

Closest Points

CS/MCS 401: Computer Algorithms I

Lecture 8

Professor: Jonathan Liu

Introduce yourself!

Icebreaker: How do you feel about rain?

Lesson Goals

Today we're going to design an algorithm together. Unlike the previous lecture, where I was more or less telling you what the algorithm is, in this lesson we're going to walk through how to go from a problem to a solution.

Pay particular attention to the questions we ask and the decisions we make.

The Problem: Closest Shops



Problem 9: Closest Shops

Times are tough, so you need to close a shop. Find the two shops that are closest to one another so that one can be removed. You only know each shop's coordinates, and you're calculating distance directly, not by driving distance.

Problem Modeling

What is the input data?

A list of every shop's coordinates.

What is the output data?

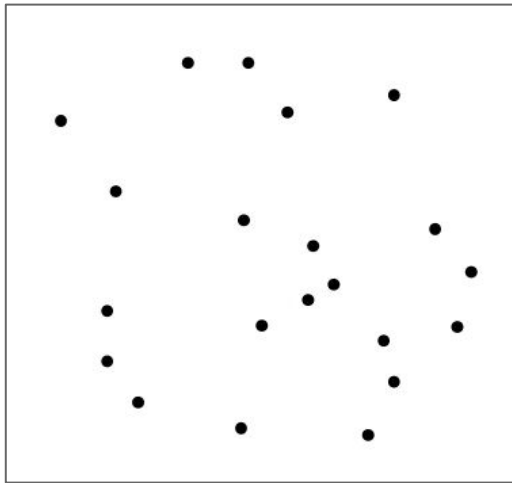
The two closest shops.



Closest Points

Broadly applicable problem in computational geometry! Helps when modeling all kinds of things – molecules, cities, planets.

Only 2 of you guessed that this and peak-finding would be the same category.

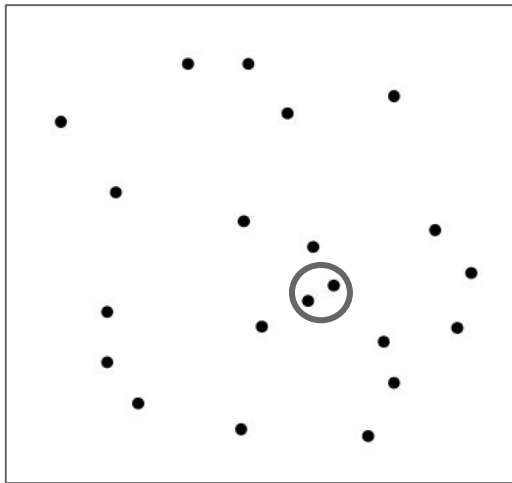


Note: Images mostly from Kevin Wayne

Closest Points

Broadly applicable problem in computational geometry! Helps when modeling all kinds of things – molecules, cities, planets.

Only 2 of you guessed that this and peak-finding would be the same category.



Note: Images mostly from Kevin Wayne

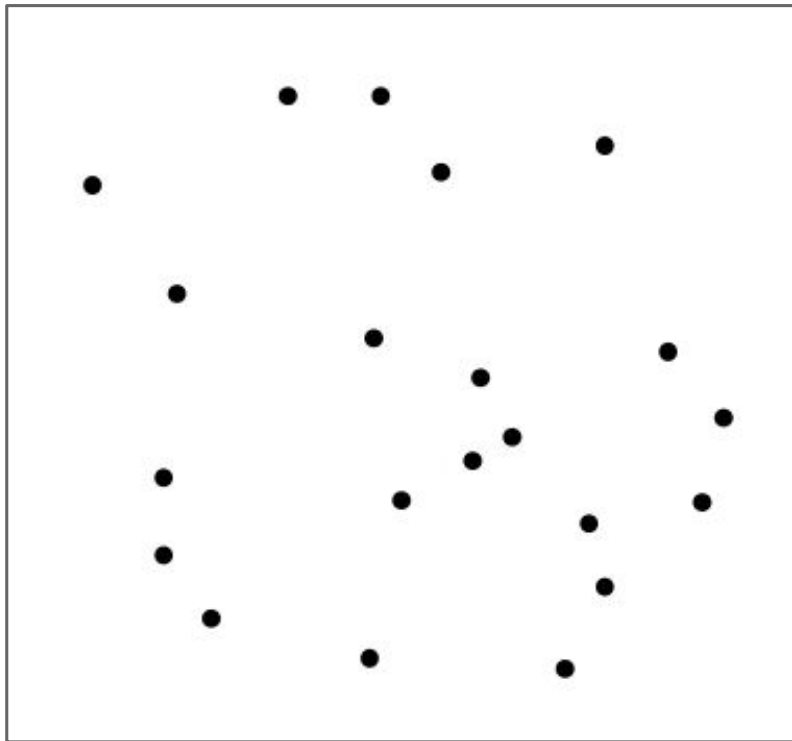
Problem Specifics

We'll assume that no points share even one coordinate – so no two points have the same x-coordinate or y-coordinate. (it's not all that different if they do)

How long does brute force take?

For every point:

Check the distance to all other points. → $O(n^2)$



Paradigm Selection

Obviously we're going to do divide and conquer here. What about the problem makes divide and conquer potentially a good idea?

If we split the input in half and found the closest pair in each half, it seems pretty likely that one of those would be the closest pairs in the whole input!

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.

A First Attempt

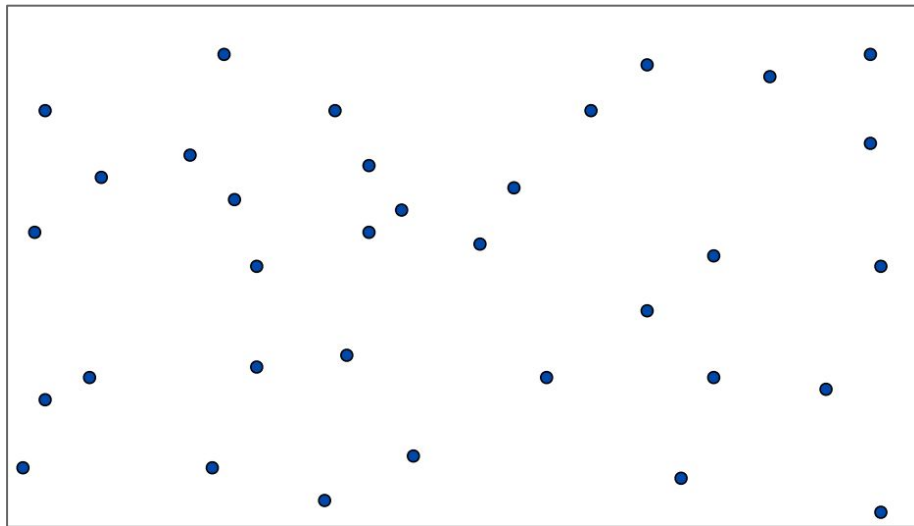


A First Attempt

Divide: How should we try to divide?

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



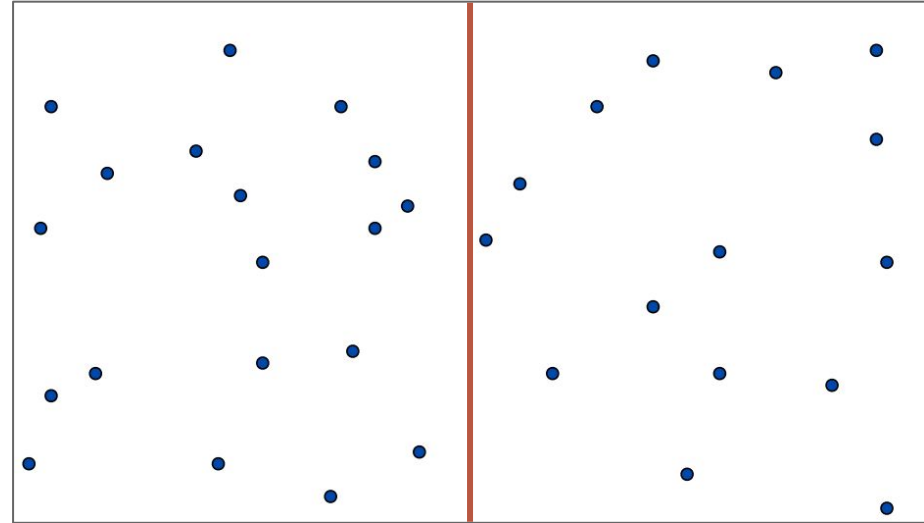
A First Attempt

Divide: How should we try to divide?

We could try splitting it in half.

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

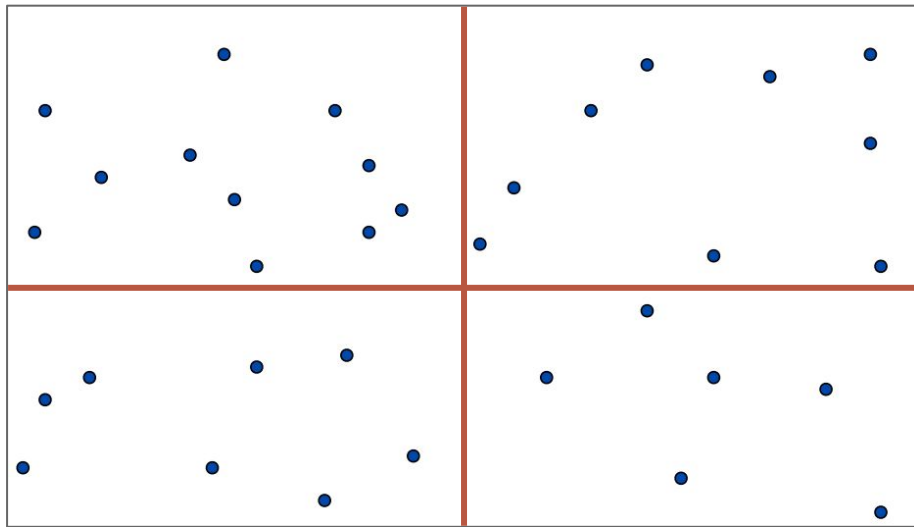
Divide: How should we try to divide?

We could try splitting it in half.

Or, we could try splitting it along the x-axis AND the y-axis.

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

Divide: How should we try to divide?

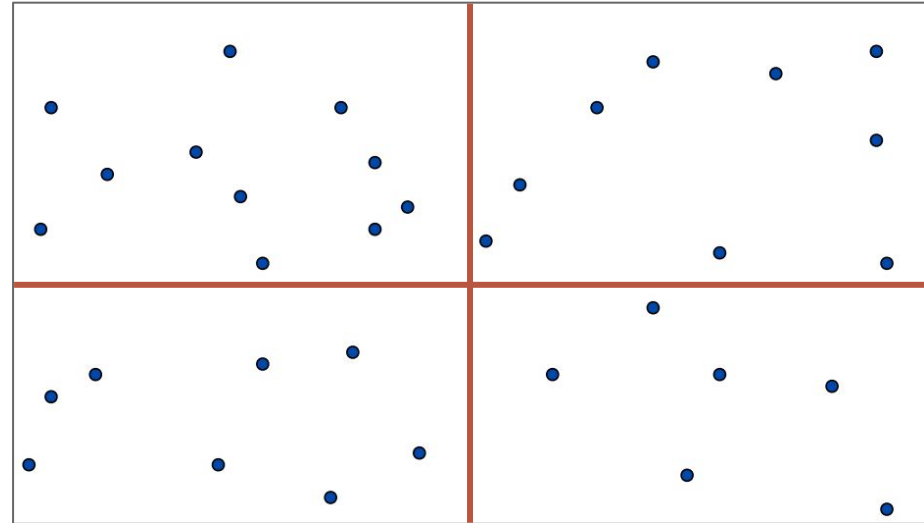
We could try splitting it in half.

Or, we could try splitting it along the x-axis AND the y-axis.

Let's consider splitting both ways. What could go wrong here?

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

Divide: How should we try to divide?

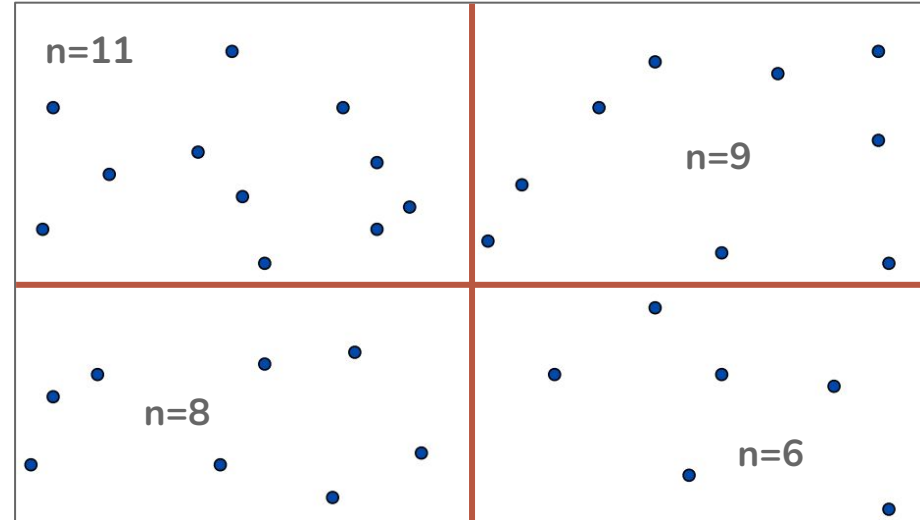
We could try splitting it in half.

Or, we could try splitting it along the x-axis AND the y-axis.

Let's consider splitting both ways. What could go wrong here?

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

Divide: How should we try to divide?

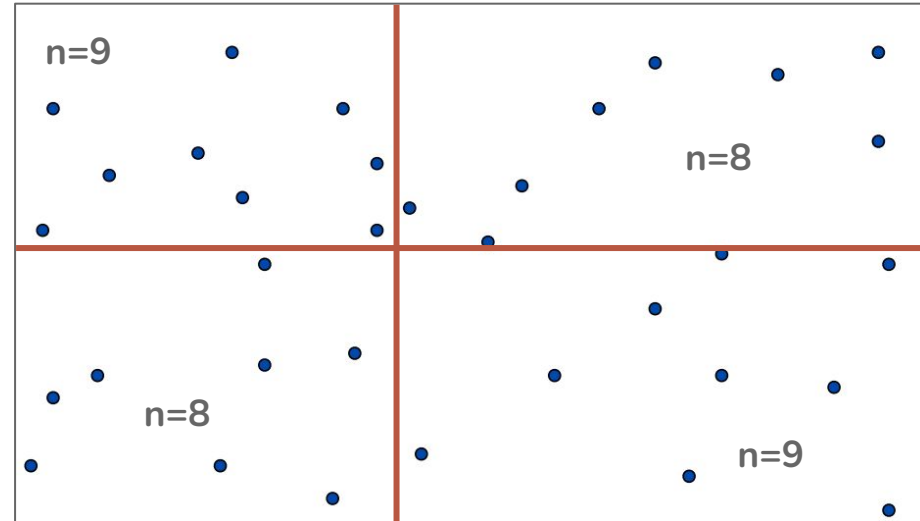
We could try splitting it in half.

Or, we could try splitting it along the x-axis AND the y-axis.

Let's consider splitting both ways. What could go wrong here?

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

Divide: How should we try to divide?

We could try splitting it in half.

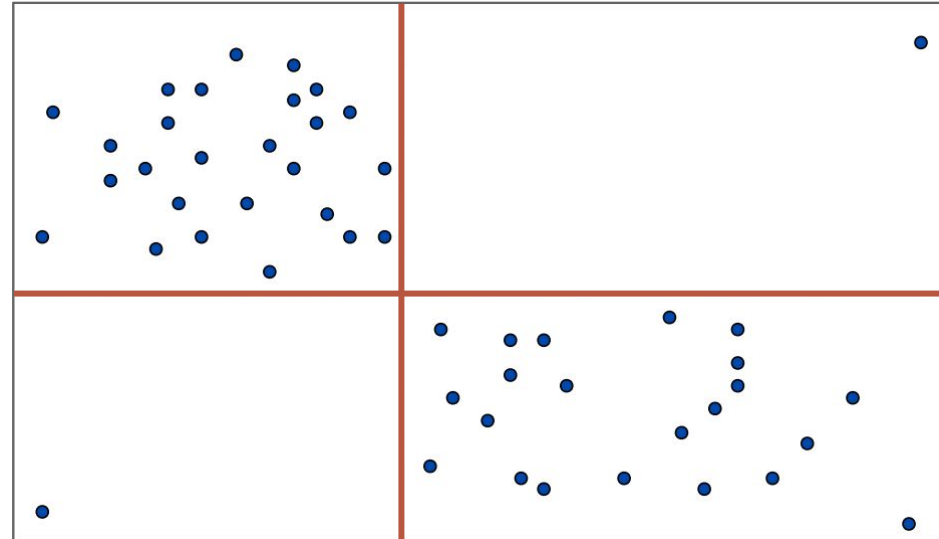
Or, we could try splitting it along the x-axis AND the y-axis.

Let's consider splitting both ways. What could go wrong here?

Subproblems may not all have same size!

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.

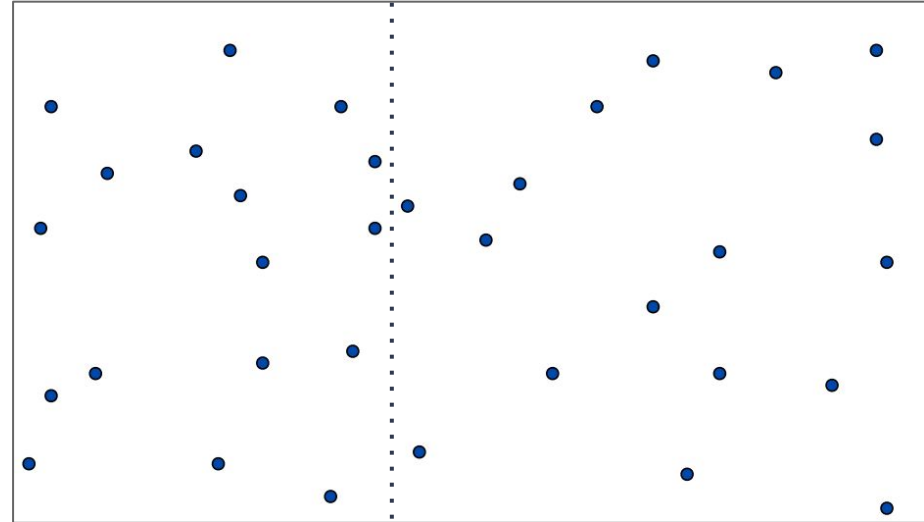


A First Attempt

Divide: Splitting down one axis.

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



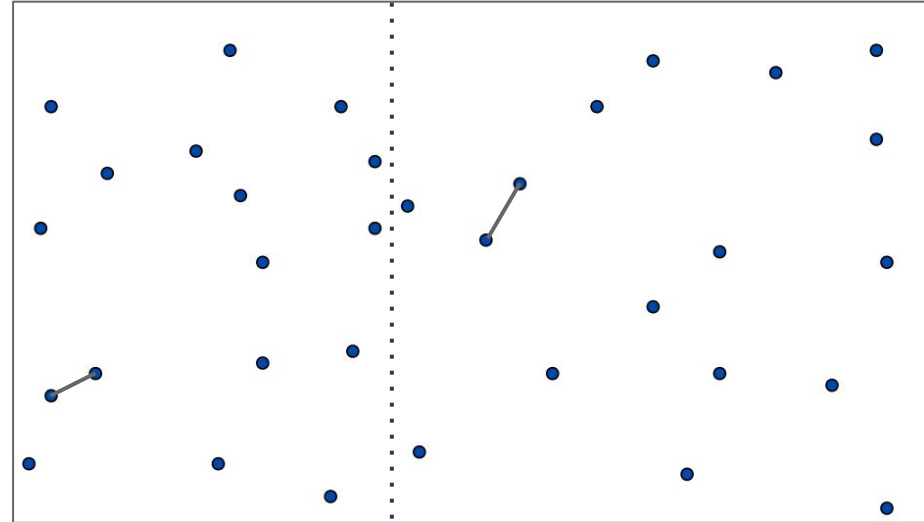
A First Attempt

Divide: Splitting down one axis.

Conquer: Recursively find the closest pair in each half.

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

Divide: Splitting down one axis.

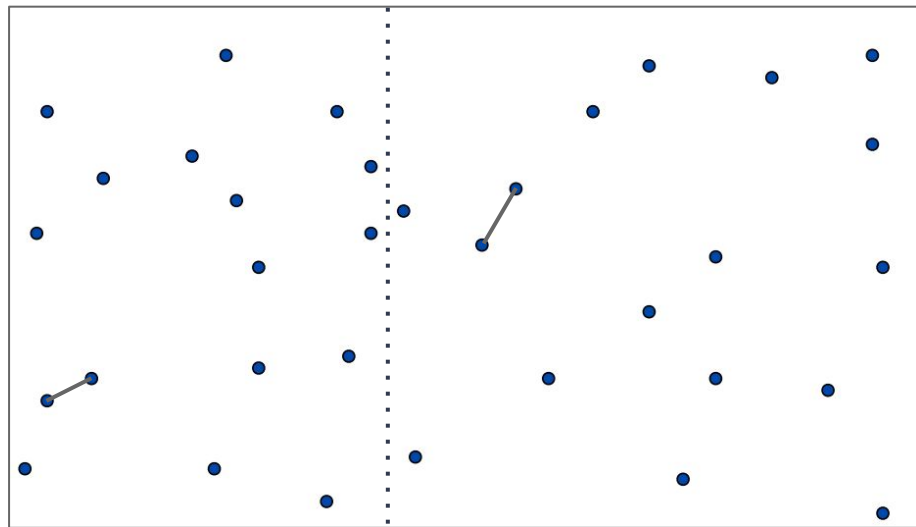
Conquer: Recursively find the closest pair in each half.

Combine: Compare between

- Closest pair on left
- Closest pair on right

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

Divide: Splitting down one axis.

Conquer: Recursively find the closest pair in each half.

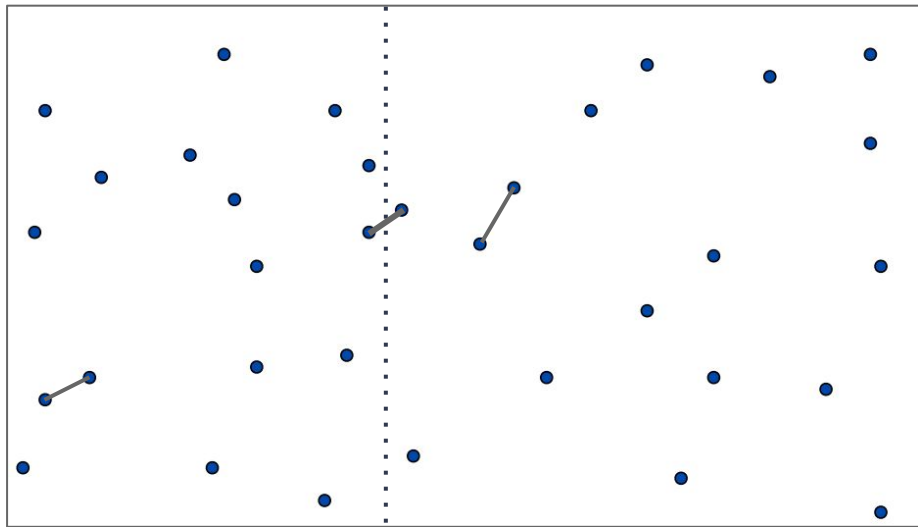
Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Done!

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

Divide: Splitting down one axis.

Conquer: Recursively find the closest pair in each half.

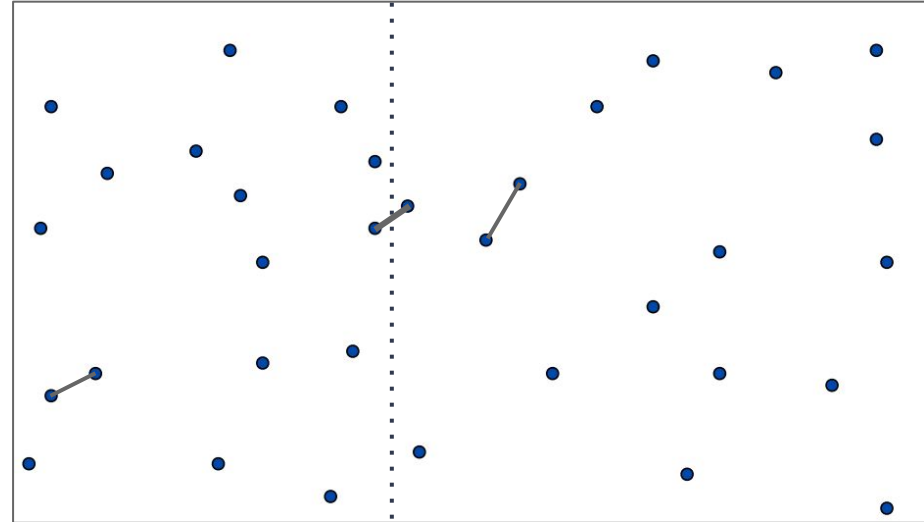
Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

~~Done!~~ Wait...

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A First Attempt

Divide: Splitting down one axis.

Conquer: Recursively find the closest pair in each half.

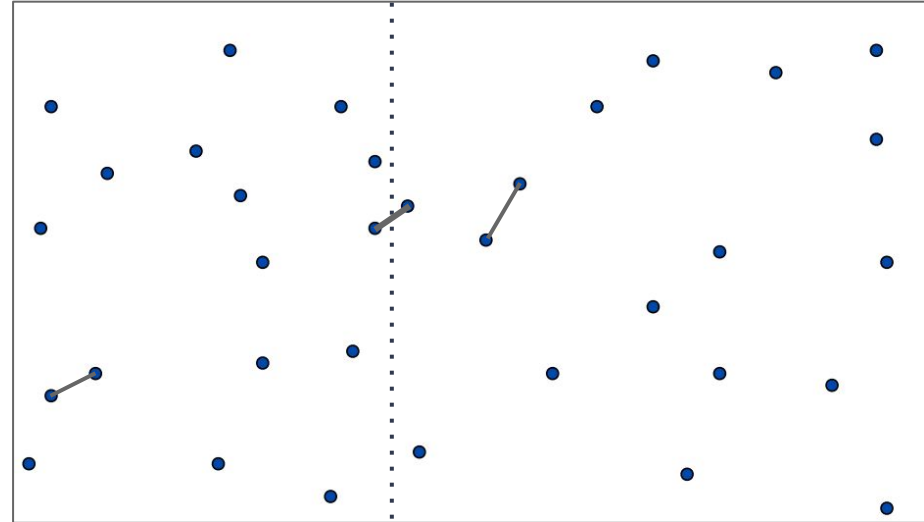
Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

There are $(n/2)(n/2)$ possible pairs between two sides, which is $O(n^2)$.

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



A Detour



When in doubt...

If you don't know how to overcome a roadblock, try to solve a simpler version of the problem!

Here, 2D is difficult. What if the problem was only 1D?

The Same Attempt, but 1D

Divide: Splitting down one axis.

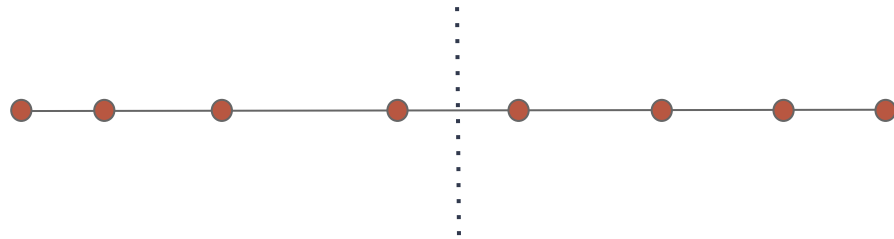
Conquer: Recursively find the closest pair in each half.

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



The Same Attempt, but 1D

Divide: Splitting down one axis.

Conquer: Recursively find the closest pair in each half.

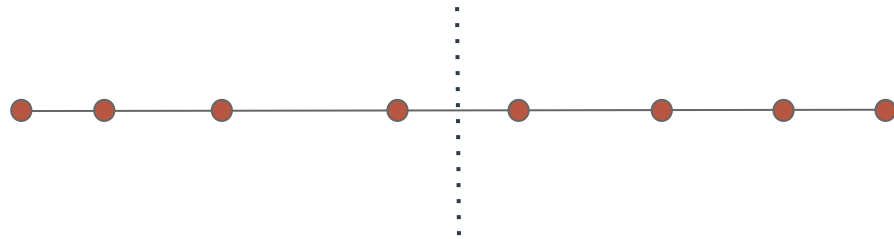
Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Why do we not have the problem here?

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



The Same Attempt, but 1D

Divide: Splitting down one axis.

Conquer: Recursively find the closest pair in each half.

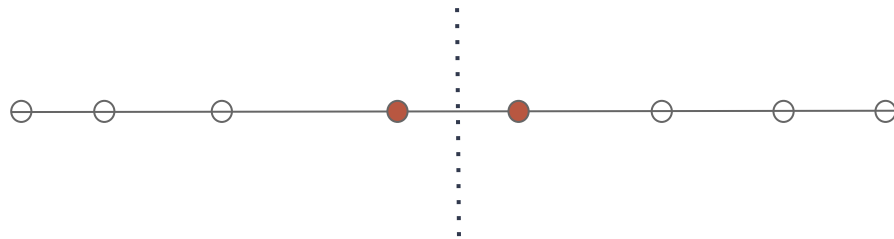
Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

There's no reason to check points too far from the center!

In general, the divide and conquer paradigm involves algorithms that look as follows:

- **Base Case:** Solve the trivial input.
- **Divide:** Split the input into smaller chunks.
- **Conquer:** Recursively solve.
- **Combine:** Do something efficient to create the final answer.



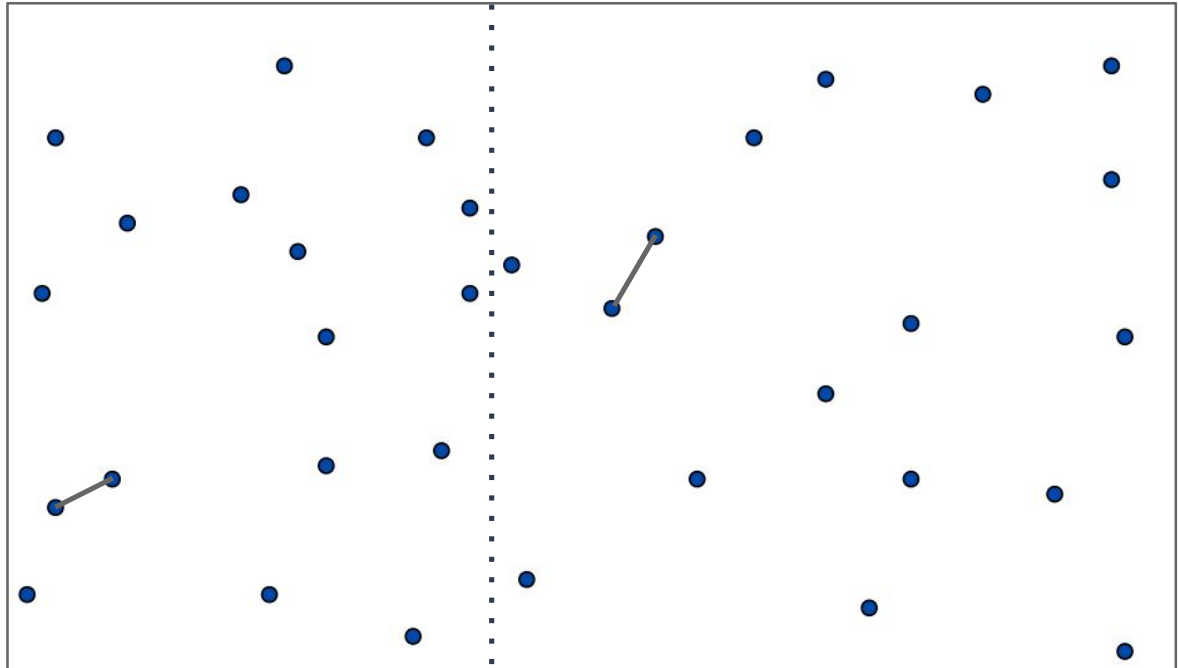
Back to the original problem!



A Second Attempt

Q: How do we find the closest pair between the two sides?

From 1D: We shouldn't look too far from the line!



A Second Attempt

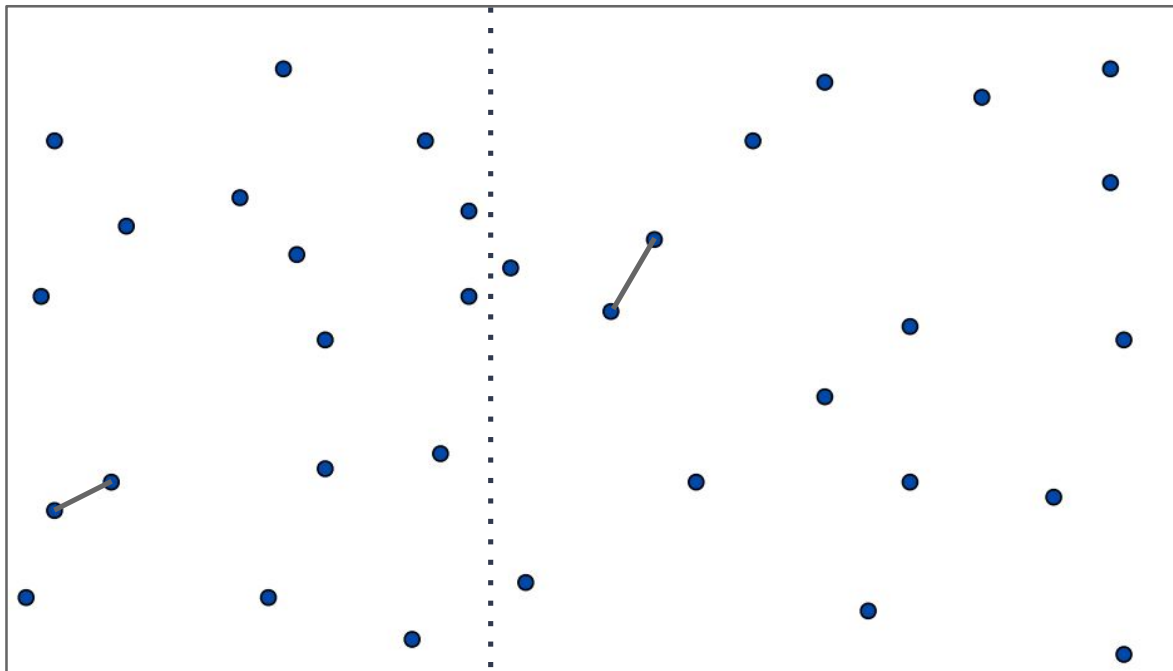
Q: How do we find the closest pair between the two sides?

From 1D: We shouldn't look too far from the line!

Can we just take the closest points to the line?

No, they might be far apart in height!

But we still shouldn't look too far from the line...



A Second Attempt

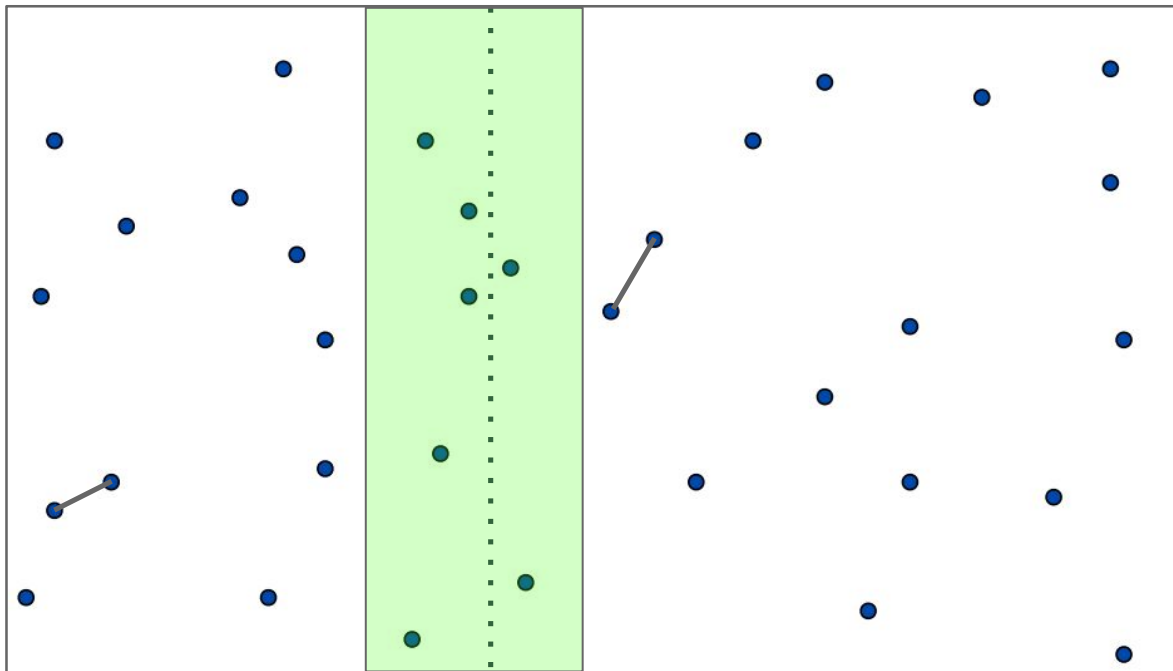
Q: How do we find the closest pair between the two sides?

From 1D: We shouldn't look too far from the line!

Can we just take the closest points to the line?

No, they might be far apart in height!

But we still shouldn't look too far from the line...

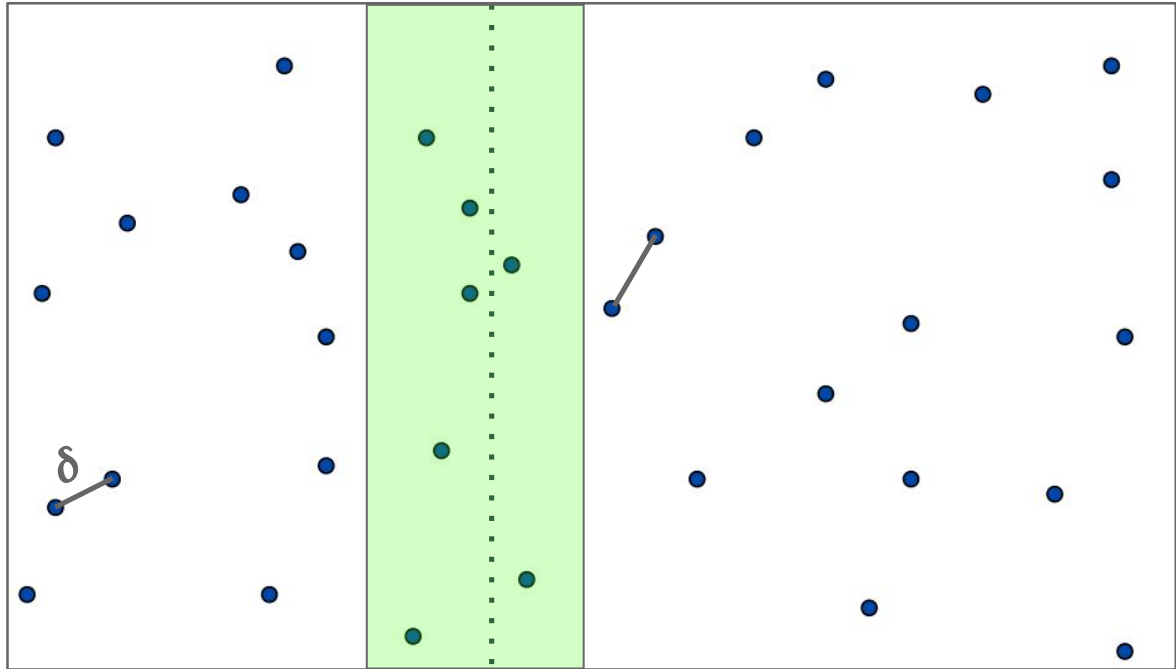


A Second Attempt

Q: How do we find the closest pair between the two sides?

From 1D: We shouldn't look too far from the line!

How wide should this band of points be?



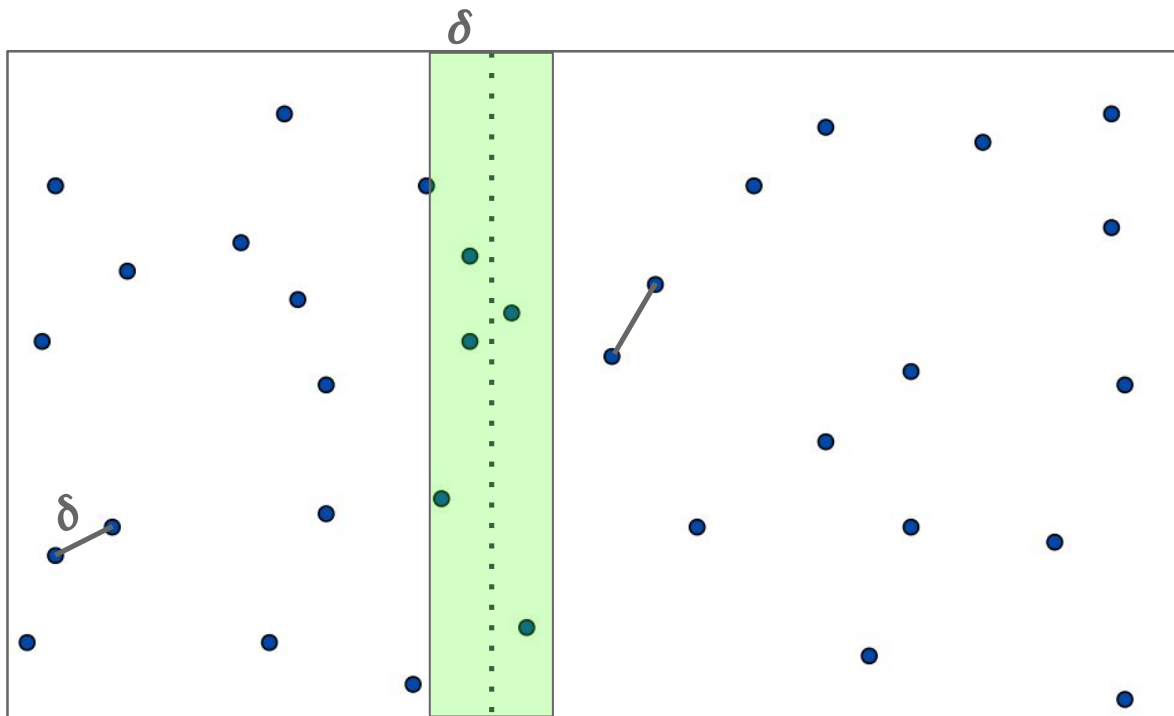
A Second Attempt

Q: How do we find the closest pair between the two sides?

From 1D: We shouldn't look too far from the line!

How wide should this band of points be?

As wide as the closest pair we know so far!



A Second Attempt

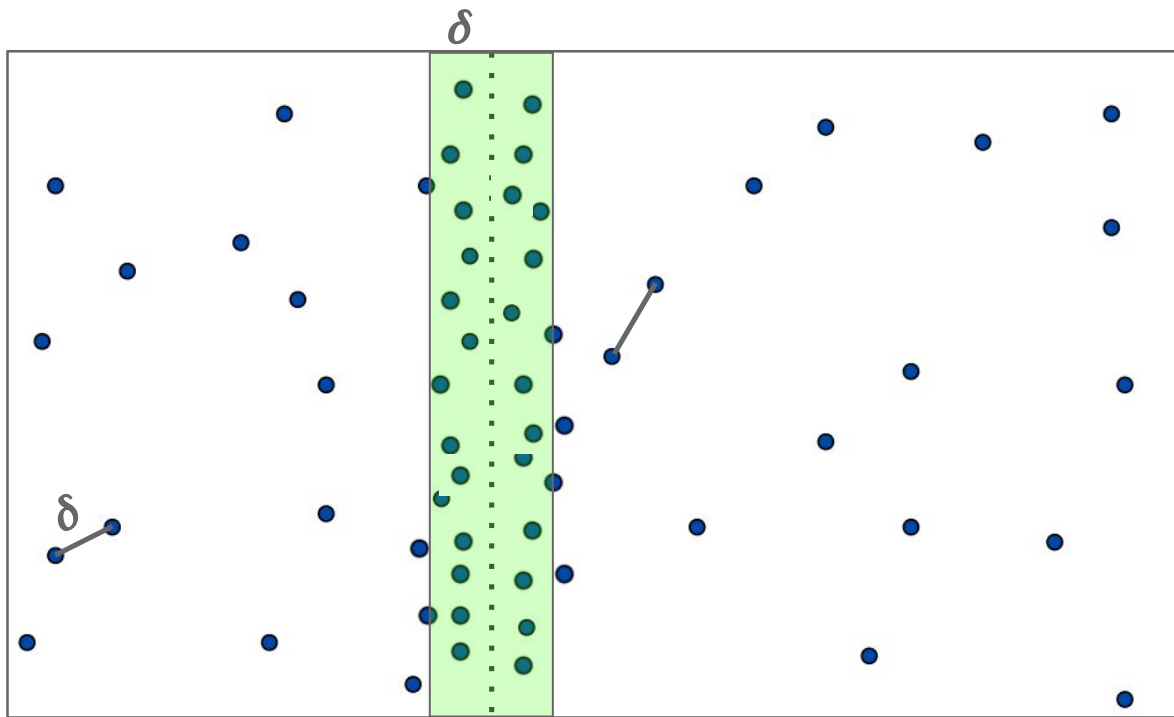
Q: How do we find the closest pair between the two sides?

From 1D: We shouldn't look too far from the line!

How wide should this band of points be?

As wide as the closest pair we know so far!

But what if all the points are in that bar...?



A Second Attempt

Q: How do we find the closest pair between the two sides?

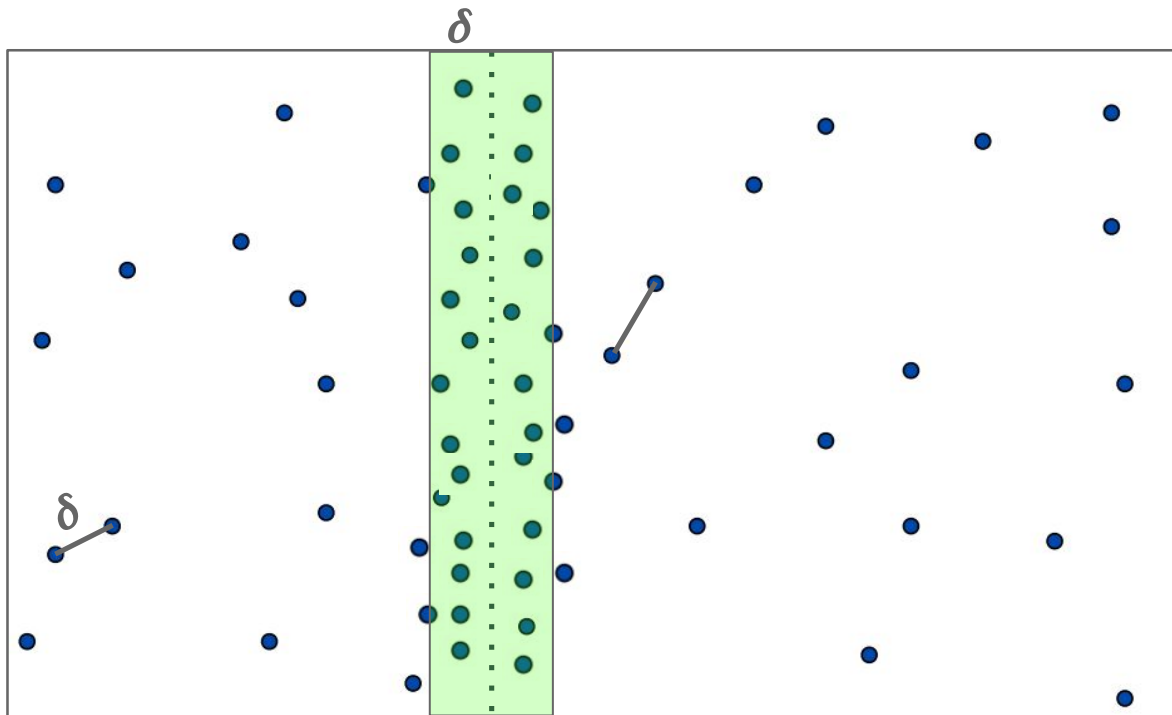
From 1D: We shouldn't look too far from the line!

How wide should this band of points be?

As wide as the closest pair we know so far!

But what if all the points are in that bar...?

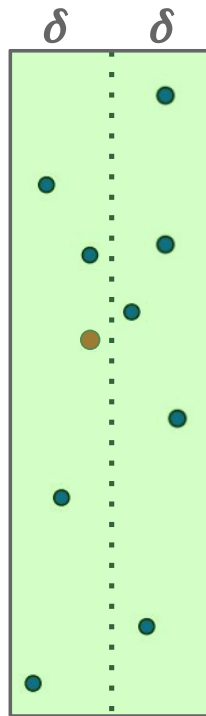
Then δ would be smaller!



A Second Attempt

Q: How do we find the closest pair between the two sides?

However, there could still be a lot of points in this band. How do we make sure we don't spend too long comparing here?

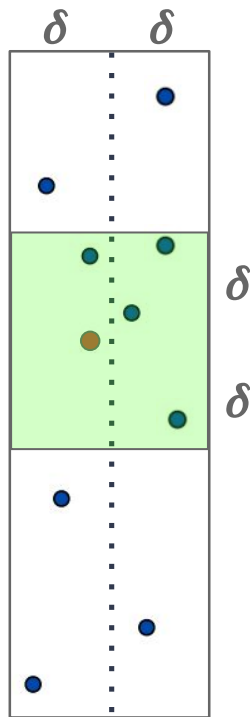


A Second Attempt

Q: How do we find the closest pair between the two sides?

However, there could still be a lot of points in this band. How do we make sure we don't spend too long comparing here?

Only check two points if their y-values are close together!



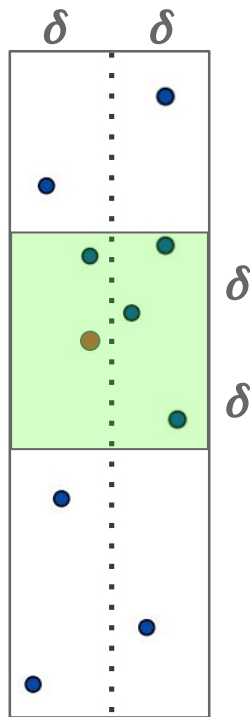
A Second Attempt

Q: How do we find the closest pair between the two sides?

However, there could still be a lot of points in this band. How do we make sure we don't spend too long comparing here?

Only check two points if their y-values are close together!

But how do we tell which values are within δ of a point?



A Second Attempt

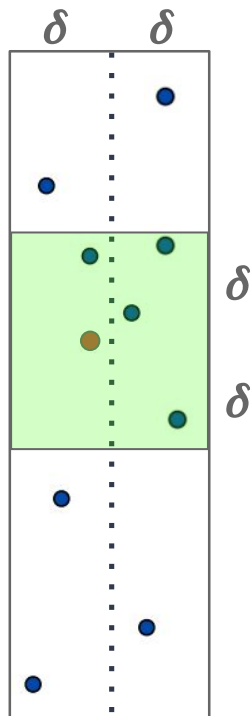
Q: How do we find the closest pair between the two sides?

From 1D: We should only need to check a constant number of points.

We can't know how many points are within δ of our point, but there can't be too many.

Let's just sort them by y-value and pick the closest ones!

How many points in the band can have a y-value within δ of our red point?



A Second Attempt

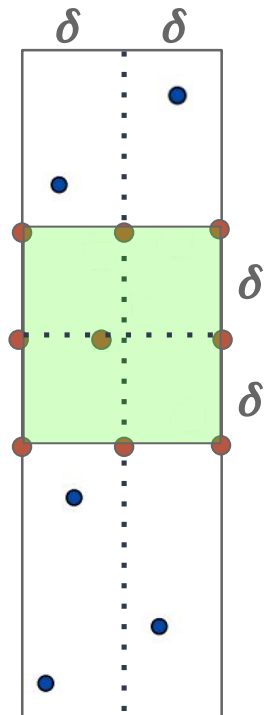
Q: How do we find the closest pair between the two sides?

From 1D: We should only need to check a constant number of points.

We can't know how many points are within δ of our point, but there can't be too many.

Let's just sort them by y-value and pick the closest ones!

How many points in the band can have a y-value within δ of our red point?



For any point, there can't be more than 8 points that have a y-value within δ of our point.

A Second Attempt

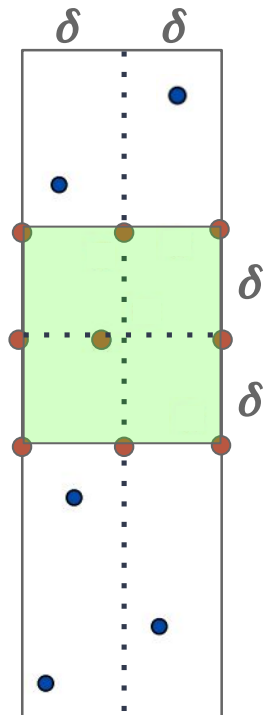
Q: How do we find the closest pair between the two sides?

From 1D: We should only need to check a constant number of points.

We can't know how many points are within δ of our point, but there can't be too many.

Let's just sort them by y-value and pick the closest ones!

How many points in the band can have a y-value within δ of our red point?



For any point, there can't be more than 8 points that have a y-value within δ of our point.

Finding closest pair within a band:

- Sort the points by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

Complete Algorithm

Base Case: Two points, return them.

Divide: Splitting down x-axis.

Conquer: Recursively find the closest pair in each half.

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

Runtime

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(n \log n)$

Conquer: Recursively find the closest pair in each half.

$2T(n/2)$

Combine: Compare between

$O(1)$

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

$O(n \log n)$

$O(n)$

$O(1)$

Total

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

$$T(n) = n \log^2(n)$$

Can we do better?

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(n \log n)$

Conquer: Recursively find the closest pair in each half.

$2T(n/2)$

Combine: Compare between

$O(1)$

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

$O(n \log n)$

$O(n)$

$O(1)$

Can we do better?

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(n \log n)$

Conquer: Recursively find the closest pair in each half.

$2T(n/2)$

Combine: Compare between

$O(1)$

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

$O(n \log n)$

$O(n)$

$O(1)$

Can we do better?

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(n \log n)$

Conquer: Recursively find the closest pair in each half.

$2T(n/2)$

Combine: Compare between

$O(1)$

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

$O(n \log n)$

$O(n)$

$O(1)$

Re-sorting is wasteful. How can we do better?

Can we do better?

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(n \log n)$

Conquer: Recursively find the closest pair in each half.

$2T(n/2)$

Combine: Compare between

$O(1)$

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

$O(n \log n)$

$O(n)$

$O(1)$

Re-sorting is wasteful. How can we do better?

x-axis: sort once at the very start, pass in sorted lists each time!

Can we do better?

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(1)$

Conquer: Recursively find the closest pair in each half.

$2T(n/2)$

Combine: Compare between

$O(1)$

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

$O(n \log n)$

$O(n)$

$O(1)$

Re-sorting is wasteful. How can we do better?

x-axis: sort once at the very start, pass in sorted lists each time!

Can we do better?

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(1)$

Conquer: Recursively find the closest pair in each half.

$2T(n/2)$

Combine: Compare between

$O(1)$

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

$O(n \log n)$

$O(n)$

$O(1)$

Re-sorting is wasteful. How can we do better?

y-axis: ???

Can we do better?

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(1)$

Conquer: Recursively find the closest pair in each half.

$2T(n/2)$

Combine: Compare between

$O(1)$

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return minimum.

$O(n \log n)$

$O(n)$

$O(1)$

Re-sorting is wasteful. How can we do better?

y-axis: Each recursive call needed to sort as well. Combining is easy!

Also Mergesorting at the same time?

Base Case: Two points, return them.

Divide: Splitting down x-axis.

Conquer: Recursively find the closest pair in each half. Return the closest pair.

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Return minimum.

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return best seen.

Also Mergesorting at the same time?

Base Case: Two points, return them.

Divide: Splitting down x-axis.

Conquer: Recursively find the closest pair in each half. Return the closest pair AND the list of points sorted by y-value.

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Return minimum.

Finding closest pair within a band:

- Sort the points in the band by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return best seen.

Also Mergesorting at the same time?

Base Case: Two points, return them.

Divide: Splitting down x-axis.

Conquer: Recursively find the closest pair in each half. Return the closest pair AND the list of points sorted by y-value.

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Return minimum.

Finding closest pair within a band:

- ~~— Sort the points in the band by y-value.~~
- Merge to get a list of points sorted by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return best seen.

Also Mergesorting at the same time?

Base Case: Two points, return them.

Divide: Splitting down x-axis.

Conquer: Recursively find the closest pair in each half. Return the closest pair AND the list of points sorted by y-value.

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

Return minimum and points sorted by y.

Finding closest pair within a band:

- ~~— Sort the points in the band by y-value.~~
- Merge to get a list of points sorted by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return best seen.

Also Mergesorting at the same time?

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(1)$

Conquer: Recursively find the closest pair in each half. Return the closest pair AND the list of points sorted by y-value.

$2T(n/2)$

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

$O(1)$

Return minimum and points sorted by y.

Finding closest pair within a band:

- ~~Sort the points in the band by y-value.~~
- Merge to get a list of points sorted by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return best seen.

$O(n \log n)$

$O(n)$

$O(1)$

Runtime Analysis

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(1)$

Conquer: Recursively find the closest pair in each half. Return the closest pair AND the list of points sorted by y-value.

$2T(n/2)$

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

$O(1)$

Return minimum and points sorted by y.

Finding closest pair within a band:

- ~~Sort the points in the band by y-value.~~
- Merge to get a list of points sorted by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return best seen.

$O(n)$

$O(n)$

$O(1)$

Total

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

Runtime Analysis

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(1)$

Conquer: Recursively find the closest pair in each half. Return the closest pair AND the list of points sorted by y-value.

$2T(n/2)$

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

$O(1)$

Return minimum and points sorted by y.

Finding closest pair within a band:

- ~~Sort the points in the band by y-value.~~
- Merge to get a list of points sorted by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return best seen.

$O(n)$

$O(n)$

$O(1)$

Total

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

Runtime Analysis

At the start – sort by x-value.

Base Case: Two points, return them.

$O(1)$

Divide: Splitting down x-axis.

$O(1)$

Conquer: Recursively find the closest pair in each half. Return the closest pair AND the list of points sorted by y-value.

$2T(n/2)$

Combine: Compare between

- Closest pair on left
- Closest pair on right
- Closest pair between two sides

$O(1)$

Return minimum and points sorted by y.

Finding closest pair within a band:

- ~~Sort the points in the band by y-value.~~
- Merge to get a list of points sorted by y-value.
- For each point, check its distance from the next 8 points on the list.
- Return best seen.

$O(n)$

$O(n)$

$O(1)$

Total

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

Summary

Today we walked through the entire process of designing a divide and conquer algorithm.

The process was not straightforward at all – there were a lot of decisions we made that ended up being wrong. Design is never straightforward, but every decision and every mistake can teach us something!