Advanced Algorithms

Lecture 7

Computational Geometry Algorithms:

Divide-and-Conquer

Tung Kieu
tungkvt@cs.aau.dk

ILO of Lecture 7

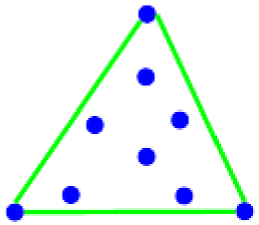
- Output sensitive algorithms and D&C algorithms
  - To understand the concept of ***output sensitive*** algorithms;
  - To be able to apply the ***divide-and-conquer*** algorithm design technique to geometric problems;
  - To recall how recurrences are used to analyze the divide-and-conquer algorithms;
  - To understand and be able to analyze the Jarvis's march algorithm and the divide-and-conquer algorithms for finding a closest pair and for finding the convex hull.

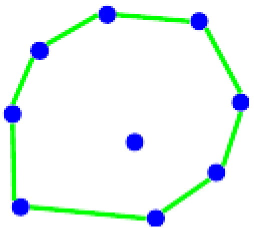
Agenda

- Output sensitive algorithms
  - Convex hull: Jarvis's march
- Divide-and-conquer algorithms
  - Finding the closest pair of points
  - Convex hull

Size of the output

- In computational geometry, the size of an algorithm's **output** may differ/depend on the **input**.
  - Line segment intersection problem vs. convex-hull problem.





  - Although both sets have 9 points, but the convex hulls have different number of vertices.
  - It would be nice to have an algorithm that runs fast if the convex hull is small.
  - Graham's scan running time depends only on the size of the **input** – it is independent of the size of the **output**

Jarvis's March: Convex Hull

- Give a set of points S, identify the convex hull of S that is the smallest convex polygon that contains all the points of S.

- Javis's march for identifying convex hulls
  - Identify the point p_0 that has the minimum y-coordinate, or the leftmost such point in case of a tie.
  - Identify the point p_H that has the maximum y-coordinate, or the furthest (w.r.t. p_0) such point in case of a tie.
  - Do the followings on the right chain and then left chain.

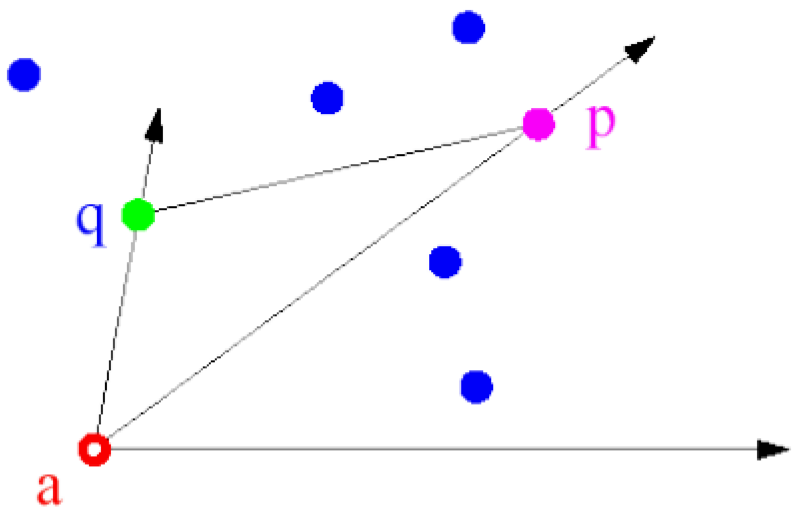
Jarvis's march

- Right chain
  - 1. Treat $\boldsymbol{p_0}$ as the anchor point.
  - 2. Choose the next vertex p_i that has the smallest polar angle with respect to the anchor point from the x-axis, and include p_i in the convex hull.
  - 3. Treat p_i as the new anchor point, and repeat step 2 until $\boldsymbol{p_H}$ is included in the convex hull.
- Left chain
  - 1. Treat $\boldsymbol{p_H}$ as the anchor point.
  - 2. Choose the next vertex p_i that has the smallest polar angle with respect to the anchor point from the negative x-axis, and include p_i in the convex hull.
  - 3. Treat p_i as the new anchor point, and repeat step 2 until $\boldsymbol{p_0}$ is included in the convex hull.

Comparing angles

- *How do we compare angles*?
  - *Observation*: We do not need to compute the actual angle.
  - We just need to be able to compare the angles

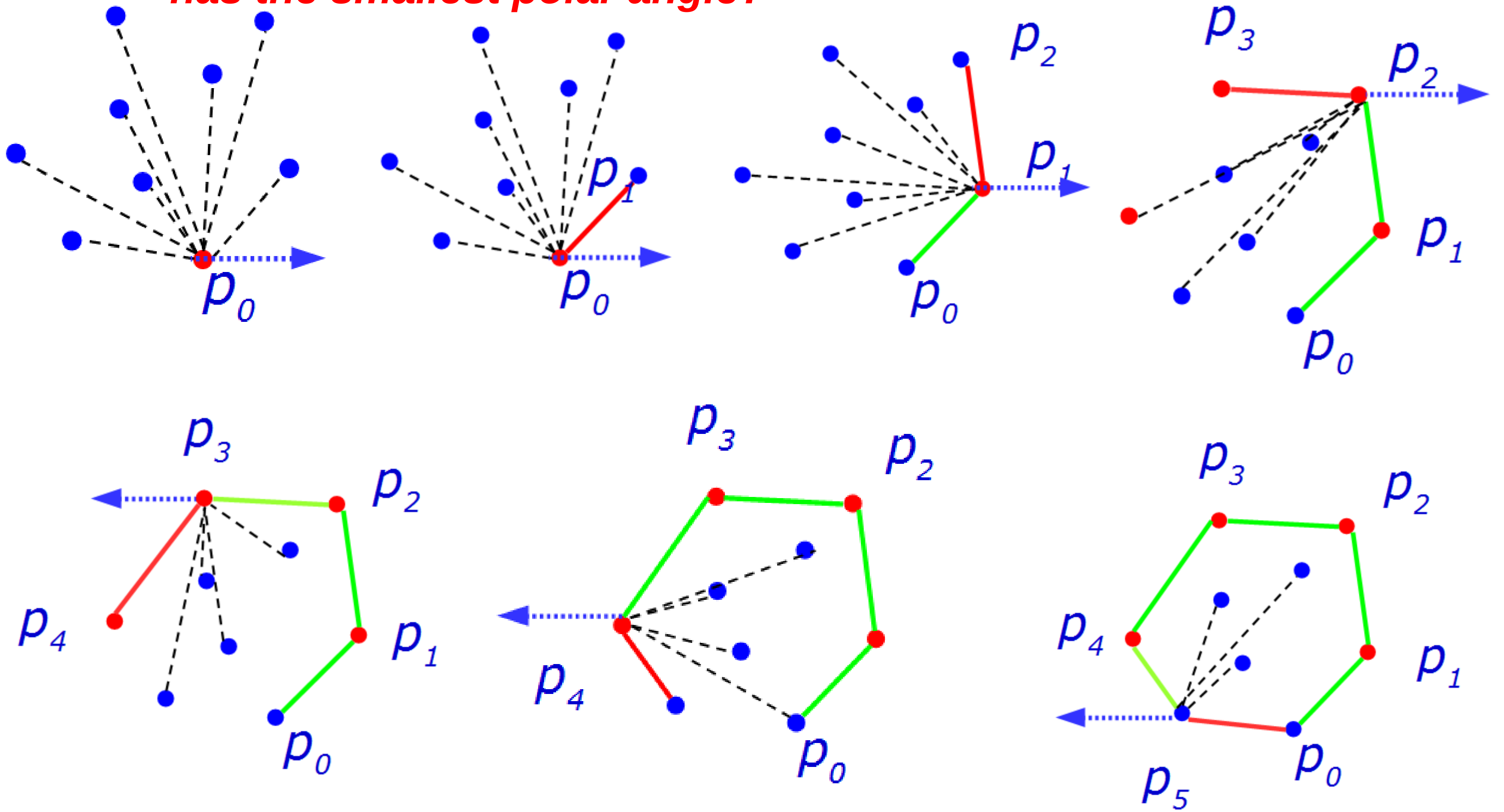
$\theta(p) < \theta(q)$
$\Leftrightarrow$ orientation(a,p,q) = counterclockwise



Example

How do you figure out which point has the smallest polar angle?

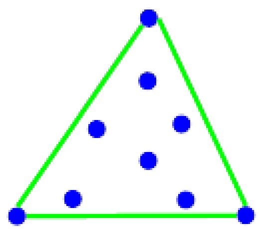
$p_H = p_3$, right chain is done

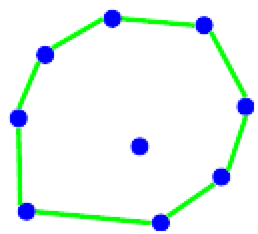


Left chain is done.

Complexity

- Mini-quiz: How many cross products are computed for the following two examples?





Step	# of cross products
1	n-2=n-(1+1)
2	n-3=n-(2+1)
3	n-4=n-(3+1)
…	…
h	n-(h+1)

- Suppose there are h vertices on the convex hull.
- (n-2)+(n-3)+(n-4)+…+(n-h) +(n-h-1)
- =n*h – (2+3+4+…+h+h+1)
- =n*h- 0.5 * (1+h)*h-h
- =n*h-0.5h^2-1.5h
- h is at most n, so that O(nh).

Left: n=9, h=3,
9*3-0.5*9-1.5*3=18

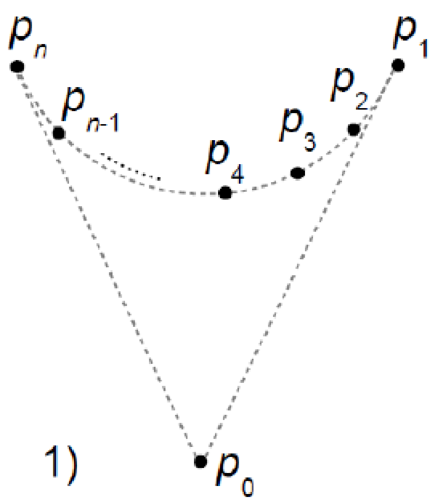
Right: n=9, h=8,
9*8-0.5*64-1.5*8=28

Complexity

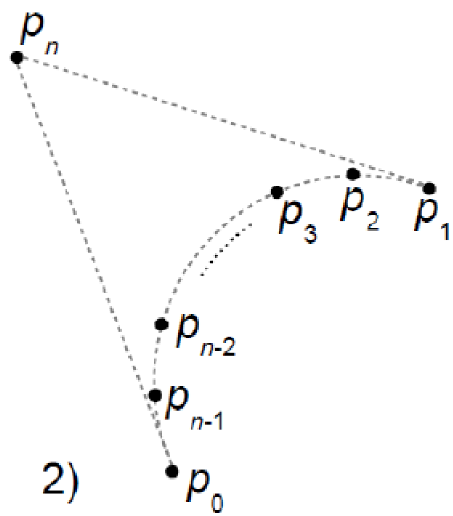
- Finding the lowest and highest points: $O(n)$.
- For each vertex in the convex hull: at most n–2 cross-product computations.
- Total: $\boldsymbol{O(nh)}$, where h is the number of vertices in the convex hull.

- **Output-sensitive** algorithm: its running time depends on the size of the output.
  - When should we use Jarvi's march instead of the Graham's scan?
  - When h<lgn, Jarvis's march is faster.

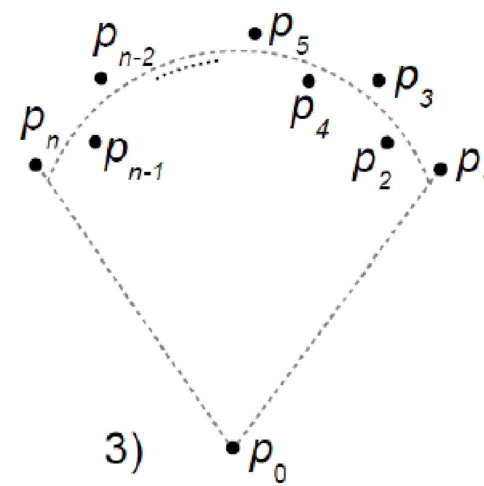
Mini quiz (from exam 2016)



Jarvi's march



Jarvi's march



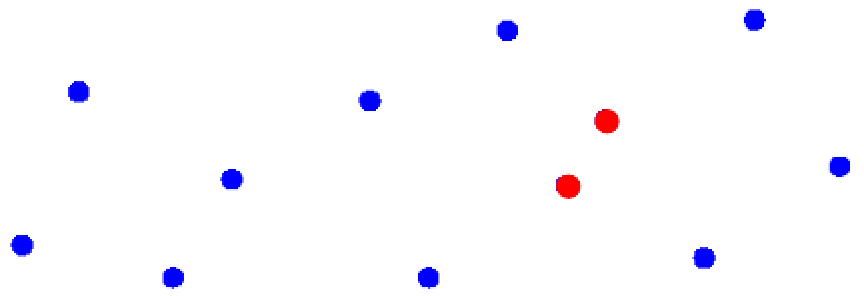
Graham's scan

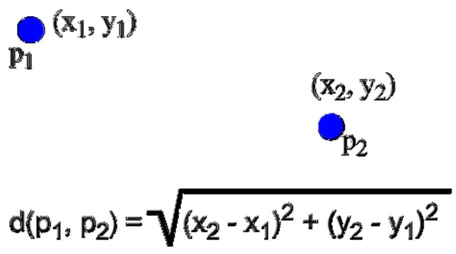
Agenda

- Output sensitive algorithms
  - Convex hull: Jarvis's march
- Divide-and-conquer algorithms
  - Finding the closest pair of points
  - Convex hull

Closest pairs of points

- Given a set P of n points, find p, q ∈ P, such that the distance d(p, q) is minimum.
- Checking the distance between two points is O(1)
  - E.g., Euclidean distance
- What is the brute-force algorithm and its running time?





- Can we do better (e.g., θ(nlgn)) if we use divide-and-conquer?

Steps of divide-and-conquer

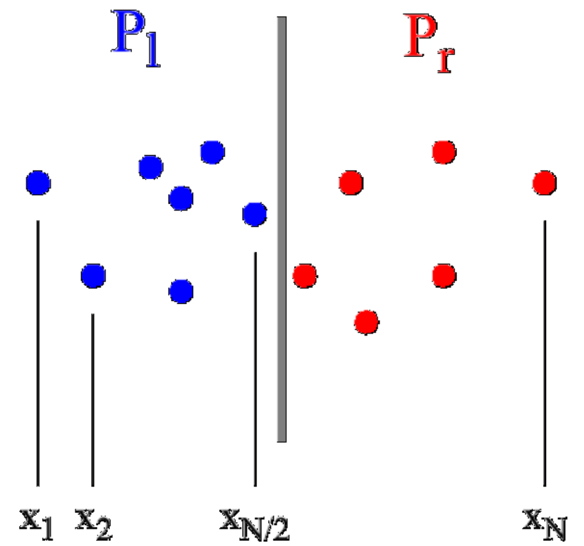
- Base case: if the problem size is small enough to solve it in a straightforward/brute-force manner, solve it.
- Otherwise do the following:
  - **Divide**: Divide the problem into a number of *disjoint* *sub-problems.*
  - **Conquer**: Use divide-and-conquer *recursively* to solve the sub-problems.
  - **Combine**: Take the solutions to the sub-problems and combine these solutions into a solution for the original problem.
    - *Often the most difficult step for computational geometry problems.*

Dividing into sub-problems

- How do we divide into sub-problems?
  - Idea: Sort on x-coordinate, and divide into left and right parts using a vertical line:

    $p_1\ p_2\ \ldots\ p_{n/2}\ p_{n/2+1}\ \ldots\ p_n$

  - Solve recursively the left sub-problem P_l (closest-pair distance d_l) and the right sub-problem P_r (closest-pair distance d_r).



- Base case
  - If P_l or P_r has p points where p is less than or equal to 3, just solve it brute-force.
    - Try all $\binom{p}{2}$ pairs of points and return the closest pair.

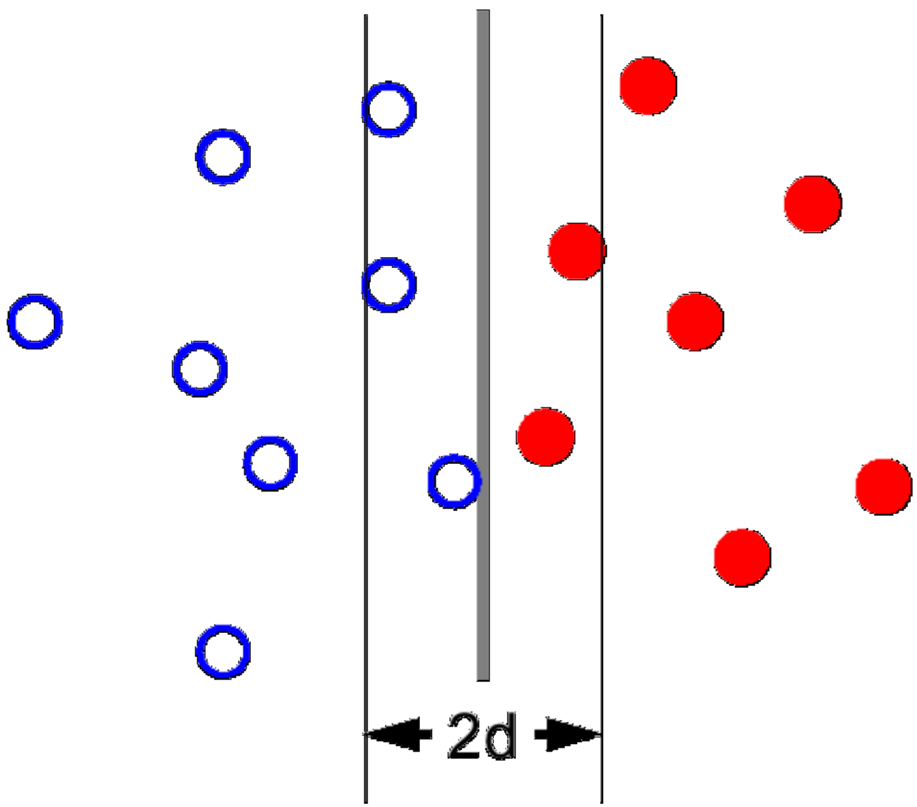
Combining two solutions

- How do we combine two solutions to sub-problems ?
  - We know that on the left side, the closet-pair distance is d_l, and on the right side, the closet-pair distance is d_r.
  - Let $d = \min\{d_l, d_r\}$. Is d the closet-pair distance for all points?

- *Observation* 1:
  - Although we already have the closest pair where both points are either in the left or in the right sub-problem, we have to check pairs where one point is from one sub-problem and another from the other.
- *Observation* 2:
  - Such closest-pair can only be somewhere in a strip of width ***2d*** around the dividing line!
  - Otherwise the points would be more than d units apart.

Combining two solutions

- *Combining solutions*: Finding the closest pair (○ ,●) in a strip of width 2*d*, knowing that no (○ , ○) or (● ,●) pair is closer than *d.*



Worst case

- In the worst case, how many points can be in the strip?
- All $\lceil n/2 \rceil$ points on the left side and all $\lfloor n/2 \rfloor$ points on the right side may be in the strip.

- If we naively compare all the points from the left side of the strip to all the points from the right side of the strip, we will end up $n^2/4$ comparisons. So that we cannot achieve nlgn run time what we expected in the beginning.
  - $T(n)=2T(n/2)+\theta(n^2)$

Solving the recurrence $T(n)=2T(n/2)+\theta(n^2)$

- Recurrence in the form of $T(n)=a*T(n/b)+f(n)$ can be solved by the master method. Can we use the master method?
- First case: if $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\varepsilon>0$, then $T(n) = \Theta(n^{\log_b a})$.
- Second case: if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
- Third case: if $f(n)= \Omega(n^{\log_b a+\epsilon})$ for some constant $\varepsilon>0$, and the regularity condition is also satisfied, then $T(n)=\Theta(f(n))$.
  - Regularity condition
    - *$a*f(n/b)\leq c*f(n)$ for some constant $c<1$and all sufficiently large n*

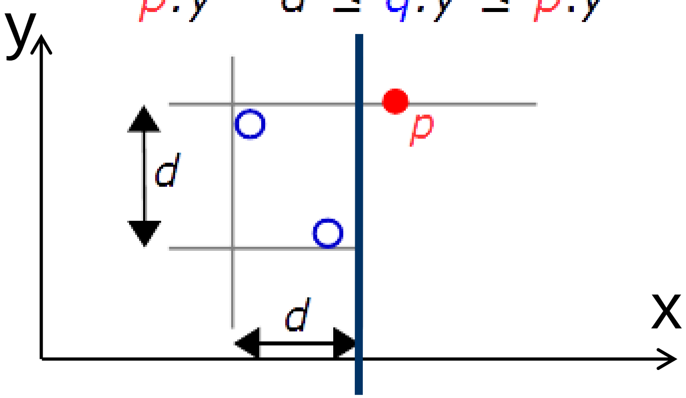
- Case 3: $f(n)=\theta(n^2)=\Omega(n^{1+\varepsilon})$
- $af(n/b)=2*(n^2/4)= n^2/2<= n^2/10$, where $c=1/10$
- $T(n)=\theta(n^2)$
- We will show how can we avoid the naive comparisons and make it faster.

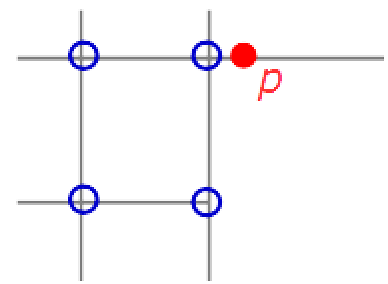
Observations

- We sort the points in the strip on the *y*-coordinate
  - *For a given point p from one partition, where can there be a point q from the other partition that can form the closest pair with p* (*considering only points q.y* ≥ *p.y*)?
  - We only need to consider the following $d \times d$ square:

    $vl.x - d \leq q.x \leq vl.x$ (within the 2d strip)

    $p.y - d \leq q.y \leq p.y$

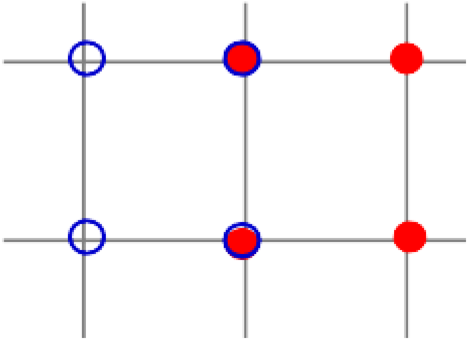




  - How many points can there be in the $d \times d$ square?
    - At most 4
    - If there are more than 4 blue points, the shortest distance between 2 of them should be smaller than d which contradicts that $d = \min\{d_l, d_r\}$.

Algorithm for checking the strip

- For each point *p,* we consider both squares, i.e., the left square and the right square. There can be at most 8 points in the two squares.



- Sort all the points in the strip on their *y*-coordinates.
- For each point *p,* only **7** points whose y-coordinates that are greater than or equal to p.y in the sorted order have to be checked to see if any of them is closer to *p* than *d.*
- It may be possible that it is enough to check fewer than 7 points, but for us it is enough to observe that a *constant* number of points have to be checked.

Pseudo code

- P is an array of points which are already sorted on the x-coordinate.
  - If P is not sorted yet, we should call a sorting algorithm to do so which takes O(nlgn).
- For the first call, we call Closest-Pair(P, 1, n).

```
Closest-Pair(P, l, r)
if r − l < 3 then return Brute-Force-CPair(P, l, r)
q = ⌈(l+r)/2⌉
dl = Closest-Pair(P, l, q-1)
dr = Closest-Pair(P, q, r)
d = min(dl, dr)
for i = l to r do
   if P[q].x - d ≤ P[i].x ≤ P[q].x + d then
      append P[i] to S
Sort S on y-coordinate
for j = 1 to size_of(S)-1 do
  Check if any of d(S[j],S[j+1]), ..., d(S[j],S[j+7]) is
   smaller than d, if so set d to the smallest of them
return d
```

Base case.

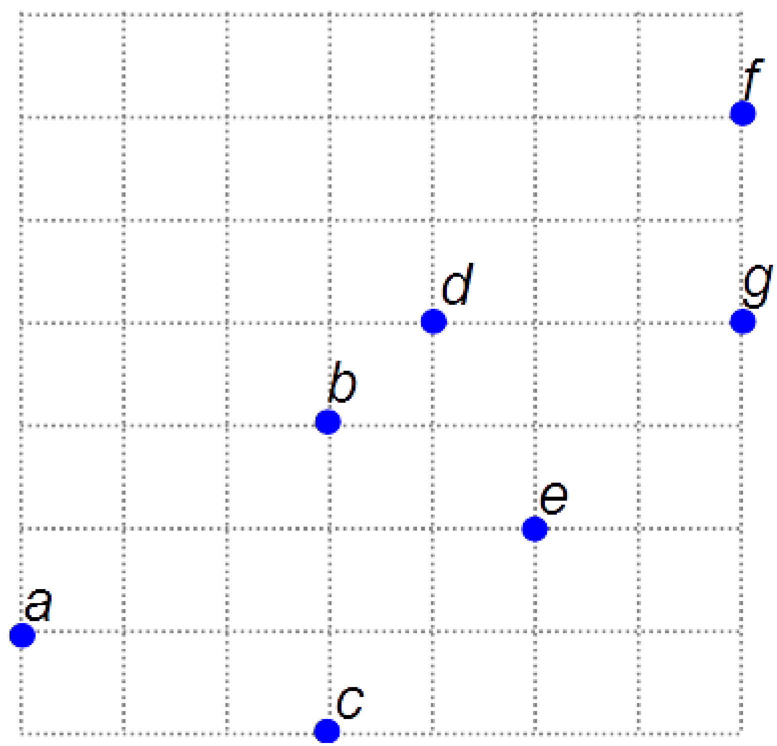
Divide and Conquer.

Filter out the points that lie outside the 2d strip.

Sort the points within the 2d strip and check them according to the order. Every time we only check the next 7 points w.r.t. the order.

Mini-quiz (also on Moodle)

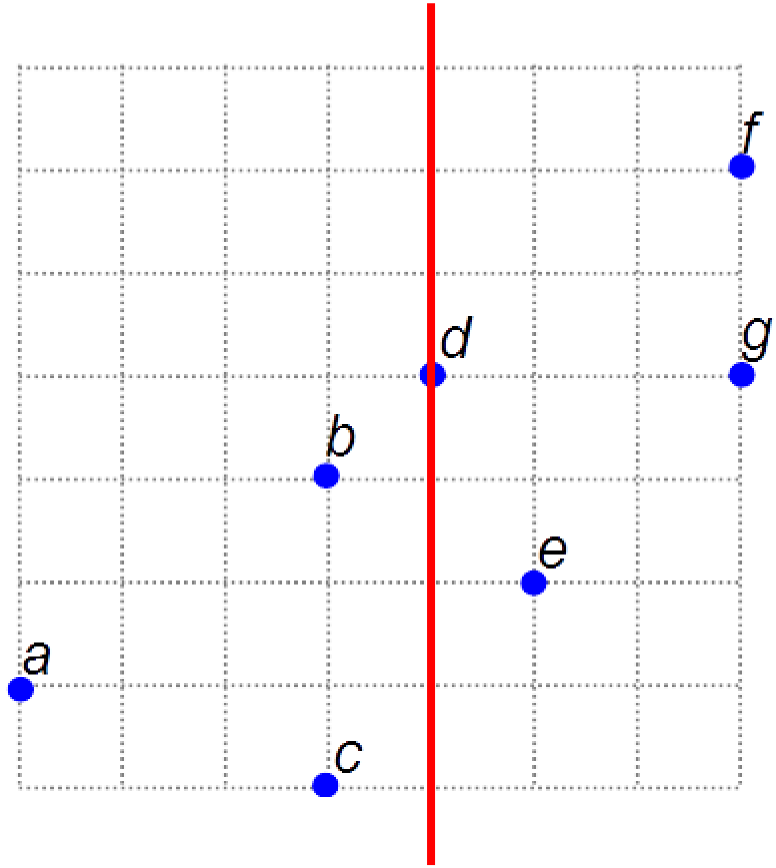
- *How many distance computations are done in this example? Distances between which points are computed?*



Mini-quiz

```
q  = ⌈(l+r)/2⌉
dl  = Closest-Pair(P, l, q-1)
dr  = Closest-Pair(P, q, r)
```

- $q = \lceil (1+7)/2 \rceil = 4$
- Closest-Pair(P, 1, 3)   Closest-Pair(P, 4, 7)
- P[q].x, i.e., d.x, is the vertical line



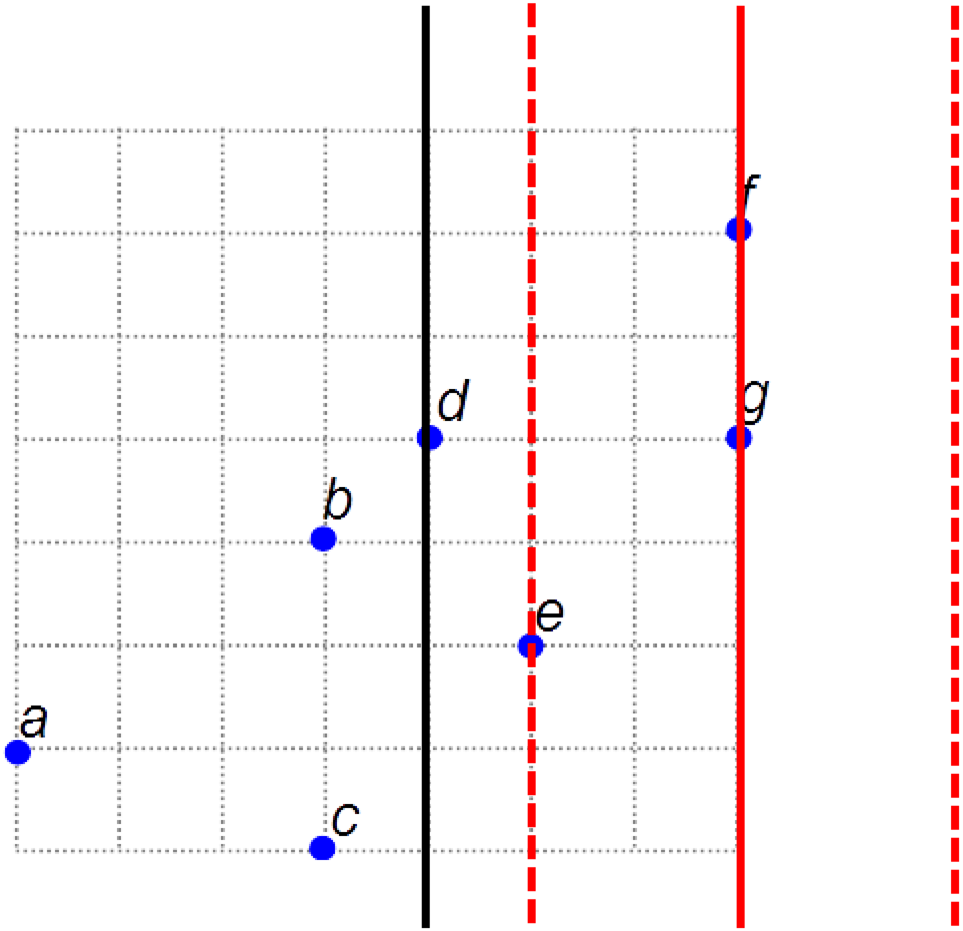
Left side: 3 points. Base case.
3 distance computations: ab, bc, ac.
bc is the closest pair, and the distance is 3.

Right side: 4 points. Divide again.

Mini-quiz

```
q  = ⌈(l+r)/2⌉
dl  = Closest-Pair(P, l, q-1)
dr  = Closest-Pair(P, q, r)
```

- $q = \lceil (4+7)/2 \rceil = 6$
- Closest-Pair(P, 4, 5)   Closest-Pair(P, 6, 7)



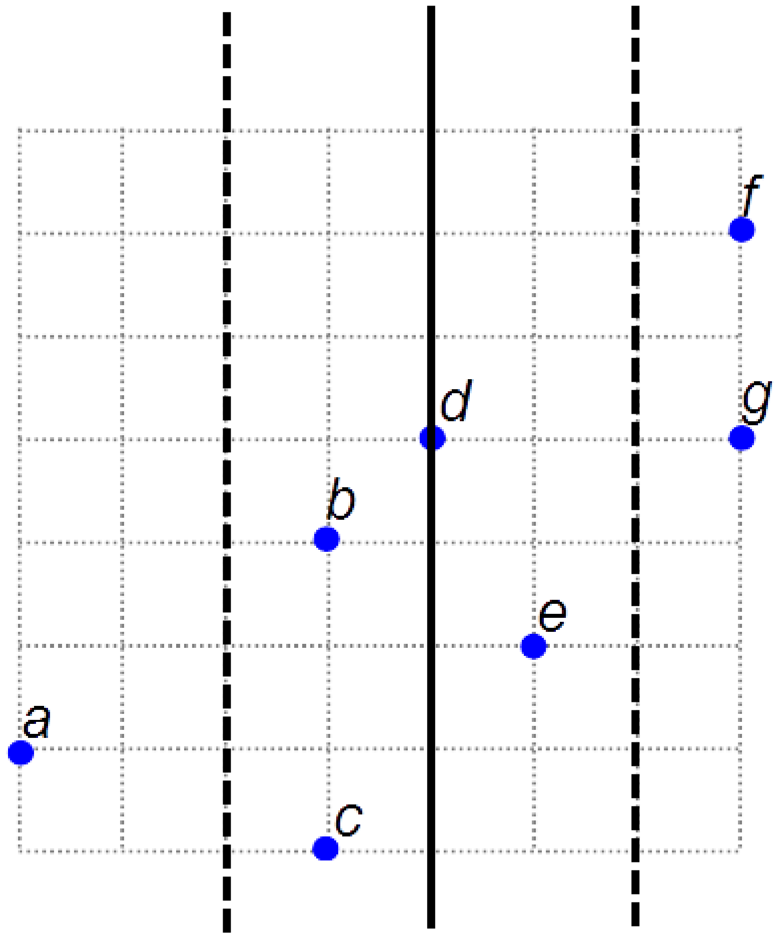
Left side: 2 points. Based case.
1 distance computation: de
de is the closest pair, $\sqrt{5}$.

Right side: 2 points. Based case.
1 distance computation: fg
fg is the closest pair, 2.

Combine: d= min($\sqrt{5}$, 2) =2
f: fg, fe, 2 times.
g: ge, 1 time. In total 3 times.
d=2

Mini-quiz

```
q = ⌈(l+r)/2⌉
dl = Closest-Pair(P, l, q-1)
dr = Closest-Pair(P, q, r)
```



Left side: bc is the closest pair, 3
Right side: fg is the closet pair, 2

Combine: d= min(3, 2) =2
d: db, de, dc, 3 times. d=$\sqrt{2}$
b: be, bc, 2 times.
e: ec, 1 time. In total 6 times.
d=$\sqrt{2}$

Mini-quiz

- 3 distance computations: ab, bc, ac.
- 1 distance computations: de
- 1 distance computations: fg
- f: fg, fe, 2 times.
- g: ge, 1 time.
- d: db, de, dc, 3 times. d=$\sqrt{2}$
- b: be, bc, 2 times.
- e: ec, 1 time.
- 3+1+1+3+6=14 times of distance computations.

Run time

```
Closest-Pair(P, l, r)
if r − l < 3 then return Brute-Force-CPair(P, l, r)
q = ⌈(l+r)/2⌉
dl = Closest-Pair(P, l, q-1)
dr = Closest-Pair(P, q, r)
d = min(dl, dr)
for i = l to r do
   if P[q].x - d ≤  P[i].x  ≤ P[q].x + d then
      append P[i] to S
Sort S on y-coordinate
for j = 1 to size_of(S)-1 do
  Check if any of d(S[j],S[j+1]), ..., d(S[j],S[j+7]) is
   smaller than d, if so set d to the smallest of them
return d
```

- Running time of a divide-and-conquer algorithm can be described by a recurrence
  - Divide: Θ(1)
  - Conquer: two sub-problems, each with half-size.
  - Combine: Θ(n lg n)
  - This gives the following recurrence:

$$T(n)=\begin{cases} c & \text{if } n \leqslant 3 \\ 2T(n/2)+n\lg n & \text{otherwise} \end{cases}$$

Solving the recurrence

- Recurrence in the form of T(n)=a*T(n/b)+f(n) can be solved by the master method. Can we use the master method?
- First case: if f(n) = $O(n^{\log_b a-\epsilon})$ for some constant ε>0, then T(n) = $\Theta(n^{\log_b a})$.
- Second case: if f(n) = $\Theta(n^{\log_b a})$, then T(n) = $\Theta(n^{\log_b a}\lg n)$
- Third case: if f(n)= $\Omega(n^{\log_b a+\epsilon})$ for some constant ε>0, and the regularity condition is also satisfied, then T(n)=Θ(f(n)).
  - Regularity condition
    - *a*f(n/b)≤c*f(n) for some constant c<1and all sufficiently large n*

$$T(n)=\begin{cases} c & \text{if } n\leqslant 3 \\ 2\mathrm{T}(n/2)+n\lg n & \text{otherwise} \end{cases}$$

- No. We cannot use case 3 of the master method. Why?
- T(n)=θ(nlg^2n), see Exercise 4.6-2.
- So far, we do not get θ(nlgn) run time which we expected in the beginning.

Improving the run time?

- The problem of not getting θ(nlgn) run time is because we need to sort on y-coordinates every time that we need to combine results from two sub-problems.
- Idea: **Sort** all the points by x and y coordinates only **once**
- Before recursive calls, **partition the sorted lists** into two sorted sub-lists for the left and right halves: Θ(n)
- When combining, run through the y-sorted list once and select all points that are in the 2d strip around partition line: Θ(n)

```
Closest-Pair(X)
Sort X on the x-coordinate
for i = 1 to n do
   Y[i].x = X[i].x
   Y[i].y = X[i].y
   Y[i].p = i
Sort Y on the y-coordinate
Closest-Pair-R(X, Y, 1, n)
```

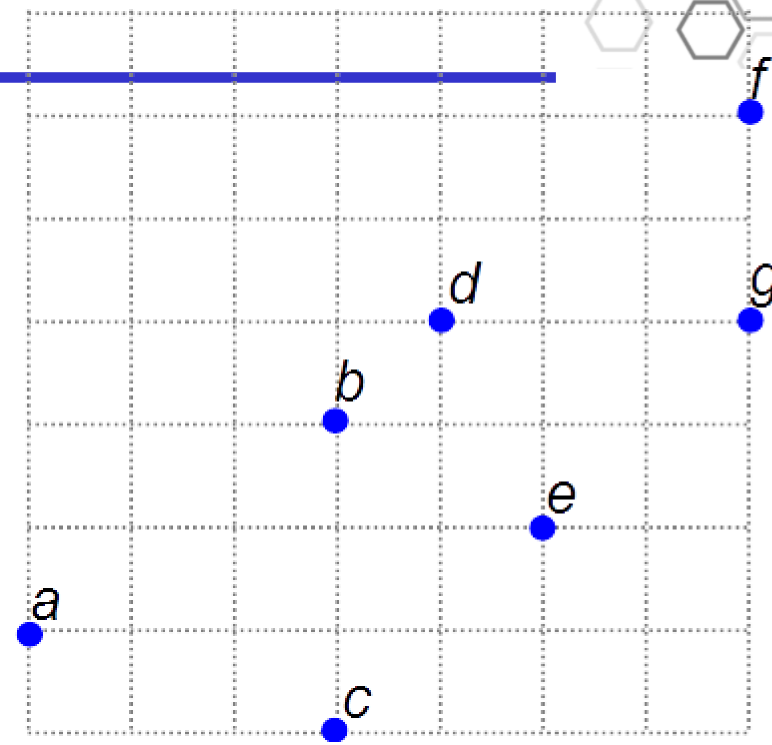
```
Closest-Pair(X)
Sort X on the x-coordinate
for i = 1 to n do
    Y[i].x = X[i].x
    Y[i].y = X[i].y
    Y[i].p = i
Sort Y on the y-coordinate
Closest-Pair-R(X, Y, 1, n)
```

X

1	2	3	4	5	6	7
a	b	c	d	e	f	g

Y

1	2	3	4	5	6	7
f	d	g	b	e	a	c
6	4	7	2	5	1	3



Pseudo code

```
Closest-Pair-R(X, Y, l, r)
// Requires: ∀i∈[1..r-l+1]: l ≤ Y[i].p ≤ r
if r − l < 3 then return Brute-Force-CPair(X, l, r)
q = ⌈(l+r)/2⌉
for i = 1 to r-l+1 do
    if Y[i].p < q then
        append Y[i] to Yl
    else
        append Y[i] to Yr
dl = Closest-Pair-R(X, Yl, l, q-1)
dr = Closest-Pair-R(X, Yr, q, r)
d = min(dl, dr)
for i = 1 to r-l+1 do
    if X[q].x − d ≤ Y[i].x ≤ X[q].x + d then
        append Y[i] to S
Sort S on y-coordinate
for j = 1 to size_of(S)-1 do
   Check if any of d(S[j],S[j+1]), ..., d(S[j],S[j+7]) is
    smaller than d, if so set d to the smallest of them
return d
```

Points lying to the left go to Yl.
Points lying to the right go to Yr.
Both Yl and Yr are still sorted on Y-coordinates.

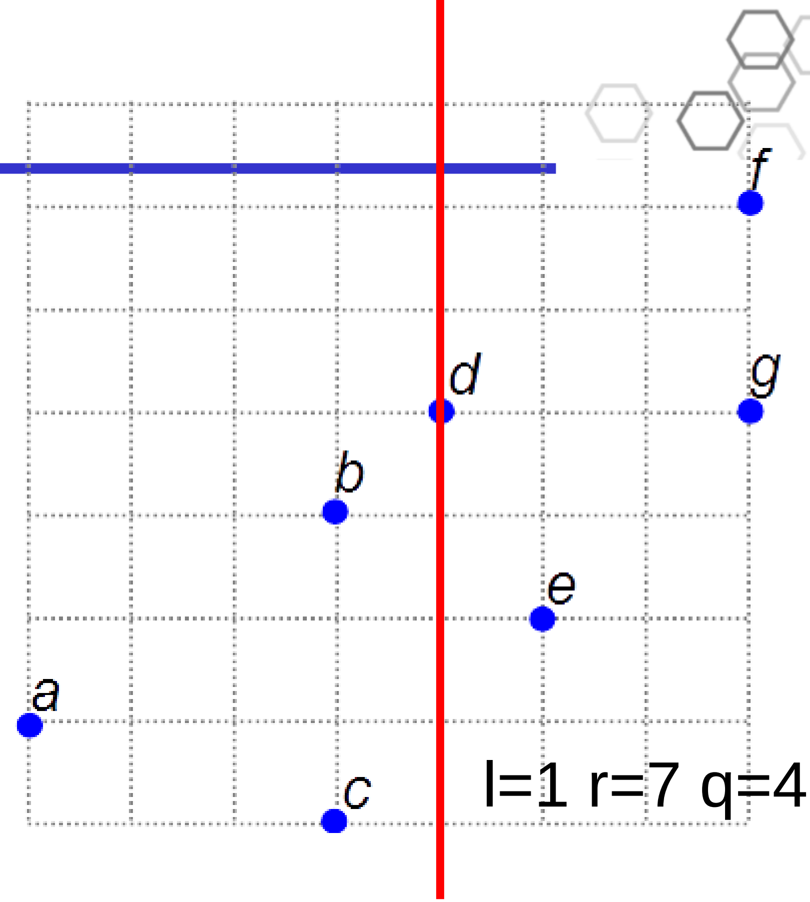
Filtering points lying outside the strip while maintaining the order on y-coordinates.

X

1	2	3	4	5	6	7
a	b	c	d	e	f	g

Y

1	2	3	4	5	6	7
f	d	g	b	e	a	c
6	4	7	2	5	1	3




```
q = ⌈(l+r)/2⌉
for i = 1 to r-l+1 do
    if Y[i].p < q then
        append Y[i] to Yl
    else
        append Y[i] to Yr
```

Yl

1	2	3
b	a	c
2	1	3

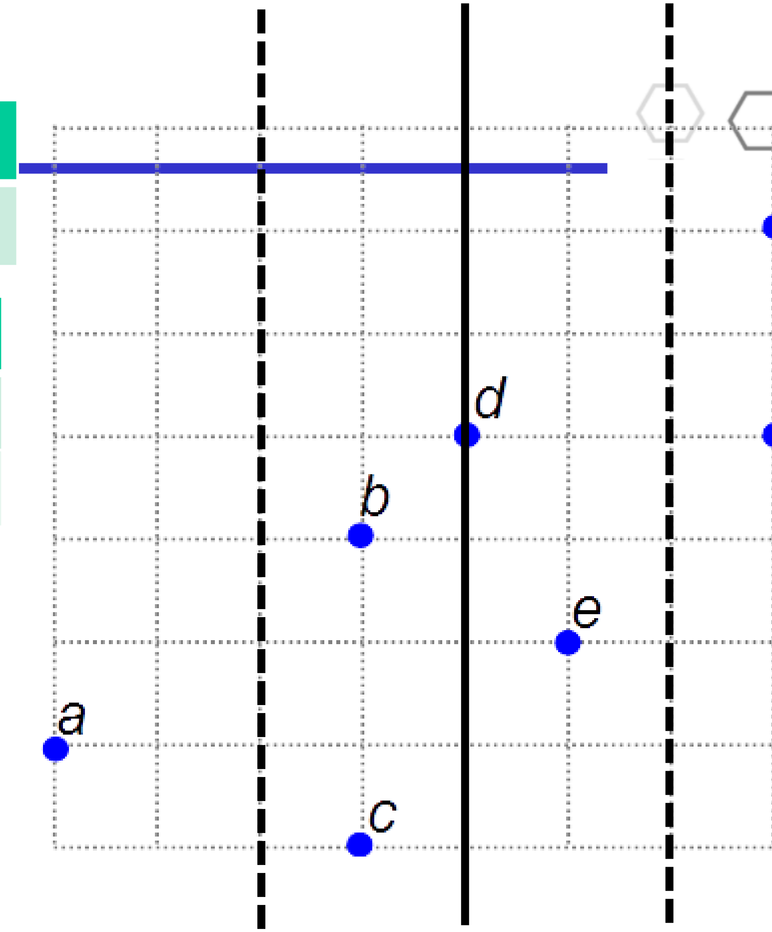
Yr

1	2	3	4
f	d	g	e
6	4	7	5

X						
1	2	3	4	5	6	7
a	b	c	d	e	f	g

Y						
1	2	3	4	5	6	7
f	d	g	b	e	a	c
6	4	7	2	5	1	3

S: d
d, b
d, b, e
d, b, e, c



l=1 r=7 q=4

```
for i = 1 to r-l+1 do
    if X[q].x - d ≤ Y[i].x ≤ X[q].x + d then
        append Y[i] to S
Sort S on y coordinate
for j = 1 to size_of(S)-1 do
   Check if any of d(S[j],S[j+1]), ..., d(S[j],S[j+7]) is
    smaller than d, if so set d to the smallest of them
```

Run time

```
Closest-Pair-R(X, Y, l, r)
// Requires: ∀i∈[1..r-l+1]: l ≤ Y[i].p ≤ r
if r - l < 3 then return Brute-Force-CPair(X, l, r)
q = ⌈(l+r)/2⌉
for i = 1 to r-l+1 do
    if Y[i].p < q then
        append Y[i] to Yl
    else
        append Y[i] to Yr
dl = Closest-Pair-R(X, Yl, l, q-1)
dr = Closest-Pair-R(X, Yr, q, r)
d = min(dl, dr)
for i = 1 to r-l+1 do
    if X[q].x - d ≤  Y[i].x  ≤ X[q].x + d then
        append Y[i] to S
```

~~Sort S on y coordinate~~

```
for j = 1 to size_of(S)-1 do
   Check if any of d(S[j],S[j+1]), ..., d(S[j],S[j+7]) is
    smaller than d, if so set d to the smallest of them
return d
```

Divide: n

Conquer: 2 sub-problems, each with half size

Combine: n

Improved run time

$$T(n) = \begin{cases} c & \text{if } n \leqslant 3 \\ 2\mathrm{T}(n/2) + n & \text{otherwise} \end{cases}$$

- Master method:
- Second case: if f(n) = $\Theta(n^{\log_b a})$, then T(n) = $\Theta(n^{\log_b a} \lg n)$
- So, θ(nlgn)

- The price we pay is the additional storage for Y.

Agenda

- Output sensitive algorithms
  - Convex hull: Jarvis's march
- Divide-and-conquer algorithms
  - Finding the closest pair of points
  - Convex hull

Convex hull: Divide-and-conquer

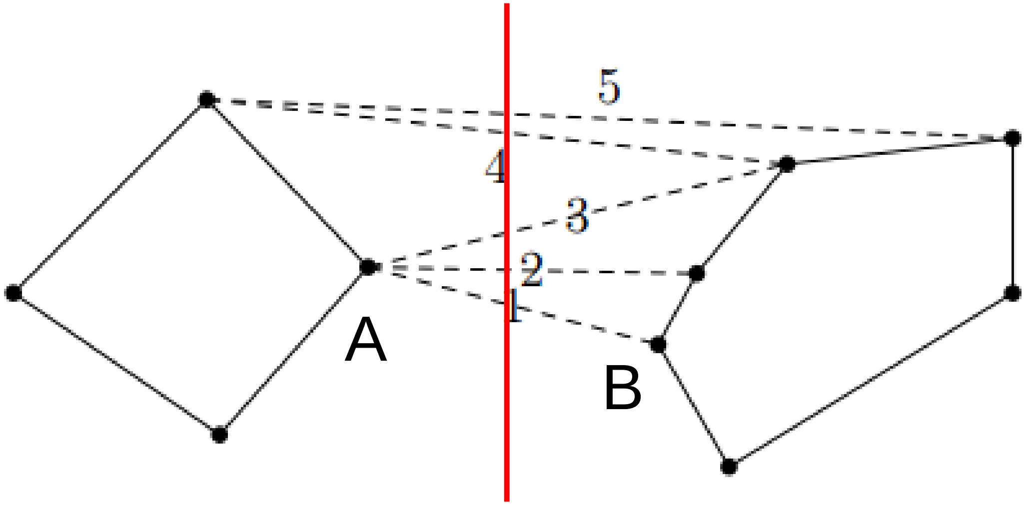
- What is a trivial problem and how do we solve it?
- How do we divide the problem into sub-problems?
- How do we combine solutions to sub-problems?
- What is the running time?

Convex hull: Divide-and-conquer

- What is a trivial problem and how do we solve it?
  - When we have only less than 3 points.
- How do we divide the problem into sub-problems?
  - Sort based on x-coordinate, and split into half.
- How do we combine solutions to sub-problems?
  - Most challenging part.
- What is the running time?
  - Hopefully, nlgn

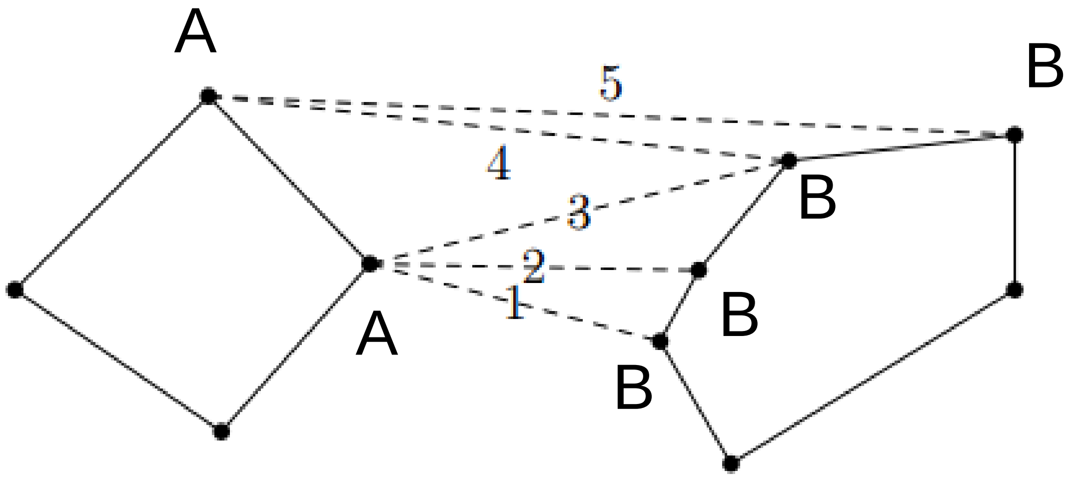
Combining two convex hulls

- A bridge is a segment connecting one point from the left partition and one point from the right partition, which does not intersects any edge of the left and right convex hulls.
- We want to identify the upper bridge and the lower bridge.
- Start with any bridge. Try to move up/down the bridge.
- Start from the bridge that connects the rightmost point A from the left partition and the leftmost point B from the right partition.



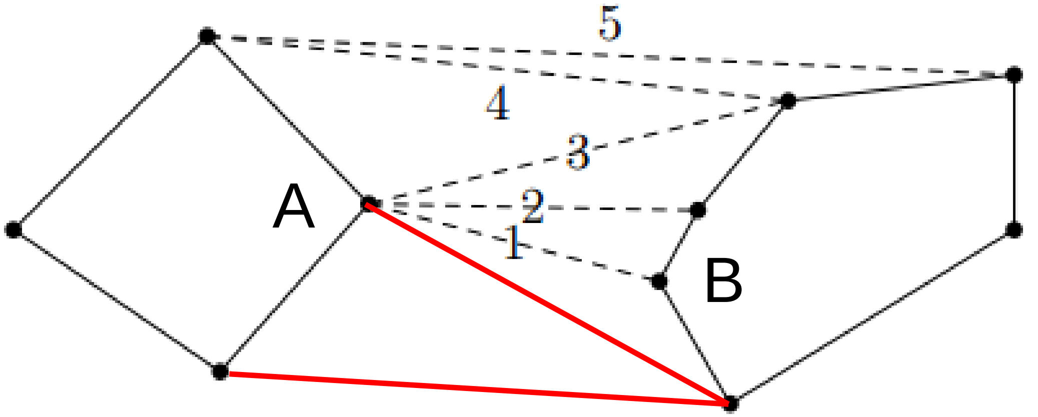
Upper bridge

- Keeping the left end of the bridge fixed, see if the right end can be raised.
  - Check the next vertex on the right polygon going clockwise, and see whether that would be a bridge.
- Otherwise, see if the left end can be raised while the right end remains fixed.
  - Check the next vertex on the left polygon going counter-clockwise, and see whether that would be a bridge.



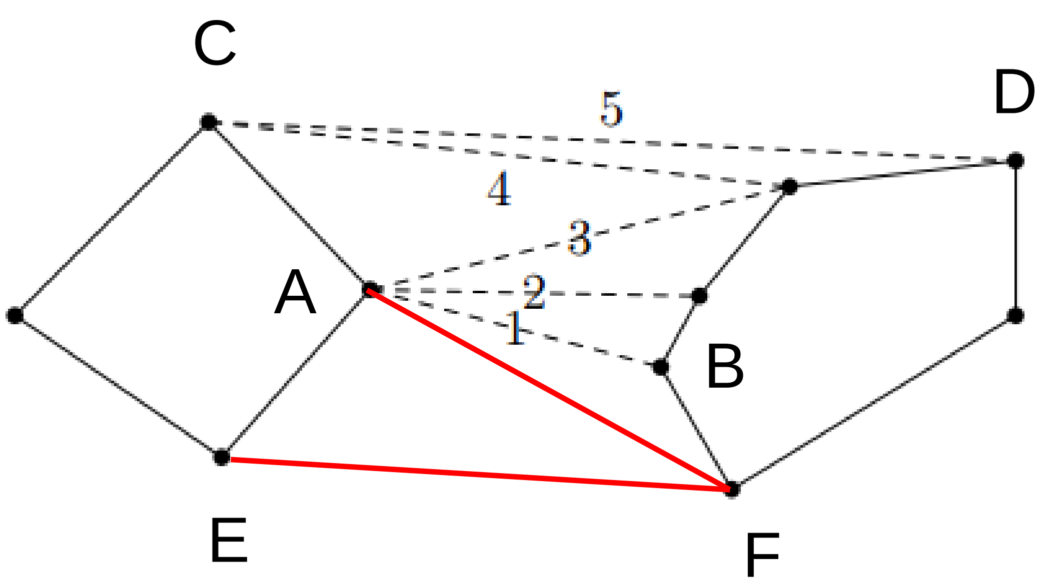
Lower bridge

- Keeping the left end of the bridge fixed, see if the right end can be lowered.
  - Check the next vertex on the right polygon going counter-clockwise, and see whether that would be a bridge.
- Otherwise, see if the left end can be lowered while the right end remains fixed.
  - Check the next vertex on the left polygon going clockwise, and see whether that would be a bridge.



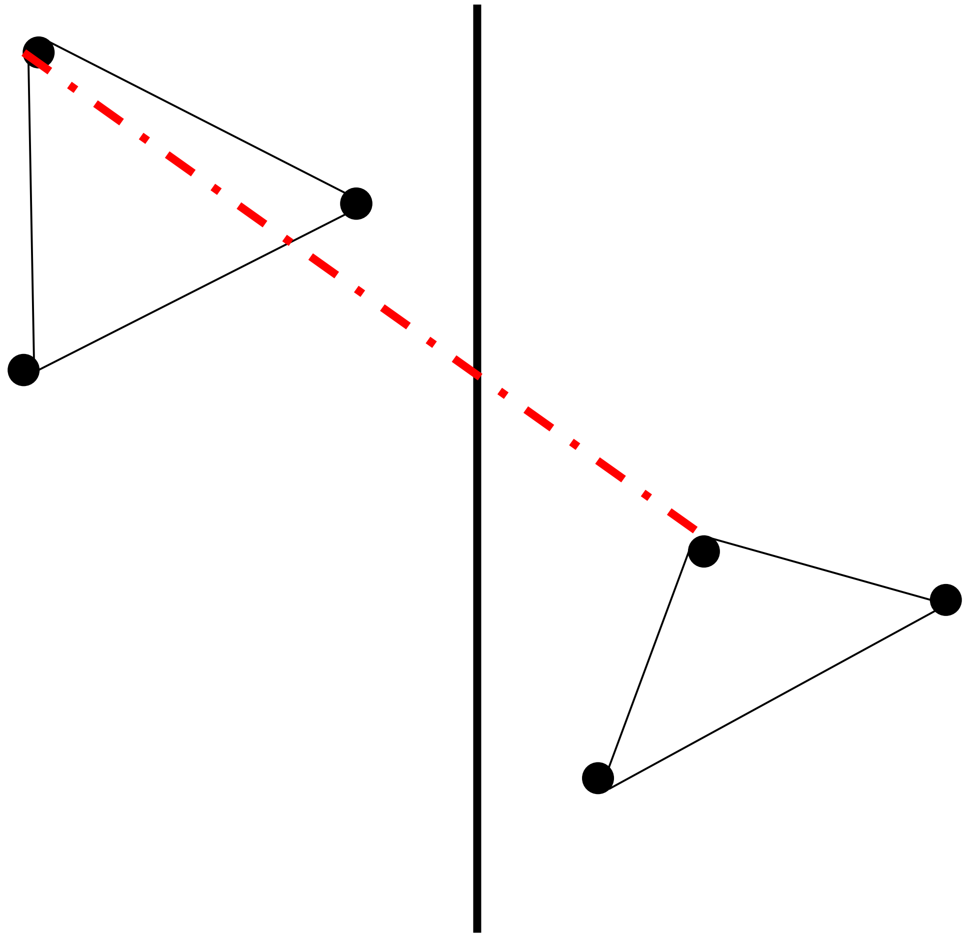
Combing two convex hulls

- Left and right end points of the upper edge: C, D
- Left and right end points of the lower edge: E, F
- Convex hull consists of the following points:
  - Left convex hull: the vertices from E to C clockwise.
  - Right convex hull: the vertices from D to F clockwise.



Upper bridge

- Can you simply connect the points with the highest y-coordinates for each of the two convex hulls to get the upper bridge?



Run time

- What is the complexity for combining two convex hulls.
  - Divide: constant time, if ordered already on x-coordinates.
  - Conquer: 2 sub-problems, each sub-problem is of half size.
  - Combine:
    - We can find the upper and lower bridges in linear time, since all the points are at most checked once: θ(n).
    - So, θ(n) for combining. We get the following recurrence.

$$T(n) = \begin{cases} c & \text{if } n \leqslant 3 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

  - Thus, θ(nlgn)
  - Exercise 3 asks you to write pseudo code and conduct detailed complexity analysis.

ILO of Lecture 7

- Output sensitive and divide-and-conquer algs
  - to understand the concept of ***output sensitive*** algorithms;
  - to be able to apply the divide-and-conquer algorithm design technique to geometric problems;
  - to remember how recurrences are used to analyze the divide-and-conquer algorithms;
  - to understand and be able to analyze the Jarvi's march algorithm and the divide-and-conquer algorithms for finding a closest pair and for finding the convex hull.

Next lecture

- Computational Geometry Algorithms: Range Searching