Algorithms

https://www.youtube.com/watch?v=ldHA8UcQI9Q&t=386s

Algorithmic Techniques

We discuss three basic and broad algorithmic techniques in this course.

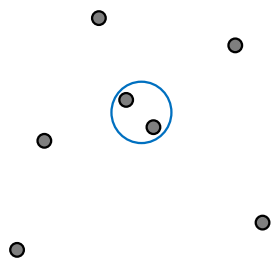
- Greedy
- Divide and conquer
- Dynamic Programming

We will start with the "divide and conquer" now.

Closest Pair of Points

Problem. Given a set of n points in the plane, find the pair of points that are the closest to each other.

Example.



Trivial Algo. Compute the distance between each pair of points.
Report the pair with the smallest distance. *$O(n^2)$ time.*

Idea. Sort the points by the x-coordinates and by the y coordinates.

The closest pair of points must be close in at least one of the lists.

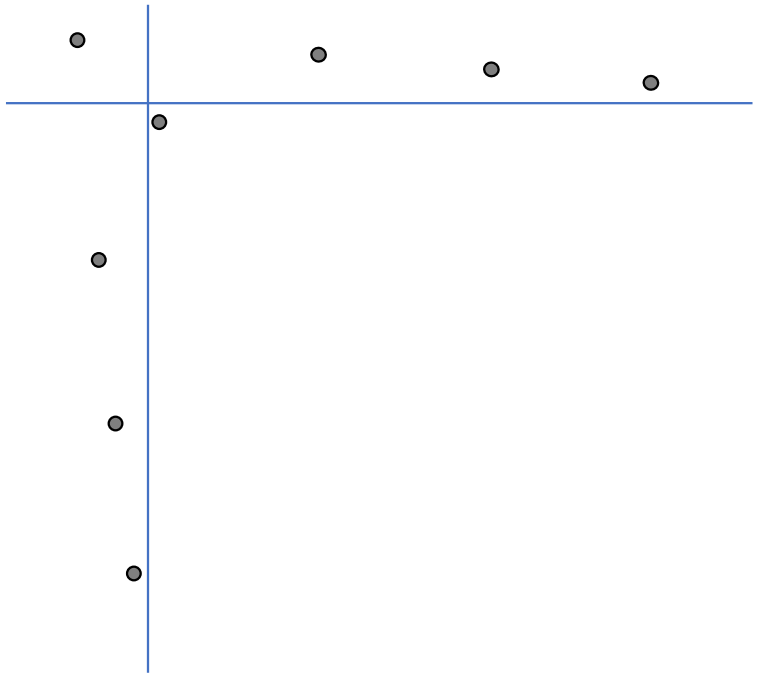
Go over both sorted lists and check each pair of "nearby" points.

Does this work?

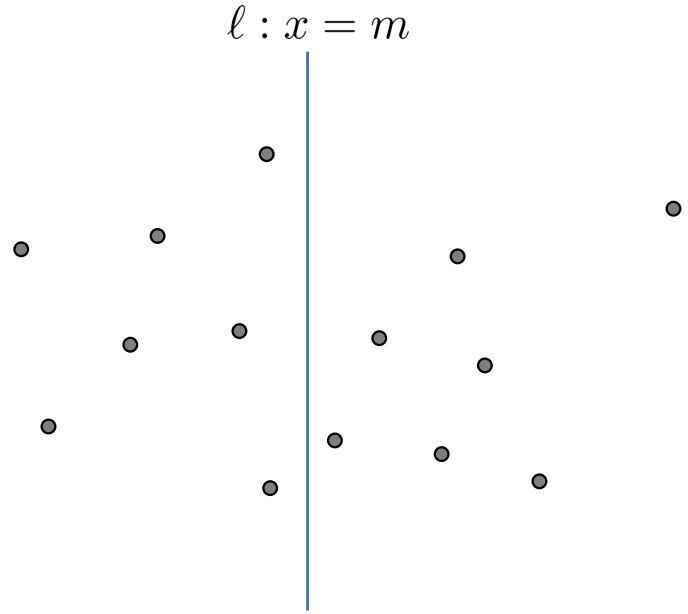
Closest Pair of Points

This is in fact a wrong idea.

The closest pair of points need not be close in either sorted order.



Closest Pair of Points

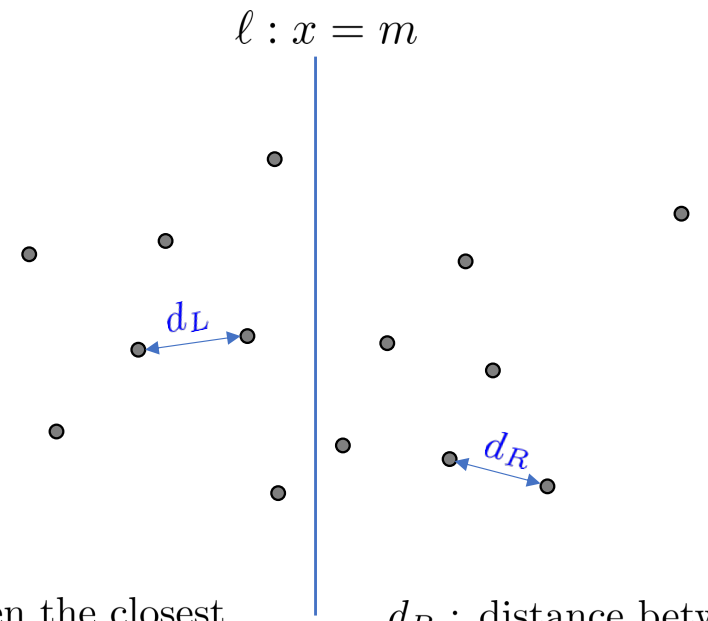


Idea. Split the set of points into two using a vertical line ℓ at the median x coordinate m.

Recursively find the closest pair on either side.

What if the closest pair is across the line?

Closest Pair of Points



d_L : distance between the closest pair of points on the left side

d_R : distance between the closest pair of points on the right side

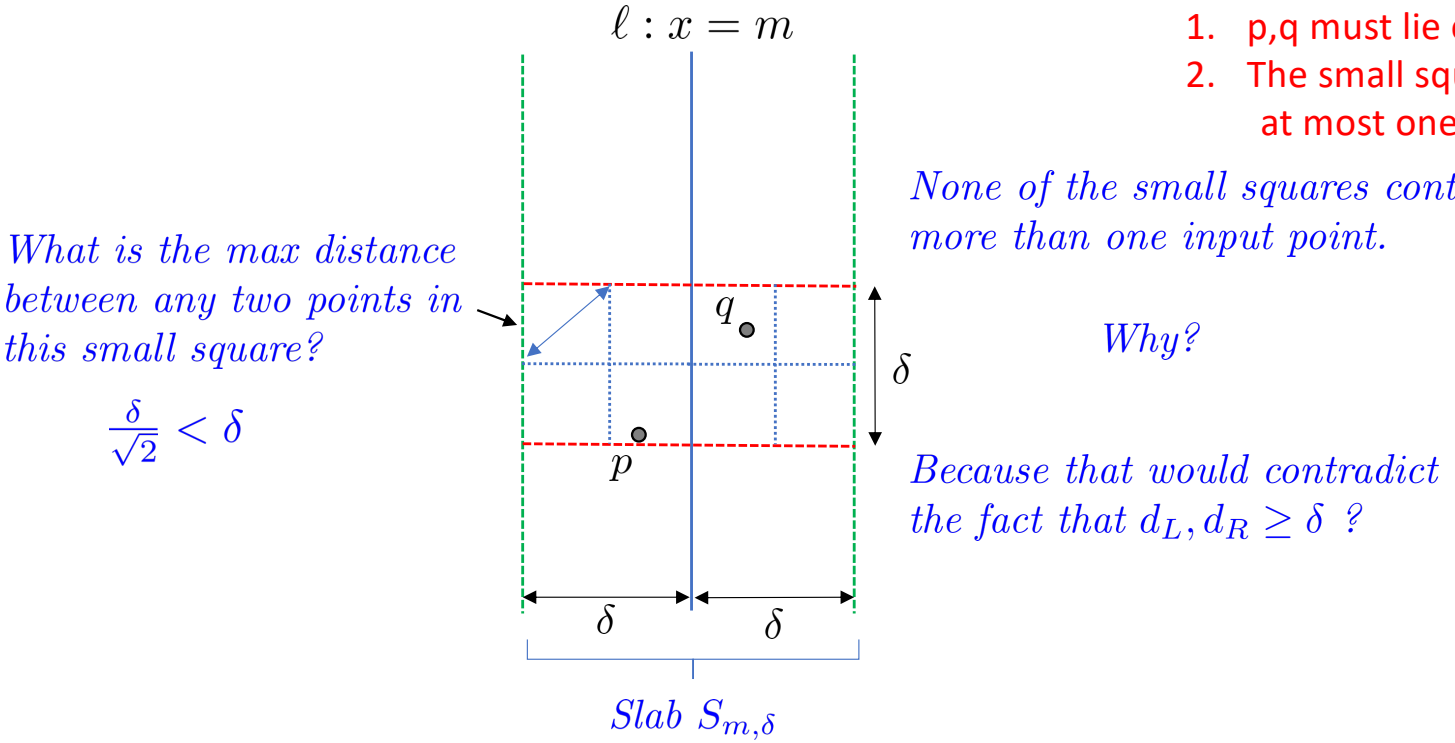
$\delta = \min(d_L, d_R)$

We need to find a closest pair of points (p, q) that lie on different sides of ℓ, **if** $\text{dist}(p, q) < \delta$.

Note: we don't need to output anything if there is no pair (p, q) s.t. $dist(p, q) < \delta$.

Closest Pair of Points

Summary: if p,q are across ℓ
Then to be the closest:
1. p,q must lie each within δ of ℓ
2. The small squares created contain at most one point



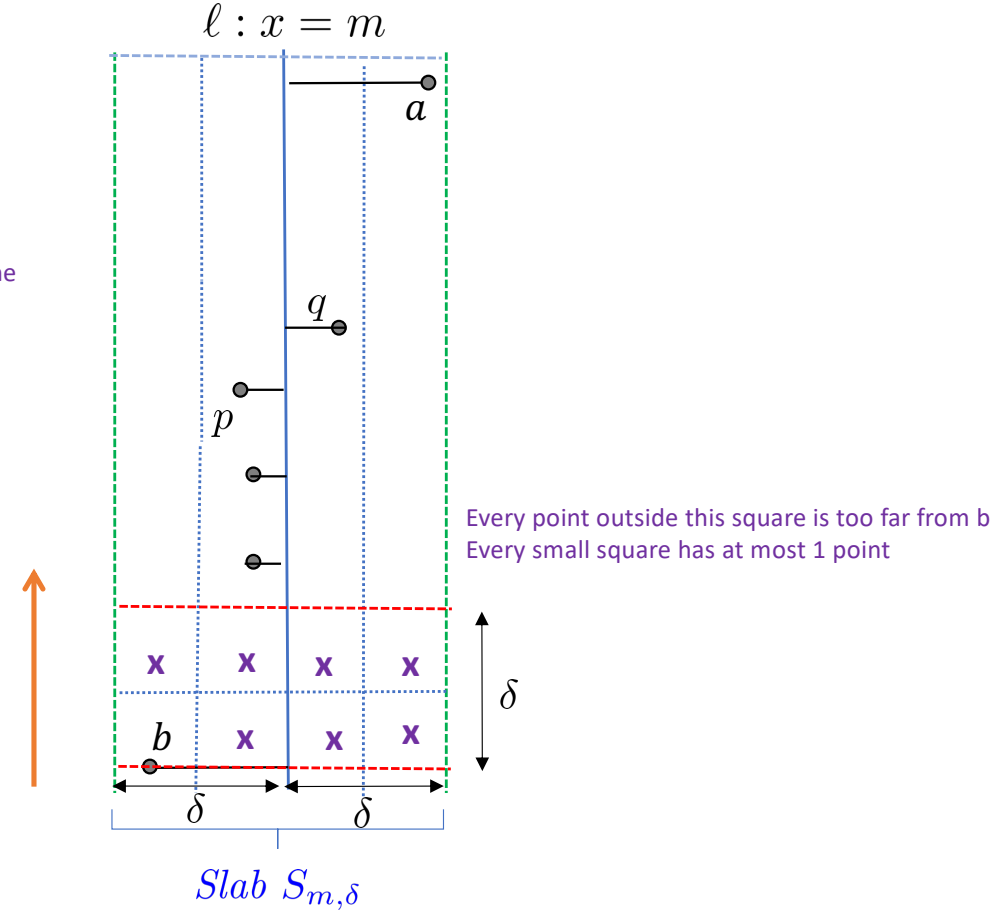
Imagine the closest pair among all pairs of points thet lie on different sides of ℓ and suppose that they are at a distance $< \delta$ of each other.

Both points must be within a distance δ of ℓ.

The vertical distance between the points is at most δ.

Closest Pair of Points

Sort the points in the slab based on their y value
Do we have to find the mutual distance for all pair of points in the slab?
Too complex!! we can do better
Recall that every ”small square” has at most one point!



If we sort the points in the slab $S_{m,\delta}$ vertically (i.e, by y coordinate), then p and q lie within 7 positions of each other (in the sorted order).

So, after sorting, we can find such a pair (p, q) in $O(n)$ time.

Closest Pair of Points

Input. n points $p_1, \cdots, p_n$ where p_i has coordinates (x_i, y_i).

Goal. Find the closest pair of points. *Assume distinct x and y coordinates.*

Algorithm.

$m = \text{median}(x_1, \cdots, x_n)$. *$O(n \log n)$ time by sorting.* BOTTLENECK

$L = \{p_i : x_i \leq m\}$, $R = \{p_i : x_i > m\}$. *$O(n)$ time.*

Recursion

- Find the closest pair of points (p_L, q_L) in L and set $d_L = \text{dist}(p_L, q_L)$. $T(\lceil n/2 \rceil)$
- Find the closest pair of points (p_R, q_R) in R and set $d_R = \text{dist}(p_R, q_R)$. $T(\lfloor n/2 \rfloor)$

$\delta = \min(d_L, d_R)$. *$O(1)$ time.*

$S = \{p_i : m - \delta \leq x_i \leq m + \delta\}$. *Set of points in the slab $S_{m,\delta}$. $O(n)$ time.*

Sort S w.r.t y coordinates. *$O(n \log n)$ time.* BOTTLENECK

Find the closest pair (p_S, q_S) among all pairs of points that lie within 7 positions of each other in S. *$O(n)$ time.*

Report the closest pair among $\{(p_L, q_L), (p_R, q_R), (p_S, q_S)\}$. *$O(1)$ time.*

Closest Pair of Points

$m = \text{median}(x_1, \cdots, x_n)$. *$O(n \log n)$ time by sorting.* BOTTLENECK

Sort S w.r.t y coordinate. *$O(n \log n)$ time.* BOTTLENECK

These bottlenecks can be easily removed if we require that the input point set is provided as two sorted lists – one sorted by x coordinates and another sorted by y coordinates.

Initially this can be done by sorting the points along x and y coordinates.

For recursing, we can obtain the sorted lists for the left and right sides in linear time by going over the lists and just keeping the points that go to one of the sides.

Similarly, to obtain the points in a slab $S_{m,\delta}$ in sorted order by y-coordinate, we go over the list of all points sorted by y coordinate and just keep the points lying in the slab.

Closest Pair of Points

The initial sorting by x and y coordinates takes $O(n \log n)$ time.

Let $T(n)$ denote the time taken by the algorithm excluding the time taken for the initial sorting.

Then we get the following recurrence relation:

$$\left.\begin{array}{l} T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + c\,n. \\ T(n) = c \text{ for } n \leq 4. \end{array}\right\} \implies T(n) = O(n \log n).$$

Overall time (including initial sorting): $O(n \log n)$.

Algorithmic Techniques

We discuss three basic and broad algorithmic techniques in this course.

- Greedy
- Divide and conquer
- Dynamic Programming

We will start with the "divide and conquer" now.

Dynamic Programming

Fibonnaci numbers:
$1, 1, 2, 3, 5, \ldots \ldots$
$f(n) = f(n-1) + f(n-2), f(0) = f(1) = 1$

Dynamic Programming

Consider the following code for computing fibonnaci numbers:

```
def fib(n):
  if n < 2:
     return 1
  else:
     return fib(n-1) + fib(n-2)
```

Recurrence relation for running time:

$$T(n) = T(n-1) + T(n-2) + O(1),\ T(1) = O(1)$$

$T(n)$ is exponential in n.

Observation. We compute `fib`(k) for the same k several times!

Why not just save it when we compute it?

Dynamic Programming

Modified Algorithm.

```
M = [1, 1, ∞, ··· , ∞]
```

array of size $n+1$ where $M[0] = M[1] = 1$, and for $i = 2..n$, $M[i] = \infty$ indicating that `fib`(i) has not yet been computed.

```
def fib(n):
    if n < 2:  return 1
    if M[n-1] == ∞:  M[n-1] = fib(n-1)
    if M[n-2] == ∞:  M[n-2] = fib(n-2)
    M[n] = M[n-1] + M[n-2]
    return M[n]
```

$O(n)$ time algorithm

Takes more space but drastically reduces the running time

The idea of saving results is called ***memoization***.

A similar idea can be used in many problems.