Advanced Algorithm Design and Analysis (Lecture 10)

SW5 fall 2004

Simonas Šaltenis

E1-215b

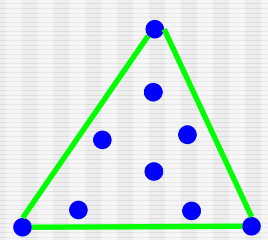
simas@cs.aau.dk

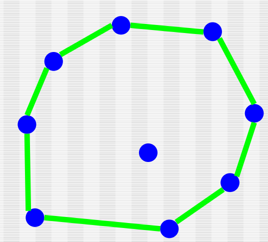
Computational geometry

- Main goals of the lecture:
  - *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 **Jarvis's march** and the divide-and-conquer **closest-pair** algorithms.*

Size of the output

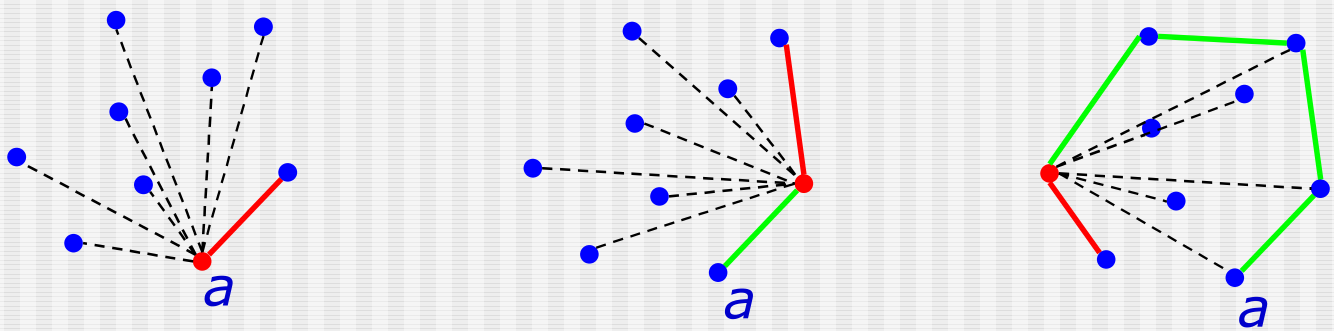
- *In computational geometry, the size of an algorithm's **output** may differ/depend on the **input**.*
  - Line-intersection problem vs. convex-hull problem
  - *Observation*: Graham's scan running time depends only on the size of the **input** – it is independent of the size of the **output**





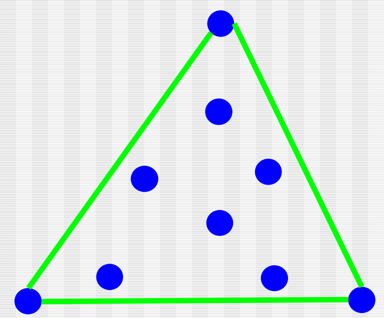
Gift wrapping

- *Would be nice to have an algorithm that runs fast if the convex hull is small*
  - Idea: ***gift wrapping*** (a.k.a **Jarvis's march**)
    - 1. Start with the lowest point *a*, include it in the convex hull
    - 2. The **next** point in the convex hull has to be in the clockwise direction with respect to all other points looking from the **current** point on the convex hull
    - 3. Repeat 2. until *a* is reached.



Jarvis's march

- *How many cross products are computed for this example*?



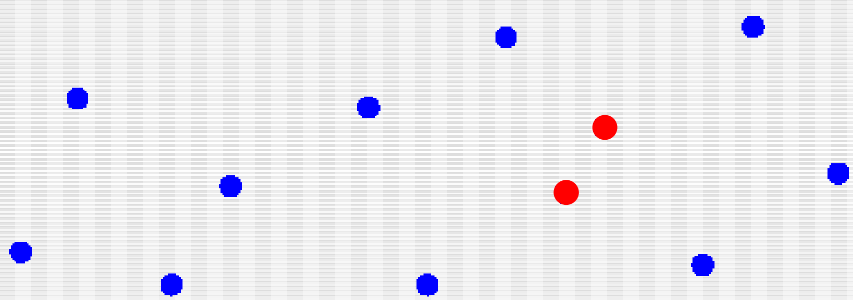
- The running time of Jarvis's march:
  - Find lowest point – $O(n)$
  - For each vertex in the convex hull: n–1 cross-product computations
  - Total: $\boldsymbol{O(nh)}$, where h is the number of vertices in the convex hull

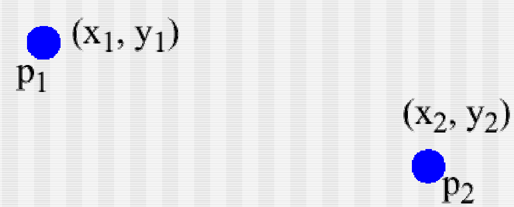
Output-sensitive algorithms

- ***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*?
  - The asymptotically optimal output-sensitive algorithm of Kirkpatrick and Seidel runs in $O(n \lg h)$

Closest-pair problem

- *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)
  - *What is the brute-force algorithm and it's running time*?





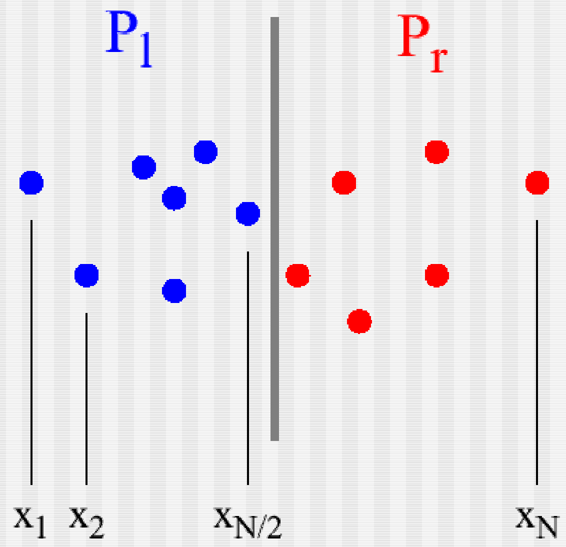
$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Steps of Divide-and-Conquer

- *What are the steps of a divide-and-conquer algorithm*?
  - If trivial (small), solve it “brute force”
  - Else
    - **1.divide** into a number of sub-problems
    - **2.solve** each sub-problem recursively
    - **3.combine** solutions to sub-problems

Dividing into sub-problems

- *How do we divide into sub-problems*?
  - *Idea*: Sort on x-coordinate, and divide into left and right parts:
    - p_1 p_2 ... $p_{n/2}$ ... $p_{n/2+1}$ ... p_n



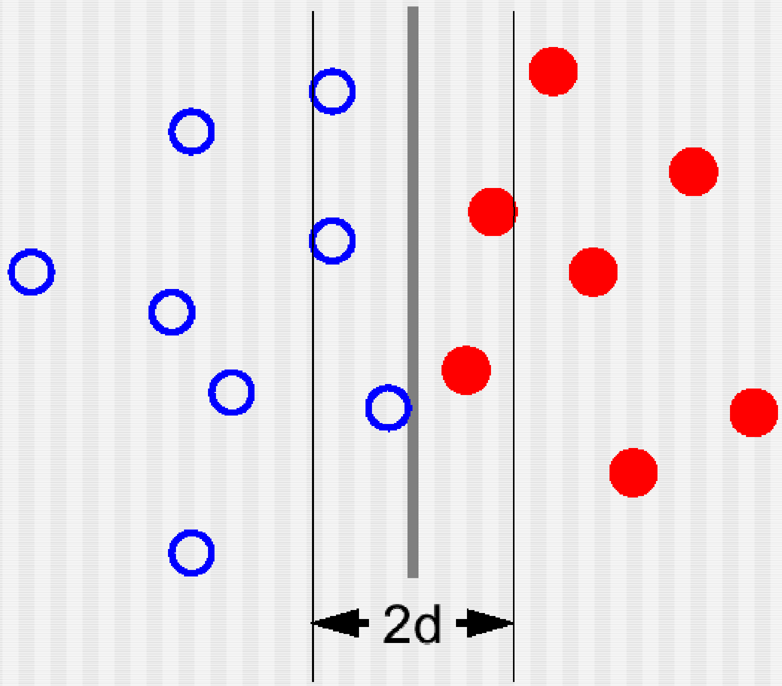
  - Solve recursively the left sub-problem P_l (closest-pair distance d_l) and the right sub-problem P_r (distance d_r)

Combining two solutions

- *How do we combine two solutions to sub-problems*?
  - Let $d = \min\{d_l, d_r\}$
  - *Observation* 1: 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 **2*d*** around the dividing line!
    - Otherwise the points would be more than *d* units apart.

Combining two solutions

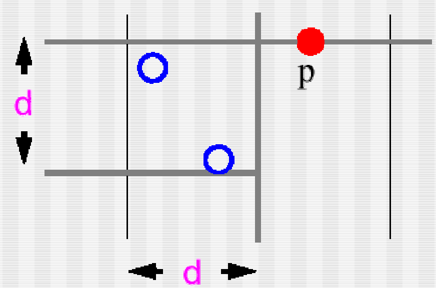
- *Combining solutions*: Finding the closest pair (○,●) in a strip of width 2*d*, knowing that no (○,○) or (●,●) pair is closer than *d*

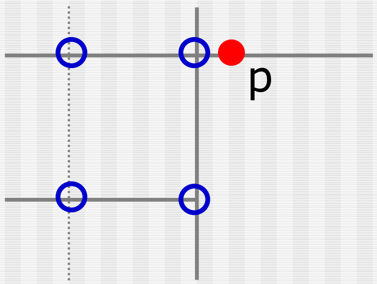


Combining Two Solutions

- *Do we have to check all pairs of points in the strip*?
  - *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*?
  - In the *d*×*d* square:

$$y(p) - d \le y(q) \le y(p)$$

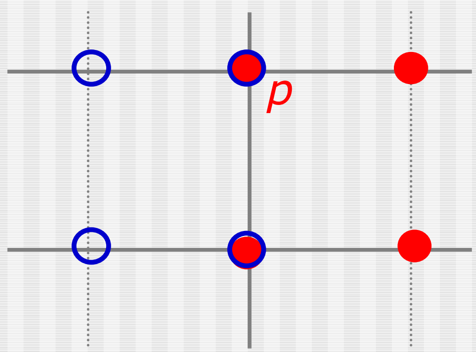




- *How many points can there be in this square*?
  - At most 4!

Combining two solutions

- Algorithm for checking the strip:
  - Sort all the points in the strip on the *y*-coordinate
  - For each point *p* only **7** points ahead of it in the order have to be checked to see if any of them is closer to *p* than *d*



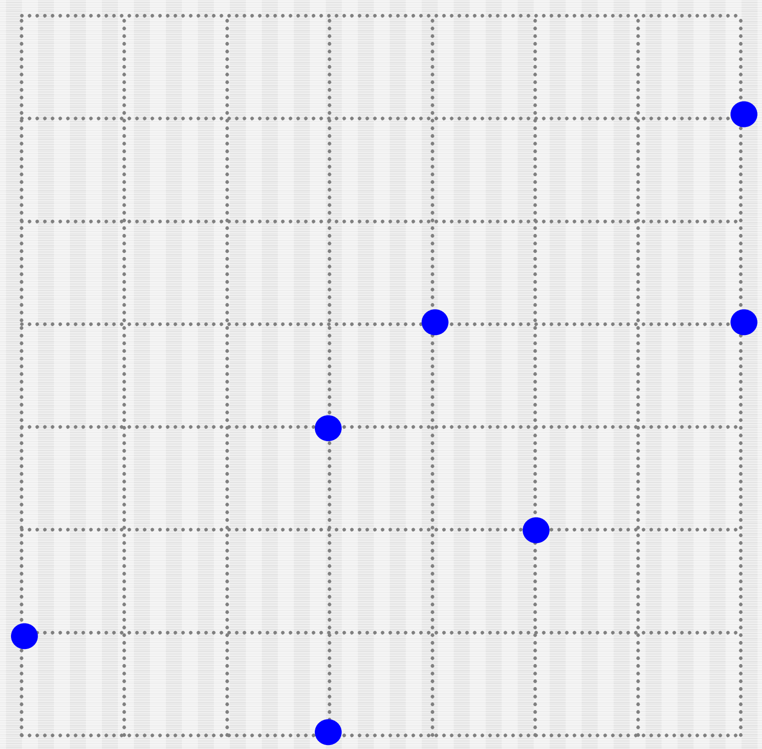
Pseudocode

- *What is the trivial problem*?
  - *That is – when do we stop recursion*?

```
Closest-Pair(P, l, r)
// First call: an array P of points sorted on x-coordinate, 1, n
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
```

Example

- *How many distance computations are done in this example*?



Running time

- *What is the running time of this algorithm*?
  - Running time of a divide-and-conquer algorithm can be described by a recurrence
  - Divide = $O(1)$
  - Combine = $O(n \lg n)$
  - This gives the following recurrence:

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

  - Total running time: $O(n \log^2 n)$
    - Better than brute force, but…

Improving the running time

- *How can we improve the running time of the algorithm*?
  - *Idea*: **Sort** all the points by x and y coordinate **once**
  - Before recursive calls, **partition the sorted lists** into two sorted sublists for the left and right halves: $O(n)$
  - When combining, run through the y-sorted list once and select all points that are in a 2d strip around partition line: $O(n)$
- *How does the new recurrence look like and what is its solution*?

Conclusion

- The closest pair can be found in $O(n \log n)$ time with divide-and-conquer algorithm
  - *Plane-sweep* algorithm with the same asymptotic running time exists
  - This is asymptotically optimal

Exercise: Convex-hull

- *Let's find the convex-hull using divide-and-conquer*
  - What is a trivial problem and how we solve it?
  - How do we divide the problem into sub-problems?
  - How do we combine solutions to sub-problems?

Repeated Substitution

- Solving recurrences by repeated substitution:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= 2T\left(n/2\right) + n && \text{substitute} \\
&= 2\left(2T\left(n/4\right) + n/2\right) + n && \text{expand} \\
&= 2^2 T(n/4) + 2n && \text{substitute} \\
&= 2^2(2T(n/8) + n/4) + 2n && \text{expand} \\
&= 2^3 T(n/8) + 3n && \text{observe the pattern} \\
T(n) &= 2^i T(n/2^i) + in \\
&= 2^{\lg n} T(n/n) + n \lg n = n + n \lg n
\end{aligned}$$

Repeated Substitution Method

- The procedure is straightforward:
  - Substitute
  - Expand
  - Substitute
  - Expand
  - …
  - Observe a pattern and write how your expression looks after the i-th substitution
  - Find out what the value of i (e.g., lg n) should be to get the base case of the recurrence (say $T(1)$)
  - Insert the value of $T(1)$ and the expression of i into your expression