El algoritmo, al igual que en el método mezcla directa, se puede escribir descomponiendo las acciones que realiza en dos rutinas: `separarNatural()` y `mezclaNatural()`. La primera *separa* tramos máximos del archivo original en los dos archivos auxiliares. La segunda *mezcla* tramos máximos de los dos archivos auxiliares y escribe el resultado en el archivo original. El algoritmo termina cuando hay un único tramo; entonces el archivo está ordenado.

```
Ordenación MezclaNatural
inicio
   repetir
      separarNatural(f, f1, f2)
      mezclaNatural(f, f1, f2, numeroTramos)
   hasta numerosTramos = 1
fin
```

7.7. MEZCLA EQUILIBRADA MÚLTIPLE

La eficiencia de los métodos de ordenación externa es directamente proporcional al número de pasadas. Para aumentar la eficiencia hay que reducir el número de pasadas, de esa forma se reduce el número de operaciones de entrada/salida en dispositivos externos. Se observa que el método de *fusión natural* reduce el número de pasadas respecto a la mezcla directa; ambos métodos tienen en común que se utilizan dos archivos auxiliares, además de las dos fases: *separación* y *mezcla*.

Otra forma de reducir el número de *pasadas* es incrementando el número de archivos auxiliares. Supóngase que se tienen `w` tramos distribuidos equitativamente en `m` archivos. La primera *pasada* mezcla los `w` tramos y da lugar a `w/m` tramos. En la siguiente *pasada*, la mezcla de los `w/m` tramos da lugar a w/m^2 tramos; en la siguiente, se reducen a w/m^3; después de `i` *pasadas,* quedarán w/m^i tramos.

Para determinar la complejidad del algoritmo mediante mezcla de *m-uples* tramos, se supone que, *en el peor de los casos,* un archivo de `n` registros tiene `n` tramos iniciales; entonces, el número de pasadas necesarias para la ordenación completa es $[Log_m n]$, como cada pasada realiza `n` operaciones de entrada/salida con los registros, la eficiencia es $O(nLog_m n)$. La mejora obtenida en cuanto a la disminución de las pasadas necesarias para la ordenación hace que los movimientos o transferencias de cada registro sea $Log_2 m$ veces menor.

La sucesión del número de tramos, suponiendo tanto tramos iniciales como registros, sería:

$$n,\ n/m,\ n/m^2,\ n/m^3 \ldots n/m^t = 1$$

tomando logaritmos en base `m`, se calcula el número de tramos `t`:

$$n = m^t;\ Log_m\ n = Log_m\ m^t \Rightarrow t = Log_m\ n$$

7.7.1. Algoritmo de la mezcla equilibrada múltiple

La mezcla equilibrada múltiple utiliza `m` archivos auxiliares, de los que `m/2` son de entrada y `m/2` de salida. Inicialmente, se distribuyen los tramos del archivo de origen en los `m/2` archivos auxiliares. A partir de esta distribución, se repiten los procesos de mezcla reduciendo a la mitad

el número de tramos hasta que queda un único tramo. De esta forma, el proceso de mezcla se realiza en una sola fase en lugar de las dos fases *(separación, fusión)* de los algoritmos mezcla directa y fusión natural. Los pasos que sigue el algoritmo son:

1. Distribuir registros del archivo original por tramos en los `m/2` primeros archivos auxiliares. A continuación, estos se consideran archivos de entrada.
2. Mezclar tramos de los `m/2` archivos de entrada y escribirlos consecutivamente en los `m/2` archivos de salida.
3. Cambiar la finalidad de los archivos, los de entrada pasan a ser de salida y viceversa; repetir a partir del segundo paso hasta que quede un único tramo, entonces la secuencia está ordenada.

7.7.2. Declaración de archivos para la mezcla equilibrada múltiple

La principal variación, en cuanto a las variables que se utilizan, está en que los flujos correspondientes a los archivos auxiliares se agrupan en un *array.* Los registros del archivo a ordenar tienen un campo clave ordinal, respecto al cual se realiza la ordenación. La constante `N` representa el número de archivos auxiliares (flujos); la constante `N2` es el número de archivos de entrada, la mitad de `N`, también es el índice inicial de los flujos de salida (en Java, un *array* se indexa con base cero).

```
static final int N = 6;
static final int N2 = N/2;
File []f = new File[N];
```

La variable `f[]` representa los archivos auxiliares, alternativamente la primera mitad y la segunda irán cambiando su cometido, entrada o salida.

7.7.3. Cambio de finalidad de un archivo: *entrada* ↔ *salida*

La forma de cambiar la finalidad de los archivos (*entrada* ↔ *salida*) se hace mediante una tabla de correspondencia entre índices de archivo, de tal forma que, en lugar de acceder a un archivo por el índice del *array,* se accede por la tabla, la cual cambia alternativamente los índices de los archivos y, de esa forma, pasan, alternativamente, de ser de entrada a ser de salida *(flujos de entrada, flujos de salida).*

`int[] c = new int[N];` *Tabla de índices de archivo.*

Inicialmente, `c[i] = i  ∀i ∈ 0 .. N-1.`

Como consecuencia, los archivos de entrada son:

```
f[c[0]], f[c[1]],…, f[c[N2-1]];
```

y los ficheros de salida son la otra mitad:

```
f[c[N2]], f[c[N2+1]], … f[c[N-1]]
```

Para realizar el cambio de archivo de entrada por salida, se intercambian los valores de las dos mitades de la tabla de correspondencia:

```
c[0]   ↔ c[N2]
c[1]   ↔ c[N2+1]
   .  .  .
c[N2] ↔ c[N-1]
```

En definitiva, con la tabla `c[]` siempre se accede de igual forma a los archivos, lo que cambia son los índices que contiene `c[]`.

Al mezclar tramos de los archivos de entrada no se alcanza el fin de tramo en todos los archivos al mismo tiempo. Un tramo termina cuando es *fin de archivo* (excepción `EOFException`), o bien cuando la siguiente clave es menor que la actual; en cualquier caso, el archivo que le corresponde ha de quedar inactivo. Para despreocuparse de si el archivo está activo o no, se utiliza otro *array* cuyas posiciones indican si el archivo correspondiente al índice está activo. Como en algún momento del proceso de mezcla no todos los archivos de entrada están activos, en otra tabla de correspondencia `cd[]`, sólo para archivos de entrada, se tienen en todo momento los índices de los archivos de entrada activos (en los que no se ha alcanzado el *fin de fichero*).

> ***Nota de programación***
>
> En Java, se procesan los archivos mediante flujos. Entonces, en la codificación se trabaja con flujos de entrada y flujos de salida asociados a los correpondientes archivos.

7.7.4. Control del número de tramos

El primer paso del algoritmo realiza la distribución de los tramos del archivo original en los archivos de entrada, a la vez determina el número de tramos del archivo. En todo momento es importante conocer el número de tramos, ya que cuando queda sólo uno el archivo está ordenado, y éste será el archivo `f[c[0]]`.

En la ejecución del método de ordenación llega un momento en que el número de tramos a mezclar va a ser menor que el número de archivos de entrada. La variable `k1` contiene el número de archivos de entrada. Cuando el número de tramos, `t`, es mayor o igual que la mitad de los archivos, el valor de `k1` es justamente la mitad; en caso de ser `t` menor que dicha mitad, `k1` será igual a `t` y, por último, cuando `t` es `1`, `k1` también es `1` y el archivo `f[c[0]]` está ya ordenado.

7.7.5. Codificación del algoritmo de mezcla equilibrada múltiple

El programa supone, por simplicidad, que el archivo que se ordena está formado por registros de tipo entero. La clase `MzclaMultiple` declara un atributo `File` con el archivo origen, `fo`, y un *array* de referencias a `File`, `f[]`, con los archivos auxiliares. Las operaciones de entrada/salida se realizan con flujos `DataInputStream` y `DataOutputStream` y sus respectivos métodos `readInt()`, `writeInt()`. El proceso de mezcla de un tramo de cada uno de los `m/2` archivos (flujos) de entrada comienza leyendo de cada uno de los flujos un registro (clave de tipo entero) y guardándolo en un *array* de registros `rs[]`. La mezcla selecciona repetidamente el registro

menor, con una llamada al método `minimo()`; el proceso de selección tiene en cuenta que los registros, que se encuentran en el *array* `rs[]`, se correspondan con archivos activos. Cada vez que es seleccionado un registro, se lee el siguiente registro del mismo flujo de entrada, a no ser que haya acabado el tramo; así hasta que se terminan de mezclar todos los tramos. El método `fintramos()` determina si se ha llegado al final de todos los tramos que están involucrados en la mezcla.

La codificación que se presenta a continuación se realiza con `6` archivos auxiliares. En primer lugar, se crea el archivo original con números enteros generados aleatoriamente con llamadas al método `Math.random()`, a continuación, se llama al método que implementa el algoritmo de ordenación *mezcla equilibrada múltiple*.

```
import java.io.*;

class MzclaMultiple
{
   static final int N = 6;
   static final int N2 = N/2;
   static File f0;
   static File []f = new File[N];
   static final int NumReg = 149;
   static final int TOPE = 999;

   public static void main(String []a)
   {
      String[] nomf = {"ar1", "ar2", "ar3", "ar4", "ar5", "ar6"};
      f0 = new File("ArchivoOrigen");
      for (int i = 0; i < N; i++)
         f[i] = new File(nomf[i]);

      DataOutputStream flujo = null;
      // se genera un archivo secuencialmente de claves enteras
      try {
         flujo = new DataOutputStream(
                 new BufferedOutputStream(new FileOutputStream(f0)));

         for (int i = 1; i <= NumReg; i++)
            flujo.writeInt((int)(1+TOPE*Math.random()));
         flujo.close();
         System.out.print("Archivo original ...  ");
         escribir(f0);
         mezclaEqMple();
      }
      catch (IOException e)
      {
         System.out.println("Error entrada/salida durante proceso" +
                                                  " de ordenación ");
         e.printStackTrace();
      }
   }
   //método de ordenación
   public static void mezclaEqMple()
   {
      int i, j, k, k1, t;
      int anterior;
      int [] c = new int[N];
      int [] cd = new int[N];
      int [] r = new int[N2];
```

```
Object [] flujos = new Object[N];
DataInputStream flujoEntradaActual = null;
DataOutputStream flujoSalidaActual = null;
boolean [] actvs = new boolean[N2];
// distribución inicial de tramos desde archivo origen
try {
   t = distribuir();
   for (i = 0; i < N; i++)
      c[i] = i;
   // bucle hasta número de tramos == 1: archivo ordenado
   do {
      k1 = (t < N2) ? t : N2;
      for (i = 0; i < k1; i++)
      {
         flujos[c[i]] = new DataInputStream(
         new BufferedInputStream(new FileInputStream(f[c[i]])));
         cd[i] = c[i];
      }
      j = N2 ; // índice de archivo de salida
      t = 0;
      for (i = j; i < N; i++)
         flujos[c[i]] = new DataOutputStream(
                              new BufferedOutputStream(
                                    new FileOutputStream(f[c[i]])));
      // entrada de una clave de cada flujo
      for (int n = 0; n < k1; n++)
      {
         flujoEntradaActual = (DataInputStream)flujos[cd[n]];
         r[n] = flujoEntradaActual.readInt();
      }

      while (k1 > 0)
      {
         t++; // mezcla de otro tramo
         for (i = 0; i < k1; i++)
            actvs[i] = true;
         flujoSalidaActual = (DataOutputStream)flujos[c[j]];
         while (!finDeTramos(actvs, k1))
         {
            int n;
            n = minimo(r, actvs, k1);
            flujoEntradaActual = (DataInputStream)flujos[cd[n]];
            flujoSalidaActual.writeInt(r[n]);
            anterior = r[n];
            try {
               r[n] = flujoEntradaActual.readInt();
               if (anterior > r[n]) // fin de tramo
                  actvs[n] = false;
            }
            catch (EOFException eof)
            {
               k1--;
               flujoEntradaActual.close();
               cd[n] = cd[k1];
               r[n] = r[k1];
               actvs[n] = actvs[k1];
               actvs[k1] = false;// no se accede a posición k1
            }
         }
```

```
            j = (j < N-1) ? j+1 : N2; // siguiente flujo de salida
         }

         for (i = N2; i < N; i++)
         {
            flujoSalidaActual = (DataOutputStream)flujos[c[i]];
            flujoSalidaActual.close();
         }
         /*
            Cambio de finalidad de los flujos: entrada<->salida
         */
         for (i = 0; i < N2; i++)
         {
            int a;
            a       = c[i];
            c[i]    = c[i+N2];
            c[i+N2] = a;
         }

      } while (t > 1);
      System.out.print("Archivo ordenado ...  ");
      escribir(f[c[0]]);
   }
   catch (IOException er)
   {
      er.printStackTrace();
   }

}
//distribuye tramos de flujos de entrada en flujos de salida
private static int distribuir() throws IOException
{
   int anterior, j, nt;
   int clave;

   DataInputStream flujo = new DataInputStream(
             new BufferedInputStream(new FileInputStream(f0)));
   DataOutputStream [] flujoSalida = new DataOutputStream[N2];
   for (j = 0; j < N2; j++)
   {
      flujoSalida[j] = new DataOutputStream(
      new BufferedOutputStream(new FileOutputStream(f[j])));
   }

   anterior = -TOPE;
   clave = anterior + 1;
   j = 0;     // indice del flujo de salida
   nt = 0;
   // bucle termina con la excepción fin de fichero
   try {
      while (true)
      {
         clave = flujo.readInt();
         while (anterior <= clave)
         {
            flujoSalida[j].writeInt(clave);
            anterior = clave;
            clave = flujo.readInt();
         }
```

```
            nt++;                          // nuevo tramo
            j = (j < N2-1) ? j+1 : 0;   // siguiente archivo
            flujoSalida[j].writeInt(clave);
            anterior = clave;
         }
      }
      catch (EOFException eof)
      {
         nt++;    // cuenta ultimo tramo
         System.out.println("\n*** Número de tramos: " + nt + " ***");
         flujo.close();
         for (j = 0; j < N2; j++)
            flujoSalida[j].close();
         return nt;
      }
   }
   //devuelve el índice del menor valor del array de claves
   private static int minimo(int [] r, boolean [] activo, int n)
   {
      int i, indice;
      int m;

      i = indice = 0;
      m = TOPE+1;

      for ( ; i < n; i++)
      {
         if (activo[i] && r[i] < m)
         {
            m = r[i];
            indice = i;
         }
      }
      return indice;
   }
   //devuelve true si no hay tramo activo
   private static boolean finDeTramos(boolean [] activo, int n)
   {
      boolean s = true;

      for(int k = 0; k < n; k++)
      {
         if (activo[k])    s = false;
      }
      return s;
   }
   //escribe las claves del archivo
   static  void escribir(File f)
   {
      int clave, k;
      boolean mas = true;
      DataInputStream flujo = null;
      try {
         flujo = new DataInputStream(
                 new BufferedInputStream(new FileInputStream(f)));
         k = 0;
         while (mas)
         {
            k++;
            System.out.print(flujo.readInt() + " ");
```

```
            if (k % 19 == 0) System.out.println();
         }
      }
      catch (IOException eof)
      {
         System.out.println("\n *** Fin del archivo ***\n");
         try
         {
            if (eof instanceof EOFException)
               flujo.close();
         }
         catch (IOException er)
         {
            er.printStackTrace();
         }

      }

   }

}
```

7.8. MÉTODO POLIFÁSICO DE ORDENACIÓN EXTERNA

La estrategia seguida en el método de mezcla equilibrada múltiple emplea `2m` archivos para ordenar `n` registros, de tal forma que si los `n` registros están distribuidos en `m` tramos, en una *pasada* (distribución + mezcla) quedan ordenados. La utilización de `2m` archivos puede hacer que el número de éstos sea demasiado elevado y no todas las aplicaciones puedan soportarlo. La mezcla equilibrada múltiple puede mejorarse consiguiendo ordenar `m` tramos con sólo `m+1` archivos; para ello hay que abandonar la idea rígida de *pasada* en la que la finalidad de los archivos de entrada no cambia hasta que se leen todos.

El *método polifásico* utiliza `m` archivos auxiliares para ordenar `n` registros de un archivo. La característica que marca la diferencia de este método respecto a los otros es que continuamente se consideran `m-1` archivos de entrada, desde los que se mezclan registros, y un archivo de salida. En el momento en que uno de los archivos de entrada alcanza su final hay un cambio de cometido, pasa a ser considerado como archivo de salida, y el archivo que en ese momento era de salida pasa a ser de entrada y la mezcla de tramos continúa. La sucesión de pasadas continúa hasta alcanzar el archivo ordenado.

Cabe recordar la propiedad base de todos los métodos de mezcla: *la mezcla de k tramos de los archivos de entrada se transforma en k tramos en el archivo de salida*.

A tener en cuenta

La mezcla polifásica se caracteriza por realizar una mezcla continua de tramos, de tal forma que si se utilizan `m` archivos auxiliares, en un momento dado uno de ellos es archivo de salida y los otros `m-1` archivos son de entrada. Durante el proceso, cuando se alcanza el *registro de fin de archivo* en un archivo de entrada, este pasa a ser de salida, el anterior archivo de salida pasa a ser de entrada y la mezcla continua. La dificultad del método es que el número de tramos iniciales debe pertenecer a una sucesión de números dependiente de `m`.

7.8.1. Mezcla polifásica con m = 3 archivos

A continuación se muestra un ejemplo en el que se supone un archivo original de 55 tramos. El número de archivos auxiliares es m = 3; entonces, en todo momento 2 archivos son de entrada y un tercero de salida.

Inicialmente, el método distribuye los tramos, de manera no uniforme, en los archivos F1, F2. Suponiendo que se dispone de 55 tramos, se sitúan 34 en F1 y 21 en F2; el archivo F3 es, inicialmente, de salida. Empieza la mezcla, y 21 tramos de F1 se fusionan con los 21 tramos de F2 dando lugar a 21 tramos en F3. La situación en este momento es que en F1 quedan 13 tramos, en F2 se ha alcanzado el *fin de fichero* y en F3 hay 21 tramos; F2 pasa a ser archivo de salida y la mezcla continua entre F1 y F3. Ahora se mezclan 13 tramos de F1 con 13 tramos de F3, dando lugar a 13 tramos que se van escribiendo en F2, se ha alcanzado *el fin de fichero* de F1. En el archivo F2 hay 13 tramos y en el F3 quedan 8 tramos, y continúa la mezcla con F1 como archivo de salida. En la nueva pasada se mezclan 8 tramos que se escriben en F1, quedan 5 tramos en F2 y ninguno en F3 ya que se ha alcanzado *el fin de fichero*. El proceso sigue hasta que queda un único tramo y el fichero ha quedado ordenado. La Tabla 7.2 muestra las sucesivas pasadas y los tramos de cada archivo hasta que termina la ordenación

Tabla 7.2 Tramos en cada archivo después de cada pasada en la mezcla polifásica con 3 archivos

		Después de cada pasada							
	Tramos iniciales	F1+F2	F1+F3	F2+F3	F1+F2	F1+F3	F2+F3	F1+F2	F1+F3
F1	34	13	0	8	3	0	2	1	0
F2	21	0	13	5	0	3	1	0	1
F3	0	21	8	0	5	2	0	1	0

Es evidente que se ha partido de una distribución óptima. Además, si se escribe la sucesión de tramos mezclados a partir de la distribución inicial, 21, 13, 8, 5, 3, 2, 1, 1, es justamente la sucesión de los números de Fibonacci:

$f_{i+1} = f_i + f_{i-1} \quad \forall i \geq 1$ tal que $f_1 = 1$, $f_0 = 0$

Entonces, si el número de tramos iniciales f_k es tal que es un número de Fibonacci, la mejor forma de hacer la distribución inicial es según la sucesión de Fibonacci, f_{k-1} y f_{k-2}. Sin embargo, el archivo de origen no siempre dispone de un número de tramos perteneciente a la sucesión de Fibonacci, y en esos casos se recurre a escribir tramos ficticios para conseguir un número de la secuencia de Fibonacci.

Distribución para m = 4 archivos

Se puede extender el proceso de mezcla polifásica a un número mayor de archivos. Por ejemplo, si se parte de un archivo origen con 31 tramos y se utilizan m = 4 archivos para la mezcla polifásica, la distribución inicial y los tramos de cada archivo se muestran en la Tabla 7.3.

Tabla 7.3 Tramos en cada archivo en la mezcla polifásica con 4 archivos

		Después de cada pasada				
	Tramos iniciales	F1+F2+F3	F1+F2+F4	F1+F3+F4	F2+F3+F4	F1+F2+F3
F1	13	6	2	0	1	0
F2	11	4	0	2	1	0
F3	7	0	4	2	1	0
F4	0	7	3	1	0	1

La Tabla 7.4 muestra en forma tabular la distribución perfecta de los tramos en los archivos para cada pasada pero numeradas en orden inverso.

Tabla 7.4 Tramos que intervienen en cada pasada para 3 tramos

L	$t_1^{L)}$	$t_2^{L)}$	$t_3^{L)}$
5	13	11	7
4	7	6	4
3	4	3	2
2	2	2	1
1	1	1	1
0	1	0	0

Observando los datos de la Tabla 7.4 se deducen ciertas relaciones entre t_1, t_2, t_3 en un nivel:

$$t_3^{L+1)} = t_1^{L)}$$
$$t_2^{L+1)} = t_1^{L)} + t_3^{L)}$$
$$t_1^{L+1)} = t_1^{L)} + t_2^{L)}$$
$$\forall\ L > 0 \quad y\ t_1^{0)} = 1\ ,\ t_2^{0)} = 0,\ t_3^{0)} = 0$$

Estas relaciones van a ser muy útiles para encontrar la distribución inicial ideal cuando se desee ordenar un archivo de un número arbitrario de tramos. Además, haciendo el cambio de variable de f_i por $t_1^{L)}$ se tiene la sucesión de los números de Fibonacci de orden 2:

$$f_{i+1} = f_i + f_{i-1} + f_{i-2}\ \forall i \geq 2 \text{ tal que } f_2 = 1,\ f_1 = 0,\ f_0 = 0$$

A tener en cuenta

En general, la sucesión de *números de fibonacci de orden k* tiene la expresión:

$$f_{i+1} = f_i + f_{i-1} + \ldots + f_{i-k+1}\ \forall i \geq k-1 \text{ tal que,}$$
$$f_{k-1} = 1,\ f_{k-2} = 0\ \ldots,\ f_0 = 0$$

7.8.2. Distribución inicial de tramos

En los dos ejemplos expuestos se ha aplicado el método polifásico de manera ideal, la distribución inicial de tramos era perfecta en función del número de archivos auxiliares y de la correspondiente sucesión de Fibonacci. Las fórmulas recurrentes que permiten obtener el número de tramos para cada pasada `L`, en el supuesto de utilizar `m` archivos, son las siguientes:

$$t_{m-1}^{L+1)} = t_1^{L)}$$
$$t_{m-2}^{L+1)} = t_1^{L)} + t_{m-1}^{L)}$$
$$\cdots$$
$$t_2^{L+1)} = t_1^{L)} + t_3^{L)}$$
$$t_1^{L+1)} = t_1^{L)} + t_2^{L)}$$
$$\forall\ L > 0 \quad y\ t_1^{0)} = 1\ ,\ t_i^{0)} = 0 \quad \forall i \le m-1$$

Con estas relaciones puede conocerse de antemano el número de tramos necesarios para aplicar el método polifásico con `m` archivos. Es evidente que no siempre el número de tramos iniciales del archivo a ordenar va a coincidir con ese número ideal, ¿qué hacer? La respuesta es sencilla: se simulan los tramos necesarios para completar la distribución perfecta con tramos vacíos o tramos ficticios. ¿Cómo hay que tratar los tramos ficticios? La selección de un tramo ficticio de un archivo i es, simplemente, ignorar el archivo y, por consiguiente, desecharlo de la mezcla del tramo correspondiente. En el supuesto de que el tramo sea ficticio para los `m-1` archivos de entrada, no habrá que hacer ninguna operación, simplemente considerar un tramo ficticio en el archivo de salida. Los distribución inicial ha de repartir los tramos ficticios lo más uniformemente posible en los `m-1` archivos.

Para tener en cuenta estas consideraciones relativas a los tramos, se utilizan dos *arrays,* `a[]` y `d[]`. El primero, contiene los números de tramos que ha de tener cada archivo de entrada en una pasada dada; el segundo, guarda el número de tramos ficticios que tiene cada archivo.

El proceso se inicia de *abajo a arriba;* por ejemplo, consideremos la Tabla 7.4, se empieza asignando al *array* `a[]` el numero de tramos correspondientes con la última mezcla, siempre `(1,1,... ,1)`. A la vez, al *array* de tramos ficticios `d[]` es también `(1,1,... ,1)`, de tal forma que cada vez que se copie un tramo, del archivo origen, en el archivo `i`, se decrementa `d[i]`, y así con cada uno de los `m-1` archivos.

Si no se ha terminado el archivo original, se determina de nuevo el número de tramos, pero del siguiente nivel, y los tramos que hay que añadir a cada archivo para alcanzar ese segundo nivel según las relaciones recurrentes. Los tramos ficticios de cada archivo, `d[i]`, coincidirán con el número de tramos que se deben añadir, `a[i]`, para que, posteriormente, según se vayan añadiendo tramos al archivo origen, se vaya decrementando `d[i]`.

Ejemplo 7.7

Se desea ordenar un archivo que dispone de 28 tramos, y se van a utilizar m = 4 archivos auxiliares. Encontrar la distribución inicial de tramos en los m-1 archivos.

La distribución se realiza consecutivamente, de nivel a nivel, según la Tabla 7.4. El primer nivel consta de tres tramos, `a(1,1,1)`, se distribuyen 3 tramos; se alcanza un segundo nivel con 5 tramos, `a(2,2,1)`, y como ya se distribuyeron 3 tramos se añaden 2 nuevos tramos, `(2,2,1)-(1,1,1) = (1,1,0)` y el *array* de tramos ficticios `d[]` se inicializa a los mismos valores, `(1,1,0)`. Una vez completados, los tramos para este segundo nivel, `d[]` se queda `a (0,0,0)`, se pasa al

siguiente nivel. Los tramos para alcanzar el tercer nivel son `9`, `a(4,3,2)`, entonces se debe añadir `(4,3,2)-(2,2,1) = (2,1,1)`, y se inician los tramos ficticios `d(2,1,1)`; se han distribuido `4` nuevos tramos.

Sucesivamente, se calcula el número de tramos de cada nuevo nivel y los tramos que hay que añadir para obtener esa cantidad; esos tramos serán, inicialmente, los tramos ficticios. A medida que se reparten tramos se decrementa el correspondiente elemento del *array* de tramos ficticios `d[]`. La Tabla 7.5 muestra los distintos valores que toman `a[]` y `d[]` en cada nivel, hasta completar los 28 tramos.

En el supuesto planteado, una vez completado el cuarto nivel se han repartido `17` tramos, y se pasa al siguiente; ahora `a = (13,11,7)`, los tramos incorporados son `(13,11,7)-(7,6,4) = = (6,5,3)`, y los tramos ficticios siempre se inicializan al número de tramos que se añaden, en este nivel `d(6,5,3)`. En el supuesto de 28 tramos, restan únicamente `28-17 = 11` tramos, que se distribuyen uniformemente en los archivos, y al finalizar la distribución el *array* de tramos ficticios queda `d = (2,1,0)` que se deben tener en cuenta, en los archivos correspondientes, durante el siguiente paso, que es la mezcla.

Cuantos más tramos haya más niveles se alcanzan. En cualquier caso, una vez terminado el proceso de distribución del archivo original, se tiene en `a[]`el número de tramos para ese nivel, y en `d[]` el número de tramos ficticios que tiene cada archivo, necesario para el proceso de mezcla.

Tabla 7.5 Distribución inicial con 28 tramos y m = 4 archivos

	Tramos iniciales	Tramos nivel, añadir			
		Número de nivel			
		2	3	4	5
a[]	(1,1,1)	(2,2,1)	(4,3,2)	(7,6,4)	**(13,11,7)**
añadir		(1,0,0)	(2,1,1)	(3,3,2)	(6,5,3)
a[]	(1,1,1)	(2,1,0)	(2,1,1)	(3,3,2)	(6,5,3)
	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	**(2,1,0)**

7.8.3. Algoritmo de la mezcla

Una vez realizada la distribución inicial en los `m-1` primeros archivos auxiliares, se repite el proceso de mezcla hasta que queda un único tramo. El número de pasadas es conocido de antemano, ha sido calculado durante la distribución, es el número de niveles alcanzado; también se conoce la distribución inicial de los tramos, que se encuentra en `a[]`, y el número inicial de tramos ficticios, `d[]`.

En cada nivel (se corresponde con cada pasada del algoritmo), el número de tramos que se mezclan es el mínimo de los elementos de `a[]`, siempre se encuentra en `a[m-1]`, siendo `m` el número de archivos. Al escribir tramos mezclados en el archivo de salida, si ocurre que todos

los tramos son ficticios, la posición que le corresponde al archivo de salida en el *array* de tramos ficticios, `d[]`, se incrementa en 1.

En el algoritmo de mezcla que a continuación se escribe, `c[]` es una tabla de correspondencia para acceder a los `m-1` archivos, de tal forma que en `c[]` se encuentran los índices de los archivos que se están mezclando. Como puede haber archivos no activos debido a los tramos ficticios, `cd[]` es la tabla de correspondencia con los índices de los archivos activos, `k` es el número de archivos activos y `actv[]`, *array* lógico indexado por `c[i]`, está a `true` si el archivo correspondiente está activo. Las rotaciones de los archivos se realizan al finalizar un archivo de entrada durante la mezcla de tramos, se hacen moviendo los elementos de la tabla `c[]`. La mezcla de tramos se realiza de igual forma que en la mezcla múltiple: una función de lectura de ítems de todos los archivos y la selección del mínimo.

Algoritmo

```
desde i <-1 hasta m-1 hacer
   c[i] <- i
fin _ desde

repetir

   { Mezcla de tramos de los archivos c[1]...c[m-1] }

   z <- a[m-1] {número de tramos a mezclar en esta pasada}
   d[m] <- 0 {inicializa tramos ficticios en archivo de salida}
   <Preparar para escribir F[c[m]]>
   repetir
        { Mezcla de un tramo de los m-1 archivos }
      k <- 0  { número de ficheros activos}
      desde i <- 1 hasta m-1 hacer
         si d[i] > 0 entonces {es un tramo ficticio}
            d[i] <- d[i]-1
         sino {es un tramo real}
            k <- k+1
            cd[k] <- c[i]
         fin _ si
      fin _ desde

      si k > 0 entonces
         <Mezclar los tramos de los k archivos>
      sino {Todos los tramos son ficticios}
         d[m] <- d[m]+1 {tramo ficticio en el archivo destino}
      fin _ si
      z <- z-1 {tramo ya mezclado}
      hasta z = 0
         { Se han mezclado todos los tramos, el archivo de salida pasa a
         ser de entrada }
      <Preparar para lectura F[c[m]]>
      <Rotar los archivos en la tabla de correspondencia c[]>
      <Calcular los números de fibonacci, a[i], del siguiente nivel>
      Nivel <- Nivel-1
   hasta Nivel = 0
Fin
```

7.7.4. Mezcla polifásica *versus* mezcla múltiple

Las principales diferencias de la ordenación polifásica respecto a la mezcla equilibrada múltiple son:

1. En cada pasada hay un sólo archivo destino (salida), en vez de los `m/2` que necesita la mezcla equilibrada múltiple.
2. La finalidad de los archivos *(entrada, salida)* cambia en cada pasada, rotando los índices de los archivos. Esto se controla mediante un tabla de correspondencia de índices de archivo. En la mezcla múltiple siempre se intercambian `m/2` archivos origen (entrada) por `m/2` archivos destino (salida).
3. El número de archivos origen (de entrada) varía dependiendo del número de tramo en proceso. Éste se determina en el momento de empezar el proceso de mezcla de un tramo, a partir del contador `di` de tramos ficticios para cada archivo `i`. Puede ocurrir que `di >0` para todos los valores de `i, i=1..m-1`, lo que significa que hay se mezclan `m-1` tramos ficticios, dando lugar a un tramo ficticio en el archivo destino, a la vez se incrementará el elemento `d[]` correspondiente al archivo destino (de salida). Normalmente, se mezclarán tramos reales de cada archivo de entrada (se cumple que `di == 0`).
4. Ahora, el criterio de terminación de una fase radica en el número de tramos a ser mezclados en cada archivo. Puede ocurrir que se alcance el último registro del archivo `m-1` y sean necesarias más mezclas que utilicen tramos ficticios de ese archivo. En la fase de distribución inicial fueron calculados los números de Fibonacci de cada nivel de forma progresiva, hasta alcanzar el nivel en el que se agotó el archivo de entrada; ahora, partiendo de los números del último nivel, pueden recalcularse hacia atrás.

RESUMEN

La ordenación de archivos se denomina ordenación externa porque los registros no se encuentran en arrays (memoria interna), sino en dispositivos de almacenamiento masivo, como son los cartuchos, cd, discos duros...; por lo que se requieren algoritmos apropiados. Una manera trivial de realizar la ordenación de un archivo secuencial consiste en copiar los registros a otro archivo de acceso directo, o bien secuencial indexado, usando como clave el campo por el que se desea ordenar.

Si se desea realizar la ordenación de archivos utilizando solamente como estructura de almacenamiento auxiliar otros archivos secuenciales de formato similar al que se desea ordenar, hay que trabajar usando el esquema de *separación* y *mezcla*.

En el caso del algoritmo de *mezcla simple,* se opera con tres archivos análogos: el original y dos archivos auxiliares. El proceso consiste en recorrer el archivo original y copiar secuencias de sucesivos registros, alternativamente, en cada uno de los archivos auxiliares. A continuación, se mezclan las secuencias de los archivos y se copia la secuencia resultante en el archivo original. El proceso continúa de tal forma que, en cada pasada, la longitud de la secuencia es el doble de la longitud de la pasada anterior. Todo empieza con secuencias de longitud 1 y termina cuando se alcanza una secuencia de longitud igual al número de registros.

El algoritmo de *mezcla natural* también opera con tres archivos, pero se diferencia de la mezcla directa en que distribuye secuencias ordenadas en vez de secuencias de longitud fija.

Se estudian métodos avanzados de ordenación externa, como la *mezcla equilibrada múltiple* y la *mezcla polifásica*. En el primero se utiliza un número par de archivos auxiliares, la mitad de ellos son archivos de entrada y la otra mitad archivos de salida. El segundo se caracteriza por hacer una distribución inicial de tramos (según las secuencias de números de Fibonacci), para después realizar una *mezcla continuada* hasta obtener un único tramo ordenado.

La primera parte del capítulo revisa la jerarquía de clases para procesar archivos. Java contiene un paquete especializado en la entrada y salida de datos, el paquete java.io. Contiene un conjunto de clases organizadas jerárquicamente para tratar cualquier tipo de flujo de entrada o de salida de datos. Es necesaria la sentencia `import java.io.*` en los programas que utilicen objetos de alguna de estas clases. Todo se basa en la *abstracción* de flujo: *corriente* de bytes que entran o que salen de un dispositivo. Para Java, un archivo, cualquier archivo, es un flujo de bytes, e incorpora clases que procesan los flujos a bajo nivel, como secuencias de bytes. Para estructurar los bytes y formar datos a más alto nivel hay clases de más alto nivel; con estas clases se pueden escribir o leer directamente datos de cualquier tipo simple (entero, char...). Estas clases se enlazan con las clases de bajo nivel que, a su vez, se asocian a los archivos.

EJERCICIOS

7.1. Escribir las sentencias necesarias para abrir un archivo de caracteres, cuyo nombre y acceso se introduce por teclado, en modo lectura; en el caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.

7.2. Un archivo contiene enteros positivos y negativos. Escribir un método para leer el archivo y determinar el número de enteros negativos.

7.3. Escribir un método para copiar un archivo. El método tendrá dos argumentos de tipo cadena, el primero es el archivo original y el segundo es el archivo destino. Utilizar flujos `FileInputStream` y `FileOutputStream`.

7.4. Una aplicación instancia objetos de las clases *NumeroComplejo* y *NumeroRacional*. La primera tiene dos variables instancia de tipo `float`, *parteReal* y *parteImaginaria*. La segunda clase tiene definidas tres variables *numerador* y *denominador,* de tipo `int`, y *frac,* de tipo `double`. Escribir la aplicación de tal forma que los objetos sean persistentes.

7.5. El archivo F, almacena registros con un campo clave de tipo entero. Suponer que la secuencia de claves que se encuentra en el archivo es la siguiente:

```
14 27 33 5 8 11 23 44 22 31 46 7 8 11 1 99 23 40 6 11 14 17
```

Aplicando el algoritmo de mezcla directa, realizar la ordenación del archivo y determinar el número de pasadas necesarias.

7.6. Considerando el mismo archivo que el del Ejercicio 7.5, aplicar el algoritmo de mezcla natural para ordenar el archivo. Comparar el número de pasadas con las obtenidas en el ejercicio anterior.

7.7. Un archivo secuencial `F` contiene registros y quiere ser ordenado utilizando 4 archivos auxiliares. Suponiendo que la ordenación se desea hacer respecto a un campo de tipo entero, con estos valores:

```
22 11 3 4 11 55 2 98 11 21 4 3 8 12 41 21 42 58 26 19 11 59 37 28 61 72 47
```

aplicar el algoritmo de mezcla equilibrada múltiple y obtener el número de pasadas necesarias para su ordenación.

7.8. Con el mismo archivo que el del Ejercicio 7.7, y también con `m = 4` archivos auxiliares, aplicar el algoritmo de mezcla polifásica. Comparar el número de pasadas realizadas en ambos métodos.

PROBLEMAS

7.1. Escribir un programa que compare dos archivos de texto (caracteres). El programa ha de mostrar las diferencias entre el primer archivo y el segundo, precedidas del número de línea y de columna.

7.2. Un atleta utiliza un pulsómetro para sus entrenamientos, que almacena sus pulsaciones cada 15 segundos, durante un tiempo máximo de 2 horas. Escribir un programa para almacenar en un archivo los datos del pulsómetro del atleta, de tal forma que el primer registro contenga la fecha, hora y tiempo en minutos de entrenamiento y, a continuación, los datos del pulsómetro por parejas: tiempo, pulsaciones.

7.3. Las pruebas de acceso a la Universidad Pontificia de Guadalajara, UPGA, constan de 4 apartados, cada uno de los cuales se puntúa de 1 a 25 puntos. Escribir un programa para almacenar en un archivo los resultados de las pruebas realizadas, de tal forma que se escriban objetos con los siguientes datos: nombre del alumno, puntuación de cada apartado y puntuación total.

7.4. Dado el archivo de puntuaciones generado en el Problema 7.3, escribir un programa para ordenar el archivo utilizando el método de ordenación externa *de mezcla natural*.

7.5. Supóngase que se dispone del archivo ordenado de puntuaciones de la UPGA (Problema 7.4) y del archivo de la Universidad de Salamanca que consta de objetos con los mismos datos y también está ordenado. Escribir un programa que mezcle ordenadamente los dos archivos en un tercero.

7.6. Se tiene guardado en un archivo a los habitantes de la comarca de Pinilla. Los datos de cada persona son los siguientes: primer apellido (campo clave), segundo apellido (campo secundario), edad, años de estancia y estado civil. Escribir un programa para ordenar el archivo por el método de mezcla equilibrada múltiple. Utilizar 6 archivos auxiliares.

7.7. Una farmacia desea mantener su *stock* de medicamentos en una archivo. De cada producto, interesa guardar el código, el precio y la descripción. Escribir un programa que genere el archivo pedido almacenando los objetos de manera secuencial.

7.8. Escribir un programa para ordenar el archivo que se ha generado en el Problema 7.7. Utilizar el método de ordenación de mezcla polifásica, con `m = 5` archivos auxiliares.

7.9. Implementar un método de ordenación externa, suponiendo un archivo de números enteros, con dos archivos auxiliares. La separación inicial del archivo en tramos sigue la siguiente estrategia: se leen 20 elementos del archivo en un *array* y se ordenan con el método de ordenación interna *Quicksort*. A continuación, se escribe el elemento menor del *array* en el archivo auxiliar y se lee el siguiente elemento del archivo origen. Si el elemento leído es mayor que el elemento escrito (forma parte del tramo actual), entonces se inserta en orden en el *subarray* ordenado; en caso contrario, se añade en las posiciones libres del *array*. Debido a que los elementos del *array* se extraen por la cabeza, quedan posiciones libres por su extremo inferior. El tramo termina en el momento en que el *subarray* ordenado queda vacío. Para formar el siguiente tramo, se empieza ordenando el *array* (recordar que los elementos que no formaban parte del tramo se iban añadiendo al *array*) y, después, el proceso continúa de la misma forma: escribir el elemento menor en otro archivo auxiliar y leer elemento del archivo origen... Una vez que se realiza la distribución la fase de mezcla es igual que en los algoritmos de mezcla directa o natural.