

More Divide and Conquer: Quicksort and Closest Pair of Points

CS 4102: Algorithms

Spring 2021

Mark Floryan and Tom Horton

Quicksort and Partition

Readings: CLRS Chapter 7 (not 7.4.2)

Quicksort: Introduction

- ▶ Developed by C.A.R. (Tony) Hoare (a Turing Award winner)
http://www.wikipedia.org/wiki/C._A._R._Hoare
 - ▶ Published in 1962
- ▶ Classic divide and conquer, but...
 - ▶ Mergesort does no comparisons to divide, but a lot to combine results (i.e. the merge) at each step
 - ▶ Quicksort does a lot of work to divide, but has nothing to do after the recursive calls. No work to combine.
 - ▶ If we're using arrays. Linked lists? Interesting to think about this!
- ▶ Dividing done with algorithm often called ***partition***
 - ▶ Sometimes called *split*. Several variations.

Quicksort's Strategy

- ▶ Called on subsection of array from *first* to *last*
 - ▶ Like mergesort
- ▶ First, choose some element in the array to be the ***pivot*** element
 - ▶ Any element! Doesn't matter for correctness.
 - ▶ Often the first item. For us, the last. Or, we often move some element into the last position (to get better efficiency)
- ▶ Second, call ***partition***, which does two things:
 - ▶ Puts the pivot in its proper place, i.e. where it will be in the correctly sorted sequence
 - ▶ All elements below the pivot are less-than the pivot, and all elements above the pivot are greater-than
- ▶ Third, use quicksort recursively on both sub-lists

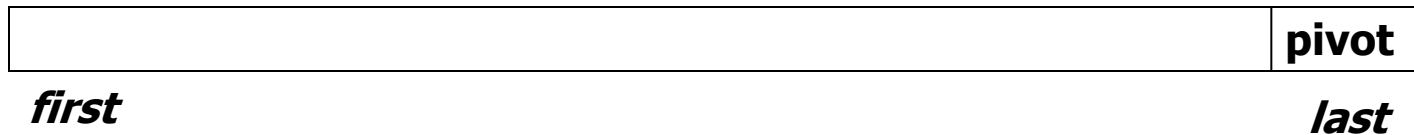
Quicksort is Divide and Conquer

- ▶ **Divide:** select **pivot** element p , **Partition**(p)
- ▶ **Conquer:** recursively sort left and right sublists
- ▶ **Combine:** Nothing!

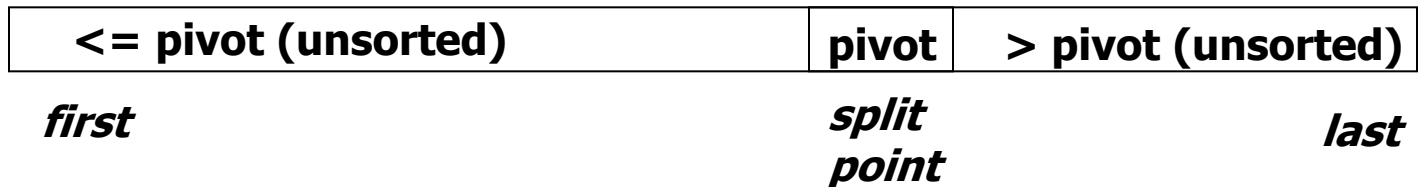
Contrast to mergesort,
where divide is simple and combine is work

Quicksort's Strategy (a picture)

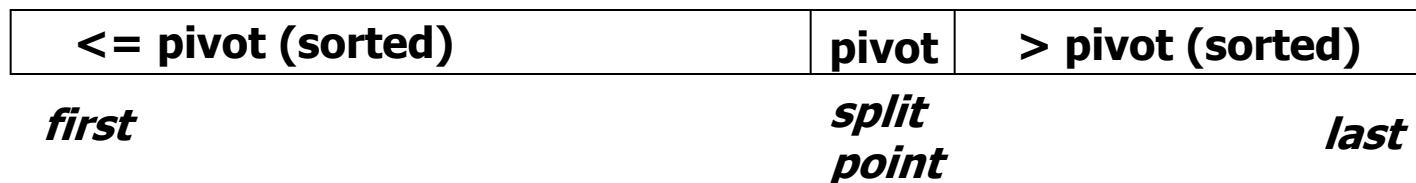
- ▶ Use last element as pivot (or pick one and move it there)



- ▶ After call to partition...



- ▶ Now sort two parts recursively and we're done!



- ▶ Note that splitPoint may be anywhere in *first..last*
- ▶ Note our assumption that all keys are distinct

Quicksort Code

Input Parameters: *list, first, last*

Output Parameters: *list*

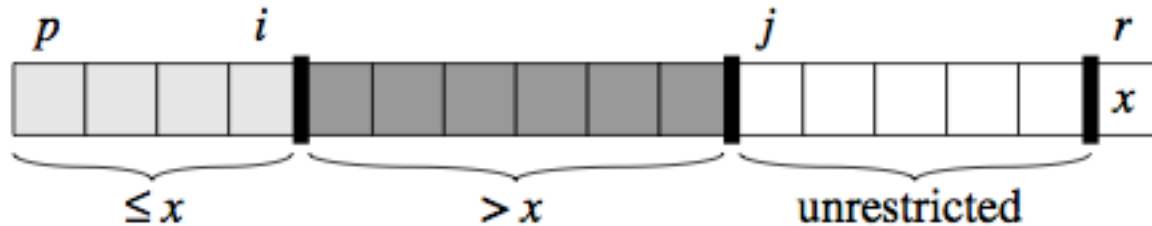
```
def quicksort(list, first, last):  
    if first < last:  
        q = partition(list, first, last)  
        quicksort(list, first, q-1)  
        quicksort(list, q+1, last)  
    return
```

Partition Does the Dirty Work

- ▶ Partition rearranges elements
 - ▶ How? How many comparisons? How many swaps?
- ▶ How? Two well-known algorithms
 - ▶ In this chapter of CLRS, Lomuto's algorithm
 - ▶ In the exercises, the original: Hoare's algorithm.
(Page 185. Look at on your own.)
 - ▶ Important:
 - ▶ Both are in-place!
 - ▶ Both are linear.

Strategy for Lomuto's Partition

- Invariant: At any point:
 - i indexes the right-most element $\leq \text{pivot}$
 - $j-1$ indexes the right-most element $> \text{pivot}$



- Strategy:
 - Look at next item $a[j]$
 - If that item $> \text{pivot}$, all is well!
 - If that item $< \text{pivot}$, increment i and then swap items at positions i and j
 - When done, swap pivot with item at position $i+1$
- Number of comparisons: $n-1$

Lomuto's Partition: Code

Input Parameters: *list, first, last*

Output Parameters: *list*.

Return value: the split point

```
def partition(list, first, last):  
    pval = list[last]  
    i = first-1  
    for j in range(first, last): # first up to before last  
        if list[j] <= pval:  
            i = i+1  
            (list[i], list[j]) = (list[j], list[i]) # swap!  
    (list[last], list[i+1]) = (list[i+1], list[last]) # swap!  
    return i+1
```

Efficiency of Quicksort

- ▶ Partition divides into two sub-lists, perhaps unequal size
 - ▶ Depends on value of pivot element
- ▶ Recurrence for Quicksort
$$T(n) = \text{partition-cost} + T(\text{size of 1st section}) + T(\text{size of 2nd section})$$
- ▶ If divides equally, $T(n) = 2 T(n/2) + n-1$
 - ▶ Just like mergesort
 - ▶ Solve by substitution or master theorem
$$T(n) \in \Theta(n \lg n)$$
- ▶ This is the best-case. But...

Worst Case of Quicksort

- ▶ What if divides in most unequal fashion possible?
 - ▶ One subsection has size 0, other has size $n-1$
 - ▶ $T(n) = T(0) + T(n-1) + n-1$
 - ▶ What if this happens every time we call partition recursively?

$$W(n) = \sum_{k=2}^n (k-1) \in \Theta(n^2)$$

- ▶ Uh oh. Same as insertion sort.
 - ▶ “Sorry Prof. Hoare – we have to take back that Turing Award now!”

Quicksort's Average Case

- ▶ Good if it divides equally, bad if most unequal.
 - ▶ Remember: when subproblems size 0 and $n-1$
 - ▶ Can worst-case happen?
Sure! Many cases. One is when elements already sorted. Last element is max, pivot around that. Next pivot is 2nd max...
- ▶ What's the average?
 - ▶ Much closer to the best case
 - ▶ A bad-split then a good-split is closer to best-case (pp. 176-178)
 - ▶ To prove $A(n)$, fun with recurrences!
 - ▶ The result: If all permutations are equal, then
$$A(n) \cong 1.386 n \lg n \text{ (for large } n)$$
- ▶ So very fast on average.
- ▶ And, we can take simple steps to avoid the worst case!

Avoiding Quicksort's Worst Case

- ▶ Make sure we don't pivot around max or min
 - ▶ Find a better choice and swap it with last element
 - ▶ Then partition as before
- ▶ Recall we get best case if divides equally
 - ▶ Could find median. But this costs $\Theta(n)$. Instead...
 - ▶ Choose a **random element** between first and last and swap it with the last element
 - ▶ Or, estimate the median by using the “median-of-three” method
 - ▶ Pick 3 elements (say, first, middle and last)
 - ▶ Choose median of these and swap with last. (Cost?)
 - ▶ If sorted, then this chooses real median. Best case!

Tuning Quicksort's Performance

- ▶ In practice quicksort runs fast
 - ▶ $A(n)$ is log-linear, and the “constants” are smaller than mergesort and heapsort
 - ▶ Often used in software libraries
 - ▶ So worth tuning it to squeeze the most out of it
 - ▶ Always do something to avoid worst-case
- ▶ Sort small sub-lists with (say) insertion sort
 - ▶ For small inputs, insertion sort is fine
 - ▶ No recursion, function calls
 - ▶ Variation: don't sort small sections at all.
After quicksort is done, sort entire array with **insertion sort**
 - ▶ It's efficient on almost-sorted arrays!

Quicksort's Space Complexity

- ▶ Looks like it's in-place, but there's a *recursion stack*
 - ▶ Depends on your definition: some people define *in-place* to not include stack space used by recursion
 - ▶ E.g. our CLRS algorithms textbook
 - ▶ Other books and people do “count” this
 - ▶ How much goes on the stack?
 - ▶ If most uneven splits, then $\Theta(n)$.
 - ▶ If splits evenly every time, then $\Theta(\lg n)$.
- ▶ Ways to reduce stack-space used due to recursion
 - ▶ Various books cover the details (not ours, though)
 - ▶ First, remove 2nd recursive call (tail-recursion)
 - ▶ Second, always do recursive call on smaller section

Summary: Quicksort

- ▶ Divide and conquer where divide does the heavy-lifting
- ▶ In worst-case, efficiency is $\Theta(n^2)$
 - ▶ But it's practical to avoid the worst-case
- ▶ On average, efficiency is $\Theta(n \lg n)$
- ▶ Better space-complexity than mergesort.
- ▶ In practice, runs fast and widely used
 - ▶ Many ways to tune its performance
- ▶ Various strategies for Partition
 - ▶ Some work better if duplicate keys
- ▶ More details? See Sedgewick's algorithms textbook
 - ▶ He's the expert! PhD on this under Donald Knuth

Closest Pair of Points

Readings: CLRS Chapter 7 (not 7.4.2)

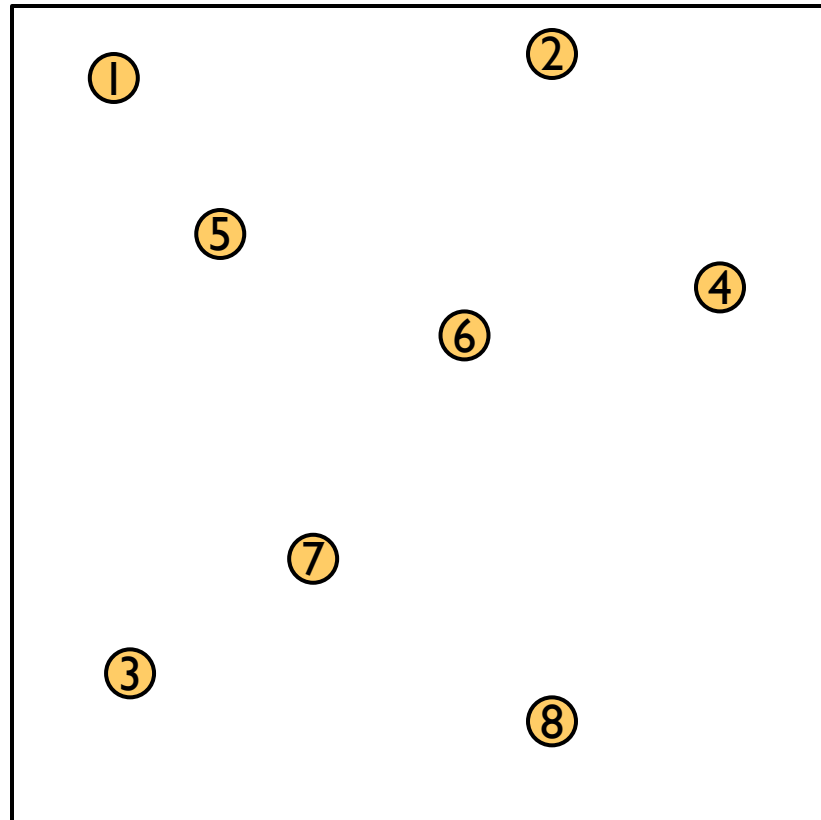
Closest Pair of Points in 2D Space

Given:

A list of points

Return:

Distance of the pair of points that are closest together
(or possibly the pair too)



Closest Pair of Points: Naïve

Given:

A list of points

Return:

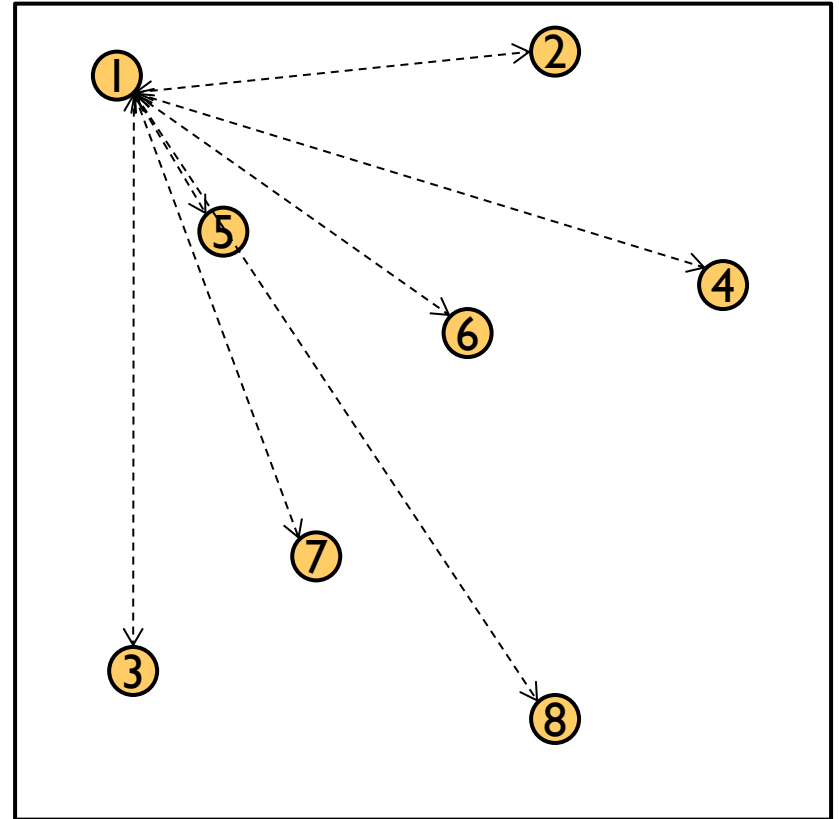
Distance of the closest pair of points

Naive Algorithm: $O(n^2)$

Test every pair of points,
return the closest.

We can do better!

$\Theta(n \log n)$



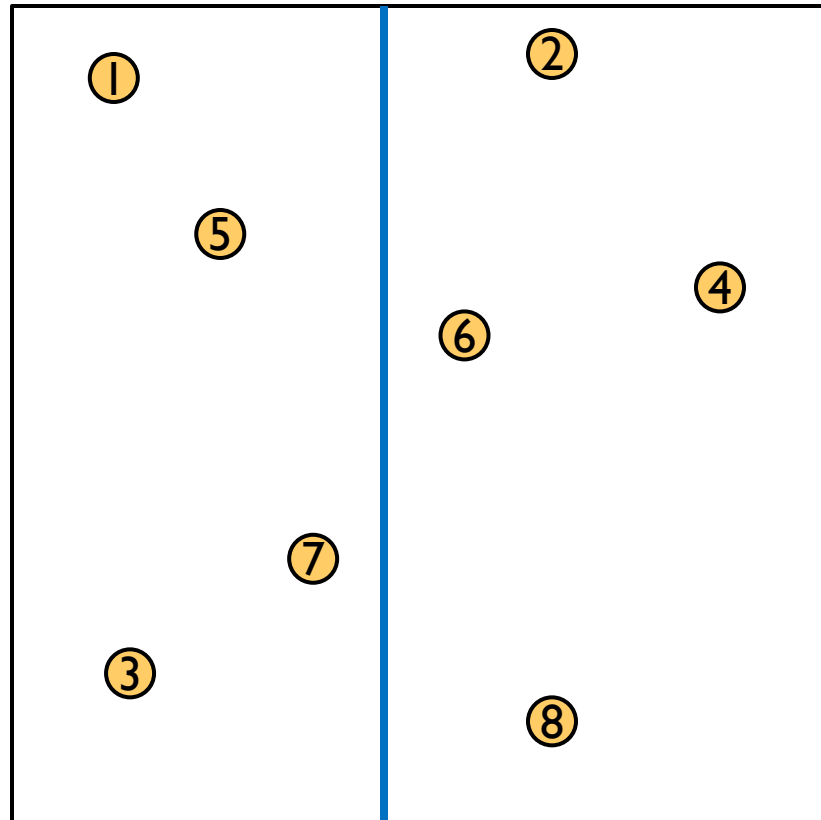
Closest Pair of Points: D&C

Divide: How?

At median x coordinate

Conquer:

Combine:



Closest Pair of Points: D&C

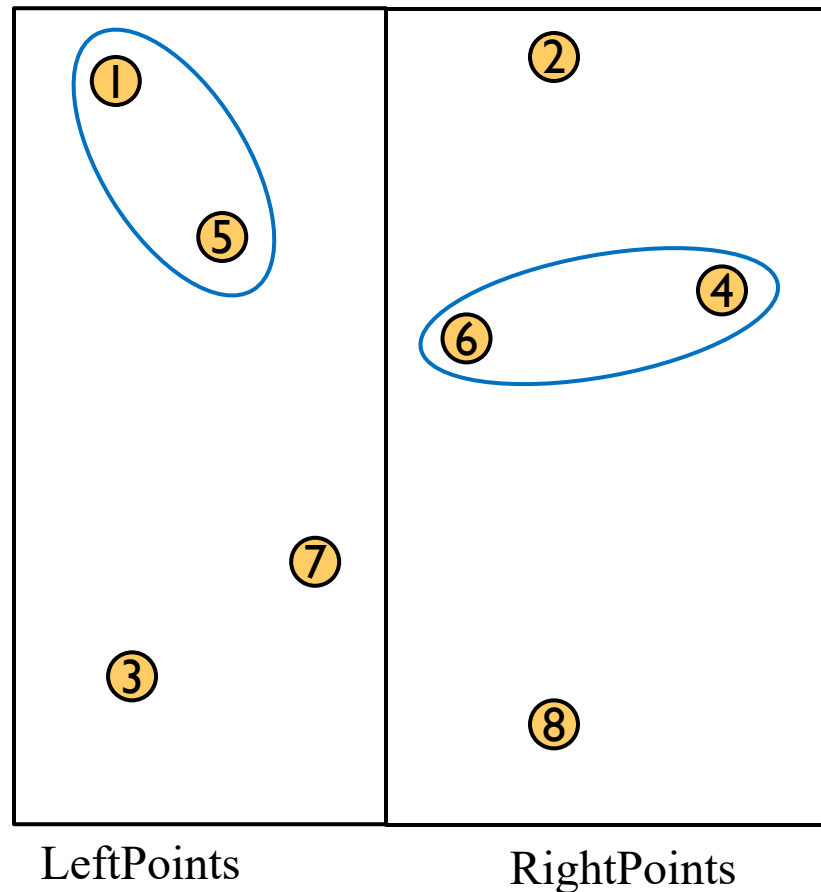
Divide:

At median x coordinate

Conquer:

Recursively find closest pairs from Left and Right

Combine:



Closest Pair of Points: D&C

Divide:

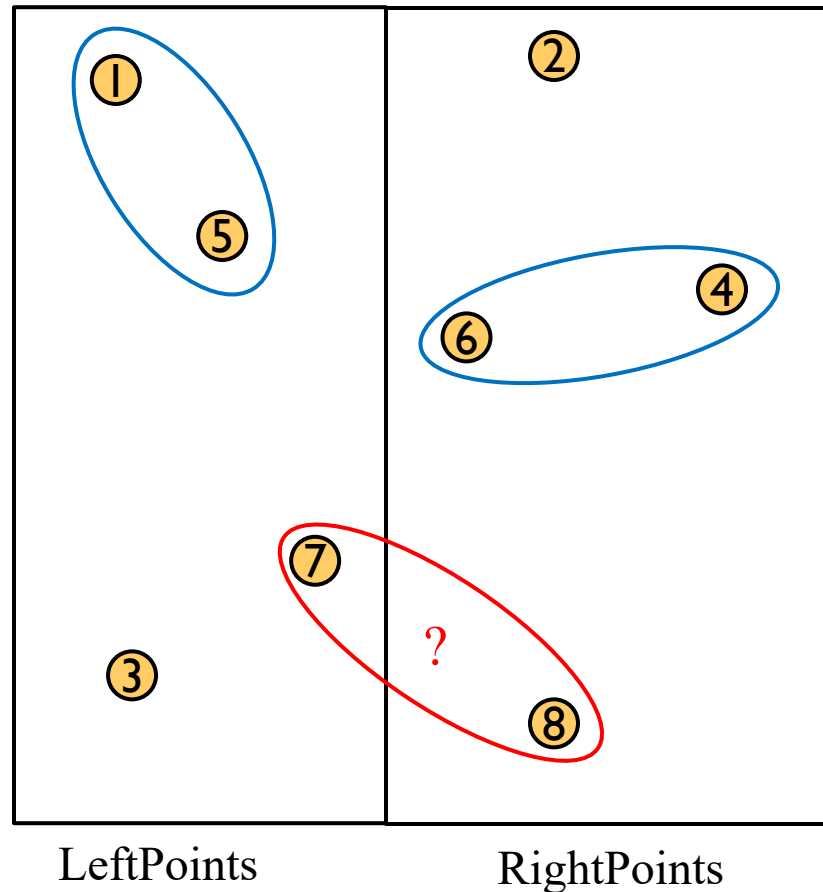
At median x coordinate

Conquer:

Recursively find closest pairs from Left and Right

Combine:

Return min of Left and Right pairs
Problem
?



Closest Pair of Points: D&C

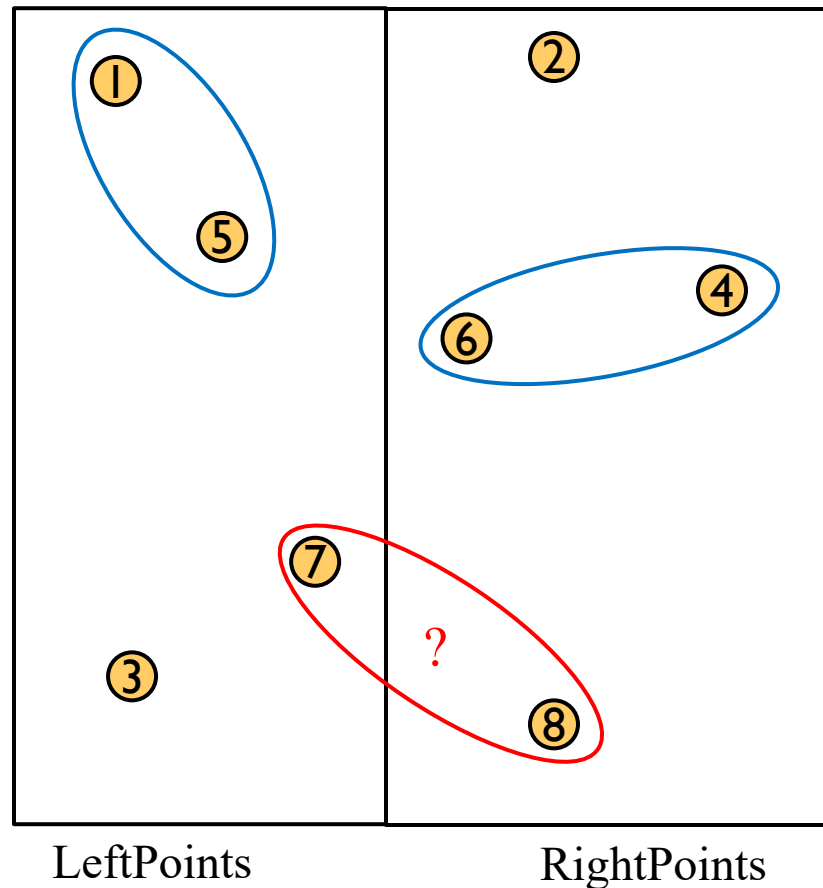
Combine:

2 Cases:

1. Closest Pair is completely in Left or Right

2. Closest Pair Spans our “Cut”

Need to test points across the cut



Spanning the Cut

Combine:

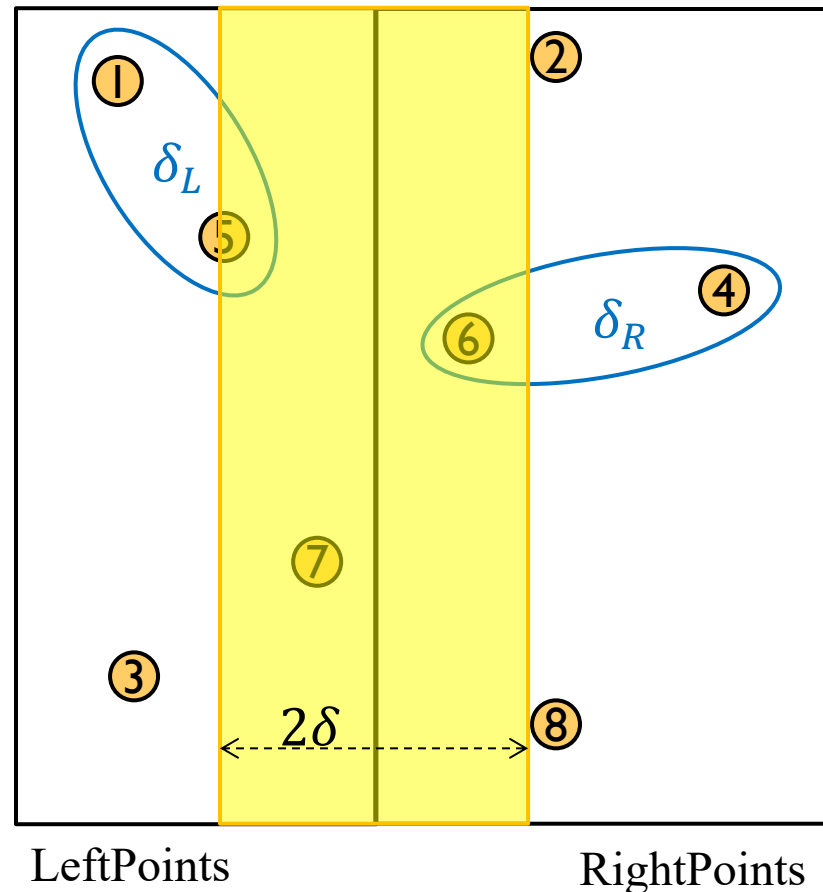
2. Closest Pair Spanned our “Cut”

Need to test points
across the cut.

Bad approach: Compare
all points within $\delta =$
 $\min\{\delta_L, \delta_R\}$ of the cut.

How many are there?

Define “runway” or
“strip” along the cut.



Spanning the Cut

Define “runway” or
“strip” along the cut.

Combine:

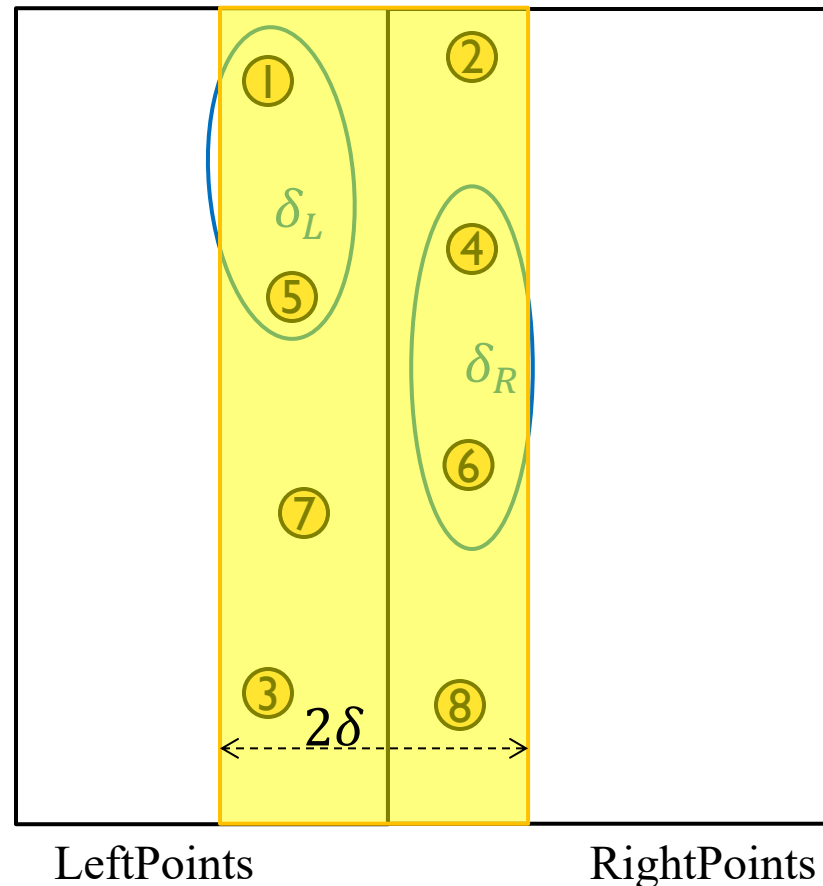
2. Closest Pair Spanned our “Cut”

Need to test points
across the cut

Bad approach: Compare
all points within $\delta =$
 $\min\{\delta_L, \delta_R\}$ of the cut.

How many are there?

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \left(\frac{n}{2}\right)^2 \\ &= \Theta(n^2) \end{aligned}$$



Spanning the Cut

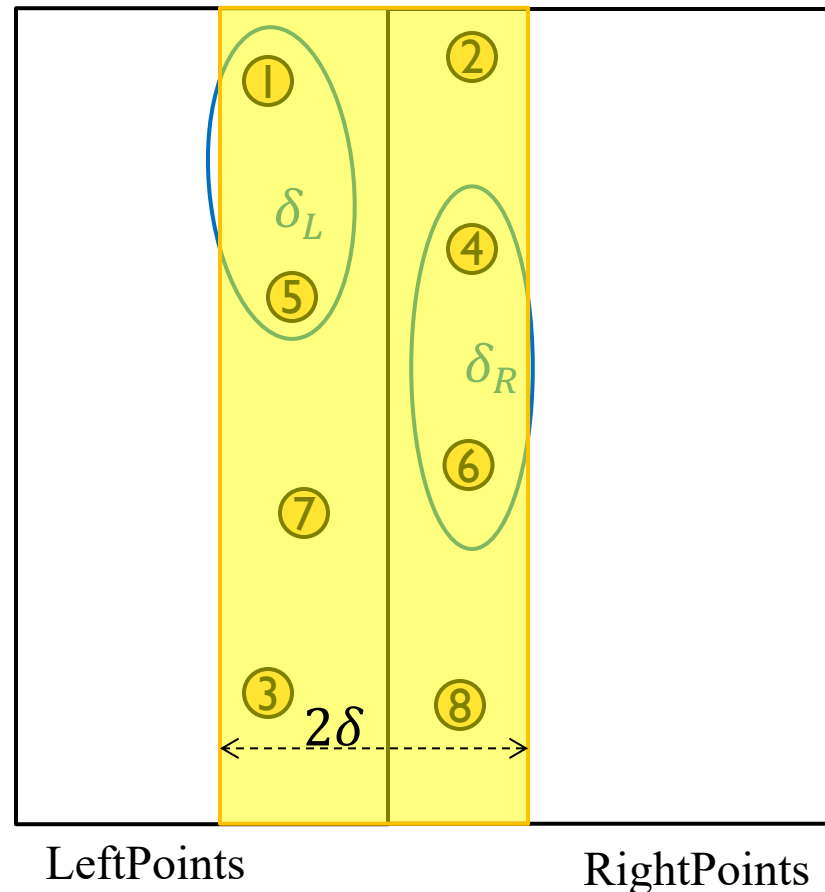
Combine:

2. Closest Pair Spanned our “Cut”

Need to test points
across the cut

We don't need to test all
pairs!

Don't need to test any
points that are $> \delta$ from
one another



Spanning the Cut

Combine:

2. Closest Pair Spanned our “Cut”

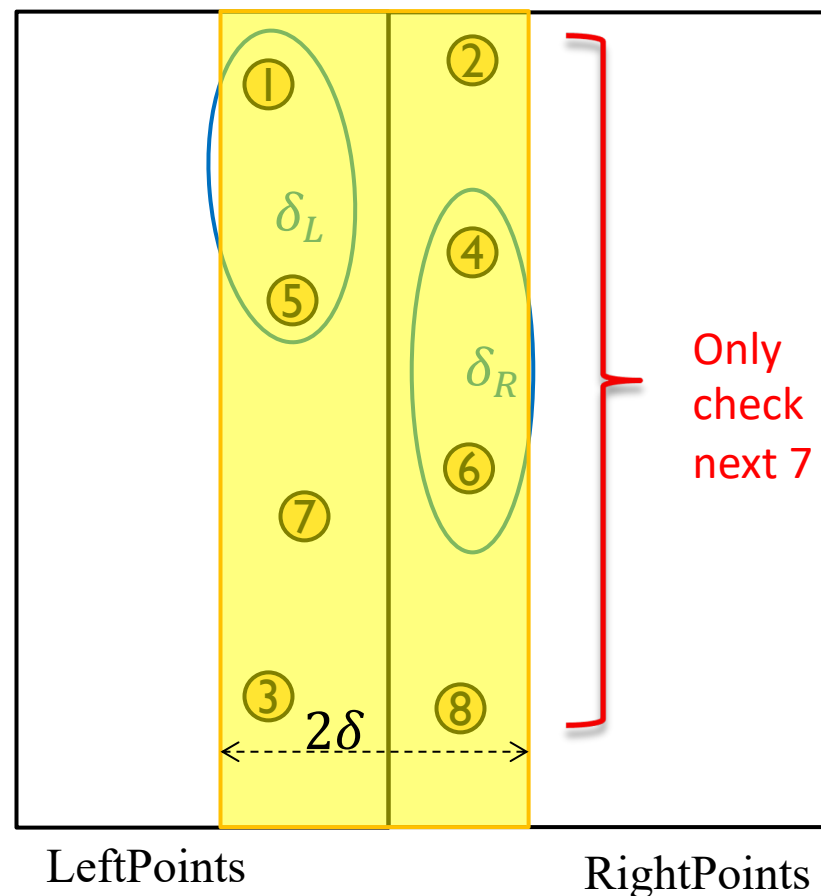
Consider points in strip in increasing y-order.

For a given point p , we can *prove* the 8th point and beyond is more than δ from p .

(pp. 1041-2 in CLRS)

So for each point in strip, check next 7 points in y-order.

$\Theta(n)$ **Better!**



Closest Pair of Points: Divide and Conquer

Initialization: Sort points by x -coordinate
(Later we'll also need to process points by y -coordinate, too.)

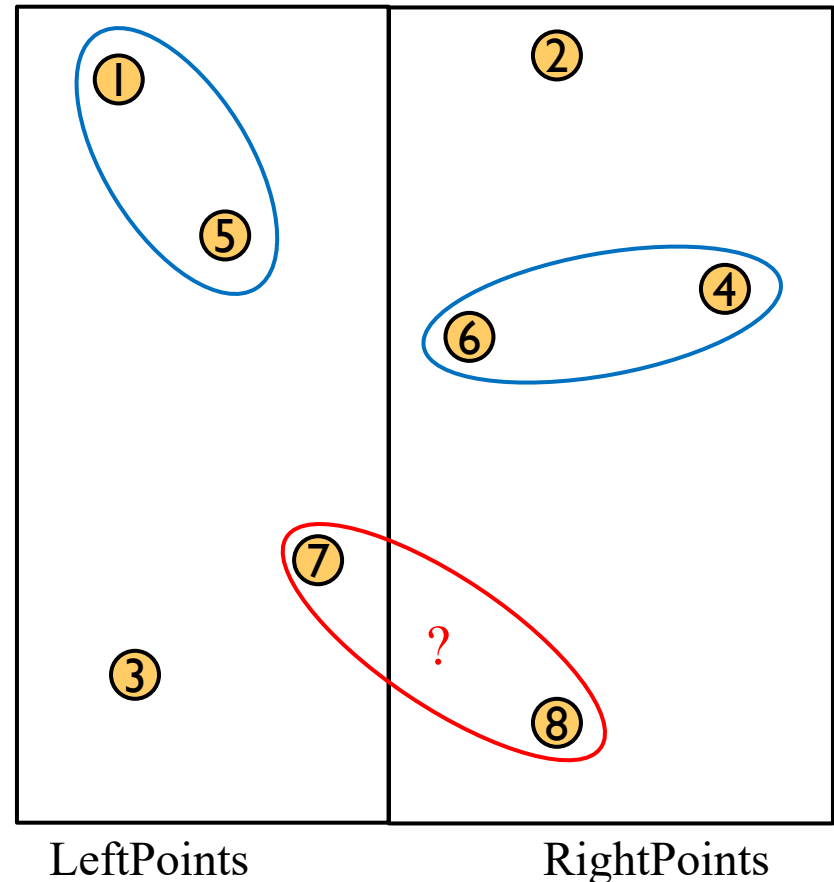
Divide: Partition points into two lists of points based on x -coordinate (split at the median x)

Conquer: Recursively compute the closest pair of points in each list

Base case?

Combine:

- Consider only points in the runway (x -coordinate within distance δ of median)
- Process runway points by y -coordinate
- Compare each point in runway to 7 points above it and save the closest pair
- Output closest pair among **left**, **right**, and **runway** points



Closest Pair of Points: Divide and Conquer

What is the running time?

$$\Theta(n \log n)$$

$$T(n)$$

$$T(n) = 2T(n/2) + \Theta(n)$$

Case 2 of Master's Theorem

$$T(n) = \Theta(n \log n)$$

$$\Theta(n \log n)$$

$$\Theta(1)$$

$$2T(n/2)$$

$$\Theta(n)$$

$$\Theta(1)$$

Initialization: Sort points by x -coordinate

Divide: Partition points into two lists of points based on x -coordinate (split at the median x)

Conquer: Recursively compute the closest pair of points in each list

Combine:

- Process runway points by y -coordinate and Compare each point in runway to 7 points above it and save the closest pair
- Output closest pair among **left**, **right**, and **runway** points

Summary for Closest Pair of Points

- ▶ Comparing all pairs is a brute-force fail
 - ▶ Except for small inputs
- ▶ Divide and conquer a big improvement
- ▶ Needed to find an efficient way for part of the combine step
 - ▶ Geometry came through for us here!
 - ▶ Only needed to look at constant number of points for each point in the strip
- ▶ Implementation subtleties
 - ▶ Don't want to sort the strip by y-coordinate in each recursive call
 - ▶ In initialization, create an “index” that lets you process all points in order by y-coordinate
 - ▶ (There are other ways to address this.)