El esquema "Divide y vencerás"

Ricardo Peña es el autor principal de este tema

Facultad de Informática - UCM

1 de diciembre de 2016

Bibliografía Recomendada

- **Fundamentos de Algoritmia**. *G. Brassard y P. Bratley*. Prentice Hall, 1997.
- **Introduction to Algorithms**. *T.H. Cormen, C.E. Leiserson, R.L. Rivest y C. Stein*. MIT Press, segunda edición, 2001.
- **Estructuras de datos y métodos algorítmicos: ejercicios resueltos**. *N. Martí Oliet, Y. Ortega Mallén y J.A. Verdejo López*. Pearson-Prentice Hall, 2004.
- **Especificación, Derivación y Análisis de Algoritmos: ejercicios resueltos**. *N. Martí Oliet, C.M. Segura Díaz y J.A. Verdejo López*. Colección Prentice Práctica, Pearson-Prentice Hall, 2006.

- **Estructuras de datos: un enfoque moderno**. *Mario Rodríguez Artalejo, Pedro Antonio González Calero y Marco Antonio Gómez Martín*. Editorial Complutense, 2011.
- **Diseño de Programas: Formalismo y Abstracción**. *Ricardo Peña*. Tercera edición, Pearson Prentice-Hall, 2005.
- **The C++ Programming Language, 3rd Edition**. *Bjarne Stroustrup*. Addison-Wesley, 1998.

Introducción

- En este capítulo iniciamos la presentación de un conjunto de *esquemas algorítmicos* que pueden emplearse como estrategias de resolución de problemas.
- Un esquema puede verse como un algoritmo *genérico* que puede resolver distintos problemas.
- Si se concretan los tipos de datos y las operaciones del esquema genérico con los tipos y operaciones específicos de un problema concreto, tendremos un algoritmo para resolver dicho problema.

- Además de *divide y vencerás*, este curso veremos el esquema de *vuelta atrás*.
- En cursos posteriores aparecerán otros esquemas con nombres propios tales como el *método voraz*, el de *programación dinámica* y el de *ramificación y poda*.
- Cada uno de ellos resuelve una familia de problemas de características parecidas.

- Los esquemas o métodos algorítmicos deben verse como un conjunto de algoritmos *prefabricados* que el diseñador puede ensayar ante un problema nuevo.
- No hay garantía de éxito, pero si se alcanza la solución, el esfuerzo invertido habrá sido menor que si el diseño se hubiese abordado desde cero.

- 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.
- Aparentemente estas son las características generales de todo diseño recursivo, y de hecho el esquema DV es un caso particular del mismo.

- 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.

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

```
template <class Problema, class Solución>
Solucion divide-y-vencerás (Problema x) {
Problema x_1,...,x_k;
Solución y_1,...y_k;

   if (base(x))
         return método-directo(x);
   else {
         (x_1,..., x_k) = descomponer(x);
         for (i=1; i<=k; i++)
               y_i = divide-y-vencerás(x_i);
         return combinar(x, y_1,..., y_k);
        }
}
```

- Los tipos `Problema`, `Solución`, y los métodos `base`, `método-directo`, `descomponer` y `combinar`, son específicos de cada problema resuelto por el esquema.

- 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 en el Capítulo 4 en la que el tamaño del problema disminuía *mediante división*:

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

- 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 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.

Ejemplos de aplicación del esquema con éxito

- Algunos de los algoritmos recursivos vistos hasta ahora encajan perfectamente en el esquema DV.
- La **búsqueda binaria** en un vector ordenado vista en el Cap. 4 es un primer ejemplo.
- En este caso, la operación `descomponer` selecciona una de las dos mitades del vector y la operación `combinar` es vacía.
- Obteníamos los siguientes parámetros de coste:

  $b = 2$ Tamaño mitad del subvector a investigar en cada llamada recursiva.

  $a = 1$ Un subproblema a lo sumo.

  $k = 0$ Coste constante de la parte no recursiva.

  dando un coste total $O(\log n)$.

- La **ordenación mediante mezcla** o *mergesort* también responde al esquema: la operación `descomponer` divide el vector en dos mitades y la operación `combinar` mezcla las dos mitades ordenadas en un vector final.
- Los parámetros del coste son:

  $b = 2$ Tamaño mitad de cada subvector.
  $a = 2$ Siempre se generan dos subproblemas.
  $k = 1$ Coste lineal de la parte no recursiva (la mezcla).

  dando un coste total $O(n \log n)$.

- La **ordenación rápida** o *quicksort*, considerando sólo el caso mejor, también responde al esquema.
- La operación `descomponer` elige el pivote, particiona el vector con respecto a él, y lo divide en dos mitades. La operación `combinar` en este caso es vacía.
- Los parámetros del coste son:

  $b = 2$ Tamaño mitad de cada subvector.
  $a = 2$ Siempre se generan dos subproblemas.
  $k = 1$ Coste lineal de la parte no recursiva (la partición).

  dando un coste total $O(n \log n)$.

- La comprobación en un vector v estrictamente ordenado de si **existe un índice i tal que** $v[i] = i$ (ver la sección de problemas del Cap. 4) sigue un esquema similar al de la búsqueda binaria:

  $b = 2$ Tamaño mitad del subvector a investigar en cada llamada recursiva.

  $a = 1$ Un subproblema a lo sumo.

  $k = 0$ Coste constante de la parte no recursiva.

  dando un coste total $O(\log n)$.

- 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 $O(n^2)$.
- 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 $O(n \log n)$.
- 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.

Organización de un campeonato

- Se tienen n participantes para un torneo de ajedrez y hay que organizar un calendario para que todos jueguen contra todos de forma que:
  1. Cada participante juegue exactamente una partida con cada uno de los $n-1$ restantes.
  2. Cada participante juegue a lo sumo una partida diaria.
  3. El torneo se complete en el menor número posible de días.

- Es fácil ver que el número de parejas distintas posibles es $\frac{1}{2}n(n-1)$.
- Si n es par, cada día pueden jugar una partida los n participantes formando con ellos $\frac{n}{2}$ parejas. Por tanto, se necesita un mínimo de $n-1$ días para que jueguen todas las parejas.
- Si n es impar, se puede hacer que un participante descanse cada día y el resto formen $\frac{n-1}{2}$ parejas. En ese caso necesitaríamos un mínimo de n días.

- Por tanto, reformulamos el problema de la siguiente forma:
  1. Si n es par, cada participante juega una partida con cada uno de los $n-1$ restantes, a lo largo de $n-1$ días, sin descansar ninguno.
  2. Si n es impar, cada participante juega una partida con cada uno de los $n-1$ restantes, a lo largo de n días, uno de los cuales descansa.
- Se aprecia que la dificultad está en planificar las parejas de cada día, de tal modo que al final todos jueguen contra todos sin repetir ninguna partida, ni descansar innecesariamente.

- Podemos ensayar una solución DV según las siguientes ideas:
  - Si n es suficientemente grande, dividimos a los participantes en dos grupos disjuntos A y B, cada uno con la mitad de ellos.
  - Se resuelven recursivamente dos torneos más pequeños: el del conjunto A jugando sólo entre ellos, y el del conjunto B también jugando sólo entre ellos.
  - Después se planifican partidas en las que un participante pertenece a A y el otro a B.

- Esta última parte parece sencilla, puesto que basta con formar dos listas enfrentadas con los elementos de A y B, emparejar primero con primero, segundo con segundo, etc., rotar circularmente una posición una de las listas y repetir el proceso.
- Contando la configuración inicial, y sabiendo que la longitud de cada lista es $\frac{n}{2}$, el número configuraciones posibles es $\frac{n}{2}$ y en cada una se forman $\frac{n}{2}$ parejas distintas.
- Hay que esperar entonces que el coste en tiempo de esta fase esté en $\Theta(n^2)$.
- Los casos base, $n = 2$ o $n = 1$, se resuelven trivialmente en tiempo constante.

- Esta solución nos da pues los parámetros de coste $a = 2,\ b = 2$ y $k = 2$, que conducen a un coste esperado de $\Theta(n^2)$.
- No puede ser menor puesto que la propia planificación a devolver (n o $n-1$ días y $\frac{n}{2}$ parejas por día) consta ya de un número cuadrático de parejas.
- Pasamos entonces a precisar los detalles.

- Podemos conseguir sin pérdida de generalidad que n sea siempre par.
- Si no lo fuera, añadiríamos al conjunto un jugador $n+1$ ficticio d (de "descanso") y resolveríamos el problema par utilizando n días.
- Cada día i, d "jugaría" con un jugador distinto x_i, simbolizando con ello que en realidad x_i descansa ese día.
- Al ser par n, los dos subconjuntos A y B tienen exactamente el mismo tamaño $\frac{n}{2}$.
- Hacemos el siguiente análisis por casos:

$n > 2 \wedge \frac{n}{2}$ par Por hipótesis de inducción, las dos llamadas recursivas producen dos planificaciones, una para A y otra para B, en $\frac{n}{2} - 1$ días, y ningún participante descansa en ninguna de ellas. A continuación, rellenamos los $\frac{n}{2}$ días siguientes enfrentando a los miembros de A con los de B según se ha explicado más arriba. En total hemos necesitado:

$$\frac{n}{2} - 1 + \frac{n}{2} = n - 1 \text{ días}$$

$n > 2 \wedge \frac{n}{2}$ impar Por hipótesis de inducción, las dos llamadas recursivas producen dos planificaciones, una para A y otra para B, en $\frac{n}{2}$ días, y cada participante descansa un día en ambas planificaciones. Sean $d_1, \ldots, d_{\frac{n}{2}}$ los jugadores de A que respectivamente descansan los días $1, \ldots, \frac{n}{2}$, y sean $d'_1, \ldots, d'_{\frac{n}{2}}$ los jugadores de B que descansan esos mismos días.
Formamos con ellos dos listas enfrentadas y generamos la lista de parejas:

$$(d_1, d'_1), (d_2, d'_2) \ldots, (d_{\frac{n}{2}}, d'_{\frac{n}{2}})$$

Esta lista representa que en los primeros $\frac{n}{2}$ días, enfrentamos al jugador que descansaría en A con el que descansaría en B, con lo que en realidad ningún jugador descansa en los primeros $\frac{n}{2}$ días. A continuación rotamos $\frac{n}{2} - 1$ veces una de las listas y generamos las parejas correspondientes después de cada rotación. En total hemos necesitado

$$\frac{n}{2} + \frac{n}{2} - 1 = n - 1 \text{ días}$$

para formar la planificación de los n jugadores. Nótese que ninguno descansa en los $n - 1$ días.

$n = 2$ Se devuelve la planificación trivial de 1 día, con esos dos jugadores compitiendo entre sí.

$n = 1$ Se devuelve la planificación trivial de 1 día en el que el jugador en cuestión descansa.

Implementación

- Primero decidimos los TADs a utilizar para comunicar los jugadores al algoritmo y para devolver el resultado.
- Dadas las operaciones que han ido apareciendo (añadir jugador ficticio, formar parejas, rotar circularmente, etc.) nos decantamos por listas genéricas:

- Los jugadores serán una **lista de enteros**. Los enteros no han de ser necesariamente consecutivos, sino tan solo distintos, lo que da la posibilidad de utilizar un código de equipo, el DNI del participante, o cualquier otro identificador único.
- Los jugadores ficticios que simbolizan *descanso* serán enteros negativos. Veremos que será necesario distinguir los descansos de distintos niveles del algoritmo, por lo que necesitaremos un argumento extra $nivel \geq 1$ que indicará el nivel de recursión. Dicho número negado representará el jugador-descanso de ese nivel.
- El calendario del torneo se devolverá como una **lista de listas de parejas de enteros**. Cada lista de parejas representará las partidas de un día.
- Suponemos un TAD genérico `Pareja<T1,T2>`, con las operaciones `prim` y `segun` para acceder a sus componentes.

- Especificamos un conjunto de funciones sencillas sobre listas, cuya implementación no se da, pero que serán utilizadas en el algoritmo:

```
template <class T, class T1, class T2>

// Dada una lista de longitud m, devuelve una pareja de la listas,
// la primera con los n primeros elementos y la segunda con el resto.
// Se admite que la lista original quede destruida o modificada.
// Si n >= m, la primera lista es una copia del argumento y la
// segunda es vacía.

Pareja <Lista<T>,Lista<T>> divideLista(Lista<T> &lis, int n)

// Dadas dos listas de longitudes posiblemente distintas, devuelve
// una lista de parejas, emparejando el primer elemento con el primer
// elemento, el segundo con el segundo, etc., hasta que se agote una
// de las dos listas.

Lista <Pareja<T1,T2>> formaParejas(Lista<T1>, Lista<T2>)
```

```
// Dadas dos listas de listas, recorre ambas en paralelo y forma una lista
// de listas cuyo primera lista es la concatenación de la primera lista de
// cada argumento, la segunda la concatenación de las dos segundas, y
// así hasta que se agote la lista-argumento más corta.
// Se admite que los argumentos queden modificados o destruidos.

Lista<Lista<T>> uneListas(Lista<Lista<T>> &l1, Lista<Lista<T>> &l2)

// Dada un lista y una lista de listas, recorre ambas en paralelo y
// modifica la lista de listas añadiendo el primer elemento del primer
// argumento a la primera lista del segundo, el segundo a la segunda
// lista, y así hasta que se agote la lista-argumento más corta.

void consListas(Lista<T> l1, Lista<Lista<T>> &l2)

// Si la lista argumento no es vacía, se la modifica de tal forma que el
// primer elemento pase a ser el útimo, el segundo el primero, etc.

void rotar(Lista<T> &lis)

// Dada una lista de listas de parejas del mismo tipo y un valor del tipo,
// busca el valor en cada miembro de cada pareja de cada lista del conjunto
// y devuelve la lista de los compañeros de ese valor.
// Adicionalmente, elimina las parejas en las que aparezca el valor, es
// decir el argumento 'lis' puede quedar modificado.

Lista<Pareja<T>> filtrar(Lista<Lista<Pareja<T,T>>> &lis,const T &elem)
```

- En el siguiente listado mostramos el algoritmo completo.
- La función `torneo` recibe la lista de jugadores, su longitud n, y el nivel de recursión (inicialmente 1).
- Devuelve la lista de listas `calend` con las parejas que compiten cada día.
- Como hay que añadir elementos espúreos a la lista de jugadores, y posiblemente ésta se destruya al dividirla en dos, trabajamos sobre una copia local.

- El coste de la parte no recursiva viene dominado por los dos bucles **for** del algoritmo, que realizan m ó $m-1$ iteraciones y llaman a la función `formaParejas` de coste presumiblemente $O(m)$.
- Siendo m aproximadamente $\frac{n}{2}$, esta parte del algoritmo es del orden de $\Theta(n^2)$, tal como habíamos adelantado.
- El coste total de `torneo` resulta ser por tanto $\Theta(n^2)$.

```
typedef Lista<Pareja<int, int>> TJornada;

Lista <TJornada> torneo(Lista<int> jugadores, int nivel){
   Lista <TJornada> calend;
   Lista<int> jug = jugadores.copiaLista();
   int n = jugadores.numElems();

   if (!par(n)) { // añadimos un jugador-descanso de este nivel
      jug.ponDr(-nivel);
     }
   // Ahora n es siempre par
   if (n==2) { // Formar un calendario de un dia con los dos jugadores
     TJornada jornada;
     jornada.ponDr(Pareja<int,int>(jug.primero(),jug.ultimo()));
     calend.ponDr(jornada);
    }
```

```
else { // Dividimos a los jugadores en dos grupos
  int m = n/2;
  Pareja <Lista<int>,Lista<int>> dosListas = divideLista(jug,m);
  Lista<int> jugA = dosListas.prim();
  Lista<int> jugB = dosListas.segun();
  // Resolvemos los dos subproblemas recursivamente
  Lista <Lista<int>> calendA = torneo(jugA,m,nivel+1);
  Lista <Lista<int>> calendB = torneo(jugB,m,nivel+1);
  if (par(m)) {
      // Los calendarios son buenos para los m-1 primeros días
      // Los unimos día a día
      calend = uneListas(calendA,calendB);
      // Completamos m días más haciendo competir a A con B
      for (int i=1; i <= m; i++) {
          TJornada jornada = formaParejas(jugA,jugB);
          calend.ponDr(jornada);
          rotar(jugA);
        }
    }
```

```
    else /* impar m */ {
        // Filtrar los jugadores que descansan en cada calendario
        Lista<int> descA = filtrar(calendA,-(nivel+1));
        Lista<int> descB = filtrar(calendB,-(nivel+1));
        // Formamos parejas con los que descansan y las añadimos al
        // calendario  de A. Unimos los dos calendarios
        Lista<Pareja<int,int>> parejas = formaParejas(descA,descB);
        consListas (parejas,calendA);
        calend = uneListas(calendA,calendB);
        // Completamos m-1 días más haciendo competir a A con B
        for (int i=1; i < m; i++) {
            rotar(descA);
            TJornada jornada = formaParejas(descA,descB);
            calend.ponDr(jornada);
           }
       }
  }
   return calend;
}
```

El algoritmo de Katsuba y Ofman

- Algunas aplicaciones (p.e. criptografía) necesitan manipular enteros de **varios cientos de cifras** decimales y realizar con ellos las operaciones de suma, resta, producto, etc., del modo más eficiente posible.
- En esos casos no es correcto considerar que el coste de esas operaciones es constante, como hacemos habitualmente cuando están implementadas directamente por el hardware.
- Eligiendo como tamaño del problema $n =$ *número de cifras en cualquier base del número o números a operar*, nos planteamos cuál es el coste de esas operaciones con los algoritmos manuales que aprendemos en la escuela.

- Nótese que el número de cifras n_a en una base a y el número n_b en otra base b, de un entero dado v, están relacionadas por una constante según las fórmulas:

$$n_a \approx \log_a v = \log_a b \times \log_b v \approx \log_a b \times n_b$$

- El coste de la suma y la resta de dos números de tamaño n, siguiendo el algoritmo clásico de la escuela, está claramente en $\Theta(n)$. El producto está en cambio en $\Theta(n^2)$, ya que hay que multiplicar cada dígito del multiplicando por cada dígito del multiplicador y realizar después n sumas de tamaño n.

- Sea $y > 1$ la base en la que están expresados dos números A y B de n cifras.
- Supongamos para simplificar que n es potencia exacta de 2 (si no lo fuera, siempre podríamos completar las cifras de los números con ceros a la izquierda, hasta alcanzar una potencia de 2).
- El principio DV sugiere entonces el siguiente enfoque:

$$A = \overbrace{\boxed{a_1 \mid a_0}}^{n} \qquad A = a_1 \times y^{\frac{n}{2}} + a_0$$

$$B = \overbrace{\boxed{b_1 \mid b_0}}^{n} \qquad B = b_1 \times y^{\frac{n}{2}} + b_0$$

$$AB = a_1 b_1 \times y^n + (a_1 b_0 + b_1 a_0) \times y^{\frac{n}{2}} + a_0 b_0$$

- Esto da lugar a 4 productos de enteros de $\frac{n}{2}$ cifras, y a una serie de sumas y productos por potencias de la base (que consisten en añadir ceros por la derecha) que son de coste $\Theta(n)$.

- Utilizando una vez más nuestra recurrencia de referencia para anticipar el coste del algoritmo, obtenemos los siguientes parámetros de coste: $a = 4,\ b = 2,\ k = 1$.
- Al ser $a > b^k$ el coste será $\Theta(n^{\log_b a}) = \Theta(n^2)$.
- En este caso, no parece haber pues ninguna ventaja por utilizar el esquema DV.

- En 1962, los investigadores soviéticos Anatolii Karatsuba (que tenía entonces 23 años) y Yuri Ofman idearon un algoritmo en el que uno de los 4 productos podía ser evitado.
- La idea consiste en calcular $a_1 b_0 + b_1 a_0$ mediante **un sólo producto** en lugar de dos:

$$a_1 b_0 + b_1 a_0 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0$$

- El producto a calcular es $(a_1 + a_0)(b_1 + b_0)$, puesto que los otros dos productos $a_1 b_1$ y $a_0 b_0$ son necesarios en cualquier caso y ya han sido calculados.
- El precio es realizar algunas sumas y restas adicionales, cuyo coste sabemos en $\Theta(n)$.

- Los parámetros de la recurrencia son ahora $a = 3,\ b = 2,\ k = 1$. Ello permite anticipar un coste $\Theta(n^{\log_2 3}) = \Theta(n^{1,585})$, que asintóticamente es bastante mejor que $\Theta(n^2)$.
- Dado que el algoritmo resultante es más complicado que el clásico, es de esperar que las constantes multiplicativas sean también más altas y que el algoritmo clásico sea más eficiente para valores bajos de n.
- Dependiendo del computador empleado, se ha observado experimentalmente que el de Karatsuba y Ofman empieza a ser mejor a partir de números de 300 a 600 bits (de 90 a 180 cifras decimales).

Implementación

- Elegimos de nuevo el TAD `Lista` para representar números enteros de longitud arbitraria.
- Supondremos un computador de 32 bits y elegimos la base $y = 2^{15} = 32{,}768$. De este modo, el producto de dos "dígitos" en base y todavía cabe en una palabra del computador y podremos realizar el caso base $n = 1$ por hardware.
- También supondremos positivos los dos números, ya que, si no lo fueran, el signo del producto siempre podría calcularse fuera del algoritmo.

- Para facilitar las operaciones de suma y resta, decidimos que el dígito menos significativo (las unidades) corresponda al primero de la lista.
- La multiplicación por potencias de la base corresponderá entonces a añadir ceros por la izquierda a la lista.
- Igual que hicimos en la sección precedente, especificaremos unas funciones auxiliares sobre listas, que supondremos implementadas. Utilizaremos también la función `divideLista` especificada en dicha sección.

```
// Dadas dos listas que representan grandes enteros positivos, devuelve
// otra lista con su suma. El algoritmo suma digito a digito los dos
// argumentos y propaga el posible acarreo cuando la suma de dos dígitos
// es mayor o igual que la base 32.768.

Lista<int> sumaListas(Lista<int> l1, Lista<int> l2)

// Dadas dos listas que representan grandes enteros positivos en base
// 32.768, devuelve su resta. El algoritmo resta digito a digito los dos
// argumentos y propaga el posible acarreo. Supone que el primer número
// es mayor o igual que el segundo. Si no fuera así, lanza una excepción.

Lista<int> restaListas(Lista<int> l1, Lista<int> l2)

// Dada una lista que representa un gran entero positivo, y un natural n,
// modifica la lista añadiendo n ceros a la izquierda.

void completaCeros(Lista<int> &l, int n)
```

- El algoritmo completo se muestra a continuación.
- Nótese que no exigimos que la longitud sea potencia exacta de 2, ni que los dos números tengan la misma longitud, sino tan solo que el primero tenga más cifras o las mismas que el segundo.
- Hay que tener cierto cuidado con el número de ceros que añadimos ($2m$ en lugar de n), y estar preparados para que los fragmentos b_1 y b_0 puedan ser listas vacías.
- Una lista vacía ha de interpretarse como un cero a efectos de suma o multiplicación por otros números.

- Se aprecian perfectamente las tres fases del esquema DV: descomposición del problema en tres subproblemas de tamaño mitad, resolución recursiva de los mismos, y composición de los resultados parciales para conseguir el resultado final.
- La parte no recursiva consiste en llamadas a las funciones `divideLista`, `sumaListas`, `restaListas`, y `completaCeros`, todas ellas de coste lineal.
- Ello confirma que el coste asintótico del algoritmo está en $\Theta(n^{\log_2 3})$.

- En la práctica, se pueden mejorar las constantes multiplicativas del algoritmo si no se subdividen los números hasta el caso base $n = 1$.
- Se ha de elegir un **umbral** n_0 a partir del cual se utiliza el algoritmo clásico $\Theta(n^2)$, que sabemos es más rápido para valores pequeños de n.
- Cómo se determina ese umbral será el objeto de la Sección 6.

```
Lista<int> Karatsuba (Lista<int> A, Lista<int> B){
   Lista<int> res;
   int n = A.numElems();

   assert(A.numElems() >= B.numElems())

   if (! B.esVacia()) { // Si B es vacia devolvemos un cero

    if (n==1) { // Caso base. Generamos un número de dos dígitos
       int C = A.primero() * B.primero();
       res.ponDr(C % 32768);
       res.ponDr(C / 32768);
    }
```

```
    else /* n > 1 */ { // Dividimos el problema en tres
        int m = n/2;
        Pareja <Lista<int>,Lista<int>> a = divideLista(A,m);
        Pareja <Lista<int>,Lista<int>> b = divideLista(B,m);
        Lista<int> a0    = a.prim();
        Lista<int> a1    = a.segun();
        Lista<int> b0    = b.prim();
        Lista<int> b1    = b.segun();
        Lista<int> a1_a0 = sumaListas(a1,a0); // a1+a0
        Lista<int> b1_b0 = sumaListas(b1,b0); // b1+b0

        // Resolvemos recursivamente los tres subproblemas
        Lista<int> a1b1 = Karatsuba(a1,b1,m);
        Lista<int> a0b0 = Karatsuba(a0,b0,m);
        Lista<int> otro = Karatsuba(a1_a0,b1_b0,m); // (a1+a0)(b1+b0)

        // Combinamos los resultados
        Lista<int> aux   = restaListas(otro,a1b1);
        Lista<int> resta = restaListas(aux,a0b0); // (a1b0+b1a0)
        completaCeros(a1b1,2*m);                  // a1b1*y^2m
        completaCeros(resta,m);          // (a1b0+b1a0)*y^m
        aux = sumaListas(a1b1,resta); // a1b1*y^2m+(a1b0+b1a0)*y^m
        res = sumaListas(aux,a0b0);   // a1b1*y^2m+(a1b0+b1a0)*y^m+a0b0
    }
   }
   return res;
}
```

El algoritmo generalizado de Toom-Cook

- Cabe preguntarse si se puede mejorar el coste asintótico del algoritmo de Karatsuba y Ofman, si en lugar de dividir los números por la mitad, los dividiéramos en tres o en más partes.
- Frecuentemente, este tipo de "mejoras" tan solo afectan a la constante multiplicativa y hacen más complicado el algoritmo (confírmese esta afirmación programando una búsqueda "ternaria" en un vector ordenado, en la que el vector se divide en tres partes).

- Si dividiéramos en tres partes los números (suponiendo n potencia de 3), obtendríamos:

$$\begin{aligned} A &= a_2 \times y^{\frac{2n}{3}} + a_1 \times y^{\frac{n}{3}} + a_0 \\ B &= b_2 \times y^{\frac{2n}{3}} + b_1 \times y^{\frac{n}{3}} + b_0 \\ AB &= (a_2 \times y^{\frac{2n}{3}} + a_1 \times y^{\frac{n}{3}} + a_0)(b_2 \times y^{\frac{2n}{3}} + b_1 \times y^{\frac{n}{3}} + b_0) \end{aligned}$$

  lo que en principio da lugar a 9 productos de tamaño $\frac{n}{3}$.
- Dando los parámetros $a = 9,\ b = 3,\ k = 1$ a nuestra recurrencia de referencia, anticipamos un coste en $\Theta(n^{\log_3 9}) = \Theta(n^2)$.

- Los investigadores Andrei Toom y Stephen Cook idearon en 1963 una forma de hacer este cálculo utilizando **tan solo 5 productos** de tamaño $\frac{n}{3}$, lo que anticipa un coste en $\Theta(n^{\log_3 5}) = \Theta(n^{1,465})$, que es mejor que el de Karatsuba y Ofman.
- Más aún, generalizaron la idea dividiendo el vector en un número k arbitrario pero fijo de segmentos, y realizando tan sólo $2k-1$ productos de tamaño $\frac{n}{k}$, lo que da un coste general $\Theta(n^{\log_k(2k-1)}) = \Theta(n^{1+\log_k(2-\frac{1}{k})})$, que tiende a $\Theta(n)$ cuando k tiende a infinito.

- La idea subyacente es relativamente sencilla: se realiza el cambio de variable $x = y^{\frac{n}{k}}$ y se expresan los números como polinomios en x de grado $k-1$:

$$\begin{aligned} A &= a_{k-1}x^{k-1} + \cdots + a_1 x + a_0 \\ B &= b_{k-1}x^{k-1} + \cdots + b_1 x + b_0 \\ AB &= (a_{k-1}x^{k-1} + \cdots + a_1 x + a_0)(b_{k-1}x^{k-1} + \cdots + b_1 x + b_0) \\ &= p_{2k-2}x^{2k-2} + \cdots + p_1 x + p_0 \end{aligned}$$

  donde los $2k-1$ coeficientes p_i son las incógnitas a determinar.

- Por ejemplo, para $k = 3$ obtendríamos:

$$(a_2x^2+a_1x+a_0)(b_2x^2+b_1x+b_0) = p_4x^4+p_3x^3+p_2x^2+p_1x+p_0$$

- Dos polinomios de grado n son iguales si y solo si se evalúan al mismo valor en $n+1$ puntos.
- Entonces, damos $2k-1$ valores distintos a x y obtenemos un sistema de $2k-1$ ecuaciones lineales que son suficientes para calcular los p_i.
- Por ejemplo, para $x=0$ se ha de cumpir $a_0 b_0 = p_0$.
- Igualmente, para $x=\infty$ ha de cumplirse $a_2 b_2 = p_4$.

- Como $k = 3$ en este caso, elegimos otros tres puntos distintos, por ejemplo $x = 1,\ x = -1,\ x = 2$, y obtenemos en conjunto 5 ecuaciones lineales en los p_i:

$$
\begin{aligned}
a_0 b_0 &= p_0 \\
a_2 b_2 &= p_4 \\
(a_2 + a_1 + a_0)(b_2 + b_1 + b_0) &= p_4 + p_3 + p_2 + p_1 + p_0 \\
(a_2 - a_1 + a_0)(b_2 - b_1 + b_0) &= p_4 - p_3 + p_2 - p_1 + p_0 \\
(4a_2 + 2a_1 + a_0)(4b_2 + 2b_1 + b_0) &= 16p_4 + 8p_3 + 4p_2 + 2p_1 + p_0
\end{aligned}
$$

  donde los valores de la parte izquierda son conocidos.

- Nótese que para evaluar cada uno, se necesita calcular un producto de tamaño $\frac{n}{k}$.

- Una vez calculados los $2k-1$ productos, con un coste adicional en $\Theta(k^2)$ se resuelve el sistema de ecuaciones y se obtienen los coeficientes p_i como combinaciones lineales de dichos productos.
- Como k es una constante, y las sumas, productos por una constante, y restas necesarias son proporcionales a n, el coste total de la parte no recursiva resulta ser $\Theta(n)$.
- La constante multiplicativa del algoritmo de Toom-Cook es mucho más alta que la del de Karatsuba y Ofman, y en la práctica sólo le supera para $k=3$ y n igual a varios miles de cifras decimales.
- Valores superiores de k sólo tienen un interés teórico.

- El algoritmo de Schönhage-Strassen (Arnold Schönhage y Volker Strassen, 1971) fue el más rápido conocido hasta 2007.
- Su complejidad asintótica es $O(n \log n \log \log n)$ y utiliza como subalgoritmo la transformada rápida de Fourier.
- Su mayor eficiencia sólo se manifiesta para números de al menos 10.000 cifras decimales.
- El algoritmo de mejor complejidad asintótica conocido es el de Martin Fürer (2007), cuya cota superior es cercana a $O(n \log n)$, que se conjetura es la cota inferior al problema de la multiplicación de enteros.
- Su interés es exclusivamente teórico y no se usa en la práctica.

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.

- Dados dos puntos, $p_1 = (x_1, y_1)$ y $p_2 = (x_2, y_2)$, su distancia euclídea viene dada por $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.
- 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**.
- 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.
- Una posible estrategia es:

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 linea 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.

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

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 l. La operación *combinar* debe investigar los puntos de dicha banda vertical

- Antes de seguir con los detalles, debemos investigar el coste esperado de esta estrategia.
- 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.

- 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.
- 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.

- 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 l.
- El filtrado puede hacerse con coste lineal.
- Llamemos B_I y B_D a los puntos de dicha banda respectivamente a la izquierda y a la derecha de l.

- Para investigar si en la banda hay dos puntos a distancia menor que δ, aparentemente debemos calcular la distancia de cada punto de B_I a cada punto de B_D.
- 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.

- 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 $\Theta(n \log^2 n)$.
- Recordando la técnica de los **resultados acumuladores** explicada en el tema 4 de estos apuntes, 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.

- 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.
  2. Filtrar la lista resultante, conservando los puntos a una distancia de la línea divisoria l 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

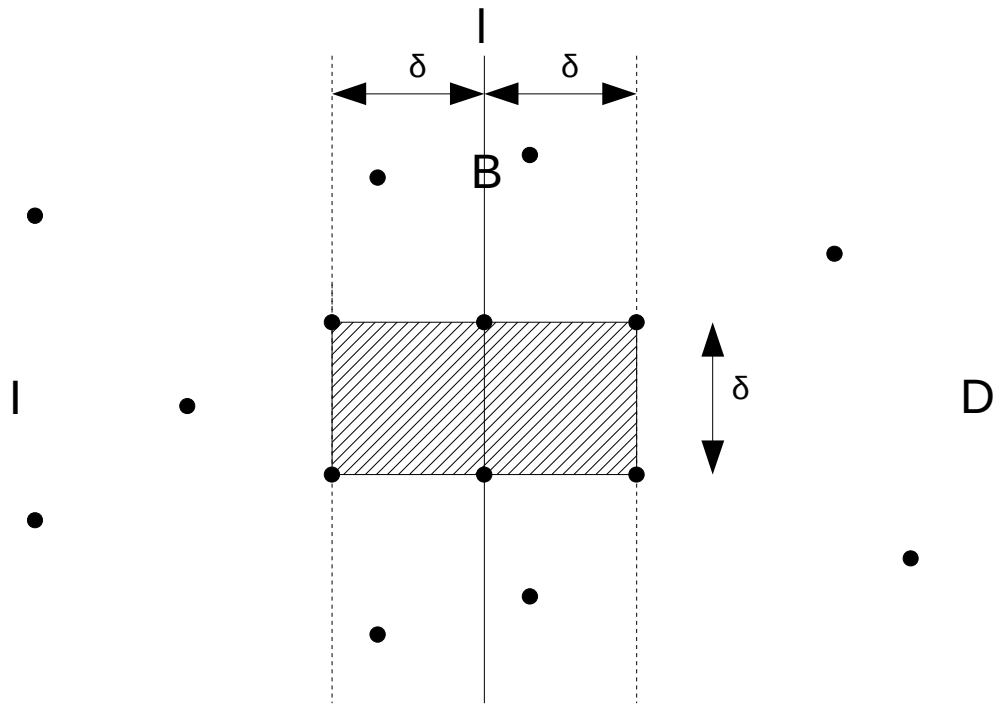

Figura 1: Razonamiento de corrección del problema del par más cercano

- Consideremos un rectángulo cualquiera de anchura 2δ y altura δ centrado en la línea divisoria l (ver Figura 1).
- Afirmamos que, contenidos en él, puede haber a lo sumo 8 puntos de la nube original.
- En la mitad izquierda puede haber a lo sumo 4, y en caso de haber 4, situados necesariamente en sus esquinas, y en la mitad derecha otros 4, también en sus esquinas.
- 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.

- 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. La entrada al algoritmo será una lista de puntos, y la solución una estructura con un par de puntos, una distancia y una lista de puntos.
- Necesitaremos algunas funciones auxiliares que especificaremos pero no implementaremos, por considerarlas suficientemente sencillas.

```
typedef Pareja<double,double> Punto;
typedef struct {
    Punto p1;
    Punto p2;
    double delta;
    Lista<Punto> lista;
  } Solucion;
```

```
// Dada una lista con dos o tres puntos, devuelve los dos puntos mas
// cercanos, su distancia y una lista con los puntos ordenados
// crecientemente por su coordenada y

Solucion solucionDirecta(Lista<Punto> puntos, int n)

// Dadas dos listas de puntos ordenadas por la coordenada y, devuelve
// otra lista con su mezcla ordenada tambien por la coordenada y

Lista<Punto> merge(Lista<Punto> l1, Lista<Punto> l2)

// Dada una lista l de puntos, una distancia d y una abcisa x,
// devuelve la lista de los puntos de l cuya abcisa diste a lo
// sumo d de x en valor absoluto

Lista<Punto> filtraBanda(Lista<Punto> l, double d, double x)

// Recorre una lista l de puntos, comparando cada uno con los 7
// siguientes si los hubiera. Devuelve el par de puntos con menor
// distancia, y dicha distancia. Si l tiene un punto o ninguno,
// devuelve una distancia +infinito.

void recorreBanda(Lista<Punto> l, Punto &p1, Punto &p2, double &d)

// Dadas tres soluciones, cada una consistente en un par de puntos y su
// distancia, devuelve el par mas cercano de los tres y su distancia.

Solucion eligeMinimo(const Solucion &s1, const Solucion &s2,
                     const Punto &p1, const Punto &p2, double d)
```

- A continuación mostramos el algoritmo principal DV.
- Todas las funciones auxiliares llamadas en el caso recursivo tienen coste lineal o constante, lo que confirma que el coste de la parte no recursiva está en $\Theta(n)$.
- Por tanto el coste total está en $\Theta(n \log n)$ como pretendíamos.

```
Solucion parMasCercano(Lista<Punto> puntos, int n) {
   Solucion sol;
   Punto p1; Punto p2; double dist;

   if (n <= 3) { // Casos base
      sol = solucionDirecta(puntos,n);
   else /* n >= 4 */ { // Dividimos la nube en dos
      int m1 = n / 2; int m2 = n - m1;
      Pareja<Lista<Punto>,Lista<Punto>> par = divideLista(puntos,m1);
      Lista<Punto> I = par.prim();
      Lista<Punto> D = par.segun();
           // Resolvemos recursivamente las dos nubes
      Solucion sol1 = parMasCercano(I,m1);
      Solucion sol2 = parMasCercano(D,m2);
          // Calculamos la coordenada x de la linea divisoria
      double xI = (I.ultimo().prim() + D.primero().prim()) / 2;
          // Ordenamos por la coordenada y la nube de puntos
      Lista<Punto> lista = merge(sol1.lista,sol2.lista);
      double delta = min(sol1.delta,sol2.delta);
          // Filtramos los puntos de la banda y la recorremos
      Lista<Punto> B = filtraBanda(lista,delta,xI);
      recorreBanda(B,p1,p2,dist);
          // Elegimos la mejor solucion de las tres
      sol = eligeMinimo(sol1,sol2,p1,p2,dist);
      sol.lista = lista;
     }
   return sol;
}
```

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.
- Se puede conseguir un algoritmo óptimo combinando ambos algoritmos de modo inteligente.
- El aspecto que tendría el algoritmo compuesto es:

```
Solucion divideYvenceras (Problema x, int n){
   if (n <= n_0)
      return algoritmoSencillo(x)
   else /* n > n_0 */ {
          descomponer x
          llamadas recursivas a divideYvenceras
          y = combinar resultados
          return y;
        }
}
```

- Se trata de convertir en casos base del algoritmo recursivo los problemas que son *suficientemente* pequeños.
- Nos planteamos cómo 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.

- A pesar de eso, se puede hacer un estudio teórico del problema para encontrar un umbral aproximado.
- Para fijar ideas, centrémosnos en el problema de encontrar el par más cercano y escribamos su recurrencia con constantes multiplicativas (suponemos n potencia de 2):

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

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

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

- 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:

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

- 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 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.