Introducción a la Programación

Tema 12. Problemas iterativos y recursivos

1. Introducción

En este capítulo vamos a estudiar un conjunto de problemas de interés. Veremos problemas resueltos con algoritmos iterativos y otros con algoritmos recursivos. Usaremos los cálculos de la complejidad vistos en temas anteriores.

También veremos formas de adaptar algoritmos para resolver problemas para los que no estaban pensados.

Comenzaremos por ver de una forma general los tratamientos iterativos que estudiamos en capítulos anteriores.

2. Adaptación de problemas al de la potencia entera.

En capítulos anteriores vimos diferentes algoritmos para resolver el problema de la potencia entera. Aquí vamos a ver la posibilidad de adaptar otros problemas para que se puedan usar los algoritmos anteriores para resolverlos.

El problema de la potencia entera es el cálculo de a^n donde *n* es un entero no negativo. En esta definición general entran muchos problemas diferentes:

1. a^n con a de tipo entero o real.
2. A^n con A una matriz cuadrada.
3. $a^n \ mod \ r$. Es decir el resto con respecto a r de a^n sin tener que hacer la operación de potencia y usando resultados intermedios razonablemente pequeños.
4. Una recurrencia lineal del tipo $f_n = a_1 f_{n-1} + \cdots + a_k f_{n-k}$. Con condiciones iniciales $f_{k-1} = c_{k-1,} \ldots, f_0 = c_0$. Este problema puede ser reducido al segundo puesto que la recurrencia puede ser rescrita como una ecuación matricial. Así para el caso $k = 3$ la recurrencia de la forma $f_n = a_1 f_{n-1} + a_2 f_{n-2} + a_3 f_{n-3}$, con condiciones iniciales $f_2 = c_{2,}, f_2 = c_{2,}, f_0 = c_0$, se puede rescribir como:

$$\begin{pmatrix} f_n \\ f_{n-1} \\ f_{n-2} \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & a_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \\ f_{n-3} \end{pmatrix}, \qquad A = \begin{pmatrix} a_1 & a_2 & a_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Y su solución es

$$\begin{pmatrix} f_{n+2} \\ f_{n+1} \\ f_n \end{pmatrix} = A^n \begin{pmatrix} c_2 \\ c_1 \\ c_0 \end{pmatrix}, \qquad n = 0,1, \ldots$$

El caso de problema de Fibonacci visto arriba se transforma por el procedimiento anterior en

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix}, \qquad M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = M^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Donde M es denominada la matriz de Fibonacci.

Como vimos al principio de esta sección el problema de Fibonacci puede representarse como un problema de potencia entera con base una matriz. En efecto tengamos en cuenta:

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix}, \qquad con \qquad \begin{pmatrix} f_1 \\ f_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad por \ lo \ tanto$$

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = M^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \qquad f_n = M^n.get(0,0)$$

Las matrices M^n, son denominadas matrices de *Fibonacci*. Son de la forma $\begin{pmatrix} a+b & a \\ a & b \end{pmatrix}$. Es decir dependen de dos parámetros *a, b.* Esto es debido al hecho de que cualquier matriz cumple su ecuación característica y por lo tanto las potencias de una matriz M de tamaño *k* se pueden expresar mediante *k* parámetros de la forma:

$$M^n = a_{k-1} M^{k-1} + \ a_{k-2} M^{k-2} + \cdots + a_0 M^0$$

Como caso particular la potencia de una matriz $M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, de tamaño 2, cumple:

$$M^n = aM^1 + bM^0 = a\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix} + b\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \begin{pmatrix}a+b & a\\ a & b\end{pmatrix}$$

Debemos tener en cuenta, además, las propiedades:

$$\begin{pmatrix}a+b & a\\ a & b\end{pmatrix}^2 = \begin{pmatrix}2a^2+2ab+b^2 & a^2+2ab\\ a^2+2ab & a^2+b^2\end{pmatrix};$$

$$\begin{pmatrix}a+b & a\\ a & b\end{pmatrix} * \begin{pmatrix}c+d & c\\ c & d\end{pmatrix} = \begin{pmatrix}ac+(a+b)(c+d) & bc+ac+ad\\ a+b & ac+bd\end{pmatrix}$$

$$base = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}$$

$$I = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

Para resolver el problema hay que modelar el tipo *MatriDeFibonacci.* Un valor de este tipo podemos implementarlo por las propiedades (a, b) y los métodos:

- $(a, b)^2 = (a^2 + 2ab, a^2 + b^2)$
- $I = (0,1)$
- $base = (1,0)$
- $(a, b) * (c, d) = (bc + ac + ad, ac, bd)$

Teniendo en cuenta las propiedades anteriores podemos modelar las Matrices de Fibonacci como un tipo con las operaciones $m^2, m_1 * m_2$ y las constantes que representan $base, I$. Otra forma es representar cada Matriz de Fibonacci por el par (a, b) con las operaciones anteriores. Siguiendo el segundo camino declararemos las variables $(au, bu), (ar, br)$ y la variable temporal (at, bt). Con esa notación, adaptando el algoritmo iterativo de la potencia entera visto en un capítulo anterior y teniendo en cuenta las ideas de asignación paralela, resulta:

```
static Long fibonacci3(Integer n){
	Long ar,br;
	Long au,bu;
	Long at,bt;
	ar = 1L;
	br = 0L;
	au = 0L;
	bu = 1L;
	while(n>0){
		if(n%2==1){
			at = bu*ar+au*ar+au*br;
			bt = au*ar+bu*br;
			au = at;
			bu = bt;
		}
		at = ar*ar+2*ar*br;
		bt = ar*ar+br*br;
		ar = at;
		br = bt;
		n =n/2;
	}
	return au;
}
```

Para valores grandes de n el número de *Fibonacci* no cabe en un *Long*. Una implementación mejor debe usar *BigInteger* en lugar de *Long*.

3. Máximo y Mínimo

Los problemas que siguen vamos a verlos de una manera más general teniendo en cuenta los siguientes elementos: problema original, problema generalizado, propiedades (compartidas e individuales del mismo) y tipo de la solución. Todo ello los resumiremos en una ficha que nos proporcione una idea general del algoritmo con suficientes detalles como para poder implementarlo en C o Java.

El problema consiste en, dada una lista, encontrar los elementos máximo y mínimo con respecto a un orden dado. El problema se generaliza para encontrar el máximo y mínimo entre las posiciones i, j de la lista. Como siempre seguiremos la convención *Java*.

El conjunto de problemas tiene dos propiedades compartidas: *d* (de tipo *List<E>*) que contiene la lista de elementos, *o* (de tipo *Comparator<E>*) que contiene un orden con respeto al cual buscamos el máximo y mínimo. Las propiedades individuales son: *i*, entero en *[0,d.size())*, consultable, y *J*, entero en *(I,d.size()]*, consultable.

La solución al problema tiene dos propiedades: *M* (de tipo *E)* consultable y *m* (de tipo *E*) consultable. Una solución la representaremos por (M, m). Usaremos las funciones *max* y *min* calculan el máximo o mínimo de dos elementos dado un orden como primer parámetro.

En general los valores de un tipo dado los representaremos identificadores con subíndices.

<table>
<tr><th colspan="2">Ficha 1</th></tr>
<tr><th colspan="2">MaxMin</th></tr>
<tr><td colspan="2">Técnica: Divide y Vencerás sin Memoria</td></tr>
<tr><td>Tamaño: N = J-I</td><td></td></tr>
<tr><td>Propiedades Compartidas</td><td>d, List<E>
o, Comparator<E></td></tr>
<tr><td>Propiedades Individuales</td><td>i, entero en [0,d.size())
j, entero en (I,d.size()]</td></tr>
<tr><td colspan="2">Solución: (M, m)</td></tr>
<tr><td colspan="2">Instanciación:
$$MaxMin(d, o) = f(x) = \begin{cases} \bot, & |d| = 0 \\ mm(0, d.size()), & |d| > 0 \end{cases}$$</td></tr>
<tr><td colspan="2">Problema Generalizado
$$mm(i, j) = \begin{cases} (M, m), & j - i = 1, \quad M = m = d(i) \\ (M, m), & j - i = 2, \quad M = \max(or, d(i), d(j)), \; m = \min(or, d(i), d(j)) \\ c(mm(i, s), mm(s, j)), & j - i > 2 \end{cases}$$</td></tr>
</table>

$s = \frac{i+j}{2}$	
$c\big((M_1, m_1), (M_2, m_2)\big) = (\max(or, M_1, M_2), \min(or, m_1, m_2))$	
Recurrencia	$T(n) = 2T\left(\frac{n}{2}\right) + c$
Complejidad	$\Theta\left(n^{\log_b a}\right) = \Theta(n)$

Como vemos la complejidad asintótica resultante es la misma que para el cálculo iterativo equivalente. El algoritmo lo podemos implementar de forma directa.

4. Búsqueda Binaria

El problema consiste en, dada una lista ordenada con respecto a un orden, encontrar, si existe la posición de un elemento dado o *-1* si no lo encuentra. El problema se generaliza añadiendo dos propiedades: *i,* entero en *[0,d.size())] y j,* entero en *[I,d.size())].* El problema generalizado busca el elemento en el segmento de lista definido por el intervalo *[I,J).*

El conjunto de problemas tiene tres propiedades comunes: *d* (de tipo *List<E>*) que contiene la lista ordenada de elementos, *o* (de tipo *Comparator<E>*) que contiene un orden con respeto al cual está ordenada la lista y *e* (de tipo *E*) el elemento a buscar. Además cada problema tiene las dos propiedades individuales *i,* entero en *[0,d.size())] y j,* entero en *[i,d.size())].*

La solución al problema es un entero en *[-1,d.size())*. Si es -1 *e* no está en *d*.

Ficha 9	
Búsqueda Binaria	
Técnica: Divide y Vencerás sin Memoria	
Tamaño: N =J-I	
Propiedades Compartidas	*d, List<E>* *o, Comparator<E>* *e, E*
Propiedades Individuales	*i, entero en [0,d.size())]* *j, entero en [i,d.size())]*
Solución: r en [-1,d.size())	
Instanciación $BusquedaBinaria(d, or, e) = b(0, d.size())$	
Problema Generalizado $b(i,j) = \begin{cases} -1, & i = j \\ s, & i < j, \quad e = d(s) \\ b(i,s), & i < j, \quad e < d(s) \\ b(s+1,j), & i < j, \quad e > d(s) \end{cases}$ $s = \frac{i+j}{2}$	

Recurrencia	$T(n) = T\left(\frac{n}{2}\right) + c$
Complejidad	$\Theta(\log n)$

El algoritmo en Java es como sigue

```
public static <E extends Comparable<? super E>>
  int binarySearch(List<E> lista, E key){
             Ordering<E> ord = Ordering.natural();
             return bSearch(lista,0,lista.size(),key,ord);
  }

public static <E>
  int binarySearch(List<E> lista, E key, Comparator<? super E> cmp){
             Ordering<? super E> ord = Ordering.from(cmp);
             return bSearch(lista,0,lista.size(),key,ord);
  }

private static <E>
  int bSearch(List<E> lista, int i,int j, E key, Ordering<? super E> ord){
             Preconditions.checkArgument(j>=i);
             int r;
             int k;
             if(j-i == 0){
                    r = -i-1;
             }else{
                    k = (i+j)/2;
                    int r1 = ord.compare(key,lista.get(k));
                    if(r1==0){
                           r = k;
                    }else if(r1 <0){
                           r = bSearch(lista,i,k,key,ord);
                    }else{
                           r = bSearch(lista,k+1,j,key,ord);
                    }
             }
             return r;
  }
```

Es conveniente hacer notar la conversión del estilo funcional de la ficha al imperativo del código por una parte. Por otra la generalización adicional para tener en cuenta el orden natural o un posible orden alternativo.

5. Otros ejemplos de algoritmos iterativos más complejos

Veamos en primer lugar un problema conocido en la literatura como **problema de la bandera holandesa.** El problema se enuncia así:

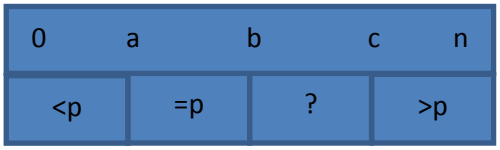
Dada una lista y un elemento del mismo tipo que las casillas, que llamaremos **pivote**, reordenarla, de menor a mayor, para que resulten tres bloques: los menores que el pivote, los iguales al pivote y los mayores que el pivote. El algoritmo debe devolver dos enteros que son las posiciones de las casillas que separan los tres bloques formados.

Para simplificar supongamos las propiedades de problema original son una lista dt de enteros, su tamaño n y un entero p que es el pivote. Es decir el problema original es (dt, n, p) y la

solución buscada $(r1, r2)$. Para que podamos usarlo en algoritmos posteriores vamos a resolver el problema más general de la bandera holandesa sobre un intervalo de la lista de enteros. El intervalo lo representamos por(i, j). El problema completo es (dt, n, i, j, p). Y el dominio que define los problemas de interés es:

$$D(dt, n, i, j, p) \equiv i \geq 0 \wedge i < n \wedge j \geq i \wedge j < n \wedge n \geq 0$$

Para el diseño del algoritmo lo vamos a generalizar añadiendo los parámetros $a, b, c.$ El problema generalizado es $(dt, n, i, j, a, b, c, p)$ con un invariante que viene reflejado en el siguiente gráfico:



Es decir los valores en las casillas en $[0, a)$ son menores que el pivote, los contenidos en $[a, b)$ iguales al pivote, los contenidos en $[b, c)$ no están definidos y los contenidos en $[c, j)$ son mayores que le pivote. Con estas ideas el invariante es:

$$I(dt, n, i, j, a, b, c, p) \equiv$$
$$i \leq a \leq b \leq c \leq j \wedge \left(\bigwedge_{k \in [0,a)} dt[k] < p\right) \wedge \left(\bigwedge_{k \in [a,b)} dt[k] = p\right) \wedge \left(\bigwedge_{k \in [c,j)} dt[k] > p\right)$$

Escogemos como tamaño del problema el número de elementos en el tercer segmento. Es decir $t(dt, n, i, j, a, b, c, p) = c - b.$ El problema final es cuando el tamaño es cero. Es decir cuando *c == b.* Con estas ideas vamos buscando transformaciones (paso de un problema generalizado al siguiente), que manteniendo el invariante, vayan disminuyendo el tamaño. Si mantenemos el invariante cuando el tamaño sea cero hemos conseguido el objetivo que buscábamos.

Ahora nos falta un problema inicial que estando dentro del dominio cumpla el invariante. Una elección posible para el estado inicial (conjunto de valores para las propiedades individuales del problema) es $(a, b, c) = (i, i, j)$. Esta elección es debida a que al empezar no tenemos información sobre los valores del array. Están vacíos los segmentos 1, 2 y 4. Todos los elementos están ubicados en el segmento 3. Ya podemos hacer un esqueleto del algoritmo.

El prototipo de la función que queremos diseñar es en C:

```
void reordena3(int * dt, int i, int, j, int p, int * r1, int * r2);
```

El esquema del algoritmo:

```
void reordena3(int * dt, int i, int j, int p, int * r1, int * r2){
      int a, b, c;
      a = i;
      b = i;
      c = j;
```

```
        while(c-b>0){
                transiciones;
        }
        *r1 = a;
        *r2 = b;
}
```

Nos falta diseñar las transiciones para que se mantenga el invariante y disminuya el tamaño. Para reducir el tamaño podemos considerar el valor en la casilla b, intentar recolocar el valor, aumentar el valor de b en una unidad y posteriormente actualizar los correspondientes valores de a y b. Según el valor de la casilla en b tendremos tres casos posibles: el valor en esa casilla sea menor, igual o mayor que el pivote. Veamos qué hacer en cada caso.

Usaremos la función que intercambia dos casillas en una lista. El prototipo completo de esta función en C es

```
void inter(int a, int b, double * dt);
```

Y el prototipo en Java

```
void inter(int a, int b, List<T> dt);
```

El algoritmo completo en C es:

```
void bh1(int * dt, int i, int j, int p, int * r1, int * r2){
        int a, b, c;
        a = i;
        b = i;
        c = j;
        while(c-b>0){
                if(dt[b]<p){
                        it(a,b,dt);
                        a++;
                        b++;
                }else if(dt[b]==p){
                        b++;
                }else{
                        it(b,c-1,dt);
                        c--;
                }
        }
        *r1 = a;
        *r2 = b;
}
```

Veamos por qué se mantiene el invariante después de cada iteración y en las condiciones iniciales. Las variables *a,b,c* van cambiando de valor en cada iteración. Para hacer que el tamaño del problema disminuya debemos disminuir la región que contiene los valores inciertos. Es decir la región *[b,c)*. Nos fijamos en el valor que ocupa la casilla b y se nos presentan tres casos. En el primero, *dt[b]<p,* debemos tener en cuenta que en la casilla a había un valor igual al pivote y se intercambia por un valor menor que el pivote. Al aumentar los valores de a y b queda restablecido el invariante. En el segundo, *dt[b]==p*, por ser b igual al pivote es suficiente con incrementar su valor. En el tercero se colca el valor de la casilla b al

principio del último bloque. El valor en esa casilla se mueve al bloque de los elementos sin información.

Veamos ahora un problema similar al anterior. Dado una lista y un elemento del mismo tipo que las casillas, que llamaremos pivote, reordenar un segmento del mismo para que queden dos bloques: los menores que el pivote, y los iguales o mayores al pivote. El algoritmo debe devolver un entero que separa los dos bloques formados.

Generalizamos el problema original ahora con dos parámetros (dt, n, i, j, a, b, p). El invariante es ahora:

$$I(dt, n, i, j, a, b, c, p) \equiv$$
$$i \leq a \leq b \leq c \leq j \wedge (\bigwedge_{k \in [0,a)} dt[k] < p) \wedge (\bigwedge_{k \in [b,j)} dt[k] \geq p)$$

Y el tamaño

$$t(dt, n, i, j, a, b, p) = b - a$$

Con todas esas ideas el algoritmo queda:

```
int reordena2(int * dt, int i, int j, int p){
	int a, b;
	a = i;
	b = j;
	while(b-a>0){
		if(dt[a]<p){
			a++;
		}else {
			it(a,b-1,dt);
			b--;
		}
	}
	return a;
}
```

En segundo lugar el problema de fusionar dos lista ordenadas. El problema se enuncia así:

Partimos de las listas ordenadas l, s de tamaños respectivos n, m y queremos encontrar una lista ordenada r que contenga los elementos de las dos listas anteriores.

Generalizamos el problema con las variables i, j, k y escogemos el invariante que la sublista $r[0, k]$ esté ordenada y contenga los elementos de las sublistas $l[0, i]$ y $s[0, j]$. Del invariante escogido podemos derivar la propiedad $k = i + j$ y una de las siguientes situaciones alternativas:

$$0 < i < n, 0 < j < m, l[i-1] \leq s[j-1], r[k-1] = l[i-1]$$
$$0 < i < n, 0 < j < m, l[i-1] > s[j-1], r[k-1] = s[j-1$$
$$0 < i < n, j = m, r[k-1] = l[i-1]$$
$$i = n, 0 < j < m, r[k-1] = s[j-1]$$
$$i = 0, j = 0, k = 0$$

No vamos a mirar con más detalle las condiciones anteriores pero todas ellas son casos que se deducen del requisito de que sublista $r[0,k]$ esté ordenada y contenga los elementos de las sublistas $l[0,i]$ y $s[0,j]$. A partir de aquí el algoritmo iterativo resulta ser:

```
List sum(List l, List s){
      int i = 0;
      int j = 0;
      int k = 0;
      List r = {};
      int n = l.size();
      int m = s.size();
      while(k<n+m){
            if(i<n && j<m && l[i-1]<=s[i-1]){
                  r[k-1] = l[i-1];
                  i = i+1;
            }
            if(i<n && j<m && l[i-1]>s[i-1]){
                  r[k-1] = s[j-1];
                  j = j+1;
            }
            if(i<n && j==m){
                  r[k-1] = l[j-1];
                  i = i+1;
            }
            if(i==n && j<m){
                  r[k-1] = s[j-1];
                  j = j+1;
            }
            k = k+1;
      }
      return r;
}
```

El anterior código es seudocódigo. Hay que instanciarlo en C o Java según convengaSe deja como ejercicio comprobar que se cumple el invariante en cada iteración y que la función de cota $C(n,m,i,j,k) \equiv n + m - k$ disminuye en cada iteración.

6. Ordenación de una lista

El problema consiste en dada una lista ordenarla con respecto a un orden dado. La solución que buscamos es la misma lista de entrada pero ordenada. El problema se generaliza como en problemas anteriores añadiendo dos propiedades: i, j. El problema generalizado ordena el trozo de lista definido por el intervalo $[i, j)$.

El conjunto de problemas tiene dos propiedades comunes: *d* (de tipo *List<E>*) que contiene la lista a ordenar, *or* (de tipo *Comparator<E>*) que contiene un orden con respeto al cual ordenar. Además cada problema tiene las dos propiedades individuales *i,* entero en *[0,d.size()]* y *j,* entero en *[i,d.size()].* En este conjunto de problemas dos problemas de este tipo son iguales si tienen iguales la propiedad *i* y *j*.

El algoritmo usa alguna de las dos versiones del algoritmo de la bandera holandesa anterior. En concreto usaremos la versión 2. El algoritmo generalizado escoge un pivote, reordena las casillas de la lista según el pivote (usando el algoritmo de la bandera holandesa) y posteriormente pasa a reordenar el boque de los menores al pivote y posteriormente el de los mayores o iguales a él.

En lo que para cada algoritmo indicamos sus parámetros de entrada y de salida (que pueden ser varios). Algunos parámetros, como el orden, los obviamos. Para plantear el algoritmo necesitamos varios algoritmos previos:

- $e\ ep(d, i, j)$*:* Elige aleatoriamente un elemento de la sublista *d* definida por el intervalo *[i,j)*. Esencialmente consiste en elegir aleatoriamente un número entero *r* en el intervalo *[i,j)* y devolver el elemento que se encuentra en esa posición en la lista *Datos*. El elemento devuelto se le suele denominar pivote y el método elegir pivote. Este método se ejecuta en tiempo $\Theta(1)$.
- $(d1, k)\ re2(d, i, j, e)$: Reordena la sublista definida por el intervalo *[i,j),* alrededor del pivote *e*. Esto quiere decir que deja todos los elementos menores que el pivote a su izquierda y los mayores o iguales a su derecha. El método devuelve la sublista reordenada y posición del pivote. Este algoritmo puede ser implementado iterativamente con complejidad $\Theta(n)$ según vimos anteriormente.
- $d1 = sb(d, i, j)$: Ordena por algún método iterativo conocido la sublista definida por el intervalo *[i,j).* De estos algoritmos vimos varios en capítulos anteriores (como el de ordenación por inserción) tienen una complejidad de $\Theta(n^2)$.

Ficha 10	
Ordenación de una Lista: Quicksort	
Técnica: Divide y Vencerás sin Memoria	
Tamaño: N =J-I	
Propiedades Compartidas	*d, List<E>* *or, Comparator<E>*
Propiedades Individuales	*i, entero en [0,d.size())* *j, entero en [i,d.size()]*
Solución: List<E>	
Instanciación: $qs(d) = s(0, d.size())$	

Problema Generalizado $s(i,j) = \begin{cases} sb(i,j), & n < n_0 \\ s(i,a); s(b,j), & n > n_0 \end{cases}$ $e = ep(i,j);$ $(a,b) = bh(e,i,j);$ *Donde* bh es el algoritmo de bandera holandesa y sb un algoritmo de ordenación elemental

Recurrencia	$T(n) = n + \frac{1}{n}\sum_{k=0}^{n-1}(T(k) + T(n-k))$
Complejidad	$\Theta(n \log n)$

La ecuación de recurrencia resultante busca el tiempo medio del algoritmo suponiendo iguales probabilidades para cada una de las posibles ubicaciones del pivote. El pivote puede caer en cualquier posición de *0* a n-1. Reordenar la sublista tiene complejidad $\Theta(n)$. La ecuación puede ser resuelta numéricamente.

Hay otras posibilidades de implementación del método *ep(…).* Es decir hay otras formas de elegir el pivote pero esta implementación es sencilla y el algoritmo resultante tiene una complejidad media de $\Theta(n \log n)$ independientemente del caso a resolver.

El cálculo del umbral debemos hacerlo por aproximaciones. El código escrito en Java es:

```
public static <E extends Comparable<? super E>>
   void sort(List<E> lista){
       Ordering<? super E> ord = Ordering.natural();
       quickSort(lista,0,lista.size(),ord);
   }

public static <E>
   void sort(List<E> lista, Comparator<? super E> cmp){
       Ordering<? super E> ord = Ordering.from(cmp);
       quickSort(lista,0,lista.size(),ord);
   }

private static <E>
   void quickSort(List<E> lista, int i, int j, Ordering<? super E> ord){
       Preconditions.checkArgument(j>=i);
       Tupla<Integer,Integer> p;
       if(j-i <= 4){
              Ordenes.ordenaBase(lista, i, j, ord);
       }else{
           E pivote = escogePivote(lista, i, j);
           p = Lists2.reordenaMedianteBanderaHolandesa(lista,pivote,i,j,ord);
           quickSort(lista,i,p.getP1(),ord);
           quickSort(lista,p.getP2(),j,ord);
       }
   }
```

Aquí las funciones *Ordering2.ordenaBase* usa algún método básico para ordenar un problema pequeño y la función *List2.reordenaMedianteBanderaHolandesa* reordena según la primera

versión de la bandera holandesa. La lista a reordenar, al ser un tipo mutable en Java, se ha convertido en un parámetro de entrada salida. Por eso el algoritmo devuelve *void.*

Un algoritmo diferente para resolver el problema es el *Mergesort.* Este algoritmo parte la lista en dos mitades iguales, ordena cada sublista y posteriormente mezcla ordenadamente los resultados. Necesitamos un algoritmo auxiliar que produzca una lista ordenada a partir de dos sublistas ya ordenadas. Es el algoritmo $m(i,k,j)$ que obtiene la sublista ordenada (dt,i,j) asumiendo ordenadas las sublistas (d,i,k) y (d,k,j). Ya hemos visto este algoritmo anteriormente. También necesitamos la función $cp(i,j)$ que copia la sublista (dt,i,j) en (d,i,j). Ambas funciones pueden ser diseñadas iterativamente. En cada caso se mantiene un invariante que dejamos como ejercicio.

<table>
<tr><th colspan="2">Ficha 11</th></tr>
<tr><th colspan="2">Ordenación de una Lista: Mergesort</th></tr>
<tr><td colspan="2">Técnica: Divide y Vencerás sin Memoria</td></tr>
<tr><td>Tamaño: N = J-I</td><td></td></tr>
<tr><td>Propiedades Compartidas</td><td>d, List<E>
dt, List<E
o, Comparator<E></td></tr>
<tr><td>Propiedades Individuales</td><td>i, entero en [0,d.size())
j, entero en [i,d.size()]</td></tr>
<tr><td colspan="2">Solución: List<E></td></tr>
<tr><td colspan="2">Instanciación:

$ms(d) = s(d, 0, d.size(), dt)$</td></tr>
<tr><td colspan="2">Problema Generalizado

$s(i,j) = \begin{cases} sb(i,j), & n < n_0 \\ s(i,k); s(k,j); m(i,k,j); cp(i,j) & n > n_0 \end{cases}$

Donde

$k = \frac{i+j}{2}$</td></tr>
<tr><td>Recurrencia</td><td>$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$</td></tr>
<tr><td>Complejidad</td><td>$\Theta(n \log n)$</td></tr>
</table>

El código asociado es:

```
public static <E extends Comparable<? super E>> void mergeSort(List<E> lista){
       Ordering<? super E> ord = Ordering.natural();
       List<E> ls = Lists.newArrayList(lista);
       mgSort(lista,0,lista.size(),ord,ls);
}

public static <E> void mergeSort(List<E> lista, Comparator<? super E> cmp){
```

```
        Ordering<? super E> ord = Ordering.from(cmp);
        List<E> ls = Lists.newArrayList(lista);
        mgSort(lista,0,lista.size(),ord,ls);
}

private static <E> void mgSort(List<E> lista, int i, int j,
                Ordering<? super E> ord, List<E> ls){
        if(j-i > 1){
                int k = (j+i)/2;
                mgSort(lista,i,k,ord,ls);
                mgSort(lista,k,j,ord,ls);
                mezcla(lista,i,k,lista,k,j,ls,i,j,ord);
                copia(lista,i,j,ls);
        }
}

private static <E> void mezcla(List<E> l1, int i1, int j1,
                List<E> l2, int i2, int j2,List<E> l3, int i3, int j3,
                Ordering<? super E> ord){
        int k1= i1;
        int k2= i2;
        int k3= i3;
        while(k3<j3){
                if(k1<j1 && k2<j2){
                        if(ord.compare(l1.get(k1), l2.get(k2))<=0){
                                l3.set(k3, l1.get(k1));
                                k1++;
                                k3++;
                        }else{
                                l3.set(k3, l2.get(k2));
                                k2++;
                                k3++;
                        }
                }else if(k2==j2){
                        l3.set(k3, l1.get(k1));
                        k1++;
                        k3++;
                }else{
                        l3.set(k3, l2.get(k2));
                        k2++;
                        k3++;
                }
        }
}

private static <E> void copia(List<E> lista, int i, int j, List<E> ls){
        for(int k = i; k<j; k++){
                lista.set(k, ls.get(k));
        }
}
```

El diseño del algoritmo de mezcla anterior es un diseño iterativo. Para su comprensión es adecuado decidir u invariante y demostrar que se mantiene en cada iteración y en el estado inicial.

7. Búsqueda del elemento k-ésimo

El problema consiste en dada una lista no ordenada encontrar el elemento que ocuparía la posición *k-ésima* con respecto a un orden dado. El problema se generaliza añadiendo dos propiedades: *i,* entero en *[0,d.size()] y j,* entero en *[i+1,d.size()].* El problema generalizado

supondrá buscar el *k-ésimo* elemento del segmento de lista *[i,j).* El conjunto de problemas tiene dos propiedades comunes: *d* (de tipo *List<E>*) que contiene la lista, *o* (de tipo *Comparator<E>*) que contiene el orden de referencia y *k* (de tipo entero en *[0,d.size())*.

Para plantear el algoritmo necesitamos varios algoritmos previos algunos de los cuales son similares a los usados en el algoritmo de ordenación previo:

- $e = ep(d, i, j)$*:* Elige aleatoriamente un elemento de la sublista *d* definida por el intervalo *[i,j)*. Esencialmente consiste en elegir aleatoriamente un número entero *r* en el intervalo *[i,j)* y devolver el elemento que se encuentra en esa posición en la lista *Datos*. El elemento devuelto se le suele denominar pivote y el método elegir pivote. Este método se ejecuta en tiempo $\Theta(1)$.
- $(a, b) = bh(e, i, j)$: Reordena la sublista definida por el intervalo *[i,j),* alrededor del pivote *e*. Esto quiere decir que deja todos los elementos menores que el pivote en *[i,a)*, los iguales en *[a,b)* y los mayores en *[b,j)*. El método devuelve la sublista reordenada y las posiciones *a,b*. Este algoritmo puede ser implementado iterativamente con complejidad $\Theta(n)$ según vimos anteriormente.

<table>
<tr><th colspan="2">Ficha 12</th></tr>
<tr><th colspan="2">Búsqueda del elemento k-ésimo</th></tr>
<tr><td colspan="2">Técnica: Divide y Vencerás sin Memoria</td></tr>
<tr><td>Tamaño: N = J-I</td><td></td></tr>
<tr><td>Propiedades Compartidas</td><td>d, List<E>
or, Comparator<E>
k, entero en [0,d.size())</td></tr>
<tr><td>Propiedades Individuales</td><td>i, entero en [0,d.siz)
j, entero en [i+1,d.size())]</td></tr>
<tr><td colspan="2">Solución: E</td></tr>
<tr><td colspan="2">Inicialización
$bk(d, or, k) = ke(d, 0, d.size())$</td></tr>
<tr><td colspan="2">Generalización
$$ke(i,j,k) = \begin{cases} d(a), & k \in [a,b) \\ ke(i,a), & k < a \\ ke(b,j), & k \geq b \end{cases}$$
$$e = ep(i,j);$$
$$(a,b) = bh(e,i,j);$$</td></tr>
<tr><td>Recurrencia</td><td>$T(n) = n + \frac{1}{n}\sum_{s=0}^{n-1} T(s)$</td></tr>
<tr><td>Complejidad</td><td>$\Theta(n)$</td></tr>
</table>

Asumimos que *a* puede caer en cualquier posición de $[0, n)$ con probabilidad $\frac{1}{n}$. Para cada valor de *a* asumimos que con igual probabilidad *k* puede ser menor que *a* o mayor que *a*. La ecuación de recurrencia se obtiene teniendo en cuenta que:

$$\sum_{s=0}^{n-1} \frac{1}{2}(T(s) + T(n-s)) = \sum_{s=0}^{n-1} T(s)$$

Casos particulares de este algoritmo son el cálculo de la mediana (*n/2*-ésimo elemento), primer cuartil (*n/4*-ésimo elemento), etc.

8. Multiplicación de enteros muy grandes

El problema consiste en diseñar un algoritmo para multiplicar enteros muy grandes. Suponemos que un entero está representado en notación decimal por una cadena de caracteres posiblemente muy larga. Sea el tamaño del problema la longitud de esta cadena. Los algoritmos clásicos de multiplicación de enteros tienen una complejidad de $\Theta(n^2)$. La suma de enteros tiene complejidad $\Theta(n)$.

Para plantear el problema modelemos en primer lugar un entero grande. Vamos a dotarlo de tres propiedades: *Sup, Inf, Add*. Si representamos un entero grande por una cadena de caracteres, que representa en el entero en base 10, entonces las propiedades anteriores nos dan respectivamente: el entero representado por la mitad superior de la cadena, el representado por mitad inferior y la suma de ambos. Representaremos por $e, e_1, e_2, \ldots$ diversos enteros grandes y por e^s, e^i las partes superior e inferior respectivamente del entero *e* y $e^a = e^s + e^i$ la suma de ambas partes del entero. También Suponiendo que el tamaño de *e* es *n* y de e^i es *k* entonces podemos escribir $e = 10^k e^s + e^i$. Podemos comprobarlo en el ejemplo $123 = 12(10)^1 + 3$. A partir de esta idea podemos multiplicar dos enteros cuyas partes inferiores tengan el mismo tamaño para obtener:

1. $e = e_1 e_2 = \left(10^k e_1^s + e_1^i\right)\left(10^k e_2^s + e_2^i\right) = 10^{2k} e_1^s e_1^s + 10^k (e_1^s e_2^i + e_1^i e_2^s) + e_1^i e_2^i$

A su vez vemos que siendo

$$p = e_1^s e_2^s, q = e_1^i e_2^i, r = \left(e_1^s + e_1^i\right)\left(e_2^s + e_2^i\right), r - p - q = e_1^s e_2^i + e_1^i e_2^s$$

Por lo que podemos reescribir la expresión para un entero *e* como:

2. $e = 10^{2k} p + 10^k (r - p - q) + q$

El entero que hemos modelado tiene las operaciones e^s, e^i y $10^k e$ cuya complejidad $\Theta(1)$ y la operación _+_ de complejidad $\Theta(n)$. A partir de las ideas anteriores podemos diseñar dos algoritmos: el primero basado en la expresión *1* y el segundo basado en la expresión *2*. Si el tamaño del entero e_1 es n_1 y el de e_2 es n_2 entonces el tamaño del problema de la multiplicación de ambos enteros es $n = \max(n_1, n_2)$. Sea, a su vez, $k = \frac{n}{2}$.

En el caso 1 el problema se divide en cuatro sub-problemas siendo cada problema de tamaño mitad y la función que combina las soluciones de la forma:

$$c(s_1, s_2, s_3, s_4) = 10^{2k}s_1 + 10^k(s_2 + s_3) + s_4$$

Cuya complejidad es $\Theta(n)$ al ser de este orden la suma de enteros y de orden constante la multiplicación por potencias de 10. La recurrencia para el tiempo de ejecución del algoritmo es para el primer caso:

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n), \qquad T(n) = \Theta(n^2)$$

En el caso 2 el problema se divide en tres sub-problemas de tamaño mitad y donde la función de combinación de las soluciones es:

$$c(s_1, s_2, s_3) = 10^{2k}s_1 + 10^k(s_2 - s_1 - s_3) + s_3$$

y el sub-problema 2 en este caso es de la forma $\left(e_1^s + e_1^i\right)\left(e_2^s + e_2^i\right) = e_1^a e_2^a$. Vemos que en el caso 2 el cálculo de los sub-problemas más el tiempo de combinación de las soluciones es $\Theta(n)$. La recurrencia para el tiempo de ejecución del algoritmo para el caso 2 es:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n), \qquad T(n) = \Theta\left(n^{\log_2 3}\right) = \Theta(n^{1{,}58})$$

La versión 2 es la que tiene complejidad más baja. En la literatura la versión 2 se le llama el algoritmo de *Karatsuba*. Los detalles de ese algoritmo se muestran en la Ficha siguiente:

Ficha 13	
Multiplicación de Enteros Grandes	
Técnica: Divide y Vencerás sin Memoria	
Tamaño: N = max(n_1, n_2)	
Propiedades Compartidas	
Propiedades Individuales	E_1, *entero* E_2, *entero*
Solución: Entero	
$m(e_1, e_2) = \begin{cases} mo(e_1, e_2), & n < n_0 \\ c\left(m(e_1^s, e_2^s), m(e_1^a, e_2^a), m\left(e_1^i, e_2^i\right)\right), & n \geq n_0 \end{cases}$ $c(s_1, s_2, s_3) = 10^{2k}s_1 + 10^k(s_2 - s_1 - s_3) + s_3$	
Recurrencia	$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$
Complejidad	$\Theta(n^{1{,}58})$

Para resolver el problema debemos modelar el entero grande. Esto lo podemos hacer con el tipo *EnteroGrande* cuya interfaz es:

```
class EnteroGrande {
       int size();
       int getK();
       void setK(int k);
```

```
        EnteroGrande por10(int k);
        EnteroGrande getS();
        EnteroGrande getI();
        EnteroGrande getA();
        EnteroGrande getM(EnteroGrande e);
}
```

El algoritmo diseñado será la implementación del método *getM.* El resto de los métodos representan las propiedades comentadas. El método *setK* fija el tamaño de la parte inferior. El método *getA* (suma de dos enteros grandes) puede hacerse en tiempo lineal y se deja como ejercicio al igual que la solución del caso base.

9. Subsecuencia de suma máxima

Supongamos ahora que tenemos una secuencia de enteros, positivos y negativos. El problema que se trata de resolver es encontrar la sub-secuencia cuya suma sea máxima. Por ejemplo, si tenemos la secuencia 1, -2, ***11, -4, 13***, -5, 2, 3, entonces la sub-secuencia de suma máxima es la sombreada, que suma 20.

La solución la modelamos mediante el tipo *SubSecuenciaMax* con tres propiedades: *I*, *J*, *V* de tipo entero. Las dos primeras indican los límites de la sub-secuencia y la tercera el valor de la suma de los elementos en la sub-secuencia. Una solución la representaremos por $s^{i,j,v}$.

El problema tiene una propiedad compartida: *d* (de tipo *List<Integer>*) que contiene la secuencia de elementos. El problema se generaliza añadiendo dos propiedades individuales: *i,* entero en *[0,d.size()) y j,* entero en *[i+1,d.size()],* que definen la sublista en el intervalo *[i,j).*

Para el diseño concreto necesitamos el algoritmo que llamaremos *Secuencia Centrada*. La llamada al algoritmo es de la forma $(a, b, v) = sc(i, j, k)$ que devuelve la sub-secuencia de suma máxima de entre todas las que se extienden en posiciones en [*i,j)* y contienen los elementos en las posiciones k-1 y k que están contenidas en [*i,j)*. La complejidad del algoritmo *sc(…)* es $\Theta(n)$.

Las ideas las resumimos en la Ficha siguiente:

Ficha 14	
Subsecuencia del Suma Máxima	
Técnica: Divide y Vencerás sin Memoria	
Tamaño: N =J-I	
Propiedades Compartidas	*d, List<Integer>*
Propiedades Individuales	*i, entero en [0,d.size())* *j, entero en [i+1,d.size()]* *v, entero*
Solución: (a,b,v) *Con relación de orden definida por v.*	
Inicialización $psm(d) = sm(0,d.size())$	

Generalización	
$sm(i,j) = \begin{cases} (i,j,d[i]), & j-i=1 \\ c(sm(i,k),sm(k,j),sc(i,j,k)), & j-i>1 \end{cases}$ $k = \frac{i+j}{2}$ $c(s_1,s_2,s_3) = \max(s_1,s_2,s_3)$	
Recurrencia	$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
Complejidad	$\Theta(n \log n)$

La implementación en Java es de la forma:

```
public static SubSecuencia getSubSecuenciaMaxima(List<Double> lista){
              Ordering<SubSecuencia> ord = Ordering.natural();
       return getSubSecuenciaMaxima(lista,0,lista.size(),ord);
}

private static SubSecuencia getSubSecuenciaMaxima(List<Double> lista,
               int i, int j, Ordering<SubSecuencia> ord){
       SubSecuencia r = null;
       if(j-i <= 1){
              r = new SubSecuencia(lista,i,j);
       }else{
              int k = (i+j)/2;
              SubSecuencia s1 = getSubSecuenciaMaxima(lista, i, k, ord);
              SubSecuencia s2 = getSubSecuenciaMaxima(lista, k, j, ord);
              SubSecuencia s3 = getSubSecuenciaMaximaCentrada(lista,i,j,k);
              r = ord.max(s1, s2, s3);
       }
       return r;
}
```

El cálculo de la subsecuencia centrada puede hacerse mediante:

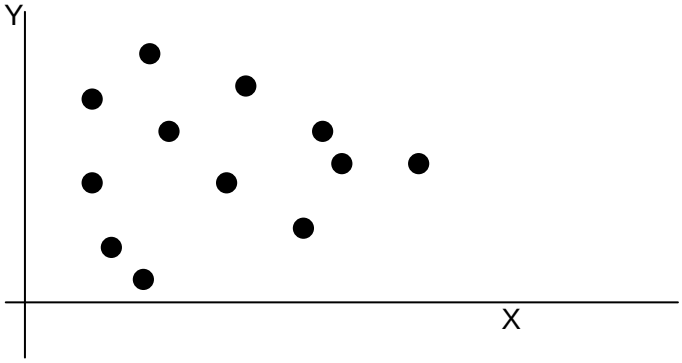
```
private static SubSecuencia getSubSecuenciaMaximaCentrada(List<Double> lista,
         int a, int b, int k){
       Double sumaMaxima = Double.MIN_VALUE;
       Double suma = 0.;
       int i1 = k;
       int j1 = k;
       int from = i1;
       int to = j1;
       for(i1 = k-1;i1 >= a; i1--){
              suma = suma + lista.get(i1);
              if(suma > sumaMaxima){
                     sumaMaxima = suma;
                     from = i1;
              }
       }
       suma = sumaMaxima;
       for(j1=k;j1<b;j1++){
```

```
                suma = suma + lista.get(j1);
                if(suma > sumaMaxima){
                        sumaMaxima = suma;
                        to = j1+1;
                }
        }
        SubSecuencia sm = new SubSecuencia(lista,from,to);
        return sm;
}
```

El algoritmo anterior calcula la subsecuencia de suma máxima centrada en la casilla *k*. Es decir una subsecuencia cuyo s límites contienen a *k*.

10. Cálculo del par más cercano

Dado un conjunto de *n* puntos diseñar mediante la técnica de **divide y vencerás** un algoritmo que devuelva los dos puntos más cercanos. Por lo tanto partimos de que todos los puntos son distintos.



La solución la modelamos con el par $(p1, p2)$ que tiene tres propiedades consultables: *P1*, *P2* de tipo *Punto* y *Distancia* de tipo *Double*. Asumimos modelado el tipo *Punto*.

El problema tiene dos propiedades compartidas: *ps,* de tipo *Set<Punto>*, es el conjunto de puntos original y *px* que es la lista anterior ordenada según la X. Además tiene las propiedades individuales *py,* la misma lista ordenada según la Y, *i, j*, que especifican un subintervalo de puntos *[i,j)* en la lista *px*.

Ficha 15	
Cálculo del Par más Cercano	
Técnica: Divide y Vencerás sin Memoria	
Tamaño: N =J-I	
Propiedades Compartidas	*ps, Set<Punto>* *px, List<Punto>*
Propiedades Individuales	*py, List<Punto>* *i, Integer en [0,p.size())* *j, Integer en (i,p.size())]*
Solución: (p1,p2). Si s=(p1,p2) sea d(s) la distancia entre los puntos del par	
Inicialización	

$pmc(ps) = mc(sort_x(ps), sort_y(ps), 0, ps.size())$
Generalización $mc(px, py, i, j) = \begin{cases} mcb(px, i, j), & j - i < 4 \\ mcc(s, pyC), & j - i \geq 4 \end{cases}$ $k = \frac{i + j}{2}$ $s_1 = mc(px, filter(py, p^x < px[k]), i, k)$ $s_2 = mc(px, filter(py, p^x \geq px[k]), k, j)$ $s = \min(s_1, s_2)$ $pyC = filter(py, \lvert p^x < px[k] \rvert < d(s))$

Recurrencia	$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
Complejidad	$\Theta(n \log n)$

La solución del caso base la hacemos con el algoritmo mcb. Este busca el par más cercano de conjunto de puntos por un método de fuerza bruta. Este algoritmo como otros de este tipo se hace recorriendo todos los pares *(i, j)* posibles. Es decir todos los valores *i, j* que cumplen $0 \leq i < j \leq px.size()$ y quedándonos con aquel para cuya distancia sea menor.

Hemos representado por p^x la propiedad correspondiente de un punto y por $filter(py, p^x < px[k])$ la construcción de una nueva lista filtrando otra mediante un predicado.

El caso recursivo con propiedades *i, j* se parte en dos problemas del mismo tamaño. Sean s_1, s_2 las soluciones de los dos subproblemas. El algoritmo $mcc(s, pyC)$ toma como parámetros s, que es el par de puntos más cercanos de entre los pares cuyos puntos están ambos en la parte derecha o en la izquierda, y pyC, la lista de puntos cuya distancia al punto en el medio es menor que la mínima encontrada en la parte derecha e izquierda. Busca el par de puntos, con un punto en parte derecha y otro en la izquierda, con menor distancia posible entre los que están en pyC. Este algoritmo se puede implementar con complejidad constante. Esto es debido a que para cada punto en pyC los puntos más cercanos a él que $d = d(s)$ están en un rectángulo de base 2*d* y altura *d*, como puede ser comprobado. El número de puntos en este rectángulo es 6 como máximo.

El algoritmo busca el par más cercano en la parte izquierda, luego en la parte derecha, posteriormente entre aquellos que tienen un punto en cada parte y por último se queda con el mínimo de los tres. Esto se implementa pasando a la función mcc el par más cercano de la parte derecha e izquierda y delegando la responsabilidad en esa función de obtener el mínimo con los pares que tienen un punto en cada parte.

El código para implementar esta funcionalidad es en *Java*:

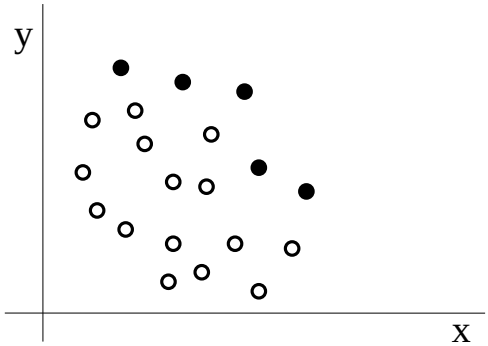
```
private static ParDePuntos masCercano(int i, int j, List<Punto2D> puntosX,
                        List<Punto2D> puntosY, int umbral,
                        Ordering<ParDePuntos> ordNatural){
            ParDePuntos r = null;
            int k = (i+j)/2;
            if(j-i <= umbral){
                  r = parMasCercanoBase(i,j,puntosX);
            }else{
                  List<Punto2D> puntosYIzq = Lists.newArrayList();
                  List<Punto2D> puntosYDer = Lists.newArrayList();
                  Double xk = puntosX.get(k).getX();
                  for(Punto2D p:puntosY){
                        if(p.getX() < xk){
                              puntosYIzq.add(p);
                        }else{
                              puntosYDer.add(p);
                        }
                  }
                  ParDePuntos s1 =
                       masCercano(i,k,puntosX, puntosYIzq,umbral,ordNatural);
                  ParDePuntos s2 =
                       masCercano(k,j,puntosX, puntosYDer,umbral,ordNatural);
                  r = ordNatural.min(s1, s2);
                  List<Punto2D> yCentral = Lists.newArrayList();
                  for(Punto2D p: puntosY){
                        if(Math.abs(p.getX()- xk) < r.getDistancia()){
                                    yCentral.add(p);
                        }
                  }
                  if(!yCentral.isEmpty()){
                        r = masCercanoCentral(r, yCentral, ordNatural);
                  }
            }
            return r;
      }

private static ParDePuntos masCercanoCentral(ParDePuntos s,
                  List<Punto> yCentral, Ordering<ParDePuntos> ordNatural) {
      ParDePuntos r = null;
      double d = s.getDistancia();
      Punto pIzq = null;
      Punto pDer = null;
      for(int i=0; i < yCentral.size(); i++){
            pIzq = yCentral.get(i);
            for(int j=i+1;j<yCentral.size();j++){
                  pDer = yCentral.get(j);
                  r = ParDePuntos.create(pIzq, pDer);
                  if(r.getDistancia()>d){
                        break;
                  }
                  s = ordNatural.min(s, r) ;
                  d = s.getDistancia();
            }
            for(int j=i-1;j>=0;j--){
                  pDer = yCentral.get(j);
                  r = ParDePuntos.create(pIzq, pDer);
                  if(r.getDistancia()>d){
                        break;
                  }
```

```
                    s = ordNatural.min(s, r) ;
                    d = s.getDistancia();
                }
        }
        return s;
}
```

11. Cálculo del conjunto de puntos maximales

Se dice que un punto *P* del plano *domina* a otro *Q* cuando ambas coordenadas de *P* son mayores o iguales que las de *Q*, siendo $P \neq Q$. Si *P* domina a *Q* decimos que *Q* es dominado por *P*. Se dice que un punto *P* es *maximal* en una lista de puntos *L* si no existe ningún punto en *L* que domine a *P*.



Utilizando la técnica de divide y vencerás, obtener un algoritmo que encuentre todos los puntos maximales de una lista de puntos. En el ejemplo de la figura, los puntos en negrita son los maximales del conjunto de todos los puntos dibujados.

La solución la modelamos con el tipo *Set<Punto>*.

El problema tiene una propiedad compartida *ps* de tipo *List<Punto>* y tiene dos propiedades individuales *i*, *j* ambas de tipo *Integer*.

Ficha 16	
Cálculo de Maximales	
Técnica: Divide y Vencerás sin Memoria	
Tamaño: N = J-I	
Propiedades Compartidas	*ps,List<Punto>*
Propiedades Individuales	*I,Integer* *J,Integer*
Solución: Set<Punto>	
Inicialización $pm(sort_x(ps), 0, ps.size())$	
$pm(i,j) = \begin{cases} pmb(i,j), & j-i<2 \\ c\big(pm(i,k), pm(k,j)\big), & j-i \geq 2 \end{cases}$ $k = \frac{i+j}{2}$	
Recurrencia	$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
Complejidad	$\Theta(n \log n)$

La solución del caso base es el conjunto formado por el único punto del problema que será maximal. Si situamos el umbral en un valor mayor que 2 entonces podemos obtener la solución del caso base por un método de fuerza bruta: comprobamos para cada punto si es dominado por alguno de los demás. Si no es dominado por ninguno lo añadimos al conjunto solución de puntos maximales. Esto puede hacerse con una complejidad de $\Theta(n^2)$.

El caso recursivo los resolvemos partiendo el problema en dos subproblemas del mismo tamaño y combinado los dos conjuntos de puntos maximales.

Para hacer la combinación debemos partir de que los puntos maximales del subproblema derecho no pueden ser dominados por ninguno de los del izquierdo. Luego todos son maximales del problema. La solución consiste en añadir a los puntos maximales del problema derecho los maximales del problema izquierdo que tenga un valor de *Y* mayor que el mayor valor de *Y* de todos los maximales del problema derecho. La implementación puede hacerse en tiempo lineal. Primero recorremos los maximales del problema derecho los añadimos a la solución y calculamos y_m (el mayor valor se sus coordenadas Y). Posteriormente añadimos a la solución los maximales del problema izquierdo cuyos valores de Y sean mayores que y_m. Una implementación en Java de la función de combinación es:

```
private static Set<Punto> puntosMaximalesCombina(
                      Set<Punto> si, Set<Punto> sd){
       Double maxYD = Double.MIN_VALUE;
       Set<Punto> r = Sets.newHashSet(sd);
       for(Punto p:sd){
              if(p.getY()>maxYD){
                     maxYD = p.getY();
              }
       }
       for(Punto p:si){
              if(p.getY()> maxYD){
                     r.add(p);
              }
       }
       return r;
}
```