CAPÍTULO

8 Algoritmos de ordenación

La ordenación es una aplicación fundamental en computación. La mayoría de los datos producidos por un programa están ordenados de alguna manera, y muchos de los cómputos son eficientes porque invocan internamente a un método de ordenación. En consecuencia, la ordenación es muy probablemente la operación más importante y mejor estudiada en computación.

Este capítulo estudia el problema de ordenar un vector de elementos, describiendo y analizando varios algoritmos de ordenación. Las ordenaciones que estudiaremos en este capítulo pueden realizarse enteramente en memoria principal, en la medida en que el número de elementos a ordenar sea moderadamente pequeño (del orden de unos pocos millones). Son también importantes las ordenaciones que no pueden realizarse en memoria principal y deben ser realizadas en disco o cinta. Este tipo de ordenación, denominada *ordenación externa*, se estudia en la Sección 20.6.

La discusión sobre la ordenación que aquí se presenta es una mezcla entre teoría y práctica. Incluye varios algoritmos con distintos rendimientos y muestra cómo el análisis del rendimiento de un algoritmo puede llevarnos a concebir una implementación que diste bastante de ser obvia.

En este capítulo veremos:

- Que la ordenación por inserción, mostrada ya en la Figura 4.18, se ejecuta en tiempo cuadrático.
- Cómo implementar Shellsort (la ordenación de Shell), un método de ordenación simple y eficiente, con un tiempo de ejecución subcuadrático.
- Cómo escribir los algoritmos un poco más complicados de quicksort y mergesort, que tienen un tiempo de ejecución en $O(N \log N)$.
- Que un algoritmo de ordenación de propósito general requiere $\Omega(N \log N)$ comparaciones.

8.1 ¿Por qué es importante la ordenación?

En la Sección 5.6 vimos cómo buscar en un vector ordenado es más sencillo que buscar en uno que no lo esté. Esto no es sólo cierto para algoritmos a ser ejecutados por máquinas, sino también para el proceder cotidiano de las personas. Por

ejemplo, encontrar el nombre de alguien en el listín telefónico es fácil, mientras que buscar un número de teléfono sin conocer el nombre de la persona a la que corresponde es, en la práctica, una tarea casi imposible. Como consecuencia, mucha de la información devuelta por un cómputo se presenta de forma ordenada de modo que su manejo resulte mucho más sencillo. Algunos ejemplos son:

- Las palabras de un diccionario están ordenadas (ignorando la distinción entre mayúsculas y minúsculas).
- Los ficheros de un directorio generalmente se muestran ordenados.
- El índice de un libro está ordenado (ignorando la distinción entre mayúsculas y minúsculas).
- El catálogo de una biblioteca está ordenado por autor y por título.
- El listado de cursos ofrecidos por una universidad se ofrece ordenado, primero por departamentos y después por número de curso, por ejemplo.
- Muchos bancos ofrecen estados de cuentas donde los cheques están ordenados por su número.
- En un periódico, el calendario de eventos está ordenado por fecha.
- Los discos compactos musicales están ordenados en una tienda de discos por el nombre del artista.
- En los programas impresos para ceremonias de graduación, los departamentos se listan en orden y los estudiantes de cada departamento también se muestran ordenados.

Una ordenación de los datos inicial puede mejorar de forma significativa el rendimiento de un algoritmo.

No es sorprendente por tanto, que gran parte del trabajo realizado en computación involucre la ordenación. Además, hay también usos indirectos de la ordenación. Por ejemplo, suponga que quisiéramos averiguar si en un vector existen elementos duplicados. La Figura 8.1 muestra un método sencillo del cual se ve inmediatamente que en el caso peor requiere un tiempo cuadrático. La ordenación nos proporciona un método alternativo: si ordenamos una copia del vector, los elementos duplicados aparecerán juntos y podrán ser detectados mediante un recorrido secuencial del vector. El tiempo de este algoritmo está dominado por la ordenación, por lo que si podemos ordenar en un tiempo subcuadrático, tendremos un algoritmo mejor que el de partida. Hay muchos otros algoritmos cuyo tiempo de ejecución se ve mejorado de forma significativa ordenando los datos inicialmente.

Una gran mayoría de los proyectos de programación utilizan la ordenación en algún momento, y en muchos casos, el coste de la ordenación determina su coste global. Por tanto, queremos ser capaces de implementar la ordenación de forma rápida y segura.

```
1  // Devuelve true si el vector contiene duplicados; false,
      en otro caso
2
3  public static boolean duplicados( Object [ ] a )
4  {
5      for( int i = 0; i < a.length; i++ )
6          for( int j = i + 1; j < a.length; j++ )
7              if( a[ i ].equals( a[ j ] ) )
8                  return true;    // Encontrado un duplicado
9
10         return false;           // No hay duplicados
11 }
```

Figura 8.1 Algoritmo simple de coste cuadrático para detectar duplicados.

8.2 Preliminares

Los *algoritmos de ordenación por comparación* toman decisiones para lograr la ordenación sólo utilizando comparaciones.

Todos los algoritmos descritos en este capítulo son intercambiables. Todos reciben un vector que contiene los elementos, y sólo pueden ser ordenados objetos que implementen el interfaz `Comparable`. Por tanto, las únicas operaciones disponibles para obtener información sobre el orden relativo de los elementos son `menorQue` y `compara`. Los algoritmos que ordenan bajo estas condiciones se denominan *algoritmos de ordenación por comparación*. Todos los algoritmos de ordenación que veremos son métodos estáticos de la clase `Ordenacion` (en el paquete `EstructurasDatos`). En este capítulo, N es el número de elementos a ordenar.

8.3 Análisis de la ordenación por inserción y otras ordenaciones simples

La ordenación por inserción es cuadrática en el peor caso y en promedio. Es más rápida si los datos ya están ordenados.

El método de ordenación más simple, denominado *ordenación por inserción*, se discutió en la Sección 4.3.1. La implementación en la Figura 8.2 repite la presentada en la Figura 4.18. Debido a los bucles anidados, cada uno de los cuales puede realizar N iteraciones, el algoritmo de ordenación por inserción es $O(N^2)$. Es más, esta cota es alcanzable, ya que un vector de entrada ordenador en orden inverso requiere en efecto un tiempo cuadrático. Un cálculo preciso muestra que el test de la línea 10 puede ejecutarse, como mucho, $P+1$ veces, por cada valor de P. Sumando sobre todos los posibles P, obtenemos un total de

$$\Sigma_{P=1}^{N-1} (P + 1) = 2 + 3 + 4 + \cdots + N = \Theta(N^2).$$

Por otro lado, si el vector de entrada ya viene ordenado, el tiempo de ejecución es $O(N)$ ya que el test del bucle `for` más interno falla inmediatamente. De hecho, si el vector de entrada está casi ordenado (el término *casi ordenado* será definido formalmente en breve), la ordenación por inserción se ejecutará rápidamente. Esto significa que el tiempo de ejecución no depende exclusivamente de la cantidad de

```
1  // ordenacionPorInsercion: ordena el vector a
2
3  public static void ordenacionPorInsercion( Comparable [ ] a )
4  {
5      for( int p = 1; p < a.length; p++ )
6      {
7          Comparable tmp = a[ p ];
8          int j = p;
9
10         for( ; j > 0 && tmp.menorQue( a[ j - 1 ] ); j-- )
11             a[ j ] = a[ j - 1 ];
12         a[ j ] = tmp;
13     }
14 }
```

Figura 8.2 Ordenación genérica por inserción.

datos, sino también del orden específico de estos datos. Debido a esta gran variación, merece la pena analizar el comportamiento en el caso medio. Resulta que el tiempo medio de la ordenación por inserción, al igual que el de otros algoritmos simples de ordenación, es $\Theta(N^2)$.

Las inversiones miden el grado de desorden.

Una *inversión* en un vector de números es todo par (i, j) con $i < j$ y $A_i > A_j$. Por ejemplo, la secuencia {8, 5, 9, 2, 6, 3} contiene diez inversiones que corresponden a los pares (8, 5), (8, 2), (8, 6), (8, 3), (5, 2), (5, 3), (9, 2), (9, 6), (9, 3) y (6, 3). Puede observarse que el número de inversiones del vector de partida es igual al número de veces que se ejecuta la línea 11 de la Figura 8.2 al ordenarlo. Esto es así pues el efecto de la instrucción de asignación es intercambiar los elementos `a[j]` y `a[j-1]`. (En realidad, evitamos tantos intercambios utilizando una variable auxiliar, pero aún así se trata de un intercambio abstracto.) Intercambiar dos elementos que no están bien colocados elimina exactamente una inversión, y un vector ordenado no contiene ninguna inversión. Por tanto, si al principio del algoritmo tenemos I inversiones, debemos realizar I intercambios implícitos. Ya que el resto del trabajo realizado por el algoritmo es $O(N)$, el tiempo de ejecución de la ordenación por inserción es $O(I + N)$, donde I es el número de inversiones en el vector original. Luego el algoritmo de ordenación por inserción se ejecuta en tiempo lineal si el número de inversiones es $O(N)$.

Podemos calcular cotas precisas del tiempo de ejecución en el caso medio del algoritmo de ordenación por inserción, calculando el número medio de inversiones en el vector. Definir correctamente lo que significa «en el caso medio» no es fácil. Asumiremos que no hay elementos repetidos (si permitimos repeticiones, no está claro lo que significaría el número medio de repeticiones). Entonces podemos suponer, sin perdida de generalidad, que los datos de entrada son una cierta reorganización de los N primeros números naturales (ya que sólo el orden relativo entre ellos importa); estas reorganizaciones se denominan *permutaciones*. Podemos razonablemente asumir que todas estas permutaciones son equiprobables. Bajo estas hipótesis, podemos establecer el Teorema 8.1.

Teorema 8.1 *El número medio de inversiones de un vector de N números distintos es* $N(N-1)/4$.

Demostración *Para cada vector de números A, denotemos por* A_r *el vector en orden inverso. Por ejemplo, el inverso del vector (1, 5, 4, 2, 6, 3) es (3, 6, 2, 4, 5, 1). Consideremos cualquier par de números* (x, y) *del vector, con* $y > x$. *Exactamente en uno de los vectores A o* A_r, *este par representa una inversión. El número total de pares en A y* A_r *es* $N(N-1)/2$. *Por tanto, en promedio tendremos la mitad de este número de inversiones, es decir,* $N(N-1)/4$.

Del Teorema 8.1 se deduce que la ordenación por inserción es cuadrática en promedio. El resultado también puede ser utilizado para proporcionar una cota inferior muy fuerte del coste de todo algoritmo de ordenación que sólo intercambie elementos adyacentes. Esto se expresa en el Teorema 8.2.

Teorema 8.2

Todo algoritmo que ordene mediante intercambio de elementos adyacentes requiere, en promedio, un tiempo $\Omega(N^2)$.

Demostración

El número medio de inversiones del vector de partida es $N(N-1)/4$. *Cada intercambio elimina exactamente una inversión, por lo que el número de intercambios necesarios es* $\Omega(N^2)$.

La *demostración de una cota inferior* muestra que el rendimiento cuadrático es inherente a cualquier algoritmo que ordene por medio de comparaciones de elementos adyacentes.

Éste es un ejemplo de *cálculo de una cota inferior*. Es válida no sólo para la ordenación por inserción, que realiza implícitamente intercambios de elementos adyacentes, sino también para otros algoritmos simples (no descritos aquí), como la ordenación por el método de la burbuja o la ordenación por selección. Más en general, es válida para toda una *clase* de algoritmos, incluyendo algunos que no se habrán descubierto aún: todos aquéllos que realizan sólo intercambios de elementos adyacentes.

Observe que la obtención experimental de resultados válidos para toda una clase de algoritmos requerirá, en general, la ejecución de todos los algoritmos de la clase. Esto no es posible en cuanto el número de posibles algoritmos sea infinito. De modo que cualquier resultado obtenido experimentalmente sólo se aplicaría a aquellos algoritmos comprobados. Esto hace que la confirmación experimental de la validez de una hipotética cota inferior sea mucho más difícil que la confirmación de la validez de una cota superior de un único algoritmo, a la que estamos acostumbrados. Un cómputo sólo podría refutar una conjetura sobre una cota inferior; pero, en general, nunca podría probarla.

Aunque la demostración de la cota inferior arriba obtenida es bastante simple, en general, probar una cota inferior es bastante más complicado que probar una cota superior. Los argumentos para probar una cota inferior son bastante más abstractos que sus contrapartidas para las cotas superiores.

La cota inferior obtenida nos indica que para que un algoritmo de ordenación sea subcuadrático, esto es, esté en $O(N^2)$, debe hacer comparaciones y, en particular, intercambiar elementos, que estén alejados entre sí. Un algoritmo de ordenación progresa eliminando inversiones. Para que se ejecute eficientemente, debe eliminar más de una inversión por intercambio.

8.4 Shellsort

Shellsort es un algoritmo subcuadrático que funciona bien en la práctica y es fácil de implementar. El rendimiento de Shellsort depende en gran medida de la secuencia de incrementos en que se base.

El primer algoritmo que mejoró de forma sustancial la ordenación por inserción fue *Shellsort*. El algoritmo de ordenación de Shell, Shellsort, fue desarrollado en 1959 por Donald Shell. Aunque hoy en día no es el algoritmo conocido más rápido, sólo es algo más largo y complejo que la ordenación por inserción, lo que le hace el más simple de los algoritmos más rápidos.

La idea de Shell fue evitar gran cantidad de movimientos de datos comparando primero elementos que estuvieran muy separados, para después comparar elementos algo más cercanos, y así sucesivamente, reduciéndose gradualmente al método de inserción. Shellsort utiliza una secuencia $h_1, h_2, \ldots, h_t$, denominada *secuencia de incrementos*. Cualquier secuencia de incrementos es válida, con tal de que $h_1 = 1$, pero algunas elecciones son mejores que otras. Después de una fase utilizando un cierto incremento h_k, tendremos que $a[i] \leq a[i + h_k]$ para todo i para el

que dicha comparación tenga sentido. Esto es, todos los elementos separados por una distancia h_k están ordenados. Se dice entonces que el vector está *h_k-ordenado*.

Por ejemplo, la Figura 8.3 muestra un vector después de la aplicación de varias fases del algoritmo Shellsort. Después de una 5-ordenación, se garantiza que los elementos a distancia 5 están correctamente ordenados. En la figura, los distintos elementos a distancia 5 están sombreados de igual forma. Como puede verse, todos están ordenados entre sí. De igual forma, después de la 3-ordenación, se garantiza que los elementos a distancia 3 están correctamente ordenados entre sí. Una propiedad importante de Shellsort (que afirmaremos sin demostración) es que un vector h_k-ordenado sigue estándolo después de una h_{k-1}-ordenación. Si no fuera así, el algoritmo sería de poca utilidad ya que el trabajo realizado por las primeras fases sería deshecho por las siguientes.

Shellsort se denomina también *ordenación por disminución de intervalos.*

En general, una h_k-ordenación requiere que para cada posición i en h_k, h_{k+1}, ..., $N-1$, se coloque el elemento en el sitio adecuado de entre las i, $i - h_k$, $i - 2h_k$ y así sucesivamente. Aunque ello no afecta a la implementación, un examen cuidadoso muestra que el efecto de una h_k-ordenación es realizar una ordenación por inserción en h_k subvectores independientes (mostrados con diferentes sombreados en la Figura 8.3). Por consiguiente, no debería sorprender que en la Figura 8.4, las líneas de la 9 a la 19 representen una *ordenación por inserción por intervalos*. En una ordenación por inserción por intervalos, después de ejecutar el bucle, los elementos del vector separados por una distancia de `intervalo` están ordenados. Por ejemplo, cuando el `intervalo` es 1, el bucle es idéntico, instrucción a instrucción, a la ordenación por inserción. En consecuencia, Shellsort también se conoce como *ordenación por disminución de intervalos*.

Es fácil mostrar varios hechos. Primero, cuando `intervalo` es 1, se garantiza que el bucle más interno ordena el vector `a`. Si `intervalo` nunca se hiciera 1, tendríamos algún vector de entrada que no podría ser ordenado correctamente. Por tanto, Shellsort ordena hasta que `intervalo` llega a valer 1, y tras ello podemos parar. La única cuestión pendiente es cómo elegir la secuencia de incrementos.

La secuencia de incrementos sugerida por Shell representa una mejora sobre el algoritmo de ordenación por inserción (aunque hoy en día se conocen mejores secuencias).

Shell sugirió empezar con `intervalo` igual a $N/2$, e ir dividiéndolo por la mitad hasta llegar a 1, momento en el que el algoritmo termina. Utilizando estos incrementos, Shellsort representa una mejora sustancial con respecto a la ordenación por inserción, a pesar de tener tres bucles `for` anidados en vez de dos, lo que habitualmente sería ineficiente. Alterando la secuencia de incrementos, uno puede seguir mejorando el rendimiento del algoritmo. En la Figura 8.5 se muestra una tabla de rendimientos de Shellsort con tres elecciones diferentes de secuencias de incrementos.

Original	**81**	**94**	**11**	**96**	**12**	**35**	**17**	**95**	**28**	**58**	**41**	**75**	**15**
Después de la 5-ordenación	35	17	11	28	12	41	75	15	96	58	81	94	95
Después de la 3-ordenación	28	12	11	35	15	41	58	17	94	75	81	96	95
Después de la 1-ordenación	11	12	15	17	28	35	41	58	75	81	94	95	96

Figura 8.3 Shellsort después de cada paso, cuando la secuencia de incrementos es (1, 3, 5).

```
1  /**
2   * Shellsort, utilizando la secuencia sugerida por Gonnet.
3   * @param a un vector de elementos Comparable.
4   */
5  public static void shellsort( Comparable [ ] a )
6  {
7      for( int intervalo = a.length / 2; intervalo > 0;
8           intervalo = intervalo == 2 ? 1 : (int) ( intervalo / 2.2 ) )
9           for( int i = intervalo; i < a.length; i++ )
10          {
11              Comparable temp = a[ i ];
12              int j = i;
13
14              for( ; j >= intervalo &&
15                     temp.menorQue( a[ j - intervalo ] );
16                                                  j -= intervalo )
17                  a[ j ] = a[ j - intervalo ];
18              a[ j ] = temp;
19          }
20 }
```

Figura 8.4 Implementación de Shellsort que divide por 2,2 en vez de por 2.

8.4.1 Rendimiento de Shellsort

El tiempo de ejecución de Shellsort depende en gran medida de la secuencia de incrementos elegida. Las correspondientes demostraciones de los resultados precisos en cada caso pueden resultar bastante complicadas. El análisis en el caso medio de Shellsort es un problema abierto desde hace mucho tiempo, excepto para las secuencias de incrementos triviales.

N	Ordenación por inserción	Shellsort		
		Secuencia de Shell	Sólo intervalos impares	Dividiendo por 2,2
1.000	122	11	11	9
2.000	483	26	21	23
4.000	1.936	61	59	54
8.000	7.950	153	141	114
16.000	32.560	358	322	269
32.000	131.911	869	752	575
64.000	520.000	2.091	1.705	1.249

Figura 8.5 Tiempo de ejecución (en milisegundos) de la ordenación por inserción y de Shellsort con varias secuencias de incrementos.

En el caso peor, la secuencia de incrementos de Shell genera un comportamiento cuadrático.

Cuando se utilizan los incrementos de Shell, se puede probar que el caso peor es $O(N^2)$. Esta cota se alcanza cuando N es una potencia de 2, todos los elementos grandes están en posiciones pares del vector y todos los pequeños en posiciones impares. Cuando se alcanza la última iteración, todos los elementos grandes estarán todavía en las posiciones pares, y todos los pequeños en posiciones impares. Un cálculo del número de inversiones restante muestra que la última iteración requerirá un tiempo cuadrático. La afirmación de que éste es el peor caso se sigue del hecho de que una h_k-ordenación consiste en h_k ordenaciones por inserción de, aproximadamente, N/h_k elementos. Por consiguiente, el coste de cada iteración es $O(h_k(N/h_k)^2)$, o sea $O(N^2/h_k)$. Al sumar sobre todas las iteraciones, obtenemos $O(N^2\Sigma h_k)$. Ya que los incrementos generan, aproximadamente, una serie geométrica, la suma está acotada por una constante. El resultado es que el tiempo de ejecución es cuadrático en el caso peor. También se puede probar, a través de un argumento complejo, que cuando N es una potencia de 2, el tiempo medio de ejecución es $O(N^{3/2})$. Por tanto, en promedio, la secuencia de incrementos de Shell representa una mejora significativa sobre la ordenación por inserción.

Si los incrementos consecutivos son primos entre sí, el rendimiento de Shellsort mejora.

Un cambio mínimo en la secuencia de incrementos puede evitar el caso peor cuadrático. Si al dividir `intervalo` por 2 el resultado es par, entonces le sumaremos 1 para hacerlo impar. En este caso se puede probar que el caso peor ya no es cuadrático, sino $O(N^{3/2})$. Aunque la demostración es complicada, se basa en que en esta nueva secuencia de incrementos, los incrementos consecutivos no comparten ningún factor común (mientras que para la secuencia de incrementos de Shell esto sí ocurría). Cualquier secuencia que cumpla esta propiedad (y cuyos incrementos decrezcan, aproximadamente, de forma geométrica) tendrá un tiempo de ejecución en el caso peor $O(N^{3/2})$[1]. Se desconoce el rendimiento en promedio del algoritmo con estos nuevos incrementos, pero basándose en simulaciones parece ser $O(N^{5/4})$.

Dividir por 2,2 da un rendimiento excelente en la práctica.

Una tercera secuencia, que se comporta bien en la práctica, aunque se desconozca el fundamento teórico de ello, consiste en dividir por 2,2 en vez de por 2. Parece que con ello conseguimos un tiempo de ejecución en promedio por debajo de $O(N^{5/4})$ (quizás $O(N^{7/6})$), aunque esto está todavía por precisar. Con un número de elementos entre 100.000 y 1.000.000, típicamente se mejora el rendimiento entre un 25 y un 35 por ciento, aunque nadie sabe muy bien por qué. Una implementación de Shellsort con esta secuencia de incrementos se presenta en la Figura 8.4. El código complicado de la línea 8 es para evitar que `intervalo` se pueda hacer igual a 0. Si tal cosa ocurriera, el algoritmo no funcionaría correctamente, pues nunca se haría una 1-ordenación. La línea 8 asegura que cuando `intervalo` vale 2, pasa a valer 1, y no 0.

La tabla de la Figura 8.5 compara el rendimiento de la ordenación por inserción y el de Shellsort, con varias secuencias de intervalos. Estos resultados se obtuvieron en una máquina razonablemente rápida. El test claramente está predispuesto en contra de la secuencia de intervalos original, pues N se elige 125 veces una potencia de 2. Así el redondear hacia arriba hasta un número impar es particularmente beneficioso, especialmente cuando N se hace grande. Podríamos concluir fácilmente que Shellsort, incluso con la secuencia de intervalos más simple, proporciona una mejora significativa sobre la ordenación por inserción, con un coste

[1] Para apreciar la sutileza involucrada, observe que restar 1, en vez de sumar 1, no funciona. Por ejemplo, si N es 186, la secuencia resultante será 93, 45, 21, 9, 3, 1, en la que todos los incrementos comparten 3 como factor común.

adicional mínimo en lo que se refiere a la complejidad lógica del algoritmo. Un simple cambio en la secuencia de intervalos puede mejorar todavía más el rendimiento. Se pueden conseguir mejoras aún mayores (como se verá en el Ejercicio 8.18). Algunas de estas mejoras tienen un fundamento teórico, pero ninguna secuencia conocida mejora apreciablemente la mostrada en la Figura 8.4.

El rendimiento de Shellsort es bastante aceptable en la práctica, aun con valores de N de decenas de miles. La simplicidad del código hace que sea el algoritmo a elegir para ordenar entradas de tamaño moderadamente grande. Es también un buen ejemplo de algoritmo muy simple con un análisis extremadamente complejo.

Shellsort es una buena elección para entradas de tamaño moderadamente grande.

8.5 Mergesort

En la Sección 7.5 vimos cómo la recursión puede utilizarse para desarrollar algoritmos subcuadráticos. En concreto, un algoritmo divide y vencerás en el cual se resuelven recursivamente dos problemas con la mitad de tamaño, con una sobrecarga $O(N)$, es un algoritmo $O(N \log N)$. Mergesort es uno de estos algoritmos, ofreciendo una mejor cota, al menos a nivel teórico, que las afirmadas para Shellsort.

Mergesort utiliza divide y vencerás para obtener un tiempo de ejecución $O(N \log N)$.

El algoritmo mergesort consta de tres pasos:

1. Si el número de elementos a ordenar es 0 o 1, acabar.
2. Ordenar recursivamente las dos mitades del vector.
3. Mezclar las dos mitades ordenadas en un vector ordenado.

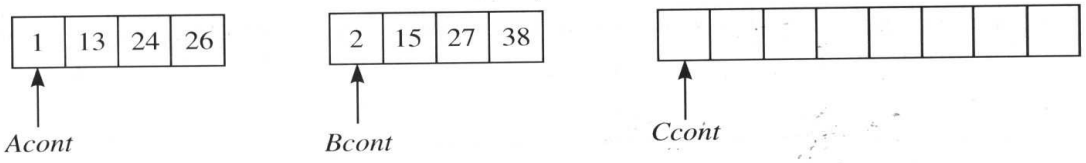
Para afirmar que el algoritmo es $O(N \log N)$, es suficiente demostrar que la mezcla ordenada de dos vectores ordenados puede realizarse en tiempo lineal. En la siguiente subsección veremos cómo mezclar dos vectores A y B, colocando el resultado en un tercer vector C. Después mostraremos una implementación sencilla de mergesort. La rutina de mezcla es la piedra angular de muchos algoritmos de ordenación externa. Esto se ilustrará en la Sección 20.6.

La mezcla ordenada de vectores ordenados puede realizarse en tiempo lineal.

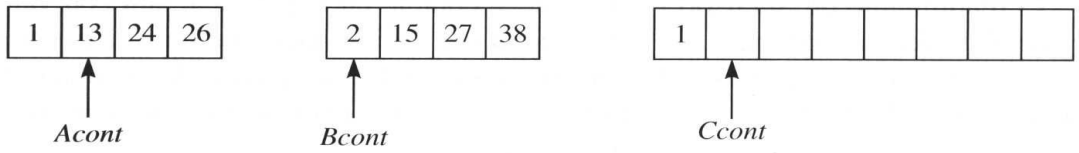
8.5.1 Mezcla lineal de vectores ordenados

El algoritmo de mezcla básico parte de dos vectores de entrada A y B, produciendo un vector de salida C, apoyándose en tres contadores *Acont*, *Bcont* y *Ccont*, inicializados al principio de los vectores respectivos. En cada paso el menor de los valores $A[Acont]$ y $B[Bcont]$ se copia en la próxima posición de C, actualizándose los contadores apropiados. Cuando uno de los vectores de entrada se acaba, el resto del otro vector se copia en C.

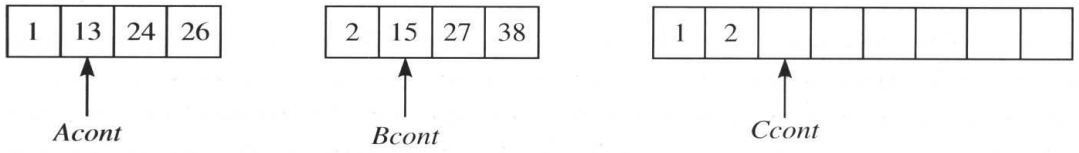
Veamos cómo funciona la rutina de mezcla sobre la siguiente entrada:



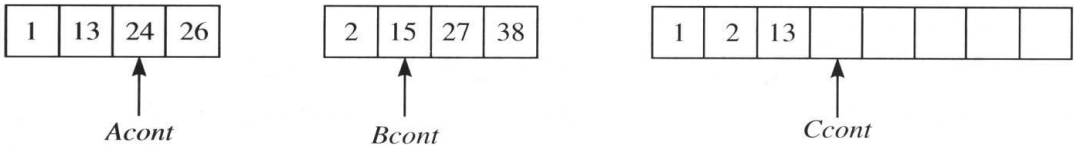
Cuando el vector *A* contiene los elementos 1, 13, 24 y 26, y *B* contiene los elementos 2, 15, 27, 38, el algoritmo se comporta de la siguiente manera: en primer lugar, se comparan los elementos 1 y 2, con lo que 1 se copia en *C*; tras ello se comparan los elementos 13 y 2:



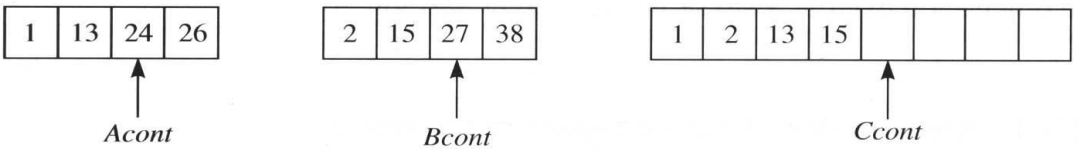
El 2 se añade a *C*, y se pasan a comparar los elementos 13 y 15:

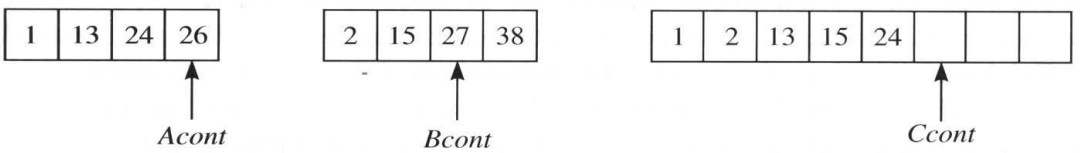


El 13 se añade a *C*, y se comparan los elementos 24 y 15:

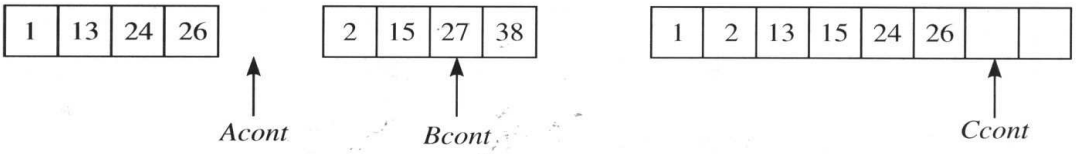


El proceso continúa comparando los elementos 26 y 27:

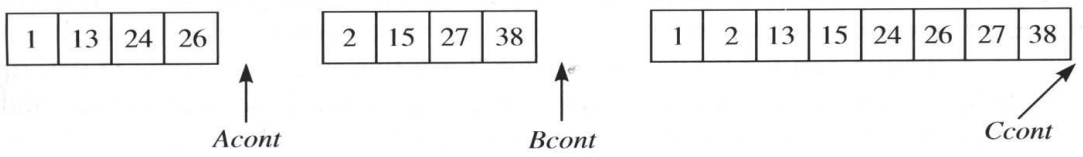




El elemento 26 se añade a *C*, con lo que el vector *A* se acaba:



Finalmente, el resto del vector *B* se copia en *C*:



El tiempo necesario para mezclar dos vectores ordenados es lineal pues cada comparación incrementa *Ccont* (limitando, por tanto, el número de comparaciones). Como resultado, un algoritmo divide y vencerás que utilice un procedimiento de mezcla lineal se ejecutará en el caso peor en un tiempo $O(N \log N)$. Éste también será el tiempo de ejecución en el caso promedio, e incluso en el caso mejor, ya que la mezcla es siempre lineal.

Como ejemplo de aplicación del algoritmo mergesort ordenemos el vector de 8 elementos (24, 13, 26, 1, 2, 27, 38, 15). Después de ordenar, recursivamente, los 4 primeros elementos y los 4 últimos, obtenemos (1, 13, 24, 26, 2, 15, 27, 38). Finalmente se mezclan las dos mitades, obteniendo el vector final (1, 2, 13, 15, 24, 26, 27, 38).

```
1  /**
2   * Método interno que realiza las llamadas recursivas.
3   * @param a un vector de elementos de tipo Comparable.
4   * @param vectorTemp un vector para colocar el resultado de la
       mezcla.
5   * @param izq el índice más a la izquierda del subvector.
6   * @param der el índice más a la derecha del subvector.
7   */
8  private static void mergeSort( Comparable [ ] a,
9              Comparable [ ] vectorTemp, int izq, int der )
10 {
11     if( izq < der )
12     {
13         int centro = ( izq + der ) / 2;
14         mergeSort( a, vectorTemp, izq, centro );
15         mergeSort( a, vectorTemp, centro + 1, der );
16         mezclar( a, vectorTemp, izq, centro + 1, der );
17     }
18 }
19
20 /**
21  * Algoritmo mergesort.
22  * @param a un vector de elementos de tipo Comparable.
23  */
24 public static void mergeSort( Comparable [ ] a )
25 {
26     Comparable [ ] vectorTemp = new Comparable[ a.length ];
27
28     mergeSort( a, vectorTemp, 0, a.length - 1 );
29 }
```

Figura 8.6 Rutinas básicas de `mergeSort`.

8.5.2 El algoritmo mergesort

Mergesort utiliza un espacio adicional lineal, lo cual representa una cierta desventaja en la práctica.

La Figura 8.6 muestra una implementación muy sencilla de mergesort. El `mergeSort` público es un simple envoltorio que declara un vector temporal y llama al `mergeSort` recursivo con los límites del vector. La rutina `mezclar` sigue la descripción dada en la sección anterior. Utiliza la primera mitad del vector (indexado desde `izq` a `centro`) como A, la segunda mitad (indexada desde `centro + 1` hasta `der`) como B y el vector auxiliar como C. La Figura 8.7 muestra la rutina `mezclar`. Por último, el vector auxiliar se copia en el vector original.

Podemos evitar copias innecesarias con más trabajo, pero el espacio lineal extra no puede ser eliminado sin un incremento excesivo del tiempo de ejecución.

Aunque el tiempo de ejecución de mergesort es $O(N \log N)$, rara vez se utiliza para ordenaciones en memoria. Esto es debido a que la mezcla de dos subvectores ordenados utiliza un espacio adicional lineal, y el trabajo adicional debido a las copias hacia y desde el vector auxiliar al original, ralentizan considerablemente la ordenación. Las copias se pueden evitar intercambiando alternativamente los papeles de `a` y `vectorAux` en las llamadas recursivas.

```
1  /**
2   * Método interno que mezcla dos mitades ordenadas de un
      subvector.
3   * @param a un vector de elementos de tipo Comparable.
4   * @param vectorTemp un vector para colocar el resultado de la
      mezcla.
5   * @param posIzq el índice más a la izquierda del vector.
6   * @param porDer el índice de comienzo de la segunda mitad.
7   * @param posFin el índice más a la derecha del vector.
8   */
9  private static void mezclar( Comparable [ ] a,
10                              Comparable [ ] vectorAux, int posIzq,
11                              int posDer, int posFin )
12 {
13     int finIzq = posDer - 1;
14     int posAux = posIzq;
15     int numElementos = posFin - posIzq + 1;
16
17     // Bucle principal
18     while( posIzq <= finIzq && posDer <= posFin )
19         if( a[ posIzq ].menorQue( a[ posDer ] ) )
20             vectorAux[ posAux++ ] = a[ posIzq++ ];
21         else
22             vectorAux[ posAux++ ] = a[ posDer++ ];
23
24     // Copiar el resto de la primera mitad
25     while( posIzq <= finIzq )
26         vectorAux[ posAux++ ] = a[ posIzq++ ];
27
28     // Copiar resto de la segunda mitad
29     while( posDer <= posFin )
30         vectorAux[ posAux++ ] = a[ posDer++ ];
31
32     // Copiar el vector temporal en el original
33     for( int i = 0; i < numElementos; i++, posFin-- )
34         a[ posFin ] = vectorAux[ posFin ];
35 }
```

Figura 8.7 Rutina `mezclar`.

Una variación de mergesort se puede implementar de forma no recursiva. De cualquier forma, para aplicaciones serias de ordenación interna el algoritmo utilizado suele ser quicksort, descrito en la próxima sección.

8.6 Quicksort

Quicksort[2] es el algoritmo de ordenación más rápido conocido. Su tiempo de ejecución promedio es $O(N \log N)$. Su rapidez se debe principalmente a un bucle interno muy ajustado y altamente optimizado. Tiene un rendimiento cuadrático en el caso peor, pero este caso puede hacerse estadísticamente improbable con poco esfuerzo. El algoritmo quicksort es relativamente fácil de entender, así como de demostrar su corrección, ya que se basa en la recursión. Sin embargo su implementación requiere atención, pues cambios mínimos en el código pueden repercutir considerablemente en el tiempo de ejecución. Esta sección comienza describiendo el algoritmo en un sentido amplio, para proporcionar después un análisis que muestra sus tiempos de ejecución en el caso peor, en promedio, y en el caso mejor. Utilizaremos este análisis para fijar ciertas decisiones sobre la implementación en Java, atendiendo a ciertos detalles, tales como el tratamiento de elementos duplicados.

Quicksort es un algoritmo divide y vencerás rápido, cuando se implementa adecuadamente. En la práctica es el algoritmo más rápido de ordenación basado en comparaciones.

8.6.1 El algoritmo quicksort

El algoritmo básico *Quicksort*(S) es recursivo, consistiendo en los siguientes cuatro pasos:

1. Si el número de elementos de S es 0 o 1, terminar.
2. Escoger un elemento *cualquiera* v de S. Este elemento se denominará *pivote*.
3. Hacer una *partición* de $S - \{v\}$ (el resto de elementos de S) en dos grupos disjuntos: $I = \{x \in S - \{v\} \mid x \leqslant v\}$ y $D = \{x \in S - \{v\} \mid x \geqslant v\}$.
4. Devolver el resultado de *Quicksort*(I), seguido de v y seguido del resultado de *Quicksort*(D).

El algoritmo quicksort básico es recursivo. Hay que elegir el pivote, decidir cómo hacer la partición y tener en cuenta los elementos repetidos.

Hay que puntualizar una serie de cuestiones. En primer lugar, el caso básico de la recursión incluye la posibilidad de que S sea vacío. Esto es necesario, ya que las llamadas recursivas pueden generar subconjuntos vacíos. Segundo, el algoritmo permite que cualquier elemento sea usado como *pivote*. El pivote divide a los elementos del vector en dos grupos: los menores y los mayores que él. El análisis que realizaremos mostrará que algunas elecciones del pivote son mejores que otras. Por tanto, al proporcionar una implementación concreta, no utilizaremos un pivote cualquiera; trataremos de hacer una buena elección al respecto.

El *pivote* divide a los elementos del vector en dos grupos: los menores y los mayores que él.

En el paso de *partición*, cada elemento de S, excepto el pivote, se coloca bien en I, que representa la parte izquierda del vector, o en D, que representa la parte derecha del vector. El objetivo es que los elementos menores que el pivote vayan a I, mientras que los elementos mayores van a D. Sin embargo, la descripción del

El paso de *partición* coloca cada elemento, excepto el pivote, en uno de dos posibles grupos.

[2] N. del T.: Ordenación rápida, en inglés.

algoritmo no puntualiza qué hacer con los hipotéticos elementos iguales al pivote. Se permite que en cada caso un elemento tal vaya a cualquiera de los subconjuntos, especificándose únicamente que debe ir a uno sólo. Parte de una buena implementación en Java residirá en tratar este caso tan eficientemente como sea posible. Una vez más, el análisis nos permitirá tomar una decisión bien fundamentada.

La Figura 8.8 muestra el efecto de quicksort sobre un conjunto de números. Se toma como pivote (sin que haya ninguna razón especial para hacerlo así) el número 65. El resto de elementos en el conjunto se separan en dos subconjuntos de tamaño menor. Cada unos de estos subconjuntos se ordena recursivamente. Aplicando la tercera regla de la recursión, podemos suponer que este paso funciona correctamente. Por último, se obtiene de forma inmediata el resultado ordenado del conjunto original. En una implementación en Java, los elementos a ordenar en

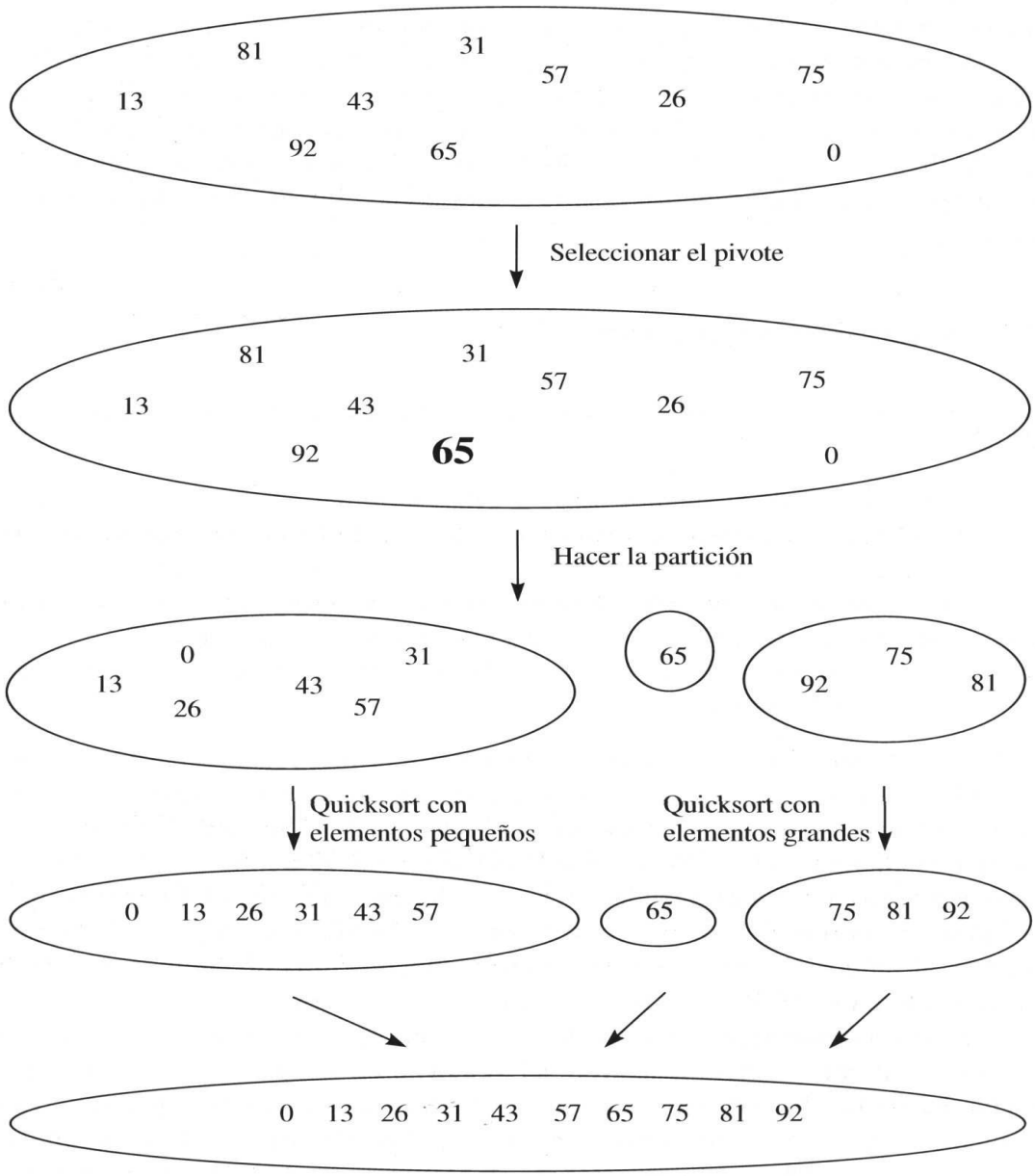


Figura 8.8 Los pasos de quicksort.

cada paso se almacenarían en una parte del vector delimitada por `ini` y `fin`. Después del paso de partición, el pivote terminaría en alguna posición `p` del vector y las llamadas recursivas se harían con los subvectores desde `ini` hasta `p-1` y desde `p+1` hasta `fin`.

La corrección del algoritmo se basa en los siguientes hechos:

- El subconjunto de elementos pequeños se ordena correctamente, en virtud de la recursión.
- El mayor elemento en el subconjunto de elementos pequeños no es mayor que el pivote, dado cómo se hace la partición.
- El pivote no es mayor que el menor elemento en el subconjunto de elementos grandes, dado cómo se hace la partición.
- El subconjunto de elementos grandes se ordena correctamente, en virtud de la recursión.

Quicksort es rápido porque la partición puede realizarse rápidamente y en el mismo sitio.

Aunque la corrección del algoritmo se establece fácilmente, no está claro por qué es más rápido que mergesort. Igual que mergesort, resuelve de manera recursiva dos subproblemas y requiere un cierto trabajo adicional (en el paso de partición). Además, al contrario de mergesort, no se garantiza que el tamaño de los subproblemas sea el mismo. Esto no es bueno para el rendimiento. Quicksort es más rápido que mergesort porque el paso de partición puede hacerse significativamente más deprisa que el paso de mezcla. En concreto, puede implementarse sin utilizar un vector auxiliar, y el código es muy compacto y eficiente. Esto compensa la no generación sistemática de subproblemas del mismo tamaño.

8.6.2 Análisis de quicksort

La descripción del algoritmo deja varias cuestiones sin contestar. ¿Cómo elegimos el pivote? ¿Cómo realizamos la partición? ¿Qué hacemos si encontramos un elemento igual al pivote? Todas estas cuestiones pueden afectar en gran medida al tiempo de ejecución del algoritmo. Realizaremos un análisis que nos ayude a decidir cómo deberíamos implementar los pasos no especificados de quicksort.

El caso mejor

El caso mejor ocurre cuando la partición siempre divide el conjunto de partida en dos subconjuntos iguales. El tiempo de ejecución es $O(N \log N)$.

El caso mejor para quicksort se presenta cuando el pivote divide al conjunto en dos subconjuntos de igual tamaño. En tal caso tenemos dos llamadas con un tamaño de la mitad y una sobrecarga adicional lineal, igual que en mergesort. En consecuencia, en este caso, el tiempo de ejecución es $O(N \log N)$. (No hemos probado formalmente que éste sea en efecto el mejor caso. Tal demostración es posible, aunque no la veremos aquí.)

El caso peor

El caso peor ocurre cuando reiteradamente uno de los subconjuntos generado por la partición es vacío. El tiempo de ejecución es $O(N^2)$.

Ya que los subconjuntos de igual tamaño son los mejores para quicksort, uno podría esperar que los de muy distintos tamaños sean malos. Y esto es, en efecto, lo

que ocurre. Supongamos que en cada paso de la recursión, el pivote es el menor elemento. En tal caso, el subconjunto I de elementos pequeños será vacío y el subconjunto D de elementos grandes contendrá todos los elementos, excepto el pivote. Habrá ahora que llamar recursivamente a quicksort con el subconjunto D. Supongamos que $T(N)$ es el tiempo de ejecución de quicksort con N elementos y que el tiempo de ordenar 0 o 1 elemento es solamente una unidad. Supongamos también que hacer la partición de N elementos requiere N unidades de tiempo. Entonces, para $N > 1$, obtenemos un tiempo de ejecución que satisface

$$T(N) = T(N - 1) + N. \tag{8.1}$$

En otras palabras, la Ecuación 8.1 afirma que en este caso el tiempo que necesita quicksort para ordenar N elementos es igual al tiempo de ordenar recursivamente los $N - 1$ elementos del subconjunto de elementos mayores más las N unidades del coste de realizar la partición. Esto asumiendo que en todos los pasos de la recursión somos tremendamente desafortunados y tomamos como pivote el menor elemento. Para simplificar el análisis, normalizamos eliminando los factores constantes. Podemos resolver la recurrencia desplegando repetidamente la Ecuación 8.1:

$$\begin{aligned} T(N) &= T(N - 1) + N \\ T(N - 1) &= T(N - 2) + (N - 1) \\ T(N - 2) &= T(N - 3) + (N - 2) \\ &\cdots \\ T(2) &= T(1) + 2 \end{aligned} \tag{8.2}$$

Cuando sumamos por columnas todos los términos de la Ecuación 8.2, podemos cancelar términos de uno y otro lado, obteniendo

$$T(N) = T(1) + 2 + 3 + \cdots + N = N(N + 1)/2 = O(N^2). \tag{8.3}$$

Este análisis corrobora la intuición de que una división desigual no es buena. Gastamos N unidades de tiempo para hacer la partición y después tenemos que hacer una llamada recursiva para $N - 1$. Entonces gastamos $N - 1$ unidades para hacer la partición de este subconjunto, sólo para hacer una llamada recursiva con $N - 2$ elementos. En esa llamada gastamos $N - 2$ unidades para realizar la partición, y así sucesivamente. El tiempo total para realizar todas las particiones a través de las llamadas recursivas coincide exactamente con el obtenido en la Ecuación 8.3. Esto nos dice que cuando implementemos la selección del pivote y el paso de partición, no debemos hacer nada que haga probable la obtención de subconjuntos de tamaño desigual.

Caso medio

El caso medio es $O(N \log N)$. Aunque esto pueda parecer claro, se precisa una demostración formal.

Los dos primeros análisis nos dicen que el mejor y el peor caso son enormemente diferentes. Naturalmente, queremos saber qué ocurre en el caso medio. Podríamos esperar que ya que cada subproblema es en promedio la mitad que el original, la cota para el caso medio sea $O(N \log N)$. Esa expectativa, aunque correcta para la

aplicación particular de quicksort que presentamos aquí, no constituye una demostración formal. Los análisis en el caso medio no pueden realizarse a la ligera. Supongamos, por ejemplo, que tenemos un algoritmo para elegir el pivote que garantiza seleccionar el menor o el mayor elemento, cada uno con probabilidad 1/2. Entonces el tamaño medio del subconjunto con elementos pequeños es aproximadamente $N/2$, así como el del subconjunto de elementos grandes (ya que cada uno de ellos tiene con igual probabilidad 0 o $N - 1$ elementos). Pero el tiempo de ejecución de quicksort con esta selección del pivote será siempre cuadrático, ya que siempre se realiza una división pobre de los elementos. Debemos ser muy cuidadosos en cómo asignamos la etiqueta «en el caso medio». Siendo más precisos, podemos argumentar que el conjunto de elementos pequeños tendrá con igual probabilidad 0, 1, 2, ..., o $N - 1$ elementos, lo que también es cierto para el subconjunto de elementos mayores. Partiendo de esta hipótesis, sí podemos probar que el tiempo de ejecución de quicksort en el caso medio sería $O(N \log N)$.

El coste en el caso medio de una llamada recursiva se obtiene haciendo la media de los costes sobre todos los posibles tamaños de los subproblemas.

Puesto que el coste de quicksort para N elementos es igual a N unidades correspondientes a la partición, más el coste de dos llamadas recursivas, necesitamos determinar el coste medio de cada llamada recursiva. Si $T(N)$ representa el coste medio de quicksort para N elementos, el coste medio de cada llamada recursiva es igual a la media, sobre todos los posibles tamaños de los subproblemas, del coste medio de una llamada recursiva con un subproblema:

$$T(I) = T(D) = \frac{T(0) + T(1) + T(2) + \cdots + T(N-1)}{N} \tag{8.4}$$

La Ecuación 8.4 afirma que miramos cada uno de los costes de las posibles llamadas y después hacemos la media. Como tenemos dos llamadas recursivas, más un tiempo lineal para hacer la partición, obtenemos

$$T(N) = 2\left(\frac{T(0) + T(1) + T(2) + \cdots + T(N-1)}{N}\right) + N. \tag{8.5}$$

El tiempo de ejecución en promedio viene dado por $T(N)$. Resolvemos la Ecuación 8.5 eliminando todos los valores de T menos el original.

Para resolver la Ecuación 8.5, empezamos por multiplicar ambos lados por N, obteniendo

$$NT(N) = 2(T(0) + T(1) + T(2) + \cdots + T(N-1)) + N^2. \tag{8.6}$$

Ahora escribimos la Ecuación 8.6 para el caso $N - 1$, con la idea de restarla de la anterior y así simplificar en gran medida la ecuación. Tenemos entonces

$$(N-1)T(N-1) = 2(T(0) + T(1) + T(2) + \cdots + T(N-2)) + (N-1)^2. \tag{8.7}$$

y si restamos la Ecuación 8.7 de la Ecuación 8.6, obtenemos

$$NT(N) - (N-1)T(N-1) = 2T(N-1) + 2N - 1.$$

Si reorganizamos los términos y eliminamos el -1 de la derecha (no significativo), obtenemos

$$NT(N) = (N+1)T(N-1) + 2N. \tag{8.8}$$

Una vez que tenemos escrito $T(N)$ en términos de $T(N-1)$ intentamos desplegar.

Ahora tenemos una fórmula para $T(N)$ en términos de $T(N-1)$. De nuevo, la idea es desplegar, pero la Ecuación 8.8 no está en la forma adecuada para hacerlo. Si dividimos la Ecuación 8.8 por $N(N+1)$, obtenemos

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N} + \frac{2}{N+1}.$$

Ahora podemos desplegar:

$$\begin{aligned}
\frac{T(N)}{N+1} &= \frac{T(N-1)}{N} + \frac{2}{N+1} \\
\frac{T(N-1)}{N} &= \frac{T(N-2)}{N-1} + \frac{2}{N} \\
\frac{T(N-2)}{N-1} &= \frac{T(N-3)}{N-2} + \frac{2}{N-1} \qquad \textbf{(8.9)} \\
&\cdots \\
\frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2}{3}
\end{aligned}$$

Si sumamos todas las ecuaciones de la Ecuación 8.9, obtenemos

$$\begin{aligned}
\frac{T(N)}{N+1} &= \frac{T(1)}{2} + 2\left(\frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{N} + \frac{1}{N+1}\right) \\
&= 2\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N+1}\right) - \frac{5}{2} \qquad \textbf{(8.10)} \\
&= O(\log N)
\end{aligned}$$

Utilizamos el hecho de que el N-ésimo número armónico es $O(\log N)$.

La última línea de la Ecuación 8.10 se deduce del Teorema 5.5. Cuando multiplicamos ambos lados por $N+1$, obtenemos el resultado final

$$T(N) = O(N \log N). \qquad \textbf{(8.11)}$$

8.6.3 Seleccionando el pivote

Ahora que hemos establecido que el tiempo de ejecución en promedio de quicksort es $O(N \log N)$, nuestro objetivo central será asegurarnos que el peor caso no se presenta nunca. Realizando un análisis complejo, podemos calcular la desviación típica del tiempo de ejecución de quicksort. El resultado es que si partimos de una permutación aleatoria, es casi seguro que el tiempo de ejecución para ordenarla estará próximo al caso medio. Pero debemos tener cuidado de que las entradas de-

generadas no lleven a tiempos de ejecución malos. Las entradas degeneradas incluyen, entre otros, el caso en el que los datos ya estén ordenados, y aquél en el que todos los elementos sean iguales. Como veremos más adelante, en ocasiones son los casos aparentemente más sencillos los que complican el algoritmo.

Una forma incorrecta

Seleccionar adecuadamente el pivote es crucial para un buen rendimiento. Nunca elija sin más el primer elemento como pivote.

La elección más popular cuando no se cuenta con la información que lo desaconseja, es tomar sin más el primer elemento (es decir, el elemento en la posición `ini`) como pivote. Esto es aceptable si la entrada es aleatoria, pero si la entrada ya está ordenada o está en orden inverso, entonces este pivote proporciona una división pobre, pues será un elemento extremo. Es más, este mal comportamiento se repetirá recursivamente. Como vimos anteriormente en este capítulo, invertiríamos un tiempo cuadrático en ¡no hacer nada! (el vector ya venía ordenado). Esto, huelga decirlo, es bastante embarazoso. En consecuencia, *nunca* use sin más el primer elemento como pivote.

Una elección segura

El elemento central es una elección razonable aunque pasiva.

Una elección perfectamente razonable es elegir como pivote el elemento central, es decir, el elemento en la posición `(ini + fin) / 2` del vector. Cuando la entrada ya está ordenada, esto nos da el pivote perfecto, en cada llamada recursiva. Por supuesto, podríamos construir un vector de entrada que forzaría un comportamiento cuadrático bajo esta estrategia (*véase* el Ejercicio 8.8), pero las posibilidades de que se ejecute aleatoriamente un caso que requiera incluso el doble del caso medio son astronómicamente pequeñas.

Partición con la mediana de tres

En la *partición con la mediana de tres,* se utiliza como pivote la mediana del primer, el último y el elemento central. Esto simplifica la fase de partición de quicksort.

Elegir el elemento central como pivote evita los casos degenerados que representan entradas no aleatorias. Sin embargo, ésta es una elección pasiva. Es decir, no intentamos elegir un buen pivote; en vez de eso, evitamos únicamente elegir uno malo. La *partición con la mediana de tres* es un intento de seleccionar un pivote mejor que el central.

La mediana de un grupo de N elementos es el $\lceil N/2 \rceil$-ésimo menor elemento. La mejor elección del pivote sería claramente la mediana, pues garantizaría una división uniforme de los elementos. Desgraciadamente, calcular la mediana no es fácil y ralentizaría considerablemente a quicksort. Lo que queremos es obtener una buena estimación de la mediana sin gastar demasiado tiempo. Dicha estimación puede hacerse por muestreo: es decir, tomamos un subconjunto de los elementos a ordenar y buscamos su mediana. Éste es el método clásico utilizado en los sondeos de opinión. Cuanto mayor sea la muestra, más acertada será la estimación. Sin embargo, cuanto mayor sea la muestra más tardaremos en hacer los cálculos. Se ha probado que una muestra de tamaño tres proporciona una pequeña mejora en el caso promedio de quicksort, y también simplifica el código de parti-

ción resultante, eliminando algunos casos especiales. Se ha probado también que muestras de gran tamaño no mejoran el rendimiento de forma significativa, por lo que no merecen la pena.

Los tres elementos que utilizaremos en nuestra muestra son el primero, el central y el último. Por ejemplo, teniendo como entrada (8, 1, 4, 9, 6, 3, 5, 2, 7, 0), el elemento más a la izquierda es 8, el de más a la derecha es 0 y el del centro es 6; por tanto, el pivote escogido sería 6 (cuando hay dos elementos en el centro, se elige el de más a la izquierda). Observe que para vectores ya ordenados, mantenemos el elemento central como pivote, lo que vuelve a representar el mejor caso.

8.6.4 Estrategia de partición

Existen varias estrategias de partición utilizadas con asiduidad. Se sabe que la descrita en esta sección da buenos resultados. A continuación se discute la estrategia de partición más simple. Funciona en tres pasos. La Sección 8.6.6 muestra las mejoras obtenidas cuando se utiliza la mediana de tres.

Paso 1: intercambiar el pivote con el último elemento.

El primer paso en el algoritmo de partición es quitar el pivote de en medio intercambiándolo con el último elemento. El resultado sobre nuestro ejemplo de entrada se muestra en la Figura 8.9. El pivote se muestra con el sombreado más oscuro, al final del vector.

Por ahora, asumiremos que todos los elementos son distintos y dejaremos para después qué hacer en presencia de duplicados. Como caso límite, nuestro algoritmo debería funcionar correctamente cuando *todos* los elementos son iguales.

Paso 2: mover `i` de izquierda a derecha, y `j` de derecha a izquierda. Cuando `i` apunta a un elemento mayor que el pivote, `i` se para. Cuando `j` apunta a un elemento menor que el pivote, `j` se para. Si `i` y `j` no se han cruzado entre sí, intercambiar los elementos y continuar. En otro caso, dar por concluido el bucle.

En el segundo paso, utilizamos nuestra estrategia de partición para mover todos los elementos menores a la parte de la izquierda, y todos los elementos mayores a la parte de la derecha. En las Figuras de la 8.10 a la 8.15, las celdas blancas son celdas que sabemos que están bien colocadas. No se sabe todavía si las que están en penumbra están bien colocadas.

Buscamos de izquierda a derecha un elemento mayor. Para hacer esto utilizamos un contador `i`, inicializado a `ini`. Por su parte, buscamos de derecha a izquierda un elemento menor. Lo hacemos utilizando un contador `j`, inicializado a `fin-1`. La Figura 8.10 muestra que la búsqueda del elemento mayor para en el 8 y la búsqueda del menor para en el 2. Estas celdas han sido sombreadas ligeramente. Observe como hemos saltado el 7, porque sabemos que el 7 no es menor que el pivote por lo que está bien colocado. En consecuencia queda coloreado de blanco. Ahora tenemos en el lado izquierdo del vector, un elemento mayor que el pivote, el 8, y uno menor, el 2, en la parte derecha. Debemos intercambiarlos para colocarlos correctamente, como muestra la Figura 8.11.

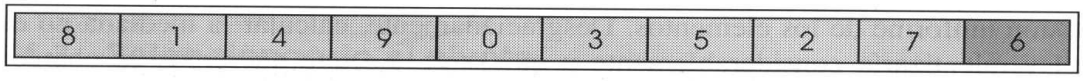


Figura 8.9 Algoritmo de partición: el pivote, 6, se coloca al final.

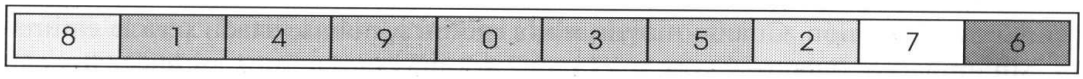


Figura 8.10 Algoritmo de partición: `i` se para en el elemento mayor que el pivote, 8; `j` se para en el elemento menor que el pivote, 2.

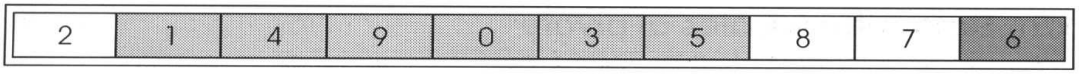


Figura 8.11 Algoritmo de partición: los elementos mal colocados 8 y 2 son intercambiados.

Al continuar la ejecución del algoritmo, `i` se para en el elemento mayor que el pivote, 9, y `j` se para en el elemento menor, 5. Una vez más, los elementos que se saltan `i` y `j` durante el proceso están bien colocados. La Figura 8.12 muestra el resultado: los elementos de los extremos (sin incluir el pivote) están colocados correctamente.

Lo siguiente que hay que hacer es intercambiar los elementos indexados por `i` y `j`, como muestra la Figura 8.13. El recorrido continúa, con `i` parándose en el elemento mayor que el pivote, 9, y `j` parándose en el elemento menor, 3. Sin embargo, en este momento los índices `i` y `j` se han cruzado, por lo que un intercambio no tendría sentido. Esto se muestra en la Figura 8.14, en la cual vemos que el elemento indexado por `j` está colocado correctamente, por lo que no debe moverse.

En la Figura 8.14 se muestra que todos los elementos salvo el pivote están correctamente colocados. ¿No sería bonito que con un intercambio pudieramos ubicarlo correctamente, para así terminar? Pues sí, podemos. Todo lo que tenemos que hacer es intercambiar el pivote con el elemento de la posición `i`. El resultado se muestra en la Figura 8.15. El elemento apuntado por `i` era mayor que el pivote por lo que pasarlo a la última posición es acertado.

Paso 3: intercambio del elemento en la posición `i` con el pivote.

Observe que el algoritmo de partición no necesita memoria auxiliar y que cada elemento es comparado con el pivote exactamente una vez. Cuando se escribe el código, esto se traduce a un bucle interno muy ajustado.

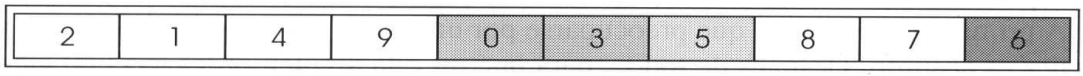


Figura 8.12 Algoritmo de partición: `i` se para en el elemento mayor que el pivote 9; `j` se para en el elemento menor, 5.

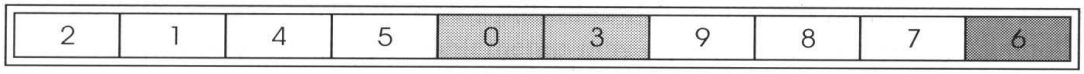


Figura 8.13 Algoritmo de partición: los elementos mal colocados 9 y 5 son intercambiados.

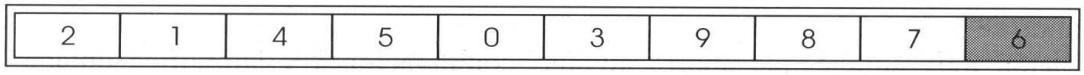


Figura 8.14 Algoritmo de partición: `i` se para en el elemento mayor que el pivote 9; `j` se para en el elemento menor, 3.

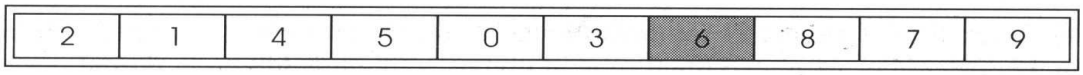


Figura 8.15 Algoritmo de partición: intercambio del pivote y el elemento en la posición `i`.

8.6.5 Elementos iguales al pivote

Para garantizar un buen rendimiento `i` y `j` deben parar cuando apuntan a un elemento igual que el pivote.

Un detalle importante que debemos considerar es cómo tratar los elementos que son iguales que el pivote. Las preguntas que debemos hacernos son si `i` debería parar al encontrar un elemento igual al pivote, y si `j` debería parar al ver un elemento igual que el pivote. `i` y `j` deberían hacer lo mismo; de otra forma, el paso de partición sería desigual en ambos lados. Por ejemplo, si `i` parara y `j` no, todos los elementos iguales al pivote terminarían en la parte de la derecha.

Consideremos el caso en el que todos los elementos del vector son idénticos. Si paran tanto `i` como `j`, habrá muchos intercambios de elementos idénticos. Aunque esto parezca inútil, el efecto positivo que se consigue es que `i` y `j` se cruzan en la mitad del vector, por lo que, cuando el pivote es recolocado, la partición crea dos subconjuntos casi iguales. En consecuencia, se puede aplicar el análisis del caso mejor, y el tiempo de ejecución sería $O(N \log N)$.

Si no paran ni `i` ni `j`, entonces `i` termina en la última posición (suponiendo, por supuesto, que controlamos que no se salga de los límites) y no se realiza ningún intercambio. Esto parece bueno, dada la simetría, hasta que nos damos cuenta de que el pivote se mantendría en la última posición pues ésta es la celda apuntada por `i`. Así se obtendrían dos conjuntos completamente dispares, y el tiempo de ejecución sería el del caso peor, $O(N^2)$. El efecto es el mismo que el de utilizar el primer elemento del vector cuando éste está ordenado. Tardamos un tiempo cuadrático en no hacer nada.

Podemos concluir que es mejor hacer intercambios innecesarios y crear subconjuntos iguales, que arriesgarse a crear conjuntos desiguales. Por tanto, haremos que `i` y `j` paren cuando encuentre un elemento igual que el pivote. Ésta resulta ser la única de las cuatro posibilidades que no conduce a un tiempo cuadrático para este tipo de entrada.

A primera vista, parece que preocuparse por un vector con todos los elementos iguales es absurdo. Después de todo, ¿para qué querría alguien ordenar 5.000 elementos idénticos? Sin embargo, recuerde que quicksort es recursivo. Suponga que hay 100.000 elementos, de los cuales 5.000 son iguales. En algún momento quicksort podría hacer una llamada recursiva con sólo los 5.000 elementos idénticos. Y en tales ocasiones es importante asegurar que los 5.000 elementos se ordenan eficientemente.

Calcular la mediana de tres requiere ordenar tres elementos del vector. Aprovechando esto, podemos hacer que la partición comience partiendo de dos elementos ya saltados, con lo que no es necesario preocuparse de si se rebasa el límite derecho del vector, pues tal cosa no puede suceder.

8.6.6 Partición con la mediana de tres

Cuando se utiliza la partición con la mediana de tres, podemos realizar una optimización muy simple que ahorra algunas comparaciones y simplifica notablemente el código. En la Figura 8.16 volvemos a mostrar nuestro vector original.

La partición con la mediana de tres requiere que encontremos la mediana del primer elemento, el central y el último. La forma más fácil de hacerlo consiste en ordenarlos sobre las posiciones que ocupan en el vector. El resultado se muestra en la Figura 8.17. Observe el sombreado resultante: el elemento que termina en la primera posición es menor (o igual) que el pivote, y el elemento en la última posición es mayor (o igual) que el pivote. Partiendo de estos hechos concluimos lo siguiente:

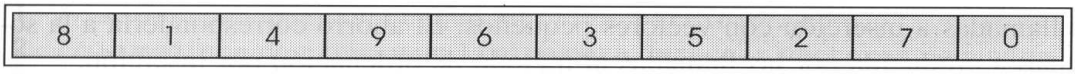


Figura 8.16 Vector original.

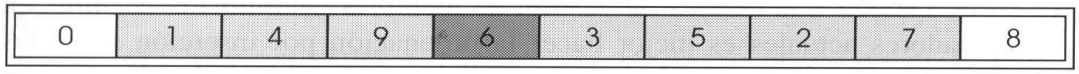


Figura 8.17 Resultado de ordenar tres elementos (el primero, el central y el último).

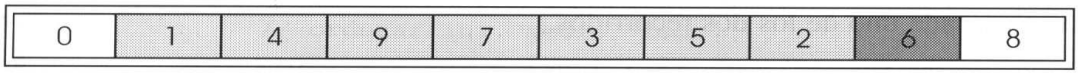


Figura 8.18 Resultado del intercambio del pivote con el penúltimo elemento.

1. No deberíamos intercambiar el pivote con el elemento en la última posición. En vez de eso, deberíamos intercambiarlo con el que ocupa la penúltima posición. Esto se ilustra en la Figura 8.18.
2. `i` puede empezar en `ini+1` y `j` en `fin-2`.
3. Podemos estar seguros de que cuando `i` busca un elemento mayor parará, porque, en el peor caso, encontrará el pivote (parando al darse la igualdad).
4. Podemos estar seguros de que cuando `j` busca un elemento menor parará, porque, en el peor caso, encontrará el primer elemento (parando al darse igualdad).

Todas estas optimizaciones se incorporarán en nuestro código final del algoritmo en Java.

8.6.7 Vectores pequeños

Ordene los vectores con a lo sumo diez elementos o menos utilizando inserción. Inserte este test en la rutina recursiva de quicksort.

Nuestra última optimización tiene que ver con los vectores pequeños. ¿Merece la pena utilizar una rutina tan potente como quicksort cuando sólo hay que ordenar diez elementos? Por supuesto, la respuesta es no. Una rutina simple, como la ordenación por inserción, será probablemente más rápida sobre vectores pequeños. La naturaleza recursiva de quicksort nos dice que se generarán muchas llamadas con subconjuntos pequeños. Por tanto, merece la pena preguntar por el tamaño del vector. Si es menor que un cierto límite, aplicaremos ordenación por inserción; en otro caso, seguiremos haciendo quicksort.

Se ha comprobado que diez elementos es un buen límite, aunque cualquier valor entre 5 y 20 produce resultados similares. El límite óptimo dependería de la máquina y del código máquina que se genere en cada caso. El utilizar un límite nos evita casos degenerados. Por ejemplo, no tiene sentido buscar la mediana de tres elementos cuando no hay ni siquiera tres elementos.

En el pasado se pensó que aún podía ser mejor dejar el vector algo desordenado no haciendo nada, de momento, cuando nos encontramos subvectores pequeños. Ya que la ordenación por inserción es muy eficiente para vectores casi ordenados, podríamos mostrar matemáticamente que una ejecución final de la ordenación por inserción para limpiar el vector sería más rápido que ejecutar todas

las llamadas a inserción con vectores pequeños. El ahorro correspondería a la sobrecarga producida por las llamadas al método de ordenación por inserción.

Hoy en día, las llamadas a métodos no son tan costosas como antes. Además, un segundo recorrido del vector para realizar la ordenación por inserción aporta un cierto coste. Debido a la técnica de gestión de memoria denominada *caching*, en los computadores actuales es mejor hacer la ordenación por inserción sobre los subvectores pequeños. Ello es debido a que los accesos a memoria localizados son más rápidos que los accesos no localizados; así, acceder a una misma posición de memoria dos veces casi seguidas en un recorrido, es más rápido que realizar un acceso en cada uno de los dos recorridos.

8.6.8 Rutina de quicksort en Java

Utilizamos un método envoltorio para establecer los valores iniciales de los argumentos.

La implementación concreta de quicksort se muestra en la Figura 8.19. El método `quicksort` visible, declarado en las líneas de la 46 a la 49, es simplemente un envoltorio que llama al método recursivo `quicksort`. En consecuencia, sólo discutiremos la implementación del método recursivo `quicksort`.

En la línea 7 se pregunta si el vector a ordenar es pequeño y se llama a la ordenación por inserción (no mostrada) cuando el tamaño del problema está por debajo de un valor concreto, dado por la constante `LIMITE`. En otro caso, se sigue con el procedimiento recursivo. Las líneas de la 12 a la 18 ordenan el primer elemento, el central y el último sobre sus propias posiciones. El elemento central se utiliza como pivote y en las líneas 21 y 22 se intercambia (utilizando `intercambiarReferencias`, que no se muestra) con el elemento en la última posición. Pasamos entonces a realizar la partición. Se inicializan los contadores `i` y `j` a una posición anterior a su valor inicial deseado, dado que los operadores prefijos de incremento y decremento los ajustarán inmediatamente en las líneas 28 y 30 antes de realizar ningún acceso al vector. Cuando el primer bucle `while` de la línea 28 termina, `i` estará apuntando a un elemento que es mayor, o posiblemente igual, que el pivote. De igual forma, cuando el segundo bucle termina, `j` apuntará a un elemento menor o igual que el pivote. Si `i` y `j` no se han cruzado, estos elementos son intercambiados, tras lo que se continúa el recorrido. En otro caso, se termina el recorrido, colocándose correctamente el pivote en la línea 39. La ordenación termina con las dos llamadas recursivas en las líneas 41 y 42.

El bucle más interno de quicksort es muy ajustado y eficiente.

Las operaciones fundamentales se realizan en las líneas 28 a 35. Los recorridos consisten en una serie de operaciones simples: incrementos, accesos al vector y comparaciones sencillas. Esto explica la rapidez de quicksort. Para asegurar la eficiencia de los bucles más internos, queremos estar seguros de que el intercambio de la línea 33 consta de las tres asignaciones esperadas y no incurre en la sobrecarga de una llamada a un método. Un buen compilador que optimice el código generado haría esto automáticamente, ya que todos los métodos en el paquete `Ordenacion` están declarados como `final`, lo que permite la optimización en línea. Sin embargo, podría ser necesario introducir manualmente el código, si el compilador no es capaz de realizar dicha optimización.

Quicksort es un ejemplo clásico de la utilización de un análisis previo para guiar la implementación del programa.

Aunque el código ahora parece sencillo, es importante no olvidar que ello se debe al análisis realizado antes de la implementación. Todavía hay algunas trampas escondidas (*véase* el Ejercicio 8.12). Quicksort es un ejemplo clásico de la utilización de un análisis previo para guiar la implementación del programa.

```
1  // Algoritmo quicksort que utiliza partición con
2  // la mediana de tres, y un límite para detectar vectores pequeños
3
4  private static void
5  quicksort( Comparable [ ] a, int ini, int fin )
6  {
7      if( ini + LIMITE > fin )
8          ordenacionPorInsercion( a, ini, fin );
9      else
10     {
11             // ordenar ini, medio, fin
12         int medio = ( ini + fin ) / 2;
13         if( a[ medio ].menorQue( a[ ini ] ) )
14             intercambiarReferencias( a, ini, medio );
15         if( a[ fin ].menorQue( a[ ini ] ) )
16             intercambiarReferencias ( a, ini, fin );
17         if( a[ fin ].menorQue( a[ medio ] ) )
18             intercambiarReferencias ( a, medio, fin );
19
20             // Colocar el pivote en la posición fin - 1
21         intercambiarReferencias ( a, medio, fin - 1 );
22         Comparable pivote = a[ fin - 1 ];
23
24             // Empezar la partición
25         int i, j;
26         for( i = ini, j = fin - 1; ; )
27         {
28             while( a[ ++i ].menorQue( pivote ) )
29                 ;
30             while( pivote.menorQue( a[ --j ] ) )
31                 ;
32             if( i < j )
33                 intercambiarReferencias ( a, i, j );
34             else
35                 break;
36         }
37
38             // Colocar el pivote
39         intercambiarReferencias ( a, i, fin - 1 );
40
41         quicksort( a, ini, i - 1 ); // Ordenar elementos pequeños
42         quicksort( a, i + 1, fin ); // Ordenar elementos grandes
43     }
44 }
45
46 public static void quicksort( Comparable [ ] a )
47 {
48     quicksort( a, 0, a.length - 1 );
49 }
```

Figura 8.19 Quicksort con partición con la mediana de tres y un límite para detectar vectores pequeños.

8.7 Selección rápida

El problema de selección consiste en encontrar el *k*-ésimo menor elemento.

Un problema muy relacionado con la ordenación es el problema de *selección*. En el mismo, dado un vector de N elementos, se debe encontrar el k-ésimo menor elemento. Un caso especial importante es el de encontrar la mediana, o sea el $N/2$-ésimo elemento. Obviamente, podemos hacerlo ordenando el vector, pero es de esperar que selección sea un proceso más rápido, al solicitarse una menor información. Esto es así, en efecto. Haciendo un pequeño cambio en quicksort, podemos resolver el problema de selección en tiempo lineal, en promedio. Llamamos a este algoritmo *selección rápida*. Los pasos que realiza *seleccionRápida*(S, k) son los siguientes:

1. Si el número de elementos en S es 1, entonces presumiblemente k también es 1, por lo que se puede devolver el único elemento en S.
2. Elegir un elemento v de S como el pivote.
3. Hacer una partición de $S - \{v\}$ en I y D, exactamente igual a como se hacía en quicksort.
4. Si k es menor o igual que el número de elementos en I, entonces el elemento que estamos buscando debe estar en I, por lo que se llama recursivamente a *seleccionRápida*(I, k). Si k es exactamente igual a uno más que el número de elementos de I, entonces el pivote es el k-ésimo menor elemento y se puede devolver como respuesta. En otro caso, el k-ésimo menor elemento estará en D. De nuevo podemos hacer una llamada recursiva y devolver el resultado obtenido.

SeleccionRápida se utiliza para realizar la selección. Es un algoritmo muy parecido a quicksort, aunque sólo hace una llamada recursiva. El tiempo de ejecución es lineal en promedio.

La selección rápida precisa una única llamada recursiva, en comparación con las dos que hace quicksort. Pese a ello, el peor caso es idéntico al de quicksort, siendo, por tanto, cuadrático. Se presenta cuando la llamada recursiva se realiza con sólo un elemento menos. En tal caso, la selección rápida no ahorra gran cosa. Sin embargo, podemos mostrar que el tiempo es lineal en promedio, utilizando un análisis similar al de quicksort (*véase* el Ejercicio 8.9).

La implementación de la selección rápida es aún más simple de lo que nuestra descripción abstracta puede hacer pensar. Una implementación completa se muestra en la Figura 8.20. Excepto por el parámetro adicional k y la realización de una sola llamada recursiva, el algoritmo es idéntico al de quicksort. Obsérvese que cuando termina la ejecución del algoritmo, el k-ésimo menor elemento está en su posición correcta en el vector ordenado. Además, en las posiciones anteriores tendremos los $k - 1$ menores elementos del vector, aunque no ordenados, y en las $n - k$ últimas los más grandes. Únicamente hay que tener cuidado con el hecho de que al empezar a contar los índices en 0, el cuarto elemento, por ejemplo, estará en la posición tres, y no en la cuatro. Por otra parte, como el proceso se hace *in situ* el orden original del vector desaparece. Si no se desea este efecto lateral, debemos hacer que la rutina envoltorio pase una copia del vector.

El algoritmo de selección lineal en el caso peor es un resultado clásico aunque muy poco útil en la práctica.

Utilizando la partición con la mediana de tres se consigue que la probabilidad del caso peor sea casi despreciable. Eligiendo cuidadosamente el pivote, puede demostrarse que el caso peor nunca puede ocurrir, con lo que el tiempo de ejecución sería lineal aun en el peor de los casos. Sin embargo, el algoritmo resultante solamente tiene un interés teórico, pues la constante que oculta la nota-

ción O es mucho mayor que la constante del algoritmo ordinario que utiliza la partición con la mediana de tres.

```
1  // Método de selección interno que hace las llamadas recursivas.
2  // Coloca el k-ésimo menor elemento en a[k-1].
3  // La llamada inicial es seleccionRapida( a, 0, a.length, k )
4
5  private static void
6  seleccionRapida( Comparable [ ] a, int ini, int fin, int k )
7  {
8      if( ini + LIMITE > fin )
9          ordenacionPorInsercion( a, ini, fin );
10     else
11     {
12             // ordenar ini, medio, fin
13         int medio = ( ini + fin ) / 2;
14         if( a[ medio ].menorQue( a[ ini ] ) )
15             intercambiarReferencias( a, ini, medio );
16         if( a[ fin ].menorQue( a[ ini ] ) )
17             intercambiarReferencias ( a, ini, fin );
18         if( a[ fin ].menorQue( a[ medio ] ) )
19             intercambiarReferencias ( a, medio, fin );
20
21             // Colocar el pivote en la posición fin - 1
22         intercambiarReferencias ( a, medio, fin - 1 );
23         Comparable pivote = a[ fin - 1 ];
24
25             // Empezar la partición
26         int i, j;
27         for( i = ini, j = fin - 1; ; )
28         {
29             while( a[ ++i ].menorQue( pivote ) )
30                 ;
31             while( pivote.menorQue( a[ --j ] ) )
32                 ;
33             if( i < j )
34                 intercambiarReferencias ( a, i, j );
35             else
36                 break;
37         }
38
39             // Colocar el pivote
40         intercambiarReferencias ( a, i, fin - 1 );
41
42             // Recursión; sólo cambia esta parte
43         if( k - 1 < i )
44             seleccionRapida( a, ini, i - 1, k );
45         else if ( k - 1 > i )
46             seleccionRapida( a, i + 1, fin, k );
47     }
48 }
```

Figura 8.20 Selección rápida con partición con la mediana de tres y un límite para detectar vectores pequeños.

8.8 Una cota inferior para la ordenación

Cualquier algoritmo de ordenación basado en comparaciones debe utilizar aproximadamente $N \log N$ comparaciones en promedio y en el caso peor.

Las demostraciones son abstractas; mostramos la cota inferior en el caso peor.

Aunque tenemos algoritmos de ordenación $O(N \log N)$, no está claro que esto sea lo mejor que podemos conseguir. En esta sección probamos que cualquier algoritmo de ordenación que sólo utilice comparaciones necesita $\Omega(N \log N)$ comparaciones (y por tanto al menos el mismo tiempo) en el caso peor. Esto significa que *cualquier algoritmo que ordene utilizando comparaciones entre elementos debe utilizar no menos de unas $N \log N$ comparaciones para alguna entrada.* Se puede utilizar una técnica similar para probar que esto también es cierto en promedio.

¿Todo método de ordenación debe basarse en comparaciones? La respuesta es no. Sin embargo, los algoritmos que no utilizan comparaciones sólo funcionan, en principio, con tipos restringidos, como los enteros. Aunque puede que a menudo sólo necesitemos ordenar enteros (*véase* el Ejercicio 8.13), un algoritmo de ordenación de propósito general no puede hacer asunciones tan restrictivas sobre la entrada. Sólo puede asumir la obvia de que si los elementos han de ser ordenados, cualesquiera dos elementos pueden ser comparados.

Probamos a continuación uno de los teoremas fundamentales de la computación (Teorema 8.3). Antes de comenzar recuerde que el producto de los N primeros enteros positivos es $N!$. La demostración es una demostración existencial, un tanto abstracta. Demuestra que siempre debe existir alguna entrada mala.

Teorema 8.3

Cualquier algoritmo que ordene utilizando únicamente comparaciones entre elementos debe usar al menos $\lceil \log(N!) \rceil$ comparaciones, con alguna secuencia de entrada.

Demostración

Podemos considerar como posibles entradas las permutaciones de 1, 2, ..., N. Esto es así pues sólo importa el orden relativo de los elementos, no su valor concreto. Por tanto, el número de posibles entradas es el número de permutaciones diferentes de N elementos, que es exactamente N!. Denotemos por P_i el máximo número de permutaciones que pueden ser consistentes con la información obtenida después de que el algoritmo ha procesado i comparaciones. Sea F el número de comparaciones realizadas cuando termina la aplicación del algoritmo. Se tiene lo siguiente: (a) $P_0 = N!$ ya que todas las permutaciones son posibles antes de la primera comparación; (b) $P_F = 1$, ya que si fueran posibles más de una permutación, el algoritmo no podría terminar todavía, al no poder estar seguros de que haya producido el resultado correcto; (c) $P_i \geq P_{i-1}/2$, porque después de una comparación, cada permutación va a uno de estos dos grupos: el grupo de las aún posibles y el grupo de las que ya no son posibles. El mayor de estos dos grupos debe contener al menos la mitad de las permutaciones, de modo que si el resultado de la última comparación realizada corresponde a dicho grupo mayor, el tamaño del conjunto de permutaciones consistentes sólo podría dividirse por la mitad tras cada comparación. El efecto del algoritmo de ordenación es, por tanto, ir desde el estado P_0, en el cual son posibles todas las N! permutaciones, al estado final P_F, en el cual sólo es posible una permutación, con la restricción de que existe una entrada tal que en cada comparación, sólo se pueden eliminar la mitad de

las permutaciones. Por el principio de sucesivas divisiones por la mitad, sabemos que al menos se necesitan $\lceil \log(N!) \rceil$ *comparaciones para esa entrada.*

Demostración 8.3 (continuación)

¿Cómo de grande es $\lceil \log(N!) \rceil$? Es aproximadamente $N \log N - 1{,}44N$.

Resumen

En la mayoría de las aplicaciones generales de ordenación interna, el método elegido es la ordenación por inserción, Shellsort o quicksort. La decisión de cuál usar depende del tamaño de la entrada.

La ordenación por inserción es apropiada para pequeñas cantidades de datos. Shellsort es una buena elección para ordenar cantidades moderadas de datos. Con una secuencia de incrementos apropiada, tiene un excelente rendimiento, necesitando sólo unas pocas líneas de código. Quicksort tiene el mejor rendimiento, pero es complicado de implementar. Asintóticamente, tiene un rendimiento cercano a $O(N \log N)$ cuando se usa una implementación cuidadosa. Hemos visto que dicho rendimiento es esencialmente tan bueno como cabe esperar. La Sección 20.5 estudia otro método interno muy común, *heapsort* (método del montículo).

Cuando queremos ordenar datos que no caben juntos en memoria principal, se necesitan técnicas diferentes. La técnica general se discute en la Sección 20.6. Utiliza el algoritmo `mezclar` descrito en la Sección 8.5.

Para probar y comparar los méritos de los diferentes algoritmos de ordenación, necesitamos ser capaces de generar entradas aleatorias. La aleatoriedad es un tema importante en general que se estudia en el capítulo siguiente.

Elementos del juego

algoritmo de ordenación basado en comparaciones Un algoritmo que toma decisiones sobre el orden utilizando sólo comparaciones.

demostración de la cota inferior de la ordenación Confirma que cualquier algoritmo de ordenación basado en comparaciones debe usar al menos unas $N \log N$ comparaciones, tanto en promedio como en el caso peor.

inversión Un par no ordenado de elementos de un vector. Utilizadas para medir el grado de desorden.

mergesort Método de ordenación que utiliza divide y vencerás para obtener una ordenación $O(N \log N)$.

ordenación por disminución de intervalos Otro nombre para Shellsort.

partición con la mediana de tres Se utiliza como pivote la mediana del primer elemento, el central y el último. Esto simplifica la fase de partición de quicksort.

partición Paso de quicksort que coloca cada elemento del vector, excepto el pivote, en uno de dos posibles grupos, uno formado por los elementos menores que el pivote y otro formado por los mayores que él.

pivote En quicksort, el elemento que sirve para dividir el vector en dos grupos: uno con los elementos menores que él y otro con los elementos mayores.

quicksort Un algoritmo divide y vencerás rápido, cuando se implementa apropiadamente. Es, de hecho, el algoritmo de ordenación basado en comparaciones más rápido en la práctica.

selección Proceso para buscar el *k*-ésimo menor elemento de un vector.

selección rápida Algoritmo utilizado para realizar una selección, similar a quicksort, pero con una sola llamada recursiva. El tiempo de ejecución en promedio es lineal.

Shellsort Algoritmo subcuadrático que funciona bien en la práctica y es simple de implementar. El rendimiento de Shellsort depende en gran media de la secuencia de incrementos en que se base. Conlleva un análisis desafiante, todavía no completamente resuelto.

Errores comunes

1. Los algoritmos de ordenación implementados en este capítulo trabajan a partir de la componente 0 del vector, y no de la 1.
2. Utilizar una secuencia de incrementos inadecuada es un error común al usar Shellsort. Asegúrese de que la secuencia acaba en 1, y evite secuencias de las que se sabe que producen malos resultados.
3. Quicksort tiene multitud de trampas. Los errores más comunes tienen que ver con entradas ordenadas, elementos duplicados y particiones degeneradas.
4. El algoritmo de ordenación por inserción es apropiado para pequeñas entradas. Pero no es bueno con entradas grandes.

En Internet

Todos los algoritmos de ordenación, y una implementación de la selección rápida, se encuentran en un único fichero, como parte del paquete `DataStructures`, traducido como `EstructurasDatos`.

DuplicateTest.java	Contiene el método de la Figura 8.1 con un programa de prueba, en el directorio **Chapter08**.
Sort.java	Traducida por `Ordenacion.java`, en el directorio **DataStructures.**
TestSort.java	Clase en el directorio **Chapter08** que prueba todos los métodos de ordenación.

Ejercicios

Cuestiones breves

8.1. Ordene la secuencia (8, 1, 4, 1, 5, 9, 2, 6, 5) utilizando los siguientes métodos:

a) Ordenación por inserción.
b) Shellsort, con los incrementos (1, 3, 5).

c) Mergesort.
d) Quicksort, utilizando el elemento central como pivote y sin límite (muestre todos los pasos).
e) Quicksort, utilizando la mediana de tres como pivote y un límite de tres.

8.2. Un algoritmo de ordenación es *estable* si los elementos iguales permanecen en su orden original. ¿Cuáles de los algoritmos de ordenación vistos en este capítulo son estables y cuáles no? ¿Por qué?

8.3. Explique por qué el algoritmo quicksort del texto es típicamente mejor que permutar aleatoriamente la entrada y elegir el elemento central como pivote.

Problemas teóricos

8.4. Cuando todos los elementos son iguales, ¿cuál es el tiempo de ejecución de los siguientes métodos?

a) Ordenación por inserción.
b) Shellsort.
c) Mergesort.
d) Quicksort.

8.5. Cuando la entrada está ya ordenada, ¿cuál es el tiempo de ejecución de los siguientes métodos?

a) Ordenación por inserción.
b) Shellsort.
c) Mergesort.
d) Quicksort.

8.6. Cuando la entrada está originalmente ordenada, pero en orden inverso, ¿cuál es el tiempo de ejecución de los siguientes métodos?

a) Ordenación por inserción.
b) Shellsort.
c) Mergesort.
d) Quicksort.

8.7. Suponga que intercambiamos los elementos `a[i]` y `a[i+k]`, que originalmente estaban fuera de sitio. Demuestre que con ello al menos se elimina una inversión, y como mucho $2k - 1$.

8.8. Construya una entrada que provoque el caso peor para quicksort en los siguientes casos:

a) Utilizando el elemento central como pivote.
b) Haciendo partición con la media de tres.

8.9. Demuestre que la selección rápida tiene un rendimiento lineal en promedio. Hágalo resolviendo la Ecuación 8.5, con la constante 2 sustituida por 1.

8.10. Utilizando la fórmula de Stirling, $N! \geqslant (N/e)^N \sqrt{2\pi N}$, deduzca una estimación para $\log(N!)$.

8.11. Demuestre que cualquier algoritmo basado en comparaciones, utilizado para ordenar cuatro elementos, necesita 5 comparaciones para alguna entrada. Demuestre después que existe un algoritmo capaz de ordenar secuencias de cuatro elementos cualesquiera utilizando a lo sumo 5 comparaciones.

Problemas prácticos

8.12. Un estudiante modifica la rutina de la Figura 8.19 realizando los siguientes cambios en las líneas de la 26 a la 31.

```
for( i = ini + 1, j = fin - 2; ; )
{
    while( a[ i ].menorQue( pivote ) )
        i++;
    while( pivote.menorQue( a[ j ] ) )
        j--;
}
```

¿Es el resultado equivalente a la rutina original?

8.13. Si se tiene información específica adecuada sobre los elementos a ordenar, la ordenación puede pasar a ser lineal. Demuestre que una colección de N enteros `short` puede ser ordenada en un tiempo lineal. *Pista*: mantenga un vector indexado desde 0 hasta 65.535.

8.14. Tenemos un vector que contiene N números. Queremos saber si contiene dos números cuya suma sea igual a un número dado K. Por ejemplo, si la entrada es (8, 4, 1, 6) y K es 10, la respuesta es sí (4 y 6). Un mismo número puede ser usado dos veces. Haga lo siguiente:

a) De un algoritmo $O(N^2)$ para resolver el problema.
b) De un algoritmo $O(N \log N)$ para resolver el problema. *Pista*: ordene primero los elementos. Después de hacer esto puede resolver el problema en tiempo lineal.
c) Escriba un programa en Java para cada una de dichas soluciones y compare sus tiempos de ejecución.

8.15. Repita el Ejercicio 8.14 para cuatro números. Intente diseñar un algoritmo $O(N^2 \log N)$. *Pista*: calcule todas las posibles sumas de dos elementos, ordénelas y continúe como en el Ejercicio 8.14.

8.16. Repita el Ejercicio 8.14 para tres números. Intente diseñar un algoritmo $O(N^2)$.

8.17. El Ejercicio 5.25 pedía buscar la única solución entera a la ecuación $A^5 + B^5 + C^5 + D^5 + E^5 = F^5$ que satisface $0 < A \leqslant B \leqslant C \leqslant D \leqslant E \leqslant F \leqslant N$, donde N es 75. Use las ideas del Ejercicio 8.15 para obtener relativamente deprisa una solución, ordenando todos los posibles valores de $A^5 + B^5 + C^5$ y $F^5 - (D^5 + E^5)$, y viendo después si algún número del primer grupo aparece también en el segundo grupo. En términos de N, ¿cuánto espacio y tiempo requiere su algoritmo?

Prácticas de programación

8.18. Compare el rendimiento de Shellsort con varias secuencias de incrementos, siguiendo el siguiente procedimiento: obtenga experimentalmente el tiempo medio para un cierto tamaño de entrada N, generando varias secuencias aleatorias de N elementos. Utilice la misma entrada para todas las secuencias de incrementos. En un test por separado obtenga el número medio de comparaciones entre objetos `Comparable`, y el número de asignaciones de referencias a objetos `Comparable`. Haga un gran número de pruebas, pero sin que sea necesario, pongamos, más de una hora de tiempo de CPU. Las secuencias de incrementos son las siguientes:

a) La secuencia original de Shell (dividir repetidamente por 2).
b) La secuencia original de Shell, sumando 1 si el resultado es distinto de cero, pero par.
c) La secuencia de Gonnet mostrada en el texto, en la que repetidamente se divide por 2,2.
d) Los incrementos de Hibbard: 1, 3, 7, ..., $2^k - 1$.
e) Los incrementos de Knuth: 1, 4, 13, ... $(3^k - 1)/2$.
f) Los incrementos de Sedgewick: 1, 5, 19, 41, 109, ...; cada término es bien de la forma $9 \cdot 4^k - 9 \cdot 2^k + 1$, o de la forma $4^k - 3 \cdot 2^k + 1$.

8.19. Implemente los métodos de ordenación de Shellsort y quicksort y compare sus tiempos de ejecución. Use las implementaciones del texto. Ejecútelos con los siguientes objetos:

a) Enteros.
b) Números reales de tipo `double`.
c) Cadenas de caracteres.

8.20. Escriba un método que elimine todos los elementos repetidos de un vector A de N elementos. Devuelva el número de elementos que permanecen en A. Su método debe ejecutarse en promedio en tiempo $O(N \log N)$, (utilice quicksort como paso de preprocesamiento).

8.21. El Ejercicio 8.2 estudia la ordenación estable. Escriba un método genérico que realice un quicksort estable. Para hacerlo, se crea un vector en el que cada entrada contenga una referencia a un elemento y su posición inicial en el vector. Se ordena este vector; si dos entradas tienen elementos de datos idénticos, se utiliza la posición inicial para romper el empate. Cuando el vector esté ordenado se reordena el vector original.

8.22. Escriba una utilidad simple de ordenación, `Ordenar`. El comando `Ordenar` toma como parámetro el nombre de un fichero, que contiene un elemento por línea. Por defecto, las líneas se consideran como cadenas de caracteres y se ordenan mediante el orden lexicográfico habitual (aunque distinguiendo entre mayúsculas y minúsculas). Añada dos opciones: la opción `-c` significa que no se distinga entre mayúsculas y minúsculas, y la opción `-n` significa que las líneas deben ser consideradas como enteros.

8.23. Suponga que tiene K ficheros ordenados. Cada línea contiene un objeto `Linea`, donde `Linea` es un interfaz que implementa el interfaz `Comparable` y especifica un método `obtenerLinea`, que lee la línea dentro del objeto.

a) Escriba un método `leerFichero` que lea los elementos de un solo fichero y los introduzca en una cola creada por el método. Se devolverá una referencia a dicha cola. Observe que los elementos en la cola estarán ordenados. La declaración es

```
Cola leerFichero( String nombreFichero );
```

b) Escriba un método denominado `mezclarDos` que tome dos colas y cree una tercera con el resultado de la mezcla. Debe devolver una referencia a la nueva cola creada, y haber vaciado las dos colas originales. La declaración es:

```
Cola mezclarDos( Cola c1, Cola c2 );
```

c) El algoritmo para producir un fichero ordenado es el siguiente. Declare una cola de colas `losElementos`. Llame a `leerFichero` con cada uno de los K ficheros e inserte los resultados devueltos en `losElementos`. Elimine dos elementos de `losElementos`, mézclelos, e introduzca el resultado en la cola, $K - 1$ veces. El resultado será que `losElementos` contendrá una única cola ordenada que contendrá todos los elementos de los ficheros. Escriba el correspondiente método.

d) Escriba un programa completo que implemente la mezcla de K ficheros ordenados.

e) Muestre que declarar `losElementos` como una pila de colas puede conducir a un mal rendimiento. *Pista*: considere el caso en el cual el primer fichero tiene N elementos y todos los demás tienen un solo elemento.

Bibliografía

La referencia clásica para los algoritmos de ordenación es [5]. Una referencia más reciente, completada con resultados puestos al día, es [3]. El algoritmo de Shellsort apareció por primera vez en [7]. Un estudio empírico de su tiempo de ejecución fue realizado en [8]. Quicksort fue descubierto por Hoare [4]. El artículo también incluye el algoritmo de selección rápida, y detalla muchas de las cuestiones de implementación. Un minucioso estudio del algoritmo de quicksort, incluyendo análisis para la variante con la mediana de tres, aparece en [6]. Una implementación detallada en C que incluye mejoras adicionales puede encontrarse en [1]. La cota inferior $\Omega(N \log N)$ para los algoritmos de ordenación basados en comparaciones se ha tomado de [2]. La presentación de Shellsort se ha adaptado de [9].

1. J. L. Bentley y M. D. McElroy, «Engineering a Sort Function», *Software-Practice and Experience* **23** (1993), 1249-1265.

2. L. R. Ford y S. M. Johnson, «A Tournament Problem», *American Mathematics Monthly* **66** (1959), 387-389.

3. G. H. Gonnet y R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, 2.ª ed., Addison-Wesley, Reading, Mass. (1991).

4. C. A. R. Hoare, «Quicksort», *Computer Journal* **5** (1962), 10-15.

5. D. E. Knuth, *The Art of Computer Programming*, *Volume* 3: *Sorting and Searching*, 2.ª ed., Addison-Wesley, Reading, Mass. (1997).

6. R. Sedgewick, *Quicksort*, Garland Publishing, New York, NY (1978). (Originally presented as the author's Ph.D. thesis, Stanford University, 1975.)

7. D. L. Shell, «A High-Speed Sorting Procedure», *Communications of the ACM* **2** 7 (1959), 30-32.

8. M. A. Weiss, «Empirical Results on the Running Time of Shellsort», *Computer Journal* **34** (1991), 88-91.

9. M. A. Weiss, *Efficient C Programming*: *A Practical Approach*, Prentice-Hall, Englewood Cliffs, NJ (1995).