lecture 16

Divide and Conquer 1

- review mergesort (COMP 250)
- Closest pair of points

Divide and conquer :

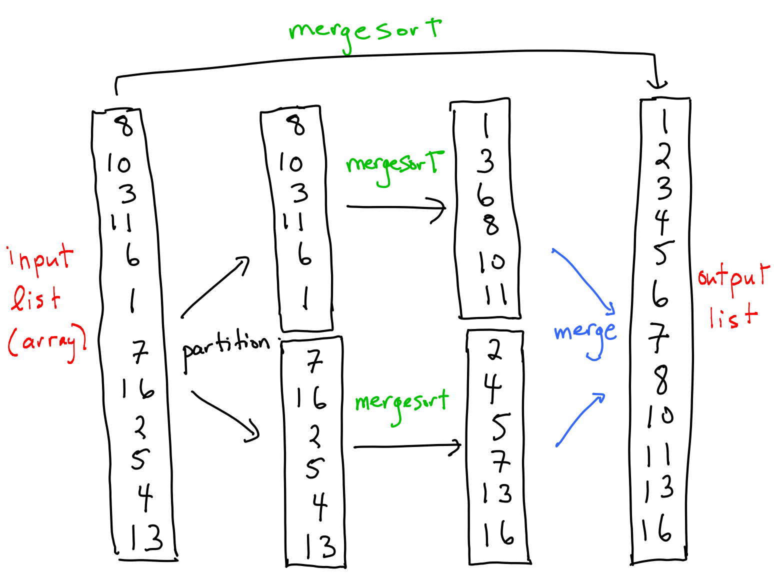
- partition problem into <u>independent</u> smaller problems (different from dynamic programming)
- solve each of smaller problems (recursion)
- Combine solutions

COMP 250 examples

- binary search
- merge sort ← most relevant for today's lecture
- quicksort

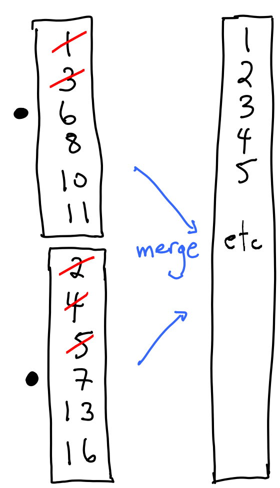
```
mergesort (list) {
   if (list.size == 1)
       return list
   else {
       partition list into two approximately
         equal size lists l1, l2
       l1 ← mergesort (l1)
       l2 ← mergesort (l2)

       return merge (l1, l2)
   }
}
```

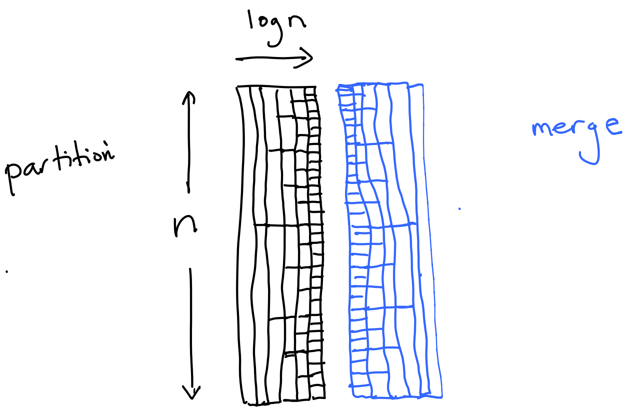


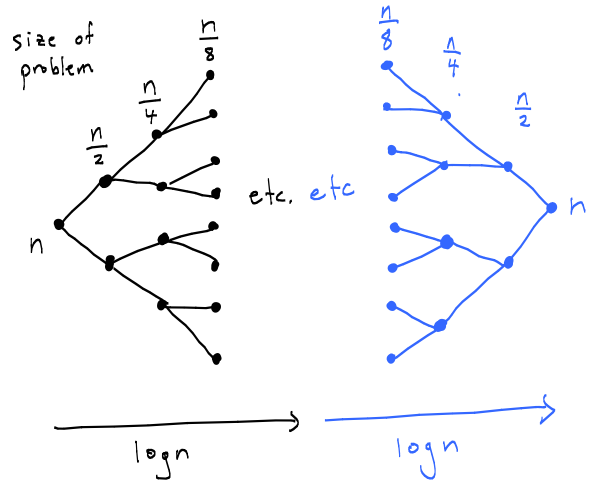
Given two sorted lists of size $\frac{n}{2}$, merge them to form a single sorted list of size n.

This can be done in time $O(n)$.



Subproblems of mergesort
(divide and conquer)





Let $t(n)$ be the time required to mergesort n items.

Recurrence:

$$t(n) = 2\,t\left(\tfrac{n}{2}\right) + cn$$

mergesort two sorted lists of size $\frac{n}{2}$

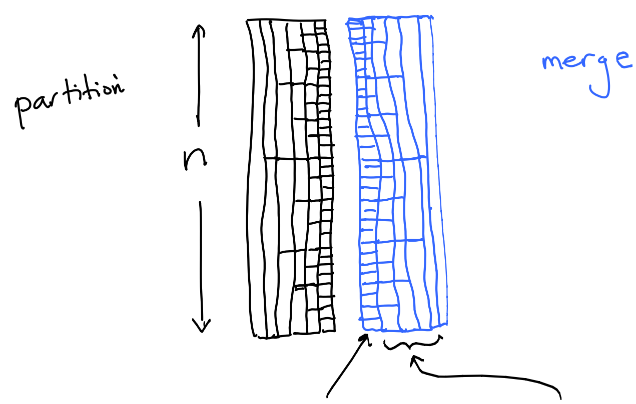
merge

Assume n is a power of 2.

$$\begin{aligned} t(n) &= 2\,t\left(\tfrac{n}{2}\right) + cn \\ &= 2\left\{2\,t\left(\tfrac{n}{4}\right) + c\tfrac{n}{2}\right\} + cn \\ &= 4\,t\left(\tfrac{n}{4}\right) + cn + cn \\ &= 4\left\{2\,t\left(\tfrac{n}{8}\right) + c\tfrac{n}{4}\right\} + 2cn \\ &= 8\,t\left(\tfrac{n}{8}\right) + 3cn \end{aligned}$$

$$\begin{aligned} t(n) &= 2\,t\left(\tfrac{n}{2}\right) + cn \\ &= 2^k\,t\left(\tfrac{n}{2^k}\right) + cn\cdot k \\ &= n\,t(1) + cn\cdot\log n \end{aligned}$$

Thus, $t(n)$ is $O(n \log n)$.



$$t(n) = n\,t(1) + cn\cdot\log n$$

Recurrences were covered in COMP 250. If you did not learn them well enough to be able to do the above <u>by yourself</u> then you need to review.

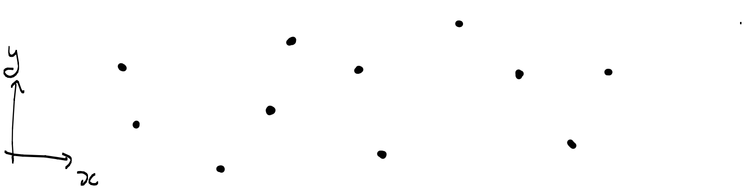
e.g. see my COMP 250 lectures 13 & 14 (+ Exercises 4)

lecture 16

Divide and Conquer I

- review mergesort (COMP 250)

- Closest pair of points



Suppose we have a set of points in a 2D plane $\{ (x_i, y_i) : i = 1 \text{ to } n \}$

For each pair of points, consider the Euclidean distance between them:

$$d(i,j) \equiv \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

<u>Problem</u>: find the closest pair i.e. minimize the distance $d(i,j)$ where $i \neq j$.

<u>Solution ("brute force")</u>:

$O(n^2)$ too slow!

```
closest pair = null
δ = ∞
for each i = 1 to n
    for each j = i+1 to n
        if d(i,j) < δ {
            closestpair = (i,j)
            δ = d(i,j)
        }
return closestpair
```

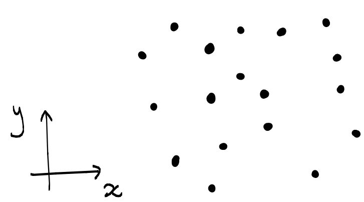
1D version of problem → $O(n \log n)$

- sort the points e.g. mergesort $O(n \log n)$

- check distances between successive points $O(n)$.

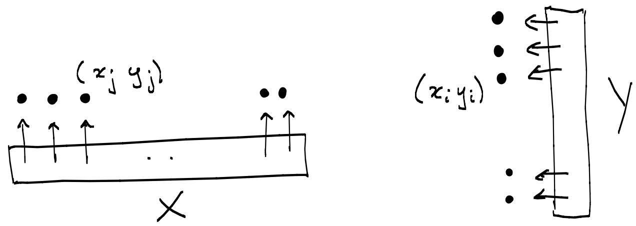
Can we solve the 2D problem in $O(n \log n)$?

<u>Solution for 2D</u> (Shamos & Hoey 1970's)



Begin by sorting points by x value, and sorting points by y value, giving two sorted arrays X and Y.

I will start explaining the algorithm using X only.



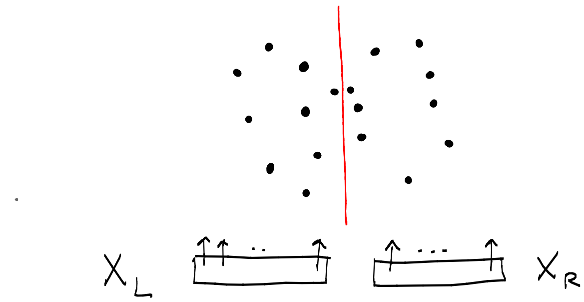
We'll see later how the Y ordering is used.

Partition X into two sets:

X_L has $n/2$ smallest x values ('left')

X_R has $n/2$ largest x values ('right')

Define a vertical line ℓ that separates X_L and X_R



Find closest pair (X) {

  Compute X_L X_R // previous slide

  Find closest pair (X_L) // recursive

  Find closest pair (X_R) // recursive

  Find the closest pair such that one point is in X_L and the other point is in X_R.

  Return the closest of the three pairs.

}

Note: this will be too slow. Later I will modify it slightly to allow speedup.

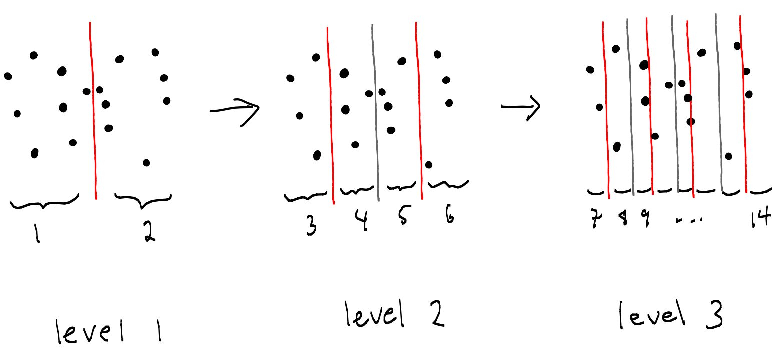
Find closest pair is recursive.

What are the subproblems?

What is the base case?

What is the recurrence?

What are the subproblems?



What is the base case?

Find closest pair (X) { // base case

  if $|X| \leq 3$ then compute closest pair by brute force and return it

  else {

    Compute X_L X_R

    Find closest pair (X_L)

    Find closest pair (X_R)

    Find the closest pair such that one point is in X_L and the other point is in X_R.

    Return the closest of the three pairs.

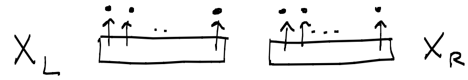
  }

}

What is the recurrence?

$t(n)$	Find closest pair (X) {
c_1	Compute X_L X_R
$t(\frac{n}{2})$	Find closest pair (X_L)
$t(\frac{n}{2})$	Find closest pair (X_R)
?	Find the closest pair such that one point is in X_L and the other point is in X_R.
c_2	Return the closest of the three pairs.
	}

$$t(n) = 2\,t\left(\frac{n}{2}\right) + \, ? \, + c$$



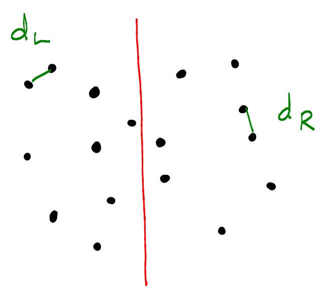
X_L and X_R each have $\frac{n}{2}$ points.

Thus there are $\frac{n}{2} \times \frac{n}{2}$ pairs of points such that one is in X_L and the other in X_R.

Finding the pair with minimum distance using "brute force" would be $O(n^2)$, which is too slow. (See Exercises.)

We next see how solve this problem in time $O(n)$ instead of $O(n^2)$.

Let the closest pair in X_L have distance d_L.
Let the closest pair in X_R have distance d_R.

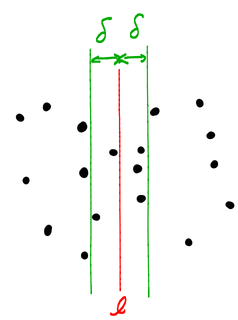


These are the pairs returned by the two recursive calls

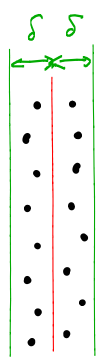
findClosestPair(X_L)
findClosestPair(X_R)

Let $\delta = \min(d_L, d_R)$.

Observe that to find the closest pair with one point in X_L and the other point in X_R, we only need to consider the points that are a distance δ from the line ℓ that separates L and R.



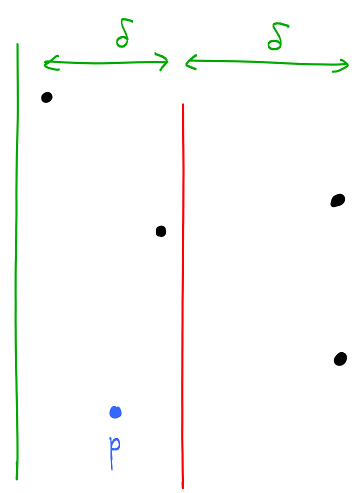
Observation doesn't necessarily reduce the number of points we need to consider. All points might be a distance $< \delta$ from line ℓ.



We need a second observation, which constrains pairs by their y values.

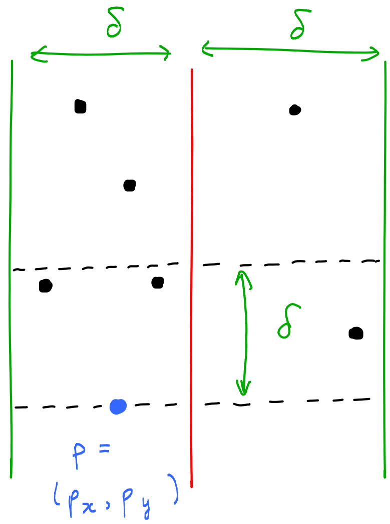
Consider a point "p" that lies between the two green lines.

Is there another point between the green lines that has a y value greater than that of p and is a distance less than δ from p?



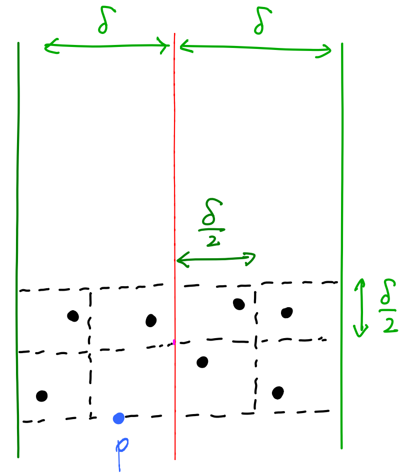
It is sufficient to check those points whose y values are between p_y and $p_y + \delta$.

Q: How many points do we need to check (worst case)?



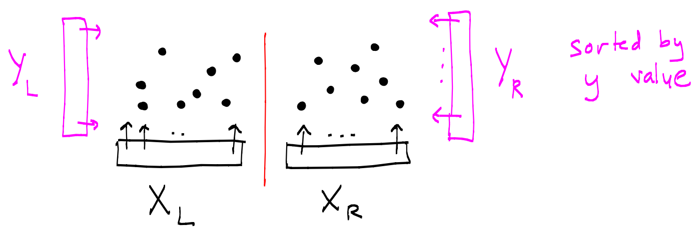
A: at most 7.

To see why, notice that square cells of width $\frac{\delta}{2}$ can contain at most 1 point.



We also now see why we need the points to be sorted by their y coordinate.

```
Find closest pair (X, Y) {
   Compute X_L, X_R, Y_L, Y_R
```




```
   Find closest pair (X_L, Y_L)
   Find closest pair (X_R, Y_R)
   Find the closest pair such that one point
     is in X_L and the other point is in X_R.      use Y
   Return the closest of the three pairs.
}
```

```
Find the closest pair such that one point
  is in X_L and the other point is in X_R. {

   Find all points that lie
      between the green lines.

   Starting from point with min y value,
     examine the distance to next
     7 points (sorted by Y). If we
     find a pair with distance < δ, make
     it the new closest pair & update δ.
}
```

More specifically (examine this at home)

```
middle = empty list
for i = 1 to n                          // find points between
   if Y[i] is between green lines       // green lines: O(n)
      middle.add( Y[i] )

for i = 1 to middle.size {              // O(n)
   for j = 1 to 7 {    // ignore out of bounds error
      tmp = d( middle[i], middle[i+j] )
      if tmp < δ {
         closest pair = ( middle[i], middle[i+j] )
         δ = tmp
   }  }
}
```

see Exercise Q2

```
t(n)       Find closest pair (X, Y) {
c_1 n         Compute X_L X_R Y_L Y_R
t(n/2)        Find closest pair( X_L, Y_L )
t(n/2)        Find closest pair( X_R, Y_R )
c_3 n         Find the closest pair such that one point
                is in X_L and the other point is in X_R.
c_2           Return the closest of the three pairs.
           }
```

uses Y? Y_L Y_R? See Exercises.

$$\boxed{t(n) = 2\,t\left(\frac{n}{2}\right) + c_3 n + c}$$

which is the same as mergesort.