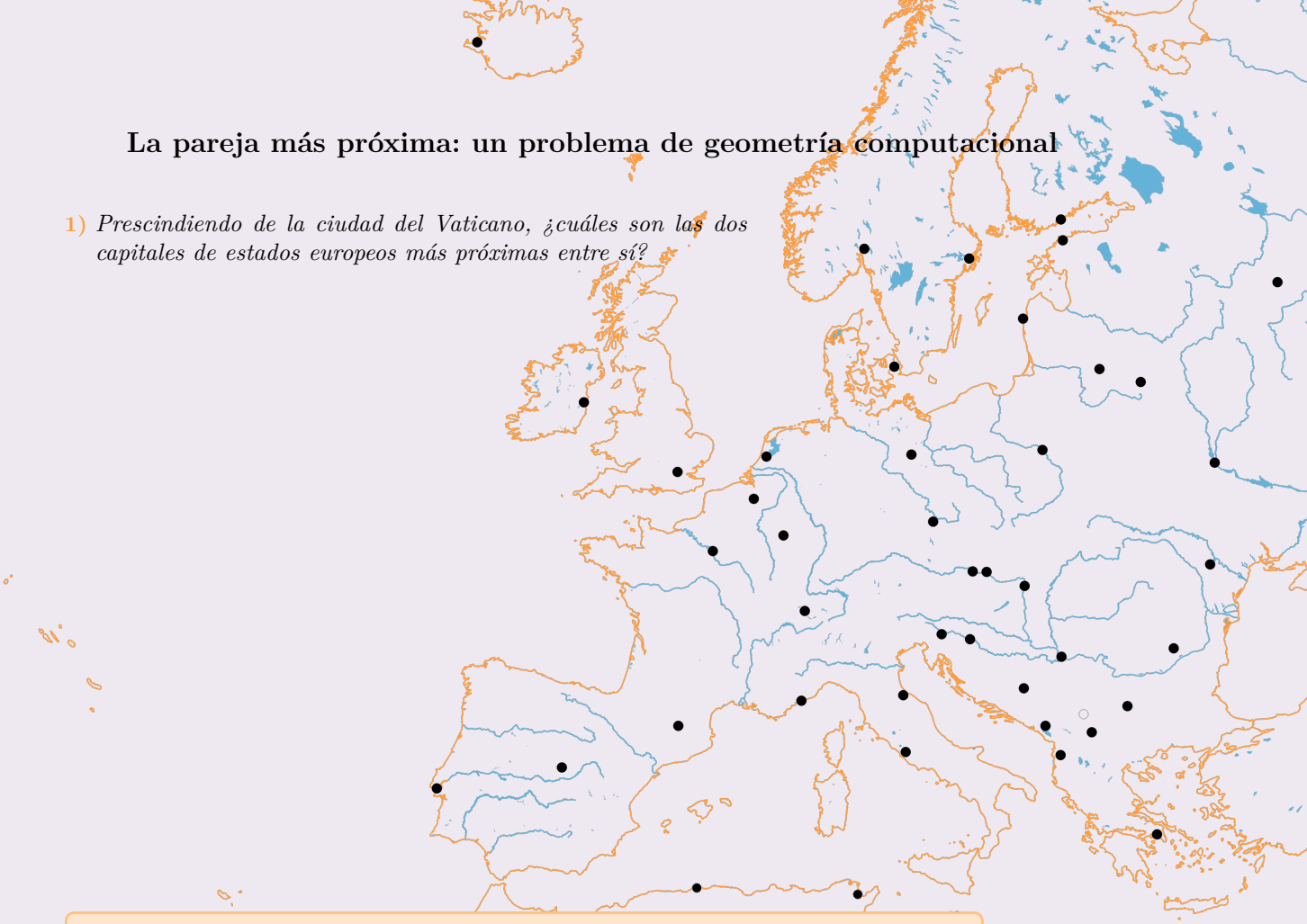
La pareja más próxima: un problema de geometría computacional



1) *Prescindiendo de la ciudad del Vaticano, ¿cuáles son las dos capitales de estados europeos más próximas entre sí?*

En una lista[1] de puntos en el plano[2] $(p_i)_{i=1,\dots,n} \in (\mathbb{Q}^2)^n$, queremos localizar dos elementos los más cercanos posible. Es decir, buscamos dos puntos p_u y p_v $(u \neq v)$ tales que

$$d(p_u, p_v) = \min\{d(p_i, p_j) \quad | \quad 1 \leq i < j \leq n\} \qquad (\geq 0).$$

La primera idea que viene a la mente para abordar esta tarea es acometer una búsqueda exhaustiva.

2) *Escribe, utilizando el lenguaje Scheme, un* script *llamado* pregunta1.rkt *que un fichero de puntos y resuelva el problema mediante esta estrategia de* fuerza bruta. *El programa debe permitir cambiar el nombre de fichero de entrada.*

3) *Estima la complejidad espacial y temporal de ese algoritmo. ¿Genera un proceso recursivo o iterativo?*

Este problema se puede resolver de manera más eficiente mediante un algoritmo de tipo *divide y vencerás* que desarrolló Michael I. Shamos a partir de una idea de H. Ray Strong.[3]

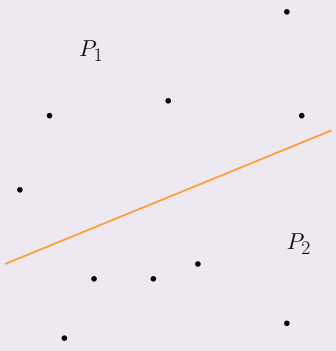
Descomposición

La estrategia comienza con la separación de la nube de puntos $(p_i)_{i\in I}$ en dos partes (dos sublistas):

[1] Obtendríamos una presentación más simple con notación conjuntista, pero estaríamos restringiendo artificialmente el problema al no considerar datos de entrada con puntos repetidos.

[2] La presentación que hemos hecho no queda concernida por este problema, salvo por analogía, ya que la Tierra no es plana.

[3] M. I. Shamos: *Computational geometry*, § 6.2. Tesis doctoral, Universidad de Yale, 1978
F. P. Preparata y M. I. Shamos: *Computational geometry*, § 5.4, 1985.
T. H. Cormen *et al.*: *Introduction to algorithms*, § 33.4

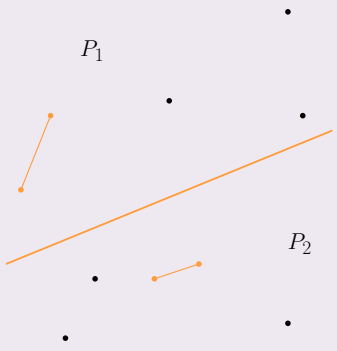


$P_1 = (p_i)_{i \in I_1}$ y
$P_2 = (p_i)_{i \in I_2}$, $\quad I = I_1 \cup I_2$,

para encontrar recursivamente sendas soluciones

$$(p_1, q_1) \quad \text{y} \quad (p_2, q_2)$$

con un par de puntos, lo más cercanos entre sí que se pueda, en cada una de esas partes.

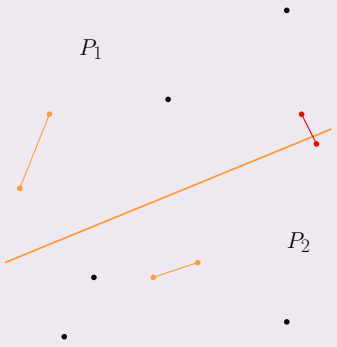


En el ejemplo del dibujo, la partición ha resultado muy bien y una de las parejas obtenidas ((p_2, q_2), en concreto) es la solución al problema. Claro está, para que esto funcionara siempre, habría que conocer la solución de antemano. Así, al separar los datos de entrada, podría evitarse que la solución quedara a caballo entre las dos partes. No parece una vía practicable.

En general, hay que considerar la posibilidad de que la pareja más próxima haya quedado separada por la partición y, por tanto, no coincida con la solución de ninguno de los dos subproblemas.

La dificultad de este planteamiento radica, entonces, en la tercera fase de la estrategia *divide y vencerás*: construir la solución del problema a partir de las soluciones parciales.

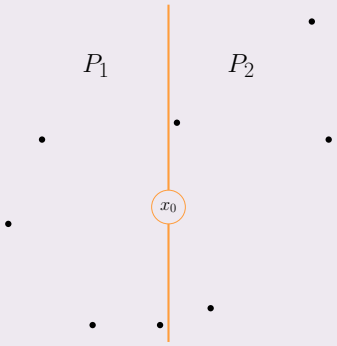
Pero nos detendremos antes en la primera fase.



En la etapa de división, siempre interesa reducir cuanto se pueda el tamaño del problema. Así, al separar la lista de entrada en dos partes, procuraremos que ambas tengan la misma cantidad de puntos (o haya uno solo de diferencia, si partimos de un cantidad impar).

Lo haremos buscando una vertical divisoria $x = x_0$.

4) *Planteamos la separación de la nube de puntos según queden a la izquierda o a la derecha de cierta recta vertical, es decir, escogemos $x_0 \in \mathbb{Q}$ y adscribimos a P_1 los puntos (x, y) que cumplan $x < x_0$ y a P_2 los que cumplan $x > x_0$.*
¿Siempre podemos encontrar $x_0 \in \mathbb{Q}$ de modo tal que este sistema reparta todos los puntos de la lista inicial en sublistas con $\big||P_1| - |P_2|\big| \leq 1$?



5) *Escribe una función que, tomando como argumento una lista de puntos, la divida en dos partes (llamémoslas P_1 y P_2) que satisfagan:*

- $\big||P_1| - |P_2|\big| \leq 1$ *y*
- $\exists x_0 \in \mathbb{Q}$ *tal que todo punto (x, y) de* P_1 *cumple* $x \leq x_0$ *y todo punto (x, y) de* P_2 *cumple* $x \geq x_0$.

Escribe tu respuesta en un fichero pregunta5.rkt. *¿Qué complejidad temporal y espacial representa tu función? Utilizar la función* sort.

Utilizando la notación $T(n)$, donde n es la longitud de la lista de entrada, para el tiempo empleado por el algoritmo de búsqueda de la pareja más próxima que estamos describiendo, podemos acotar

$$T(n) \leq 2 \cdot T(n/2) + f(n),$$

siendo $f(n)$ el tiempo necesario para efectuar la partición y construir la solución global a partir de las dos parciales.

6) *Utilizando el teorema maestro, estima la complejidad del algoritmo en los dos casos siguientes:*

   a) $f(n) = \Theta(n)$ *y*

   b) $f(n) = \Theta(n^2)$.

Nuestro objetivo es ajustarnos al primero de estos supuestos. El logro clave de la estrategia que estamos presentando es completar la tercera fase (de combinación) en tiempo lineal (nos dedicaremos a ello enseguida), pero también hay que prestarle atención a la primera: el procedimiento de partición de la nube de puntos en dos mitades debe funcionar en $O(n)$.

De las soluciones que hemos propuesto para resolver (5), la que adapta el cálculo de la mediana sí funciona en $O(n)$. Por otra parte, la ordenación funciona en $O(n \cdot \log n)$, ligeramente peor de lo que buscamos.

Ahora bien, es posible organizar el algoritmo de manera que esta ordenación de las proyecciones de los puntos sobre el eje de abscisas no se repita en cada paso, sino que forme parte de un **preproceso**. De esta forma, en cada paso recursivo, el algoritmo recibe una lista de puntos ordenada de menor a mayor abscisa y el reparto se puede realizar de manera ágil, conservando el orden en las partes generadas.

7) *Resuelve (5) bajo el supuesto añadido de que los puntos de la lista de entrada están ordenados con abscisa no decreciente, y con el requisito adicional de mantener esta propiedad en las listas de salida.*

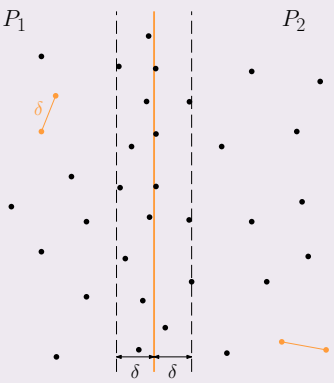
Esta opción introduce una complicación sobre el esquema básico de los algoritmos divide y vencerás: el preproceso. Aunque mejora la complejidad de la primera fase, no lo necesitamos para conseguir tiempo lineal (gracias al cálculo de la mediana), pero sí en la tercera.

Combinación

Vamos ya a abordar la búsqueda de la pareja de puntos más próximos, después de haber obtenido las respuestas (p_1, q_1) en P_1 y (p_2, q_2) en P_2. Trataremos de conseguirlo en tiempo lineal, de acuerdo con el primer supuesto de la pregunta (6).

De entre las dos parejas obtenidas, elegimos como candidata inicial la que tenga menor distancia:

$$\delta = \min(d(p_1, q_1),\ d(p_2, q_2)).$$



Se trata ahora de explorar las parejas «mixtas» (p, q) —con p extraído de P_1 y q de P_2— para las que, potencialmente, pueda darse $d(p, q) < \delta$. Dicho con más claridad: hemos de comprobar todas las parejas, salvo aquellas para las que podamos asegurar que $d(p, q) \geq \delta$.

Proyectando esta restricción sobre el eje de abscisas, resulta claro que cabe limitar la búsqueda a las bandas

$B_1 = \{(x, y) \mid x_0 - \delta < x \leq x_0\}$ y $B_2 = \{(x, y) \mid x_0 \leq x < x_0 + \delta\}$.

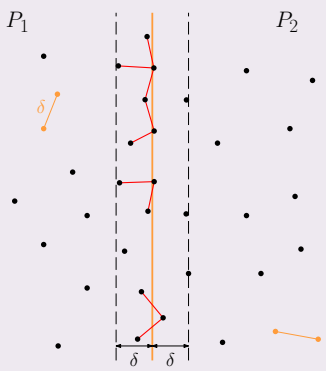
Sin embargo, estas bandas pueden estar demasiado pobladas, haciendo inasumible la comprobación exhaustiva de todas las parejas si se quiere mejorar la complejidad cuadrática del algoritmo ingenuo propuesto en la pregunta (2).

8) *Escribe un algoritmo completo para la búsqueda de una pareja de puntos con distancia mínima, combinando la respuesta del ejercicio (7) con la búsqueda exhaustiva de los puntos en estas bandas.*

   Prueba esta solución con el conjunto de puntos representado en el fichero `ej_pts_01.txt`.

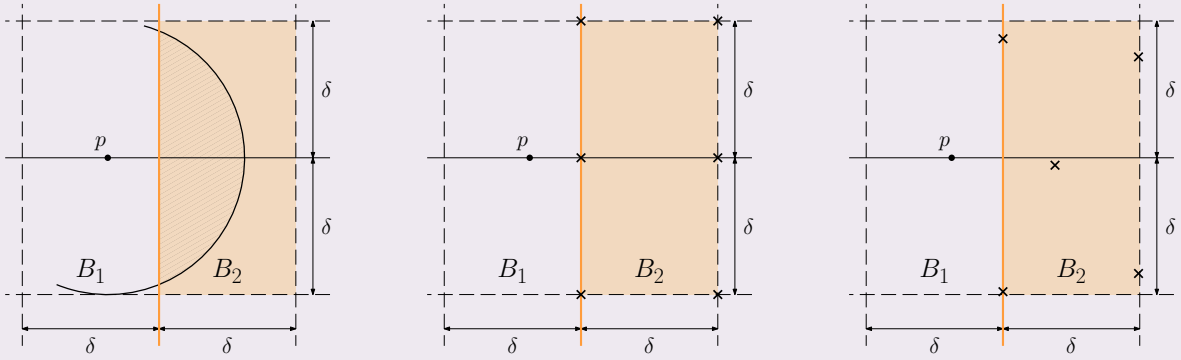
No basta entonces con esa restricción.

En la imagen de la derecha, vemos destacadas en rojo las parejas de puntos con distancia menor que δ.



> La eficiencia que consigue este algoritmo descansa sobre la idea de que, para cada punto de la banda izquierda, podemos desechar *a priori* todos aquellos de la derecha que estén muy alejados verticalmente.
>
> Los que quedan son una pequeña cantidad, que se puede acotar por una **constante**.

En efecto, de la proyección de la condición $d(p, q) < \delta$ sobre el eje de ordenadas, se sigue que, dado un elemento $p = (p_x, p_y)$ en P_1, se pueden descartar de la búsqueda los puntos de P_2 que estén fuera del rectángulo $\{(q_x, q_y) \in B_2 \mid p_y - \delta < q_y < p_y + \delta\}$.



Esto reduce a $O(1)$ los posibles compañeros de un punto dado: fijando $p = (p_x, p_y)$ en la banda izquierda, en el rectángulo $[x_0, x_0 + \delta) \times (p_y - \delta, p_y + \delta)$ no puede haber más de **cinco** puntos de P_2.[4]

Se trata ahora de aprovechar este dato en el diseño del algoritmo. Date cuenta de que no sería una estrategia útil mirar, para cada punto de la banda izquierda, si cada uno de los puntos de la otra banda está o no en el rectángulo, pues estaríamos llevando a cabo una búsqueda exhaustiva.

> El algoritmo opera sobre los puntos de las bandas, **ordenados ahora según su** proyección sobre el eje **vertical**. Para cada punto de una banda, solo es necesario comprobar unos pocos puntos (una cantidad fija), entre los que sabemos que deben estar todos aquellos con ordenada cercana.

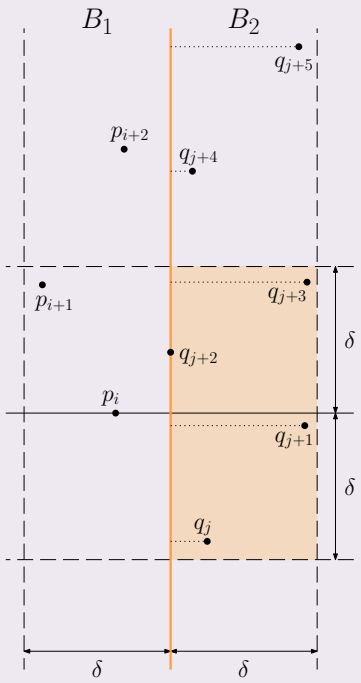
[4] Si se trata de encajar puntos con separación mínima δ en un cuadrado de lado δ, es fácil convencerse de que son **cuatro**, como mucho, los que pueden colocarse.

De manera similar, en un rectángulo de dimensiones $\delta \times 2\delta$ caben **seis** puntos, todos ellos en los bordes. En nuestro caso, al manejar un rectángulo con tres de sus lados abiertos, podemos reducir a **cinco** el máximo número de puntos.

Veamos que es suficiente recorrer una sola vez (empleando, por tanto, tiempo lineal) estas bandas:

- Utilizamos los índices $\mathtt{i}$ y $\mathtt{j}$ para los puntos de cada una de ellas.
- Llegados a $\mathtt{p_i}$, **avanzamos** (nunca es necesario retroceder) el índice $\mathtt{j}$ hasta que $\mathtt{q_j}$ sea una pareja potencial ($\mathtt{q_{j,y}} > \mathtt{p_{i,y}} - \delta$).
- Comprobamos si las parejas que forma con $\mathtt{p_i}$ cada uno de los cinco puntos sucesivos $\mathtt{q_{j+k}}$, $\mathtt{k} = 0, \ldots, 4$, mejoran la distancia mínima registrada.

  Sabemos que los puntos de $\mathtt{q_{j+5}}$ en adelante están forzosamente fuera de la región de interés.



Con algo más de detalle:

$\mathtt{i} \leftarrow 0, \quad \mathtt{j} \leftarrow 0$
while $\mathtt{i} < |\mathtt{B_1}|$ **do**
  while $\mathtt{B_2[j][1]} \leq \mathtt{B_1[i][1]} - \delta$ **do** $\mathtt{j} \leftarrow \mathtt{j} + 1$
  if $\mathtt{j} \geq |\mathtt{B_2}|$ **then break**
  for $\mathtt{k} \leftarrow 0$ **to** *mín*$(5, |\mathtt{B_2}| - \mathtt{j}) - 1$ **do**
    Comprobar la pareja $(\mathtt{B_1[i]}, \mathtt{B_2[j + k]})$
  end
  $\mathtt{i} \leftarrow \mathtt{i} + 1$
end

Tropezamos por segunda vez con la necesidad de ordenar una lista de puntos, tarea incompatible (en principio) con la complejidad lineal que perseguimos. De nuevo, podemos recurrir al preproceso para evitar repetir la ordenación en cada paso de la recursión.

Veamos cómo narra el propio Shamos el descubrimiento del algoritmo:[5]

[5] M. I. Shamos: «The early years of computational geometry—a personal memoir», en *Advances in discrete and computational geometry*, Contemporary Mathematics **223**, 1999, pp. 325—326.

During the talk, I described efforts to solve the closest points problem and said that I felt irked that we had not be[en] able to do better than $O(n^2)$*, despite having "tried everything." After the talk, Strong invited me back to his office to discuss an idea he came up with during my lecture. When we got there, he drew a set of points on the board and outlined a procedure. First find the median* x*-coordinate of the* n *points in linear time by the method of Blum et al. The vertical line* M *having this coordinate divides the set in half. Now find the closest pair of points in the left set and in the right set and let the smaller of the two distances be* δ*. Now draw lines at distance* δ *parallel to and on either side of the line* M*. Strong said it was sufficient to consider only points that lie in the two vertical slabs formed by these lines.*

Well, Hoey and I had seen these slabs many times as we had tried the same construction in applying divide-and-conquer to the problem. So I immediately said to Strong, "But in the worst case all n *points will lie in the slabs, and you get no improvement." He said, "That's true, but the points in the slabs now have a special structure. They're sparse. Within a slab, no two points are closer together than* δ*." My response was, "So what?" How can you use that fact to do anything?" There followed one of those moments that scientists live for—a flash of insight that produces eternal clarity. Strong said, "That means that for every point in the left slab you only have to look at points in the right slab that are within distance* δ*, and there can be at most a* constant *number of them. For every point in the left you don't have to look at every point in the right, so the merge step won't be quadratic." This was the whole key to the closest-point problem.*

There was a little more work to do; we had to find a procedure for locating that constant number of points (at most six, it turned out) to be examined for each point in each slab. This we did by sorting points by y*-coordinate at each recursive step, which led to an* $O(n \log^2 n)$ *algorithm. I wasn't happy about the extra factor of* $\log n$*, but I drove back to New Haven ecstatic in the knowledge that the closest-point problem was subquadratic. That suggested that a fast algorithm for the Voronoi diagram might be possible. I showed Strong's result to Hoey and we huddled to see whether we could squeeze out that extra factor of* $\log n$*. It turns out that if the set of points is sorted initially by* y*-coordinate, no further sorting steps are required, and the merge step of the divide-and-conquer can be performed in linear time, which gives an* $O(n \log n)$ *algorithm.*

Estamos ya en condiciones de ensamblar el algoritmo completo. Tras un preproceso que termina en $O(n \cdot \log n)$, contamos con *arrays* X e Y que representan la misma nube de puntos. En el primero, están dispuestos con abscisa no decreciente; en el segundo, con ordenada no decreciente.

a) Separar X en dos *arrays* (X_1 y X_2), según se pide en (7).

   Es decir: de tamaños iguales o diferenciados en una sola unidad, manteniendo la ordenación de las abscisas y de modo que exista $x_0 \in \mathbb{Q}$ tal que la proyección de X_1 sobre la primera coordenada esté en$(-\infty, x_0]$ y la de X_2 en $[x_0, \infty)$.

b) Generar *arrays* Y_1 e Y_2 con los mismos puntos que X_1 y X_2, respectivamente, pero ordenados con ordenada no decreciente.

c) Ejecutar recursivamente este algoritmo sobre (X_1, Y_1) y (X_2, Y_2). Sea δ la menor de las dos distancias obtenidas.

d) Generar *arrays* B_1 y B_2, restringiendo Y_1 e Y_2 (y manteniendo su orden) a los puntos de ordenada mayor que $x_0 - \delta$, en un caso, y menor que $x_0 + \delta$, en el otro.

e) Comprobar, en tiempo lineal, si alguna pareja «mixta» (un elemento en cada banda) mejora la distancia δ.

9) *Asegúrate de que todos estos pasos, salvo la llamada recursiva (c), puedan completarse en tiempo lineal sobre la cantidad de puntos.*

   Escribe este algoritmo de forma detallada en lenguaje Scheme, incluyendo el preproceso necesario para completar la solución del problema.

En el libro de Cormen *et al.*, encontramos una organización alternativa del algoritmo que, con la misma complejidad computacional esencialmente, permite escribirlo de manera más sencilla.

En lugar de manejar las dos bandas por separado, con sendos *arrays*, coloca en uno solo los puntos de ambas bandas (con la notación Y' en el libro). Después, recorre estos puntos de menor a mayor ordenada y evalúa la separación de las parejas que forma cada uno con los cinco siguientes,[6] sin comprobar *a priori* que estén en bandas distintas.[7]

10) *Modifica el programa que responde a la pregunta anterior, adaptándolo a la presentación del libro.*

Acotación inferior asintótica para la complejidad

En estas hojas hemos expuesto el algoritmo de Shamos y Strong para la búsqueda, en una nube de puntos del plano, de la pareja de puntos más próxima. La complejidad computacional de este algoritmo está en $O(n \cdot \log n)$.

¿Hay otro mejor?

Para responder afirmativamente a esta pregunta, bastaría encontrar uno más eficiente. En cambio, podría decirse que la estrategia presentada es óptima (asintóticamente) si se demuestra que la complejidad intrínseca del problema está en $\Omega(n \cdot \log n)$.

Un escenario habitual para la obtención de cotas inferiores para la complejidad de un problema consiste en restringir el modelo computacional, «encorsetando» el tipo de algoritmos que se contemplan. Veámoslo en otra situación.

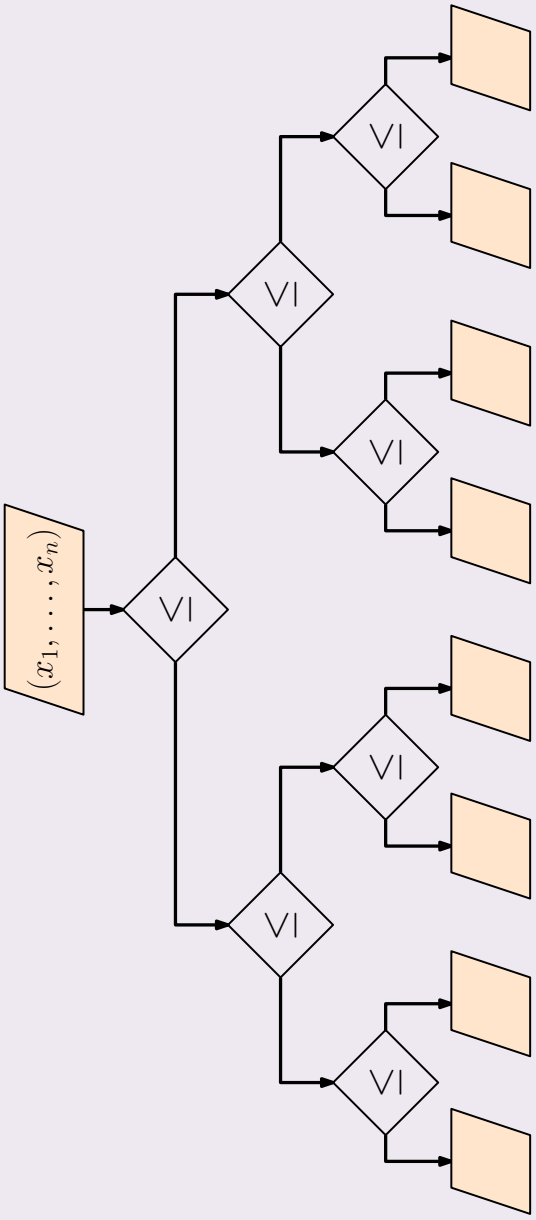
[6] En un primero momento, propone la comprobación de siete parejas para cada punto; para después reducir esa cantidad a cinco en el ejercicio 33.4-2

[7] En caso de encontrar una pareja con separación menor que δ, debe estar formada necesariamente por un punto de cada banda.

Ordenación de un *array*

En el caso de la ordenación de listas de números, sabemos que la complejidad está en $\Omega(n \cdot \log n)$, si consideramos solamente algoritmos **basados en comparaciones**. Esta cota se obtiene considerando el árbol formado por los posibles itinerarios recorridos por ese tipo de programas, en el que cada nodo tiene, a lo más, dos descendientes (es un árbol binario).[8]

Las hojas de tal árbol se identifican con las posibles salidas. En este caso, debe haber cabida para $n!$ hojas, porque, dependiendo de los números concretos que la formen, la lista de entrada puede reordenarse de $n!$ maneras distintas (tantas como permutaciones hay en $\mathfrak{S}_n$).



La profundidad del árbol indica la cantidad máxima de comparaciones efectuadas. Es, como poco, el logaritmo binario del número de hojas:

$$\log_2(n!) \in \Theta(n \cdot \log\ n).$$

Con otro tipo de algoritmos (como *radix sort*), se puede ordenar en tiempo lineal en ciertos contextos concretos (típicamente, cuando todos los números están en un rango determinado).

[8] *Cf.* S. Dasgupta, Ch. Papadimitriou y U. Varizani: *Algorithms*, p. 52

Pareja más próxima

Volvamos ahora al problema abordado en esta actividad. Su complejidad también está en $\Omega(n \cdot \log n)$, bajo cierta restricción de los algoritmos contemplados.

En concreto, se puede demostrar[9] que el problema de determinar la unicidad en una lista de números (¿hay alguno repetido?) presenta esa misma cota inferior, si se manejan solamente algoritmos que operen a base de comparaciones entre funciones lineales de los datos de entrada. Ese problema, conocido en inglés como *element uniqueness*, se puede reducir al de la pareja más próxima (de hecho, relajado a dimensión uno).

De este modo, el algoritmo de tipo divide y vencerás de Shamos y Strong es asintóticamente óptimo en ese marco computacional limitado.

Sin embargo, sí que hay algún algoritmo más eficiente. En concreto, Michael Rabin propuso uno probabilista que termina en tiempo lineal,[10] matizado posteriormente por otros autores que ponderaron la importancia del modelo computacional sobre la introducción de la aleatoriedad, describiendo un algoritmo determinista en $O(n \log \log n)$.[11]

---

11) *Hemos abordado el problema de hallar la pareja (en rigor, una pareja) de puntos más próxima. ¿Cuál es la complejidad computacional de la búsqueda de la pareja más distante? Este otro problema, resuelve, en particular, el cálculo del diámetro del conjunto de entrada.*

[9] D. P. Dobkin y R. J. Lipton: «On the complexity of computations under varying sets of primitives» (Corollary 2), LNCS **33**, 1975; otra versión del artículo está recogida en la recopilación *Excursions into geometry*, Universidad de Yale, 1976 (p. 40).

[10] M. Rabin: «Probabilistic algorithms». En *Algorithms and Complexity*, Academic Press, 1976.

[11] S. Fortune y J. Hopcroft: «A note on Rabin's nearest-neighbor algorithm», Information Processing Letters **8**(1), 1979.