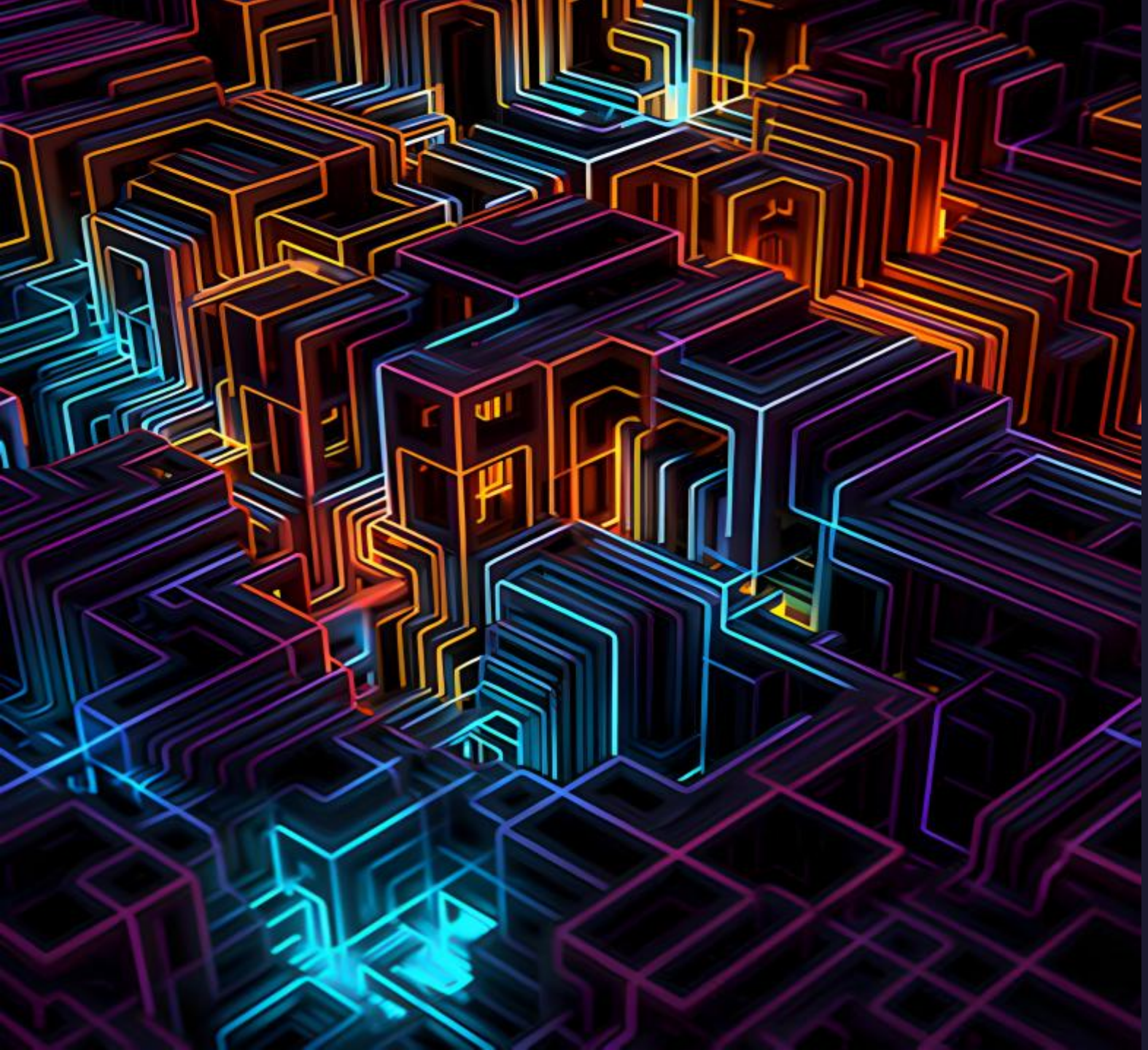


Geometría computacional

Análisis y diseño de algoritmos avanzados

Dra. Valentina Narváez Terán

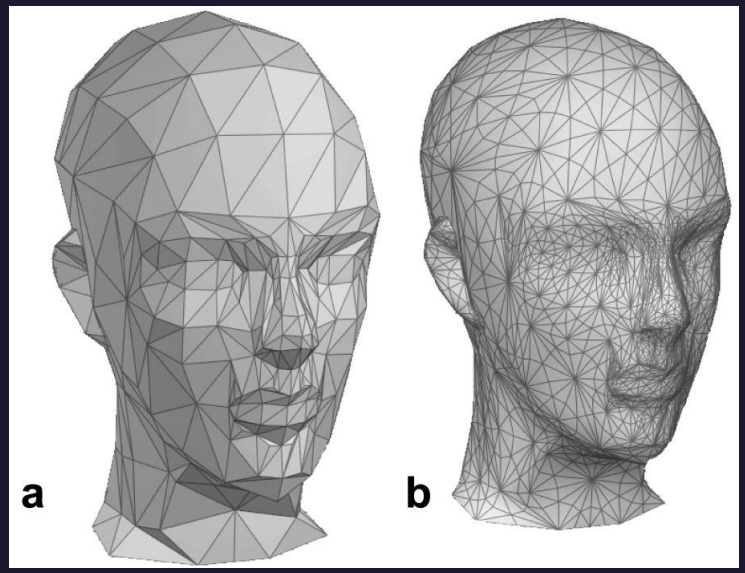
Tecnológico de Monterrey

Geometría computacional

Es una rama de la computación dedicada a resolver problemas geométricos

Relacionados con:

- Puntos (¿cual problema ya conoces?)
- Líneas y segmentos de línea
- Polígonos
- Distancias, intersecciones, inclusiones, rotaciones…



Aplicaciones:

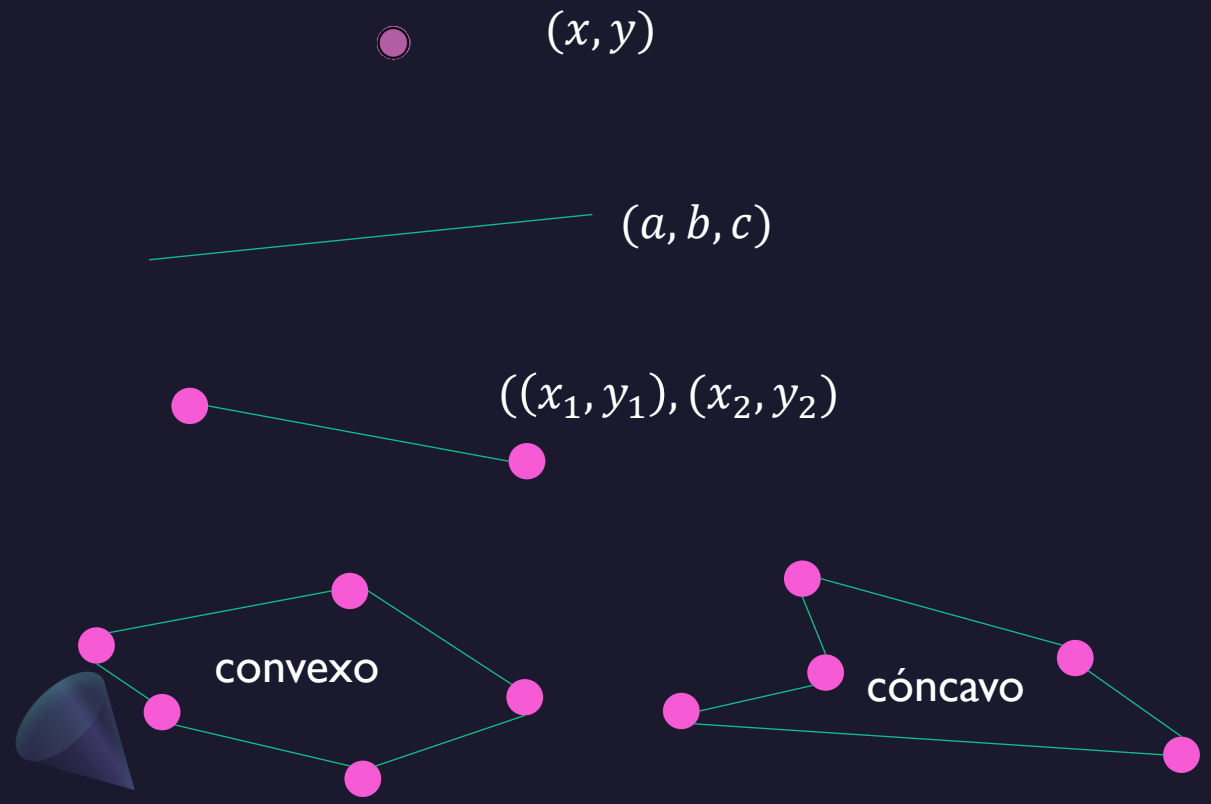
Gráficos computacionales, modelado y renderizado, impresión 3D, videojuegos (detección de colisiones, hitboxes)

Robótica, visión por computadora, reconocimiento de imágenes, clustering en machine learning

Geometría computacional : definiciones

- Punto: en 2D, un par ordenado (x, y) (se puede generalizar a una tupla para n dimensiones)

- Línea recta: una sucesión infinita de puntos, representada por la ecuación $ax + by + c = 0$
- Segmento de recta: una de una línea comprendida entre 2 puntos
- Polígono: conjunto de vértices conectados por segmentos de recta que no se cruzan y que forman un ciclo



Representados como conjuntos de segmentos de recta

Problema: closest-pair

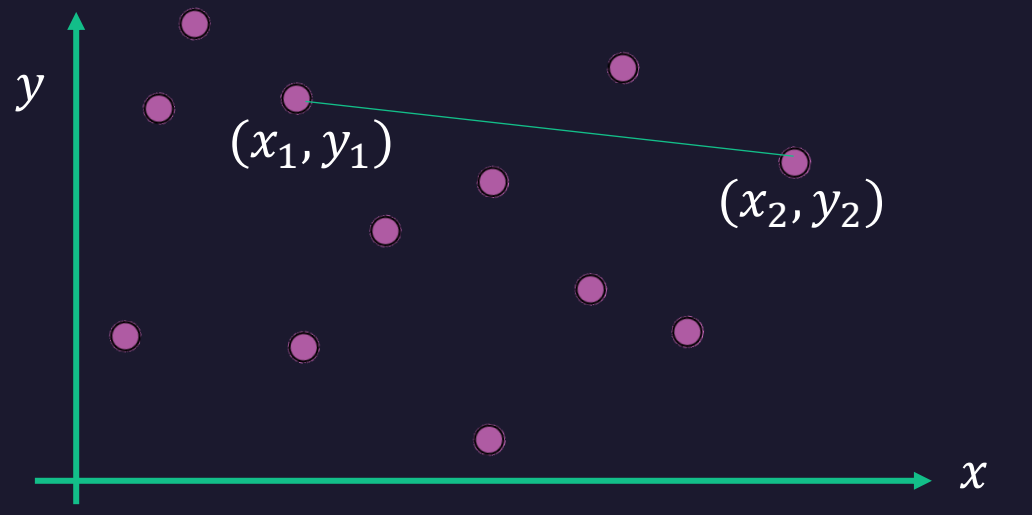
El problema: Hay n puntos con d dimensiones
¿Cuál es el par de puntos con distancia más cercana entre sí?

Algoritmo de fuerza bruta:
Calcular la distancia euclidiana entre todos los pares de puntos. Encontrar el mínimo de las distancias

Complejidad: cuadrática $O(n^2)$

Distancia euclidiana entre 2 puntos

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



La distancia euclidiana es la longitud del segmento de recta cuyos extremos son estos 2 puntos

Closest-pair: ¿un algoritmo mas eficiente?

Imaginemos una versión más simple del problema:

Si los puntos solo tuvieran la coordenada x ... ¿de verdad tendríamos que comparar todos los pares?

No.

Podríamos ordenarlos de acuerdo a la coordenada x, luego comparar solo los pares contiguos

Este algoritmo tendría complejidad $O(n \log n)$

Esta es la idea base de un algoritmo más eficiente, generalizándolo para versiones del problema donde los puntos tienen 2 coordenadas

OX = puntos ordenados según la x, OY = puntos ordenados según la y

findClosest(OX, OY)

n = número de puntos
If $n = 2$
    return $d(OX[0], OX[1]),\ OX$
If $n = 3$
    return $closest3(OX)$

$medio = \lceil n/2 \rceil$
$minL, closestL = findClosest(OX[0 \ldots medio], OY)$
$minR, closestR = findClosest(OX[medio + 1 \ldots n], OY)$

if $minL < minR$: $closestA = closestL$, $minA = minL$
else: $closestA = closestR$, $minA = minR$

S = elementos de OY cuya x este a distancia menor de $minA$ de $OX[medio]$
For cada i hasta $|S|$
    For j de $i + 1$ a $\min(7, |S|)$
        Si la distancia entre $S[i]$ y $S[j]$ es menor que $minA$, actualizar $minA$ y $closestA$

return $minA, closestA$

Casos base (no es necesario codificar $n = 1$)

Si $n = 1$ — No hay closest-pair

Si $n = 2$ — Los 2 puntos son el closest pair

Si $n = 3$ — Habría que calcular la distancia entre los 3 posibles pares

Nota que la función devuelve la distancia mínima y el par de puntos

OX = puntos ordenados según la x, OY = puntos ordenados según la y

findClosest(OX, OY)

n = número de puntos
If $n = 2$
    return $d(OX[0], OX[1])$, OX
If $n = 3$
    return $closest3(OX)$

$medio = \lceil n/2 \rceil$
$minL$, $closestL = findClosest(OX[0 \ldots medio], OY)$
$minR$, $closestR = findClosest(OX[medio + 1 \ldots n], OY)$

if $minL < minR$: $closestA = closestL$, $minA = minL$
else: $closestA = closestR$, $minA = minR$

S = elementos de OY cuya x este a distancia menor de $minA$ de $OX[medio]$
For cada i hasta $|S|$
    For j de $i + 1$ a $\min(7, |S|)$
        Si la distancia entre $S[i]$ y $S[j]$ es menor que $minA$, actualizar $minA$ y $closestA$

return $minA, closestA$

Llamadas recursivas, dividiendo el arreglo de puntos OX en dos mitades, aka particiones

- $minL$ y $closestL$ son el resultado de distancia y puntos mas cercanos de la partición izquierda
- $minR$ y $closestR$ son el resultado de distancia y puntos mas cercanos de la partición derecha
- $minA$ y $closestA$ son la distancia mínima y los puntos de entre ambas mitades

OX = puntos ordenados según la x, OY = puntos ordenados según la y

findClosest(OX, OY)

n = número de puntos

If $n = 2$

return $d(OX[0], OX[1])$, OX

If $n = 3$

return $closest3(OX)$

$medio = \lceil n/2 \rceil$

$minL$, $closestL = findClosest(OX[0 \ldots medio], OY)$

$minR$, $closestR = findClosest(OX[medio + 1 \ldots n], OY)$

if $minL < minR$: $closestA = closestL$, $minA = minL$

else: $closestA = closestR$, $minA = minR$

S = elementos de OY cuya x este a distancia menor de $minA$ de $OX[medio]$

For cada i hasta $|S|$

For j de $i + 1$ a $\min(7, |S|)$

Si la distancia entre $S[i]$ y $S[j]$ es menor que $minA$, actualizar $minA$ y $closestA$

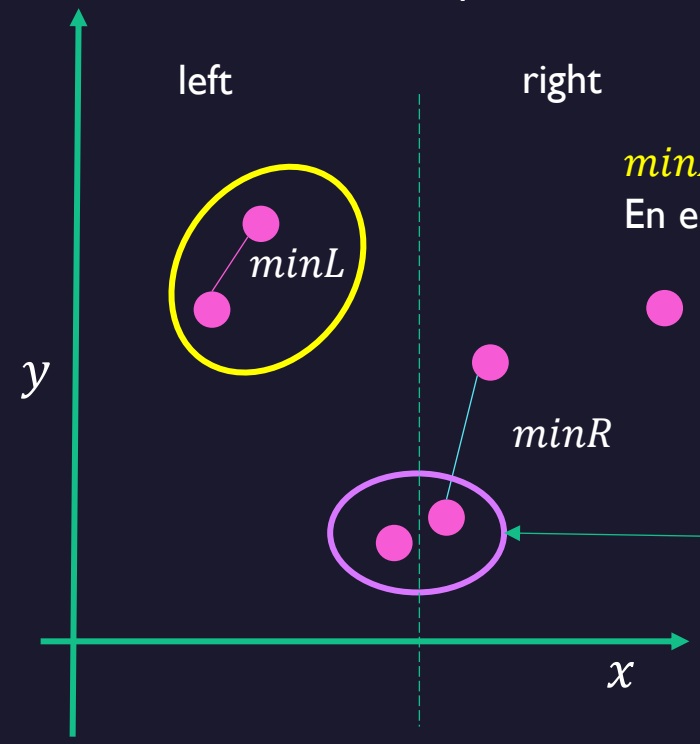
return $minA, closestA$

Hasta este punto, el algoritmo no es tan distinto de la versión para una sola dimensión…

Ya calcula los 2 puntos más cercanos, creando particiones recursivas de acuerdo a la coordenada X

Pero ¿qué pasa si los puntos más cercanos están en distintas particiones?

división en dos particiones



$minA$ = mínimo entre $minL$ y $minR$
En este ejemplo, $minA = minL$

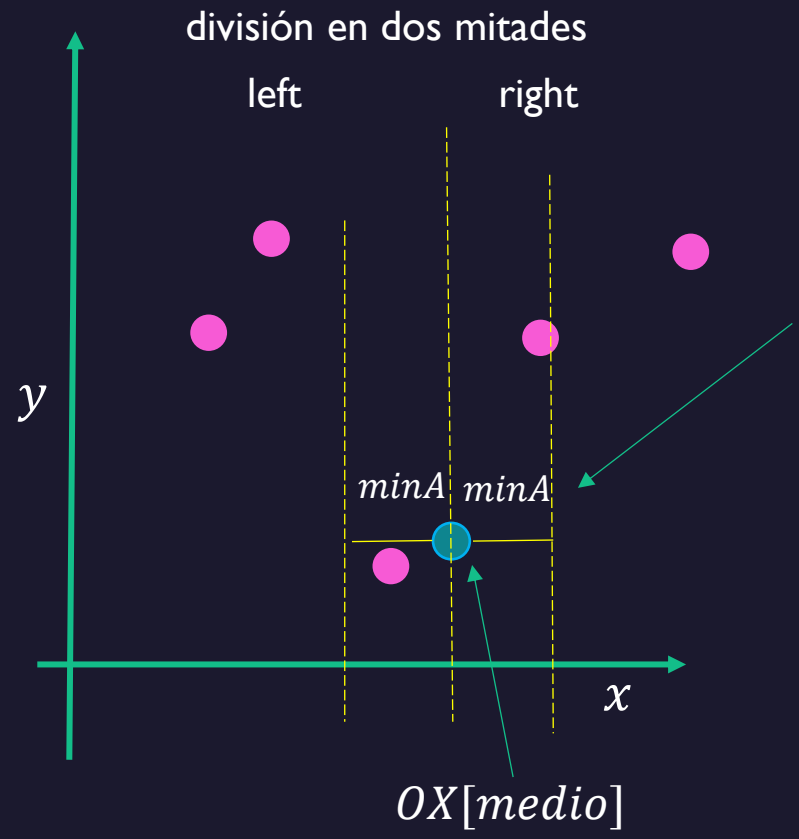
Estos 2 puntos están mas cerca, pero están en particiones diferentes, y no se han comparado aun…

El algoritmo necesita una forma de comparar puntos en diferentes particiones.

Pero sin compararlos todos, solo los que potencialmente puedan dar una distancia menor que minA

¿Cómo reconocer cuales son esos?

Recuerda:
OX = puntos ordenados por su X
OY = puntos ordenados por su Y



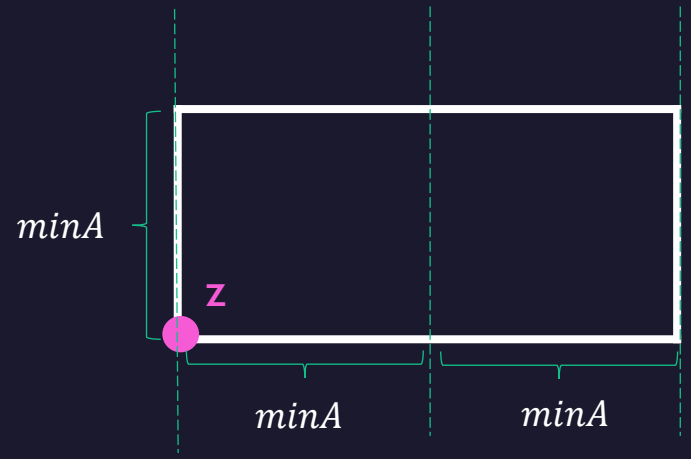
Primero, encontremos un punto en la frontera de ambas particiones.
Sera el elemento en el índice $medio = n/2$ de X. O sea $OX[medio]$

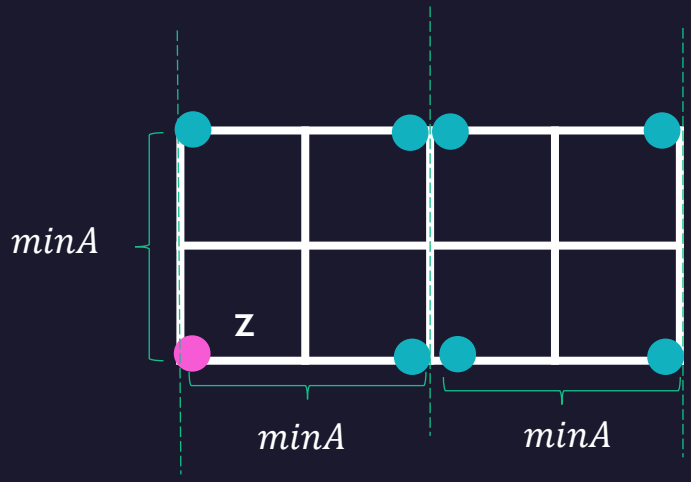
Los puntos que nos interesan están a una distancia en x menor a $minA$ de $OX[medio]$ (región amarilla)

Son todos los puntos de OY cuya coordenada x tenga una diferencia menor a $minA$ respecto a $OX[medio]$
Estos serán los puntos seleccionados en la lista S

Los puntos en S son cercanos a la frontera entre particiones: algún par de ellos podría tener una distancia menor que $minA$

Entonces ¿comparamos todos los pares de puntos en S entre si? No.
Solo cada punto con (máximo) los 7 siguientes. ¿Porque solo 7?





Importante:
Los puntos en S fueron elegidos de OY (ordenados según la coordenada Y).
z es el primer punto en S, por tanto tiene la Y mas pequeña, y no hay puntos debajo de z

Imagina una ventana de tamaño $2(minA) \times minA$
Cualquier punto fuera de la ventana tendrá distancia a z mayor a $minA$. Así que z solo necesita compararse con puntos en la ventana

¿Cuántos puntos podría haber en la ventana? Máximo 8, 4 de cada partición.

Si hubiese siquiera uno mas, tendría distancia menor a $minA$ con algún punto de su misma partición, y ya habrían sido comparados antes.

En este ejemplo, imagina la ventana dividida en 8 partes.
Nota que no puedes agregar otro punto sin que este a distancia menor a $minA$ de otro en su misma partición

OX = puntos ordenados según la x, OY = puntos ordenados según la y

findClosest(OX, OY)

n = número de puntos
If $n = 2$
    return $d(OX[0], OX[1]),\ OX$
If $n = 3$
    return $closest3(OX)$

$medio = \lceil n/2 \rceil$
$minL, closestL = findClosest(X[0 \ldots medio], OY)$
$minR, closestR = findClosest(X[medio + 1 \ldots n], OY)$

if $minL < minR$: $closestA = closestL$, $minA = minL$
else: $closestA = closestR$, $minA = minR$

S = elementos de OY cuya x este a distancia menor de $minA$ de $OX[medio]$
For cada i hasta $|S|$
    For j de $i + 1$ a $\min(7, |S|)$
        Si la distancia entre $S[i]$ y $S[j]$ es menor que $minA$, actualizar $minA$ y $closestA$

return $minA, closestA$

La ultima parte del algoritmo elige los elementos de S

Luego, compara distancias para cada uno con, máximo, sus 7 siguientes en S (o menos, si S tiene menos de 7 elementos)

Si encuentra puntos con distancias menores, actualiza. Y al final, regresa el resultado

Así, consigue complejidad de O(???)

OX = puntos ordenados según la x, OY = puntos ordenados según la y

findClosest(OX, OY)

n = número de puntos
If $n = 2$
    return $d(OX[0], OX[1]),\ OX$
If $n = 3$
    return $closest3(OX)$

$medio = \lceil n/2 \rceil$
$minL, closestL = findClosest(OX[0 \ldots medio], OY)$
$minR, closestR = findClosest(OX[medio + 1 \ldots n], OY)$

if $minL < minR$: $closestA = closestL$, $minA = minL$
else: $closestA = closestR$, $minA = minR$

S = elementos de OY cuya x este a distancia menor de $minA$ de $OX[medio]$
For cada i hasta $|S|$
    For j de $i + 1$ a $\min(7, |S|)$
        Si la distancia entre $S[i]$ y $S[j]$ es menor que $minA$, actualizar $minA$ y $closestA$

return $minA, closestA$

La ultima parte del algoritmo elige los elementos de S

Luego, compara distancias para cada uno con, máximo, sus 7 siguientes.

Si encuentra puntos con distancias menores, los actualiza. Y al final, regresa el resultado

Así, consigue complejidad de $O(n \log n)$

La técnica que usa es divide and conquer