El esquema "Divide y vencerás"

Ricardo Peña y Clara Segura son los autores principales de este tema

Facultad de Informática - UCM

2 de diciembre de 2013

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.

- 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*:

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

- Solución :

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

- 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.
- 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.

Ejemplos de aplicación del esquema con éxito

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

Especificación:

$\{0 \leq num \leq long(v)\}$

```
proc quickSort ( TElem v[], int num )
```

$\{ord(v, num)\}$

$\{0 \leq a \leq long(v) \wedge -1 \leq b \leq long(v) - 1 \wedge a \leq b + 1\}$

```
proc quickSort( TElem v[], int a, int b)
```

$\{ord(v, a, b)\}$

```
void quickSort ( TElem v[], int num ) {
  quickSort(v, 0, num-1);
}
void quickSort( TElem v[], int a, int b) {...}
```

- Planteamiento recursivo:
  - Elegir un pivote: un elemento cualquiera del subvector v[a..b]. Normalmente se elige v[a].
  - Particionar el subvector v[a..b], 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 recursivamente los dos fragmentos que han quedado a la izquierda y a la derecha del pivote.

- Análisis de casos:
  - Caso directo: $a = b + 1$ El subvector está vacío y, por lo tanto, ordenado.
  - Caso recursivo: $a \leq b$
  - Cubren todos los posibles casos.
- Función de acotación:

$$t(v, a, b) = b - a + 1$$

  decrece en cada llamada recursiva.
- Funciones sucesor (p es la posición del elemento pivote al terminar la partición):

$$s_1(v, a, b) =< v, a, p - 1 >$$

$$s_2(v, a, b) =< v, p + 1, b >$$

  - Se alcanza el caso base porque $b > p - 1$ y $a < p + 1$, por lo que en algún momento los índices se cruzan.
  - Los argumentos de la función sucesor cumplen los requisitos de la precondición.

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

```
void quickSort( TElem v[], int a, int b ) {
// Pre: 0 ≤ a ≤ long(v) && −1 ≤ b < long(v) && a ≤

  int  p;

  if ( a <= b ) {
    particion(v, a, b, p);
    quickSort(v, a, p-1);
    quickSort(v, p+1, b);
  }

// Post: v está ordenado entre a y b
}
```

- Diseño de partición. (algoritmo iterativo)
- Especificación

```
{ P: 0 ≤ a ≤ b < long(v) }
  void particion(int v[], int a, int b, out int p)
{ Q: 0 ≤ a ≤ p ≤ b ≤ num − 1 ∧ (∀x : a ≤ x ≤ p − 1 : v[x] ≤ v[p])
       ∧(∀x : p + 1 ≤ x ≤ b : v[x] ≥ v[p]) }
```

- La idea es obtener un bucle que mantenga invariante la siguiente situación

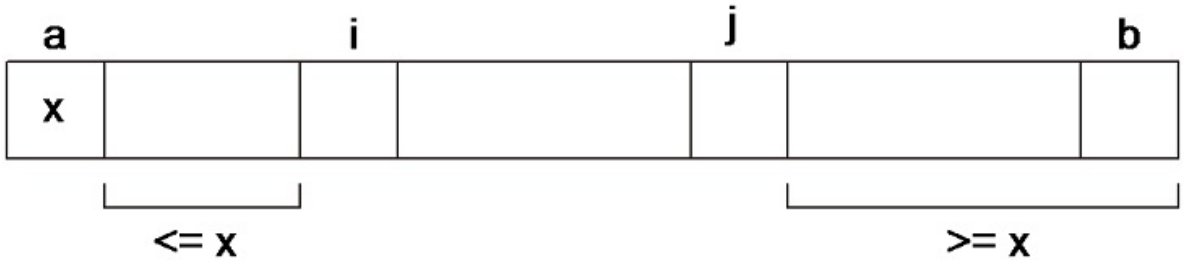

Figura 1 : Diseño de *particion*

de forma que i y j se vayan acercando hasta cruzarse, y finalmente intercambiemos $v[a]$ con $v[j]$

- Invariante: generalizar la postcondición con la introducción de dos variables nuevas, i, j, que indican el avance por los dos extremos del subvector

$$I \equiv \{a+1 \leq i \leq j+1 \leq b+1 \wedge \\ (\forall x : a+1 \leq x \leq i-1 : v[x] \leq v[a]) \wedge \\ (\forall x : j+1 \leq x \leq b : v[x] \geq v[a])\}$$

- Condición de parada

  $\neg B \equiv i = j+1$, $B \equiv i \leq j$

  A la salida del bucle, el vector estará particionado salvo por el pivote $v[a]$. Para terminar el proceso basta con intercambiar los elementos de las posiciones a y j, quedando la partición en la posición j.

- Expresión de acotación:
  $C : j - i + 1$
- Acción de inicialización
  $i = a + 1;$
  $j = b;$
- Acción de avance:
  El objetivo del bucle es conseguir que i y j se vayan acercando, y además se mantenga el invariante en cada iteración. Para ello, se hace un análisis de casos comparando las componentes $v[i]$ y $v[j]$ con $v[a]$
  - $v[i] \leq v[a] \rightarrow$ incrementamos i
  - $v[j] \geq v[a] \rightarrow$ decrementamos j
  - $v[i] > v[a] \wedge v[j] < v[a] \rightarrow$ intercambiamos $v[i]$ con $v[j]$, incrementamos i y decrementamos j

Algoritmo:

```
void particion ( TElem v[], int a, int b, int & p ) {
  int i, j;    TElem aux;

  i = a+1;  j = b;
  while ( i <= j ) {
    if ( (v[i] > v[a]) && (v[j] < v[a]) ) {
      aux = v[i]; v[i] = v[j]; v[j] = aux;
      i = i + 1; j = j - 1;
    }
    else {  if ( v[i] <= v[a] ) i = i + 1;
            if ( v[j] >= v[a] ) j = j - 1;
    }
  }
  p = j;
  aux = v[a]; v[a] = v[p]; v[p] = aux;
}
```

Mejoras al algoritmo de partición:

- Recibe como argumento el pivote.
- Devuelve dos posiciones p y q tales que
  - los elementos desde a hasta p-1 son $<$ pivote
  - los elementos desde p hasta q son $=$ pivote
  - los elementos desde q+1 hasta b son $>$ pivote

- Implementación: Se utilizan 3 índices: p y k se mueven hacia la derecha, q se mueve hacia la izquierda.
- Invariante: (el pivote es un parámetro de entrada piv)
  $I \equiv \{a <= p <= k <= q+1 <= b+1 <= long(v)$
  $(\forall x : a \leq x \leq p-1 : v[x] < piv) \wedge$
  $(\forall x : p \leq x \leq k-1 : v[x] = piv) \wedge$
  $(\forall x : q+1 \leq x \leq b : v[x] > piv)\}$
  La parte $v[k..q]$ está sin explorar hasta que $k = q+1$ en cuyo caso el bucle termina y tenemos lo que queremos.
- Avance y recuperación del invariante: se compara $v[k]$ con el pivote:
  - si es igual, está bien colocado y se incrementa k;
  - si es menor se intercambia con $v[p]$ para ponerlo junto a los menores y se incrementan los indices p y k;
  - si es mayor se intercambia con $v[q]$ para ponerlo junto a los mayores y se decrementa la q.

```
void particion2(TElem v[], int a, int b, TElem pivote,
                        int& p, int& q)
{//PRE: 0<=a<=b<long(v)
int k;
TElem aux;
p=a;k=a;q=b;
while (k<=q)
{
  if (v[k] == pivote) {k = k+1;}
  else if (v[k] < pivote)
         {aux = v[p]; v[p] = v[k]; v[k] = aux;
          p= p+1; k=k+1;}
  else {aux = v[q]; v[q] = v[k]; v[k] = aux;
          q=q-1;}
}
//POST: los elementos desde a hasta p-1 son < pivote
//      los elementos desde p hasta q son = pivote
//      los elementos desde q+1 hasta b son > pivote
}
```

- Ordenación por mezcla (*mergesort*)
- Especificación.

$\{0 \leq num \leq long(v)\}$
```
proc mergeSort ( TElem v[], int num )
```
$\{ord(v, num)\}$

$\{0 \leq a \leq long(v) \wedge -1 \leq b \leq long(v) - 1 \wedge a \leq b + 1\}$
```
proc mergeSort( TElem v[], int a, int b )
```
$\{ord(v, a, b)\}$

```
void mergeSort ( TElem v[], int num ) {
  mergeSort(v, 0, num-1);
}

void mergeSort( TElem v[], int a, int b) {...}
```

- Planteamiento recursivo. Para ordenar el subvector $v[a..b]$
  - Obtenemos el punto medio m entre a y b, y ordenamos recursivamente los subvectores $v[a..m]$ y $v[(m+1)..b]$.
  - Mezclamos ordenadamente los subvectores $v[a..m]$ y $v[(m+1)..b]$ ya ordenados.
- Análisis de casos.
  - Caso directo: $a \geq b$
    El subvector está vacío o tiene longitud 1 y, por lo tanto, está ordenado.
    $P_0 \wedge a \geq b \Rightarrow a = b \vee a = b + 1$
  - Caso recursivo: $a < b$
- Función de acotación. $t(v, a, b) = b - a + 1$ (decrece en cada llamada recursiva)
- Funciones sucesor
  $s_1(v, a, b) =< v, a, (a + b)/2 >$
  $s_2(v, a, b) =< v, (a + b)/2 + 1, b >$
  Cumplen la precondición del algoritmo.

Algoritmo:

```
void mergeSort( TElem v[], int a, int b ) {
// Pre: 0 ≤ a ≤ long(v) && −1 ≤ b < long(v) && a ≤ b+1

  int  m;

  if ( a < b ) {
    m = (a+b) / 2;
    mergeSort( v, a, m );
    mergeSort( v, m+1, b );
    mezcla( v, a, m, b );
  }

// Post: v está ordenado entre a y b
}
```

- Diseño de mezcla. (algoritmo iterativo)
  - Para conseguir una solución eficiente, O(n), utilizaremos un vector auxiliar donde iremos realizando la mezcla, para luego copiar el resultado al vector original.
  - La idea del algoritmo es colocarse al principio de cada subvector e ir tomando, de uno u otro, el menor elemento, y así ir avanzando.
  - Uno de los subvectores se acabará primero y habrá entonces que copiar el resto del otro subvector.

- En el array auxiliar tendremos los índices desplazados pues mientras el subvector a mezclar es $v[a..b]$, en el array auxiliar tendremos los elementos almacenados en $v[0..b-a]$, y habrá que ser cuidadoso con los índices que recorren ambos arrays.

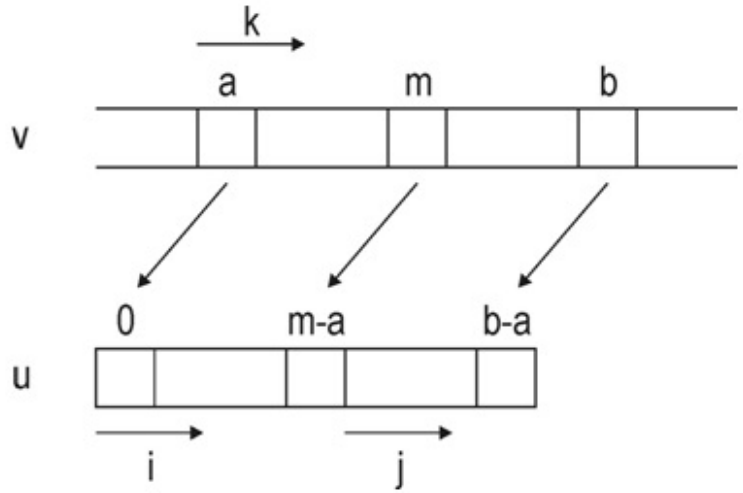

Figura 2 : Diseño de *mezcla*

- Con todo esto, el procedimiento de mezcla queda:

```
void mezcla( TElem v[], int a, int m, int b ) {
// Pre: a ≤ m < b y v ord. entre a y m y v ord. entre m+1
  TElem *u = new TElem[b-a+1];
  int i, j, k;
  for ( k = a; k <= b; k++ )  u[k-a] = v[k];
  i = 0; j = m-a+1; k = a;
  while ( (i <= m-a) && (j <= b-a) ) {
    if ( u[i] <= u[j] ){ v[k] = u[i];  i = i + 1; }
    else {  v[k] = u[j];  j = j + 1; }
    k = k + 1;
  }
  while ( i <= m-a ) {
    v[k] = u[i];  i = i+1;  k = k+1;
  }
  while ( j <= b-a ) {
    v[k] = u[j];  j = j+1; k = k+1;
  }
  delete[] u;
// Post: v está ordenado entre a y b
}
```

- 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-esima indica la posición en el vector de entrada del elemento que debe ocupar el i-esimo lugar en la ordenación.

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

```
void mergeSort( TElem v[], int a, int b , int ind[])
{
  int  m;

 if (a == b) ind[a] = a;
 else if ( a < b ) {
    m = (a+b) / 2;
    mergeSort( v, a, m, ind );
    mergeSort( v, m+1, b, ind );
    mezcla( v, a, m, b, ind );
  }
}
```

```
void mezcla( TElem v[], int a, int m, int b, int l[] ) {
  TElem *u = new TElem[b-a+1];
  int i, j, k;
  i = a; j = m+1; k = a;
  while ( (i <= m) && (j <= b) ) {
    if ( v[l[i]] <= v[l[j]] ){ u[k] = l[i];  i = i + 1;
    else {  u[k] = l[j];  j = j + 1; }
    k = k + 1;
  }
  while ( i <= m ) {
    u[k] = l[i];  i = i+1;  k = k+1;
  }
  while ( j <= b ) {
    u[k] = l[j];  j = j+1; k = k+1;
  }
  for ( k = a; k <= b; k++ )  l[k] = u[k];
  delete[] u;
}
```

- 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.

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 a y b que nos indican la parte del vector que nos interesa en cada momento. La llamada inicial que deseamos es $seleccion(v, 0, num - 1, k)$.
- 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.

```
TElem seleccion1(TElem v[], int a, int b, int k)
//Pre: 0<=a<=b<=long(v)-1 && a<=k<=b
{int p;
 if (a==b) {return v[a];}
 else
  { particion(v,a,b,p);
   if (k==p) {return v[p];}
   else if (k<p) { return seleccion1(v,a,p-1,k);}
   else {return seleccion1(v,p+1,b,k);}
  }
};
```

Caso peor: el pivote queda siempre en un extremo del subvector correspondiente. El coste está en $O(n^2)$ siendo $n = b - a + 1$ 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 de la mediana, conocida como *mediana de las 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*, son:

1. calcular la mediana de cada grupo de 5 elementos. En total n *div* 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 *div* 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 $b - a + 1 <= 12$, es más 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).

```
TElem seleccion2(TElem v[], int a, int b, int k)
{//0<=a<=b<=long(v)-1 && a<=k<=b
{int l,p,q,s,pm,t;
 TElem aux,mm;
 t = b-a+1;
 if (t <=12) { ordenarInsercion(v,a,b);return v[k]; }
 else { s = t / 5;
      for (l=1; l<=s;l++) {
          ordenarInsercion(v,a+5*(l-1),a+5*l-1);
          pm = a+5*(l-1)+(5 / 2);
          aux = v[a+l-1]; v[a+l-1]=v[pm]; v[pm] = aux;
    };
    mm=seleccion2(v,a,a+s-1,a+(s-1)/2);
    particion2(v,a,b,mm,p,q);
    if ((k>=p)&&(k<=q)) {return mm;}
    else if (k<p) { return seleccion2(v,a,p-1,k);}
    else {return seleccion2(v,q+1,b,k);}
 } };
//POST: v[k] es mayor o igual que v[0..k-1] y menor o
// igual que v[k+1..num-1]
```

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.

  En este tema veremos una solución para el caso, más sencillo, en que n es potencia de 2.

- El número de parejas distintas posibles es $\frac{1}{2}n(n-1)$. Como 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.
- Representamos la solución con una matriz $(n \times n)$. a_{ij}, representa el día que se enfrentarán i y j, con $j < i$.
- Se ha de planificar las parejas de cada día, de tal modo que al final todos jueguen contra todos sin repetir ninguna partida, ni descansar innecesariamente.

- Idea DV:
  - 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.
  - Se planifican partidas en las que un participante pertenece a A y el otro a B. Para ello se rellena la matriz fila por fila, rotando, en cada nueva fila, el orden de las fechas disponibles. El coste está en en $\Theta(n^2)$.
- Los casos base, $n = 2$ o $n = 1$, se resuelven trivialmente en tiempo constante.
- El coste esperado es de $\Theta(n^2)$. No puede ser menor puesto que la propia planificación consiste en rellenar $\Theta(n^2)$ celdas.

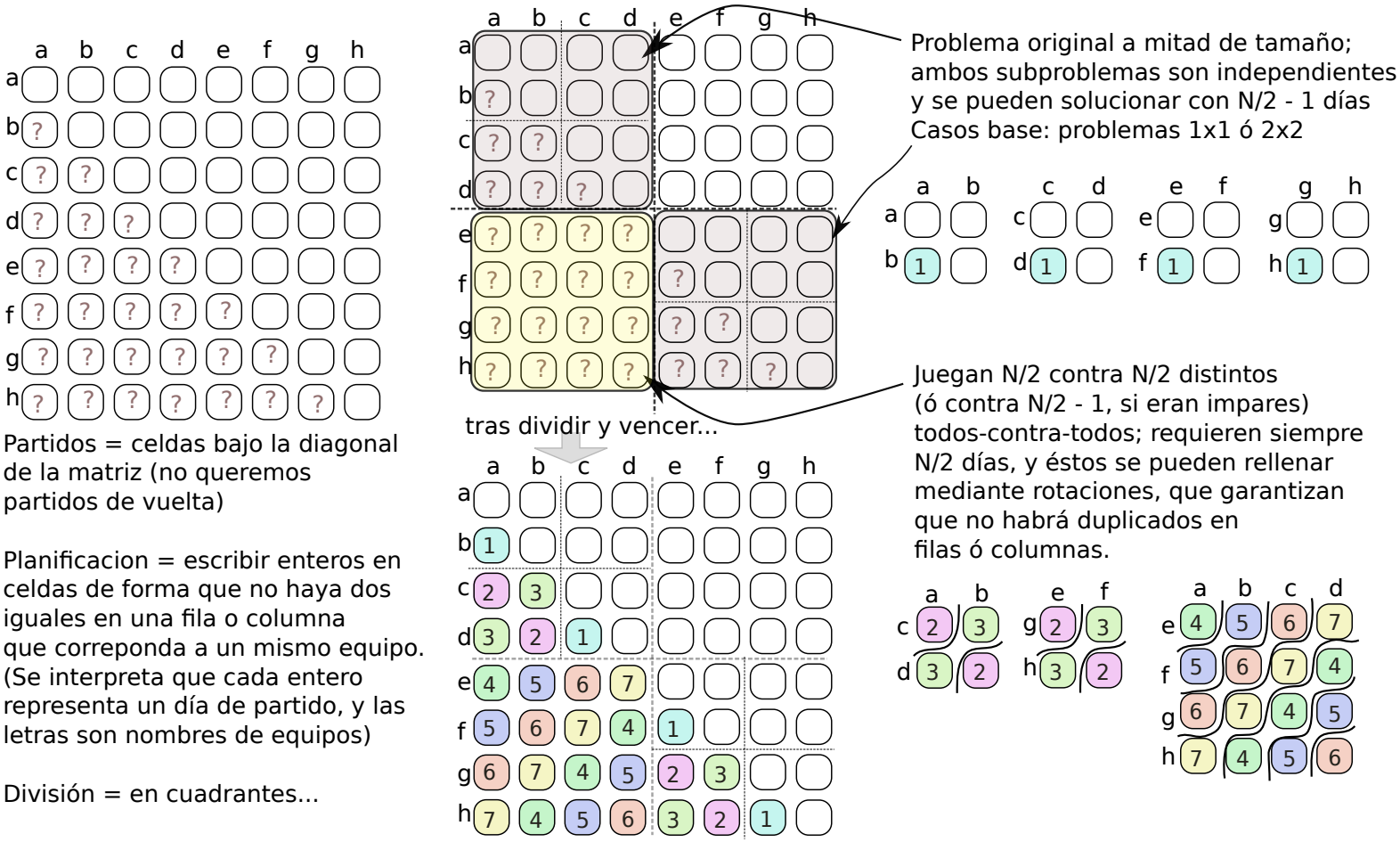

Figura 3 : Solución gráfica del problema del torneo

- Usaremos una matriz cuadrada para almacenar la solución. La primera fecha disponible será el día 1.
- La función recursiva, recibe dos parámetros, c y f que delimitan el trozo de la matriz que estamos rellenando. Se asume que $dim = f - c + 1$ es potencia de 2.
- En el caso base en que solo haya un equipo ($dim = 1$) no se hace nada. Si hay dos equipos ($dim = 2$), juegan el dia 1.
- El cuadrante inferior izquierdo representa los partidos entre los equipos de los dos grupos. El primer dia disponible es $mitad = dim/2$ y hacen falta $mitad$ dias para que todos jueguen contra todos. Las rotaciones se consiguen con la fórmula $mitad + (i + j + 1)\,\%mitad$, ya que $i \neq j$.

```
void rellena(int a[MAX][MAX], int c, int f)
{//PRE: ∃k : f − c + 1 = 2^k ∧ 0 <= c <= f < MAX∧
//  ∀i, j : c <= i, j <= f : a[i][j] = 0
 int dim = f-c+1;
 int mitad = dim/2;
 //si dim = 1 nada, la diagonal principal no se rellena
  if (dim ==2) { a[c+1][c]=1;}
  else if (dim>2)
     {rellena(a,c,c+mitad-1);        //cuad superior izq.
      rellena(a,c+mitad,f);           //cuad inferior der
      for (int i=c+mitad; i<=f; i++){  //cuad inf izq
       for (int j=c; j<=c+mitad-1;j++)
             {a[i][j]=mitad+(i+j+1) %mitad;}
        }
      }
//POST:∀i, j : c ≤ j < i ≤ f : 1 ≤ a[i][j] ≤ f − c  ∧
//       ∀i : c < i ≤ f : (∀j, k : c ≤ j < k < i : a[i][j] ≠ a[i][k])∧
//       ∀j : c ≤ j < f : (∀i, k : j < i < k ≤ f : a[i][j] ≠ a[k][j])
}
```

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.

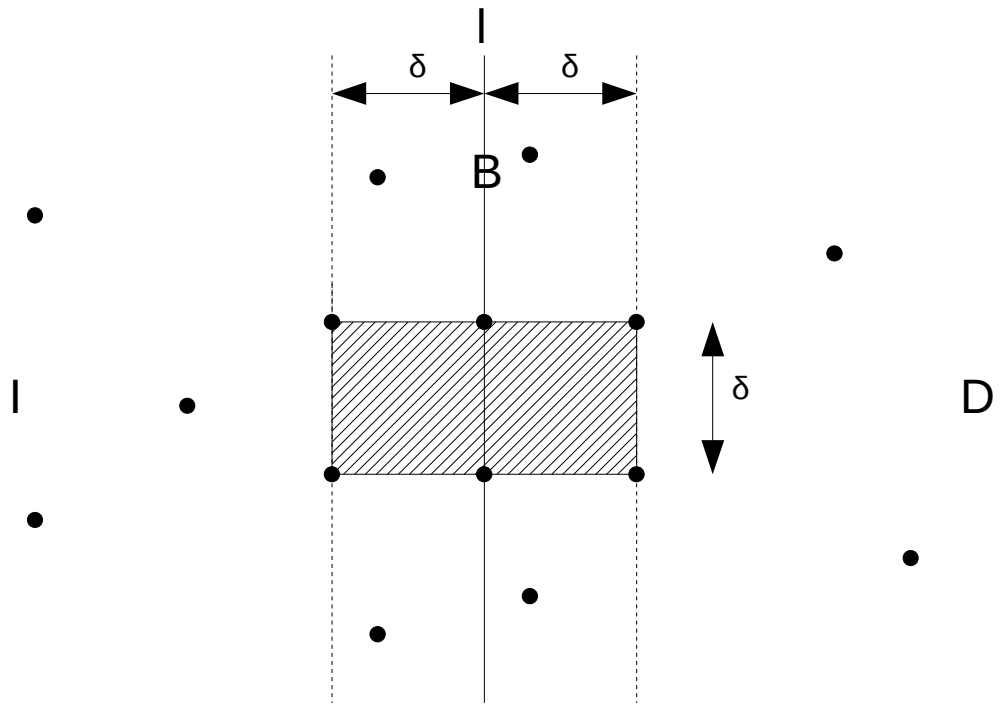

Figura 4 : Razonamiento de corrección del problema del par más cercano

- 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(Punto p[], int c, int f,
                   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 *ini* que representan 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(Punto p[], int c, int f,
                     int indY[], int& ini,
                     double& d, int& p1, int& p2)
{   double d1,d2,d3;
    if (f==c+1)
     { d = distancia(p[c],p[f]);
       if ((p[c].y) <= (p[f].y))
         {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)
    {
    ..........
    }
};
```

```
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
      {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(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) solucionDirecta(p,c,f,indY,ini,d,p1,p2);

 else { m = (c+f)/2;
   parMasCercano(p,c,m,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+1];
int k=iniA;
for (int l=0;l<=f-c+1;l++){aux[l]=-1;};
while (i!=-1)
{
   if (absolute(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 mas 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 más 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 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.