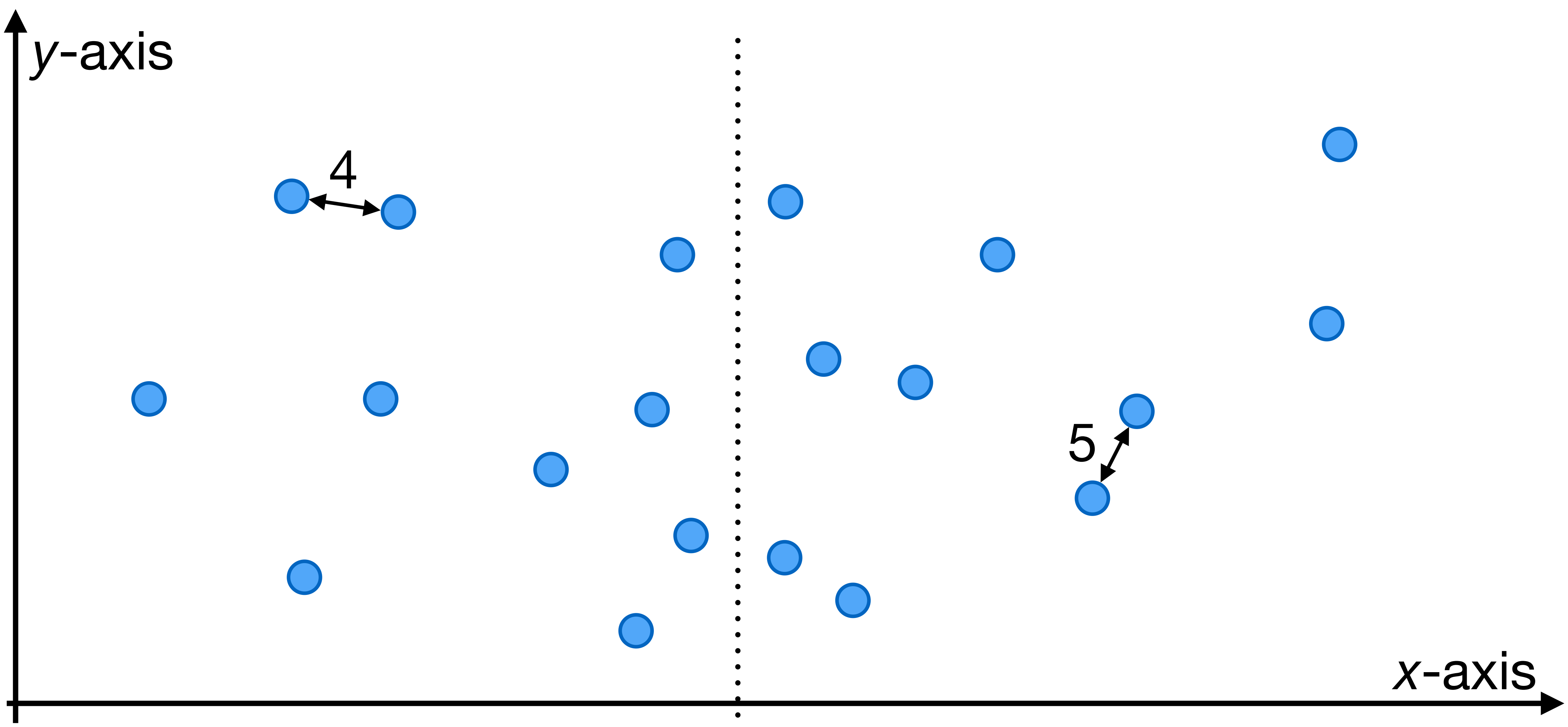
Let δ_L and δ_R be the distance of the closest pairs on the left and on the right respectively. Let δ = *min* (δ_L, δ_R).



Closest pair with one point on each side

Let δ_L and δ_R be the distance of the closest pairs on the left and on the right respectively. Let $\delta = min\ (\delta_L, \delta_R)$.

Example: $\delta_L = 4$, $\delta_R = 5$, and $\delta = 4$.

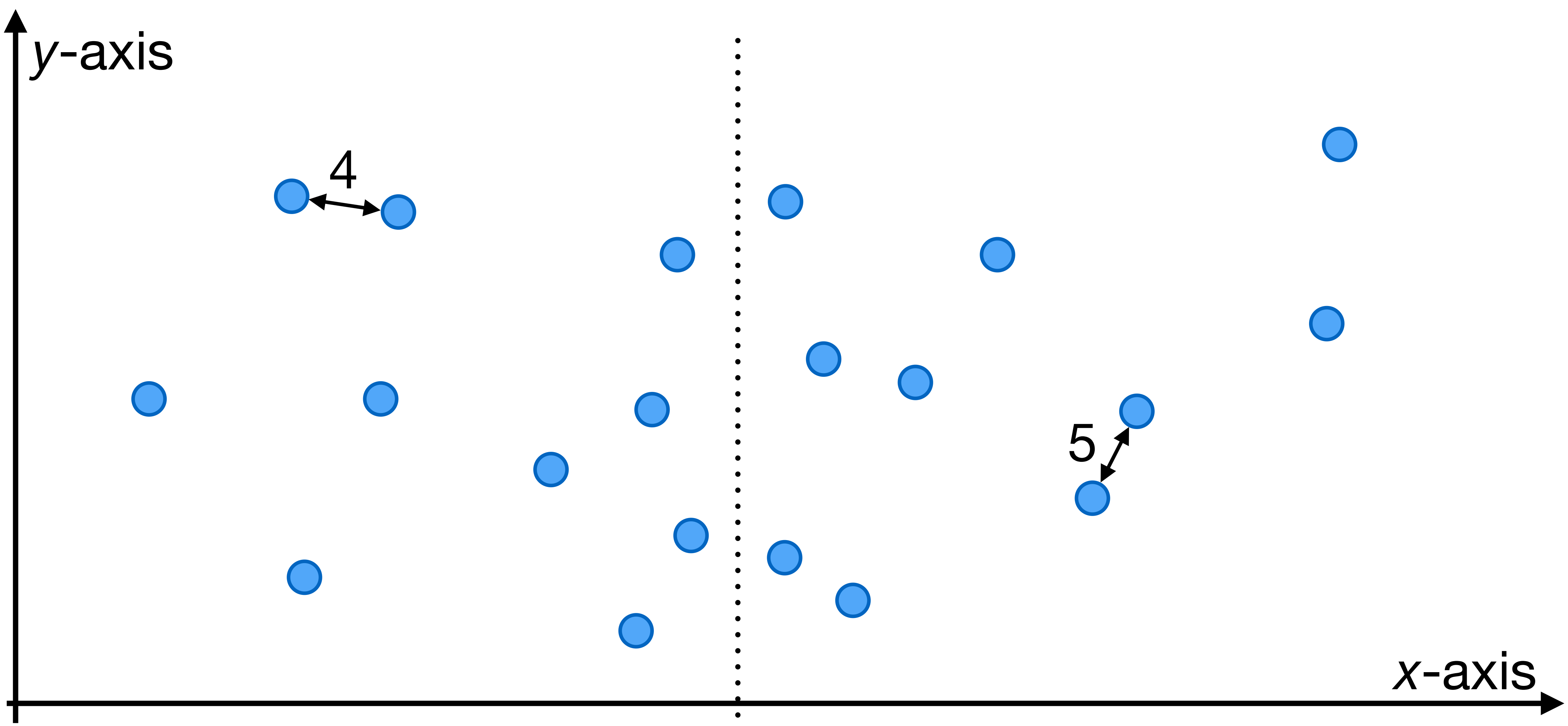


Closest pair with one point on each side

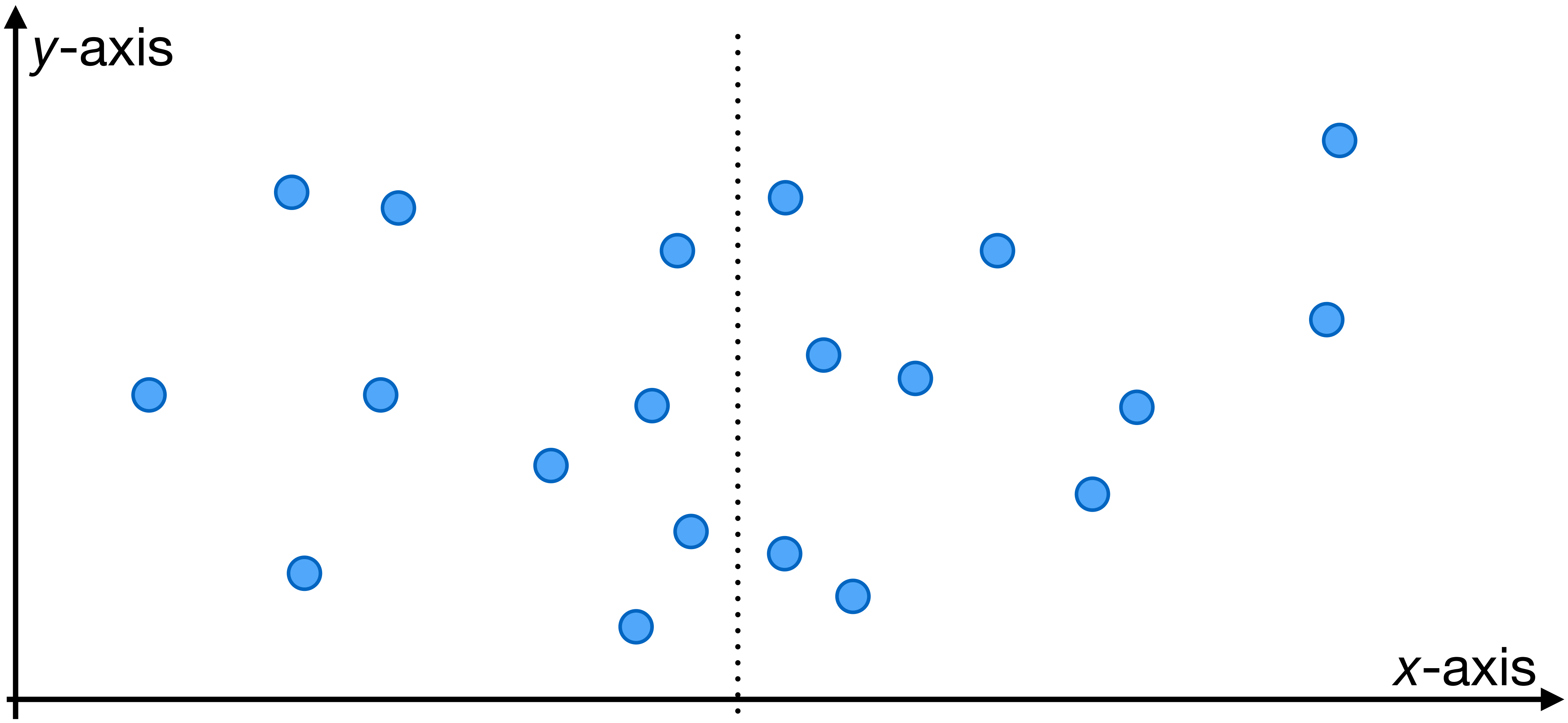
Let δ_L and δ_R be the distance of the closest pairs on the left and on the right respectively. Let $\delta = min\ (\delta_L, \delta_R)$.

Example: $\delta_L = 4$, $\delta_R = 5$, and $\delta = 4$.

Idea: Focus on pairs with one point in each side and has distance $< \delta$.

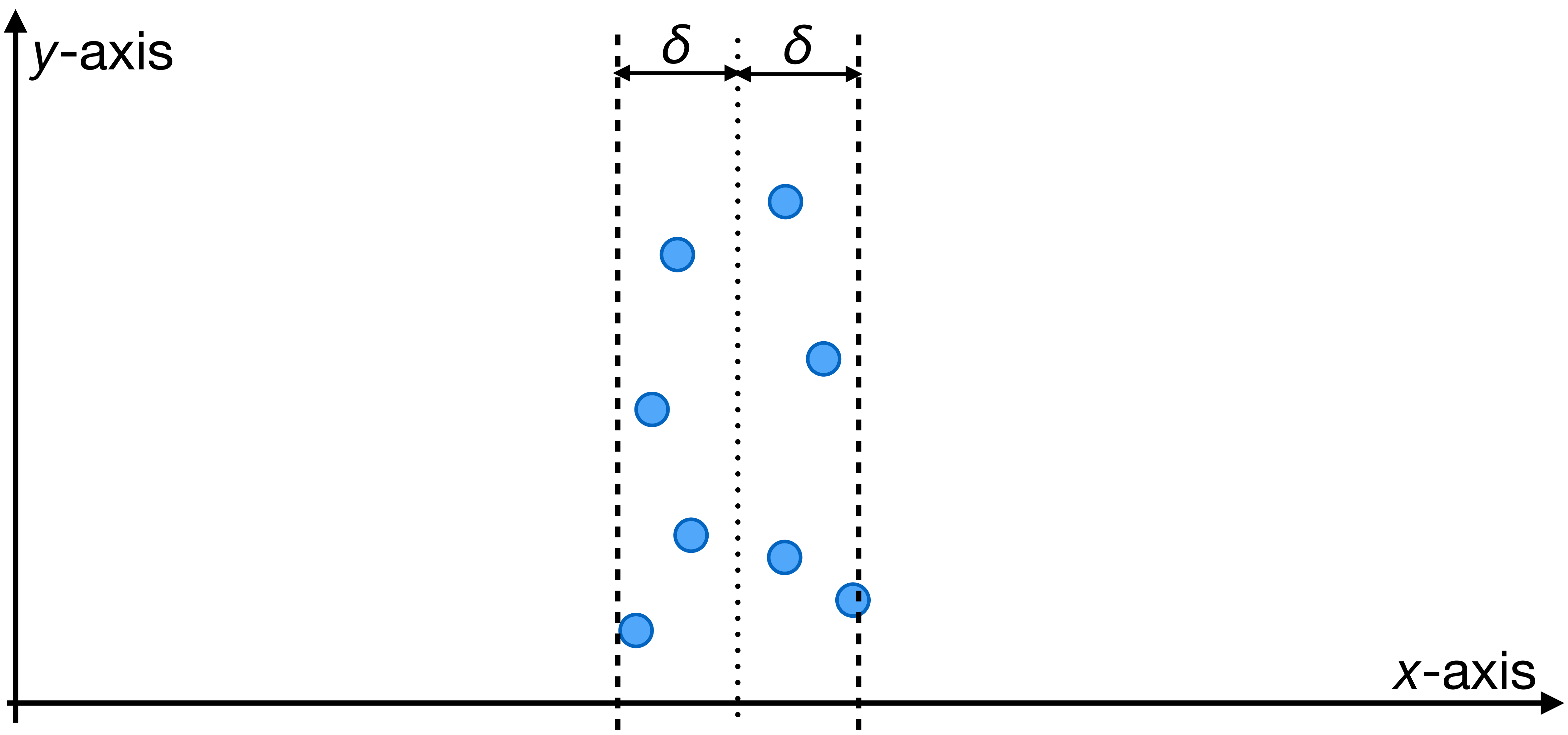


Closest pair with one point on each side



Closest pair with one point on each side

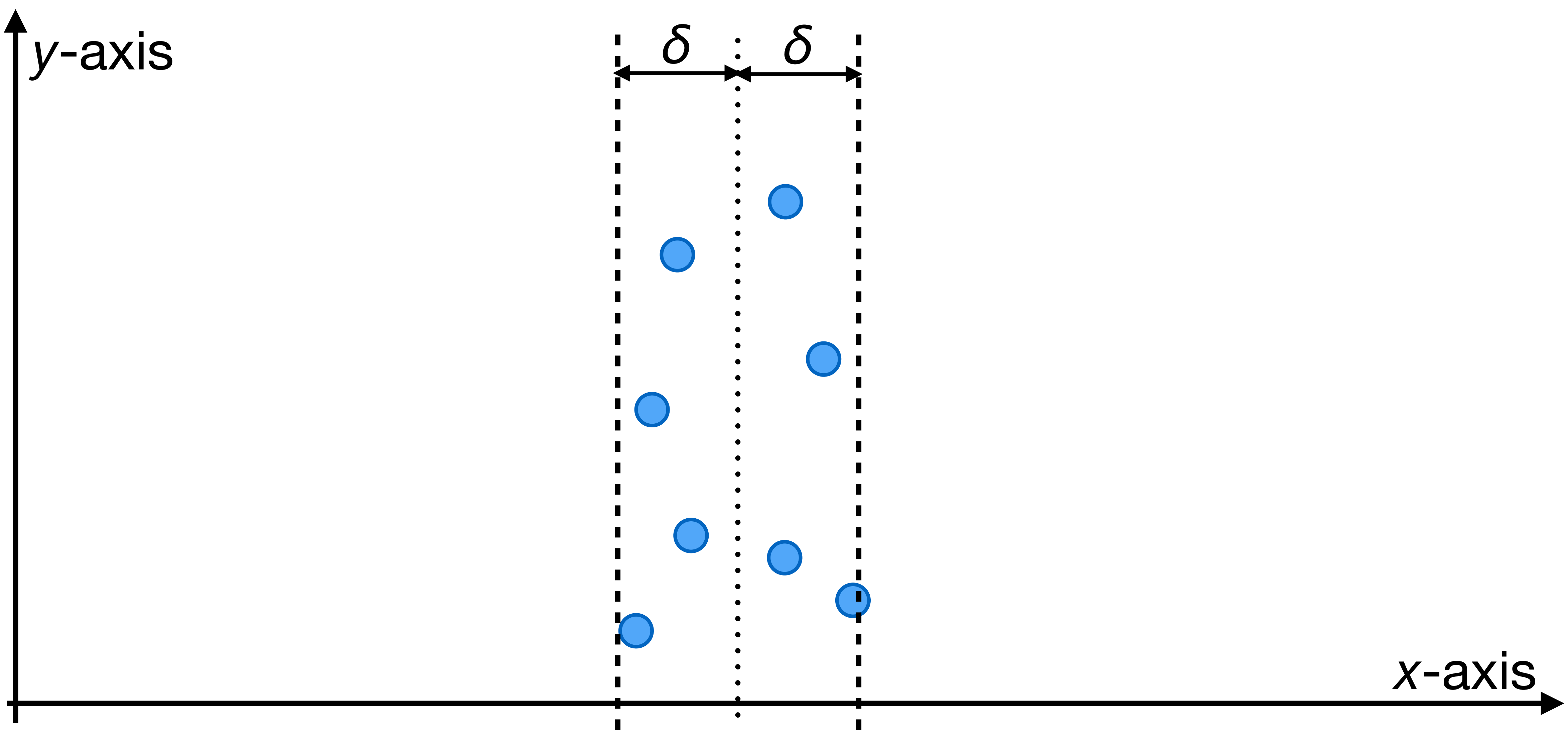
Note: We only need to consider points within δ of the dividing line.



Closest pair with one point on each side

Note: We only need to consider points within δ of the dividing line.

1) Sort points in the 2δ-strip in ascending order of the y-coordinate.

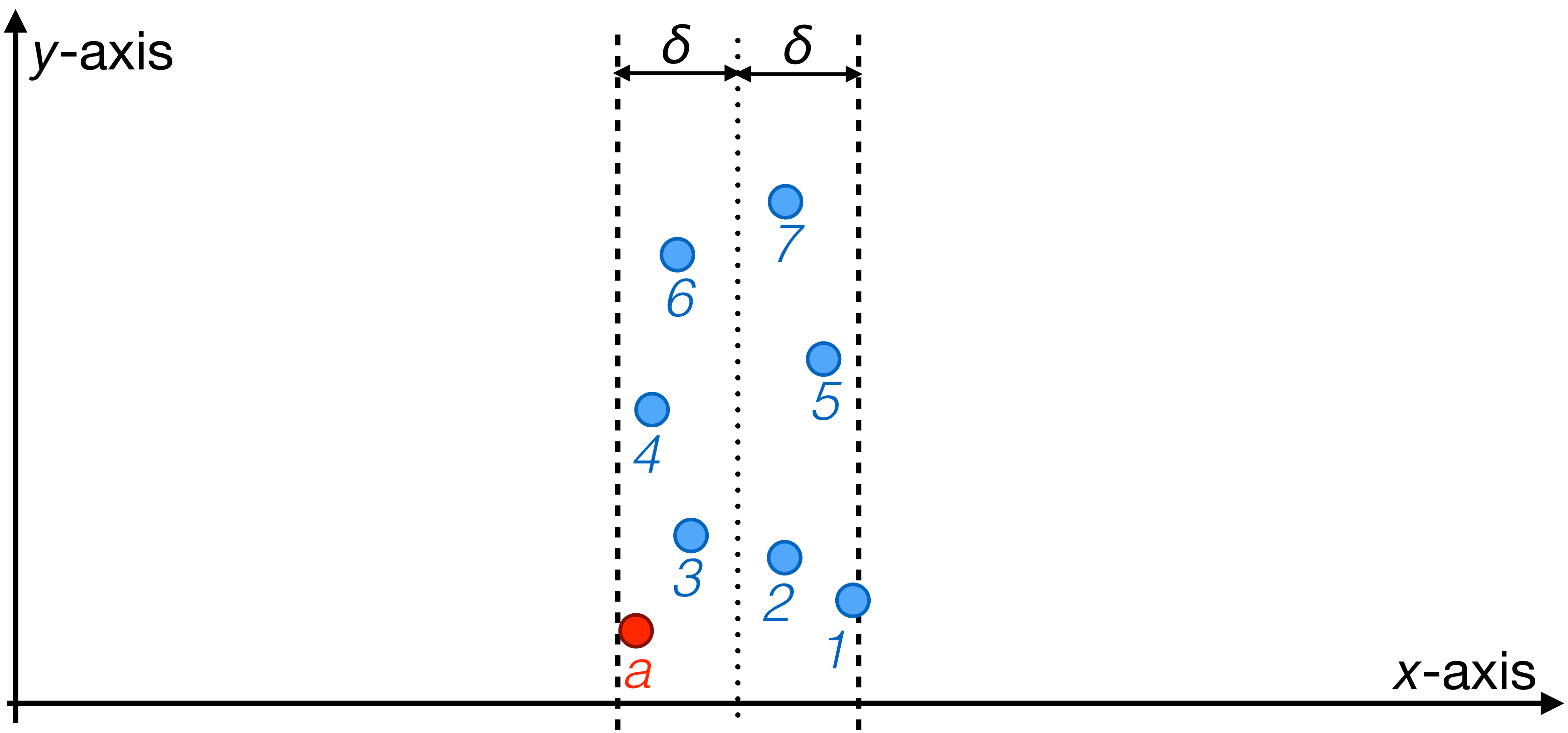


Closest pair with one point on each side

Note: We only need to consider points within δ of the dividing line.

1) Sort points in the 2δ-strip in ascending order of the y-coordinate.

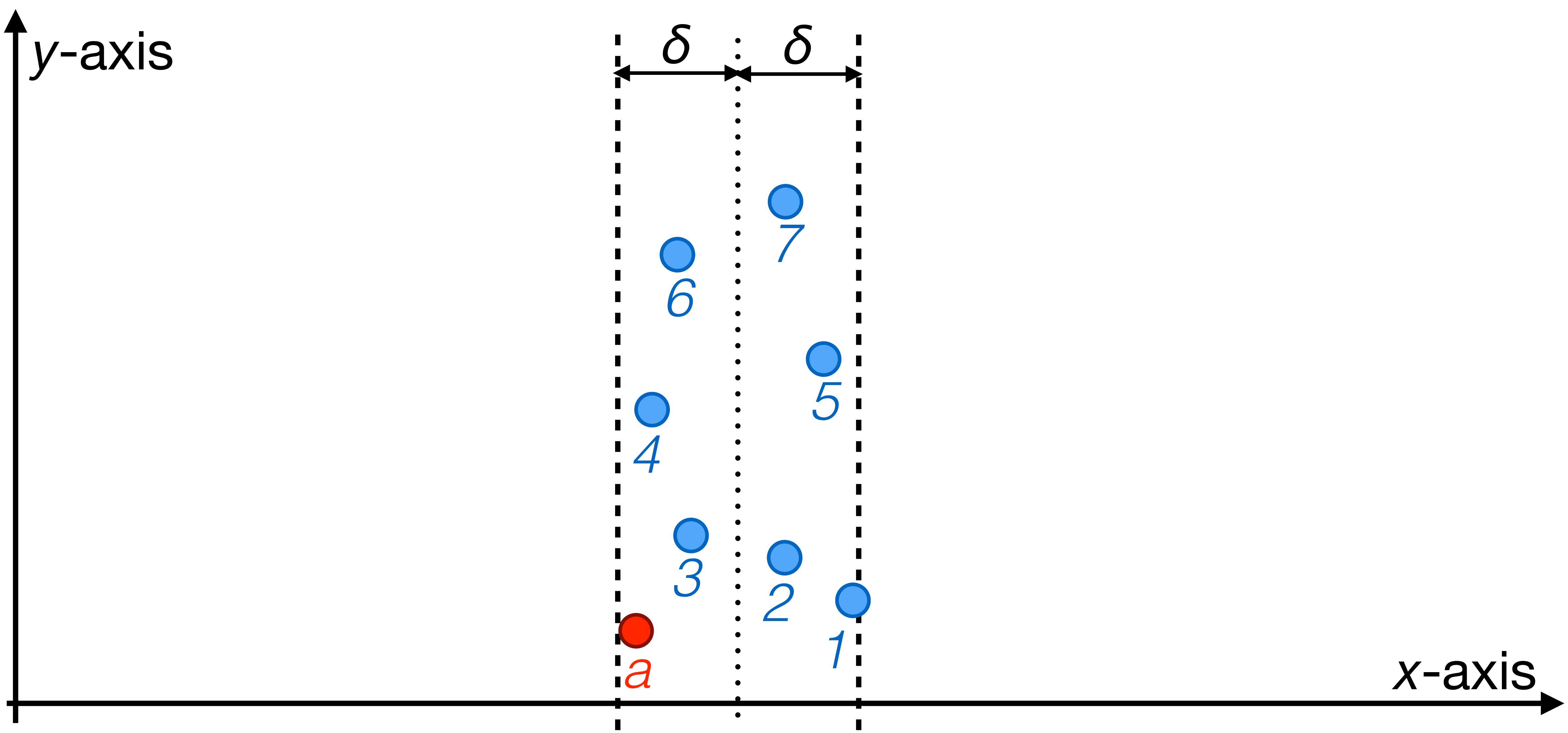
2) For each point a, check the distances to its 7 subsequent points.



Closest pair with one point on each side

Note: We only need to consider points within δ of the dividing line.

1) Sort points in the 2δ-strip in ascending order of the y-coordinate.
2) For each point a, check the distances to its 7 subsequent points.
3) Output the closest pair found in step 2.



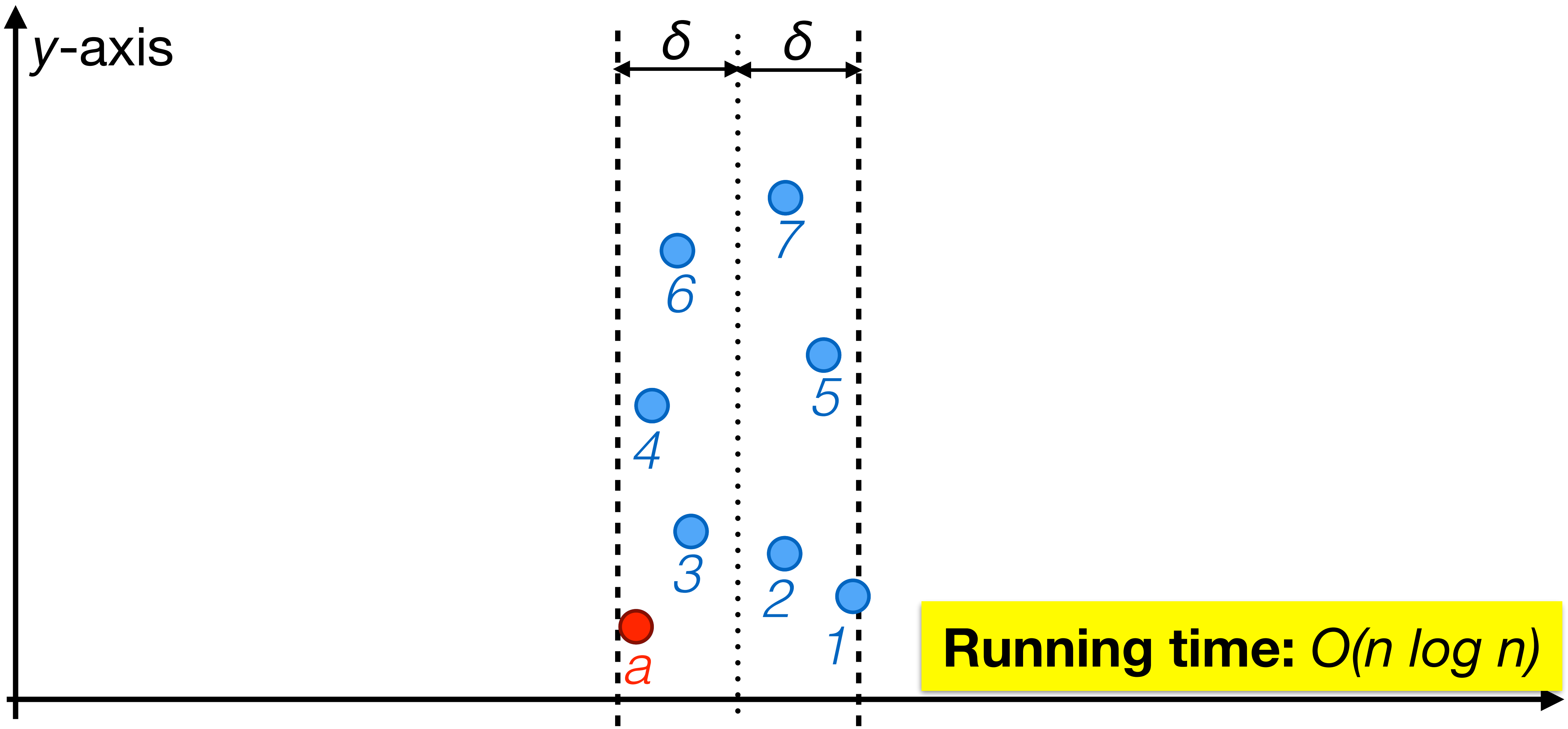
Closest pair with one point on each side

Note: We only need to consider points within δ of the dividing line.

1) Sort points in the 2δ-strip in ascending order of the y-coordinate.

2) For each point a, check the distances to its 7 subsequent points.

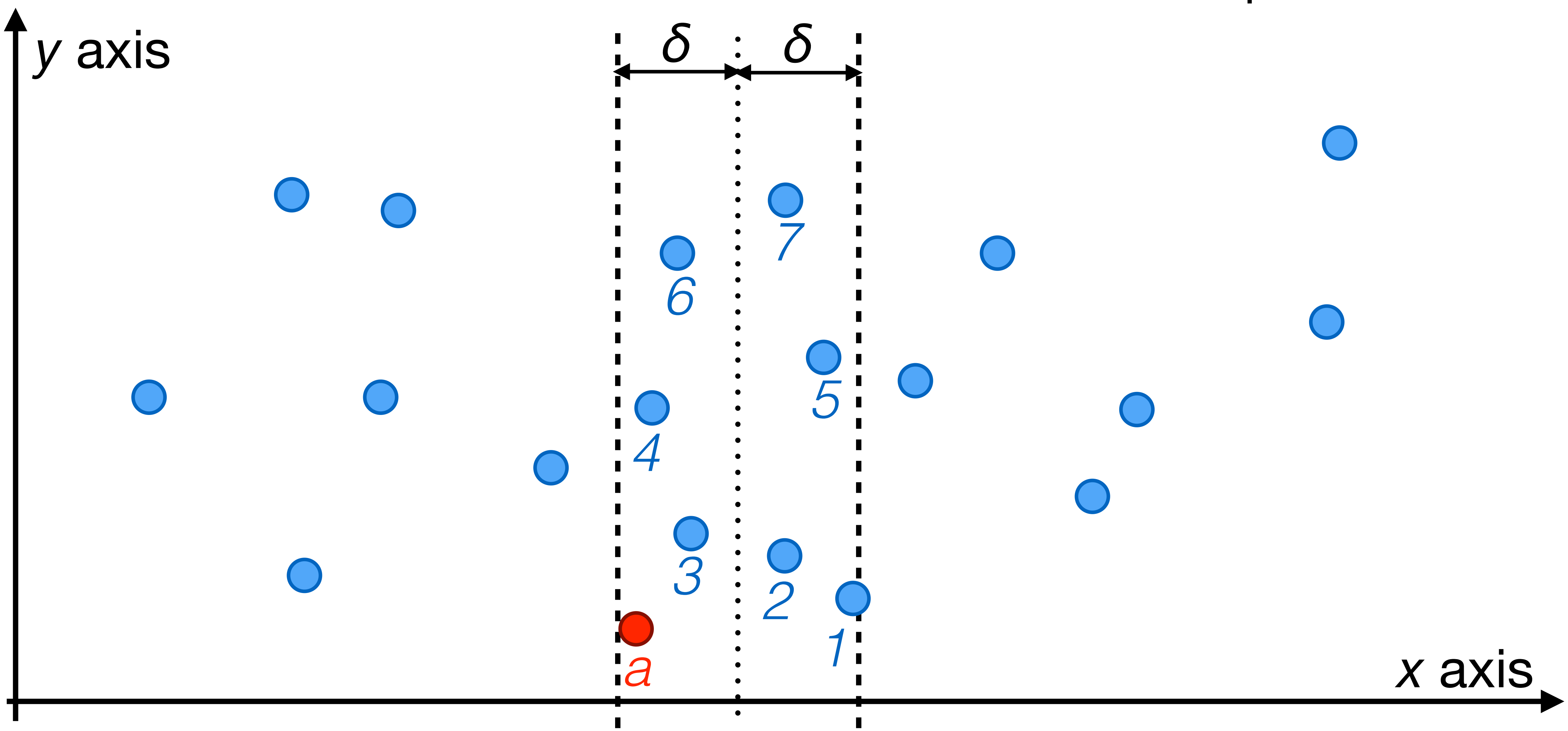
3) Output the closest pair found in step 2.



Running time: $O(n \log n)$

Why is it correct?

- Let *a* and *b* be a pair of points with one point on each side such that their distance is ≤ δ, and *a* is lower than *b* in the y-coordinate.
- We will prove that *b* is among the 7 subsequent points of *a* in the sorted list, i.e., $b \in \{1, 2, 3, 4, 5, 6, 7\}$. Then, the algorithm would have checked and remembered their distance in step 2.



Why b must be in $\{1, 2, 3, 4, 5, 6, 7\}$?

Observation 1: There are at most *4* points in any square of size δ on the left of the dividing line.

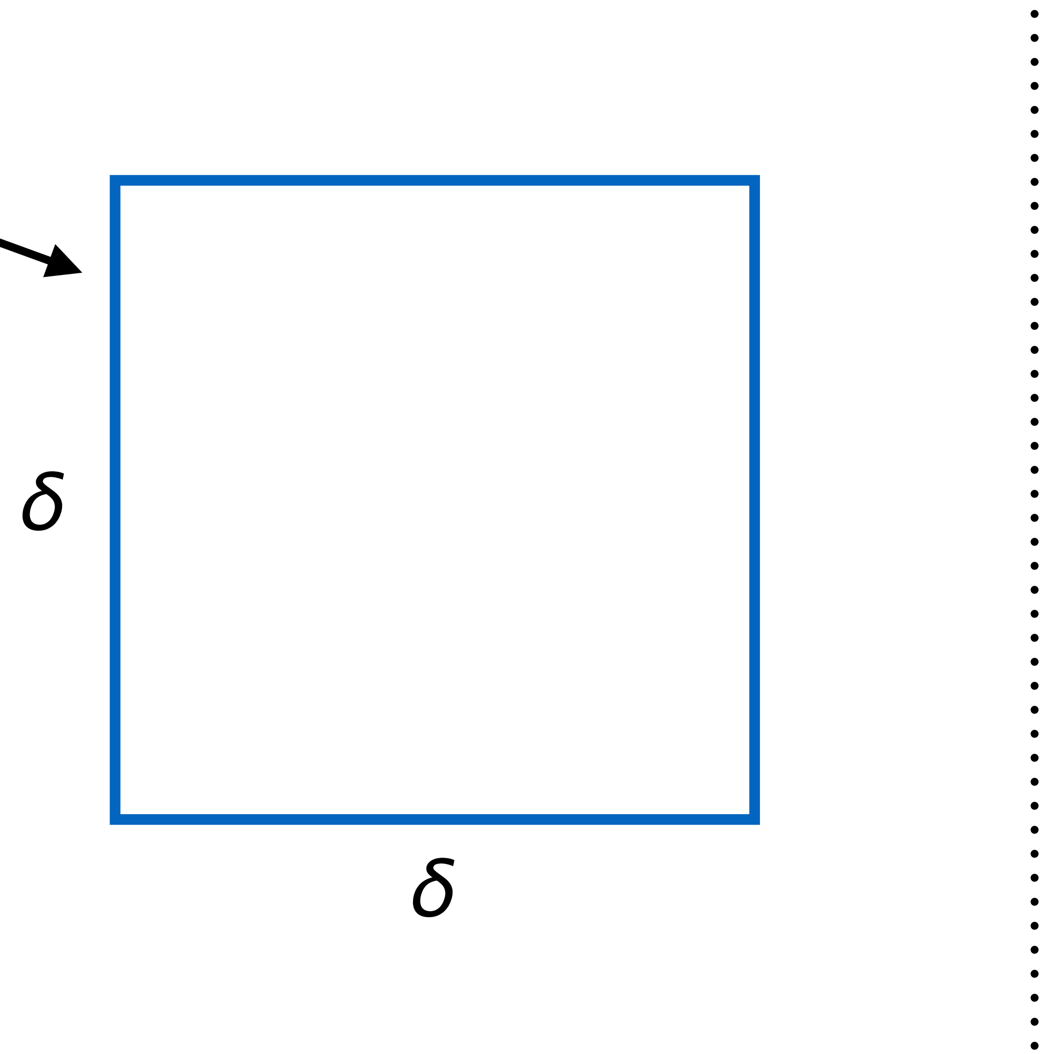
- Why? Recall that $\delta = min(\delta_L, \delta_R)$. Thus, $\delta \leq \delta_L$.

Why *b* must be in *{1, 2, 3, 4, 5, 6, 7}*?

Observation 1: There are at most *4* points in any square of size δ on the left of the dividing line.

- Why? Recall that $\delta = min\,(\delta_L, \delta_R)$. Thus, $\delta \leq \delta_\mathrm{L}$.

1) Consider any square of size δ on the left the dividing line

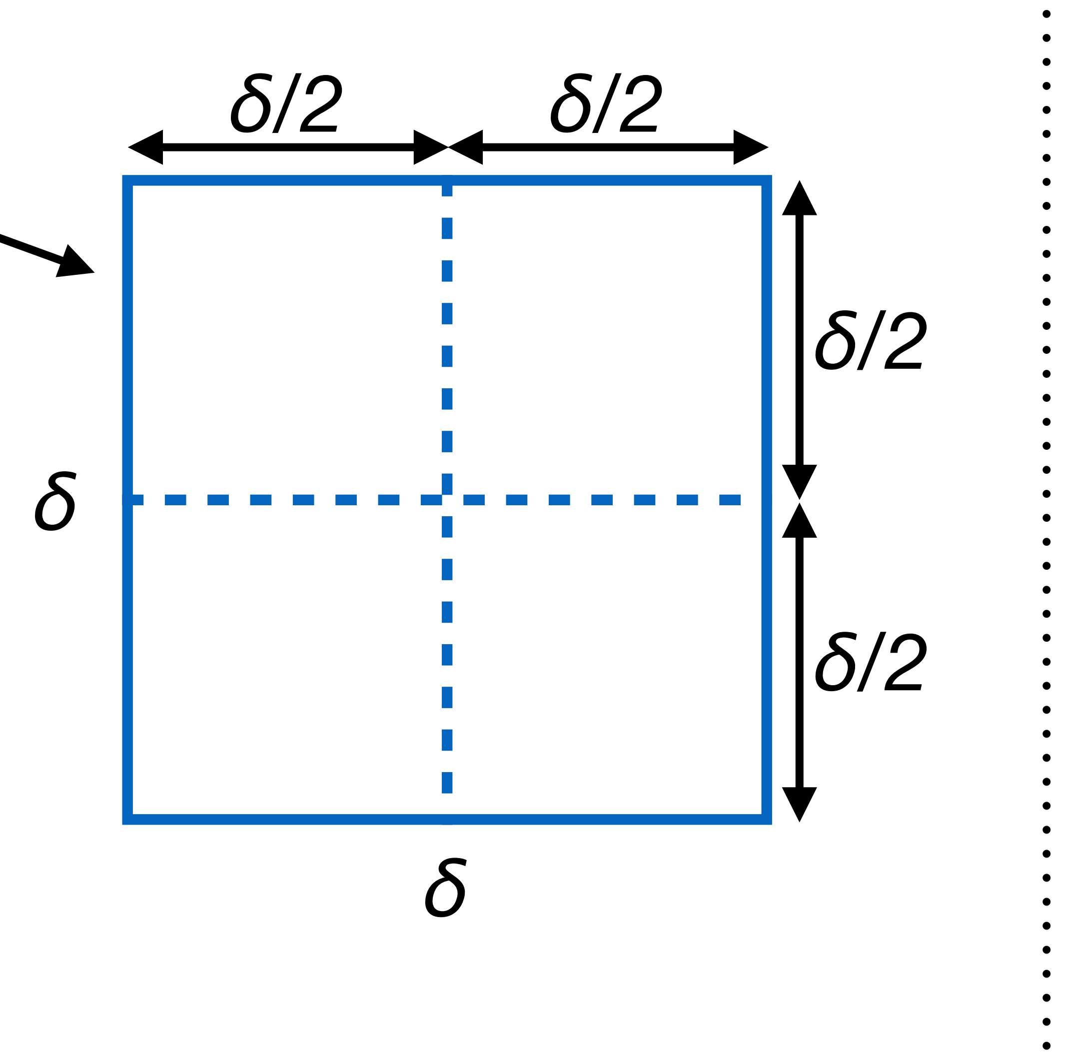


Why *b* must be in *{1, 2, 3, 4, 5, 6, 7}*?

Observation 1: There are at most *4* points in any square of size δ on the left of the dividing line.

- Why? Recall that $\delta = min(\delta_L, \delta_R)$. Thus, $\delta \leq \delta_L$.

1) Consider any square of size δ on the left the dividing line
2) Divide the square into 4 sub-squares of size $\delta/2$

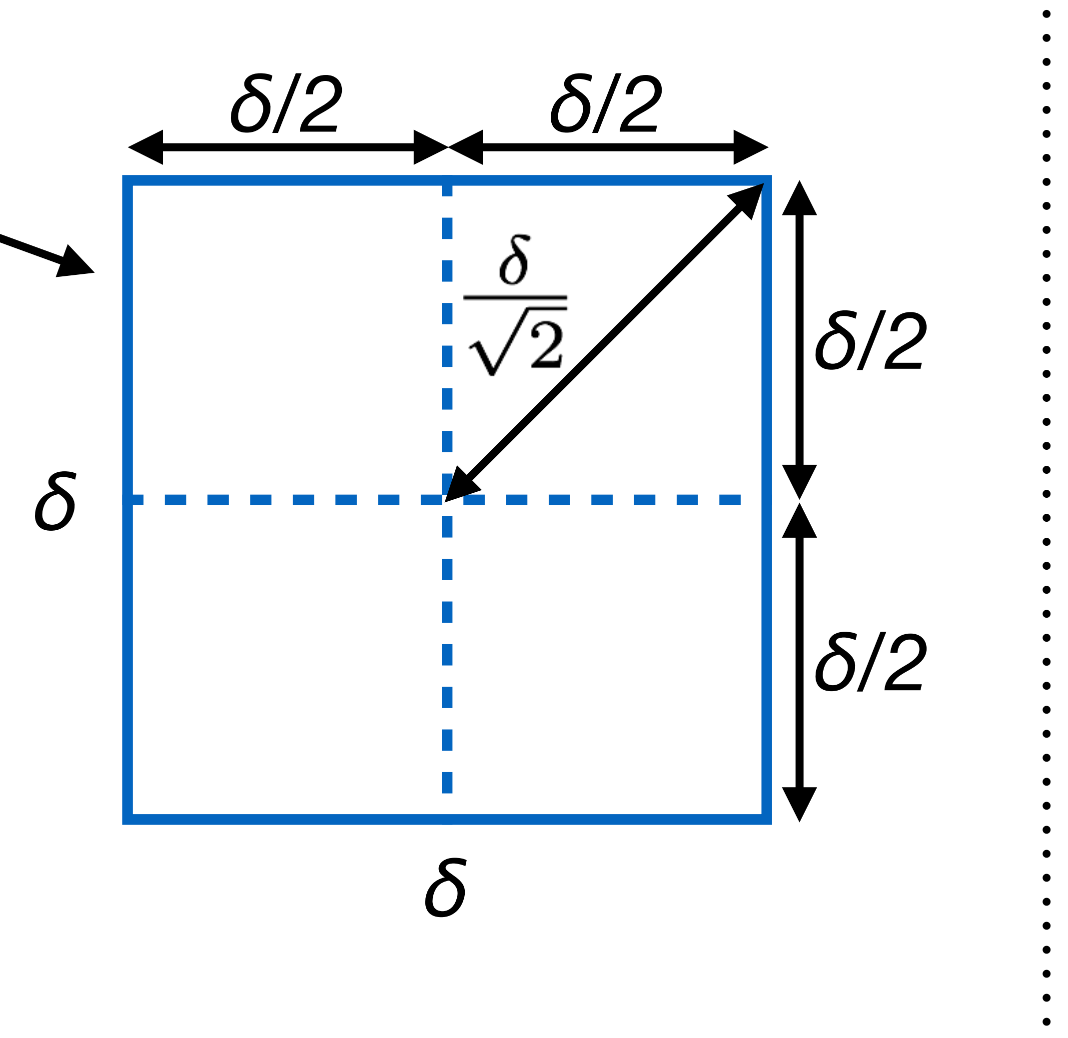


Why *b* must be in *{1, 2, 3, 4, 5, 6, 7}*?

Observation 1: There are at most *4* points in any square of size δ on the left of the dividing line.

- Why? Recall that $\delta = min\,(\delta_L, \delta_R)$. Thus, $\delta \leq \delta_L$.

1) Consider any square of size δ on the left the dividing line
2) Divide the square into 4 sub-squares of size $\delta/2$
3) Points in the same sub-square are at most $\frac{\delta}{\sqrt{2}} < \delta \leq \delta_L$ apart.

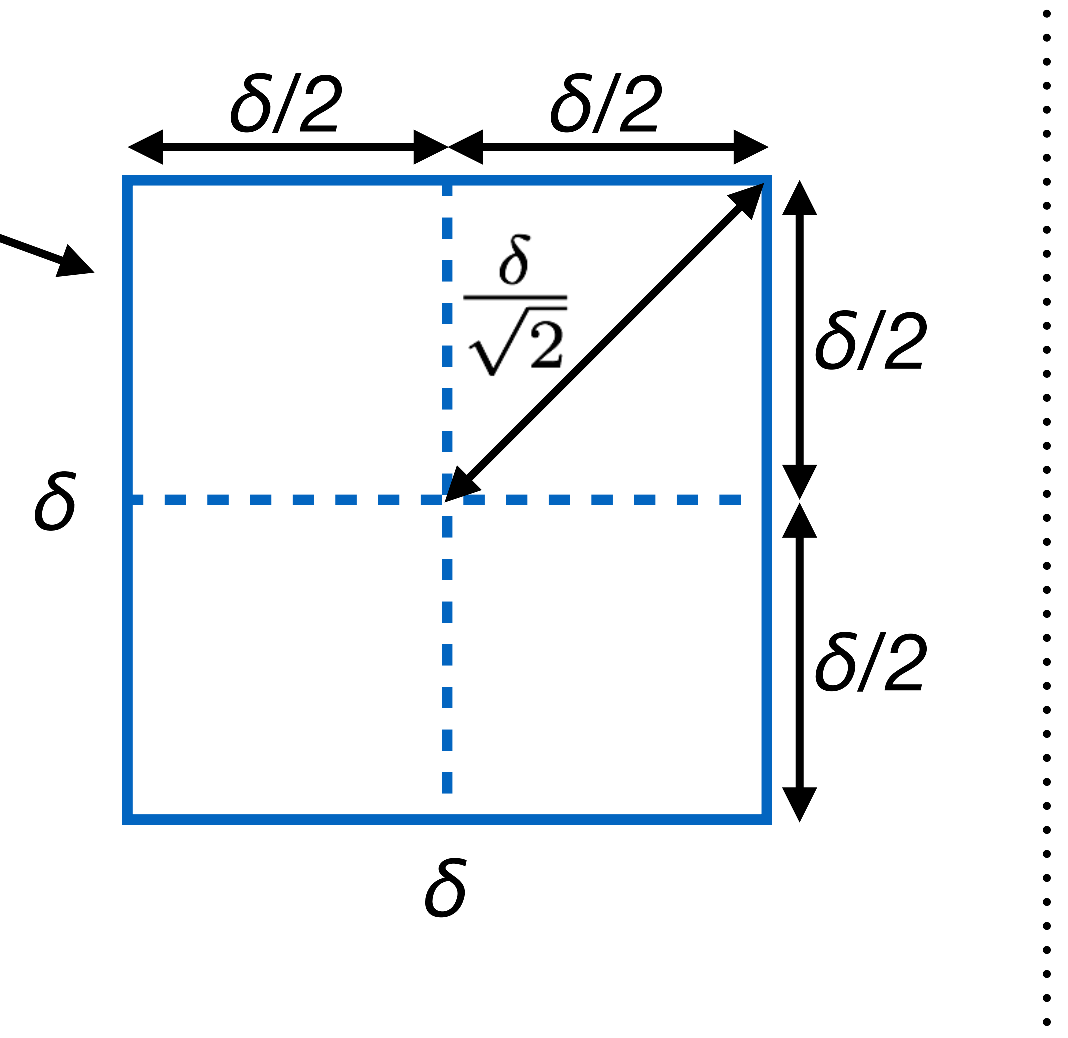


Why b must be in $\{1, 2, 3, 4, 5, 6, 7\}$?

Observation 1: There are at most 4 points in any square of size δ on the left of the dividing line.

- Why? Recall that $\delta = min\,(\delta_L, \delta_R)$. Thus, $\delta \leq \delta_L$.

1) Consider any square of size δ on the left the dividing line
2) Divide the square into 4 sub-squares of size $\delta/2$
3) Points in the same sub-square are at most $\frac{\delta}{\sqrt{2}} < \delta \leq \delta_L$ apart.
4) Points on the left of the dividing line are at least δ_L apart. So there are ≤ 1 point in each sub-square, and ≤ 4 points in the square.

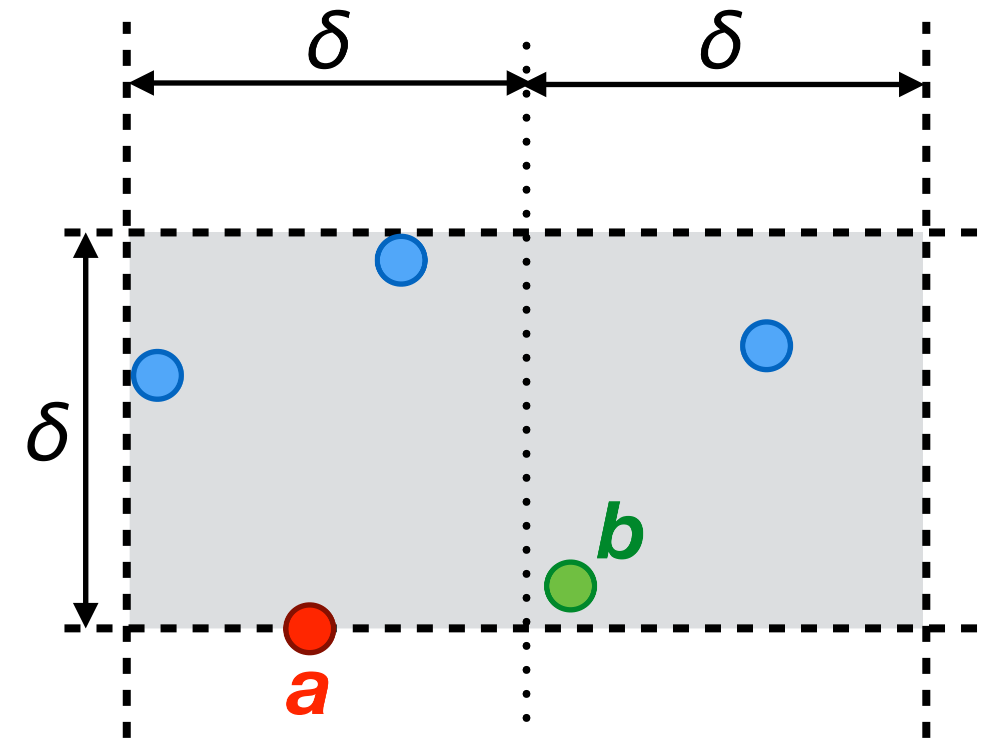


Why *b* must be in *{1, 2, 3, 4, 5, 6, 7}*?

Observation 1: There are at most *4* points in any square of size δ on the left of the dividing line.

Observation 2: There are at most *4* points in any square of size δ on the right of the dividing line. (Same argument)

1) Recall that the distance between *a* and *b* is $\leq \delta$ and *a* is lower than *b* in the y-coordinate.
2) *b* must be in the shaded area, which is comprised of two squares of size δ.
3) There are ≤ 4 points in each square, and thus ≤ 8 points in the shaded area.
4) There are ≤ 7 points in the shaded area other than point *a*. So *b* must be one of *1, 2, 3, 4, 5, 6, 7*.



An $O(n \log^2 n)$ Time Divide and Conquer Algorithm for Closest Pair

Divide:

1) Sort the points by their x-coordinates.
2) Draw a vertical line L so that $n/2$ points on each side.

Recurse:

3) Find the closest pair on the left of L, let δ_L be the distance.
4) Find the closest pair on the right of L, let δ_R be the distance.

Combine:

5) Let $\delta = min\,(\delta_L, \delta_R)$.
6) Let S be the set of points that are at most δ from L.
7) Sort points in S in the y-coordinate and check the distance between each point and next 7 points.
8) Return the closest pair among step 3, 4, and 7

Running Time Analysis

- How to analyze $T(n)$?
  - Divide step takes $O(n \log n)$ time (bottleneck is sorting).
  - Recurse step take $2\,T(n/2)$ time.
  - Combine step takes $O(n \log n)$ time (bottleneck is sorting).
- $T(n) = 2\,T(n/2) + O(n \log n) = O(n \log^2 n)$
  - **Intuition:**
    - If $T(n) = 2\,T(n/2) + O(n)$, then $T(n) = O(n \log n)$.
    - The extra log factor in the recurrence relation becomes an extra log factor in the final answer.
  - Note that we cannot directly use the Master theorem here.
  - We can prove it either by repeatedly expanding T(.) using the recurrence relation, or by mathematical induction.

Optional Reading

- We can actually implement the same algorithm in O(n log n) time, with some extra efforts
- See, e.g., the slides below: https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap15d.pdf
- YouTube video by Tim Roughgarden: https://www.youtube.com/watch?v=jAigdwcATNw
- There is a ton of other resources available online