Divide & Conquer

CSE 421 Fall 2022
Lecture 9

This Week

Divide and Conquer. Using recursion to solve problems efficiently.

Start on Dynamic Programming (a 2+ week adventure in using recursive thinking to solve problems efficiently).

Classic, beautiful algorithms.

Goal: see how far recursive thinking can take you.

Today:

What is D&C?

A classic D&C example

Define our problem for Wednesday

Divide & Conquer

Algorithm Design Paradigm

1. Divide instance into subparts.
2. Solve the parts recursively.
3. Conquer by combining the answers

You’ve seen Divide & Conquer before!

Merge Sort

https://www.youtube.com/watch?v=XaqR3G_NVoo

Divide

0	1	2	3	4	5	6	7	8	9
8	2	91	22	57	1	10	6	7	4

0	1	2	3	4
8	2	91	22	57

5	6	7	8	9
1	10	6	7	4

Sort the pieces through the magic of recursion

Combine

0	1	2	3	4
2	8	22	57	91

5	6	7	8	9
1	4	6	7	10

0	1	2	3	4	5	6	7	8	9
1	2	4	6	7	8	10	22	57	91

Merge Sort

```
mergeSort(input) {
   if (input.length == 1)
      return
   else
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```

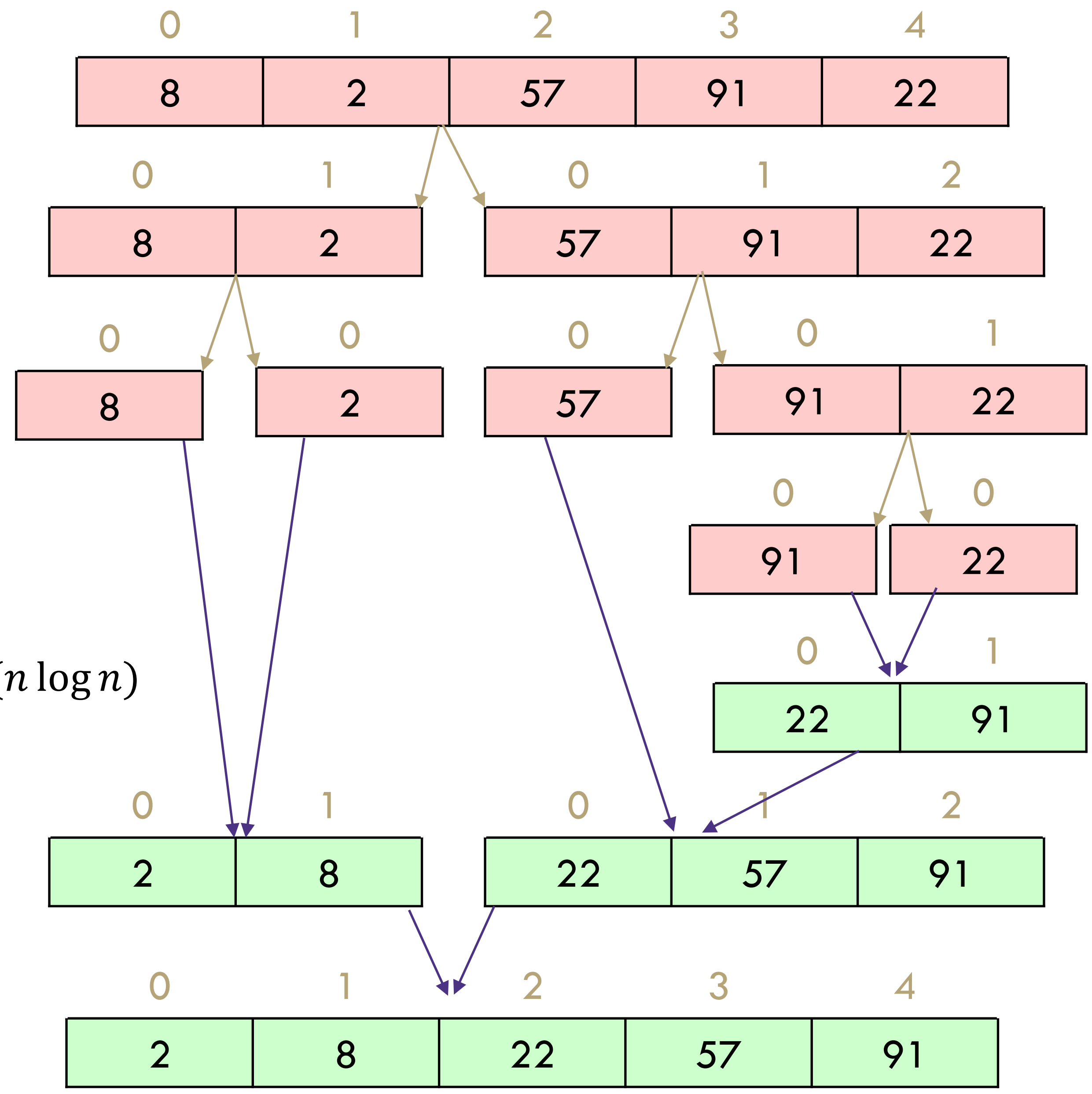
Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases} = O(n \log n)$

Best case runtime? Same

Average runtime? Same

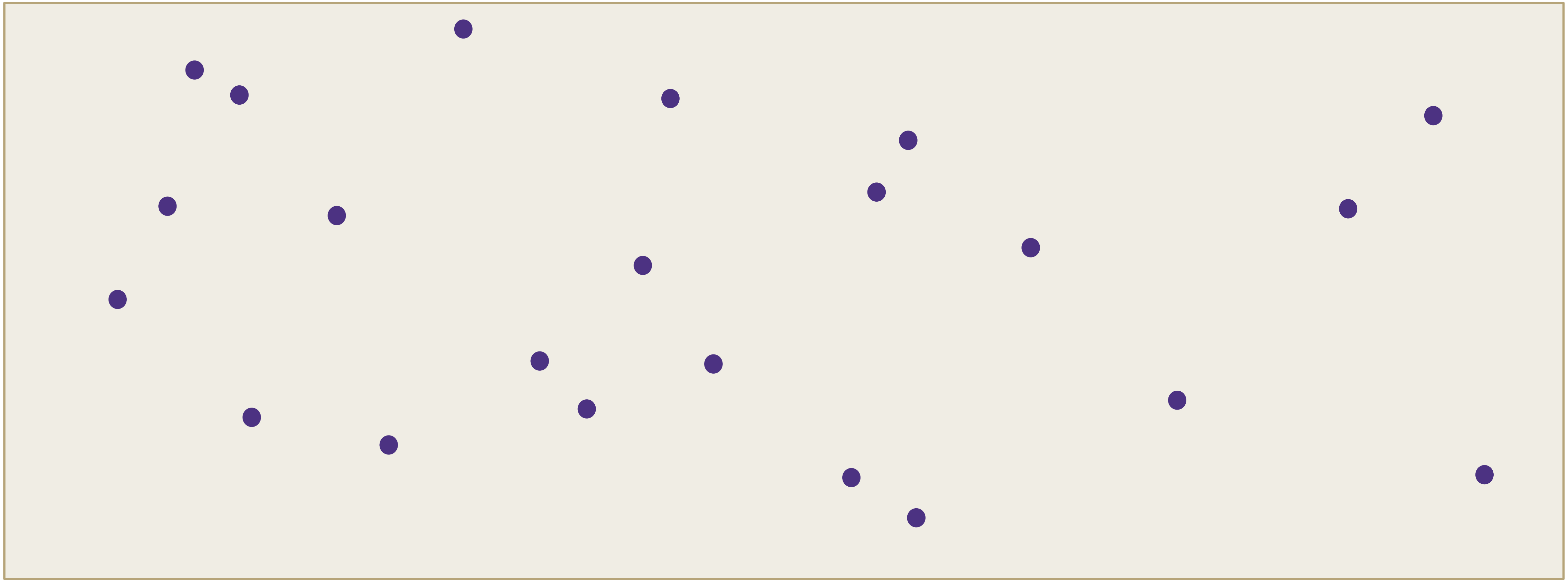
Stable? Yes

In-place? No



Our First Problem

Given: A list of points in 2-dimensions
Return: The distance between the two points that are closest to each other.



Our First Problem – Baseline

Given: A list of points in 2-dimensions
Return: The distance between the two points that are closest to each other.

What's the first algorithm you can think of? What's its running time?

Our First Problem – Baseline

Given: A list of points in 2-dimensions
Return: The distance between the two points that are closest to each other.

What's the first algorithm you can think of? What's its running time?

Just calculate the distance between all pairs.

$\Theta(n^2)$.

Keep in the back of your minds: if we aren't doing better than $\Theta(n^2)$ we haven't made any progress.

An Easier Version

Given: A list of points in 1-dimension
Return: The distance between the two points that are closest to each other.

Your input will be a list of `doubles` (in no particular order).

What's your algorithm?

An Easier Version

Given: A list of points in 1-dimension
Return: The distance between the two points that are closest to each other.

Your input will be a list of `doubles` (in no particular order).

What's your algorithm?

Sort the list! Find the minimum `dist(A[i], A[i+1])`

Correctness:

Running time:

An Easier Version

Given: A list of points in 1-dimension
Return: The distance between the two points that are closest to each other.

Your input will be a list of `doubles` (in no particular order).

What's your algorithm?

Sort the list! Find the minimum `dist(A[i], A[i+1])`

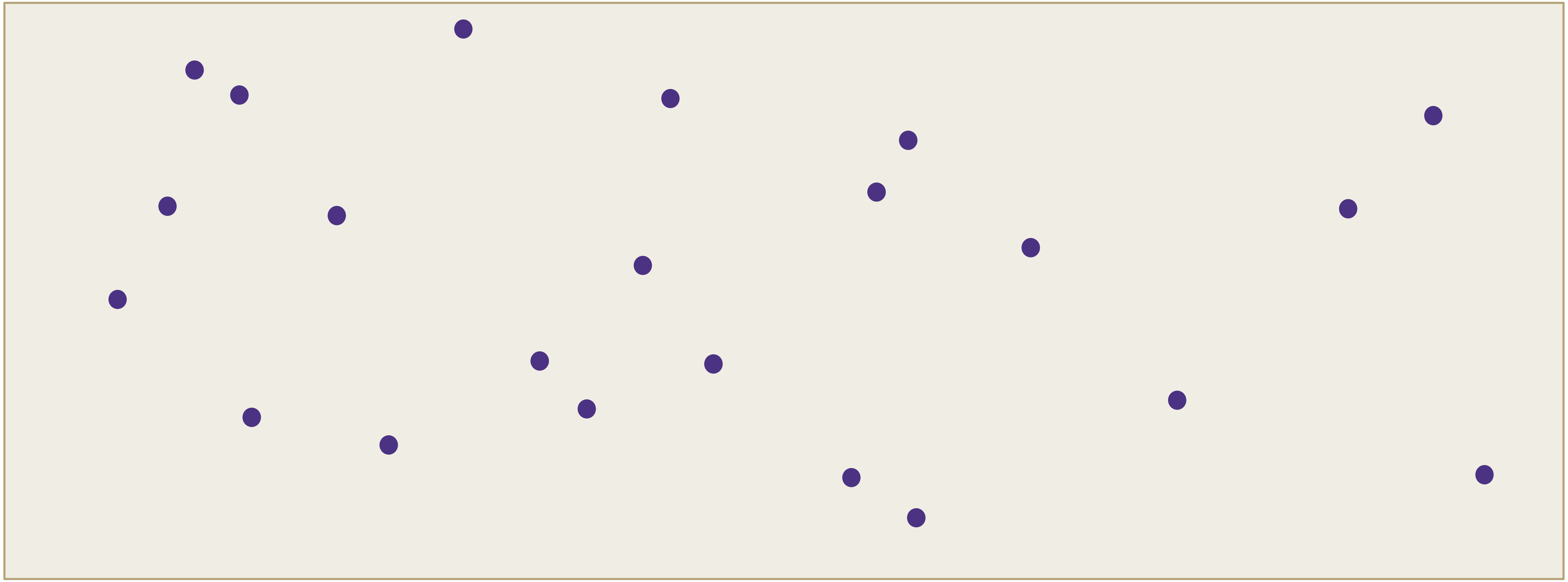
Correctness: After sorting, closest elements must be consecutive.

Running time: $O(n \log n) + O(n) = O(n \log n)$.

2D Closest Points

Back to the main problem.
Baseline is $\Theta(n^2)$
1D version is $\Theta(n \log n)$

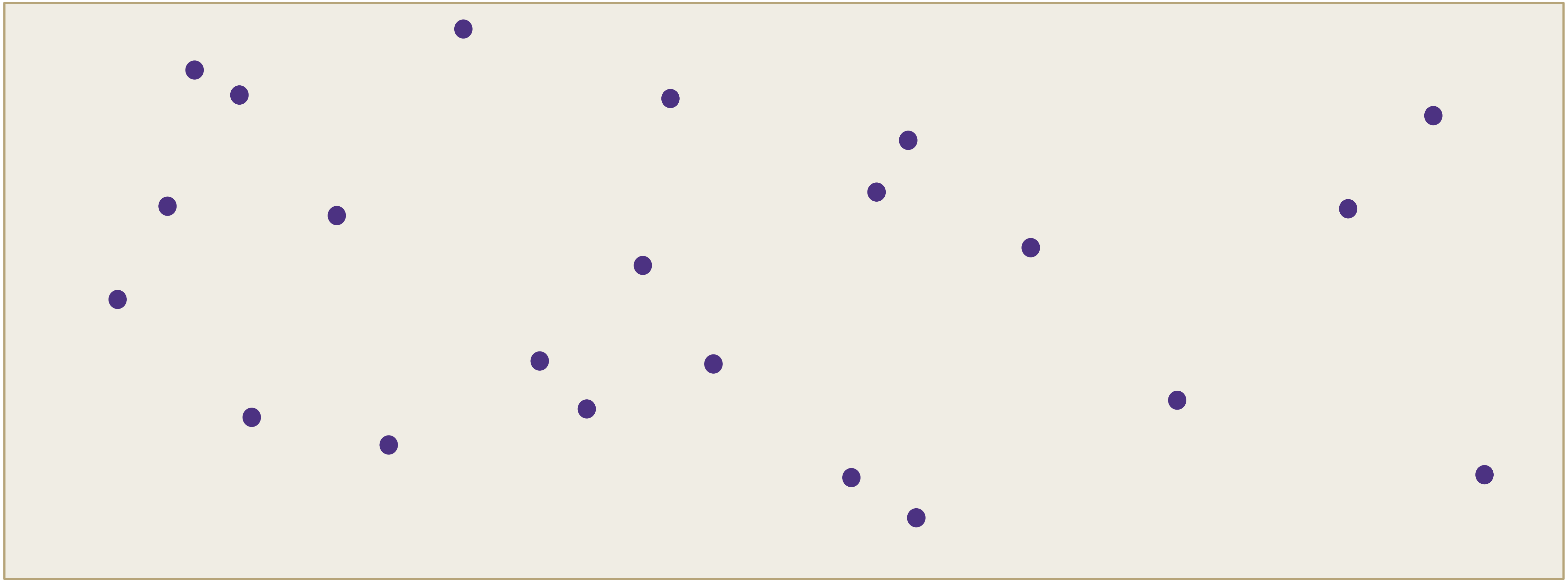
Given: A list of points in 2-dimensions
Return: The distance between the two points that are closest to each other.



2D Closest Points

Back to the main problem.
Baseline is $\Theta(n^2)$
1D version is $\Theta(n \log n)$

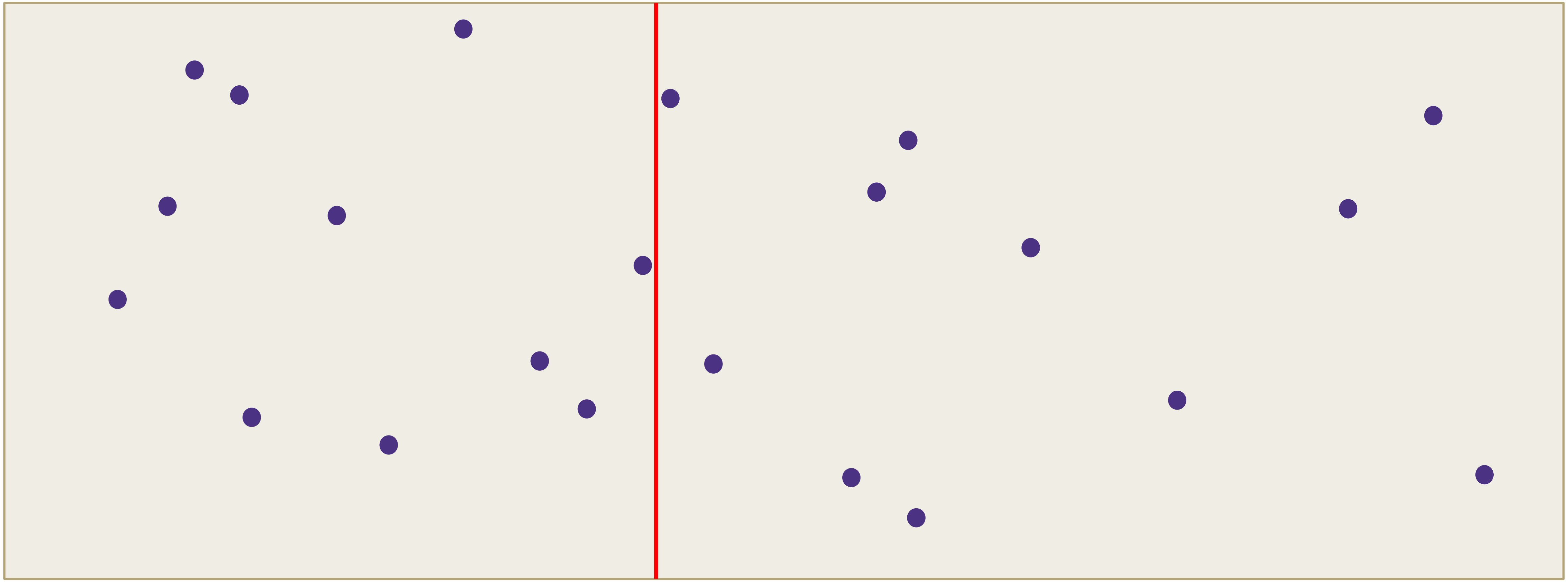
Need to Divide and Conquer. How do we divide?



2D Closest Points

Back to the main problem.
Baseline is $\Theta(n^2)$
1D version is $\Theta(n \log n)$

Need to Divide and Conquer. How do we divide?
Median in one dimension seems like a good spot to split!

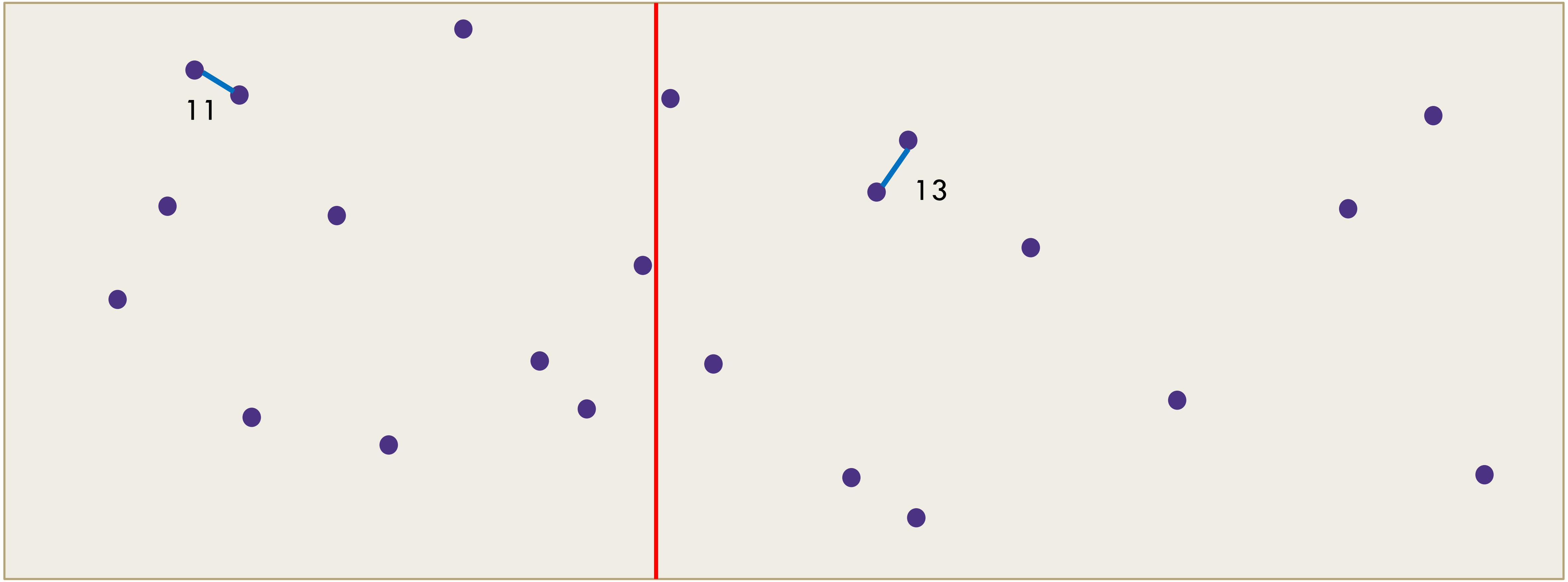


Pseudocode

```
double 2DClosestPoints(P[1..n])
        if(n ≤ 100)  //pick a cutoff you like; doesn't matter for big-O
                check all possible pairs, return the smallest distance.
        Sort P[] by x-coordinate
        δ ←min{2DClosestPoints(P[1..n/2]), 2DClosestPoints(P[n/2+1,n])}
        //TODO: conquer
```

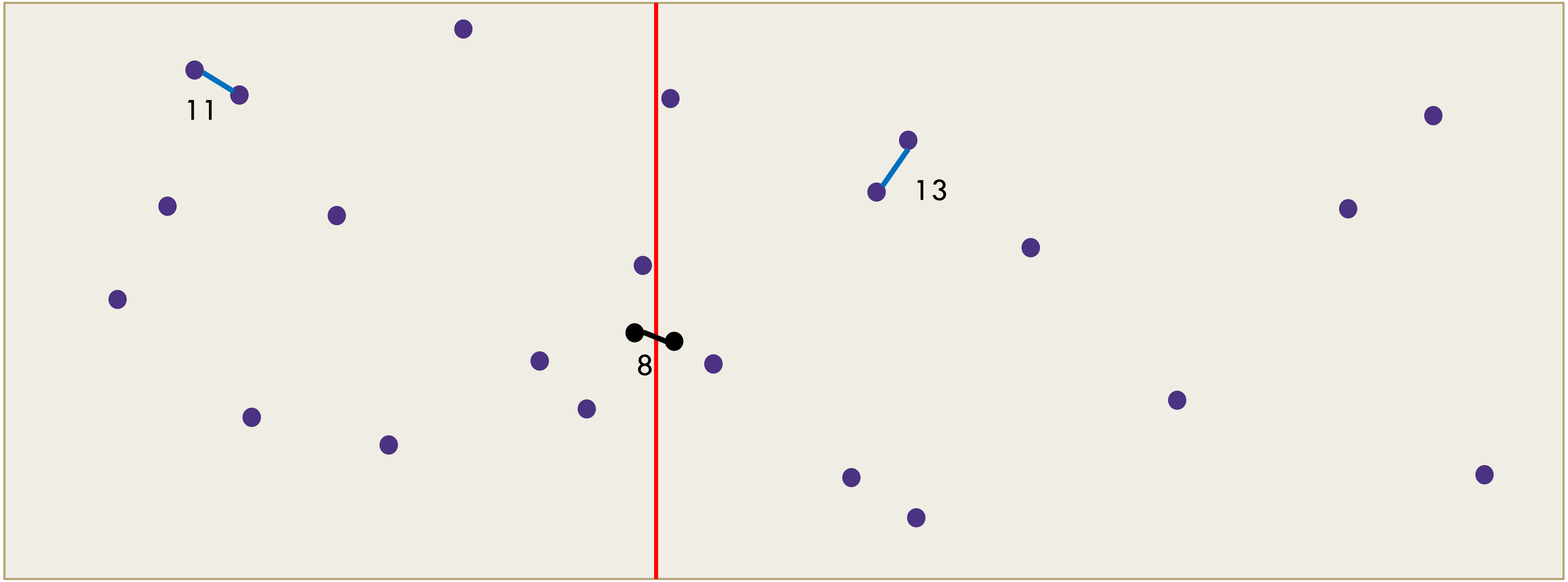
2D Closest Points

Are we done? Just return δ?



2D Closest Points

Are we done? Just return δ?
No! What if the closest pair goes from the left to the right?



Conquer

Can we just check every pair that goes from the left to the right?

Some Questionable Pseudocode

```
double 2DClosestPoints(P[1..n])
    if(n ≤ 100)  //pick a cutoff you like; doesn't matter for big-O
        check all possible pairs, return the smallest distance.
    Sort P[] by x-coordinate
    δ ←min{2DClosestPoints(P[1..n/2]), 2DClosestPoints(P[n/2+1,n])}
    for(i from 1 to n/2)
        for(j from n/2 +1 to n)
            δ ← min{δ, dist(P[i],P[j])}
    return δ
```

Conquer

Can we just check every pair that goes from the left to the right?

Well…it'd, give us the right answer, but…

The key to divide & conquer is making sure the conquer step is a faster calculation than brute force.

Otherwise it's brute force all the way down!

Some Questionable Pseudocode

The Questionable Code on the last slide is equivalent to

$\delta \leftarrow \infty$

for(i from 1 to n)

        for(j from $i+1$ to n)

                $\delta \leftarrow$min{ `dist(P[i],P[j])` }

Literally. They both check every possible i, j. The questionable code just does the checks in a different order!

We need to use the results of our recursive calls to narrow down the problem! How does δ help?

Deep Breath

Where were we?

We found the median, made two recursive calls.

We need to see if there is a pair with one point on the left, the other on the right, with a closer distance than the recursive calls gave.

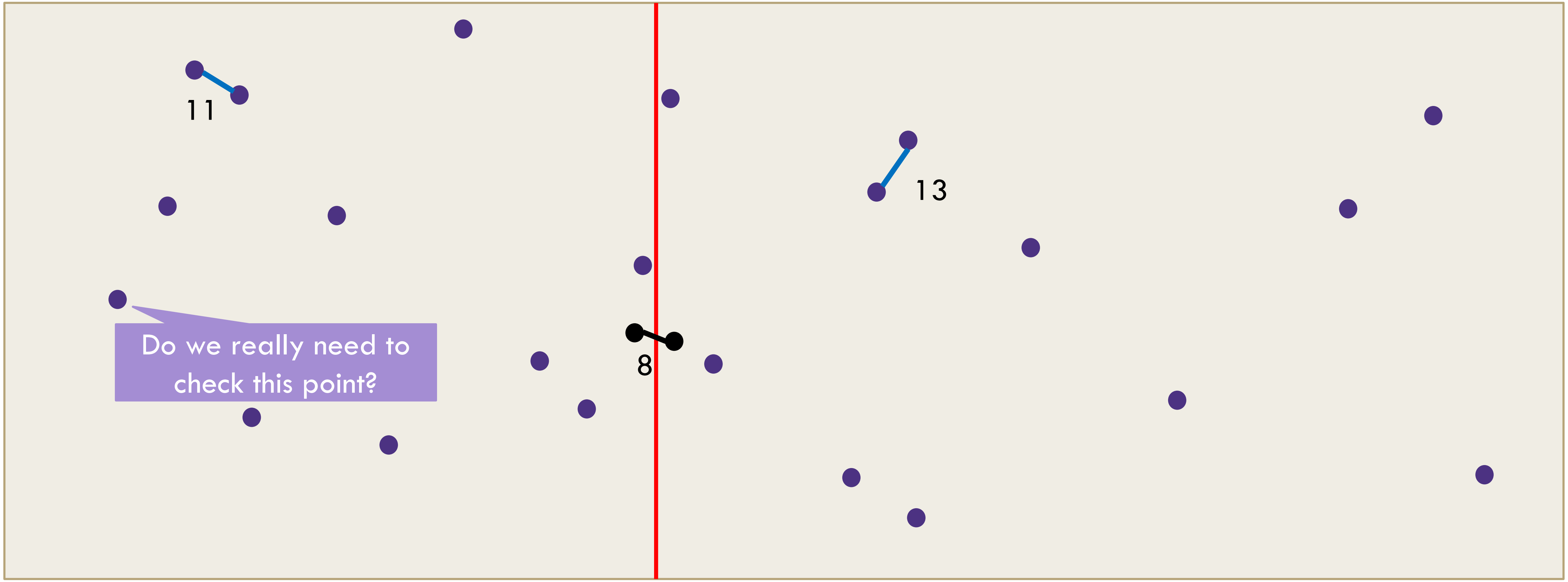
And we need to do it without looking at all possible pairs.

Pseudocode

```
double 2DClosestPoints(P[1..n])
    if(n ≤ 100)  //pick a cutoff you like; doesn't matter for big-O
        check all possible pairs, return the smallest distance.
    Sort P[] by x-coordinate
    δ ←min{2DClosestPoints(P[1..n/2]), 2DClosestPoints(P[n/2+1,n])}
    //TODO: conquer
```

2D Closest Points

Conquer: Find the closest "crossing pair."



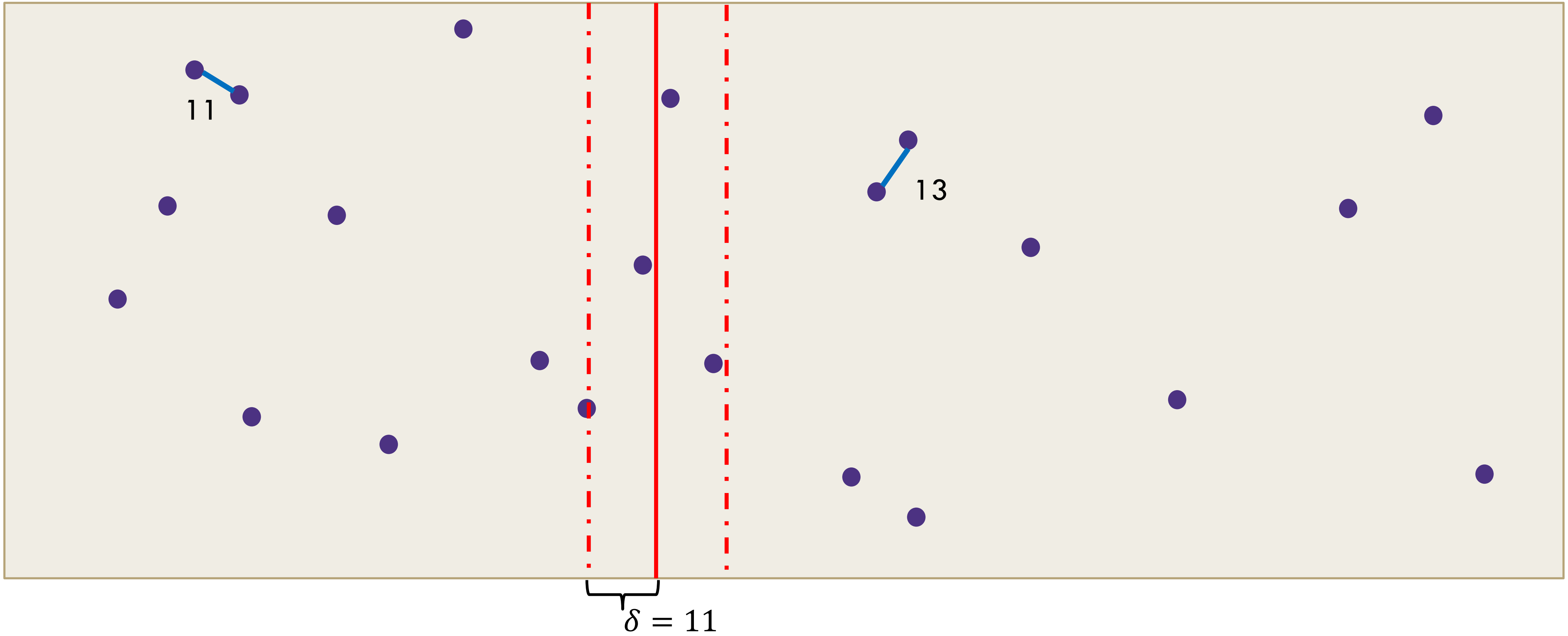
Pseudocode

```
double 2DClosestPoints(P[1..n])
        if(n ≤ 100)  //pick a cutoff you like; doesn't matter for big-O
                check all possible pairs, return the smallest distance.
        Sort P[] by x-coordinate
        δ ←min{2DClosestPoints(P[1..n/2]), 2DClosestPoints(P[n/2+1,n])}
        //TODO: conquer
```

Idea: We only care about a particular point if it is distance δ or less from the dividing line. Otherwise it can't possibly be in a "crossing" pair.

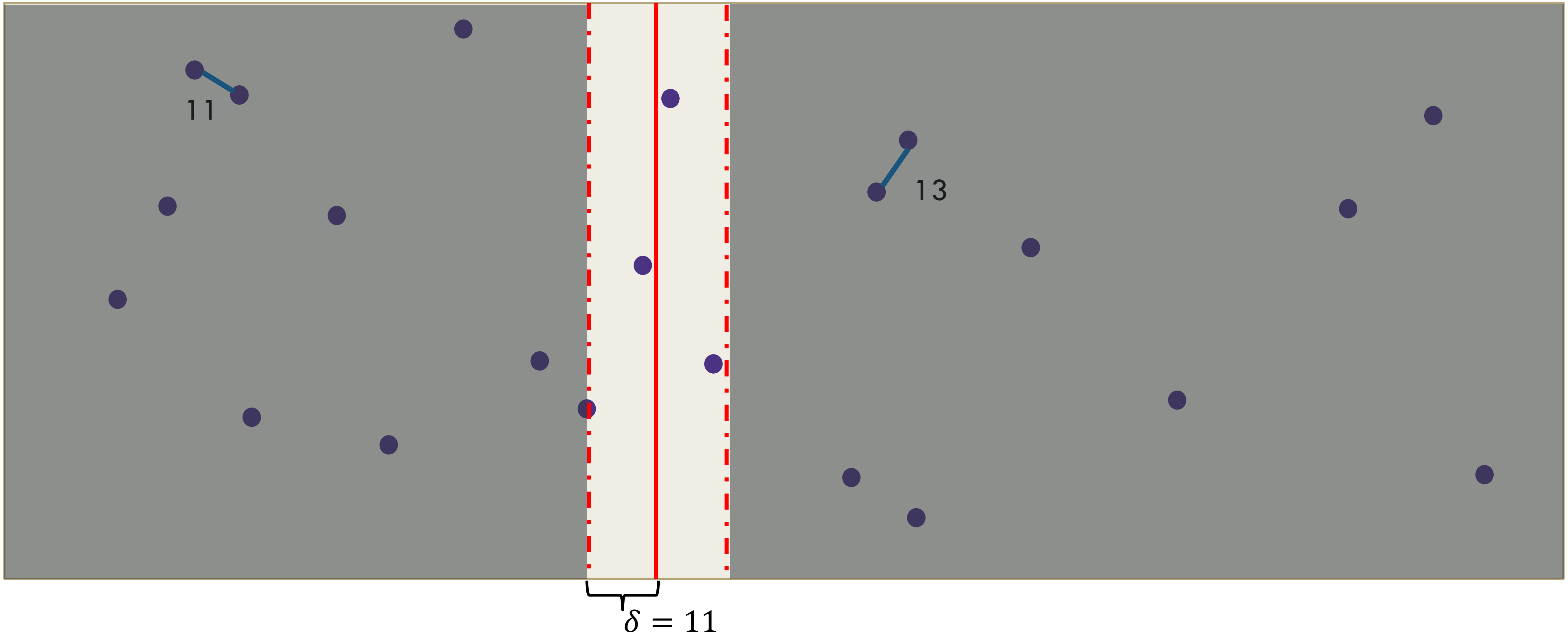
2D Closest Points

Do we need to check every pair?
Only care if distance is less than δ. (Won't be right answer otherwise)



2D Closest Points

Can ignore the shaded regions. Only candidates are in the narrow strip.



Conquering

In that example, we made the problem smaller because there were fewer points…

…That doesn’t always happen.

But we’ve still made our problem easier!

How? Are problem is “almost” one-dimensional.

Almost One-Dimensional

If the problem were **really** one-dimensional, we could just sort and check adjacent elements.

I.e. if P[i], P[j] were the closest points, then $|i - j| = 1$.

We can get pretty close.

If `dist(P[i],P[j])` $< \delta$ and `P[i],P[j]` are both in the middle strip, then $|i - j| \leq 11$

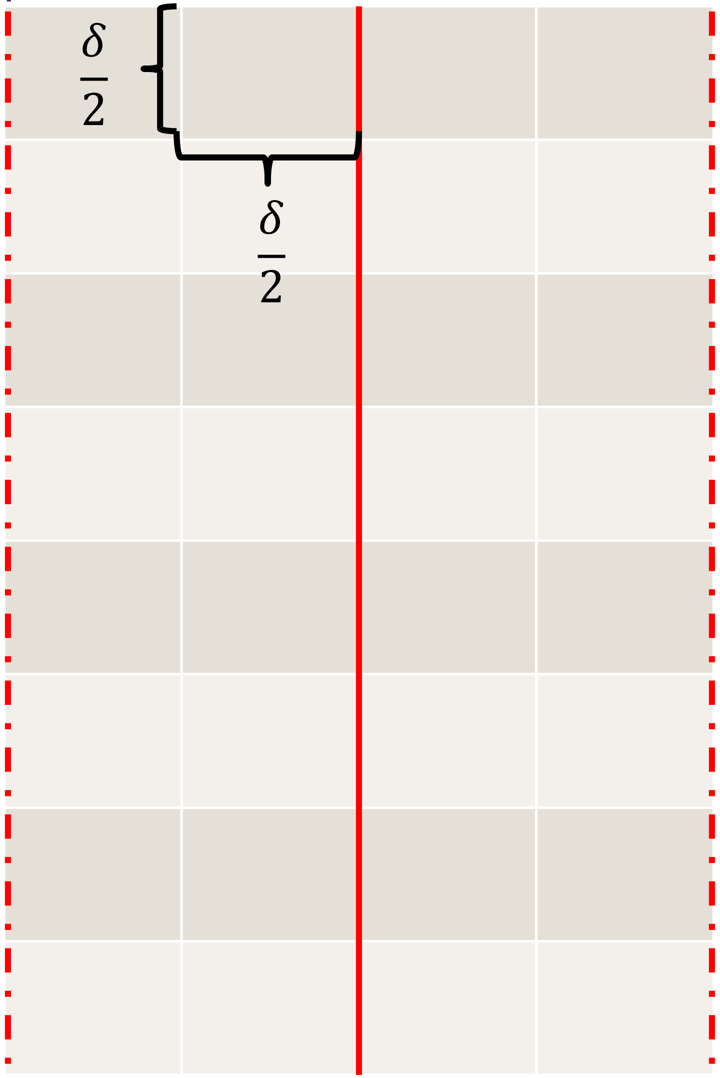
Intuition: two points on the left must be at least δ from each other. Only so much room in a 2δ wide-strip to fit points in δ height too.

Pseudocode

```
double 2DClosestPoints(P[1..n])
        if(n ≤ 100)  //pick a cutoff you like; doesn't matter for big-O
                check all possible pairs, return the smallest distance.
        Sort P[] by x-coordinate
        δ ←min{2DClosestPoints(P[1..n/2]), 2DClosestPoints(P[n/2+1,n])}
        Let C be all points within δ of the median in x-coordinate.
        Sort C by y-coordinate
        for(i from 1 to C.length)
                for(j from i + 1 to i + 11)
                        δ ←min{dist(C[i], C[j]), δ}
        return δ
```

Prove the Lemma

If `dist(P[i],P[j])` $\leq \delta$ and `P[i]`, `P[j]` are both in the middle strip, then $|i - j| \leq 11$

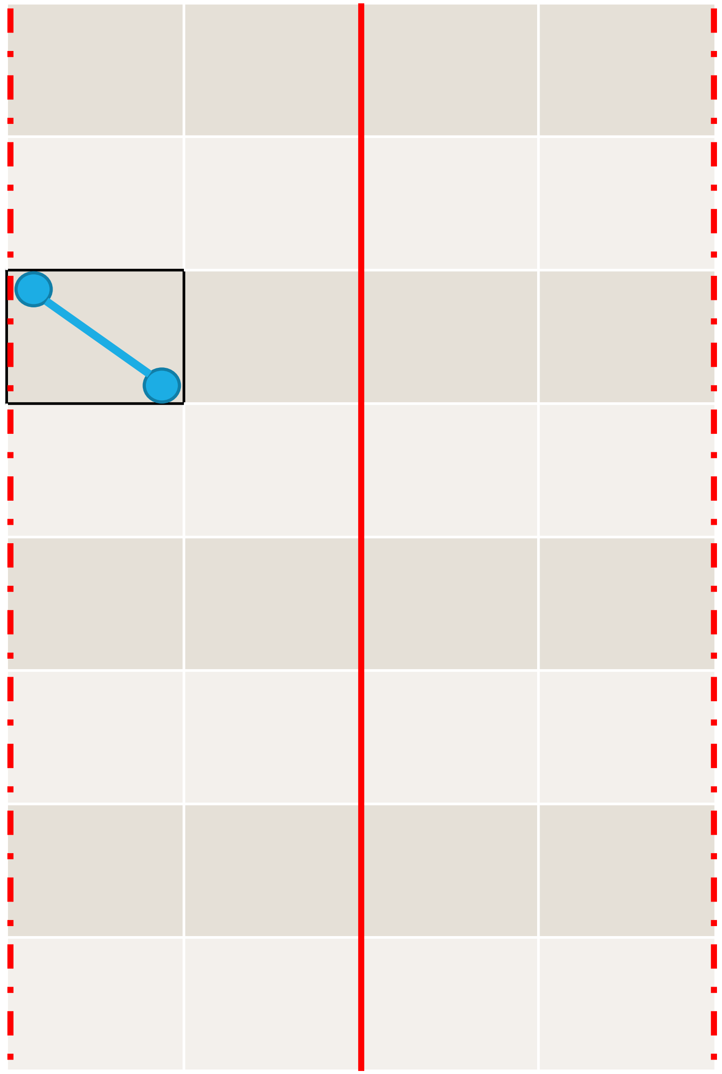


Place a grid of $\delta/2$x$\delta/2$ squares on the strip.

Strip is 4 squares wide.

Prove the Lemma

If `dist(P[i],P[j])` $\leq \delta$ and `P[i]`, `P[j]` are both in the middle strip, then $|i - j| \leq 11$



Place a grid of $\delta/2$x$\delta/2$ squares on the strip.

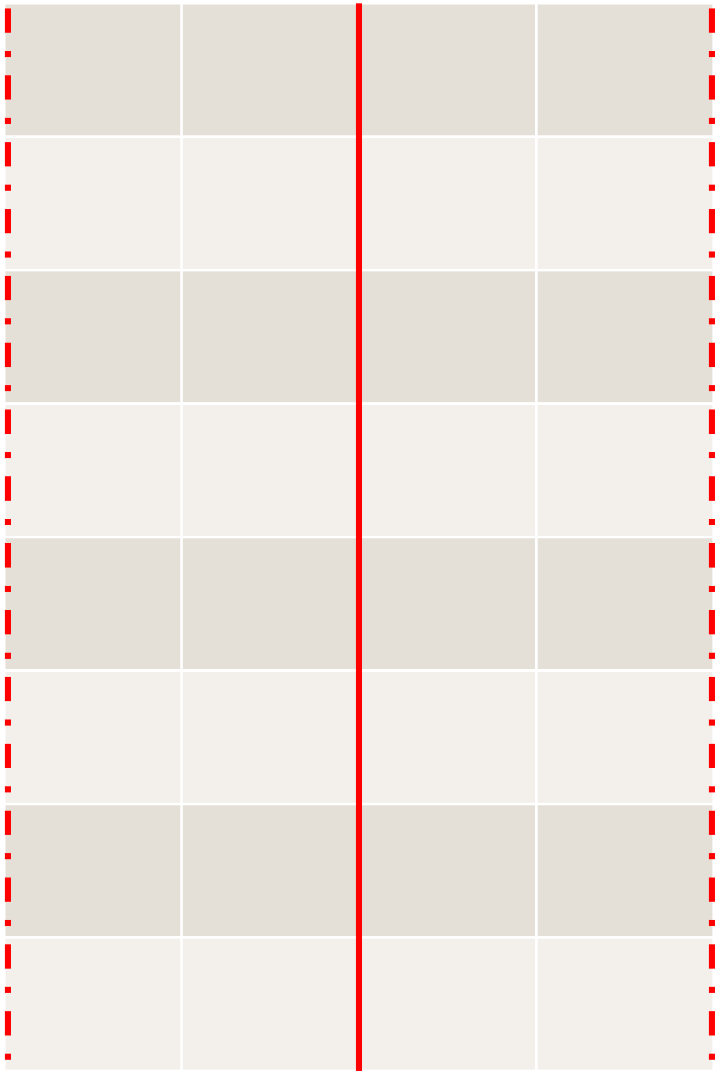
Only one point in each cell.

$$\texttt{dist} \leq \sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \sqrt{2 \cdot \delta^2/4} < \sqrt{\delta^2} = \delta.$$

But δ is the smallest distance between points on the same side (found via recursive calls).

Prove the Lemma

If `dist(P[i],P[j])` $\leq \delta$ and `P[i]`, `P[j]` are both in the middle strip, then $|i - j| \leq 11$



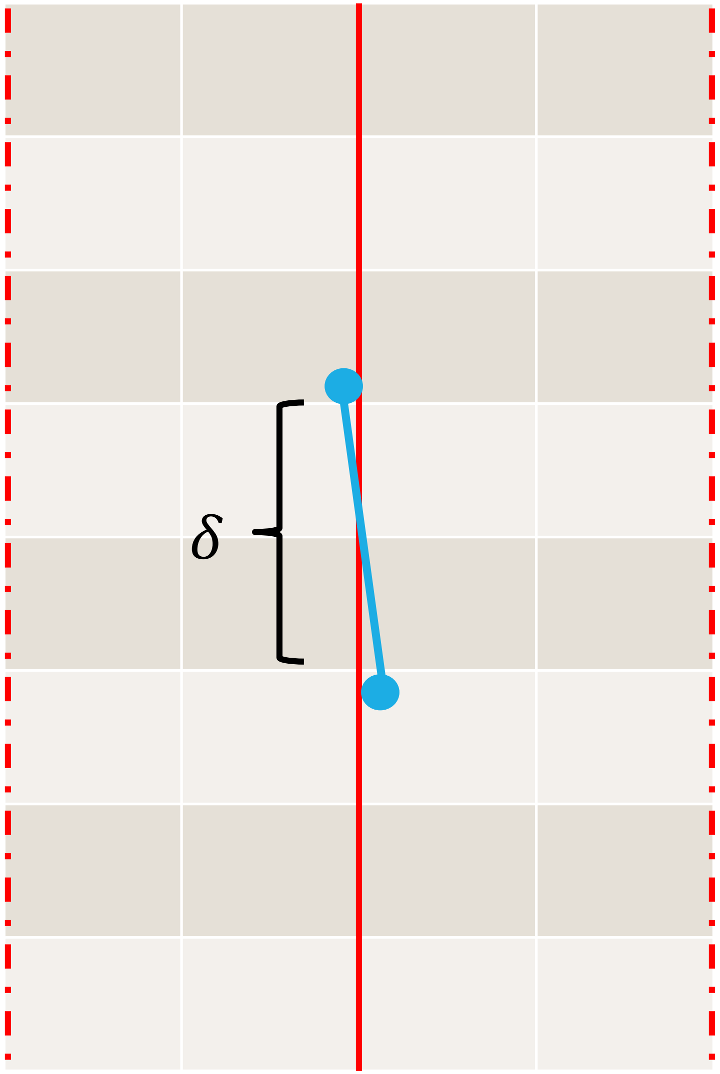
Place a grid of $\delta/2$x$\delta/2$ squares on the strip.

Only one point in each cell.

Two points in the same cell are on the same side and distance $< \delta$.

Prove the Lemma

If `dist(P[i],P[j])` $\leq \delta$ and `P[i]`, `P[j]` are both in the middle strip, then $|i - j| \leq 11$



Place a grid of $\delta/2$x$\delta/2$ squares on the strip.

Only one point in each cell.

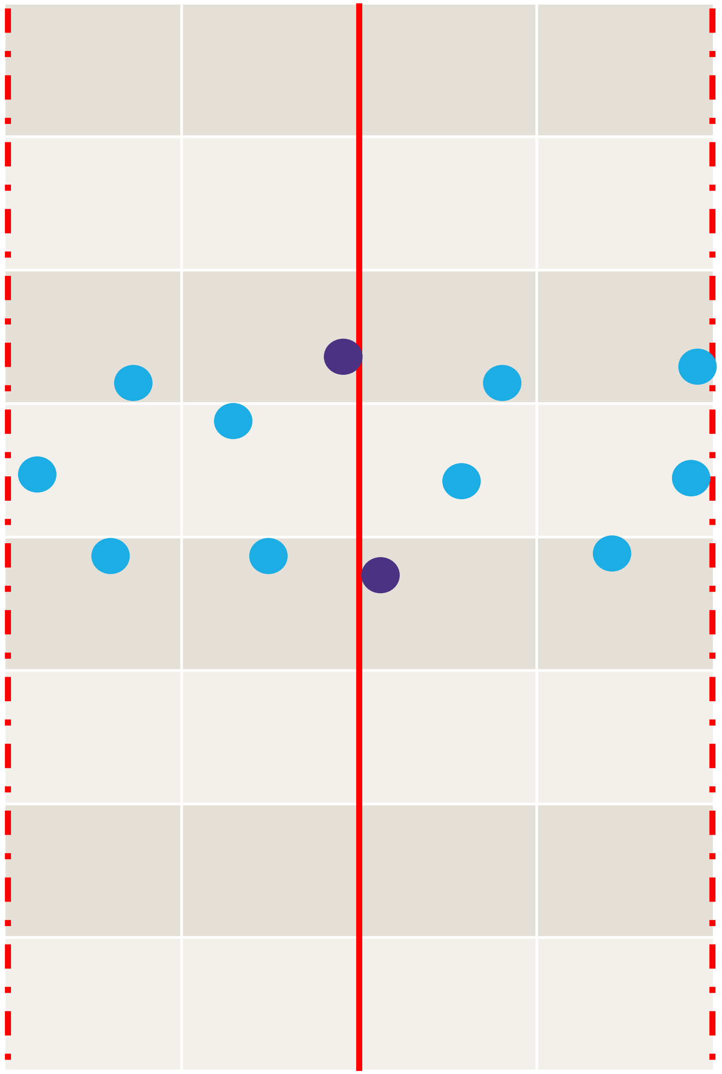
Two points in the same cell are on the same side and distance $< \delta$.

If two points have distance $< \delta$, they are in the same row, adjacent rows, or have one row between them.

Two rows between gives δ distance in the y-coordinate.

Prove the Lemma

If `dist(P[i],P[j])` $\leq \delta$ and `P[i], P[j]` are both in the middle strip, then $|i-j| \leq 11$



Place a grid of $\delta/2$x$\delta/2$ squares on the strip.

Only one point in each cell.

Two points in the same cell are on the same side and distance $< \delta$.

If two points have distance $< \delta$, they are in the same row, adjacent rows, or have one row between them.

How far apart can those points be in the array?

Only 12 elements can be in those rows, so when ordered by y-coordinate, difference is at most 11.

It works!

A formal proof of correctness would be by induction.

Idea for IS:

Closest pair in overall instance is in one half, the other, or split. By IH, recursive calls give distance between closest pairs in each half.

Case 1: closest pair is in a half,

Recursive call gives true shortest distance, by IH. δ will never be overwritten because we only compare to other distances between points.

Case 2: closest pair crosses,

Then its distance is less than the (correct) distances found by the recursive calls. By lemma, we will check the closest pair, and since we take the min, δ will be set to that value and (because we only compare to other distances) never overwritten.

In both cases we return the true shortest distance.

Write a recurrence for the running time

```
double 2DClosestPoints(P[1..n])
    if(n ≤ 100)  //pick a cutoff you like; doesn't matter for big-O
        check all possible pairs, return the smallest distance.
    Sort P[] by x-coordinate
    δ ←min{2DClosestPoints(P[1..n/2]), 2DClosestPoints(P[n/2+1,n])}
    Let C be all points within δ of the median in x-coordinate.
    Sort C by y-coordinate
    for(i from 1 to C.length)
        for(j from i + 1 to i + 10)
            δ ←min{dist(C[i], C[j]), δ}
    return δ
```

Write a recurrence for the running time

double 2DClosestPoints(P[1..n])

  if($n \leq 100$) //pick a cutoff you like; doesn't matter for big-O

    check all possible pairs, return the smallest distance.

$\Theta(n \log n)$

  Sort P[] by x-coordinate

Recursive calls size $n/2$

  δ ←min{2DClosestPoints(P[1..n/2]), 2DClosestPoints(P[n/2+1,n])}

$\Theta(n \log n)$

  Let C be all points within δ of the median in x-coordinate.

  Sort C by y-coordinate

  for(i from 1 to C.length)

    for(j from $i+1$ to $i+10$)

      δ ←min{`dist`($C[i]$, $C[j]$), δ}

$O(1)$ per iteration

$O(n)$ iterations

  return δ

Running Time

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 100 \\ 2T\left(\frac{n}{2}\right) + O(n \log n) & \text{otherwise} \end{cases}$$

$O(n \log^2 n)$

How did I find this closed form? For today, just believe it, we'll give you a tool on Friday.

Small optimization: just sort by x and y once, and store the sorted versions.

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 100 \\ 2T\left(\frac{n}{2}\right) + O(n) & \text{otherwise} \end{cases}$$

$O(n \log n)$ for sorting, $O(n \log n)$ for everything else; total $O(n \log n)$.

It works!

Note that $\delta/2$ grid thing was ***just*** for the proof. Look back at the pseudocode, the grid isn't there.

Takeaways

A clever algorithm for finding closest points.

To make divide & conquer efficient, your conquer step must be faster than brute forcing the “crossing” pairs.

- Look for how the problem becomes simpler.