Today’s plan

We will discuss divide and conquer with two problems

1. Inversion counting

2. Closest pair (IA1)

Divide and Conquer

- General approach:
  - Break problem into smaller sub-problems
  - Solve smaller sub-problems via recursion
  - Combine solutions of sub-problems to get a solution to the original problem
- A special case of recursion
- Reduce a given problem to multiple smaller instances of the original problem
  - Typically constant factor smaller (n -> n/b)

Problem: Inversion counting

- Input: an array A containing numbers 1,2,…,n in arbitrary order
- Output: number of inversions, i.e., the number of pairs (i, j) such that $i < j$ and $A[i] > A[j]$

Example

- Input $(1, 4, 2, 5, 3)$

Why study this problem?

- Consider a set of n movies

You and I can both rank them according to how much we like them

Mine: 1, 2, 3, 4, …n

Yours: 5, 4, 1, …

Measure the difference between two ranked lists --- a fundamental operation behind all the recommender systems (collaborative filtering)

High level idea

- Brute force counting

- Divide and conquer
  - Break the array into two parts
  - Count the left part
  - Count the right part
  - Count in the inversions cross the two parts

High level algorithm

```
Count(A, n)
    if n=1  return 0
    else
        x=Count(A_l, n/2)
        y=Count(A_r, n/2)
        z=CountCross(A_l, A_r)
    return x+y+z
```

countcross(A_l, A_r) needs to run in $O(n)$ in order to achieve an overall runtime of $O(n \log n)$

What if A_l and A_r are already sorted?

Can we count the cross inversions in $O(n)$ time?

$$A_l = (2,4,5) \qquad A_r = (1,3,6)$$

Building on Merge_sort

Sort-and-Count(A, n)

if n=1 return 0

else

(L_s, x)=Sort-and-Count$\left(A_l, \frac{n}{2}\right)$

(R_s, y)=Sort-and-Count$\left(A_r, \frac{n}{2}\right)$

(A_s, z)=Merge-and-CountCross(L_s, R_s)

return $(A_s, x + y + z)$

Pseudo code for merge

L: left-half sorted array (size n)

R: right-half sorted array (size m)

A: output sorted merged array

$i = 0;\ j = 0$

while $i < n$ and $j < m$

  if $L(i) \leq R(j)$

    $A(k) = L(i);\ \ i++$

  else

    $A(k) = R(j);\ \ j++$

  end if

End while

(Ignore the end case)

What would happen if there is no inversion between L and R?

How many inversions can we be sure of when the else branch is taken?

Example

- Consider merging $(1, 3, 5)$ and $(2, 4, 6)$

Pseudo code for Merge-and-CountCross

L: left-half sorted array (size n)

R: right-half sorted array (size m)

A: output sorted merged array

z: the # of cross inversions

$i = 0;\ j = 0; z = 0$

While $i < n$ and $j < m$

  if $L(i) \leq R(j)$

    $A(k) = L(i);\ \ i++$

  else

    $A(k) = R(j);\ j++$

    $z = z + (n - i)$

  end if

End while

(Ignore the end case)

Run time of subroutine:

$$O(n)$$

Run time of the overall algorithm:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Same as merge_sort:

$$O(n\log n)$$

2^{nd} Problem: Closest Pair of Points

Problem Definition:

Given n points in 2-d plane, find a pair or pairs with the smallest Euclidean distance between them.

Our goal:

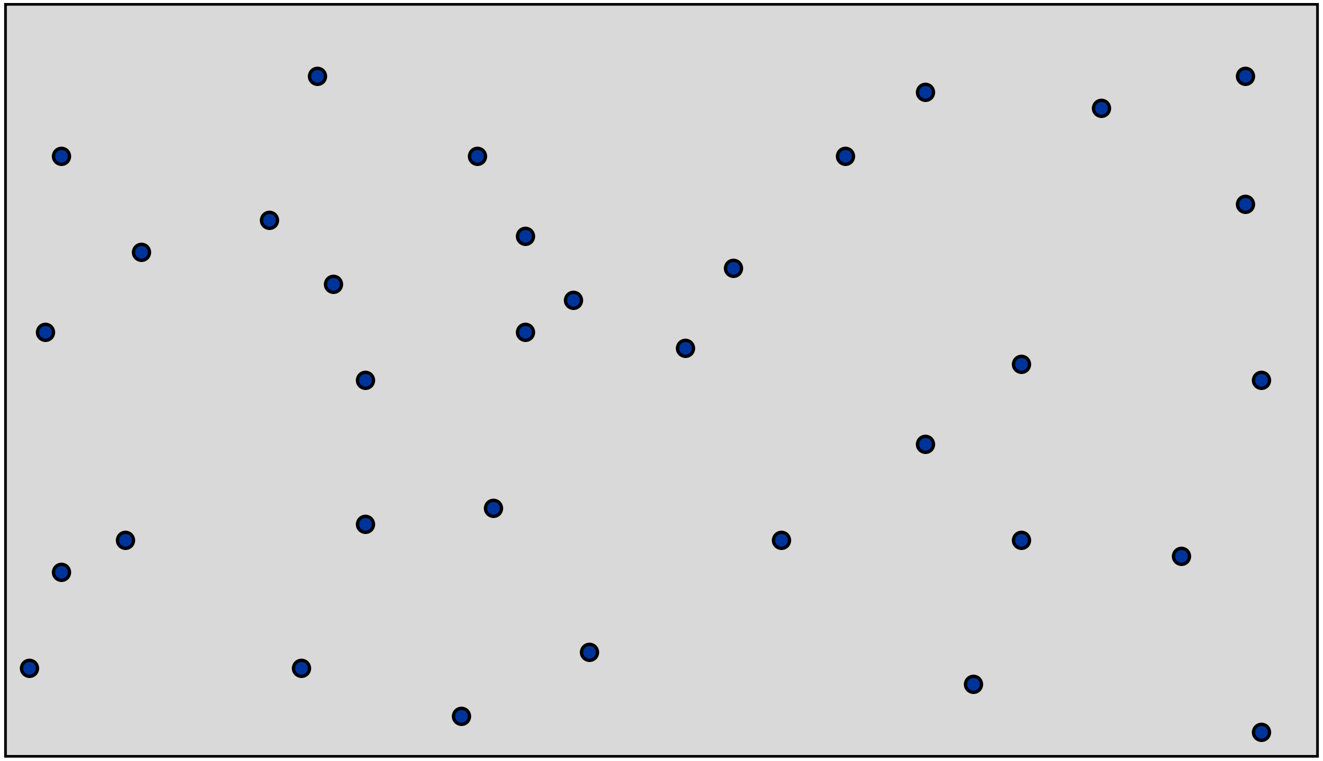
Design a divide and conquer algorithm that runs in $O(n \log n)$

Preliminary thoughts:

- Brute force – directly compute the distances between all pairs and find and record the pairs with the smallest distance
- If 1-d, we have an easy $n \log n$ algorithm by sorting the points and scan the distances between neighboring points.

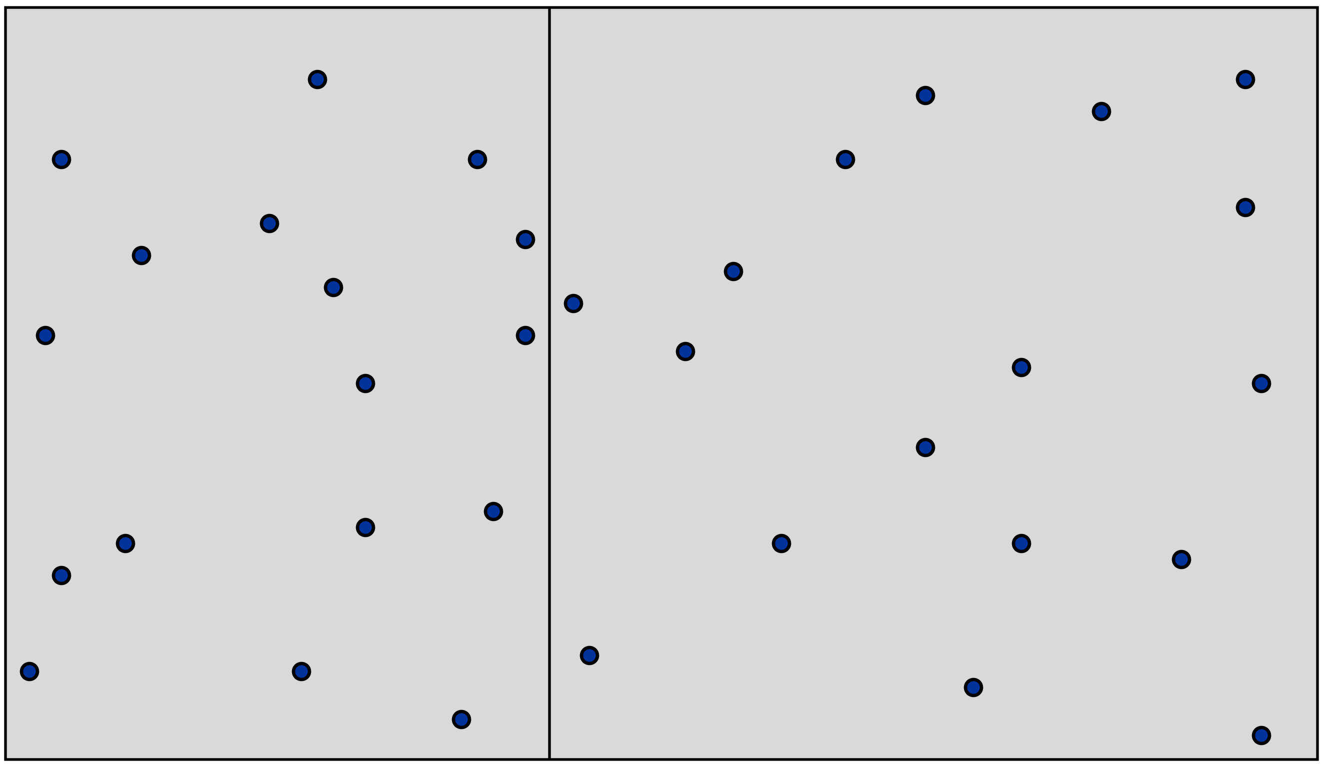
Let’s try to design a divide and conquer algorithm

- How to divide?



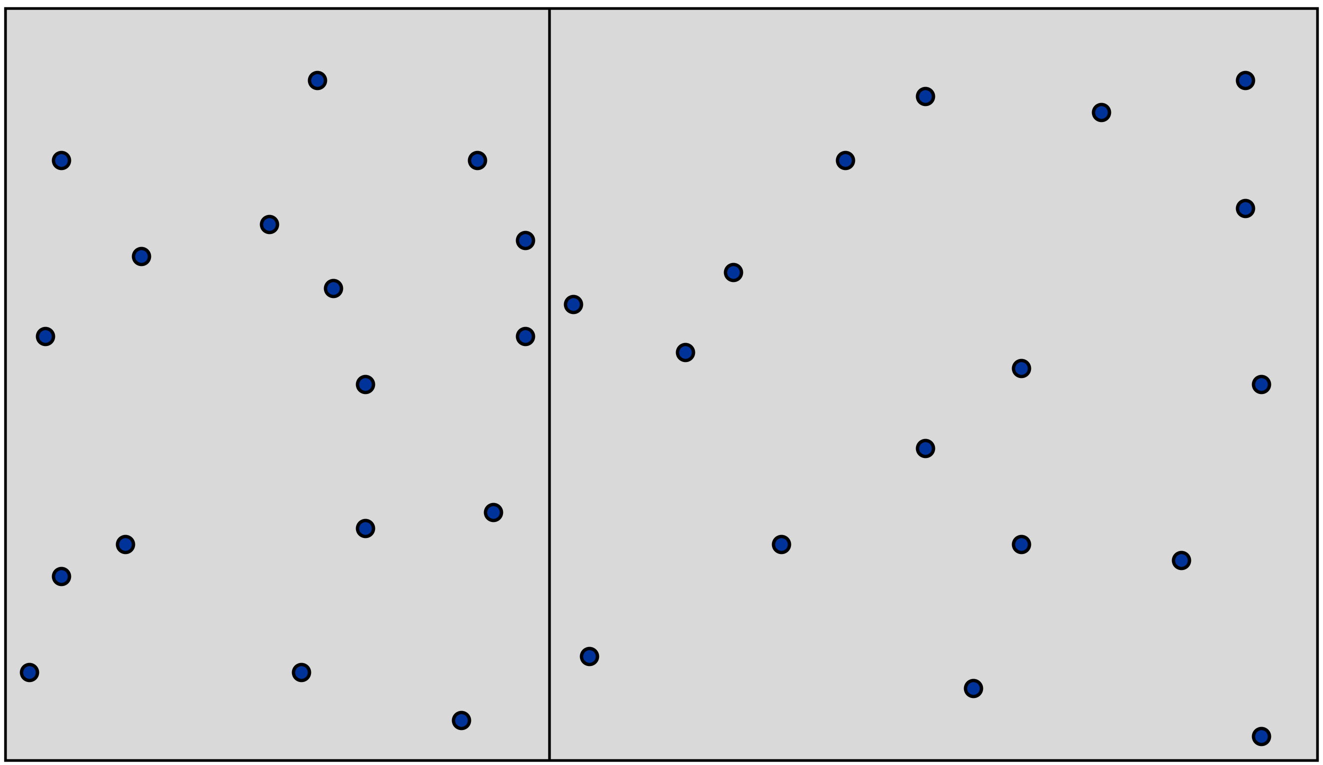
High level idea

Divide: draw vertical line s.t. ~n/2 points on each side.
Conquer: find closest pair in each side recursively.
Combine: find closest pair with one point in each side.
Return best of 3 solutions.



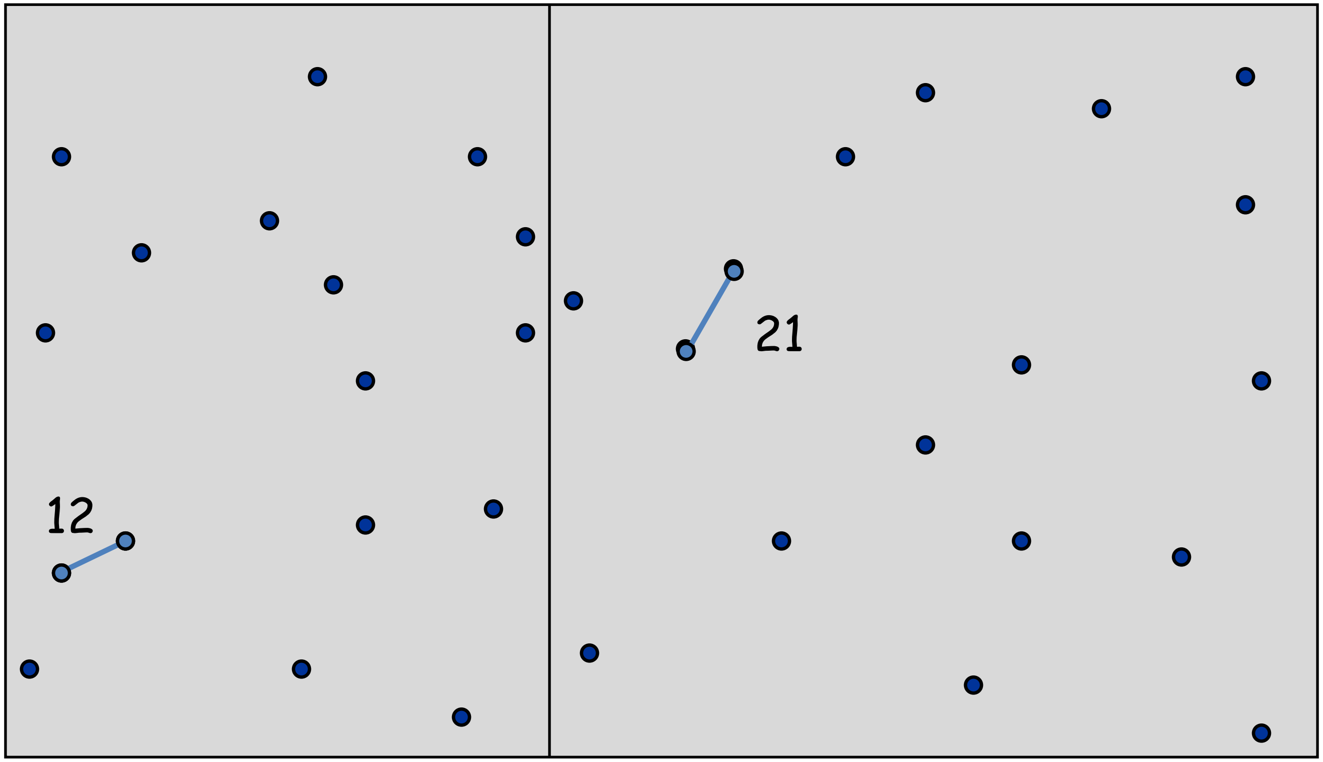
Finding the closest pair across

- Naïve approach: consider every pair across

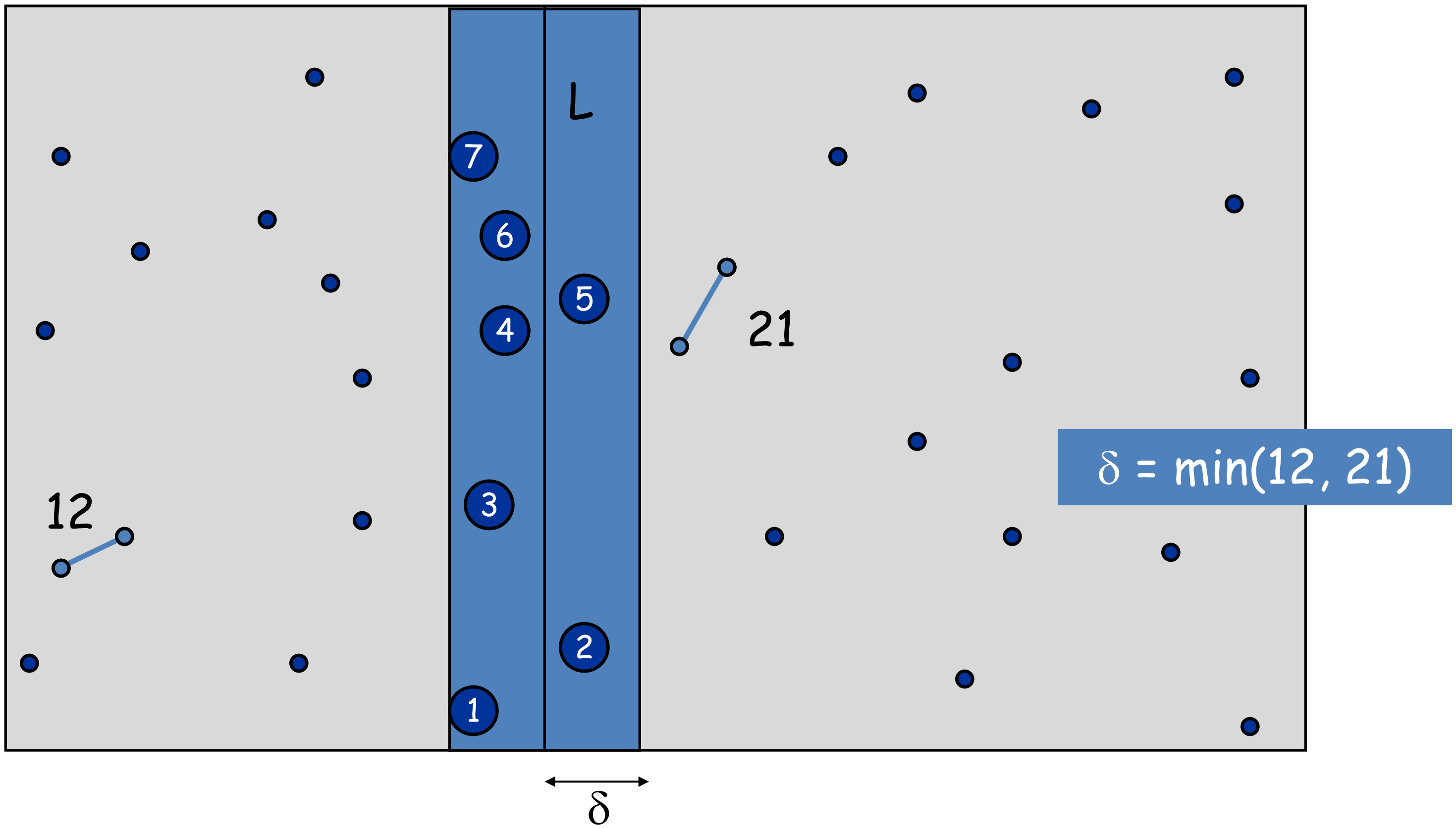


Getting smarter

- If the shortest so far is δ, then no need to consider pairs more than δ apart
- No need to consider points δ away from the median line



Focus on the middle strip



- Issue: in the worst case $O(n)$ points will be in the middle strip
- We need more pruning!!!
  - Again, we do not need consider anything more than δ apart (use the y axis to prune)

Scan the middle strip for cross pairs
Closest-cross-pairs(M_y,δ)

$M_y = \{p_1, p_2, \dots, p_m\}$: the list of points in the middle strip sorted in increasing order of y

1. $d_m = \delta$
2. for $i = 1$ to $m - 1$
3. $j = i + 1$
4.   while $p_{j(y)} - p_{i(y)} \leq \delta$ and $j \leq m$
5. $d = D(p_i, p_j)$
6. $d_m = \min\{d, d_m\}$
7. $j = j + 1$
8.   end while
9. end for
10. Return d_m

Claim:
For any pair (i, j), if $D(i, j) \leq \delta$, we must have $p_{j(y)} - p_{i(y)} \leq \delta$, thus it will be compared with line 5,6!

But this has double loop? Could it be linear?

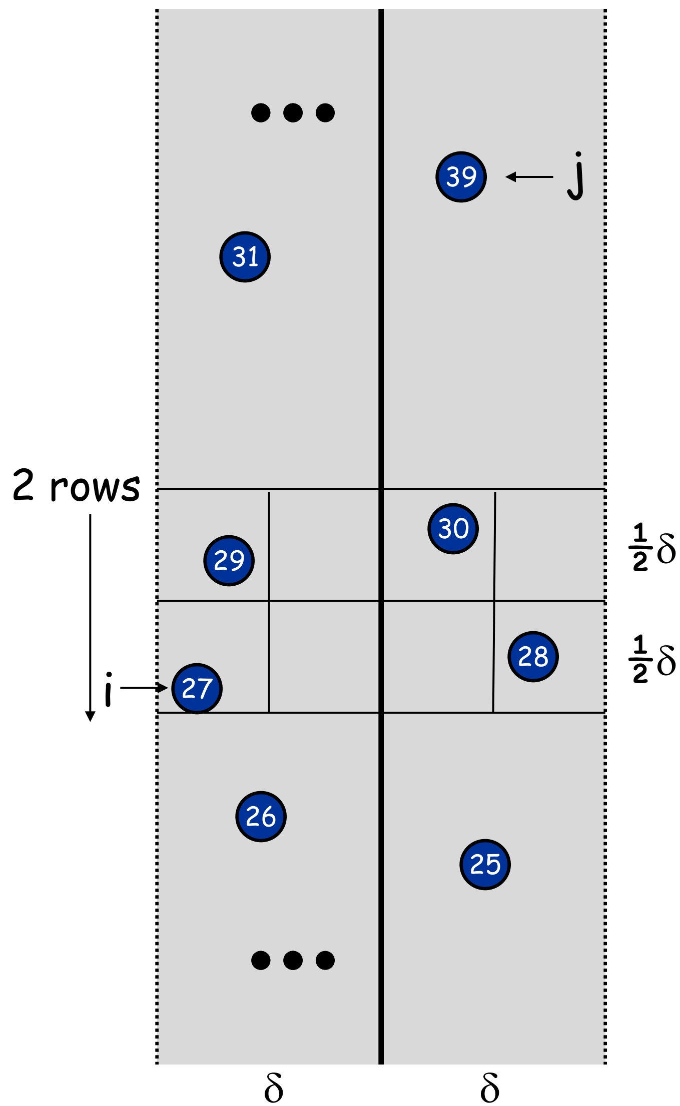
Claim. For any point p_i, the while loop will execute at most 7 times

Proof:

Consider all points above p_i that have y value within δ of $p_i(y)$

Together with p_i they must lie in side the rectangle of height δ and width 2δ

The rectangle can be divided into 8 cells. There are at most 1 point inside each cell.



Closest Pair Algorithm
(this does not keep track of all the pairs, needs to be added)

Closest-Pair(p_1, …, p_n) {
1. if $n \leq 3$
2.   compute and return the min distance
3. else
4.   Compute separation line L — $O(n \log n)$
5. δ_1 = Closest-Pair(left half) — $2T(n / 2)$
6. δ_2 = Closest-Pair(right half)
7. δ = min(δ_1, δ_2)

8.   Identify all points within δ from L — $O(n)$
9.   Sort them by y-coordinate into M_y — $O(n \log n)$
10. d_m = closest-cross-pair (M_y, δ) — $O(n)$

11.   return d_m.
}

Enhanced Divide and Conquer

- Pre-sort all points based on x and y coordinates respectively
- For Line 8-9, scan the master list pre-sorted based on y to create M_y by excluding points δ away from L in x-coordinate