Divide and Conquer: Introduction

Lecture 5

Akshar Varma

11th July, 2023

CS3000 Algorithms and Data

Divide-And-Conquer Paradigm

Sorting: Mergesort

Comparison Sorting Lower Bound

Closest pair of points

Summary

1. Divide-And-Conquer Paradigm

Blah

Divide-And-Conquer Paradigm

Divide-and-conquer:

1. Divide up problem into several subproblems (of the same kind).
2. Solve (conquer) each subproblem recursively.
3. Combine solutions to subproblems into overall solution.

The most common usage (two examples today):

1. Divide problem of size n into 2 subproblems of size $n/2$. $\longleftarrow O(n)$
2. Solve (conquer) two subproblems recursively.
3. Combine two solutions into overall solution. $\longleftarrow O(n)$

Consequence:

- Brute force: $O(n^2)$.
- Divide-and-conquer: $O(n \log n)$.

2. Sorting: Mergesort

The Sorting Problem

- **Problem:** Given a list L of n elements from a totally ordered universe, rearrange them in ascending order.
- Example: $[3, 2, 5, 1, 9] \longrightarrow [1, 2, 3, 5, 9]$
- *Obvious applications:*
  - Organize an MP3 library (by artist/album name/title).
  - Display (DuckDuckGo/Google) search results in order of relevance.
  - List timeline/newsfeed items in reverse chronological order.
- *Some problems become easier once elements are sorted:*
  - Identify statistical outliers.
  - Binary search in a database.
  - Remove duplicates in a mailing list.
- *Many non-obvious applications:* Closest pair of points, Counting Inversions, Convex hull, Interval scheduling/Interval partitioning, Scheduling to minimize maximum lateness, Minimum spanning trees (Kruskal's algorithm), etc.

Mergesort

- Split array into two halves.
- Recursively sort left half.
- Recursively sort right half.
- Merge the two sorted halves to make a whole sorted array.
- *Example:*
  - Input
    $[A, L, G, O, R, I, T, H, M, S]$
  - Split into two halves
    $[A, L, G, O, R\}, \{I, T, H, M, S]$
  - Sort left half
    $[A, G, L, O, R], [I, T, H, M, S]$
  - Sort right half
    $[A, G, L, O, R], [H, I, M, S, T]$
  - Merge results
    $[A, G, H, I, L, M, O, R, S, T]$

Components of Mergesort

- When there's a single element, just return input as is. [Base case]
- Merging is the core of the algorithm.
- *Goal:* Given sorted lists A and B, merge them into a sorted list C.
- Example on board: $A = [2, 3, 5, 6, 8], B = [1, 3, 5, 7, 10]$

- General algorithm:
  - Scan A and B from left to right.
  - Compare A_i and B_j.
  - If $A_i \leq B_j$, append A_i to C (remaining elements in B is at least as big).
  - If $A_i > B_j$, append B_j to C (smaller than remaining elements in A).

Implementation of MERGESORT(L)

Input: List L of n elements from a totally ordered universe.
Output: The n elements of L in ascending order.

if $n = 1$ **then**	If there is only one element
return L	then it is already sorted
A = *Mergesort*(*L*$[1 \mathinner{..} \frac{n}{2}]$)	$T(n/2)$ time; recursive call
B = *Mergesort*(*L*$[\frac{n}{2} \mathinner{..} n]$)	$T(n/2)$ time; recursive call
L = *Merge*(*A*, *B*)	$\Theta(n)$ time
return L	Merged array is sorted L

(Recursive) Implementation of MERGE(A, B)

Input: Two sorted lists A, B.
Output: Single sorted list with values from both A and B.

if $\lvert A \rvert == 0$ **then**	if A is empty
return B	then just return B
if $\lvert B \rvert == 0$ **then**	if B is empty
return A	then just return A
if $A[1] < B[1]$ **then**	if first element of A is smaller
init = $[A[1], B[1]]$	it should go first
else	otherwise
init = $[B[1], A[1]]$	it goes second
return *init* $++$ *Merge*$(A[2\,..], B[2\,..])$	recurse on remaining elements

Def: $T(n) =$ max number of comparisons to mergesort a list of length n.

Recurrence:

$$T(n) \leq \begin{cases} 0 & n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & n > 1 \end{cases} \tag{1}$$

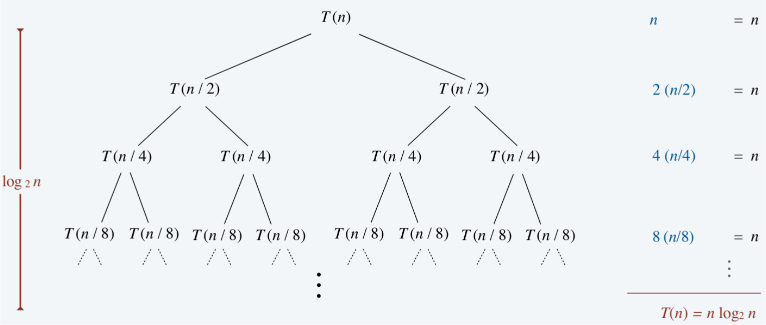
Solution: $T(n) = O(n \log_2 n)$

Proofs: We'll go over various ways to prove this. Inductive proofs, Recurrence trees, Master Theorem.

Recursion Tree Proof $(n = 2^k)$

Proposition: Assuming $n = 2^k$ (a power of 2), $T(n) = n \log n$ if $T(n)$ satisfies the following recurrence.

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T\left(\frac{n}{2}\right) + n & n > 1 \end{cases} \tag{2}$$



Recursion Tree Based Proof

Inductive Proof $(n = 2^k)$

Proposition: Assuming $n = 2^k$ (a power of 2), $T(n) = n \log n$ if $T(n)$ satisfies the following recurrence.

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T\left(\frac{n}{2}\right) + n & n > 1 \end{cases} \tag{2}$$

Proof: [by induction on n]

- Base case: when $n = 1, T(1) = 0 = n \log_2 n$
- Inductive hypothesis: assume $T(n) = n \log_2 n$
- Goal: show that $T(2n) = 2n \log_2 (2n)$

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n(\log_2 (2n) - 1) + 2n \\ &= 2n \log_2 (2n) \end{aligned}$$

Proposition: $T(n) \le n\lceil \log n \rceil$ if $T(n)$ satisfies the following recurrence.

$$T(n) \le \begin{cases} 0 & n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + n & n > 1 \end{cases} \quad (1)$$

Proof: [by strong induction on n]

- Base case: when $n = 1, T(1) = 0 \le n \log_2 n$
- Define $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$. Note that $n = n_1 + n_2$
- Inductive hypothesis: Assume true for $1, 2, \ldots, n-1$

$$\begin{aligned} T(n) \le T(n_1) + T(n_2) + n &= n_1\lceil \log_2 n_1 \rceil + n_2\lceil \log_2 n_2 \rceil + n \\ &\le n_1\lceil \log_2 n_2 \rceil + n_2\lceil \log_2 n_2 \rceil + n \\ &= n\lceil \log_2 n_2 \rceil + n \\ &= n(\lceil \log_2 n \rceil - 1) + n \qquad (\because n_2 \le \lceil 2^{\lceil \log_2 n \rceil}/2 \rceil) \\ &= n\lceil \log_2 n \rceil \end{aligned}$$

3. Comparison Sorting Lower Bound

Can we sort faster?

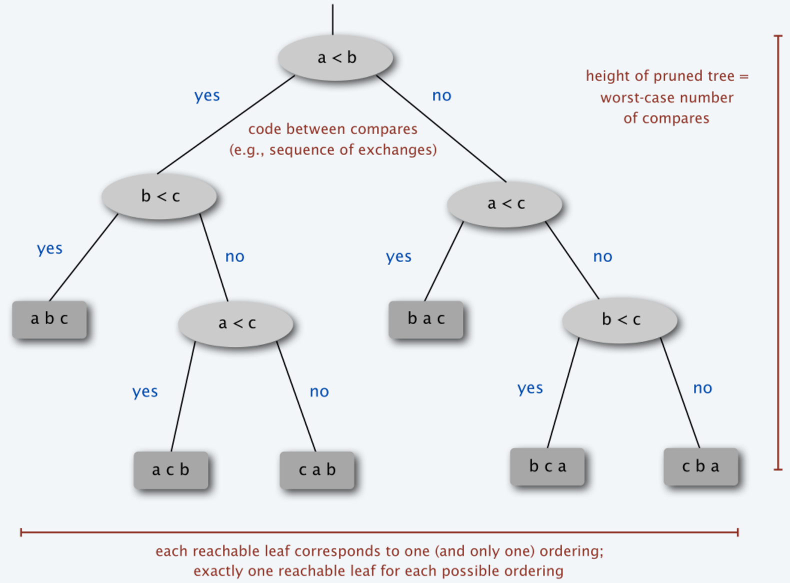
- We saw an $O(n \log n)$ algorithm. Can we do better?
- If not, can we show that any conceivable algorithm will be $\Omega(n \log n)$?
- *Model of Computation:* Comparison Trees
  - Can access the elements only through pairwise comparisons.
  - All other operations (control, data movement, etc.) are free.
- *Cost Model:* Number of Comparisons
- Is this realistic? Depends
  - Yes, for most languages you'll see/know: Python, Java, C/C++

    `sort`(*, *key=None*, *reverse=None*)

    This method sorts the list in place, using only `<` comparisons between items. Exceptions are not suppressed - if any comparison operations fail, the entire sort operation will fail (and the list will likely be left in a partially modified state).

  - Yes, for most sorts you'll see: Mergesort, Heapsort, Quicksort
  - No, for certain sorts that assume something about your data.

Comparison Tree (for three distinct elements a, b, and c)

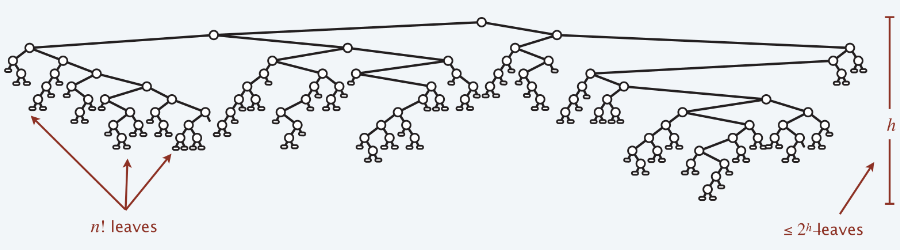


Lower Bound for Comparison Based Sorting

Theorem: Any deterministic comparison-based sorting algorithm must make $\Omega(n \log n)$ comparisons in the worst-case.

Proof: [Information theoretic]

- Assume array consists of n distinct values $a_1, \dots, a_n$.
- Worst-case number of compares $=$ height h of comparison tree.
- Binary tree of height h can have at most 2^h leaves.
- $n!$ different possible orderings means we need $n!$ reachable leaves.



Lower Bound for Comparison Based Sorting

Theorem: Any deterministic comparison-based sorting algorithm must make $\Omega(n \log n)$ comparisons in the worst-case.

Proof: [Information theoretic]

- Assume array consists of n distinct values $a_1, \dots, a_n$.
- Worst-case number of compares $=$ height h of comparison tree.
- Binary tree of height h can have at most 2^h leaves.
- $n!$ different possible orderings means we need $n!$ reachable leaves.

$$
\begin{aligned}
2^h &\geq \text{Number of leaves} \geq n! \\
\implies h &\geq \log_2(n!) \\
\implies h &\geq n \log_2(n) - n/\ln 2
\end{aligned}
$$

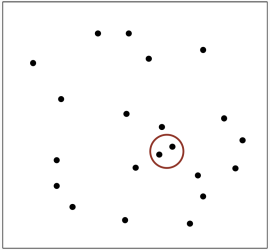
Summary of Divide-and-Conquer for Sorting

- We saw that Sorting can benefit from Divide-and-Conquer.
- Naively $O(n^2)$ time by comparing all pairs of elements.
- With Divide-and-Conquer, we reduce it to $O(n \log n)$ time.
- Any comparison based algorithm needs $\Omega(n \log n)$ time.
- So Divide-and-Conquer gets us to the “best” possible algorithm.

4. Closest pair of points

Closest Pair of Points

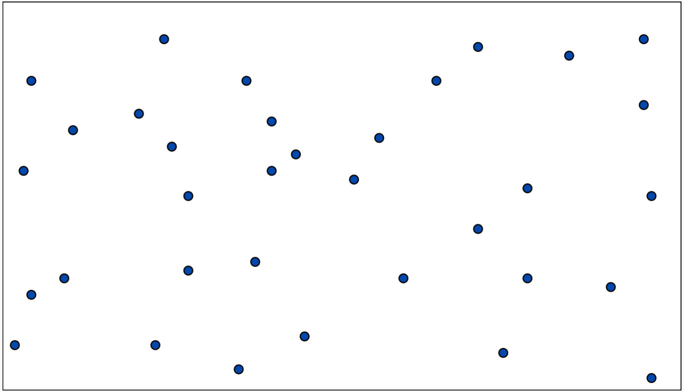
- *Closest Pair Problem:* Given n points in the plane, find a pair of points with the smallest Euclidean distance between them.



- *Brute Force:* Check all pairwise distances. In $\Theta(n^2)$ time.
- *1D version:* Just sort all points are on a line. In $O(n \log n)$ time!
- *Non-degeneracy assumption:* Note that to avoid weird situations we assume that no two points have exactly the same x-coordinate.

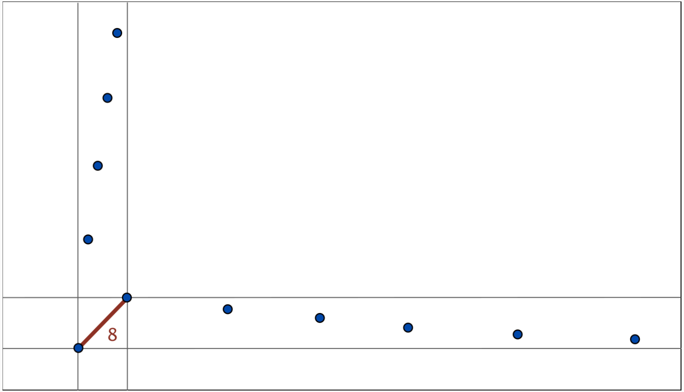
2D Closest Pair - First Attempt

- Sort by x-coordinate and look at nearby points.
- Similarly, sort by y-coordinate and look at nearby points.



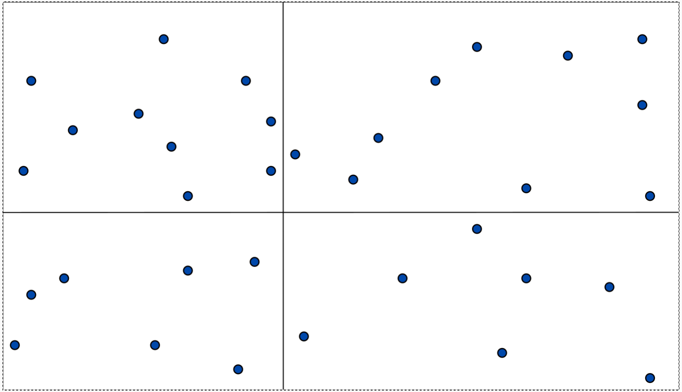
2D Closest Pair - First Attempt

- Sort by x-coordinate and look at nearby points.
- Similarly, sort by y-coordinate and look at nearby points.
- *Obstacle:* May miss a close pair that's not the closest in x or in y.



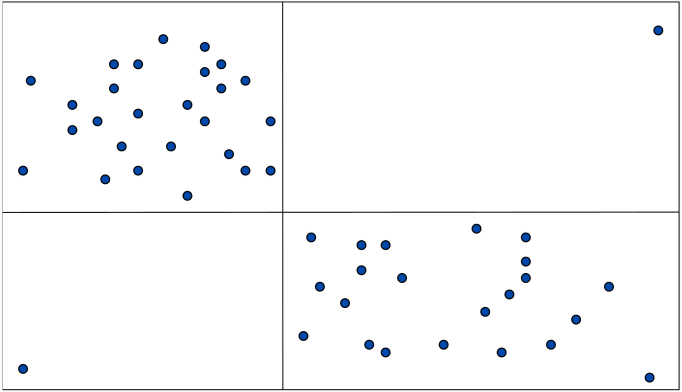
2D Closest Pair - Second Attempt

- Divide region into 4 quadrants.



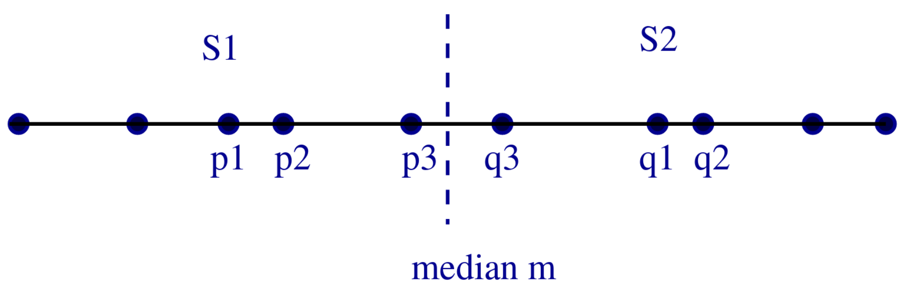
2D Closest Pair - Second Attempt

- Divide region into 4 quadrants.
- *Obstacle:* Impossible to ensure $n/4$ points in each piece. Without that, there is no real benefit to divide and conquer.



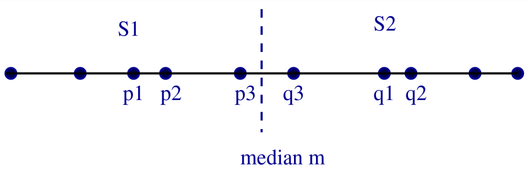
Divide and Conquer for 1D Closest Pair

- In 1D, we can sort the points. Allows solving in $O(n \log n)$ time.
- But sorting doesn't generalize to higher dimensions. Let's attempt a Divide and Conquer algorithm instead.
- Divide the points S into S_1 and S_2 of equal size such that $p < q$ for all $p \in S_1, q \in S_2$.



- Recursively compute closest pair (p_1, p_2) in S_1 and (q_1, q_2) in S_2.
- Let δ be the smallest distance yet: $\delta = \min(|p_1 - p_2|, |q_1 - q_2|)$
- The closest pair will either be (p_1, p_2) or (q_1, q_2) or a pair (p_3, q_3) for $p_3 \in S_1, q_3 \in S_2$.

Divide and Conquer for 1D Closest Pair (continued)



- The closest pair will either be (p_1, p_2) or (q_1, q_2) or (p_3, q_3).
- *Note 1:* p_3 and q_3 must be within δ of the median coordinate/line.
- *Note 2:* In 1D, p_3 must be the rightmost point in S_1 before m and q_3 the leftmost point in S_2 after m.
- *Note 3:* By the definition of δ, only one point of S_1 can exist in the range $[m - \delta, m]$. Same holds for S_2, with the range $[m, m + \delta]$.
- In high dimensions: Note 1 holds, Note 2 doesn't, Note 3 doesn't.
- In high dimensions: There is a sparse structure in the 2δ band.

Implementation of 1D-Closest-Pair

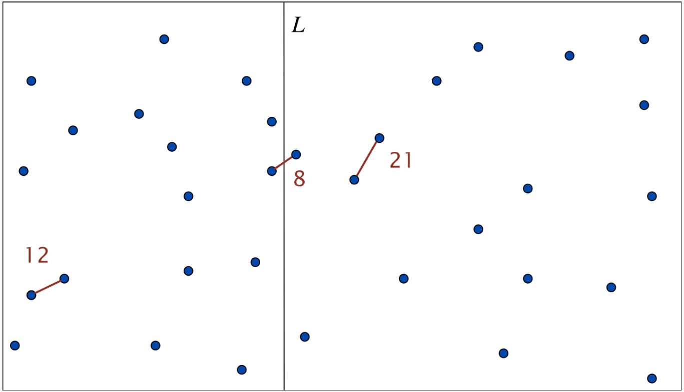
Input: List S of 1D points

Output: The closest pair of points in S and the distance between them.

if $\vert S\vert = 1$ **then**	if only single point
return $(), \delta = \infty$	then no closest pair
if $\vert S\vert = 2$ **then**	if only two points
return $(p_1, p_2), \delta = \vert p_1 - p_2\vert$	then they are the closest pair
Let m **be the median of** S**,**	$\Theta(n)$ time
S_l **be points** $< m$ **and** S_r **be points** $> m$	$\Theta(n)$ time
$(l_1, l_2), \delta_l$ = *1D-Closest-Pair*(S_l)	$T(n/2)$ time; recursive call
$(r_1, r_2), \delta_r$ = *1D-Closest-Pair*(S_r)	$T(n/2)$ time; recursive call
$(l_3, r_3), \delta_c$ = **closest pair;** $l_3 \in S_l$**,** $r_3 \in S_r$	$\Theta(n)$ time since we know from *Note 2*
	that l_3 is largest in S_l and r_3 is smallest in S_r
return pair with $\delta = \min(\delta_l, \delta_r, \delta_c)$	

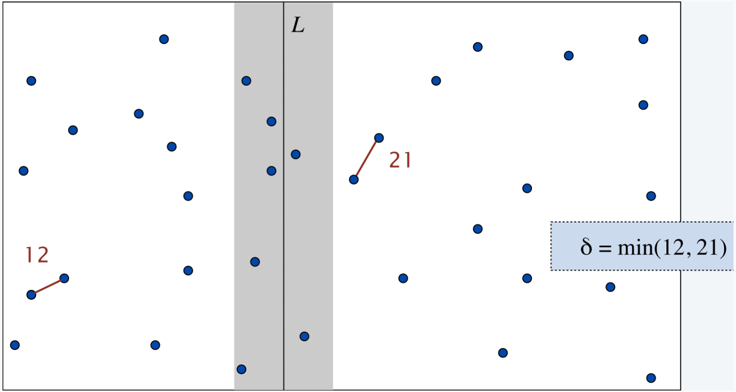
Adapting 1D Algorithm to the 2D case

- Divide all points into two halves using a vertical line L.
- Recursively solve for closest pair on left and right sides of L.
- Find closest pair with one point on each side of L. $[O(n^2)?]$
- Return best solution.



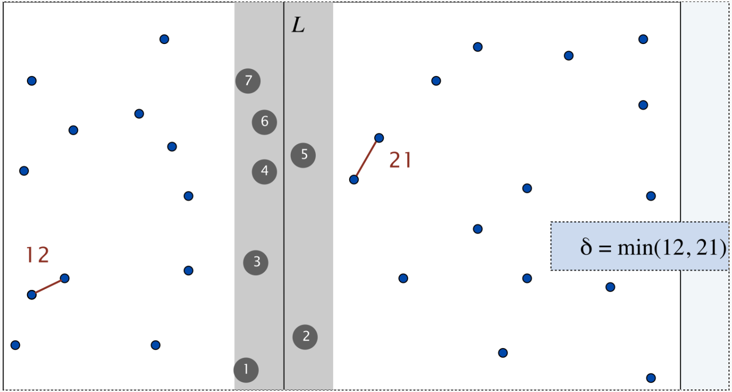
Find closest pair with one point on each side

- Via *Note 1*, it suffices to look at a 2δ band around line L.
- Sort the points in this band by their y coordinates.
- *Sparsity Claim:* For every point in this band, we only need to check distance to points within 7 positions in sorted order.



Find closest pair with one point on each side

- Via *Note 1*, it suffices to look at a 2δ band around line L.
- Sort the points in this band by their y coordinates.
- *Sparsity Claim:* For every point in this band, we only need to check distance to points within 7 positions in sorted order.



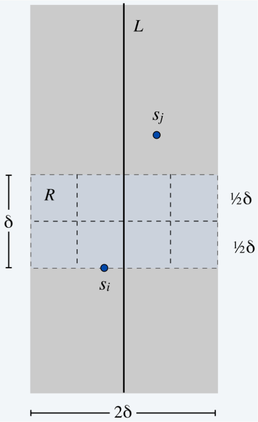
Proving the Sparsity Claim

Definition: Let s_i be the point with the i^{th} smallest y-coordinate.

Claim: If $|j - i| > 7$, the distance between s_i and s_j is at least δ.

Proof:

- Consider the 2δ-by-δ rectangle R in the band whose min y-coordinate is y-coordinate of s_i.
- Distance from s_i to any s_j above R is $\geq \delta$.
- Subdivide R into 8 squares each of side $\delta/2$. The diagonals will have length $\delta/\sqrt{2}$.
- There can be at most 1 point per square.
- At most 7 other points can be in R.



Implementation of 2D-Closest-Pair(S)

Input: List S of 2D points

Output: The closest pair of points in S and the distance between them.

if $|S| = 1$ **then** — if only single point
    return $(), \delta = \infty$ — then no closest pair
if $|S| = 2$ **then** — if only two points
    return $(p_1, p_2), \delta = |p_1 - p_2|$ — then they are the closest pair
Find “median” line L in x-coordinates — ?? time
split S into $S_l < L, S_r > L$
$(l_1, l_2), \delta_l$ = *2D-Closest-Pair*(S_l) — $T(n/2)$ time; recursive call
$(r_1, r_2), \delta_r$ = *2D-Closest-Pair*(S_r) — $T(n/2)$ time; recursive call
$\delta = \min(\delta_l, \delta_r)$
find 2δ band around L, sort by y-coordinate
Find closest crossing pair — $O(n)$ time since we only compare each point to $<= 7$ points
return closest pair found until now

Refining Closest Pair to $O(n \log n)$

- Note that we need $O(n \log n)$ time in lines 3 and 8 for sorting points, first by their x-coordinates and then by y-coordinates.
- This will cause the overall running time to be $O(n \log^2 n)$. (Verify!)
- Can we avoid this?

- Yes! Remember Mergesort?
- We could have the recursive calls return two sorted lists, one sorted by x-coordinate and the other sorted by y-coordinate.
- We could then merge these lists using the Merge part of Mergesort.
- Now the dominant time outside recursive calls is $O(n)$ and the overall time complexity would be $O(n \log n)$. (Verify!)

5. Summary

Summary

- General structure of Divide and Conquer
  - Break problem into pieces (usually equally sized)
  - Solve each piece (pieces can be ”solved” by being discarded as in binary search, sometimes called Decrease-and-Conquer)
  - Combine the solutions to get the overall solution
- Lots of cleverness combining (Closest Pair) and/or in breaking into subproblems (we’ll see in Selection next time).
- Set up and solve recurrence to get complexity.