CSE 417
Divide & Conquer (pt 3)
More Examples

UNIVERSITY *of* WASHINGTON

Reminders

> HW2 due Sunday

> Extra office hours after class (CSE 212)

> HW3 will be posted tomorrow
  - construct Voronoi diagrams on paper using the algorithm we discuss today
  - (should be quick)

Divide & Conquer Review

- Apply the steps:
  1. Divide the input data into 2+ parts
  2. Recursively solve the problem on each part
  3. Combine the sub-problem solutions into a problem solution

- Key questions:
  1. Can you solve the problem by combining solutions from sub-problems?
  2. Is that easier than solving it directly?

- Use master theorem to calculate the running time

Famous Algorithm Review

- **Integer Multiplication**: Karatsuba
  - key point: only 3 recursive calls, so $T(n) = 3\,T(n/2) + O(n)$ ~> $O(n^{\lg 3}) = O(n^{1.585})$
  - sub-problems are multiplications on numbers half as large
  - Matrix Multiplication: Strassen
    - $7\,T(n/2) + O(n^2)$ ~> $O(n^{\lg 7}) = O(n^{2.808})$

- **FFT**: Cooley & Tukey
  - key point: divides data into odd and even indexes
  - $2\,T(n/2) + O(n)$ ~> $O(n \log n)$
  - Integer Multiplication: Schönhage & Strassen
    - use FFTs to reduce to $O(n)$ multiplication problem — $O(n \log n \log\log n)$

Outline for Today

> Counting inversions
> Voronoi diagrams
> Closest pair of points

Counting Inversions

- **Problem**: Given an array A of length n, count the number of index pairs (i, j) such that i < j but A[i] > A[j]
- Example: if A is sorted, then there are 0 inversions
- Example: if A is in decreasing order, there are n(n-1)/2 inversions
  - there are n(n-1)/2 pairs (i, j) satisfying 0 ≤ i < j < n
  - every pair is an inversion

Counting Inversions: Brute Force

- Brute-force solution:

```
int count = 0;
for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++)
    if (A[i] > A[j])
      count += 1;
return count;
```

- Runs in $\Theta(n^2)$ time

Counting Inversions: Application

- Measure the **difference** between two lists of rankings of n things
  - music, candidates, web sites, etc.
- Replace each element in one list with the ranking (a number) of that item in the second list
- Example: ranking Beetles band members
  - Your List: John, Paul, George, Ringo
  - My List: Paul, George, John, Ringo
  - Result is [3, 1, 2, 4]
    - has 2 inversions (3,1) and (3,2)

Counting Inversions: Application

- Measure the **difference** between two lists of rankings of n things
  - music, candidates, web sites, etc.
- Replace each element in one list with the ranking (a number) of that item in the second list
- If rankings are the same, result is sorted, so no inversions
  - use number of inversions as a measure of how close they are
- See the textbook for more applications

Counting Inversions: Divide & Conquer

> Apply divide & conquer...

1. Divide A[0..n-1] into halves, A[0..n/2-1] and A[n/2..n-1]
   > same as merge sort

2. Recursively count inversions in each half

3. Combine...

Counting Inversions: Divide & Conquer

> Combine step…

> Consider any pair of indices (i, j)
> Four possibilities
> - **i** in first half, **j** in first half
> - **i** in first half, **j** in second half
> - **i** in second half, **j** in first half
> - **i** in second half, **j** in second half

from recursive call on A[0..n/2-1]

need to count these…

doesn’t satisfy i < j

from recursive call on A[n/2..n-1]

Counting Inversions: Divide & Conquer

- Combine step…
  - count pairs (i, j) with i in first half, j in second half, and A[i] > A[j]
  - answer is that count plus answers from two recursive calls

- Brute force solution:

```
for (int i = 0; i < n/2; i++)
  for (int j = n/2; j < n; j++)
    if (A[i] > A[j])
      count++;
```

- Runs in $\Theta((n/2)(n/2)) = \Theta(n^2)$ time

Counting Inversions: Divide & Conquer

- Need a <u>faster</u> way to answer this question:

  How many elements in A[n/2..n-1] are smaller than A[i]?

- What technique have we learned that can answer this sort of question?

- Can apply binary search if A[n/2..n-1] is **sorted**
  - so let's sort it
  - one *non-obvious* trick: I'll do this recursively

Counting Inversions: Divide & Conquer

> Apply divide & conquer...

1. Divide A[0..n-1] into halves, A[0..n/2-1] and A[n/2..n-1]
2. Recursively **sort & count** inversions in each half
3. Combine:
   - for i = 0 .. n/2, binary search for A[i] in A[n/2..n-1]
     - index gives number of j's with A[i] > A[j]
     - add to count of inversions from recursive calls
   - apply "two finger" merge to make A sorted

Counting Inversions: Divide & Conquer

> Apply divide & conquer...

> Divide + combine is O(n log n) because of n/2 binary searches
  - $C = \log_2 2 = 1$, so compare to $n^C = n$
  - master theorem does not give a specific answer
    > must be $\Omega(n \log n)$ since divide + combine is $\Omega(n)$
    > must be $O(n^{1+\varepsilon})$ for any $\varepsilon > 0$ since divide + combine is $O(n^{1+\varepsilon})$
    > could be $O(n (\log n)^2)$ or something like that (can't tell from this analysis)

> We can improve it exactly as in earlier examples...
  - binary searches are doing a lot of wasted work

Counting Inversions: Divide & Conquer

- Apply divide & conquer...
- Divide + combine is O(n log n) because of n/2 binary searches
- We can improve it exactly as in earlier examples...
  - since A[0] ≤ A[1] ≤ ... (it's sorted now),
    indexes returned by each binary search can only increase as we go
  - rather than binary search, just use linear search
    - total number of steps to the right is n/2 so total time is O(n)
- This is another "two finger" algorithm

Counting Inversions: Divide & Conquer

- Apply divide & conquer...
- Divide + combine is O(n log n) because of n/2 binary searches
- We can improve it exactly as in earlier examples...
  - sequence of linear searches takes O(n) time
- This is another ”two finger” algorithm
  - in fact, the fingers make the same steps as in merging
  - in fact, we could count and merge simultaneously (exercise!)

Counting Inversions: Divide & Conquer

- Apply divide & conquer…

  1. Divide A[0..n-1] into halves, A[0..n/2-1] and A[n/2..n-1]
  2. Recursively sort & count inversions in each half
  3. Combine with "two finger" merge & count inversions

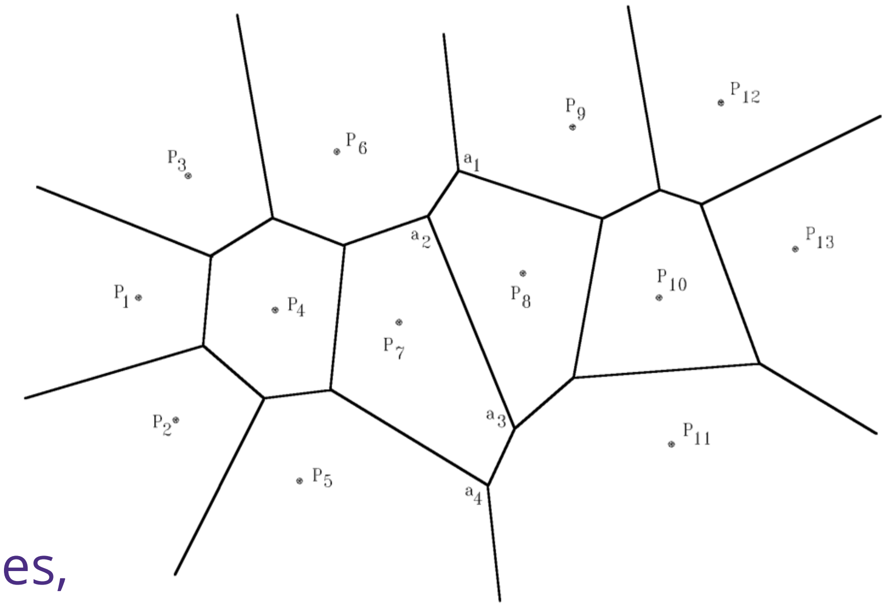
- Divide + combine in O(n) time, so O(n log n)
  - only slightly faster than binary search approach
  - BUT we look much smarter after covering our tracks

Outline for Today

> Counting inversions
> Voronoi diagrams
> Closest pair of points

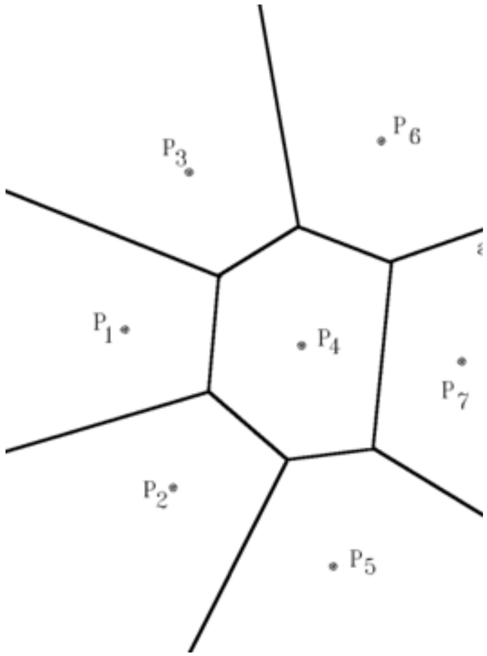
Voronoi Diagrams

- Given a set of sites $p_1, \ldots, p_n$, separate the plane into the regions *closest* to each site
- Example on the right has 14 sites, so 14 regions as well
- (Coding details are complicated, so we'll stay high-level.)



Picture from par.cse.nsysu.edu.tw/~cbyang/course/algo/algonote/algo4.ppt

Voronoi Diagrams



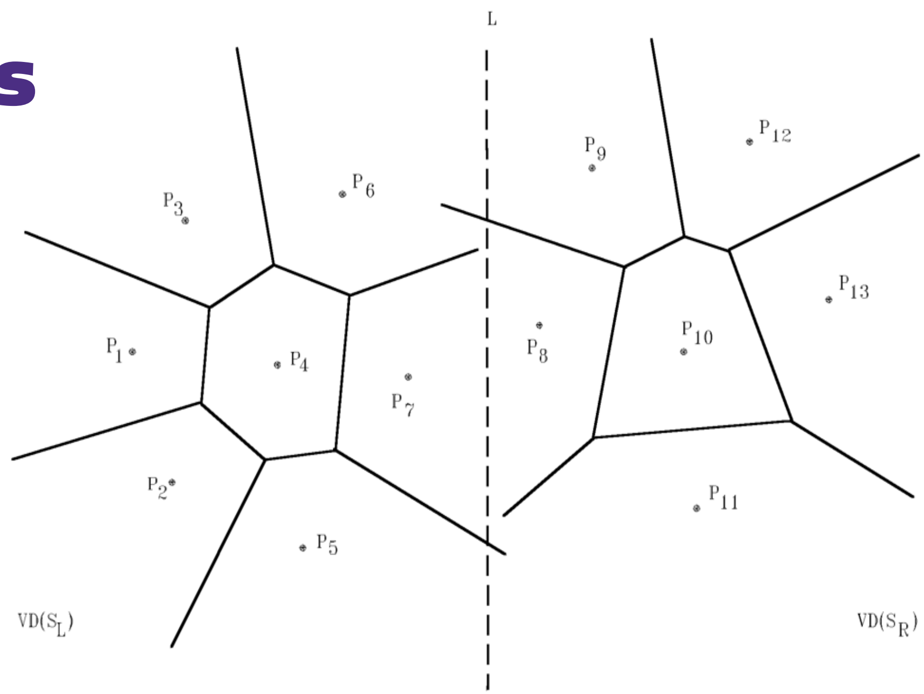
> Brute force solution:
  – for each site, find its Voronoi region

> Finding the Voronoi region for a site:
  – for every *other* site, find the separating line
    > line that is equal distance from each site
    > perpendicular bisector of line segment drawn between them
  – somehow fit the closest ones together into the region boundary (complicated)

> $\Omega(n)$ per region, so $\Omega(n^2)$ all together

Voronoi Diagrams
Divide & Conquer

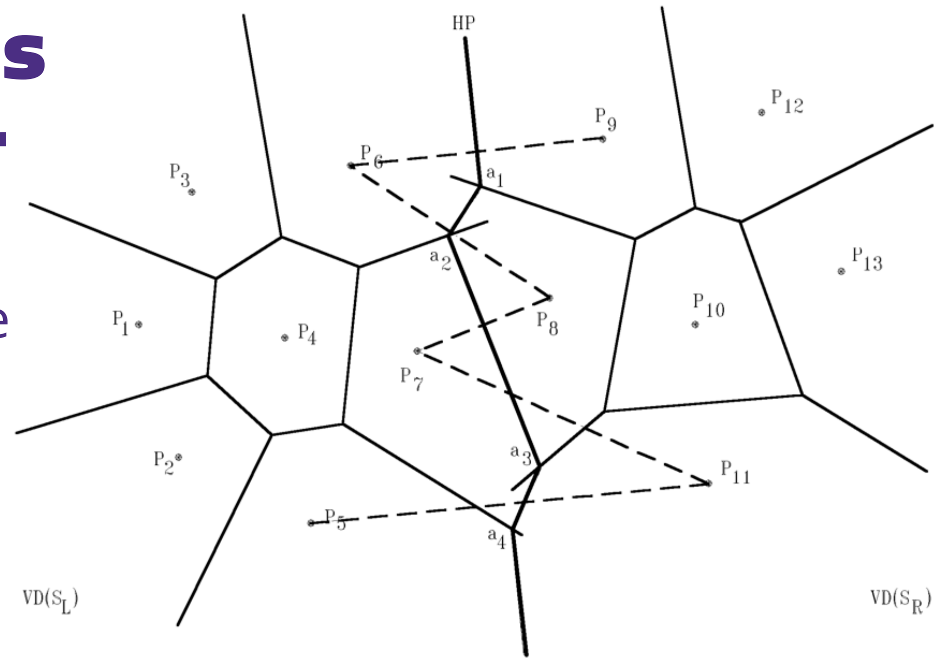
> Divide the sites in half by drawing a line
  - usually horizontal or vertical
  - in principle, any line is fine



> Recursively find the Voronoi diagrams for each half
  - use brute force when there are 1–3 sites (easy cases)

Picture from par.cse.nsysu.edu.tw/~cbyang/course/algo/algonote/algo4.ppt

Voronoi Diagrams Divide & Conquer

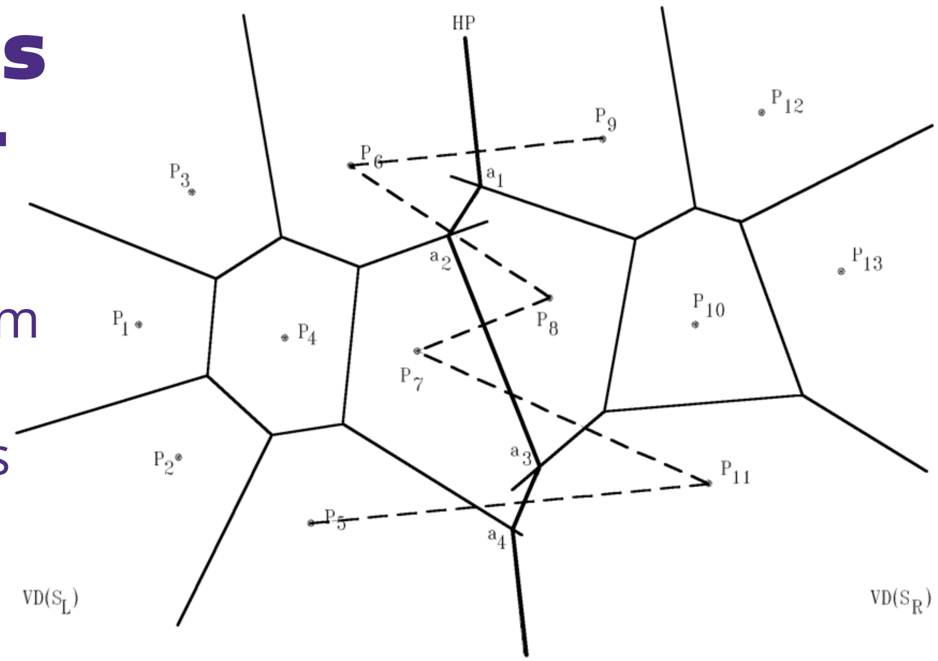


> Only segments missing are where two sides meet
  - already know which site is closest on each side
  - only need to figure out which side is closer

> Find piecewise-linear, separating path on boundary
  - equal distance between a site on each side

Picture from par.cse.nsysu.edu.tw/~cbyang/course/algo/algonote/algo4.ppt

Voronoi Diagrams Divide & Conquer



> Use a “two finger” algorithm

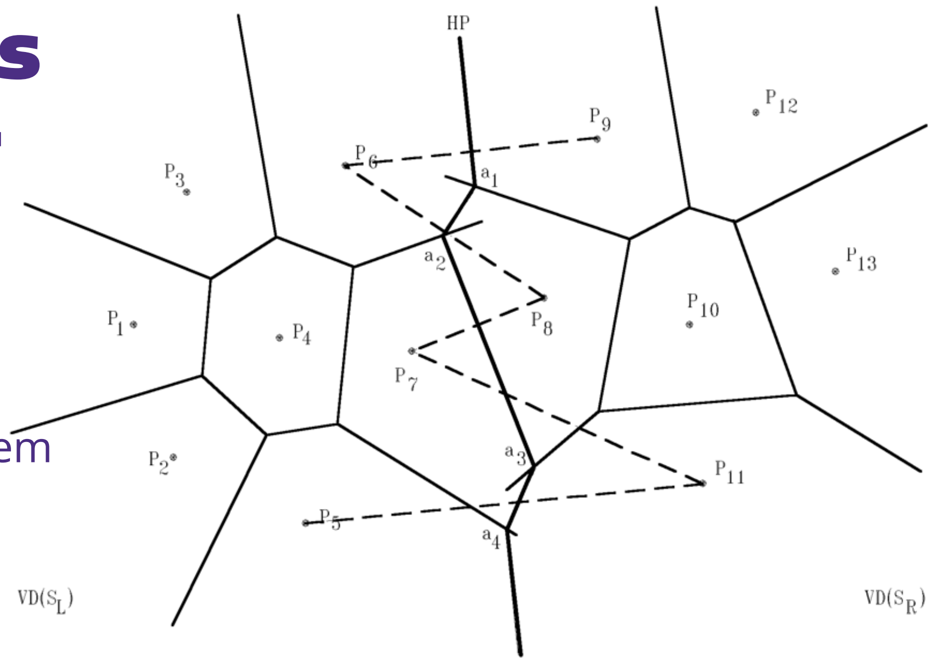
> Start with two highest sites near the boundary
  – p_6 and p_9 in the picture
  – draw bisector between them

> Extend the bisector until it hits edge of a Voronoi region
  – one of those sites is no longer closest on that side

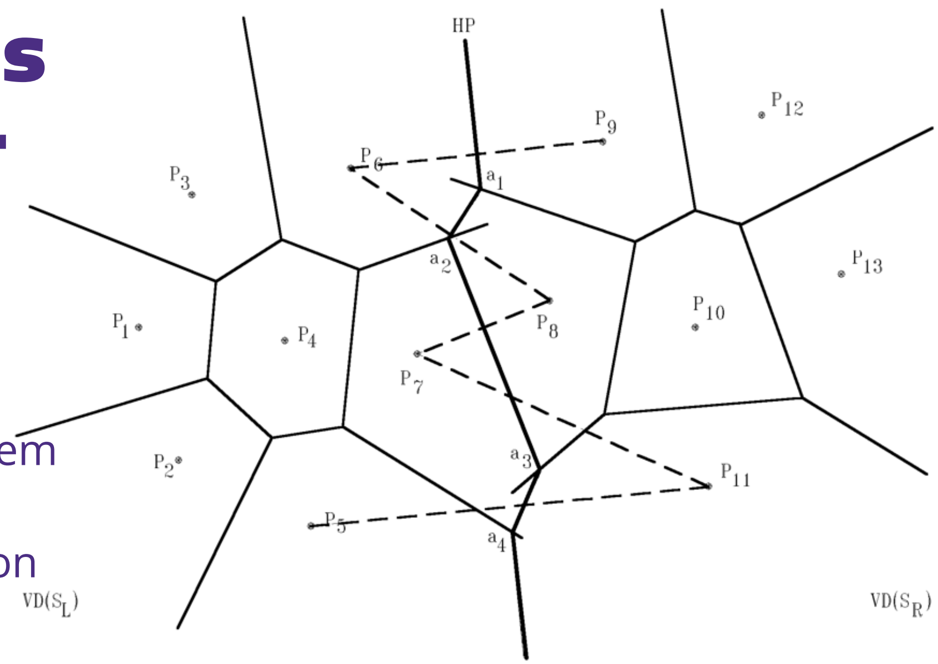
Voronoi Diagrams
Divide & Conquer

> "Two finger" algorithm
  - finger on site from each side
  - drawing bisector between them

> Extend bisector to edge of a Voronoi region
  - one of those sites is no longer closest on that side
  - move finger to new closest site
  - start drawing the new bisector (must connect!)

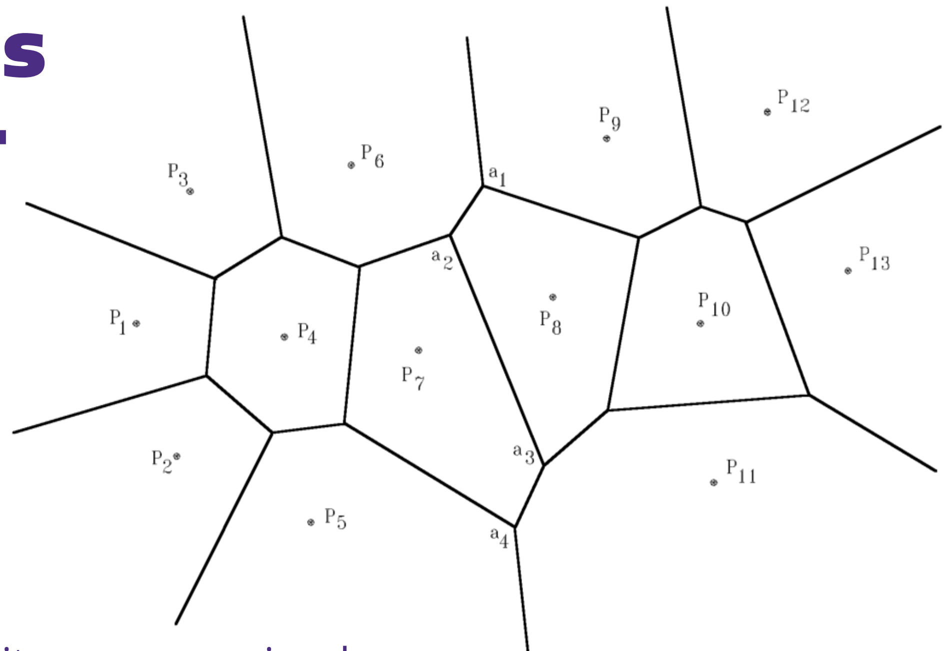


Voronoi Diagrams Divide & Conquer



- “Two finger” algorithm
  - finger on site from each side
  - drawing bisector between them
  - move finger when bisector crosses edge of Voronoi region
- Done when current bisector goes off to infinity
- Bisectors are equal distance between **closest** two p_i’s

Voronoi Diagrams Divide & Conquer



- Combine step takes time proportional to number of sites on the boundary
- Worst case is O(n)
  - typically only a fraction of n sites are examined
- Master theorem says we get an O(n log n) algorithm
  - huge improvement over brute force

Picture from par.cse.nsysu.edu.tw/~cbyang/course/algo/algonote/algo4.ppt

HW3: Voronoi diagrams

- HW3 asks you to compute a Voronoi diagram **on paper**
  - divide by copying half the sites to two separate pieces of paper
  - combine by copying answers back and then adding separating path
- See the assignment for more details on the algorithm
- Note: your answer ***does not*** *have to be pixel perfect*
  - can just eye-ball the bisectors
  - key is to understand the algorithm:
    - why is it correct? — only boundary is missing & this finds it
    - why is it efficient? — O(n) to compute boundary

Outline for Today

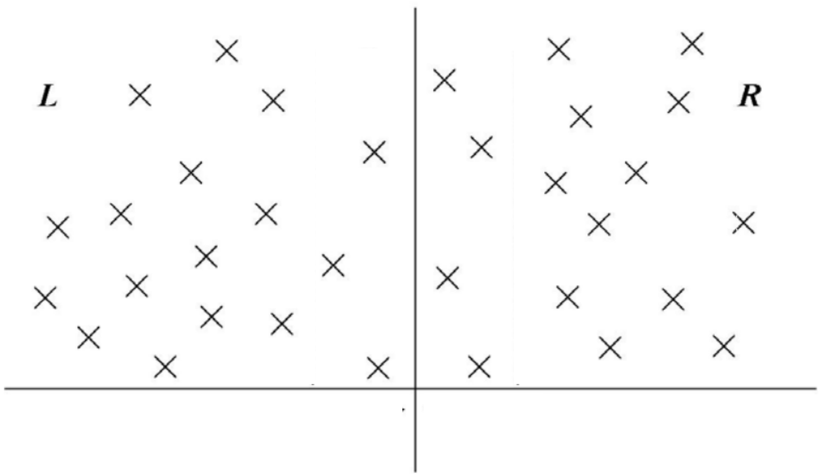
- Counting inversions
- Voronoi diagrams
- Closest pair of points

Closest Pair of Points

> **Problem**: Given a set of points $p_1, \ldots, p_n$ in the plane, find the pair of distinct points (p_i, p_j) with minimum distance $|p_i - p_j|$

> Brute force solution:
  – compute the distance of every pair; keep track of the closest
  – takes $\Theta(n^2)$ time since there are $n(n-1)/2 = \Theta(n^2)$ pairs

Closest Pair of Points

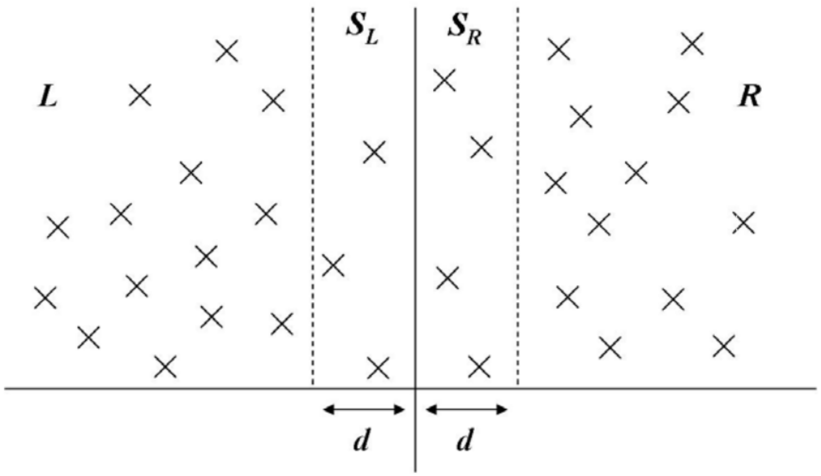


> Apply divide & conquer...

1. Divide the points horizontally
   > let L & R be the two halves
2. Recursively finds the closest pair in L and R
   > let d be the smallest distance of the two
3. Combine: find closest pair p in L, q in R
   > return closest of the three pairs

Picture from www.inrg.csie.ntu.edu.tw/algorithm2014/course/Divide%20&%20Conquer.pdf

Closest Pair of Points

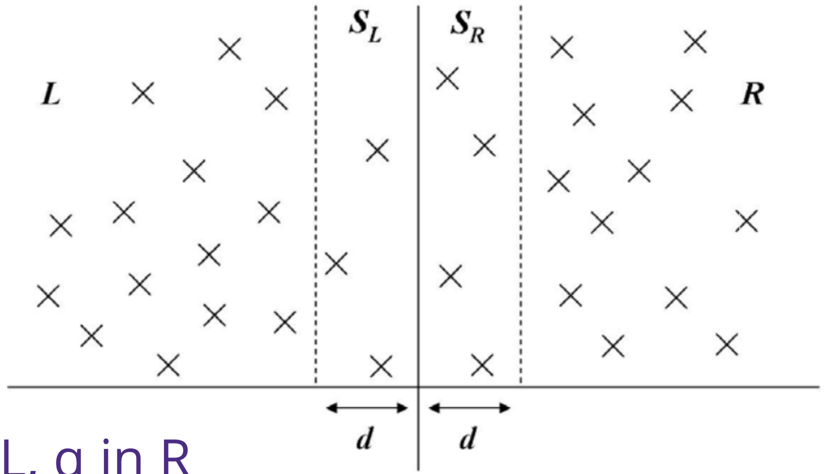


> Apply divide & conquer...

1. Divide the points horizontally
2. Recursively find the closest pairs in L and R
   > let d be the smallest distance of the two
3. Combine: find closest pair p in L, q in R with |p – q| < d
   > only need to consider p in S_L and q in S_R
   > where S_L = L within d of line, S_R = R within d of line

Picture from www.inrg.csie.ntu.edu.tw/algorithm2014/course/Divide%20&%20Conquer.pdf

Closest Pair of Points



> Apply divide & conquer...

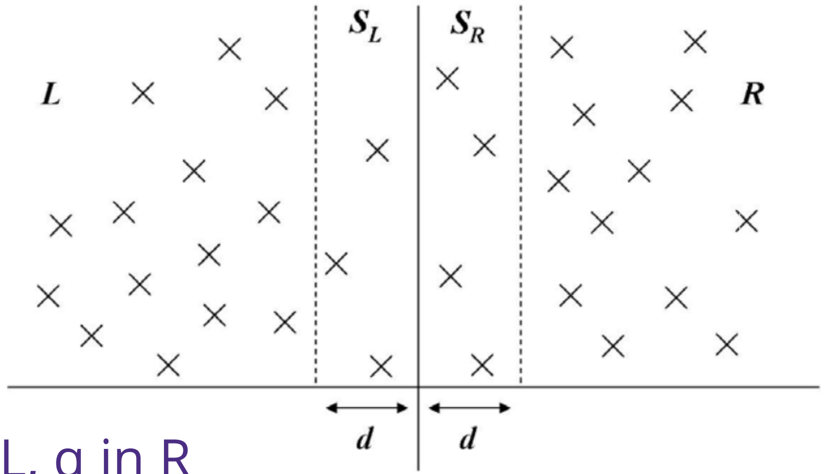
3. Combine: find closest pair p in L, q in R

> only need to consider p in S_L and q in S_R

> Unfortunately, this is still too many to check by brute force... could be, say, n/6 on each side, giving $\Theta((n/6)(n/6)) = \Theta(n^2)$ time

> Will reduce the search by considering vertical distance also...

Closest Pair of Points



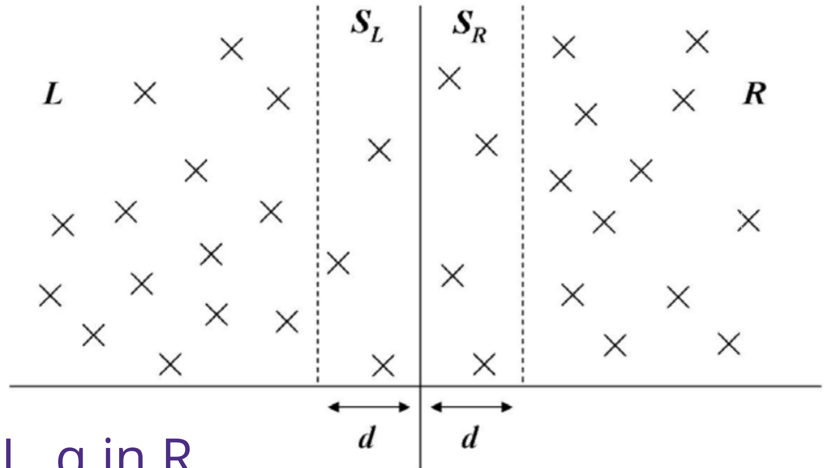
- Apply divide & conquer...

  3. Combine: find closest pair p in L, q in R
     - only need to consider p in S_L and q in S_R

- Before we start, sort all the points by y-coordinate.
- Now get S_L and S_R sorted by y-coordinate in O(n) time (by filtering big list)

- For each p in S_L, just compare to those in S_R whose y-coordinate is within d of p's

Closest Pair of Points



> Apply divide & conquer...

3. Combine: find closest pair p in L, q in R

> only need to consider p in S_L and
> q in S_R whose y-coordinate is within d of p's

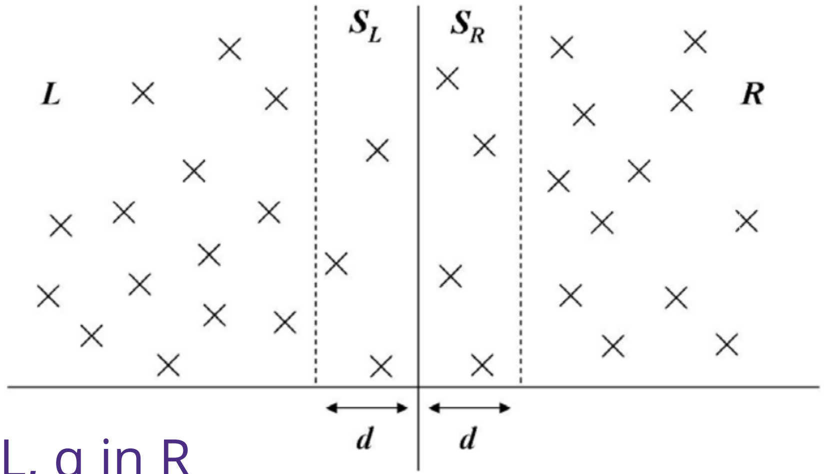
> Three finger algorithm:

- one finger on p in S_L
- one finger first q in S_R with $q.y \geq p.y - d$
- one finger last q in S_R with $q.y \leq p.y + d$

O(n) total finger moves

compare p to all q's in this range

Closest Pair of Points



> Apply divide & conquer...

3. Combine: find closest pair p in L, q in R

   > only need to consider p in S_L and q in S_R whose y-coordinate is within d of p's

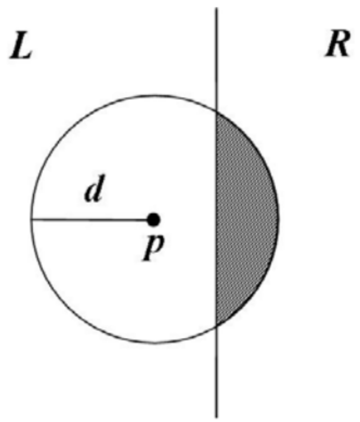
> Three finger algorithm:
- one finger on p in S_L
- one finger first q in S_R with $q.y \geq p.y - d$
- one finger last q in S_R with $q.y \leq p.y + d$

compare p to all q's in this range

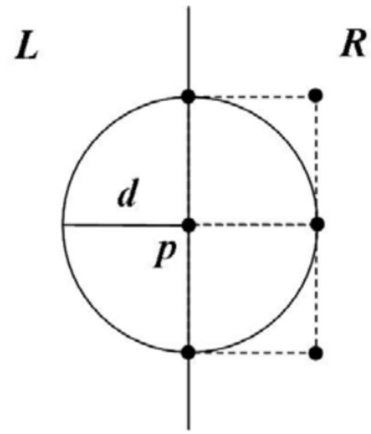
Q: how many are there?

Closest Pair of Points

Want to check for points in this shaded region:

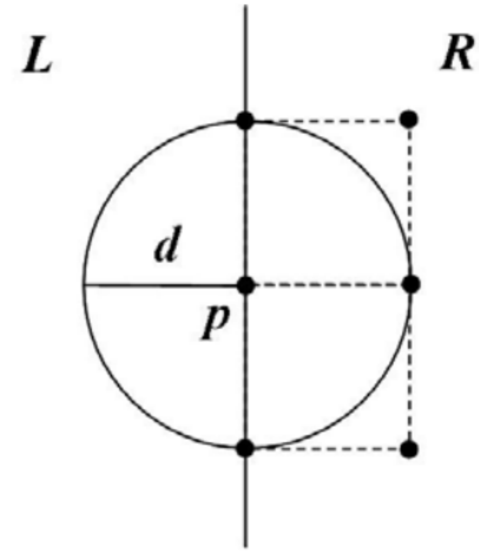


Will check all points in this d x 2d region:



Pictures from www.inrg.csie.ntu.edu.tw/algorithm2014/course/Divide%20&%20Conquer.pdf

Closest Pair of Points

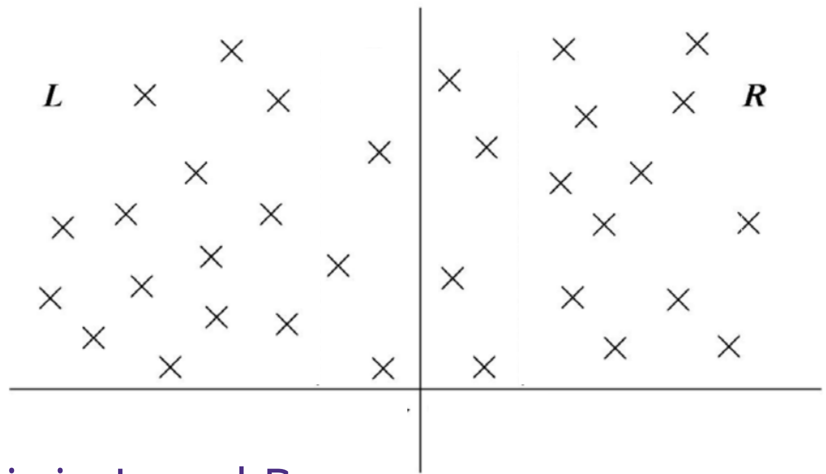


> **Q**: How many points could be in the d x 2d box?

> **A**: no more than 6 (see picture)
- key point is that *no points in R are closer than d apart*!
- this limits the number that could be close to p to O(1)

> Example where a lot of elbow grease and problem-specific analysis is needed to find the best solution
- used general techniques to get most of the way there

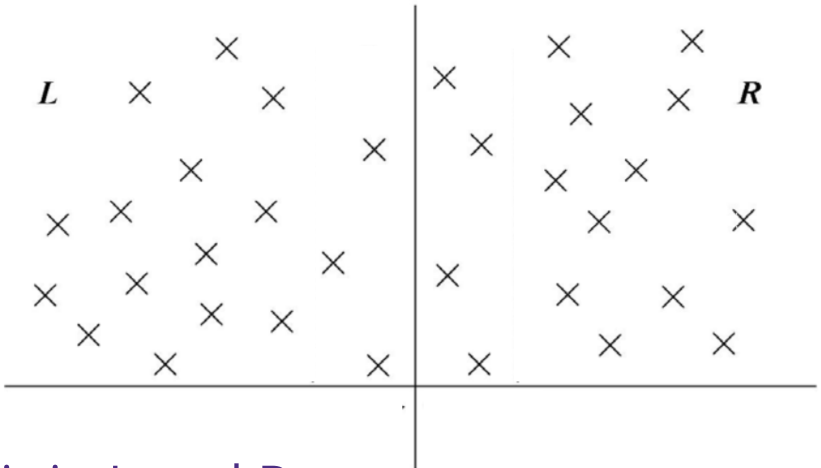
Closest Pair of Points



> Apply divide & conquer…

1. Divide the points into L & R
2. Recursively finds the closest pair in L and R
   > let d be the smallest distance of the two
3. Combine: find closest pair p in L, q in R with |p – q| < d
   > only need to consider p in S_L and q in S_R whose y-coordinate is within d of p's
   > only O(1) such points for each p
   > return smallest of the 3 pairs above

Closest Pair of Points



- Apply divide & conquer...
  1. Divide the points into L & R
  2. Recursively finds the closest pair in L and R
  3. Combine: find closest pair p in L, q in R with $|p - q| < d$
     - each p is compared to O(1) q's, so O(n) time
- Divide + combine in O(n) time
- Running time is O(n log n) by master theorem