El esquema *Divide y vencerás*

Isabel Pita. 2017

Facultad de Informática - UCM

21 de noviembre de 2017

Bibliografía recomendada

- Estructuras de datos y métodos algorítmicos. 213 ejercicios resueltos. *Narciso Martí Oliet, Yolanda Ortega Mallén, y José Alberto Verdejo López*. Ibergaceta Publicaciones, 2º edición 2013.
- Estructuras de datos y métodos algorítmicos. Ejercicios resueltos. *Narciso Martí Oliet, Yolanda Ortega Mallén, y José Alberto Verdejo López*. Colección Prentice Práctica, Pearson Prentice-Hall, 2010.

Capítulo 11. Ejercicios resueltos: 11.4, 11.9, 11.13, 11.15, 11.16, 11.20, 11.21, 11.22, 11.23

Objetivos

1. Esquema algorítmico de Divide y Vencerás:
   1. Búsqueda binaria
   2. Ordenación rápida (quicksort)
   3. Ordenación por mezclas (mergesort)
   4. Problema del par de puntos más cercanos.

Problemas propuestos de acepta el reto

- 230. Desordenes temporales
- 295. Elévame
- 306. Dos igualdades sorprendentes

Introducción

- Los esquemas algorítmicos son estrategias de resolución de problemas.
- Se aplican en la resolución de problemas que presentan unas características comunes.
- Esquemas algorítmicos más comunes:
  - Divide y vencerás,
  - vuelta atrás.
  - método voraz,
  - programación dinámica,
  - ramificación y poda.

- El esquema divide y vencerás (DV) es un caso particular del diseño recursivo.
- Ha 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. Los parámetros de una llamada no pueden depender de los resultados de otra previa.
  - 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.

- Anticipar el coste de la solución DV. Si el coste sale igual o peor que el de un algoritmo ya existente, entonces no merece la pena aplicar DV.
- Para saber si la aplicación del esquema DV a un problema dado resultará en una solución eficiente o no, se deberá utilizar la recurrencia vista en el Capítulo 4 en la que el tamaño del problema disminuía mediante división:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

- Solución:

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

Implementación recursiva de la búsqueda binaria

Algoritmo que decide si un valor está o no en un vector. El vector ordenado en orden estrictamente creciente.

Especificación.

```
method search (v:array<int>, x:int) returns (b:bool)
requires v != null
requires forall u,w:0<=u w<v.Length ==> v[u]<v[w]
ensures b == (x in v[..])
```

Implementación recursiva de la búsqueda binaria

Planteamiento recursivo.
Contamos con una función `search(v,ini,fin,x)` tal que si $fin - ini < v.Length$ devuelve cierto si el elemento `x` esta en el vector `v` entre los índices `ini` y `fin` y falso en caso contrario.

- Primera aproximación: Comparar el elemento de la posición `fin-1` del vector con el elemento buscado y si no es igual utilizar la función `search(v,ini,fin-1,x)` para buscar en el intervalo $[ini..fin - 1)$.
- El coste de esta solución será lineal. Plantear la recurrencia y resolverla.
- ¿Podemos aprovechar el hecho de que el vector está ordenado para mejorar el coste del algoritmo?.

Implementación recursiva de la búsqueda binaria

Planteamiento recursivo.

- Segunda aproximación: Dividir el vector por la mitad y buscar solo en una de las dos partes.
- Estrategia recursiva:
  $search(v, ini, fin, x) =$

$$\begin{cases} v[m] > x & search(v, ini, m, x) \\ v[m] = x & \texttt{caso base} \\ v[m] < x & search(v, m+1, fin, x) \end{cases}$$

donde $m = (ini + fin + 1)/2$

La función auxiliar obtenida es:

```
bool search (std::vector<int> const& v, int ini,
      int fin, int x ) {
  if (ini == fin) return false; // vector vacio
  else if (ini+1 == fin)  // vector de 1 elemento
    return v[ini] == x;
  else  { int m = (ini + fin + 1) / 2;
     if (v[m] == x) return true;
     else if (v[m] > x)  return search(v,ini,m,x);
     else return search(v,m+1,fin,x);
}
```

La función principal se reduce a llamar a la función auxiliar con los parámetros adecuados.

```
bool search (std::vector<int> const& v,  int x ) {
  return search(v,0,v.size(),x);
}
```

El coste de la función está en $\mathcal{O}(\log n)$

¿Mejora el algoritmo si dividimos el vector de otras formas?

Caso 1. Dividimos el vector en subvectores de tamaño $\frac{1}{3}$ y $\frac{2}{3}$.

```
bool search2 (std::vector<int> const& v, int ini,
         int fin, int x ) {
  if ( ini == fin )  return false; // vector vacio
  else if (ini+1 == fin) // vector de 1 elemento
    return v[a] == x;
  else {  int t = ini + (fin - ini) / 3;
     if (v[t] == x)  return true;
     else if (v[t]>x)  return search2(v,ini,t,x);
     else return search2(v,t+1,fin,x); }
}
```

Coste en el caso peor (Tamaño de los datos: $n = fin - ini$):

$$T(n) = \begin{cases} c_1 & \text{si } n = 0, 1 \\ T(2n/3) + c & \text{si } n > 1 \end{cases} \in \mathcal{O}(\log_{3/2} n) \equiv \mathcal{O}(\log n)$$

Como $\frac{3}{2} < 2$ se tiene $\log_{3/2} n > \log_2 n$. Luego aunque el orden de complejidad es el mismo, la constante multiplicativa es mayor.

- Caso 2. Número de elementos del vector *num*.
  Dividimos el vector en subvectores de tamaño k y $num - k$.
- Coste en el caso peor (Tamaño de los datos: $n = num$):

$$T(n) = \begin{cases} c_1 & \text{si } n = 0, 1 \\ T(max(\frac{k}{num}, \frac{num-k}{num})n) + c & \text{si } n > 1 \end{cases}$$

$$\in \mathcal{O}(\log_{\frac{num}{max(k,num-k)}} n) \equiv \mathcal{O}(\log n)$$

- Como

$$\frac{num}{max(k,num-k)} < 2$$

  se tiene

$$\log_{num/max(k,num-k)} n > \log_2 n.$$

  Luego el coste menor se obtiene dividiendo el vector por la mitad.

Ampliación. Búsqueda ternaria

- Algoritmo que divide el vector en 3. El orden de complejidad es logarítmico pero la constante es mayor. Estudiar a partir de que n puede ser mejor dividir en 3 o si no depende de la n

Ejemplos de aplicación del esquema con éxito

Ordenación rápida (quicksort). La solución en el caso mejor responde al esquema DV.

```
method quicksort (v : array<int>)
requires v != null
ensures forall u, w::0<=u<w<v.Length ==> v[u] <= v[w]
modifies v
```

Método auxiliar

```
method quicksort (v : array<int>, ini : int, fin : int)
requires v != null
requires 0 <= ini <= fin <= v.Length
ensures forall u, w::0<=u<w<v.Length ==> v[u] <= v[w]
modifies v
```

```
void quicksort (std::vector<T> const& v) {
   quicksort(v,0,v.size());
}
void quicksort (std::vector<T> const&v,int ini,int fin)
{ ... }
```

- Planteamiento recursivo: Tenemos una función `quicksort(v,ini,fin)` que ordena las componentes de un vector entre `ini` y `fin`, siendo *fin* − *ini* menor que el tamaño del vector.
  - Elegir un pivote: un elemento cualquiera del subvector $v[ini..fin)$. Normalmente se elige `v[ini]`.
  - Particionar el subvector $v[ini..fin)$, colocando a la izquierda los elementos menores que el pivote y a la derecha los mayores. Los elementos iguales al pivote pueden quedar indistintamente a la izquierda o a la derecha. Al final del proceso de partición, el pivote debe quedar en el centro, separando los menores de los mayores.
  - Ordenar los dos fragmentos que han quedado a la izquierda y a la derecha del pivote.

- Suponiendo que tenemos una implementación correcta de *particion*, el algoritmo nos queda:

```
void quickSort( std::vector<T> const& v,
                int ini, int fin ) {
  int  p;
  if ( ini + 1 < fin ) { // mas de un elemento
    particion(v, ini, fin, p);
    quickSort(v, ini, p);
    quickSort(v, p+1, fin);
  }
}
```

- Ordenación por mezcla (*mergesort*)

```
method mergeSort (v : array<int>)
requires v != null
ensures forall u, w::0<=u<w<v.Length ==> v[u] <= v[w]
modifies v
```

Método auxiliar

```
method mergeSort (v : array<int>, ini : int, fin : int)
requires v != null
requires 0 <= ini <= fin <= v.Length
ensures forall u, w::0<=u<w<v.Length ==> v[u] <= v[w]
modifies v
```

```
void mergesort (std::vector<T> const& v) {
   mergeSort(v,0,v.size());
   }

void mergeSort(std::vector<T> const& v,int ini,int fin)
 { ... }
```

- Planteamiento recursivo. Para ordenar el subvector $v[ini..fin)$
  - Obtenemos el punto medio m entre ini y fin, y ordenamos recursivamente los subvectores $v[ini..m+1)$ y $v[(m+1)..fin)$.
  - Mezclamos ordenadamente los subvectores $v[ini..m+1)$ y $v[(m+1)..fin)$ ya ordenados.

Algoritmo:

```
void mergeSort( std::vector<T> const&  v,
     int ini, int fin ) {
  int  m;
  if ( ini +1 < fin  ) { // mas de un elemento
    m = (ini+fin+1) / 2;
    mergeSort( v, ini, m+1 );
    mergeSort( v, m+1, fin );
    mezcla( v, ini, m, fin );
  }
}
```

- En algunas aplicaciones se necesita conocer la ordenación de los elementos de un vector, pero se desea mantener el vector sin modificar. (problema 11.6, Libro Estruturas de datos y métodos algoríticos.)
- El algoritmo de ordenación devuelve un vector de índices tal que la componente i-ésima indica la posición en el vector de entrada del elemento que debe ocupar el i-ésimo lugar en la ordenación.

v :	5	7	2	9	1
mid :	4	2	0	1	3

```
void mergeSort( std::vector<T> const& v, int ini,
      int fin, std::vector<int> &ind ) {
  int  m;
  if (ini+1 == fin) ind[ini] = ini; // 1 elemento
  else if (ini+1 < fin) { // mas de 1 elemento
    m = (ini+fin+1) / 2;
    mergeSort( v, ini, m+1 , ind);
    mergeSort( v, m+1, fin , ind);
    mezcla( v, ini, m, fin , ind);
  }
}
```

- Un problema históricamente famoso es el de la solución DV a la transformada discreta de Fourier (DFT), dando lugar al algoritmo conocido como transformada rápida de Fourier, o FFT (J.W. Cooley y J.W. Tukey, 1965).
- La transformada discreta convierte un conjunto de muestras de amplitud de una señal, en el conjunto de frecuencias que resultan del análisis de Fourier de la misma.
- Esta transformación y su inversa (que se realiza utilizando el mismo algoritmo DFT) tienen gran interés práctico, pues permiten filtrar frecuencias indeseadas (p.e. ruido) y mejorar la calidad de las señales de audio o de vídeo.

- La transformada en esencia multiplica una matriz $n \times n$ de números complejos por un vector de longitud n de coeficientes reales, y produce otro vector de la misma longitud.
- El algoritmo clásico realiza esta tarea del modo obvio y tiene un coste $\mathcal{O}(n^2)$.
- La FFT descompone de un cierto modo el vector original en dos vectores de tamaño $n/2$, realiza la FFT de cada uno, y luego combina los resultados de tamaño $n/2$ para producir un vector de tamaño n.
- Las dos partes no recursivas tienen coste lineal, dando lugar a un algoritmo FFT de coste $\mathcal{O}(n \log n)$.
- El algoritmo se utilizó por primera vez para analizar un temblor de tierra que tuvo lugar en Alaska en 1964.
- El algoritmo clásico empleó 26 minutos en analizar la muestra, mientras que la FFT de Cooley y Tukey lo hizo en 6 segundos.

Problema de selección

- Dado un vector v de n elementos que se pueden ordenar, y un entero $1 \leq k \leq n$, encontrar el k-ésimo menor elemento.
- Encontrar la mediana de un vector consiste en encontrar el elemento $(n-1)/2$ menor.
- Primera solución: ordenar el vector y tomar el elemento $v[k]$. Complejidad: la del algoritmo de ordenación utilizado.

- Segunda solución: utilizar el algoritmo *particion* de quicksort con algún elemento del vector:
  - Si la posición p donde se coloca el pivote es igual a k, entonces $v[p]$ es el elemento que estamos buscando.
  - Si $k < p$ pasar a buscar el k-ésimo elemento en las posiciones anteriores a p.
  - Si $k > p$ pasar a buscar el k-ésimo elemento en las posiciones posteriores a p.
- Implementación: generalizar el problema con dos parámetros *ini* y *fin*, que nos indican la parte del vector que nos interesa en cada momento. La llamada inicial que deseamos es $seleccion(v, 0, v.size(), k)$.
- La posición k es una posición absoluta dentro del vector. Se puede escribir una versión alternativa en la que k hace referencia a la posición relativa dentro del subvector que se está tratando.

```
T seleccion1(std::vector<T> const& v,
      int ini, int fin, int k)
{
 int p;
 if (ini+1==fin) {return v[ini];} // 1 elemento
 else
  { particion(v,ini,fin,p);
   if (k==p) {return v[p];}
   else if (k<p) { return seleccion1(v,ini,p,k);}
   else {return seleccion1(v,p+1,fin,k);}
  }
};
```

Caso peor: el pivote queda siempre en un extremo del subvector correspondiente. El coste está en $O(n^2)$ siendo $n = fin - ini$ el tamaño del vector.

- Si usásemos la mediana del vector como pivote el tamaño del problema se dividiría por la mitad, lo que nos da un coste en $O(n)$ siendo n el tamaño del vector.
- El problema de la mediana es un caso particular del problema que estamos intentando resolver y del mismo tamaño.
- Es suficiente una aproximación a la mediana, conocida como *mediana de medianas*, para obtener el coste lineal.
- Para calcularla se divide el vector en trozos consecutivos de 5 elementos, y se calcula directamente la mediana para cada uno de ellos. Después, se calcula recursivamente la mediana de esas n *div* 5 medianas mediante el algoritmo de selección. Con este pivote se puede demostrar que el caso peor anterior ya no puede darse.

Pasos del nuevo algoritmo, *seleccion2*:

1. Calcular la mediana de cada grupo de 5 elementos. En total $n/5$ medianas, y cada una se puede calcular en tiempo constante: ordenar los 5 elementos, y quedarnos con el tercero. Para no usar espacio adicional, dichas medianas se trasladan al principio del vector.
2. calcular la mediana de las medianas, *mm*, con una llamada recursiva a *seleccion2* con $n/5$ elementos.
3. llamar a *particion2 (v,a,b,mm,p,q)*, utilizando como pivote *mm*.
4. hacer una distinción de casos similar a la de *seleccion1*:
5. Es necesario elegir adecuadamente los casos base, ya que si hay 12 elementos o menos, es decir $b - a + 1 \leq 12$, es mas costoso seguir el proceso recursivo que ordenar directamente y tomar el elemento k.
6. Se puede demostrar, por inducción constructiva, que el tiempo requerido por *seleccion2* en el caso peor es lineal en $n = b - a + 1$, (Fundamentos de algoritmia. G. Brassard).

```
T seleccion2(std::vector<T> const& v, int ini, int fin,
   int k)
{int p,q,s;
 int t = fin-ini;
 if (t <=12) {ordenarInsercion(v,ini,fin);return v[k];}
 else { s = t / 5;
   for (int l=1; l<=s;++l) {
    ordenarInsercion(v,ini+5*(l-1),ini+5*l-1);
    int pm = ini+5*(l-1)+(5 / 2);
    std::swap(v[ini+l-1],v[pm]);
  };
  T mm=seleccion2(v,ini,ini+s,ini+(s-1)/2);
  particion2(v,ini,fin,mm,p,q);
  if ((k>=p) && (k<=q)) {return mm;}
  else if (k<p) { return seleccion2(v,ini,p,k);}
  else {return seleccion2(v,q+1,fin,k);}
 }
};
```

El problema del par más cercano

- Dada una nube de n puntos en el plano, $n \geq 2$, 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).
- Aplicación: en un sistema de control del tráfico aéreo, el par más cercano nos informa del mayor riesgo de colisión entre dos aviones.
- La distancia Euclidea de dos puntos $p_1 = (x_1, y_1)$ y $p_2 = (x_2, y_2)$, se define como:

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

- 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**.

Enfoque DV: encontrar el par más cercano a partir de los pares más cercanos de conjuntos de puntos que sean una fracción del original.
Estrategia :

- Ordenar los puntos por la coordenada x
- Dividir los puntos por la mitad: izquierda I, y derecha D.
- Resolver recursivamente los problemas I y D. Sean δ_I y δ_D las respectivas distancias mínimas encontradas y sea $\delta = \text{mín}(\delta_I, \delta_D)$.
- 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.
- Basta con comprobar los puntos que se hallan a lo sumo a una distancia δ de la línea que separa las dos mitades.

Coste esperado de esta estrategia.

- Ordenación de los puntos por la coordenada de x: $\Theta(n \log n)$ en el caso peor. Se puede realizar fuera del algoritmo recursivo.
- División de la nube de puntos: $\Theta(1)$
- Filtrado de los puntos de I y D para conservar sólo los que estén en la banda vertical de anchura 2δ y centrada en la mitad: $\Theta(n)$
- Si calculamos la distancia de cada punto del lado izquierdo a cada punto del lado derecho el coste es: $\Theta(n^2)$. Todos los puntos pueden estar dentro de la banda.

Si los puntos de la banda están ordenados por la coordenada y, si un punto está a una distancia de otro menor que δ, este debe estar entre los 7 siguientes.

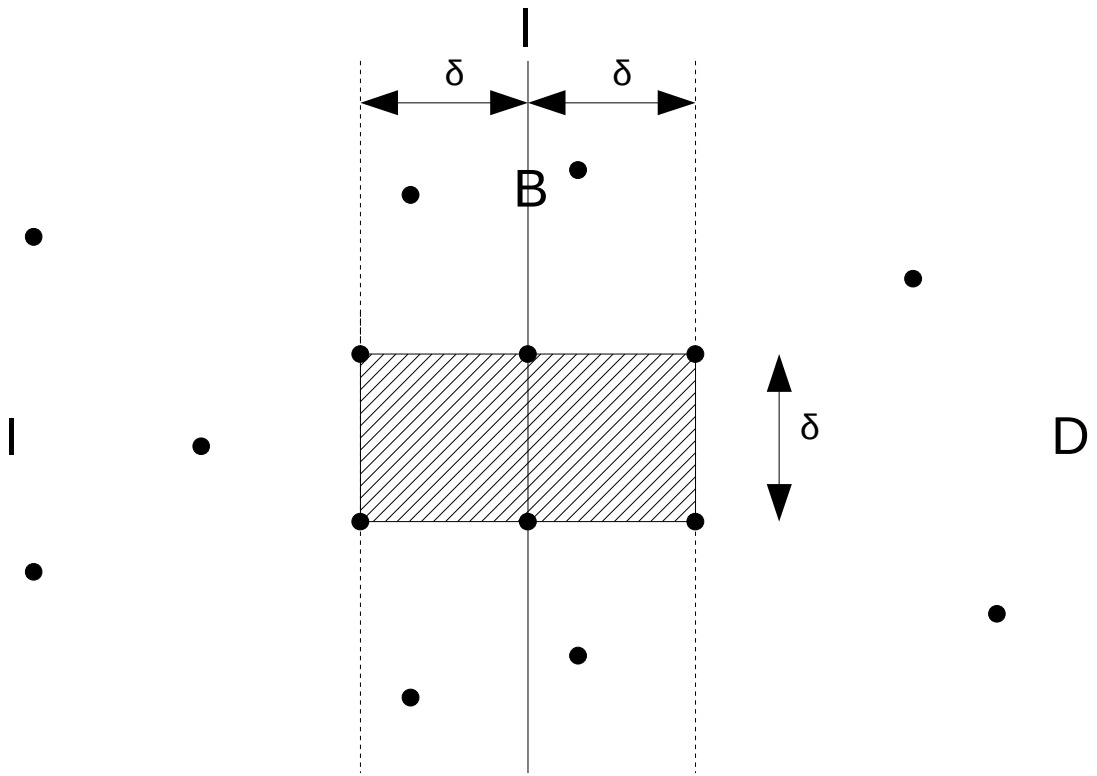

Figura 1: Razonamiento de corrección del problema del par más cercano

- Si ordenáramos $B_I \cup B_D$ en cada llamada recursiva, gastaríamos un coste $\Theta(n \log n)$ en cada llamada, lo que conduciría a un coste total en $\Theta(n \log^2 n)$.
- Cada llamada recursiva puede devolver un resultado extra: la lista de sus puntos ordenada por la coordenada y. Se puede hacer aplicando el algoritmo de mezcla de dos listas ordenadas en tiempo $\Theta(n)$.

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

void parMasCercano(std::vector<Punto> const& p,
        int c, int f, std::vector<int> &indY, int& ini,
        double& d, int& p1, int& p2)
```

Parámetros de entrada: vector p de puntos; límites c y f de la parte del vector que estamos considerando. El vector de puntos no se modificará en ningún momento y se supone ordenado respecto a la coordenada x.
Parámetros de salida:

- La distancia d entre los puntos más cercanos.
- 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.

Se consideran casos base, cuando hay 2 o 3 puntos

```
void solucionDirecta(std::vector<Punto> const& p,
        int c, int f, std::vector<int> &indY, int& ini,
        double& d, int& p1, int& p2)
{  double d1,d2,d3;
    if (f==c+2) // dos elementos
     { d = distancia(p[c],p[f-1]);
       if ((p[c].y) <= (p[f-1].y))
         {ini=c;indY[c]=f;indY[f-1]=-1;p1=c;p2=f-1;}
       else
         {ini=f-1; indY[f-1]=c; indY[c]=-1;p1=f;p2=c;}
     }
    else if (f==c+3)
    {
    ....
    }
};
```

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

Algoritmo:

```
void parMasCercano(std::vector<Punto> const& p,
     int c, int f, std::vector<int> &indY, int& ini,
     double& d, int& p1, int& p2)
{
  int i,j,ini1,ini2,p11,p12,p21,p22;double d1,d2;
  if (f-c < 4)
    solucionDirecta(p,c,f,indY,ini,d,p1,p2);
  else {
    int m = (c+f+1)/2;
    parMasCercano(p,c,m+1,indY,ini1,d1,p11,p12);
    parMasCercano(p,m+1,f,indY,ini2,d2,p21,p22);
    if (d1<=d2)  {d=d1;p1=p11;p2=p12;}
    else  {d=d2;p1=p21;p2=p22;};

  //Mezcla ordenada por la y
  mezclaOrdenada(p,ini1,ini2, indY, ini);
```

```
 //Filtrar la lista
 i=ini;
while (absolute(p[m].x-p[i].x)>d){i=indY[i];};
int iniA=i;
 int aux[f-c];
for (int l=0;l<=f-c;l++){aux[l]=-1;};
while (i!=-1)
{
  if (abs(p[m].x-p[i].x)<=d){aux[k]=i;k=i;};
   i=indY[i];
};
```

```
    //Calcular las distancias
     i=ini;
    while (i!=-1)
    {
      int count = 0;   j=aux[i];
      while ((j!=-1)&&(count<7))
      {
        double daux = distancia(p[i],p[j]);
        if (daux<d) {d=daux; p1=i; p2=j;}
        j=aux[j];
        count=count+1;
      };
      i=aux[i];
    };
  }
};
```

La determinación del umbral

- Dado un algoritmo DV, casi siempre existe otro algoritmo asintóticamente menos eficiente pero de constantes multiplicativas más pequeñas que resuelve el mismo problema. Le llamaremos el *algoritmo sencillo*.
- Para valores pequeños de n, será más eficiente el algoritmo sencillo que el algoritmo DV.
- Se puede conseguir un algoritmo óptimo combinando ambos algoritmos. Se convierten en casos base del algoritmo recursivo los problemas que son *suficientemente* pequeños.

- Determinar **el umbral** n_0 a partir del cual compensa utilizar el algoritmo sencillo con respecto a continuar subdividiendo el problema.
- La determinación del umbral es un tema fundamentalmente **experimental**, depende del computador y lenguaje utilizados, e incluso puede no existir un óptimo único sino varios en función del tamaño del problema.
- Buscamos un umbral aproximado mediante un estudio teórico del problema.
- Ejemplo: problema del par mas cercano.

Recurrencia (suponemos n potencia de 2):

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

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

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

- 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:

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

- 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.

- La ecuación que necesitamos es la siguiente:

$$2T_2(n/2) + c_1 n = c_2 n^2 = T_2(n)$$

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:

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

- 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.

- 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 $n_0 = 64$.

- 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:

$$T_3(n) = 2^i T_3\left(\frac{n}{2^i}\right) + ic_1 n$$

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

- Entonces sustituimos i:

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

- Comparando el coste $T_3(n)$ del algoritmo hibrido con el coste $T_1(n)$ del algoritmo DV puro, se aprecia una diferencia importante en la constante multiplicativa del término de segundo orden.