

Tema 5: Divide y vencerás

Clara María Segura Díaz
Fundamentos de Algoritmia, Curso 2020-21

Dpto. de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

- **Fundamentos de Algoritmia.** G. Brassard, P. Bratley. Prentice Hall, 1997.
Capítulo 7
- **Estructuras de datos y métodos algorítmicos: ejercicios resueltos.** N. Martí , Y. Ortega, J.A. Verdejo. Pearson-Prentice Hall, 2004.
Capítulo 11
- **Foundations of algorithms.** R. E. Neapolitan. Jones & Bartlett Learning, Fifth Edition, 2015.
Capítulo 2

- Esquema divide y vencerás formas de resolver problemas.

- Problema de selección

Problema un poco difícil. No está explicado al 100%. Leérnoslo bien

- Par de puntos más cercano

Nube de puntos. Muy difícil

- Determinación del umbral

Explicación muy rápida. Leérnoslo tranquilamente y bien.

Esquema divide y vencerás

Esquema algorítmico

- Un esquema algorítmico puede verse como un algoritmo genérico que puede resolver distintos problemas que se ajustan a dicho esquema. Cada uno de ellos resuelve una familia de problemas de características parecidas. El resto ya los veremos.
Ejemplos: divide y vencerás, vuelta atrás, programación dinámica, ramificación y poda, esquema voraz ...



Esto es lo que nosotros damos en este curso.

Esquema algorítmico

- Un esquema algorítmico puede verse como un **algoritmo genérico** que puede resolver distintos problemas que se ajustan a dicho esquema. Cada uno de ellos resuelve una familia de problemas de características parecidas.
Ejemplos: divide y vencerás, vuelta atrás, programación dinámica, ramificación y poda, esquema voraz ...
- El esquema **divide y vencerás (DV)** consiste en **descomponer** el problema dado en uno o varios subproblemas del mismo tipo, pero cuyos datos son **una fracción** del tamaño original.
- Una vez resueltos los subproblemas por medio de la aplicación recursiva del algoritmo, se **combinan** sus resultados para construir la solución del problema original.
- Existirá uno o más **casos base** en los que el problema no se subdivide más y se **resuelve**, o bien directamente si es sencillo, o bien utilizando un algoritmo distinto.

Esquema algorítmico

Mismo esquema para problemas distintos pero parecidos. Se aplica a DV y vuelta atrás
(siguiente tema)

- Un esquema algorítmico puede verse como un algoritmo genérico que puede resolver distintos problemas que se ajustan a dicho esquema. Cada uno de ellos resuelve una familia de problemas de características parecidas.

Ejemplos: divide y vencerás, vuelta atrás, programación dinámica, ramificación y poda, esquema voraz ...

Estos son los que nosotros damos en este curso

- El esquema **divide y vencerás (DV)** consiste en **descomponer el problema dado en uno o varios subproblemas del mismo tipo**, pero cuyos datos son **una fracción del tamaño original**.
Siempre que ponga DV = divide y vencerás.
- Una vez resueltos los subproblemas por medio de la aplicación recursiva del algoritmo, se **combinan sus resultados para construir la solución del problema original**. Dividimos los problemas pero combinamos los resultados obtenidos en cada fracción del problema
- Existirá **uno o más casos base** en los que el problema no se subdivide más y se resuelve, o bien directamente si es sencillo, o bien utilizando un algoritmo distinto.
- Aparentemente **estas son las características generales de todo diseño recursivo**, y de hecho el esquema DV es un caso particular del mismo.

Esquema divide y vencerás

¿Cómo podemos diferenciar entre un algoritmo de divide y vencerás y uno de recursividad?

- Para distinguirlo de otros diseños recursivos que no responden a DV, se han de cumplir las siguientes condiciones:
 - Los subproblemas han de tener un tamaño fracción del tamaño original (un medio, un tercio, etc ...). No basta simplemente con que sean más pequeños.

Esquema divide y vencerás

- Para distinguirlo de otros diseños recursivos que no responden a DV, se han de cumplir las siguientes condiciones:
 - Los subproblemas han de tener un tamaño *fracción* del tamaño original (un medio, un tercio, etc ...). No basta simplemente con que sean más pequeños.
 - Los subproblemas se generan *exclusivamente* a partir del problema original. En algunos diseños recursivos, los parámetros de una llamada pueden depender de los resultados de otra previa. En el esquema DV, no.

Esto creo que quiere decir que los resultados que obtengamos en una de las llamadas recursivas son independientes a los de la otra llamada recursiva (no comparten datos). Para ello evitar parámetros de entrada salida

Esquema divide y vencerás

- Para distinguirlo de otros diseños recursivos que no responden a DV, se han de cumplir las siguientes condiciones:
 - Los subproblemas han de tener un tamaño *fracción* del tamaño original (un medio, un tercio, etc ...). No basta simplemente con que sean más pequeños.
 - Los subproblemas se generan *exclusivamente* a partir del problema original. En algunos diseños recursivos, los parámetros de una llamada pueden depender de los resultados de otra previa. En el esquema DV, no.
 - La solución del problema original se obtiene *combinando los resultados de los subproblemas entre sí, y posiblemente con parte de los datos originales*. Otras posibles combinaciones no encajan en el esquema.



Para obtener la solución final del problema deberemos de combinar los distintos resultados obtenidos en los subproblemas.

Esquema divide y vencerás

- Para distinguirlo de otros diseños recursivos que no responden a DV, se han de cumplir las siguientes condiciones:
 - Los subproblemas han de tener un tamaño *fracción* del tamaño original (un medio, un tercio, etc ...). No basta simplemente con que sean más pequeños.
 - Los subproblemas se generan *exclusivamente* a partir del problema original. En algunos diseños recursivos, los parámetros de una llamada pueden depender de los resultados de otra previa. En el esquema DV, no.
 - La solución del problema original se obtiene *combinando los resultados* de los subproblemas entre sí, y posiblemente con parte de los datos originales. Otras posibles combinaciones no encajan en el esquema.
- Los casos base no son necesariamente los casos triviales. Como veremos más adelante podría utilizarse como caso base (incluso debería utilizarse en ocasiones) un algoritmo distinto al algoritmo recursivo DV.

No necesariamente en estos ejercicios los casos base son casos triviales.

Esquema divide y vencerás

Cuando estudiemos esto, enfocar las explicaciones a algoritmos como quicksort y mergesort.

- Para distinguirlo de otros diseños recursivos que no responden a DV, se han de cumplir las siguientes condiciones:
 - Los subproblemas han de tener un tamaño fracción del tamaño original (un medio, un tercio, etc...). No basta simplemente con que sean más pequeños.
En los algoritmos vistos en el tema4 (quick/mergesort y binaria) dividimos en un medio.
 - Los subproblemas se generan exclusivamente a partir del problema original. En algunos diseños recursivos, los parámetros de una llamada pueden depender de los resultados de otra previa. En el esquema DV, no. (uso de parámetros de entrada y salida provoca que no se cumple el DV(divide y vencerás))
 - La solución del problema original se obtiene combinando los resultados de los subproblemas entre sí, y posiblemente con parte de los datos originales. Otras posibles combinaciones no encajan en el esquema.
 - Los casos base no son necesariamente los casos triviales. Como veremos más adelante podría utilizarse como caso base (incluso debería utilizarse en ocasiones) un algoritmo distinto al algoritmo recursivo DV.

Ejemplos de aplicación ya vistos:

- Búsqueda binaria. quicksort
y mergesort
- Algoritmo de ordenación rápida.
- Algoritmo de ordenación por mezclas

Son problemas de divide y vencerás

Incluir en ejemplos la transformada rápida de Fourier.

Si la segunda llamada recursiva utiliza resultados de la primera llamada NO es divide y vencerás.

Esquema divide y vencerás

- Puesto en forma de código, el **esquema DV** tiene el siguiente aspecto:

```
template <class Problema, class Solución>
Solucion divide-y-venceras (Problema x) {
    Problema x1,...,xk; dividimos el problema en subproblemas
    Solucion y1 ,...yk; obtenemos los distintos resultados

    if (base(x))
        return metodo-directo(x); caso base
    else {
        (x1 ,..., xk) = descomponer(x); descomponemos el problema en k
        for (i=1; i<=k; i++)
            yi= divide-y-venceras(xi); Aplicar el algoritmo DV a cada
        return combinar(x, y1 ,..., yk);
    }
}
```

x sub k problemas ==> y sub k soluciones.

Los tipos **Problema**, **Solucion**, y los métodos **base**, **metodo-directo**, **descomponer** y **combinar**, son específicos de cada problema resuelto por el esquema.

RECORDATORIO DEL TEMA ANTERIOR.

- Para saber si la aplicación del esquema DV a un problema dado resultará en una solución eficiente o no, se deberá utilizar la recurrencia en la que el tamaño del problema disminúa mediante división:

Recurrencia de un algoritmo de divide y vencerás

$$T(n) = \begin{cases} c_1 & \text{caso base} \\ a * T(n/b) + c * n^k & \text{caso recursivo} \end{cases} \begin{matrix} & \text{si } 0 \leq n < n_0 \\ & \text{si } n \geq n_0 \end{matrix}$$

- Recordemos que la solución de la misma era:

$$T(n) = \begin{cases} O(n^k) & \text{si } a < b^k \\ O(n^k * \log n) & \text{si } a = b^k \\ O(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Para saber si el resultado obtenido en un problema de divide y vencerás es correcto aplicamos el teorema de la división. NOS LO TENEMOS QUE APRENDER.

a = número de llamadas recursivas en cada ejecución

b = número de veces que divido el vector

k= número de veces que entramos a otras funciones de coste lineal o no.

Análisis del coste

En cuantos menos subproblemas tengamos, en cuanto menor sea el coste de lo que NO es recursivo y cuanto menor sea el tamaño de cada subproblema MEJOR. Es decir, cuanto mayor b (menor tamaño), en cuanto a menor a (menos subproblemas) y menor k (menor coste de la parte no recursiva) mejor. Buscamos esto siempre.

- Para obtener una solución eficiente, hay que conseguir a la vez:
 - que el tamaño de cada subproblema sea lo más pequeño posible, es decir maximizar b.
 - que el número de subproblemas generados sea lo más pequeño posible, es decir minimizar a.
 - que el coste de la parte no recursiva sea lo más pequeño posible, es decir minimizar k.
- La recurrencia puede utilizarse para anticipar el coste que resultará de la solución DV, sin tener por qué completar todos los detalles.
- Si el coste sale igual o peor que el de un algoritmo ya existente, entonces no merecerá la pena aplicar DV.

Problema de selección

EJEMPLO DE PROBLEMA DE DIVIDE Y VENCERÁS

Selección

Dado un vector v de elementos que se pueden ordenar, y un entero k , $0 \leq k \leq v.size() - 1$, el problema de selección consiste en encontrar el k -ésimo menor elemento.

Los que sean.

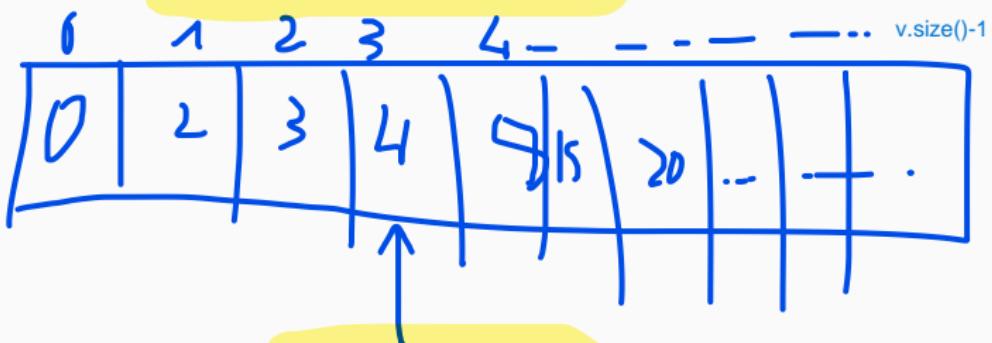
distintas posiciones del vector.

Sería el elemento k si el vector ordenado de manera decreciente.

En particular, encontrar la mediana consiste en encontrar el $\lceil \frac{n}{2} \rceil$ -ésimo elemento, es decir, el elemento que ocupa la posición $(n + 1) \text{ div } 2$ del vector v cuando este se ordena.

Problema muy conocido.

el tercero menor de este vector ordenado



el 3-ésimo elemento menor

Aquí, en el ejemplo que he puesto tenemos un vector YA ORDENADO.. Sin embargo, en los ejercicios que hagamos nos pedirán un elemento concreto PERO sin ordenar el vector ya que necesitamos que sea de coste lineal.

Selección

Dado un vector v de elementos que se pueden ordenar, y un entero k , $0 \leq k \leq v.size() - 1$, el **problema de selección** consiste en encontrar el k -ésimo menor elemento.

En particular, encontrar la **mediana** consiste en encontrar el $\lceil \frac{n}{2} \rceil$ -ésimo elemento, es decir, el elemento que ocupa la posición $(n + 1) \text{ div } 2$ del vector v cuando este se ordena.

Una primera idea para resolver este problema consiste en ordenar el vector v y tomar $v[k]$. Esto tiene una complejidad $O(n \log n)$. ¿Lo podemos hacer mejor?

Cuando se pregunta si se puede hacer mejor es probable que sí se pueda.

Selección

Dado un vector v de elementos que se pueden ordenar, y un entero k , $0 \leq k \leq v.size() - 1$, el **problema de selección** consiste en encontrar el k -ésimo menor elemento.

si tenemos 40 elementos será el que se encuentre en la posición 20

En particular, encontrar la **mediana** consiste en encontrar el $\lceil \frac{n}{2} \rceil$ -ésimo elemento, es decir, el elemento que ocupa la posición $(n + 1) \text{ div } 2$ del vector v cuando este se ordena. Pero nosotros queremos conseguir saber cual es ese elemento pero sin ordenarlos. Una primera idea para resolver este problema consiste en ordenar el vector v y tomar $v[k]$. Esto tiene una complejidad $O(n \log n)$. ¿Lo podemos hacer mejor?

Queremos que sea lineal

Otra posibilidad es utilizar primero el algoritmo **partición** y después distinguir casos, llamando recursivamente con un subvector del original. Para ello es necesario generalizar la función para buscar el k -ésimo en un subvector $[c, f]$:

divide vector en menores mayores
(¿y iguales?) que el pivote

caso base. Si está en la mitad o iguales

- Si el índice p , donde se coloca el pivote, es igual a k , el elemento $V[p]$ es el k -ésimo.
- Si $k < p$, podemos pasar a buscar el k -ésimo en $V[c..p - 1]$, ya que estos elementos son menores o iguales a $V[p]$ y $V[p]$ es el p -ésimo. si esta en los menores
- Si $k > p$, buscamos el k -ésimo en $V[p + 1..f]$, ya que estos elementos son mayores o iguales que el p -ésimo. si está en los mayores.

Selección

```
//k representa una posición absoluta, c <= k <= f
int seleccion1(vector<int> const& v, int c, int f, int k) {
    int e; falta declarar p y q
    if (c == f) {e = v[c]; } importante a la hora de hacer divide y vencerás
    else { caso base entonces k = c = f.
        particion2(v, c, f, v[c], p, q);
        if (p <= k <= q) {e = v[k]; } si es uno de los iguales
        else if (k < p) { e = seleccion1(v, c, p-1, k); } si es de los menores
        else { e = seleccion1(v, q+1, f, k); } buscamos en los mayores.
    };
    return e;
}

int seleccion1(vector<int> const& v, int k) {
    return seleccion1(v, 0, v.size()-1, k);
}
```

la partición que hemos visto en el tema anterior.

posición inicial 0 y final es v.size()-1

Selección

```
//k representa una posicion absoluta, c <= k <= f
int seleccion1(vector<int> const& v, int c, int f, int k) {
    int e;
    if (c == f) {e = v[c]; }
    else {
        particion2(v, c, f, v[c], p, q); 
        if (p <= k <= q) {e = v[k]; }
        else if (k < p) { e = seleccion1(v, c, p-1, k); }
        else { e = seleccion1(v, q+1, f, k); }
    };
    return e;
}
```

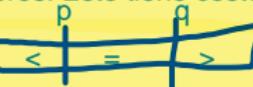
```
int seleccion1(vector<int> const& v, int k) {
    return seleccion1(v, 0, v.size()-1, k);
}
```

El caso peor del algoritmo tiene lugar cuando el pivote queda siempre en un extremo del subvector correspondiente: $\Theta(n^2)$.

El coste en el caso medio está en $\Theta(n)$.

Encontrar el késimo elemento.

menores iguales y mayores. Esto tiene coste lineal



mas el coste de la llamada recursiva.

Vamos a mejorar esto. Esto aún no es lo mejor que podemos conseguir aún.

Selección: mediana de las medianas

Nosotros normalmente cogemos el primer elemento como pivote. Pero si el vector está totalmente ordenado el coste será cuadrático

La forma de mejorar este algoritmo es asegurarnos de que el pivote elegido no va a quedar en un extremo.

Hallar la mediana

Ya que así no caemos en los casos peores.

La mejor elección sería tomar como pivote la *mediana* del subvector, pero calcular la mediana es precisamente un caso particular del problema de selección, cuando

$$k = (c + f) \text{ div } 2.$$

Selección: mediana de las medianas

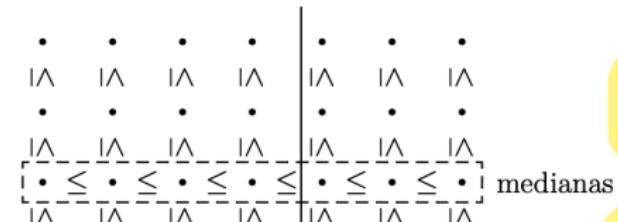
La forma de mejorar este algoritmo es asegurarnos de que el pivote elegido no va a quedar en un extremo.

La mejor elección sería tomar como pivote la *mediana* del subvector, pero calcular la mediana es precisamente un caso particular del problema de selección, cuando $k = (c + f) \text{ div } 2$.

Nos conformamos con una aproximación suficientemente buena de la mediana, conocida como mediana de las medianas.

Lo explica después, pero hay que dividir el vector en trozos de 5 elementos y de cada cachito buscamos el elemento de la mitad. Para coger la mediana de cada uno de los trozos primero debemos ordenar, como son trozos pequeños no hace falta hacer labores

Habrá $n/5$ medianas



Cogemos el vector de elementos y se divide en trozos de 5



Se ordenan y se coge la mediana (el que está en el centro.)

Sacamos $n/5$ medianas
Llamamos a selección 1 para e pivote el cual es la MEDIANA DE LAS MEDIANAS

Selección: mediana de las medianas

Los pasos del algoritmo `seleccion2` son:

- ① calcular la mediana de cada grupo de 5 elementos. En total $n \text{ div } 5$ medianas ($n = f - c + 1$), y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y quedarnos con el tercero.

Este primer paso tiene coste constante.

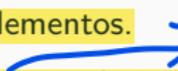
Selección: mediana de las medianas

Los pasos del algoritmo `seleccion2` son:

- ① calcular la mediana de cada grupo de 5 elementos. En total $n \text{ div } 5$ medianas ($n = f - c + 1$), y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y quedarnos con el tercero.
- ② calcular la mediana de las medianas, mm , con una llamada recursiva a `seleccion2` con $n \text{ div } 5$ elementos.

Selección: mediana de las medianas

Los pasos del algoritmo `seleccion2` son:

- ① calcular la mediana de cada grupo de 5 elementos. En total $n \text{ div } 5$ medianas ($n = f - c + 1$), y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y quedarnos con el tercero.
- ② calcular la mediana de las medianas, mm , con una llamada recursiva a `seleccion2` con $n \text{ div } 5$ elementos.


Utiliza la mediana de las medianas.
- ③ llamar a `seleccion2(V, c, f, mm, p, q)`, utilizando como pivote mm .

Selección: mediana de las medianas

Los pasos del algoritmo `seleccion2` son:

- ① calcular la mediana de cada grupo de 5 elementos. En total $n \text{ div } 5$ medianas ($n = f - c + 1$), y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y quedarnos con el tercero.
- ② calcular la mediana de las medianas, mm , con una llamada recursiva a `seleccion2` con $n \text{ div } 5$ elementos.

- ③ llamar a `seleccion2(V, c, f, mm, p, q)`, utilizando como pivote mm .
- ④ hacer una distinción de casos similar a la de `seleccion1`

Selección: mediana de las medianas

Los pasos del algoritmo `seleccion2` son:

- ① calcular la mediana de cada grupo de 5 elementos. En total $n \text{ div } 5$ medianas ($n = f - c + 1$), y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y quedarnos con el tercero. El del centro del grupo (la mediana del subvector)
- ② calcular la mediana de las medianas, mm , con una llamada recursiva a `seleccion2` con $n \text{ div } 5$ elementos.
- ③ llamar a `seleccion2(V, c, f, mm, p, q)`, utilizando como pivote mm .
- ④ hacer una distinción de casos similar a la de `seleccion1`

Si se encuentra en los menores, iguales o mayores

mediana de las medianas calculadas
será el nuevo pivote.
como maximo.

Se puede demostrar que las llamadas recursivas se realizan con $\frac{7n+12}{10}$ elementos como mucho.

$$T(n) \leq dn + T(n/5) + \max \left\{ T(m) \mid m \leq \frac{7n+12}{10} \right\}$$

y utilizando inducción constructiva se puede demostrar que

casi 1/2 (lo que
sería la verdadera
mediana.)

$$\exists c \text{ tal que } T(n) \leq cn \quad \forall n \geq 1.$$

es decir $T(n) \in O(n)$. Nos lo tenemos que creer, en la bibliografía aparece la explicación.

Selección: mediana de las medianas

k es el pivote

```
//k representa una posición absoluta, c <= k <= f
int seleccion2(vector<int> v, int c, int f, int k) {
    int e, pm, aux;
    //s es para dividir el vector
    int s, p, q, t = f - c + 1;
    no se si ese 12 tiene que ver con el umbral que veremos después al final del if (t<12) tema. Si numElem es <12 mejor inserción. No afectaría a nuestro coste.
    { ordInsercion(v, c, f); e = v[k]; }
    else {
        t= tamaño del vector
        s = t / 5; //Las medianas de las medianas al principio
        ordenamos por inserción cada grupo de 5 elementos.
        for (int l = 1; l <= s; l++)
            { ordInsercion(v, c+5*(l-1), c+5*l-1);
                pm = c + 5 * (l - 1) + 5 / 2; pm= mediana de la partición.
                aux = v[c + l - 1]; v[c + l - 1] = v[pm]; v[pm] = aux; };
        ponemos la mediana del vector.
        seleccion2(v, c, c + s - 1, c + (s-1)/ 2, mm); mediana de las medianas
        particion2(v, c, f, mm, p, q);
        mm (mediana de las medianas) es el nuevo pivote.
        if (p <= k <= q) {e = v[k]; }
        else if (k < p) { e = seleccion2(v, c, p-1, k); }
        else { e = seleccion2(v, q+1, f, k); }
    };
    return e;
}
```

caso base es 12. Si tengo menos de 12 elementos hago una ordenación por inserción.

Se puede hacer la ordenación de quicksort con la mediana y en ese caso siempre sería $n \log n$. Dice que hay veces que no merece la pena porque a lo mejor en casos muy aislados es bueno hacer esto.

calcula

similar a selección 1

Porque es más sencillo y NO aumenta el coste con ese umbral.

Par de puntos más cercano

Problema muy complicado de geometría. Utiliza todo lo que hemos visto ahora de la recursión

El problema del par más cercano

Podemos mirar cada punto con todos.---->esto tendría coste $n \log n$

Número de puntos ≥ 2

- Dada una nube de n puntos en el plano, $n \geq 2$, se trata de encontrar el par de puntos cuya distancia euclídea es menor (si hubiera más de un par con esa distancia mínima, basta con devolver uno de ellos). Si hay dos iguales devolvemos 1 o los que nos pidan.
- El problema tiene interés práctico. Por ejemplo, en un sistema de control del tráfico aéreo, el par más cercano nos informa del mayor riesgo de colisión entre dos aviones.

¿Qué puntos están a la menor distancia.?

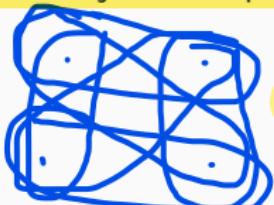
Esto es útil como se dice arriba para comprobar qué dos aviones tienen más probabilidades de chocar.

El problema del par más cercano

- Dada una nube de n puntos en el plano, $n \geq 2$, se trata de encontrar el par de puntos cuya distancia euclídea es menor (si hubiera más de un par con esa distancia mínima, basta con devolver uno de ellos).
- El problema tiene interés práctico. Por ejemplo, en un sistema de control del tráfico aéreo, el par más cercano nos informa del mayor riesgo de colisión entre dos aviones.
- El algoritmo de “fuerza bruta” calcularía la distancia entre todo posible par de puntos, y devolvería el mínimo de todas ellas.
- Como hay $\frac{1}{2}n(n - 1)$ pares posibles, el coste resultante sería **cuadrático**.

El problema del par más cercano

- Dada una nube de n puntos en el plano, $n \geq 2$, se trata de encontrar el par de puntos cuya distancia euclídea es menor (si hubiera más de un par con esa distancia mínima, basta con devolver uno de ellos).
- El problema tiene interés práctico. Por ejemplo, en un sistema de control del tráfico aéreo, el par más cercano nos informa del mayor riesgo de colisión entre dos aviones.
- El algoritmo de "fuerza bruta" calcularía la distancia entre todo posible par de puntos, y devolvería el mínimo de todas ellas. Esto no lo debemos hacer nunca ya que no es eficiente
- Como hay $\frac{1}{2}n(n - 1)$ pares posibles, el coste resultante sería cuadrático.
- El enfoque DV trataría de encontrar el par más cercano a partir de los pares más cercanos de conjuntos de puntos que sean una fracción del original.



Si hay 4 puntos, pares posibles = $6 = 4/2(3) = 3 \cdot 2$

dividir el problema si no
me equivoco en dos
nubes de puntos, y
encontrar de cada nube
el par más cercano

Par de puntos más cercano



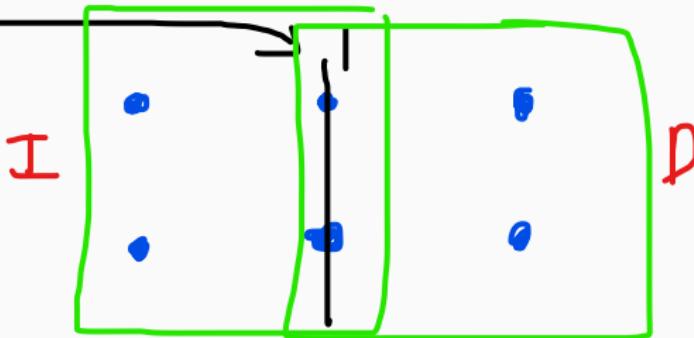
Dos nubes de puntos de tamaño la mitad

Dividir Crear dos nubes de puntos de tamaño mitad. Podríamos ordenar los puntos por la coordenada x y tomar la primera mitad como nube izquierda I , y la segunda como nube derecha D . Determinamos una línea vertical imaginaria l tal que todos los puntos de I están sobre l , o a su izquierda, y todos los de D están sobre l , o a su derecha.

Par de puntos más cercano

Dividir Crear dos nubes de puntos de tamaño mitad. Podríamos ordenar los puntos por la coordenada x y tomar la primera mitad como nube izquierda I , y la segunda como nube derecha D . Determinamos una línea vertical imaginaria I tal que todos los puntos de I están sobre I , o a su izquierda, y todos los de D están sobre I , o a su derecha.

Conquistar Resolver recursivamente los problemas I y D . Sean δ_I y δ_D las respectivas distancias mínimas encontradas y sea $\delta = \min(\delta_I, \delta_D)$.



Par de puntos más cercano

Si por ejemplo el número de puntos es de 40, tener dos nubes de puntos, una con 20 puntos y otra con los otros 20 puntos. (es por eso que es un divide y vencerás este problema.) Tenemos dos coordenadas la x y la y y lo ordenamos según uno de los ejes. Si lo ordenamos por las X lo podemos dividir en la mitad con una línea imaginaria y todos los puntos de la Izquierda están a la izquierda de la línea imaginaria y al revés con los de la derecha.

Dividir

Crear dos nubes de puntos de tamaño mitad. Podríamos ordenar los puntos por la coordenada x y tomar la primera mitad como nube izquierda I , y la segunda como nube derecha D . Determinamos una línea vertical imaginaria $/$ tal que todos los puntos de I están sobre $/$, o a su izquierda, y todos los de D están sobre $/$, o a su derecha.

Conquistar

Resolver recursivamente los problemas I y D . Sean δ_I y δ_D las respectivas distancias mínimas encontradas y sea $\delta = \min(\delta_I, \delta_D)$.

Combinar

El par más cercano de la nube original, o bien es el par con distancia δ , o bien es un par compuesto por un punto de la nube I y otro punto de la nube D . En ese caso, ambos puntos se hallan a lo sumo a una distancia δ de I . La operación *combinar* debe investigar los puntos de dicha banda vertical.

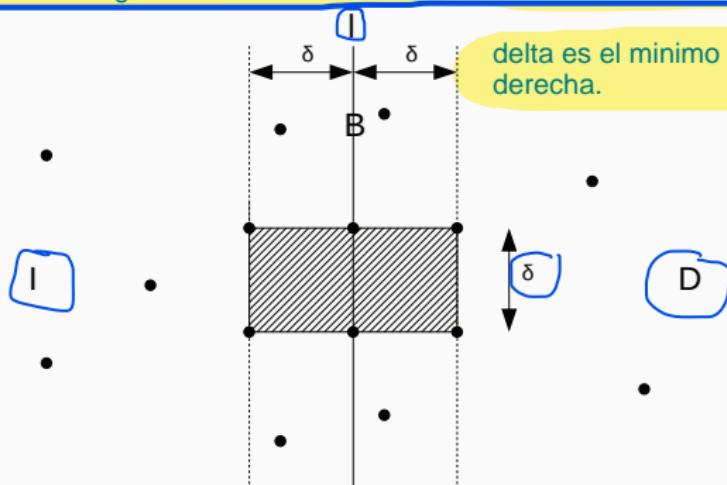
Si el coste es cuadrático no nos sirve.

Esto dice que hallamos la menor distancia de puntos entre ambas mitades y después ver cual de las dos es la mínima.

El par más cercano o bien será el mínimo que hemos calculado o bien será la distancia entre un punto que se encuentra en la mitad izquierda y otro punto de la mitad derecha. CREO que esto daría coste cuadrático

Par de puntos más cercano

Los ordeno por x y divido la mitad. Aquí tenemos los puntos ordenados por x. Tenemos 2 subnubes de puntos, la mitad en un lado y la mitad en el otro. Calcular en cada subnube el par de puntos más cercano. Luego mirar el del conjunto. Esto daría coste cuadrático luego NO NOS SIRVE.



Según lo que pone abajo, si utilizamos el algoritmo de fuerza bruta ENTONCES tendremos un coste cuadrático.

Par de puntos más cercano

- Como el algoritmo fuerza-bruta tiene coste $\Theta(n^2)$, trataremos de conseguir un coste $\Theta(n \log n)$ en el caso peor.
- Sabemos por la experiencia de algoritmos como *mergesort* que ello exige unos parámetros $a = 2$, $b = 2$, $k = 1$, en decir, tan solo podemos consumir un coste lineal en las operaciones de dividir y combinar.

OBJETIVO A CONSEGUIR EN ESTE EJERCICIO. (CASO PEOR).

Par de puntos más cercano

- Como el algoritmo fuerza-bruta tiene coste $\Theta(n^2)$, trataremos de conseguir un coste $\Theta(n \log n)$ en el caso peor.
 - Sabemos por la experiencia de algoritmos como *mergesort* que ello exige unos parámetros $a = 2$, $b = 2$, $k = 1$ en decir, tan solo podemos consumir un coste lineal en las operaciones de dividir y combinar. QUE SOLO ENTREMO UNA VEZ EN LA FUNCIÓN NO RECURSIVA SE REFIERE CREO
- ① La ordenación de los puntos por la coordenada de x se puede realizar una sola vez al principio (es decir, fuera del algoritmo recursivo DV) con un coste $\Theta(n \log n)$ en el caso peor, lo que es admisible para mantener nuestro coste total.

- Como el algoritmo fuerza-bruta tiene coste $\Theta(n^2)$, trataremos de conseguir un coste $\Theta(n \log n)$ en el caso peor.
- Sabemos por la experiencia de algoritmos como *mergesort* que ello exige unos parámetros $a = 2$, $b = 2$, $k = 1$, en decir, tan solo podemos consumir un coste lineal en las operaciones de dividir y combinar.
 - ① La ordenación de los puntos por la coordenada de x se puede realizar una sola vez al principio (es decir, fuera del algoritmo recursivo DV) con un coste $\Theta(n \log n)$ en el caso peor, lo que es admisible para mantener nuestro coste total.
 - ② Una vez ordenada la nube, la división en dos puede conseguirse con coste constante o lineal, dependiendo de si se utilizan vectores o listas como estructuras de datos de la implementación.

Segundo cuatrimestre

- Como el algoritmo fuerza-bruta tiene coste $\Theta(n^2)$, trataremos de conseguir un coste $\Theta(n \log n)$ en el caso peor.
- Sabemos por la experiencia de algoritmos como *mergesort* que ello exige unos parámetros $a = 2$, $b = 2$, $k = 1$, en decir, tan solo podemos consumir un coste lineal en las operaciones de dividir y combinar.
 - ① La ordenación de los puntos por la coordenada de x se puede realizar una sola vez al principio (es decir, fuera del algoritmo recursivo DV) con un coste $\Theta(n \log n)$ en el caso peor, lo que es admisible para mantener nuestro coste total.
 - ② Una vez ordenada la nube, la división en dos puede conseguirse con coste constante o lineal, dependiendo de si se utilizan vectores o listas como estructuras de datos de la implementación.
 - ③ Una vez resueltos los dos subproblemas, se pueden filtrar los puntos de I y D para conservar sólo los que estén en la banda vertical de anchura 2δ y centrada en I . El filtrado puede hacerse con coste lineal.

Par de puntos más cercano

- Como el algoritmo fuerza-bruta tiene coste $\Theta(n^2)$, trataremos de conseguir un coste $\Theta(n \log n)$ en el caso peor.
 - Sabemos por la experiencia de algoritmos como *mergesort* que ello exige unos parámetros $a = 2$, $b = 2$, $k = 1$, en decir, tan solo podemos consumir un coste lineal en las operaciones de dividir y combinar.
- Es decir, coste $n \log n$
- ① La ordenación de los puntos por la coordenada de x se puede realizar una sola vez al principio (es decir, fuera del algoritmo recursivo DV) con un coste $\Theta(n \log n)$ en el caso peor, lo que es admisible para mantener nuestro coste total. con cualquier alg
 - ② Una vez ordenada la nube, la división en dos puede conseguirse con coste constante o lineal, dependiendo de si se utilizan vectores o listas como estructuras de datos de la implementación.
Coste constante = $O(1)$
Coste lineal = $O(n)$
 - ③ Una vez resueltos los dos subproblemas, se pueden filtrar los puntos de I y D para conservar sólo los que estén en la banda vertical de anchura 2δ y centrada en I . El filtrado puede hacerse con coste lineal.
 - ④ Llameemos B_I y B_D a los puntos de dicha banda respectivamente a la izquierda y a la derecha de I . Falta comprobar las distancias de cada punto de B_I a cada punto de B_D .

Par de puntos más cercano

- Es fácil construir nubes de puntos en las que todos ellos caigan en la banda tras el filtrado, de forma que en el caso peor podríamos tener $|B_l| = |B_D| = \frac{n}{2}$.
- En ese caso, el cálculo de la distancia mínima entre los puntos de la banda sería cuadrático, y el coste total del algoritmo DV también.

Par de puntos más cercano

- Es fácil construir nubes de puntos en las que todos ellos caigan en la banda tras el filtrado, de forma que en el caso peor podríamos tener $|B_I| = |B_D| = \frac{n}{2}$.
- En ese caso, el cálculo de la distancia mínima entre los puntos de la banda sería cuadrático, y el coste total del algoritmo DV también.
- Demostraremos que basta ordenar por la coordenada y el conjunto de puntos $B_I \cup B_D$ y después recorrer la lista ordenada comparando cada punto con los 7 que le siguen.
- Si de este modo no se encuentra una distancia menor que δ , concluimos que todos los puntos de la banda distan más entre sí.
- Este recorrido es claramente de coste lineal.

Coste lineal $O(n)$

Par de puntos más cercano

A lo mejor lo de ordenar por la coordenada y no nos funciona. Vamos a ver si esto funciona

- Suponiendo que esta estrategia fuera correcta, todavía quedaría por resolver la ordenación por la coordenada y .
- Si ordenáramos $B_I \cup B_D$ en cada llamada recursiva, gastaríamos un coste $\Theta(n \log n)$ en cada una, lo que conduciría un coste total en $\Theta(n \log^2 n)$.

Ordenar por la coordenada "y" tiene ese coste

Par de puntos más cercano

- Suponiendo que esta estrategia fuera correcta, todavía quedaría por resolver la ordenación por la coordenada y . No sabemos si es correcta
- Si ordenáramos $B_I \cup B_D$ en cada llamada recursiva, gastaríamos un coste $\Theta(n \log n)$ en cada una, lo que conduciría un coste total en $\Theta(n \log^2 n)$.
- Recordando la técnica de los **resultados adicionales** podemos exigir que cada llamada recursiva devuelva un resultado extra: la lista de sus puntos ordenada por la coordenada y .
- Este resultado puede propagarse hacia arriba del árbol de llamadas con un coste lineal porque basta aplicar el algoritmo de mezcla de dos listas ordenadas.

Par de puntos más cercano

- La secuencia de acciones de la operación *combinar* es entonces la siguiente:
 - ➊ Realizar la mezcla ordenada de las dos listas de puntos devueltas por las llamadas recursivas. Esta lista se devolverá al llamante.

Par de puntos más cercano

- La secuencia de acciones de la operación *combinar* es entonces la siguiente:
 - Realizar la mezcla ordenada de las dos listas de puntos devueltas por las llamadas recursivas. Esta lista se devolverá al llamante.
 - Filtrar la lista resultante, conservando los puntos a una distancia de la línea divisoria $/$ menor o igual que δ . Llámemos B a la lista filtrada.

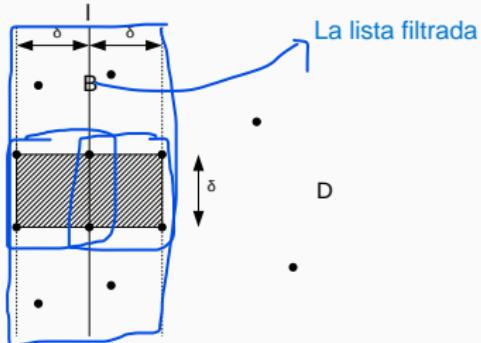
Par de puntos más cercano

- La secuencia de acciones de la operación *combinar* es entonces la siguiente:
 - Realizar la mezcla ordenada de las dos listas de puntos devueltas por las llamadas recursivas. Esta lista se devolverá al llamante.
 - Filtrar la lista resultante, conservando los puntos a una distancia de la línea divisoria $/$ menor o igual que δ . Llamemos B a la lista filtrada.
 - Recorrer B calculando la distancia de cada punto a los 7 que le siguen, comprobando si aparece una distancia menor que δ .

Par de puntos más cercano

- La secuencia de acciones de la operación combinar es entonces la siguiente:
 - ① Realizar la mezcla ordenada de las dos listas de puntos devueltas por las llamadas recursivas. Esta lista se devolverá al llamante.  **Bi and Bd**
 - ② Filtrar la lista resultante, conservando los puntos a una distancia de la línea divisoria / menor o igual que δ . Llamemos B a la lista filtrada.
 - ③ Recorrer B calculando la distancia de cada punto a los 7 que le siguen, comprobando si aparece una distancia menor que δ .
 - ④ Devolver los dos puntos a distancia mínima, considerando los tres cálculos realizados: parte izquierda, parte derecha y lista B .

Corrección de la estrategia



- Consideremos un rectángulo cualquiera de anchura 2δ y altura δ centrado en la línea divisoria I .
- Contenidos en él, puede haber a lo sumo 8 puntos de la nube original; 4 a lo sumo en la mitad izquierda, situados en sus esquinas; y otros 4 en la mitad derecha.
- Ello es así porque, por hipótesis de inducción, los puntos de la nube izquierda están separados entre sí por una distancia de al menos δ , e igualmente los puntos de la nube derecha entre sí.
- En la línea divisoria podrían coexistir hasta dos puntos de la banda izquierda con dos puntos de la banda derecha.

Corrección de la estrategia

- Si colocamos dicho rectángulo con su base sobre el punto p_1 de menor coordenada y de B , estamos seguros de que a los sumo los 7 siguientes puntos de B estarán en dicho rectángulo.
- A partir del octavo, él y todos los demás distarán más que δ de p_1 .
- Desplazando ahora el rectángulo de punto a punto, podemos repetir el mismo razonamiento.
- No es necesario investigar los puntos con menor coordenada y que el punto en curso, porque esa comprobación ya se hizo cuando se procesaron dichos puntos.
- Elegimos como caso base de la inducción $n < 4$. De este modo, al subdividir una nube con $n \geq 4$ puntos, nunca generaremos problemas con un solo punto.

Implementación

- Definimos un punto como una pareja de números reales.

```
struct Punto  
{ double x;  
double y;};
```

Coordenadas.

Cómo se ordenan los elementos de p respecto a la coordenada de las y's.

- La solución

```
void parMasCercano(Punto p[], int c, int f, int indY[], int& ini,  
double& d, int& p1, int& p2)  
distancia entre los puntos más cercanos Cuáles son esos puntos más cercanos
```

constará de:

- Los límites c y f , que cumplen $0 \leq c \wedge f \leq longitud(p) - 1 \wedge f \geq c + 1$.
- La distancia d entre los puntos más cercanos. \rightarrow es la d , por eso es de entrada salida (se va actualizando)
- Los puntos p_1 y p_2 más cercanos.
- Un vector de posiciones $indY$ y un índice inicial ini que representa cómo se ordenan los elementos de p con respecto a la coordenada y . El índice inicial ini nos indica que el punto $p[ini]$ es el que tiene la menor coordenada y . El vector $indY$ contiene en cada posición la posición del siguiente punto en la ordenación, siendo -1 el valor utilizado para indicar que no hay siguiente. Por ejemplo, si el vector de puntos es:

$\{\{0.5, 0.5\}, \{0, 3\}, \{0, 0\}, \{0, 0.25\}, \{1, 1\}, \{1.25, 1.25\}, \{2, 2\}\}$

la variable ini valdrá 2 y el vector $indY$ será:

en $p[2]$ el y vale 0 es el menor

$\{4, -1, 3, 0, 5, 6, 1\}$

ordenarlo por $x \rightarrow 0, 0, 0, 0.5, 1...$

Esta ordenado a partir de la posición 2

Funciones auxiliares

En un ejercicio del juez nos va a preguntar por parejas invertidas (en sentido de orden) si hay un 2 primero y después un 1 será una pareja invertida.

```
double absolute(double x)
{ if (x>=0) {return x;}
  else {return -x;}
}
```

```
double distancia(Punto p1,Punto p2)
{
  return (sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y)));
};
```

Como calcular la distancia entre coordenadas

```
double minimo(double x, double y)
{ double z;
  if (x<=y) { z = x;} else {z=y;};
  return z;
};
```



ordenas una mitad, ordenas otra y compruebas si los de la izquierda son mayores que los de la derecha. Si si entonces forma pareja.

Caso base: 2 o 3 puntos

```
array de puntos    zona que estudiamos
void solucionDirecta(Punto p[], int c, int f, int indY[], int& ini,
{   double d1,d2,d3;           vector de      donde empieza el
    double& d, int& p1, int& p2) indirecciones. que tiene la y
    distancia      los dos puntos más pequeña.

    if (f==c+1)
    { d = distancia(p[c],p[f]);     calculamos la distancia
        if ((p[c].y) <= (p[f].y)) si la posicion de la y es más pequeña que el de la f
            entonces lo ordena.
            { ini=c; indY[c]=f; indY[f]=-1; p1=c; p2=f; }
        else
            {ini=f; indY[f]=c; indY[c]=-1; p1=f; p2=c; };
    }
    else if (f==c+2)
    {
        //Menor distancia y puntos que la producen
        d1=distancia(p[c],p[c+1]);
        d2=distancia(p[c],p[c+2]);
        d3=distancia(p[c+1],p[c+2]);
        d = minimo(minimo(d1,d2),d3);   mínimo entre las tres parejas de puntos
        if (d==d1) { p1=c; p2=c+1; }
        else if (d==d2) { p1=c; p2=c+2; }
        else { p1=c+1; p2=c+2; };
    }
}
```

Cuando solo hay 2 o 3 puntos estaríamos en el caso base y habría que proceder de esta misma forma.

Caso base: 2 o 3 puntos

Hace una distinción de casos para realizar la ordenación del eje y.

```
//Ordenar
if (p[c].y<=p[c+1].y) comparación de coordenadas del eje y.
{ if (p[c+1].y<=p[c+2].y)
    { ini=c; indY[c]=c+1; indY[c+1]=c+2; indY[c+2]=-1; }
    else if (p[c].y<=p[c+2].y)
    { ini=c; indY[c]=c+2; indY[c+2]=c+1; indY[c+1]=-1; }
    else
    { ini=c+2; indY[c+2]=c; indY[c]=c+1; indY[c+1]=-1; }
}
else
{
    if (p[c+1].y>p[c+2].y)
    { ini=c+2; indY[c+2]=c+1; indY[c+1]=c; indY[c]=-1; }
    else if (p[c].y>p[c+2].y)
    {ini=c+1; indY[c+1]=c+2; indY[c+2]=c; indY[c]=-1; }
    else
    {ini=c+1; indY[c+1]=c; indY[c]=c+2; indY[c+2]=-1; }
}
```

3 puntos.

Caso recursivo

```
void parMasCercano(Punto p[], int c, int f, int indY[], int& ini,
                    double& d, int& p1, int& p2)
{ int m; int i,j,ini1,ini2,p11,p12,p21,p22;double d1,d2;

    if (f-c+1<4) Esto es lo que dice la teoría de que si el número de puntos es menor que 4
        { solucionDirecta(p,c,f,indY,ini,d,p1,p2); }

    else Si hay más de 4 puntos
        { m = (c+f)/2;
            parMasCercano(p,c,m,indY,ini1,d1,p11,p12);
            parMasCercano(p,m+1,f,indY,ini2,d2,p21,p22);
            división en dos nubes de puntos
            if (d1<=d2)
                { d=d1; p1=p11; p2=p12; } guarda los puntos
            else
                { d=d2; p1=p21; p2=p22;};

    //Mezcla ordenada por la y
    mezclaOrdenada(p,ini1,ini2, indY, ini);
    dadas dos listas ordenadas mediante indirecciones obtener una lista TOTALMENTE ORDENADA.
```

Probablemente combina ambos resultados para obtener la solución final

primera mitad de la
nube de puntos del
dibujo que vimos arriba

segunda mitad de la
nube de puntos del
dibujo que hemos visto
anteriormente.

Caso recursivo

Tiene que ver con el eje X

```
//Filtrar la lista
i=ini;
while (absolute(p[m].x-p[i].x)>d) { i=indY[i]; };

int iniA=i;
int indF[f-c+1];
for (int l=0;l<=f-c+1;l++) { indF[l]=-1; };
int k=iniA;
while (i!=-1)
{
    Si está a distancia mayor pruebo el siguiente.
    if (absolute(p[m].x-p[i].x)<=d) { indF[k]=i; k=i; };
    i=indY[i];
};
```

En resumen, este fragmento de código identifica y almacena los índices de puntos en una lista que caen dentro de una franja específica determinada por la diferencia de coordenadas x entre el punto en la posición m y otros puntos, utilizando un valor de distancia d como límite para filtrar los puntos.

Caso recursivo

```
//Calcular las distancias
    i=iniA;
    while (i!=-1)
    {
        int count = 0; j=indF[i];
        while ((j!=-1)&&(count<7)) hasta el 7ptimo punto
        {
            double daux = distancia(p[i],p[j]);
            if (daux<d) { d=daux; p1=i; p2=j; }
            j=indF[j]; vete al siguiente del vector de indirecciones.
            count=count+1;
        };
        i=indF[i];
    };
}};


```

Mezcla ordenada

Es más sencillo que el merge.

```
void mezclaOrdenada(Punto p[], int ini1, int ini2,
                     int indY[], int& ini){
    int i=ini1; int j=ini2; int k;
    if (p[i].y<=p[j].y)
        { ini=ini1; k=ini1; i=indY[i]; }
    else
        { ini=ini2; k=ini2; j=indY[j]; }
    while ((i!=-1)&&(j!=-1)){
        if (p[i].y<=p[j].y)
            { indY[k] = i; k=i; i=indY[i]; }
        else
            { indY[k]=j; k=j; j=indY[j]; }
    };
    if (i== -1) { indY[k]=j; }
    else { indY[k]=i; };
}
```

Nos permite tener un algoritmo de coste $n \log n$ en vez de cuadrático.

Determinación del umbral

La determinación del umbral

- Dado un algoritmo DV, casi siempre existe una versión asintóticamente menos eficiente pero de constantes multiplicativas más pequeñas.
- Le llamaremos el algoritmo sencillo.
- Eso hace que para valores pequeños de n , sea más eficiente el algoritmo sencillo que el algoritmo DV. Determinar un valor de n para el cual no es más eficiente utilizar el DV que utilizar una función concreta
- Se puede conseguir un algoritmo óptimo combinando ambos algoritmos de modo inteligente Elegimos un tamaño para cortar de forma que sea al menos tan eficiente seguir dividiendo como aplicar el algoritmo base

```
Solucion divideYvenceras (Problema x, int n){  
    if (n <= n_0) umbral. Si es menor hace algo sencillo  
        { return algoritmoSencillo(x); }  
    else /* n > n_0 */ {  
        descomponer x;  
        llamadas recursivas a divideYvenceras;  
        y = combinar resultados;  
        return y;  
    }  
}
```

Si el número de elementos que quedan es menor que el umbral entonces hace el algoritmo sencillo

Podemos combinar un algoritmo de divide y vencerás con otra función que a partir de un umbral haga lo mismo pero menos difícil.

Afecta al coste de los casos pequeños.

Determinación del umbral

- La recursión exige más tiempo y espacio (pila) y además hay que dividir/componer (constante multiplicativa grande). ¿Cuándo merece la pena resolver el problema dividiendo? **Determinar n_0 .**

Determinar un umbral para el cual resolvamos el problema de otra forma (puede ser con una solución iterativa) de forma de que, a pesar de que el coste de esa función es mayor la eficiencia del algoritmo no varía debido a que los casos para los que se realiza son muy pequeños.

Determinación del umbral

- La recursión exige más tiempo y espacio (pila) y además hay que dividir/componer (constante multiplicativa grande). ¿Cuándo merece la pena resolver el problema dividiendo? **Determinar n_0 .**
- La determinación del umbral es un tema fundamentalmente **experimental**: depende del computador y lenguaje utilizados.
- El ideal sería que existiera un *umbral óptimo* n_0 tal que
 $n \leq n_0 \rightarrow$ es tan rápido llamar al subalgoritmo básico como dividir
 $n > n_0 \rightarrow$ más rápido dividir.
Pero este umbral óptimo no siempre existe.
- A pesar de eso, se puede hacer un estudio teórico del problema para encontrar un umbral aproximado.

Esto es pregunta del test

Determinación del umbral

- La recursión exige más tiempo y espacio (pila) y además hay que dividir/componer (constante multiplicativa grande). ¿Cuándo merece la pena resolver el problema dividiendo? **Determinar n_0 .** Para evitar ese tiempo y espacio determinamos el umbral.
- La determinación del umbral es un tema fundamentalmente **experimental**: depende del computador y lenguaje utilizados. Y no afecta al coste asintótico.
- El ideal sería que existiera un *umbral óptimo n_0* tal que
 $n \leq n_0 \rightarrow$ es tan rápido llamar al subalgoritmo básico como dividir
 $n > n_0 \rightarrow$ más rápido dividir.

Pero este umbral óptimo no siempre existe.
- A pesar de eso, se puede hacer un estudio teórico del problema para encontrar un umbral aproximado.
- Para fijar ideas, centrémonos en el problema de encontrar el par más cercano y escribamos su recurrencia con constantes multiplicativas (suponemos n potencia de 2):

$$T_1(n) = \begin{cases} c_0 & \text{si } 0 \leq n \leq 3 \\ 2T_1(n/2) + c_1 n & \text{si } n \geq 4 \end{cases}$$

O($n\log n$)

Determinación del umbral

- Si desplegamos esta recurrencia y la resolvemos exactamente, la expresión de coste resulta ser:

$$T_1(n) = c_1 n \log n + \left(\frac{1}{2}c_0 - c_1\right)n$$

Determinación del umbral

- Si desplegamos esta recurrencia y la resolvemos exactamente, la expresión de coste resulta ser:

$$T_1(n) = c_1 n \log n + \left(\frac{1}{2}c_0 - c_1\right)n$$

- Por otra parte, el algoritmo sencillo tendrá un coste $T_2(n) = c_2 n^2$. Las constantes c_0 , c_1 y c_2 dependen del lenguaje y de la máquina subyacentes, y han de ser determinadas experimentalmente para cada instalación.

Determinación del umbral

- Si desplegamos esta recurrencia y la resolvemos exactamente, la expresión de coste resulta ser:

$$T_1(n) = c_1 n \log n + \left(\frac{1}{2}c_0 - c_1\right)n$$

- Por otra parte, el algoritmo sencillo tendrá un coste $T_2(n) = c_2 n^2$. Las constantes c_0 , c_1 y c_2 dependen del lenguaje y de la máquina subyacentes, y han de ser determinadas experimentalmente para cada instalación.
- Aparentemente, para encontrar el umbral hay que resolver la ecuación $T_1(n) = T_2(n)$, es decir encontrar un n_0 que satisfaga:

$$c_1 n \log n + \left(\frac{1}{2}c_0 - c_1\right)n = c_2 n^2$$

Determinación del umbral

- Si desplegamos esta recurrencia y la resolvemos exactamente, la expresión de coste resulta ser:

$$T_1(n) = c_1 n \log n + \left(\frac{1}{2}c_0 - c_1\right)n$$

cuadrática, todos con todos.

- Por otra parte, el algoritmo sencillo tendrá un coste $T_2(n) = c_2 n^2$. Las constantes c_0 , c_1 y c_2 dependen del lenguaje y de la máquina subyacentes, y han de ser determinadas experimentalmente para cada instalación. *No pasa nada porque es para una n muy pequeña.*
- Aparentemente, para encontrar el umbral hay que resolver la ecuación $T_1(n) = T_2(n)$, es decir encontrar un n_0 que satisfaga:

$$c_1 n \log n + \left(\frac{1}{2}c_0 - c_1\right)n = c_2 n^2$$

ESTO ES LO QUE NO HAY QUE HACER. IGUALAR LOS DOS T'S

- Sin embargo, este planteamiento es **incorrecto** porque el coste del algoritmo DV está calculado subdividiendo n hasta los casos base.
- Es decir, estamos comparando el algoritmo DV puro con el algoritmo sencillo puro y lo que queremos saber es cuándo subdividir es más costoso que no subdividir.

Determinación del umbral

- La ecuación que necesitamos es la siguiente:

$$2T_2(n/2) + c_1n = c_2n^2 = T_2(n)$$

que expresa que en una llamada recursiva al algoritmo DV decidimos subdividir **por última vez** porque es tan costoso subdividir como no hacerlo.

- Nótese que el coste de las dos llamadas internas está calculado con el algoritmo sencillo, lo que confirma que esta subdivisión es la última que se hace.

Determinación del umbral

Idea es comparar la recurrencia. En qué tamaño es igual dividir y combinar, con aplicar directamente el algoritmo sencillo.

- La ecuación que necesitamos es la siguiente:

$$2T_2(n/2) + c_1n = c_2n^2 = T_2(n)$$

que expresa que en una llamada recursiva al algoritmo DV decidimos subdividir **por última vez** porque es tan costoso subdividir como no hacerlo.

- Nótese que el coste de las dos llamadas internas está calculado con el algoritmo sencillo, lo que confirma que esta subdivisión es la última que se hace.
- Resolviendo esta ecuación obtenemos: Es el punto en el cual tenemos que cortar.

$$2c_2 \left(\frac{n}{2}\right)^2 + c_1n = c_2n^2 \Rightarrow n_0 = \frac{2c_1}{c_2}$$

- Para $n > n_0$, la expresión de la izquierda crece más despacio que la de la derecha y merece la pena subdividir.
- Para valores menores que n_0 , la expresión de la derecha es menos costosa.

Determinación del umbral

- Como sabemos, c_1 mide el número de operaciones elementales que hay que hacer con cada punto de la nube de puntos en la parte no recursiva del algoritmo DV.
- Es decir, la suma por punto de dividir la lista en dos, mezclar las dos mitades ordenadas, filtrar los puntos de la banda y recorrer la misma, comparando cada punto con otros siete.

Determinación del umbral

- Como sabemos, c_1 mide el número de operaciones elementales que hay que hacer con cada punto de la nube de puntos en la parte no recursiva del algoritmo DV.
- Es decir, la suma por punto de dividir la lista en dos, mezclar las dos mitades ordenadas, filtrar los puntos de la banda y recorrer la misma, comparando cada punto con otros siete.
- Por su parte, c_2 mide el coste elemental de cada una de las n^2 operaciones del algoritmo sencillo.
- Este coste consiste en esencia en la mitad de calcular la distancia entre dos puntos y comparar con el mínimo en curso.

Determinación del umbral

- Como sabemos, c_1 mide el número de operaciones elementales que hay que hacer con cada punto de la nube de puntos en la parte no recursiva del algoritmo DV.
- Es decir, la suma por punto de dividir la lista en dos, mezclar las dos mitades ordenadas, filtrar los puntos de la banda y recorrer la misma, comparando cada punto con otros siete.
- Por su parte, c_2 mide el coste elemental de cada una de las n^2 operaciones del algoritmo sencillo.
- Este coste consiste en esencia en la mitad de calcular la distancia entre dos puntos y comparar con el mínimo en curso.
- Supongamos que, una vez medidas experimentalmente, obtenemos $c_1 = 32c_2$. Ello nos daría un umbral $n_0 = 64$.

Esto es una hipótesis.

Determinación del umbral

- Es interesante escribir y resolver la recurrencia del algoritmo híbrido así conseguido y comparar el coste con el del algoritmo DV original:

$$T_3(n) = \begin{cases} c_2 n^2 & \text{si } n \leq 64 \\ 2T_3(n/2) + c_1 n & \text{si } n > 64 \end{cases}$$

la cambiamos la recurrencia. No siempre existe una solución óptima.

Determinación del umbral

- Es interesante escribir y resolver la recurrencia del algoritmo híbrido así conseguido y comparar el coste con el del algoritmo DV original:

$$T_3(n) = \begin{cases} c_2 n^2 & \text{si } n \leq 64 \\ 2T_3(n/2) + c_1 n & \text{si } n > 64 \end{cases}$$

- Si desplegamos i veces, obtenemos:

$$T_3(n) = 2^i T_3\left(\frac{n}{2^i}\right) + i c_1 n$$

que alcanza el caso base cuando $\frac{n}{2^i} = 2^6 \Rightarrow i = \log n - 6$.

Determinación del umbral

- Es interesante escribir y resolver la recurrencia del algoritmo híbrido así conseguido y comparar el coste con el del algoritmo DV original:

$$T_3(n) = \begin{cases} c_2 n^2 & \text{si } n \leq 64 \\ 2T_3(n/2) + c_1 n & \text{si } n > 64 \end{cases}$$

- Si desplegamos i veces, obtenemos:

$$T_3(n) = 2^i T_3\left(\frac{n}{2^i}\right) + i c_1 n$$

que alcanza el caso base cuando $\frac{n}{2^i} = 2^6 \Rightarrow i = \log n - 6$.

- Entonces sustituimos i :

$$\begin{aligned} T_3(n) &= \frac{n}{2^6} T_3(2^6) + c_1(\log n - 6)n \\ &= c_1 n \log n + c_2 \frac{n}{2^6} 2^{12} - 6c_1 n \\ &= c_1 n \log n - 4c_1 n \end{aligned}$$

Determinación del umbral

- Es interesante escribir y resolver la recurrencia del algoritmo híbrido así conseguido y comparar el coste con el del algoritmo DV original:

$$T_3(n) = \begin{cases} c_2 n^2 & \text{si } n \leq 64 \\ 2T_3(n/2) + c_1 n & \text{si } n > 64 \end{cases}$$

- Si desplegamos i veces, obtenemos:

$$T_3(n) = 2^i T_3\left(\frac{n}{2^i}\right) + i c_1 n$$

que alcanza el caso base cuando $\frac{n}{2^i} = 2^6 \Rightarrow i = \log n - 6$.

- Entonces sustituimos i :

$$\begin{aligned} T_3(n) &= \frac{n}{2^6} T_3(2^6) + c_1(\log n - 6)n \\ &= c_1 n \log n + c_2 \frac{n}{2^6} 2^{12} - 6c_1 n \\ &= c_1 n \log n - 4c_1 n \end{aligned}$$

- Comparando el coste $T_3(n)$ del algoritmo híbrido con el coste $T_1(n)$ del algoritmo DV puro, se aprecia una diferencia importante en la constante multiplicativa del término de segundo orden.
Para esto habrá que hacer pruebas experimentales.

Se rebaja el coste para los casos pequeños.