Capítulo 5

El esquema "Divide y vencerás" [1]

Divide et impera

Julio César (100 a.C.-44 a.C)

Resumen: En este tema se presenta el esquema algorítmico *Divide y vencerás*, que es un caso particular del diseño recursivo, y se ilustra con ejemplos significativos en los que la estrategia reporta beneficios claros. Se espera del alumno que incorpore este método a sus estrategias de resolución de problemas.

1. 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, el tamaño de cuyos datos es **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. Existirán uno o más **casos base** en los que el problema no se

[1] Ricardo Peña es el autor principal de este tema. Modificado por Clara Segura en el curso 2013-14.

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 que sea una *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.
- El (los) caso(s) 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>
Solución 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 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.

2. 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 solo 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.

3. Problema de selección

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

⋆ El problema de encontrar la mediana de un vector es un caso particular de este problema en el que se busca el elemento $\lceil n/2 \rceil$-ésimo del vector en el caso de estar ordenado (en los vectores de C++ corresponde a la posición $(n-1) \div 2$).

⋆ Una primera idea para resolver el problema consiste en ordenar el vector y tomar el elemento $v[k]$, lo cual tiene la complejidad del algoritmo de ordenación utilizado. Nos preguntamos si podemos hacerlo más eficientemente.

⋆ Otra posibilidad es utilizar el algoritmo *partición* que vimos en el Capítulo 4 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 $k < p$ entonces podemos pasar a buscar el k-ésimo elemento en las posiciones anteriores a p, ya que en ellas se encuentran los elementos menores o iguales que $v[p]$ y $v[p]$ es el p-ésimo elemento del vector.
- Si $k > p$ entonces podemos pasar a buscar el k-ésimo elemento en las posiciones posteriores a p, ya que en ellas se encuentran los elementos mayores o iguales que $v[p]$ y $v[p]$ es el p-ésimo elemento del vector.

- ⋆ Al igual que hemos hecho en anteriores ocasiones generalizamos el problema añadiendo dos parámetros adicionales a y b tales que $0 \leq a \leq b \leq long(v) - 1$, que nos indican la parte del vector que nos interesa en cada momento. La llamada inicial que deseamos es `seleccion(v,0,long(v)-1,k)`.
- ⋆ En esta versión del algoritmo 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);}
  }
};
```

- ⋆ El caso peor de este algoritmo se da cuando el pivote queda siempre en un extremo del subvector correspondiente y la llamada recursiva se hace con todos los elementos menos el pivote: por ejemplo si el vector está ordenado y le pido el último elemento. En dicho caso el coste está en $O(n^2)$ siendo $n = b - a + 1$ el tamaño del vector. La situación es similar a la del algoritmo de ordenación rápida.
- ⋆ Nos preguntamos qué podemos hacer para asegurarnos de que el tamaño del problema se divida aproximadamente por la mitad. Si en lugar de usar el primer elemento del vector como pivote usásemos la mediana del vector, entonces solamente tendríamos una llamada recursiva de la mitad de tamaño, lo que nos da un coste en $O(n)$ siendo n el tamaño del vector.
- ⋆ Pero resulta que el problema de la mediana es un caso particular del problema que estamos intentando resolver y del mismo tamaño. Por ello, nos vamos a conformar con una aproximación suficientemente buena de la mediana, conocida como *mediana de las medianas*, con la esperanza de que sea suficiente para tener un coste mejor.
- ⋆ 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.
- ⋆ Para implementar este algoritmo vamos a definir una versión mejorada de partición:
  - Recibe como argumento el pivote con respecto al cual queremos hacer la partición, con lo que es más general. Así podemos calcular primero la mediana de las medianas y dársela después a partición.
  - En lugar de devolver solamente una posición p vamos a devolver dos posiciones p y q que delimitan la parte de los elementos que son estrictamente menores que el pivote (desde a hasta $p-1$), las que son iguales al pivote (desde p hasta q) y las que son estrictamente mayores (desde $q+1$ hasta b). De esta forma, si el pivote se repite muchas veces el tamaño se reduce más.

- Este algoritmo también se puede usar en el de la ordenación rápida. Así, por ejemplo, si todos los elementos del vector son iguales, con el anterior algoritmo de partición se tiene coste $O(n \log n)$ mientras con el nuevo, descartando la parte de los que son iguales al pivote, tenemos coste en $O(n)$.
- Para implementarlo usamos tres índices p, q, k: los dos primeros se mueven hacia la derecha y el tercero hacia la izquierda. El invariante nos indica que los elementos en $v[a..p-1]$ son estrictamente menores que el pivote, los de $v[p..k-1]$ son iguales al pivote y los de $v[q+1..b]$ son estrictamente mayores. La parte $v[k..q]$ está sin explorar hasta que $k = q+1$ en cuyo caso el bucle termina y tenemos lo que queremos.
- En cada paso 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, ya que $v[i]$ es igual al pivote; 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)-1
int k;
TElem aux;
p=a;k=a;q=b;

//INV: a<=p<=k<=q+1<=b+1<=long(v)
//     los elementos desde a hasta p-1 son < pivote
//     los elementos desde p hasta k-1 son = pivote
//     los elementos desde q+1 hasta b son > pivote
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
}
```

⋆ Por tanto, los 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`:

```
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);}
```

⋆ 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.

⋆ Con estas ideas el algoritmo queda:

```
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..long(v)-1]
```

donde *ordenarInsercion* es una versión del algoritmo de ordenación por inserción más general que el que vimos en el Capítulo 3, ya que podemos indicar el trozo del vector que deseamos ordenar.

⋆ La llamada inicial es `seleccion2(v, 0, long(v)-1, k)`.

⋆ Se puede demostrar, por inducción constructiva, que el tiempo requerido por *seleccion2* en el caso peor es lineal en $n = b - a + 1$ Brassard y Bratley (1997).

4. 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.

⋆ Es fácil ver que 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.

⋆ Una posible forma de representar la solución al problema es en forma de matriz de n por n, donde se busca rellenar, en cada celda a_{ij}, el día que se enfrentarán entre sí los contrincantes i y j, con $j < i$. Es decir, tratamos de rellenar, con fechas de encuentros, el área bajo la diagonal de esta matriz; sin que en ninguna fila o columna haya días repetidos.

⋆ 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.

⋆ Podemos ensayar una solución DV según las siguientes ideas (ver Figura 2):

- 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. En estos sub-torneos las condiciones son idénticas a las del torneo inicial por ser n una potencia de 2; con la salvedad de que se pueden jugar ambos en paralelo.
- Después se planifican partidas en las que un participante pertenece a A y el otro a B. En estas partidas, que no se pueden solapar con los sub-torneos, hay que rellenar todas las celdas de la matriz correspondiente.

⋆ Esta última parte se puede resolver fácilmente fila por fila, rotando, en cada nueva fila, el orden de las fechas disponibles. Como hay que rellenar $\frac{n}{2} \cdot \frac{n}{2}$ celdas, el coste de esta fase está en 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 consiste en rellenar $\Theta(n^2)$ celdas. Pasamos entonces a precisar los detalles.

4.1. Implementación

⋆ Usaremos una matriz cuadrada declarada como `int a[MAX][MAX]` (donde `MAX` es una constante entera) para almacenar la solución, inicializada con ceros. La primera fecha disponible será el dia 1.

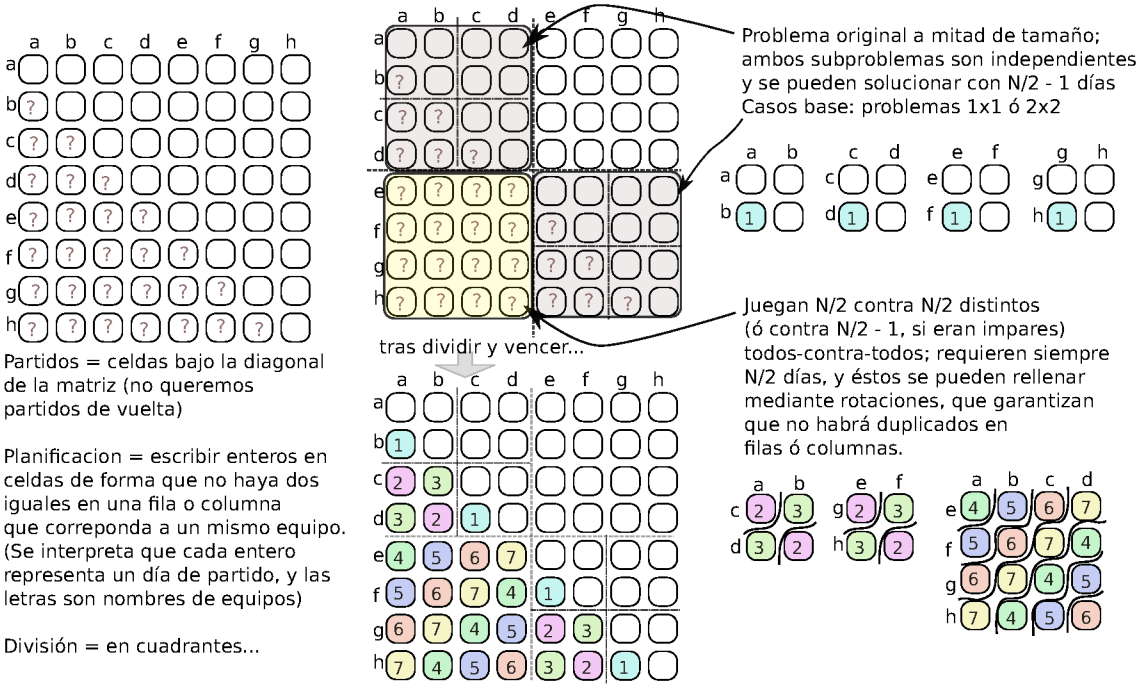


Figura 1: Solución gráfica del problema del torneo

⋆ De forma similar a ejemplos anteriores la función recursiva, llamada *rellena*, recibe dos parámetros adicionales, c y f que delimitan el trozo de la matriz que estamos rellenando, y que por tanto se inicializan respectivamente con 0 y $num - 1$, siendo num el número de participantes. 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 $mitad + (i + j)\,\%mitad$.

```
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);       //cuadrante superior izquierdo
      rellena(a,c+mitad,f);          //cuadrante inferior derecho
      for (int i=c+mitad; i<=f; i++){  //cuadrante inferior izquierdo
        for (int j=c; j<=c+mitad-1;j++)
             {a[i][j]=mitad+(i+j)%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])∧
//       ∀i : c ≤ i ≤ f : (∀j : c ≤ j < i : (∀k : i < k ≤ f : a[i][j] ≠ a[k][i]))
```

```
}
```

5. 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 tanto en tiempo como en espacio adicional. 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 la Sección 4.2 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.

5.1. Corrección

⋆ Consideremos un rectángulo cualquiera de anchura 2δ y altura δ centrado en la línea divisoria l (ver Figura 2). 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

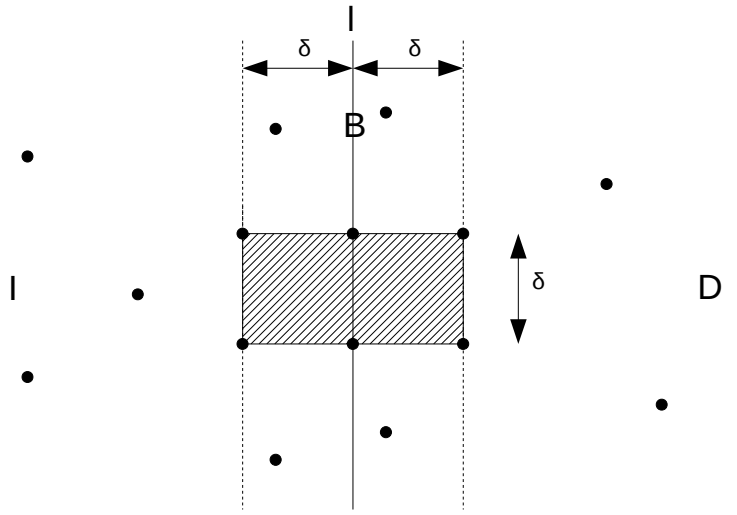


Figura 2: Razonamiento de corrección del problema del par más cercano

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.

5.2. Implementación

⋆ Definimos un punto como una pareja de números reales.

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

La entrada al algoritmo será un vector p de puntos y los límites c y f de la nube que estamos mirando. El vector de puntos no se modificará en ningún momento y se supone que está ordenado respecto a la coordenada x.

⋆ La solución

```
void parMasCercano(Punto p[], int c, int f, int indY[], int& ini,
                   double& d, int& p1, int& p2)
```

constará de:

- Los límites c y f. Las llamadas correctas cumplen $0 \leq c \wedge f \leq long(v) - 1 \wedge f \geq c + 1$.
- 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. El índice inicial ini

nos indica que el punto $p[ini]$ es el que tiene la menor coordenada y. El vector $indY$ contiene en cada posición la posición del siguiente punto en la ordenación, siendo -1 el valor utilizado para indicar que no hay siguiente. Por ejemplo, si el vector de puntos es:

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

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

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

Es decir:

- el elemento con menor y es el punto $p[2]$;
- como $indY[2] = 3$ el siguiente es $p[3]$;
- como $indY[3] = 0$, el siguiente es $p[0]$;
- como $indY[0] = 4$, el siguiente es $p[4]$;
- como $indY[4] = 5$, el siguiente es $p[5]$;
- como $indY[5] = 6$, el siguiente es $p[6]$;
- como $indY[6] = 1$, el siguiente es $p[1]$;
- como $indY[1] = -1$, ya no hay más puntos

⋆ Usaremos las siguientes funciones auxiliares:

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

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

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

⋆ Los casos base, cuando hay 2 o 3 puntos los resuelve la función

```
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)
    {
    //Menor distancia y puntos que la producen
```

```
        d1=distancia(p[c],p[c+1]);
        d2=distancia(p[c],p[c+2]);
        d3=distancia(p[c+1],p[c+2]);
        d = minimo(minimo(d1,d2),d3);
        if (d==d1) {p1=c;p2=c+1;}
        else if (d==d2) {p1=c;p2=c+2;}
        else {p1=c+1;p2=c+2;};

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

⋆ El método *mezclaOrdenada* recibe en *indY* dos listas de enlaces que comienzan en *ini1* y *ini2* que representan respectivamente la ordenación con respecto a y de los puntos de la nube izquierda y derecha, y las mezcla para obtener una única lista de enlaces con todos los puntos de la nube.

```
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;};
};
```

En el ejemplo de antes, despues de las dos llamadas recursivas tendríamos que *ini1* =

2, $ini2 = 4$ y el vector $indY$ es:

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

representando dos listas de puntos $p[2]$, $p[3]$, $p[0]$, $p[1]$ y por otro lado $p[4]$, $p[5]$, $p[6]$. Después de ejecutar *mezclaOrdenada* obtenemos el resultado de arriba. Este método se puede utilizar igualmente en una versión del algoritmo *mergesort* que devuelva un vector de enlaces.

⋆ Así el algoritmo queda:

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

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

```
            count=count+1;
            };
         i=indF[i];
        };
  }

};
```

⋆ La llamada inicial es:

```
parMasCercano(p, 0, long(v)-1, indY, ini, d, p1, p2).
```

6. 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;
         }
}
```

⋆ Es decir, 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:

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

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

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

⋆ 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.

Notas bibliográficas

En (Martí Oliet et al., 2013, Cap. 11) se repasan los fundamentos de la técnica DV y hay numerosos ejercicios resueltos. El capítulo (Brassard y Bratley, 1997, Cap. 7) también está dedicado a la técnica DV y uno de los ejemplos es el algoritmo de Karatsuba y Ofman. El ejemplo del par más cercano está tomado de (Cormen et al., 2001, Cap. 33).

Ejercicios

1. Dos amigos matan el tiempo de espera en la cola del cine jugando a un juego muy sencillo: uno de ellos piensa un número natural positivo y el otro debe adivinarlo preguntando solamente si es menor o igual que otros números. Diseñar un algoritmo eficiente para adivinar el número.

2. Desarrollar un algoritmo DV para multiplicar n número complejos usando tan solo $3(n-1)$ multiplicaciones.

3. Dados un vector $v[0..n-1]$ y un valor k, $1 \leq k \leq n$, diseñar un algoritmo DV de coste constante en espacio que trasponga las k primeras posiciones del vector con las $n-k$ siguientes. Es decir, los elementos $v[0] \ldots v[k-1]$ han de copiarse a las posiciones $n-k \ldots n-1$, y los elementos $v[k] \ldots v[n-1]$ han de copiarse a las posiciones $0 \ldots n-k-1$.

4. Dado un vector $T[1..n]$ de n elementos (que tal vez no se puedan ordenar), se dice que un elemento x es *mayoritario en* T cuando el número de veces que x aparece en T es estrictamente mayor que $N/2$.

   a) Escribir un algoritmo DV que en tiempo $O(n \log n)$ decida si un vector $T[0..n-1]$ contiene un elemento mayoritario y devuelva tal elemento cuando exista.

   b) Suponiendo que los elementos del vector $T[0..n-1]$ se puedan ordenar, y que podemos calcular la mediana de un vector en tiempo lineal, escribir un algoritmo DV que en tiempo lineal decida si $T[0..n-1]$ contiene un elemento mayoritario y devueva tal elemento cuando exista. La mediana se define como el valor que ocuparía la posición $\frac{n-1}{2}$ si el vector estuviese ordenado.

   c) Idear un algoritmo que no sea DV, tenga coste lineal, y no suponga que los elementos se pueden ordenar.

5. *a*) Dado un valor x fijo, escribir un algoritmo para calcular x^n con un coste $O(\log n)$ en términos del número de multiplicaciones.

b) Sea F la matriz $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Calcular el producto del vector $(i\ j)$ y la matriz F. ¿Qué ocurre cuando i y j son números consecutivos de la sucesión de Fibonacci?

c) Utilizando las ideas de los dos apartados anteriores, desarrollar un algoritmo para calcular el n-ésimo número de Fibonacci $f(n)$ con un coste $O(\log n)$ en términos del número de operaciones aritméticas elementales.

6. Diseñar un algoritmo para resolver el siguiente problema e implementarlo como procedimiento. Dado un árbol binario ordenado cuyos nodos contienen información de tipo entero, y dados dos enteros *k1* y *k2*, crear una lista ordenada que contenga los valores almacenados en el árbol que sean mayores o iguales que *k1* y menores o iguales que *k2*.

7. En un habitación oscura se tienen dos cajones, en uno de los cuales hay n tornillos de varios tamaños, y en el otro las correspondientes n tuercas. Es necesario emparejar cada tornillo con su tuerca correspondiente, pero debido a la oscuridad no se pueden comparar tornillos con tornillos ni tuercas con tuercas, y la única comparación posible es la de intentar enroscar una tuerca en un tornillo para comprobar si es demasiado grande, demasiado pequeña, o se ajusta perfectamente al tornillo. Desarrollar un algoritmo para emparejar los tornillos con las tuercas que use $O(n \log n)$ comparaciones en promedio.

8. Dado un vector $C[0..n-1]$ de números enteros distintos, y un número entero S, se pide:

   a) Diseñar un algoritmo de complejidad $\Theta(n \log n)$ que determine si existen o no dos elementos de C tales que su suma sea exactamente S.

   b) Suponiendo ahora ordenado el vector C, diseñar un algoritmo que resuelva el mismo problema en tiempo $\Theta(n)$.

9. Mr. Scrooge ha cobrado una antigua deuda, recibiendo una bolsa con n monedas de oro. Su olfato de usurero le asegura que una de ellas es falsa, pero lo único que la distingue de las demás es su peso, aunque no sabe si este es mayor o menor que el de las otras. Para descubrir cuál es la falsa, Mr. Scrooge solo dispone de una balanza con dos platillos para comparar el peso de dos conjuntos de monedas. En cada pesada lo único que puede observar es si la balanza queda equilibrada, si pesan más los objetos del platillo de la derecha o si pesan más los de la izquierda. Suponiendo $n \geq 3$, diseñar un algoritmo DV para encontrar la moneda falsa y decidir si pesa más o menos que las auténticas.

10. Se dice que un punto del plano $A = (a_1, a_2)$ domina a otro $B = (b_1, b_2)$ si $a_1 > b_1$ y $a_2 > b_2$. Dado un conjunto S de puntos en el plano, el rango de un punto $A \in S$ es el número de puntos que domina. Escribir un algoritmo de coste $O(n \log^2 n)$ que dado un conjunto de n puntos calcule el rango de cada uno; demostrar que su coste efectivamente es el requerido.

11. **La línea del cielo de Manhattan**. Dado un conjunto de n rectángulos (los edificios), cuyas bases descansan todas sobre el eje de abcisas, hay que determinar mediante DV la cobertura superior de la colección (la línea del cielo).

    La línea del cielo puede verse como una lista ordenada de segmentos horizontales, cada uno comenzando en la abcisa donde el segmento precedente terminó. De esta

forma, puede representarse mediante una secuencia alternante de abcisas y ordenadas, donde cada ordenada indica la altura de su correspondiente segmento:

$$(x_1, y_1, x_2, y_2, \ldots, x_{m-1}, y_{m-1}, x_m)$$

con $x_1 < x_2 < ... < x_m$. Por convenio, la línea del cielo está a altura 0 hasta alcanzar x_1, y vuelve a 0 después de x_m. Se usa la misma representación para cada rectángulo, es decir (x_1, y_1, x_2) corresponde al rectángulo con vértices $(x_1, 0)$, (x_1, y_1), (x_2, y_1) y $(x_2, 0)$.

12. Sea un vector $V[0..n-1]$ que contiene valores enteros positivos que se ajustan al perfil de una curva cóncava; es decir, para una cierta posición k, tal que $0 \leq k < n$, se cumple que $\forall j \in \{0..k-1\}.V[j] > V[j+1]$ y $\forall j \in \{k+1..n-1\}.V[j-1] < V[j]$ (por ejemplo el vector $V = [9, 8, 7, 3, 2, 4, 6]$). Se pide diseñar un algoritmo que encuentre la posición del mínimo en el vector (la posición 4 en el ejemplo), teniendo en cuenta que el algoritmo propuesto debe de ser más eficiente que el conocido de búsqueda secuencial del mínimo en un vector cualquiera.

13. La *envolvente convexa* de una nube de puntos en el plano es el menor polígono convexo que incluye a todos los puntos. Si los puntos se representaran mediante clavos en un tablero y extendiéramos una goma elástica alrededor de todos ellos, la forma que adoptaría la goma al soltarla sería la envolvente convexa. Dada una lista l de n puntos, se pide un algoritmo de coste $\Theta(n \log n)$ que calcule su envolvente convexa.

14. Las matrices de Hadamard H_0, H_1, H_2, ..., se definen del siguiente modo:

    - $H_0 = (1)$ es una matriz 1×1.
    - Para $k > 0$, H_k es la matriz $2^k \times 2^k$

$$H_k = \left(\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right)$$

    Si v es un vector columna de longitud $n = 2^k$, escribir un algoritmo que calcule el producto $H_k \cdot v$ en tiempo $O(n \log n)$ suponiendo que las operaciones aritméticas básicas tienen un coste constante. Justificar el coste.

15. Diseñar un algoritmo de coste en $O(n)$ que dado un conjunto S con n números, y un entero positivo $k \leq n$, determine los k números de S más cercanos a la mediana de S.

16. La p–mediana generalizada de una colección de n valores se define como *la media* de los p valores ($p + 1$ si p y n tuvieran distinta paridad) que ocuparían las posiciones centrales si se ordenara la colección. Diseñar un algoritmo que resuelva el problema en un tiempo a lo sumo $O(n \log(n))$, asumiento p constante distinto de n.

17. Se tiene un vector V de números enteros distintos, con pesos asociados $p_1, \ldots, p_n$. Los pesos son valores no negativos y verifican que $\sum_{i=1}^{n} p_i = 1$. Se define la *mediana ponderada* del vector V como el valor $V[m]$, $1 \leq m \leq n$, tal que

$$\Big(\sum_{V[i]<V[m]} p_i\Big) < \frac{1}{2} \qquad \text{y} \qquad \Big(\sum_{V[i]\leq V[m]} p_i\Big) \geq \frac{1}{2}.$$

Por ejemplo, para $n = 5$, $V = [4, 2, 9, 3, 7]$ y $P = [0.15, 0.2, 0.3, 0.1, 0.25]$, la media ponderada es $V[5] = 7$ porque

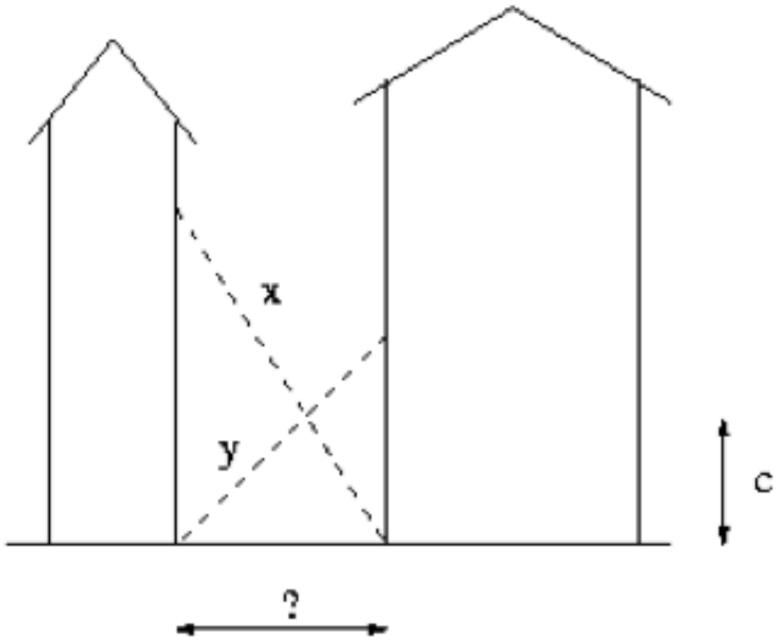
$$\sum_{V[i]<7} p_i = p_1 + p_2 + p_4 = 0.15 + 0.2 + 0.1 = 0.45 < \frac{1}{2}, \text{ y}$$

$$\sum_{V[i]\leq 7} p_i = p_1 + p_2 + p_4 + p_5 = 0.15 + 0.2 + 0.1 + 0.25 = 0.7 \geq \frac{1}{2}.$$

Diseñar un algoritmo de tipo *divide y vencerás* que encuentre la mediana ponderada en un tiempo lineal en el caso peor. (Obsérvese que V puede no estar ordenado.)

18. Se tienen n bolas de igual tamaño, todas ellas de igual peso salvo dos más pesadas, que a su vez pesan lo mismo. Como único medio para dar con dichas bolas se dispone de una balanza romana clásica. Diseñar un algoritmo que permita determinar cuáles son dichas bolas, con el mínimo posible de pesadas (que es logarítmico).
Indicación: Considerar también el caso en el que solo hay una bola diferente.

19. **Escaleras cruzadas** Una calle estrecha está flanqueada por dos edificios muy altos. Se colocan dos escaleras en dicha calle como muestra el dibujo: una de ellas, de longitud x metros, colocada en la base del edificio que está en el lado derecho de la calle y apoyada sobre la fachada del edificio situado en el lado izquierdo de la calle; la otra, de longitud y metros, colocada en la base del edificio que está en el lado izquierdo de la calle y apoyada sobre la fachada del edificio situado en el lado derecho de la calle. El punto donde se cruzan ambas escaleras está a altura exactamente c metros del suelo. Se pide diseñar un algoritmo que calcule la anchura de la calle con tres decimales de precisión.



(Enunciado original en `http://uva.onlinejudge.org/external/105/10566.pdf`)

20. **Resolución de una ecuación** Dados números reales p, q, r, s, t tales que $0 \leq p, r \leq 20$ y $-20 \leq q, s, t \leq 0$, se desea resolver la siguiente ecuación en el intervalo $[0, 1]$ con cuatro decimales de precisión:

$$p * e^{-x} + q * sen(x) + r * cos(x) + s * tan(x) + t * x^2 + u = 0$$

(Enunciado original en `http://uva.onlinejudge.org/external/103/10341.pdf`)