CS 381 – FALL 2019

Week 4.2, Wed, Sept 11

Homework 2 Due: September 16th, 2019 @ 11:59PM on Gradescope
Homework 1: Graded (see Gradescope)

Homework 1

- Maximum: 100
- Mean: 89.2
- Median: 92.5
- Standard Deviation: 13.26

Regrade Requests?

- Submit on Gradescope before Sept 24 (10PM)
- Your score may go up or down
- Appeal Result of Regrade Request?
  - Contact me directly
  - 2 point penalty/bonus depending on outcome

Homework 2 Reminders

- You must include a resource & collaborator statement (0 points without one).
- You Must Typeset Your Solutions
  - Photocopies of handwritten work will receive 0 points
  - Exception: You may include photocopies of diagrams, but the main solution should be typed.
  - Expectation to use mathematical symbols
    - Sum (n^(1/2)+n^(4n))^2/2^n from n =1 to k versus

$$\sum_{n=1}^{k} \frac{(\sqrt{n} + n^{4n})^2}{2^n}$$

5.4 Closest Pair of Points

Closest Pair of Points in 1-Dimension

Input: Array A[1…n] of numbers (not sorted)

Output: (i.j) minimizing $|A[i] - A[j]|$

Example:

Input: A = [-1 7 **2** 9 5 **1** 11]

Output: (3,6)

$$|A[3] - A[6]| = |2 - 1| = 1$$

Clicker Question: Suppose the array A is already sorted. How long does it take to find (i,j)? Find the tightest answer.

A. O(1) B. O(log n) C. O(n) D. O(n log n) E. O(n^2)

Closest Pair of Points in 1-Dimension

Input: Array A[1…n] of numbers (not sorted)

Output: (i.j) minimizing $|A[i] - A[j]|$

Example:

Input: A = [-1 7 **2** 9 5 **1** 11]

Output: (3,6)

$$|\boldsymbol{A}[3] - \boldsymbol{A}[6]| = |2 - 1| = 1$$

Easy Solution:

- **Observation:** if list is sorted can find optimal pair with j=i+1

```
Sort(A)
Min = 0
For i = 1 to n-1
    If Min > |A[i]-A[i+1]| then Min = |A[i]-A[i+1]| ;
```

Closest Pair of Points

Closest pair. Given n points in the plane, find a pair with smallest Euclidean distance between them.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

fast closest pair inspired fast algorithms for these problems

Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

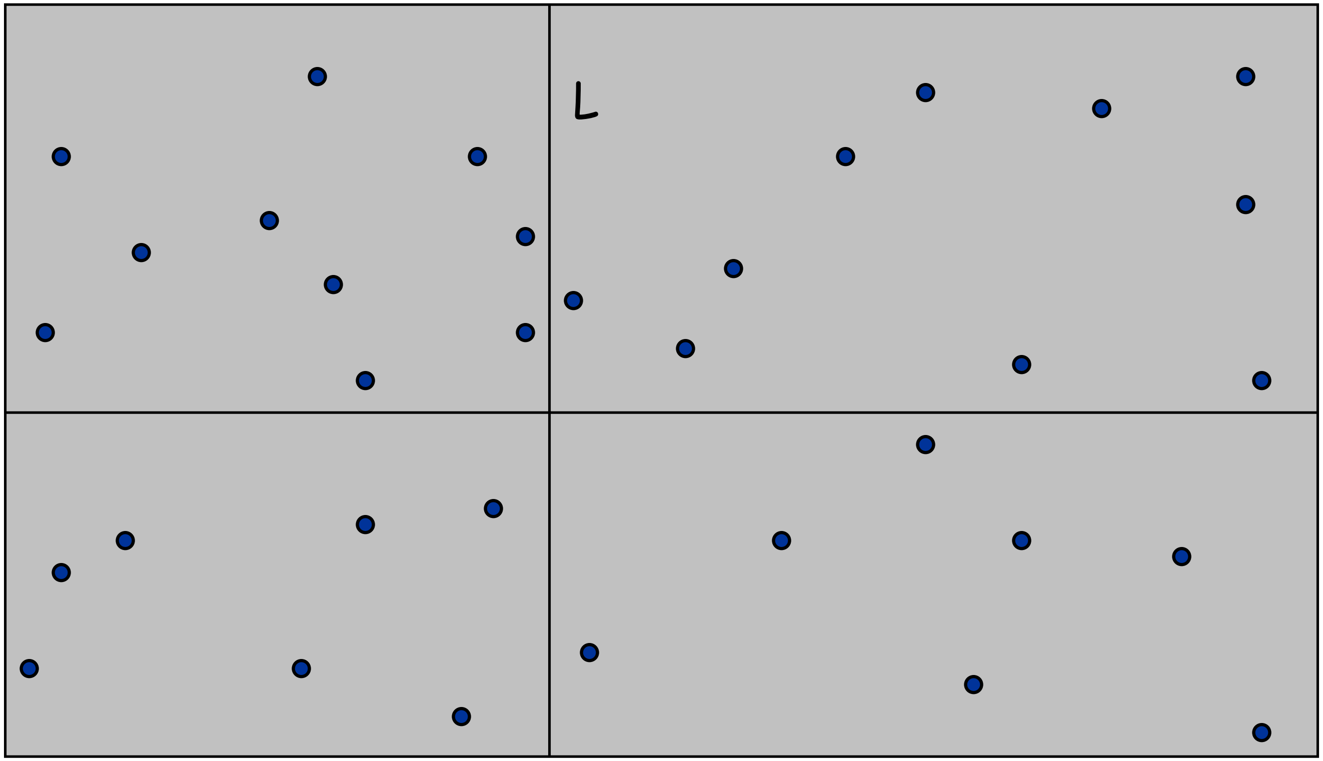
1-D version. O(n log n) easy if points are on a line.

Assumption. No two points have same x coordinate.

to make presentation cleaner

Closest Pair of Points: First Attempt

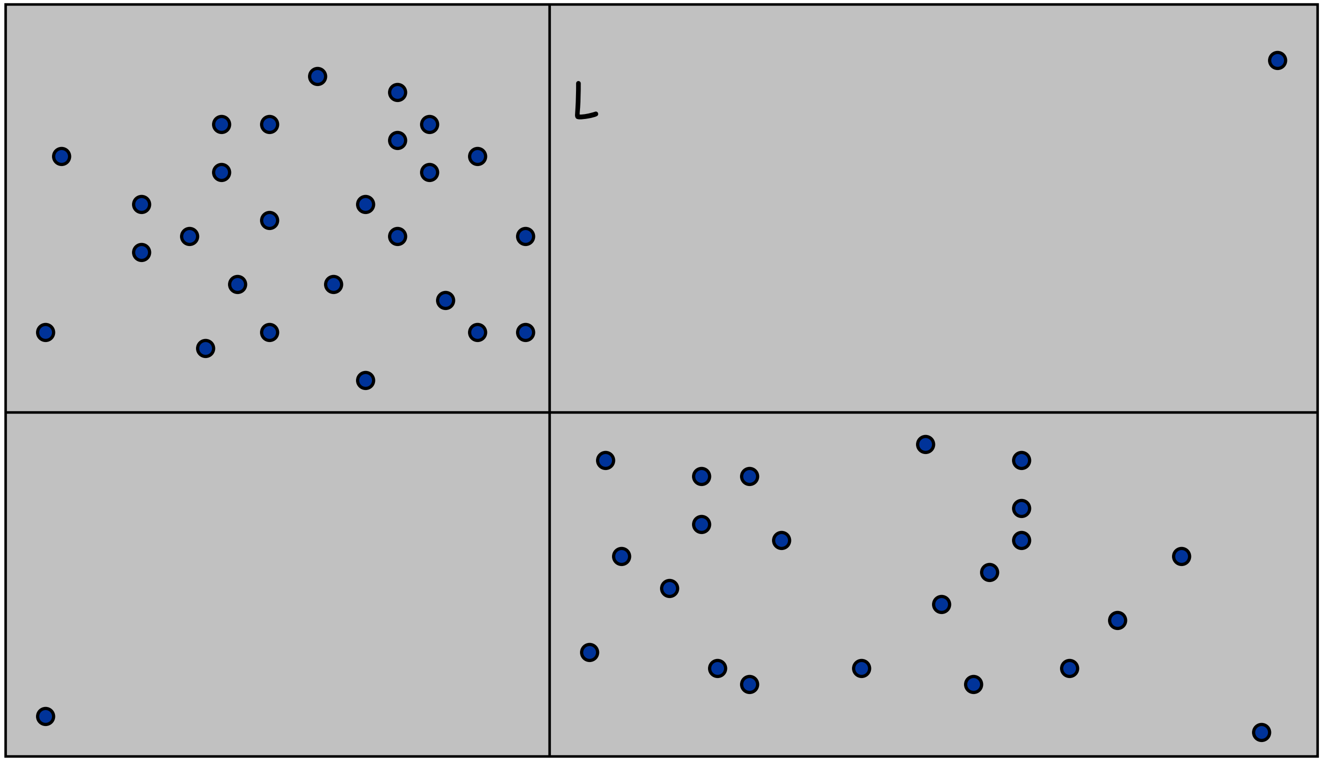
Divide. Sub-divide region into 4 quadrants.



Closest Pair of Points: First Attempt

Divide. Sub-divide region into 4 quadrants.

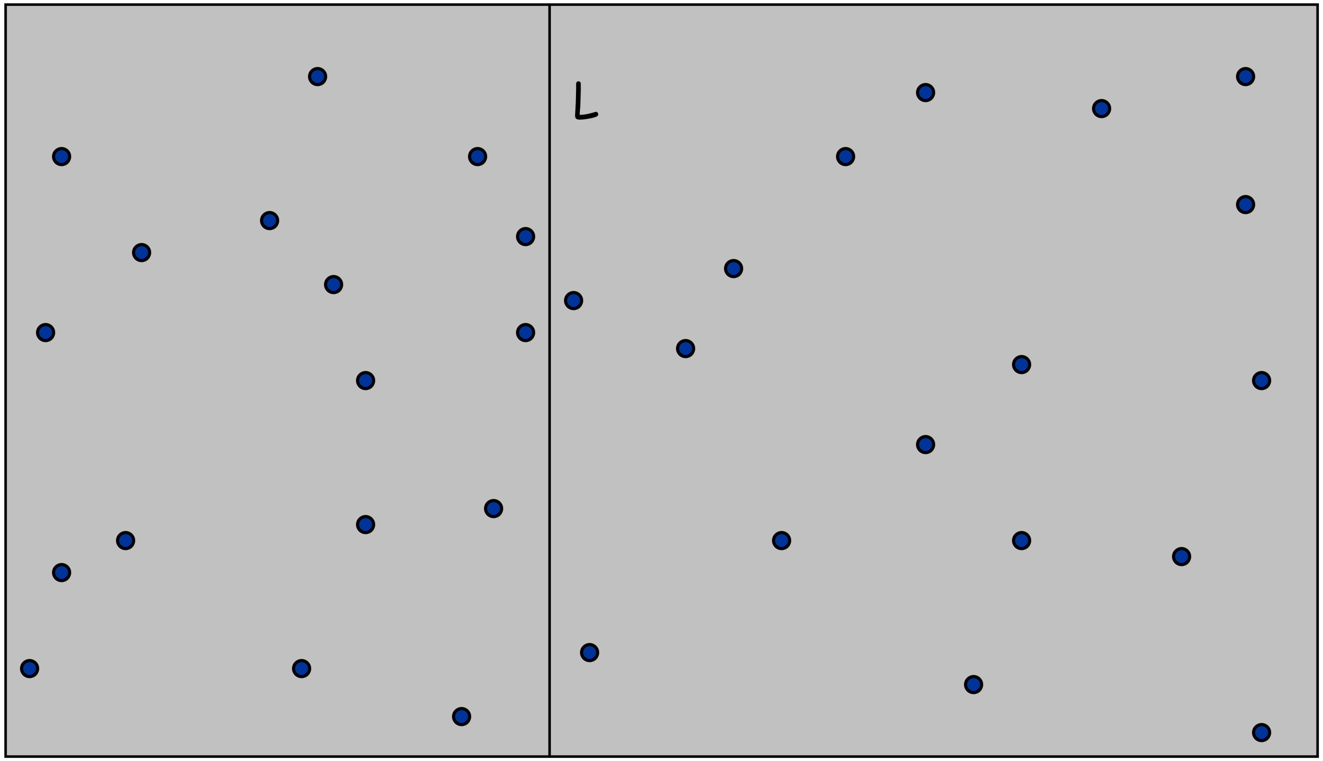
Obstacle. Impossible to ensure n/4 points in each piece.



Closest Pair of Points

Algorithm.

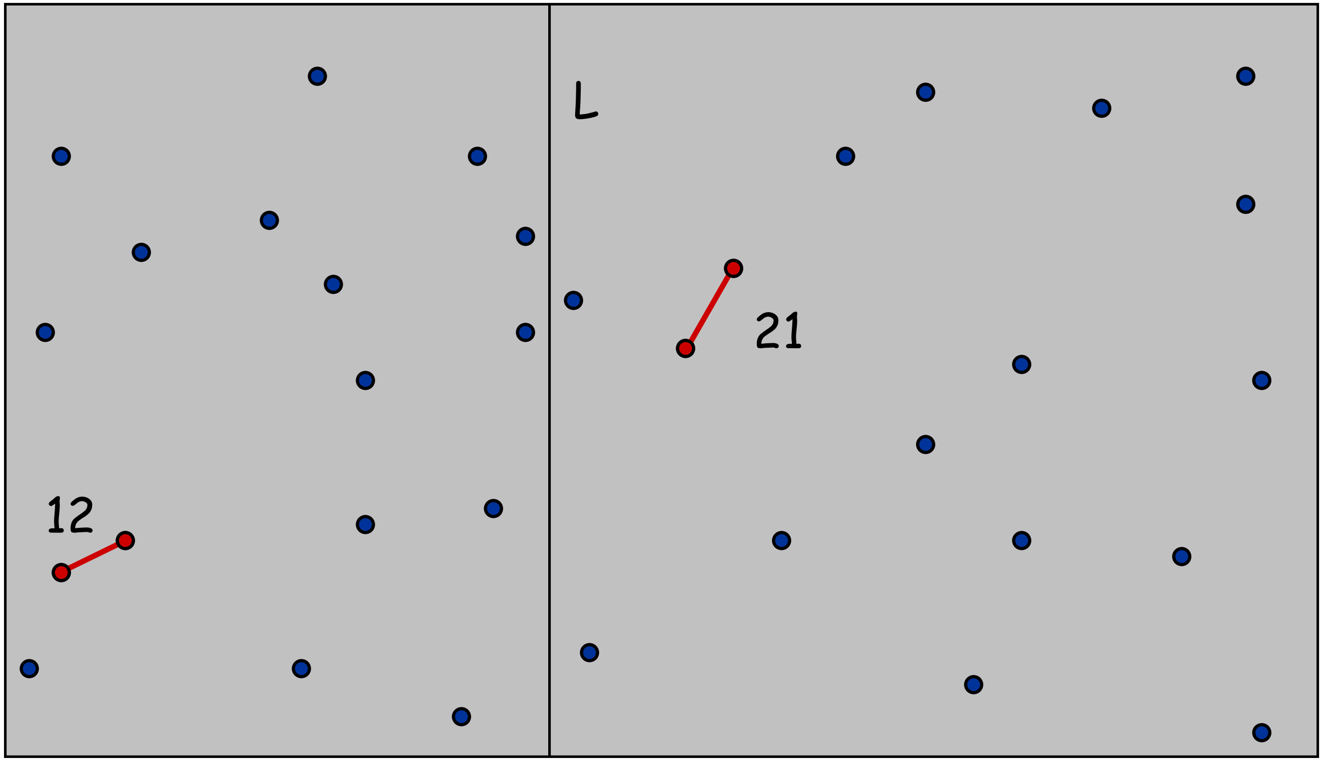
- Divide: draw vertical line L so that roughly $\frac{1}{2}$n points on each side.



Closest Pair of Points

Algorithm.

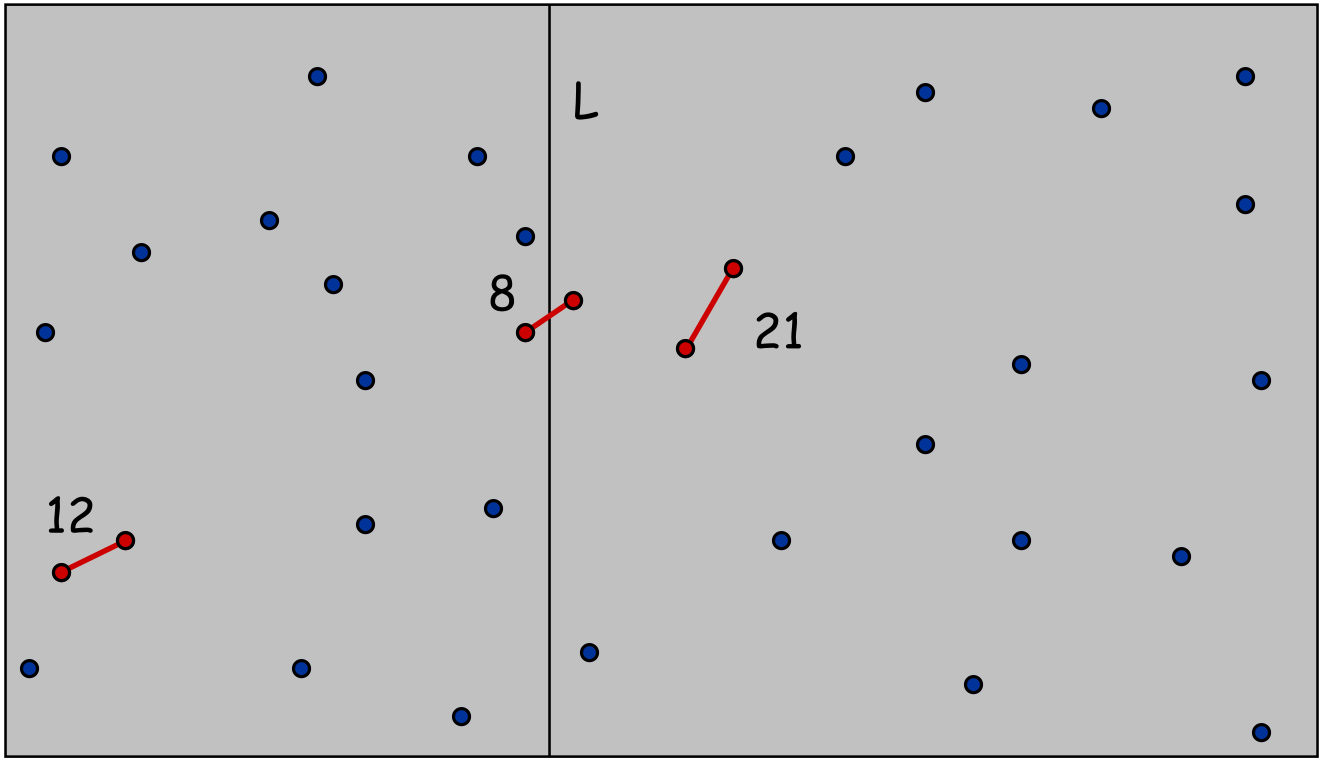
- Divide: draw vertical line L so that roughly $\frac{1}{2}$n points on each side.
- Conquer: find closest pair in each side recursively.



Closest Pair of Points

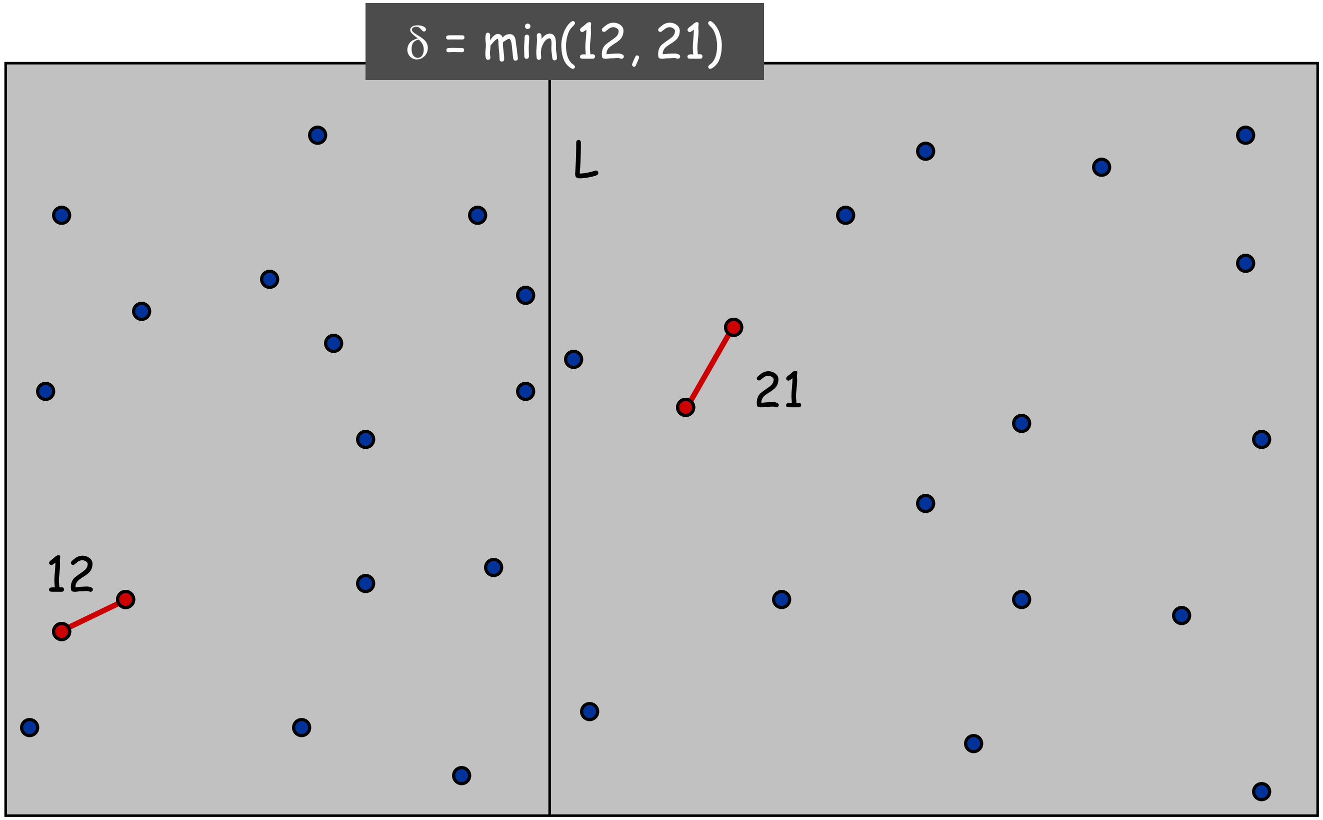
Algorithm.

- Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
- Conquer: find closest pair in each side recursively.
- Combine: find closest pair with one point in each side. ← seems like $\Theta(n^2)$
- Return best of 3 solutions.



Closest Pair of Points

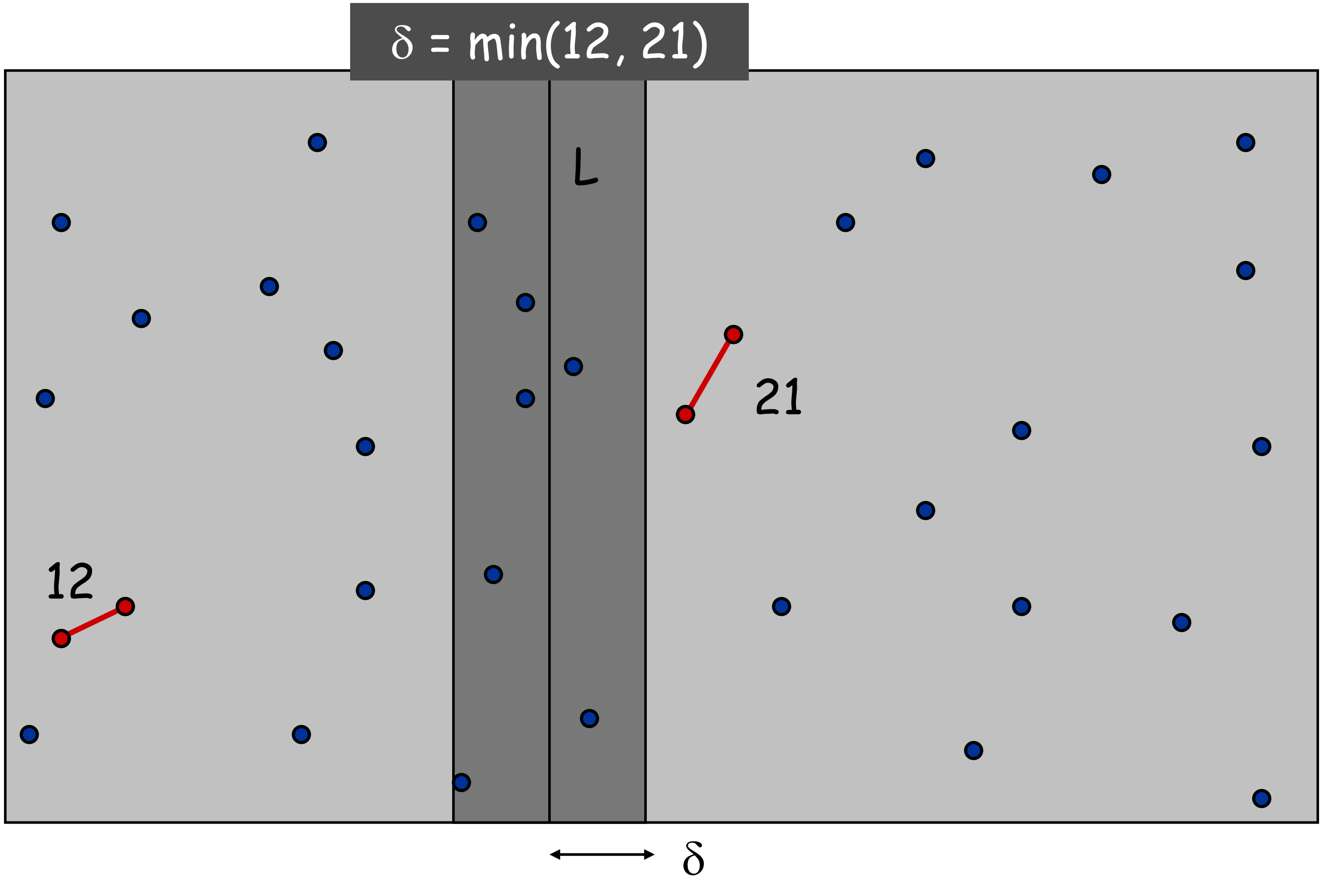
Find closest pair with one point in each side, assuming that distance < δ.



Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

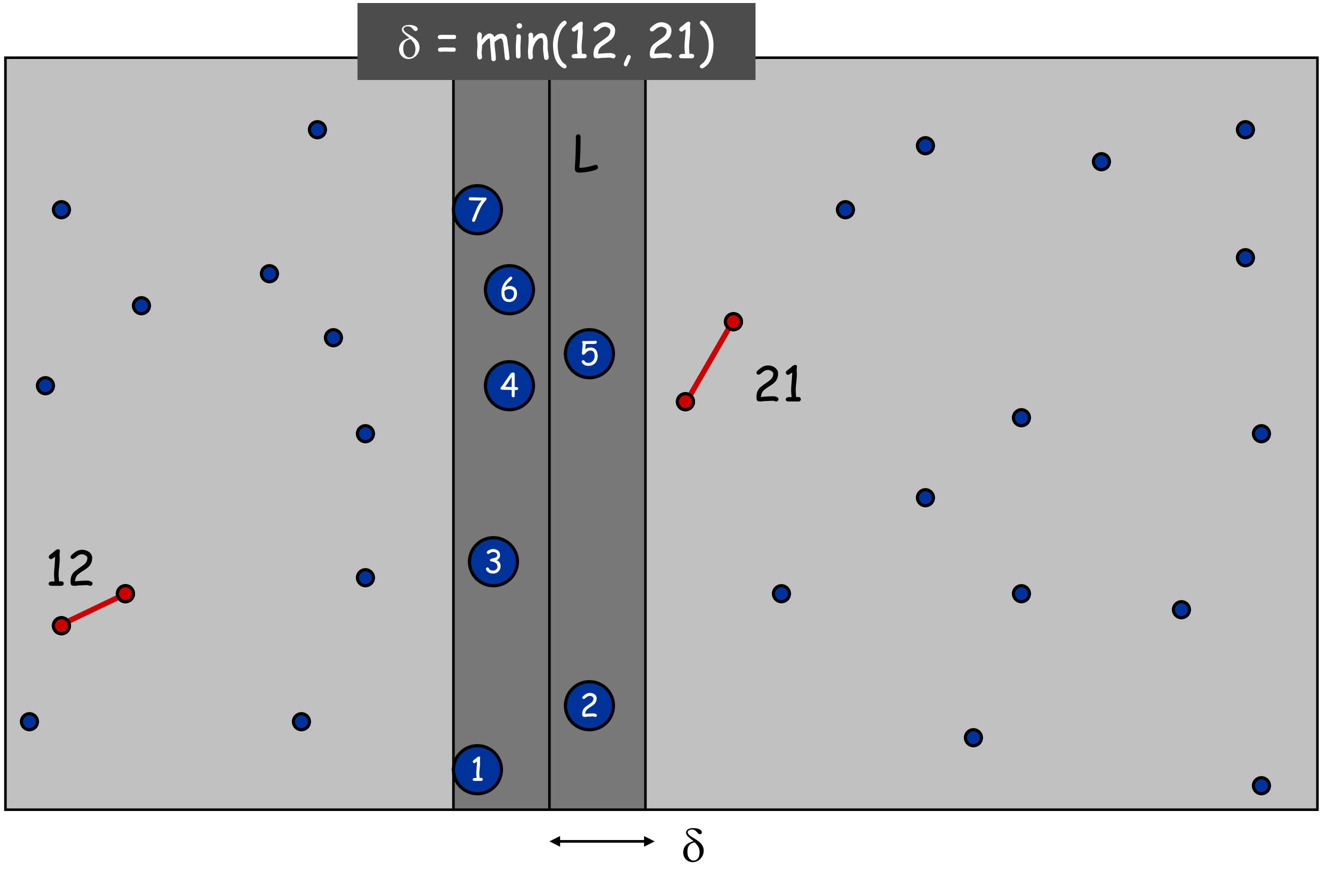
- Observation: only need to consider points within δ of line L.



Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

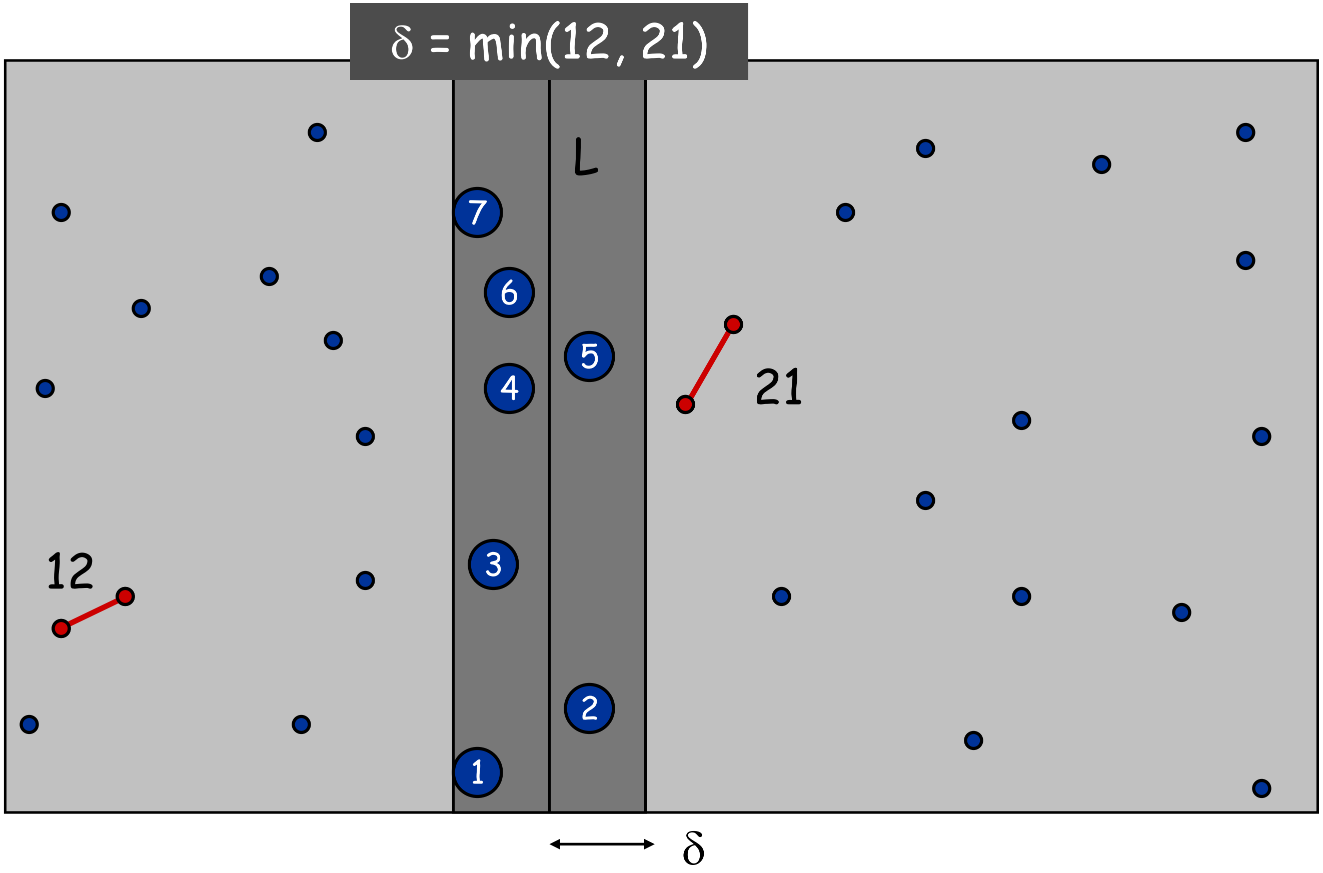
- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.



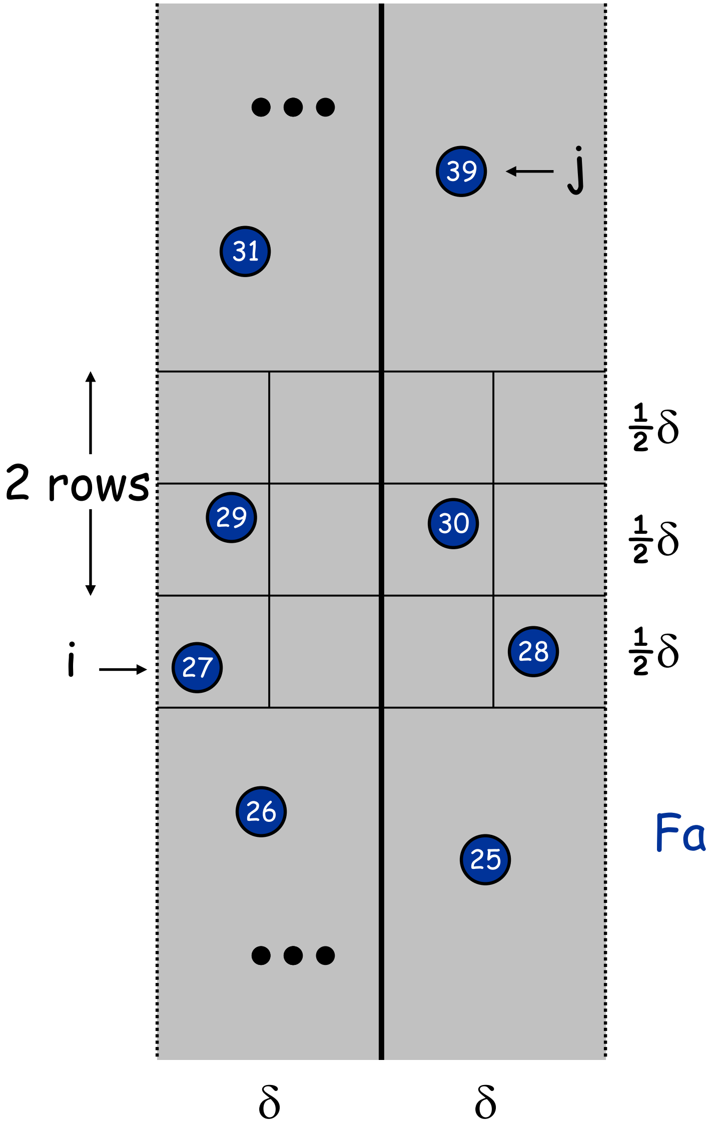
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.
- Only check distances of those within 11 positions in sorted list!



Closest Pair of Points



Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

Claim. If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ.

Pf.

- No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
- Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta)$. ▪

Fact. Still true if we replace 12 with 7.

Final Merge Step

Precondition:

- Filtered Out points **further than δ from separation line L**
- Remaining points R sorted by y

```
For i=1 to |R|
   For j=i+1 to i+11
      p1 = R[i],  p2 = R[j]
      oppositeSide = (p1.x < L and p2.x >= L)
                          OR (p1.x > L and p2.x < L)
      if oppositeSide  and dist(p1,p2) < $\delta$  then
            $\delta$= dist(p1,p2)
```

Running Time: O(n)

Scan points in y-order and compare distance between each point and next 11 neighbors. If any of these distances is less than δ, update δ.

Closest Pair Algorithm

```
Closest-Pair($p_1$, …, $p_n$) {
   Compute separation line L such that half the points          O(n log n)
   are on one side and half on the other side.

   $\delta_1$ = Closest-Pair(left half)                          2T(n / 2)
   $\delta_2$ = Closest-Pair(right half)
   $\delta$  = min($\delta_1$, $\delta_2$)

   Delete all points further than $\delta$ from separation line L   O(n)

   Sort remaining points by y-coordinate.                        O(n log n)

   Scan points in y-order and compare distance between
   each point and next 11 neighbors. If any of these             O(n)
   distances is less than $\delta$, update $\delta$.

   return $\delta$.
}
```

Closest Pair of Points

Recurrence of DQ algorithm

$$T(n) \le 2T(n/2) + O(n \log n) \quad \Rightarrow \quad T(n) = O(n \log^2 n)$$

Use a generalization of case 2 in Master theorem

$f(n) = \Theta(n^{\log_b a} \lg^k n)$ **for some constant** $k \ge 0$.

- $f(n)$ **and** $n^{\log_b a}$ **grow at similar rates.**

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

How do we achieve O(n log n)? We will just give an outline, look at the book for details.

Solution (Idea)

- Use linear time median finding algorithm to determine line L and make the two recursive calls.

- Each recursive call returns its δ ***and*** the points in its region sorted by y coordinates.

- Merge: Sort y-lists by merging two returned sorted lists in time O(n). Same as mergesort!

Terminate recursion when n<4 or some other small constant and solve the resulting small problem by brute force.

$$T(n) \leq 2T\big(n/2\big) + O(n) \quad \Rightarrow \quad \mathrm{T}(n) = O(n \log n)$$

Closest Pair Algorithm

```
Closest-Pair-And-Sort ($p_1$, …, $p_n$) {
   Compute separation line L such that half the points              O(n)
   are on one side and half on the other side.

   ($\delta_1$,$S_1$) = Closest-Pair-And-Sort(left half)   //sort by y         2T(n / 2)
   ($\delta_2$,$S_2$) = Closest-Pair-And-Sort(right half)  //sort by y
     $\delta$    = min($\delta_1$, $\delta_2$)

   S = Merge($S_1$,$S_2$)                                             O(n)
   S'= Filter(L,$\delta$, S)
    Delete all points further than $\delta$ from separation line L

   ~~Sort remaining points by y-coordinate.~~                         ~~O(n log n)~~

   Scan points in y-order and compare distance between
   each point and next 11 neighbors. If any of these                  O(n)
   distances is less than $\delta$, update $\delta$.

   return ($\delta$,S).
}
```

Majority Element Problem

Input: Array A[1…n] of numbers (not sorted)

Output:

x ---- if x=A[i] for more than n/2 array elements
"N/A" ---- if no majority element exists

Example 1:

Input: A = [1 7 2 9 7 2 7]
Output: "N/A"

Example 2:

Input: A = [1 7 2 **7** 7 2 7]
Output: 7

Observation: If A does contain a majority element x then x=Median(A)

Clicker Question: Sorted Majority problem

Suppose A is an array of size n containing increasingly sorted entries. We can determine whether A has a majority element in what time (check best bound)

A. O(1)
B. O(log n)
C. O($\log^2$ n)
D. O(n)
E. O(n log n)

Clicker Question: Sorted Majority problem

Suppose A is an array of size n containing increasingly sorted entries. We can determine whether A has a majority element in what time (check best bound)

~~A. O(1)~~

B. O(log n)

C. O($\log^2$ n)

D. O(n)

E. O(n log n)

Majority Element Problem

Input: Array A[1…n] of numbers (not sorted)

Output:

x ---- if x=A[i] for more than n/2 array elements

"N/A" ---- if no majority element exists

Solution 1: Sort array, find median and binary search $O(n \log n)$

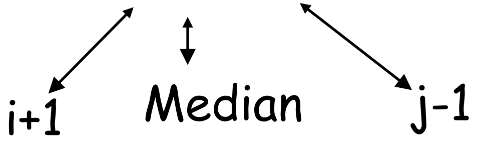
- i := location of greatest element smaller than median x
- j:= location of smallest element greater than median x
- **Return**

$$\begin{cases} \mathrm{N/A} & j - i - 1 \le \dfrac{n}{2} \\ \boldsymbol{x} = A\left[\left\lceil \dfrac{n}{2} \right\rceil\right] & j - i - 1 > \dfrac{n}{2} \end{cases}$$

Example:

Input: A = [1 2 7 7 7 7 8]

Output: 7



i+1 Median j-1

Majority Element Problem

Input: Array A[1…n] of numbers (not sorted)

Output:

x ---- if x=A[i] for more than n/2 array elements

"N/A" ---- if no majority element exists

Solution 2: Find Median and Scan Array to Count Matches

```
X= Median(A)
Count = 0
For i = 1 to n
    If X=A[i] then Count= Count + 1;
Return
```

$$\begin{cases} \mathrm{N/A} & count \le \frac{n}{2} \\ \boldsymbol{x} = A\left[\left\lceil\frac{n}{2}\right\rceil\right] & count > \frac{n}{2} \end{cases}$$

Running time: O(n)