Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

19

Búsqueda, ordenamiento y Big O

Con sollozos y lágrimas él sorteó los de mayor tamaño…

—Lewis Carroll

Intenta el final, y nunca dejes lugar a dudas; no hay nada tan difícil que no pueda averiguarse mediante la búsqueda.

—Robert Herrick

Está bloqueado en mi memoria, Y tú deberás guardar la llave.

—William Shakespeare

Una ley inmutable en los negocios es que las palabras son palabras, las explicaciones son explicaciones, las promesas son promesas; pero sólo el desempeño es la realidad.

—Harold S. Green

Objetivos

En este capítulo aprenderá:

- A buscar un valor dado en un arreglo usando la búsqueda lineal y la búsqueda binaria.
- A ordenar arreglos usando los algoritmos iterativos de ordenamiento por selección y por inserción.
- A ordenar arreglos usando el algoritmo recursivo de ordenamiento por combinación.
- A determinar la eficiencia de los algoritmos de búsqueda y ordenamiento.
- A introducir la notación Big O para comparar la eficiencia de los algoritmos.

Plan general

19.1 Introducción

La **búsqueda** de datos implica el determinar si un valor (conocido como la **clave de búsqueda**) está presente en los datos y de ser así, hay que encontrar su ubicación. Dos algoritmos populares de búsqueda son la búsqueda lineal simple y la búsqueda binaria, que es más rápida pero a la vez más compleja. El **ordenamiento** coloca los datos en orden ascendente o descendente, con base en una o más **claves de ordenamiento**. Una lista de nombres se podría ordenar en forma alfabética, las cuentas bancarias podrían ordenarse por número de cuenta, los registros de nóminas de empleados podrían ordenarse por número de seguro social, etcétera. En este capítulo se presentan dos algoritmos de ordenamiento simples: el ordenamiento por selección y el ordenamiento por inserción, junto con el ordenamiento por combinación, que es más eficiente pero también más complejo. En la figura 19.1 se sintetizan los algoritmos de búsqueda y ordenamiento que veremos en los ejemplos y ejercicios de este libro.

Observación de ingeniería de software 19.1

En aplicaciones que requieren búsqueda y ordenamiento, use las capacidades predefinidas de la API Collections de Java (capítulo 16). Las técnicas que se presentan en este capítulo se proporcionan para presentar a los estudiantes los conceptos detrás de los algoritmos de búsqueda y ordenamiento; por lo general en los cursos de ciencias computacionales de nivel superior se describen dichos algoritmos con detalle.

Capítulo	Algoritmo	Ubicación
Algoritmos de búsqueda:		
16	Método `binarySearch` de la clase `Collections`	Figura 16.12
19	Búsqueda lineal	Sección 19.2
	Búsqueda binaria	Sección 19.4
	Búsqueda lineal recursiva	Ejercicio 19.8
	Búsqueda binaria recursiva	Ejercicio 19.9

Fig. 19.1 | Los algoritmos de búsqueda y ordenamiento que se cubren en este libro (parte 1 de 2).

Capítulo	Algoritmo	Ubicación
21	Búsqueda lineal en un objeto `List` Búsqueda con árboles binarios	Ejercicio 21.21 Ejercicio 21.23
Algoritmos de ordenamiento:		
16	método `sort` de la clase `Collections` Colección `SortedSet`	Figuras 16.6 a 16.9 Figura 16.17
19	Ordenamiento por selección Ordenamiento por inserción Ordenamiento recursivo por combinación Ordenamiento de burbuja Ordenamiento de cubeta Ordenamiento rápido (*quicksort*) recursivo	Sección 19.6 Sección 19.7 Sección 19.8 Ejercicios 19.5 y 19.6 Ejercicio 19.7 Ejercicio 19.10
21	Ordenamiento con árboles binarios	Sección 21.7

Fig. 19.1 | Los algoritmos de búsqueda y ordenamiento que se cubren en este libro (parte 2 de 2).

19.2 Búsqueda lineal

Buscar un número telefónico, buscar un sitio web a través de un motor de búsqueda y comprobar la definición de una palabra en un diccionario son acciones que implican buscar entre grandes cantidades de datos. En esta sección y en la sección 19.4 hablaremos sobre dos algoritmos de búsqueda comunes: uno que es fácil de programar pero relativamente ineficiente (la búsqueda lineal), y uno que es relativamente eficiente pero más complejo y difícil de programar (la búsqueda binaria).

Algoritmo de búsqueda lineal

El **algoritmo de búsqueda lineal** busca cada elemento de un arreglo en forma secuencial. Si la clave de búsqueda no coincide con un elemento en el arreglo, el algoritmo evalúa cada elemento y cuando se llega al final del arreglo informa al usuario que la clave de búsqueda no está presente. Si la clave de búsqueda se encuentra en el arreglo, el algoritmo evalúa cada elemento hasta encontrar uno que coincida con la clave de búsqueda y devuelve el índice de ese elemento.

Como ejemplo, considere un arreglo que contiene los siguientes valores:

```
34   56   2   10   77   51   93   30   5   52
```

y un programa que busca el número 51. Usando el algoritmo de búsqueda lineal, el programa primero comprueba si el 34 coincide con la clave de búsqueda. Si no es así, el algoritmo comprueba si 56 coincide con la clave de búsqueda. El programa continúa recorriendo el arreglo en forma secuencial y evalúa el 2, luego el 10, después el 77. Cuando el programa evalúa el número 51, que coincide con la clave de búsqueda, devuelve el índice 5 que está en la posición del 51 en el arreglo. Si después de comprobar cada elemento del arreglo el programa determina que la clave de búsqueda no coincide con ningún elemento del arreglo, devuelve un valor centinela (por ejemplo, `-1`).

Implementación de la búsqueda lineal

La clase `ArregloLineal` (figura 19.2) contiene el método `static busquedaLineal` para realizar búsquedas de un arreglo `int` y el método `main` para probar `busquedaLineal`.

```
1  // Fig. 19.2: PruebaBusquedaLineal.java
2  // Busca un elemento en el arreglo, en forma secuencial.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5  import java.util.Scanner;
6
7  public class PruebaBusquedaLineal
8  {
9     // realiza una búsqueda lineal en los datos
10    public static int busquedaLineal(int datos[], int claveBusqueda)
11    {
12       // itera a través del arreglo en forma secuencial
13       for (int indice = 0; indice < datos.length; indice++)
14          if (datos[indice] == claveBusqueda)
15             return indice; // devuelve el índice del entero
16
17       return -1; // no se encontró el entero
18    } // fin del método busquedaLineal
19
20    public static void main(String[] args)
21    {
22       Scanner entrada = new Scanner(System.in);
23       SecureRandom generador = new SecureRandom();
24
25       int[] datos = new int[10]; // crea arreglo
26
27       for (int i = 0; i < datos.length; i++) // llena el arreglo
28          datos[i] = 10 + generador.nextInt(90);
29
30       System.out.printf("%s%n%n", Arrays.toString(datos)); // muestra el arreglo
31
32       // obtiene la entrada del usuario
33       System.out.print("Escriba un valor entero (-1 para terminar): ");
34       int enteroBusqueda = entrada.nextInt();
35
36       // recibe en forma repetida un entero como entrada; -1 termina el programa
37       while (enteroBusqueda != -1)
38       {
39          int posicion = busquedaLineal(datos, enteroBusqueda); // realiza busqueda
40
41          if (posicion == -1) // no se encontró
42             System.out.printf("%d no se encontro%n%n", enteroBusqueda);
43          else // se encontró
44             System.out.printf("%d se encontro en la posicion %d%n%n",
45                enteroBusqueda, posicion);
46
47          // obtiene la entrada del usuario
48          System.out.print("Escriba un valor entero (-1 para terminar): ");
49          enteroBusqueda = entrada.nextInt();
50       }
51    } // fin de main
52 } // fin de la clase PruebaBusquedaLineal
```

Fig. 19.2 | Búsqueda de un elemento en un arreglo, en forma secuencial (parte 1 de 2).

```
[59, 97, 34, 90, 79, 56, 24, 51, 30, 69]

Escriba un valor entero (-1 para terminar): 79
79 se encontro en la posicion 4.

Escriba un valor entero (-1 para terminar): 61
61 no se encontro.

Escriba un valor entero (-1 para terminar): 51
51 se encontro en la posicion 7.

Escriba un valor entero (-1 para terminar): -1
```

Fig. 19.2 | Búsqueda de un elemento en un arreglo, en forma secuencial (parte 2 de 2).

Método `busquedaLineal`

El método `busquedaLineal` (líneas 10 a 18) realiza la búsqueda lineal. Este método recibe como parámetros el arreglo a buscar (`datos`) y la `claveBusqueda`. En las líneas 13 a 15 se itera a través de los elementos en el arreglo `datos`. En la línea 14 se compara cada elemento en el arreglo con `claveBusqueda`. Si los valores son iguales, en la línea 15 se devuelve el índice del elemento. Si hay valores *duplicados* en el arreglo, la búsqueda lineal devuelve el índice del *primer* elemento en el arreglo que coincide con la clave de búsqueda. Si el ciclo termina sin encontrar el valor, en la línea 17 se devuelve `-1`.

El método `main`

El método `main` (líneas 20 a 51) permite al usuario buscar en un arreglo. En las líneas 25 a 28 se crea un arreglo de `10` valores `int` y se llena con valores `int` aleatorios de `10` a `99`. Después, en la línea 30 se muestra en pantalla el contenido del arreglo mediante el método `static toString` de `Arrays`, que devuelve una representación `String` del arreglo con los elementos entre corchetes (`[` y `]`) y separados por comas.

En las líneas 33 y 34 se pide al usuario la clave de búsqueda y se almacena. En la línea 39 se hace una llamada al método `busquedaLineal` para determinar si `enteroBusqueda` está en el arreglo `datos`. Si no está, `busquedaLineal` devuelve `-1` y el programa notifica al usuario (líneas 41 y 42). Si `enteroBusqueda` está en el arreglo, `busquedaLineal` devuelve la posición del elemento, la cual el programa muestra en pantalla en las líneas 44 y 45. En las líneas 48 y 49 se obtiene la siguiente clave de búsqueda del usuario.

19.3 Notación Big O

Todos los algoritmos de búsqueda logran el *mismo* objetivo: encontrar un elemento o elementos que coincidan con una clave de búsqueda dada, si es que existen dichos elementos. Sin embargo, hay varias cosas que diferencian a un algoritmo de otro. *La principal diferencia es la cantidad de esfuerzo que requieren para completar la búsqueda*. Una forma de describir este esfuerzo es mediante la **notación Big O**, la cual indica qué tan duro tendrá que trabajar un algoritmo para resolver un problema. En los algoritmos de búsqueda y ordenamiento, esto depende específicamente de cuántos elementos de datos haya. En este capítulo usaremos Big O para describir los tiempos de ejecución para el peor caso de varios algoritmos de búsqueda y ordenamiento.

19.3.1 Algoritmos O(1)

Suponga que un algoritmo está diseñado para evaluar si el primer elemento de un arreglo es igual al segundo elemento. Si el arreglo tiene 10 elementos, este algoritmo requiere una comparación. Si el arreglo tiene 1000 elementos, sigue requiriendo una comparación. De hecho, el algoritmo es completamente independiente del número de elementos en el arreglo. Se dice que este algoritmo tiene un **tiempo de**

ejecución constante, el cual se representa en la notación Big O como ***O*(1)** y se pronuncia como "orden uno". Un algoritmo que es $O(1)$ no necesariamente requiere sólo de una comparación. $O(1)$ sólo significa que el número de comparaciones es *constante*; no crece conforme aumenta el tamaño del arreglo. Un algoritmo que evalúa si el primer elemento de un arreglo es igual a los siguientes tres elementos sigue siendo $O(1)$, aun cuando requiera tres comparaciones.

19.3.2 Algoritmos O(*n*)

Un algoritmo que evalúa si el primer elemento de un arreglo es igual a *cualquiera* de los demás elementos del arreglo requerirá cuando menos de $n - 1$ comparaciones, en donde n es el número de elementos en el arreglo. Si el arreglo tiene 10 elementos, este algoritmo requiere hasta nueve comparaciones. Si el arreglo tiene 1000 elementos, requiere hasta 999 comparaciones. A medida que n aumenta en tamaño, la parte de la expresión correspondiente a la n "domina", y si le restamos uno no hay consecuencias. Big O está diseñado para resaltar estos términos dominantes e ignorar los términos que pierden importancia a medida que n crece. Por esta razón se dice que un algoritmo que requiere un total de $n - 1$ comparaciones (como el que describimos antes) es ***O*(*n*)**. Se considera que un algoritmo $O(n)$ tiene un **tiempo de ejecución lineal**. A menudo, $O(n)$ significa "en el orden de n", o dicho en forma más simple, "orden n".

19.3.3 Algoritmos O(n^2)

Ahora suponga que tiene un algoritmo que evalúa si *cualquier* elemento de un arreglo se duplica en cualquier otra parte del mismo. El primer elemento debe compararse con todos los demás elementos del arreglo. El segundo elemento debe compararse con todos los demás elementos, excepto con el primero (ya se comparó con éste). El tercer elemento debe compararse con todos los elementos, excepto los primeros dos. Al final, este algoritmo terminará realizando $(n - 1) + (n - 2) + \ldots + 2 + 1$, o $n^2/2 - n/2$ comparaciones. A medida que n aumenta, el término n^2 domina y el término n se vuelve inconsecuente. De nuevo, la notación Big O resalta el término n^2, dejando a $n/2$. Pero como veremos pronto, en la notación Big O los factores constantes se omiten.

Big O se enfoca en la forma en que aumenta el tiempo de ejecución de un algoritmo en relación con el número de elementos procesados. Suponga que un algoritmo requiere n^2 comparaciones. Con cuatro elementos, el algoritmo requiere 16 comparaciones; con ocho elementos, 64 comparaciones. Con este algoritmo, *al duplicar* el número de elementos *se cuadruplica* el número de comparaciones. Considere un algoritmo similar que requiere $n^2/2$ comparaciones. Con cuatro elementos, el algoritmo requiere ocho comparaciones; con ocho elementos, 32 comparaciones. De nuevo, *al duplicar* el número de elementos *se cuadruplica* el número de comparaciones. Ambos algoritmos aumentan como el cuadrado de n, por lo que Big O ignora la constante y ambos algoritmos se consideran como ***O*(n^2)**, lo cual se conoce como **tiempo de ejecución cuadrático** y se pronuncia como "en el orden de n al cuadrado", o dicho en forma más simple, "orden n al cuadrado".

Cuando n es pequeña, los algoritmos $O(n^2)$ (que se ejecutan en las computadoras personales de la actualidad) no afectan el rendimiento en forma considerable. Pero a medida que n aumenta, se empieza a notar la reducción en el rendimiento. Un algoritmo $O(n^2)$ que se ejecute en un arreglo de un millón de elementos requeriría un billón de "operaciones" (en donde cada una requeriría en realidad varias instrucciones de máquina para ejecutarse). Esto podría tardar varias horas. Un arreglo de mil millones de elementos requeriría un trillón de operaciones, ¡un número tan grande que el algoritmo tardaría décadas! Por desgracia, los algoritmos $O(n^2)$ son fáciles de escribir, como veremos en este capítulo. También veremos algoritmos con medidas de Big O más favorables. Estos algoritmos eficientes comúnmente requieren un poco más de astucia y trabajo para crearlos, pero su rendimiento superior bien vale la pena el esfuerzo adicional, en especial a medida que n aumenta y los algoritmos se combinan en programas más grandes.

19.3.4 Big O de la búsqueda lineal

El algoritmo de búsqueda lineal se ejecuta en un tiempo $O(n)$. El peor caso en este algoritmo es cuando se debe comprobar cada elemento para determinar si el elemento que se busca existe en el arreglo. Si el tamaño del arreglo *se duplica*, el número de comparaciones que el algoritmo debe realizar también *se duplica*. La búsqueda lineal puede proporcionar un rendimiento sorprendente si el elemento que coincide con la clave de búsqueda se encuentra en (o cerca de) la parte frontal del arreglo. Pero buscamos algoritmos que tengan un buen desempeño en promedio en *todas* las búsquedas, incluyendo aquellas en las que el elemento que coincide con la clave de búsqueda se encuentra cerca del final del arreglo.

La búsqueda lineal es fácil de programar pero puede ser lenta si se le compara con otros algoritmos de búsqueda. Si un programa necesita realizar muchas búsquedas en arreglos grandes, puede ser mejor implementar un algoritmo más eficiente, como la búsqueda binaria, que presentaremos en la siguiente sección.

Tip de rendimiento 19.1

Algunas veces los algoritmos más simples tienen un desempeño pobre. Su virtud es que son fáciles de programar, probar y depurar. En ocasiones se requieren algoritmos más complejos para obtener el máximo rendimiento.

19.4 Búsqueda binaria

El **algoritmo de búsqueda binaria** es más eficiente que el algoritmo de búsqueda lineal, pero requiere que el arreglo se ordene. La primera iteración de este algoritmo evalúa el elemento *medio* del arreglo. Si éste coincide con la clave de búsqueda, el algoritmo termina. Suponiendo que el arreglo se ordene en forma *ascendente*, entonces si la clave de búsqueda es *menor que* el elemento de en medio, no puede coincidir con ningún elemento en la segunda mitad del arreglo, y el algoritmo continúa sólo con la primera mitad (es decir, el primer elemento hasta, pero sin incluir, el elemento de en medio). Si la clave de búsqueda es *mayor que* el elemento de en medio, no puede coincidir con ninguno de los elementos de la primera mitad del arreglo, y el algoritmo continúa sólo con la segunda mitad del arreglo (es decir, desde el elemento *después* del elemento de en medio, hasta el último elemento). Cada iteración evalúa el valor medio de la porción restante del arreglo. Si la clave de búsqueda no coincide con el elemento, el algoritmo elimina la mitad de los elementos restantes. Para terminar, el algoritmo encuentra un elemento que coincide con la clave de búsqueda o reduce el subarreglo hasta un tamaño de cero.

Ejemplo

Como ejemplo, considere el siguiente arreglo ordenado de 15 elementos:

```
2  3  5  10  27  30  34  51  56  65  77  81  82  93  99
```

y una clave de búsqueda de 65. Un programa que implemente el algoritmo de búsqueda binaria primero comprobaría si el 51 es la clave de búsqueda (ya que 51 es el elemento *de en medio* del arreglo). La clave de búsqueda (65) es mayor que 51, por lo que este número se descarta junto con la primera mitad del arreglo (todos los elementos menores que 51). Con esto obtenemos lo siguiente:

```
56  65  77  81  82  93  99
```

A continuación, el algoritmo comprueba si 81 (el elemento de en medio del resto del arreglo) coincide con la clave de búsqueda. La clave de búsqueda (65) es menor que 81, por lo que se descarta este número junto con los elementos mayores de 81. Después de sólo dos pruebas, el algoritmo ha reducido el número de valores a comprobar a tres (56, 65 y 77). Después el algoritmo comprueba el 65 (que sin

duda coincide con la clave de búsqueda), y devuelve el índice del elemento del arreglo que contiene el 65. Este algoritmo sólo requirió tres comparaciones para determinar si la clave de búsqueda coincidió con un elemento del arreglo. Un algoritmo de búsqueda lineal habría requerido 10 comparaciones. [*Nota*: en este ejemplo optamos por usar un arreglo con 15 elementos, para que siempre haya un elemento obvio en medio del arreglo. Con un número par de elementos, la parte media del arreglo se encuentra entre dos elementos. Implementamos el algoritmo para elegir el mayor de esos dos elementos].

19.4.1 Implementación de la búsqueda binaria

La clase `PruebaBusquedaBinaria` (figura 19.3) contiene:

- El método `static busquedaBinaria` para buscar una clave especificada en un arreglo `int`
- El método `static elementosRestantes` para mostrar los elementos restantes en el arreglo en donde se realiza la búsqueda, y
- `main` para probar el método `busquedaBinaria`.

El método `main` (líneas 57 a 90) es casi idéntico a `main` en la figura 19.2. En este programa creamos un arreglo de 15 elementos (línea 62) y en la línea 67 se hace una llamada al método `static sort` de la clase `Arrays` para ordenar los elementos del arreglo `datos` en un arreglo por orden ascendente (de manera predeterminada). Recuerde que el algoritmo de búsqueda binaria sólo funciona en un arreglo ordenado. La primera línea de salida de este programa muestra el arreglo ordenado de valores `int`. Cuando el usuario instruye al programa para que busque el número `18`, el programa prueba primero el elemento medio, que es `57` (como lo indica el `*`). La clave de búsqueda es menor que `57`, por lo que el programa elimina la segunda mitad del arreglo y prueba el elemento medio de la primera mitad. La clave de búsqueda es menor que `36`, por lo que el programa elimina la segunda mitad del arreglo y sólo quedan tres elementos. Por último, el programa revisa el `18` (que coincide con la clave de búsqueda) y devuelve el índice `1`.

```
1  // Fig. 19.3: PruebaBusquedaBinaria.java
2  // Usa la búsqueda binaria para localizar un elemento en un arreglo.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5  import java.util.Scanner;
6
7  public class PruebaBusquedaBinaria
8  {
9     // realiza una búsqueda binaria en los datos
10    public static int busquedaBinaria(int[] datos, int clave)
11    {
12       int inferior = 0; // extremo inferior del área de búsqueda
13       int superior = data.length - 1; // extremo superior del área de búsqueda
14       int medio = (inferior + superior + 1) / 2; // elemento medio
15       int ubicacion = -1; // devuelve el valor; -1 si no lo encontró
16
17       do // ciclo para buscar un elemento
18       {
19          // imprime el resto de los elementos del arreglo
20          System.out.print(elementosRestantes(datos, inferior, superior));
21
```

Fig. 19.3 | Uso de la búsqueda binaria para localizar un elemento en un arreglo (el símbolo * en la salida marca el elemento medio) (parte 1 de 3).

```
22          // imprime espacios para alineación
23          for (int i = 0; i < medio; i++)
24             System.out.print("   ");
25          System.out.println(" * "); // indica el elemento medio actual
26
27          // si el elemento se encuentra en medio
28          if (clave == datos[medio])
29             ubicacion = medio; // la ubicación es el elemento medio actual
30          else if (clave < datos[medio]) // el elemento medio es demasiado alto
31             superior = medio - 1; // elimina la mitad superior
32          else  // el elemento medio es demasiado bajo
33             inferior = medio + 1; // elimina la mitad inferior
34
35          medio = (inferior + superior + 1) / 2; // recalcula el elemento medio
36       } while ((inferior <= superior) && (ubicacion == -1));
37
38       return ubicacion; // devuelve la ubicación de la clave de búsqueda
39    } // fin del método busquedaBinaria
40
41    // método para imprimir ciertos valores en el arreglo
42    private static String elementosRestantes(int[] datos, int inferior, int superior)
43    {
44       StringBuilder temporal = new StringBuilder();
45
46       // imprime espacios para alineación
47       for (int i = 0; i < inferior; i++)
48          temporal.append("   ");
49
50       // imprime los elementos que quedan en el arreglo
51       for (int i = inferior; i <= superior; i++)
52          temporal.append(datos[i] + " ");
53
54       return String.format("%s%n", temporal);
55    } // fin del método elementosRestantes
56
57    public static void main(String[] args)
58    {
59       Scanner entrada = new Scanner(System.in);
60       SecureRandom generador = new SecureRandom();
61
62       int[] datos = new int[15]; // crea el arreglo
63
64       for (int i = 0; i < datos.length; i++) // llena el arreglo
65          datos[i] = 10 + generador.nextInt(90);
66
67       Arrays.sort(datos); // busquedaBinaria requiere un arreglo
68       System.out.printf("%s%n%n", Arrays.toString(datos)); // muestra el arreglo
69
70       // obtiene la entrada del usuario
71       System.out.print("Escriba un valor entero (-1 para salir): ");
72       int enteroABuscar = entrada.nextInt();
73
```

Fig. 19.3 | Uso de la búsqueda binaria para localizar un elemento en un arreglo (el símbolo * en la salida marca el elemento medio) (parte 2 de 3).

```
74         // recibe un entero como entrada en forma repetida; -1 termina el programa
75         while (enteroABuscar != -1)
76         {
77            // realiza la búsqueda
78            int posicion = busquedaBinaria(datos, enteroABuscar);
79
80            if (posicion == -1) // no lo encontró
81               System.out.printf("%d no se encontro%n%n", enteroABuscar);
82            else // lo encontró
83               System.out.printf("%d se encontro en la posicion %d%n%n",
84                  enteroABuscar, posicion);
85
86            // obtiene entrada del usuario
87            System.out.print("Escriba un valor entero (-1 para salir): ");
88            enteroABuscar = entrada.nextInt();
89         }
90      } // fin de main
91   } // fin de la clase PruebaBusquedaBinaria
```

```
[25, 31, 46, 48, 52, 58, 59, 59, 71, 72, 72, 79, 83, 86, 98]

Escriba un valor entero (-1 para salir): 31
25 31 46 48 52 58 59 59 71 72 72 79 83 86 98
                     *
25 31 46 48 52 58 59
          *
25 31 46
    *
31 se encontro en la posicion 1

Escriba un valor entero (-1 para salir): 83
25 31 46 48 52 58 59 59 71 72 72 79 83 86 98
                     *
                       71 72 72 79 83 86 98
                                 *
                                   83 86 98
                                       *
                                   83
                                    *
83 se encontro en la posicion 12

Escriba un valor entero (-1 para salir): 73
25 31 46 48 52 58 59 59 71 72 72 79 83 86 98
                     *
                       71 72 72 79 83 86 98
                                 *
                       71 72 72
                            *
                               72
                                *
73 no se encontro

Escriba un valor entero (-1 para salir): -1
```

Fig. 19.3 | Uso de la búsqueda binaria para localizar un elemento en un arreglo (el símbolo * en la salida marca el elemento medio) (parte 3 de 3).

En las líneas 10 a 39 se declara el método `busquedaBinaria` que recibe como parámetros el arreglo en donde se va a realizar la búsqueda (`datos`) y la clave de búsqueda (`clave`). En las líneas 12 a 14 se calcula el índice del extremo `inferior`, el índice del extremo `superior` y el índice `medio` de la parte del arreglo en donde el programa está buscando en ese momento. Al principio del método, el extremo `inferior` es `0`, el extremo `superior` es la longitud del arreglo menos `1` y `medio` es el promedio de estos dos valores. En la línea 15 se inicializa la `posicion` del elemento con `-1`: el valor que se devolverá si no se encuentra la `clave`. En las líneas 17 a 36 se itera hasta que `inferior` sea mayor que `superior` (esto ocurre cuando no se encuentra la `clave`) o cuando `posicion` no sea igual a `-1` (lo que indica que se encontró la `clave`). En la línea 28 se prueba si el valor en el elemento `medio` es igual a la `clave`. De ser así, en la línea 29 se asigna `medio` a `posicion`, el ciclo termina y se devuelve `posicion` al método que hizo la llamada. Cada iteración del ciclo prueba un solo valor (línea 28) y *elimina la mitad del resto de los valores en el arreglo* (línea 31 o 33) si el valor no es la `clave`.

19.4.2 Eficiencia de la búsqueda binaria

En el peor de los casos, el proceso de buscar en un arreglo *ordenado* de 1023 elementos *sólo requiere 10 comparaciones* cuando se utiliza una búsqueda binaria. Al dividir 1023 entre 2 en forma repetida (ya que después de cada comparación podemos eliminar la mitad del arreglo) y redondear (porque también eliminamos el elemento medio), se producen los valores 511, 255, 127, 63, 31, 15, 7, 3, 1 y 0. El número 1023 ($2^{10} - 1$) se divide entre 2 sólo 10 veces para obtener el valor 0, lo cual indica que no hay más elementos para probar. La división entre 2 equivale a una comparación en el algoritmo de búsqueda binaria. Por ende, un arreglo de 1,048,575 ($2^{20} - 1$) elementos requiere un *máximo de 20 comparaciones* para encontrar la clave, y un arreglo de más de mil millones de elementos requiere un *máximo de 30 comparaciones* para encontrar la clave. Ésta es una enorme mejora en el rendimiento, en comparación con la búsqueda lineal. Para un arreglo de mil millones de elementos, ésta es una diferencia entre un promedio de 500 millones de comparaciones para la búsqueda lineal, ¡y un máximo de sólo 30 comparaciones para la búsqueda binaria! El número máximo de comparaciones necesarias para la búsqueda binaria de cualquier arreglo ordenado es el exponente de la primera potencia de 2 mayor que el número de elementos en el arreglo, que se representa como $\log_2 n$. Todos los logaritmos crecen aproximadamente a la misma proporción, por lo que en notación Big O se puede omitir la base. Esto produce un valor Big O de ***O*(log *n*)** para una búsqueda binaria, que también se conoce como **tiempo de ejecución logarítmico** y se pronuncia como "orden log n".

19.5 Algoritmos de ordenamiento

El ordenamiento de datos (es decir, colocar los datos en cierto orden específico, como ascendente o descendente) es una de las aplicaciones computacionales más importantes. Un banco ordena todos los cheques por número de cuenta, de manera que pueda preparar instrucciones bancarias individuales al final de cada mes. Las compañías telefónicas ordenan sus listas de cuentas por apellido paterno y luego por primer nombre, para facilitar el proceso de buscar números telefónicos. Casi cualquier organización debe ordenar datos, y a menudo cantidades masivas de ellos. El ordenamiento de datos es un problema intrigante, que requiere un uso intensivo de la computadora y ha atraído un enorme esfuerzo de investigación.

Un punto importante a comprender acerca del ordenamiento es que el resultado final (el arreglo ordenado) será el *mismo*, sin importar qué algoritmo se utilice para ordenar el arreglo. La elección del algoritmo sólo afecta al tiempo de ejecución y el uso que haga el programa de la memoria. En el resto del capítulo se introducen tres algoritmos de ordenamiento comunes. Los primeros dos (*ordenamiento por selección* y *ordenamiento por inserción*) son simples de programar, pero *ineficientes*. El último algoritmo (*ordenamiento por combinación*) es mucho más *rápido* que el ordenamiento por selección y el ordenamiento por inserción, pero *más difícil de programar*. Nos enfocaremos en ordenar arreglos de datos de tipos primitivos, es decir, valores `int`. Es posible ordenar arreglos de objetos de clases también,

como demostramos en la sección 16.7,1 mediante el uso de las herramientas de ordenamiento integradas de la API Collections.

19.6 Ordenamiento por selección

El **ordenamiento por selección** es un algoritmo de ordenamiento simple, pero ineficiente. Si se va a ordenar en forma ascendente, en la primera iteración del algoritmo se selecciona el elemento *más pequeño* en el arreglo y se intercambia con el primer elemento. En la segunda iteración se selecciona el *segundo elemento más pequeño* (que viene siendo el elemento más pequeño de los elementos restantes) y se intercambia con el segundo elemento. El algoritmo continúa hasta que en la última iteración se selecciona el *segundo elemento más grande* y se intercambia con el índice del segundo al último, dejando el elemento más grande en el último índice. Después de la *i*-ésima iteración, los *i* elementos más pequeños del arreglo se ordenarán en forma ascendente, en los primeros *i* elementos del arreglo.

Como ejemplo, considere el siguiente arreglo:

```
34   56   4   10   77   51   93   30   5   52
```

Un programa que implemente el ordenamiento por selección primero determinará el elemento más pequeño (4) de este arreglo, que está contenido en el índice 2. El programa intercambia 4 con 34, dando el siguiente resultado:

```
4   56   34   10   77   51   93   30   5   52
```

Después el programa determina el valor más pequeño del resto de los elementos (todos los elementos excepto el 4), que es 5 y está contenido en el índice 8. El programa intercambia el 5 con el 56, dando el siguiente resultado:

```
4   5   34   10   77   51   93   30   56   52
```

En la tercera iteración, el programa determina el siguiente valor más pequeño (10) y lo intercambia con el 34.

```
4   5   10   34   77   51   93   30   56   52
```

El proceso continúa hasta que el arreglo quede completamente ordenado.

```
4   5   10   30   34   51   52   56   77   93
```

Después de la primera iteración, el elemento más pequeño está en la primera posición. Después de la segunda iteración los dos elementos más pequeños están en orden, en las primeras dos posiciones. Después de la tercera iteración los tres elementos más pequeños están en orden, en las primeras tres posiciones.

19.6.1 Implementación del ordenamiento por selección

La clase `PruebaOrdenamientoSeleccion` (figura 19.4) contiene:

- El método `static ordenamientoSeleccion` para ordenar un arreglo `int` mediante el algoritmo de ordenamiento por selección
- El método `static swap` para intercambiar los valores de dos elementos del arreglo
- El método static `imprimirPasada` para mostrar el contenido del arreglo después de cada pasada, y
- `main` para probar el método `ordenamientoSeleccion`.

Como en los ejemplos de búsqueda, `main` (líneas 57 a 72) crea un arreglo de valores `int` llamado `datos` y lo llena con valores `int` aleatorios en el rango de `10` a `99`. En la línea 68 se prueba el método `ordenamientoSeleccion`.

```
1  // Fig. 19.4: PruebaOrdenamientoSeleccion.java
2  // Cómo ordenar un arreglo mediante el ordenamiento por selección.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class PruebaOrdenamientoSeleccion
7  {
8     // ordena el arreglo usando el ordenamiento por selección
9     public static void ordenamientoSeleccion(int[] datos)
10    {
11       // itera a través de datos.length - 1 elementos
12       for (int i = 0; i < datos.length - 1; i++)
13       {
14          int masPequenio = i; // primer índice del resto del arreglo
15
16          // itera para buscar el índice del elemento más pequeño
17          for (int indice = i + 1; indice < datos.length; indice++)
18             if (datos[indice] < datos[masPequenio])
19                masPequenio = indice;
20
            intercambiar(datos, i, masPequenio); // intercambia el elemento más
21                                                  pequeño en la posición
22          imprimirPasada(i + 1, masPequenio); // imprime la pasada del algoritmo
23       }
24    } // fin del método ordenamientoSeleccion
25
26    // método ayudante para intercambiar los valores de dos elementos
27    private static void intercambiar(int[] datos, int primero, int segundo)
28    {
29       int temporal = datos[primero]; // almacena primero en temporal
30       datos[primero] = datos[segundo]; // sustituye primero con segundo
31       datos[segundo] = temporal; // coloca temporal en segundo
32    }
33
34    // imprime una pasada del algoritmo
35    private static void imprimirPasada(int[] datos, int pasada, int indice)
36    {
37       System.out.printf("despues de pasada %2d: ", pasada);
38
39       // imprime elementos hasta el elemento seleccionado
40       for (int i = 0; i < indice; i++)
41          System.out.printf("%d  ", datos[i]);
42
43       System.out.printf("%d* ", datos[indice]); // indica intercambio
44
45       // termina de imprimir el arreglo en pantalla
46       for (int i = indice + 1; i < datos.length; i++)
47          System.out.printf("%d  ", datos[i]);
48
49       System.out.printf("%n               "); // para alineación
50
51       // indica la cantidad del arreglo que está ordenada
52       for (int j = 0; j < pasada; j++)
53          System.out.print("--  ");
```

Fig. 19.4 | Cómo ordenar un arreglo mediante el ordenamiento por selección (parte 1 de 2).

```
54          System.out.println();
55       }
56
57       public static void main(String[] args)
58       {
59          SecureRandom generador = new SecureRandom();
60
61          int[] datos = new int[10]; // crea el arreglo
62
63          for (int i = 0; i < datos.length; i++) // llena el arreglo
64             datos[i] = 10 + generador.nextInt(90);
65
66          System.out.printf("Arreglo desordenado:%n%s%n%n",
67             Arrays.toString(datos)); // muestra el arreglo
68          ordenamientoSeleccion(datos); // ordena el arreglo
69
70          System.out.printf("Arreglo ordenado:%n%s%n%n",
71             Arrays.toString(data)); // muestra el arreglo
72       }
73    } // fin de la clase PruebaOrdenamientoSeleccion
```

```
Arreglo desordenado:
[58, 21, 96, 58, 42, 54, 16, 21, 14, 30]

despues de pasada  1: 14  21  96  58  42  54  16  21  58* 30
                      --
despues de pasada  2: 14  16  96  58  42  54  21* 21  58  30
                      --  --
despues de pasada  3: 14  16  21  58  42  54  96* 21  58  30
                      --  --  --
despues de pasada  4: 14  16  21  21  42  54  96  58* 58  30
                      --  --  --  --
despues de pasada  5: 14  16  21  21  30  54  96  58  58  42*
                      --  --  --  --  --
despues de pasada  6: 14  16  21  21  30  42  96  58  58  54*
                      --  --  --  --  --  --
despues de pasada  7: 14  16  21  21  30  42  54  58  58  96*
                      --  --  --  --  --  --  --
despues de pasada  8: 14  16  21  21  30  42  54  58* 58  96
                      --  --  --  --  --  --  --  --
despues de pasada  9: 14  16  21  21  30  42  54  58  58* 96
                      --  --  --  --  --  --  --  --  --
Arreglo ordenado:
[14, 16, 21, 21, 30, 42, 54, 58, 58, 96]
```

Fig. 19.4 | Cómo ordenar un arreglo mediante el ordenamiento por selección (parte 2 de 2).

Métodos `ordenamientoSeleccion` *e* `intercambiar`

En las líneas 9 a 24 se declara el método `ordenamientoSeleccion`. En las líneas 12 a 23 se itera `datos.length - 1` veces. En la línea 14 se declara e inicializa (con el índice `i` actual) la variable `masPequenio`, que almacena el índice del elemento más pequeño en el arreglo restante. En las líneas 17 a 19 se itera a través del resto de los elementos en el arreglo. Para cada uno de estos elementos, en la línea 18 se compara su valor con el valor del elemento más pequeño. Si el elemento actual es menor que el elemento más pequeño, en la línea 19 se asigna el índice del elemento actual a `masPequenio`. Cuando

termine este ciclo, `masPequenio` contendrá el índice del elemento más pequeño en el resto del arreglo. En la línea 21 se hace una llamada al método `intercambiar` (líneas 27 a 32) para colocar el elemento restante más pequeño en la siguiente posición ordenada en el arreglo.

El método `imprimirPasada`

En la salida del método `imprimirPasada` se utilizan guiones cortos (líneas 52 y 53) para indicar la porción del arreglo que se ordenó después de cada pasada. Se coloca un asterisco enseguida de la posición del elemento que se intercambió con el elemento más pequeño en esa pasada. En cada pasada, el elemento que sigue al asterisco (especificado en la línea 43) y el elemento por encima del conjunto de guiones cortos del extremo derecho fueron los dos valores que se intercambiaron.

19.6.2 Eficiencia del ordenamiento por selección

El algoritmo de ordenamiento por selección se ejecuta en un tiempo igual a $O(n^2)$. El método `ordenamientoSeleccion` (líneas 9 a 24) contiene dos ciclos `for`. El ciclo `for` exterior (líneas 12 a 23) itera a través de los primeros $n - 1$ elementos en el arreglo, intercambiando el elemento más pequeño restante a su posición ordenada. El ciclo interior (líneas 17 a 19) itera a través de cada elemento en el arreglo restante, buscando el elemento más pequeño. Este ciclo se ejecuta $n - 1$ veces durante la primera iteración del ciclo exterior, $n - 2$ veces durante la segunda iteración, después $n - 3, \ldots, 3, 2, 1$. Este ciclo interior iterará un total de $n(n - 1)/2$ o $(n^2 - n)/2$. En notación Big O, los términos más pequeños se eliminan y las constantes se ignoran, lo cual nos deja un valor Big O final de $O(n^2)$.

19.7 Ordenamiento por inserción

El **ordenamiento por inserción** es otro algoritmo de ordenamiento *simple pero ineficiente*. En la primera iteración de este algoritmo se toma el *segundo elemento* en el arreglo y si es *menor que el primero, se intercambian*. En la segunda iteración se analiza el tercer elemento y se inserta en la posición correcta con respecto a los primeros dos elementos, de manera que los tres elementos estén ordenados. En la *i*-ésima iteración de este algoritmo, los primeros *i* elementos en el arreglo original estarán ordenados.

Considere como ejemplo el siguiente arreglo, el cual es idéntico al que se utiliza en las explicaciones sobre el ordenamiento por selección y el ordenamiento por combinación.

```
34   56   4   10   77   51   93   30   5   52
```

Un programa que implemente el algoritmo de ordenamiento por inserción primero analizará los primeros dos elementos del arreglo, 34 y 56. Éstos ya se encuentran ordenados por lo que el programa continúa (si estuvieran desordenados, el programa los intercambiaría).

En la siguiente iteración, el programa analiza el tercer valor, 4. Este valor es menor que 56, por lo que el programa almacena el 4 en una variable temporal y mueve el 56 un elemento a la derecha. Después, el programa comprueba y determina que 4 es menor que 34, por lo que mueve el 34 un elemento a la derecha. Ahora el programa ha llegado al principio del arreglo, por lo que coloca el 4 en el elemento cero. Entonces, el arreglo es ahora

```
4   34   56   10   77   51   93   30   5   52
```

En la siguiente iteración, el programa almacena el valor 10 en una variable temporal. Después el programa compara el 10 con el 56 y mueve el 56 un elemento a la derecha, ya que es mayor que 10. Luego, el programa compara 10 con 34 y mueve el 34 un elemento a la derecha. Cuando el programa compara el 10 con el 4 observa que el primero es mayor que el segundo, por lo cual coloca el 10 en el elemento 1. Ahora el arreglo es

```
4   10   34   56   77   51   93   30   5   52
```

Utilizando este algoritmo, en la *i*-ésima iteración, los primeros *i* elementos del arreglo original están ordenados, pero tal vez no se encuentren en sus posiciones finales; puede haber valores más pequeños en posiciones más adelante en el arreglo.

19.7.1 Implementación del ordenamiento por inserción

La clase `PruebaOrdenamientoInsercion` (figura 19.5) contiene:

- El método `static ordenamientoInsercion` para ordenar valores `int` mediante el algoritmo de ordenamiento por inserción
- El método `static imprimirPasada` para mostrar el contenido del arreglo después de cada pasada, y
- El método `main` para probar el método `ordenamientoInsercion`.

El método `main` (líneas 53 a 68) es idéntico a `main` de la figura 19.4, excepto que en la línea 64 se hace una llamada al método `ordenamientoInsercion`.

```
1  // Fig. 19.5: PruebaOrdenamientoInsercion.java
2  // Cómo ordenar un arreglo mediante el ordenamiento por inserción.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class PruebaOrdenamientoInsercion
7  {
8     // ordena el arreglo usando el ordenamiento por inserción
9     public static void ordenamientoInsercion(int[] datos)
10    {
11       // itera a través de datos.length - 1 elementos
12       for (int siguiente = 1; siguiente < datos.length; siguiente++)
13       {
14          int insercion = datos[siguiente]; // valor a insertar
15          int moverElemento = siguiente; // posición en donde se va a colocar el elemento
16
17          // busca un lugar para colocar el elemento actual
18          while (moverElemento > 0 && datos[moverElemento - 1] > insercion)
19          {
20             // desplaza el elemento una posición a la derecha
21             datos[moverElemento] = datos[moverElemento - 1];
22             moverElemento--;
23          }
24
25          datos[moverElemento] = insercion; // coloca el elemento insertado
26          imprimirPasada(datos, siguiente, moverElemento); // imprime la pasada
                                                             del algoritmo
27       }
28    }
29
30    // imprime una pasada del algoritmo
31    public static void imprimirPasada(int[] datos, int pasada, int indice)
32    {
33       System.out.printf("despues de pasada %2d: ", pasada);
34
35       // imprime los elementos hasta el elemento intercambiado
36       for (int i = 0; i < indice; i++)
37          System.out.printf("%d  ", datos[i]);
```

Fig. 19.5 | Cómo ordenar un arreglo mediante el ordenamiento por inserción (parte 1 de 3).

```
38
39          System.out.printf("%d* ", datos[indice]); // indica intercambio
40
41          // termina de imprimir el arreglo en pantalla
42          for (int i = indice + 1; i < datos.length; i++)
43             System.out.printf("%d  ", datos[i]);
44
45          System.out.printf("%n               "); // para alineación
46
47          // indica la cantidad del arreglo que está ordenado
48          for(int i = 0; i <= pasada; i++)
49             System.out.print("--  ");
50          System.out.println();
51       }
52
53       public static void main(String[] args)
54       {
55          SecureRandom generador = new SecureRandom();
56
57          int[] datos = new int[10]; // crea el arreglo
58
59          for (int i = 0; i < datos.length; i++) // llena el arreglo
60             datos[i] = 10 + generador.nextInt(90);
61
62          System.out.printf("Arreglo desordenado:%n%s%n%n",
63             Arrays.toString(datos)); // muestra el arreglo
64          ordenamientoInsercion(datos); // ordena el arreglo
65
66          System.out.printf("Arreglo ordenado:%n%s%n%n",
67             Arrays.toString(datos)); // muestra el arreglo
68       }
69    } // fin de la clase OrdenamientoInsercion
```

```
Arreglo desordenado:
[61, 14, 55, 32, 38, 18, 72, 76, 98, 56]

despues de pasada  1: 14* 61  55  32  38  18  72  76  98  56
                      --  --
despues de pasada  2: 14  55* 61  32  38  18  72  76  98  56
                      --  --  --
despues de pasada  3: 14  32* 55  61  38  18  72  76  98  56
                      --  --  --  --
despues de pasada  4: 14  32  38* 55  61  18  72  76  98  56
                       --  --  --  --  --
despues de pasada  5: 14  18* 32  38  55  61  72  76  98  56
                      --  --  --  --  --  --
despues de pasada  6: 14  18  32  38  55  61  72* 76  98  56
                      --  --  --  --  --  --  --
despues de pasada  7: 14  18  32  38  55  61  72  76* 98  56
                      --  --  --  --  --  --  --  --
despues de pasada  8: 14  18  32  38  55  61  72  76  98* 56
                      --  --  --  --  --  --  --  --  --
```

Fig. 19.5 | Cómo ordenar un arreglo mediante el ordenamiento por inserción (parte 2 de 3).

```
despues de pasada  9: 14  18  32  38  55  56* 61  72  76  98
                      --  --  --  --  --  --  --  --  --  --
Arreglo ordenado:
[14, 18, 32, 38, 55, 56, 61, 72, 76, 98]
```

Fig. 19.5 | Cómo ordenar un arreglo mediante el ordenamiento por inserción (parte 3 de 3).

El método `ordenamientoInsercion`

En las líneas 9 a 28 se declara el método `ordenamientoInsercion`. En las líneas 12 a 27 se itera a través de `datos.length - 1` elementos en el arreglo. En cada iteración, en la línea 14 se declara e inicializa la variable `inserciones`, que contiene el valor del elemento que se insertará en la parte ordenada del arreglo. En la línea 15 se declara e inicializa la variable `moverElemento` que lleva la cuenta de la posición en la que se insertará el elemento. En las líneas 18 a 23 se itera para localizar la posición correcta en la que debe insertarse el elemento. El ciclo terminará ya sea cuando el programa llegue a la parte frontal del arreglo o cuando llegue a un elemento que sea menor que el valor a insertar. En la línea 21 se mueve un elemento a la derecha y en la línea 22 se decrementa la posición en la que se insertará el siguiente elemento. Una vez que termina el ciclo, en la línea 25 se inserta el elemento en su posición.

El método `imprimirPasada`

En la salida del método `imprimirPasada` (líneas 31 a 51) se utilizan guiones cortos para indicar la parte del arreglo que se ordena después de cada pasada. Se coloca un asterisco enseguida del elemento que se insertó en su posición en esa pasada.

19.7.2 Eficiencia del ordenamiento por inserción

El algoritmo de ordenamiento por inserción también se ejecuta en un tiempo igual a $O(n^2)$. Al igual que el ordenamiento por selección, la implementación del ordenamiento por inserción (líneas 9 a 28) contiene dos ciclos. El ciclo `for` (líneas 12 a 27) itera `datos.length - 1` veces, insertando un elemento en la posición apropiada en los elementos ordenados hasta ahora. Para los fines de esta aplicación, `datos.length - 1` es equivalente a $n - 1$ (ya que `datos.length` es el tamaño del arreglo). El ciclo `while` (líneas 18 a 23) itera a través de los elementos anteriores en el arreglo. En el peor de los casos, el ciclo while requerirá $n - 1$ comparaciones. Cada ciclo individual se ejecuta en un tiempo $O(n)$. En notación Big O, los ciclos anidados indican que debemos *multiplicar* el número de comparaciones. Para cada iteración de un ciclo exterior habrá cierto número de iteraciones en el ciclo interior. En este algoritmo, para cada $O(n)$ iteraciones del ciclo exterior habrá $O(n)$ iteraciones del ciclo interior. Al multiplicar estos valores se produce un valor Big O de $O(n^2)$.

19.8 Ordenamiento por combinación

El **ordenamiento por combinación** es un algoritmo de ordenamiento *eficiente* pero conceptualmente *más complejo* que los ordenamientos por selección y por inserción. Para ordenar un arreglo, el algoritmo de ordenamiento por combinación lo *divide* en dos subarreglos de igual tamaño, *ordena* cada subarreglo y después los *combina* en un arreglo más grande. Con un número impar de elementos, el algoritmo crea los dos subarreglos de tal forma que uno tenga más elementos que el otro.

En este ejemplo, la implementación del ordenamiento por combinación es recursiva. El caso base es un arreglo con un elemento que desde luego está ordenado, por lo que el ordenamiento por combinación regresa de inmediato en este caso. El paso recursivo divide el arreglo en dos piezas de un tamaño aproximadamente igual, las ordena en forma recursiva y después combina los dos arreglos ordenados en un arreglo ordenado de mayor tamaño.

Suponga que el algoritmo ya ha combinado arreglos más pequeños para crear los arreglos ordenados A:

```
    4   10   34   56   77
```

y B:

```
    5   30   51   52   93
```

El ordenamiento por combinación combina estos dos arreglos en un arreglo ordenado de mayor tamaño. El elemento más pequeño en A es 4 (que se encuentra en el índice cero de A). El elemento más pequeño en B es 5 (que se encuentra en el índice cero de B). Para poder determinar el elemento más pequeño en el arreglo más grande, el algoritmo compara 4 y 5. El valor de A es más pequeño, por lo que el 4 se convierte en el primer elemento del arreglo combinado. El algoritmo continúa, para lo cual compara 10 (el segundo elemento en A) con 5 (el primer elemento en B). El valor de B es más pequeño, por lo que 5 se convierte en el segundo elemento del arreglo más grande. El algoritmo continúa comparando 10 con 30, en donde 10 se convierte en el tercer elemento del arreglo, y así en lo sucesivo.

19.8.1 Implementación del ordenamiento por combinación

En la figura 19.6 se declara la clase `PruebaOrdenamientoCombinacion`, que contiene:

- El método `static ordenamientoCombinacion` para iniciar el ordenamiento de un arreglo `int` mediante el algoritmo de ordenamiento por combinación
- El método `static ordenarArreglo` para ejecutar el algoritmo de ordenamiento por combinación recursivo; éste es invocado por el método `ordenamientoCombinacion`
- El método `static combinar` para combinar dos subarreglos ordenados en un solo subarreglo ordenado
- El método `static subarregloString` para obtener la representación `String` de un subarreglo con el fin de imprimirlo en pantalla, y
- El método `main` para probar el método `ordenamientoCombinacion`.

El método `main` (líneas 101 a 116) es idéntico a `main` de las figuras 19.4 y 19.5, excepto que en la línea 112 se hace una llamada al método `ordenamientoCombinacion`. La salida de este programa muestra las divisiones y combinaciones que realiza el ordenamiento por combinación, indicando el progreso del ordenamiento en cada paso del algoritmo. Vale la pena invertir tiempo en recorrer paso a paso estos resultados para que entienda por completo este elegante algoritmo de ordenamiento.

```
1  // Fig. 19.6: PruebaOrdenamientoCombinacion.java
2  // Cómo ordenar un arreglo mediante el ordenamiento por combinación.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class PruebaOrdenamientoCombinacion
7  {
8     // llama al método de división recursiva para comenzar el ordenamiento por
      combinación
9     public static void ordenamientoCombinacion(int[] datos)
10    {
11       ordenarArreglo(datos, 0, datos.length - 1); // divide todo el arreglo
12    } // fin del método ordenamientoCombinacion
13
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 1 de 5).

```
14    // divide el arreglo, ordena los subarreglos y los combina en un arreglo ordenado
15    private static void ordenarArreglo(int[] datos, int inferior, int superior)
16    {
17       // evalúa el caso base; el tamaño del arreglo es igual a 1
18       if ((superior - inferior) >= 1) // si no es el caso base
19       {
20          int medio1 = (inferior + supeior) / 2; // calcula el elemento medio del
                                                      arreglo
21          int medio2 = medio1 + 1; // calcula el siguiente elemento arriba
22
23          // imprime en pantalla el paso de división
24          System.out.printf("division:   %s%n",
25             subarregloString(datos, inferior, superior));
26          System.out.printf("          %s%n",
27             subarregloString(datos, inferior, medio1));
28          System.out.printf("          %s%n%n",
29             subarregloString(datos, medio2, superior));
30
31          // divide el arreglo a la mitad; ordena cada mitad (llamadas recursivas)
32          ordenarArreglo(datos, inferior, medio1); // primera mitad del arreglo
33          ordenarArreglo(datos, medio2, superior); // segunda mitad del arreglo
34
35          // combina dos arreglos ordenados después de que regresan las llamadas
               de división
36          combinar (datos, inferior, medio1, medio2, superior);
37       } // fin de if
38    } // fin del método ordenarArreglo
39
40    // combina dos subarreglos ordenados en un subarreglo ordenado
41    private static void combinar(int[] datos, int izquierdo, int medio1,
42       int medio2, int derecho)
43    {
44       int indiceIzq = izquierdo; // indice en subarreglo izquierdo
45       int indiceDer = medio2; // índice en subarreglo derecho
46       int indiceCombinado = izquierdo; // índice en arreglo de trabajo temporal
47       int[] combinado = new int[datos.length]; // arreglo de trabajo
48
49       // imprime en pantalla los dos subarreglos antes de combinarlos
50       System.out.printf("combinacion:   %s%n",
51          subarregloString(datos, izquierdo, medio1));
52       System.out.printf("          %s%n",
53          subarregloString(datos, medio2, derecho));
54
55       // combina los arreglos hasta llegar al final de uno de ellos
56       while (indiceIzq <= medio1 && indiceDer <= derecho)
57       {
58          // coloca el menor de dos elementos actuales en el resultado
59          // y lo mueve al siguiente espacio en los arreglos
60          if (datos[indiceIzq] <= datos[indiceDer])
61             combinado[indiceCombinado++] = datos[indiceIzq++];
62          else
63             combinado[indiceCombinado++] = datos[indiceDer++];
64       }
65
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 2 de 5).

```
66          // si el arreglo izquierdo está vacío
67          if (indiceIzq == medio2)
68             // copia el resto del arreglo derecho
69             while (indiceDer <= derecho)
70                combinado[indiceCombinado++] = datos[indiceDer++];
71          else // el arreglo derecho está vacío
72             // copia el resto del arreglo izquierdo
73             while (indiceIzq <= medio1)
74                combinado[indiceCombinado++] = datos[indiceIzq++];
75
76          // copia los valores de vuelta al arreglo original
77          for (int i = izquierdo; i <= derecho; i++)
78             datos[i] = combinado[i];
79
80          // imprime en pantalla el arreglo combinado
81          System.out.printf("         %s%n%n",
82             subarregloString(datos, izquierdo, derecho));
83       } // fin del método combinar
84
85       // método para imprimir en pantalla ciertos valores en el arreglo
86       private static String subarregloString(int[] datos, int inferior, int superior)
87       {
88          StringBuilder temporal = new StringBuilder();
89
90          // imprime en pantalla espacios para la alineación
91          for (int i = 0; i < inferior; i++)
92             temporal.append("   ");
93
94          // imprime en pantalla el resto de los elementos en el arreglo
95          for (int i = inferior; i <= superior; i++)
96             temporal.append(" " + datos[i]);
97
98          return temporal.toString();
99       }
100
101      public static void main(String[] args)
102      {
103         SecureRandom generador = new SecureRandom();
104
105         int[] datos = new int[10]; // crea el arreglo
106
107         for (int i = 0; i < datos.length; i++) // llena el arreglo
108            datos[i] = 10 + generador.nextInt(90);
109
110         System.out.printf("Arreglo desordenado:%n%s%n%n",
111            Arrays.toString(datos)); // muestra el arreglo
112         ordenamientoCombinacion(datos); // ordena el arreglo
113
114         System.out.printf("Arreglo ordenado:%n%s%n%n",
115            Arrays.toString(datos)); // muestra el arreglo
116      }
117   } // fin de la clase PruebaOrdenamientoCombinacion
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 3 de 5).

```
Arreglo desordenado:
[53, 40, 28, 85, 77, 63, 76, 77, 16, 88]

division:    53 40 28 85 77 63 76 77 16 88
             53 40 28 85 77
                           63 76 77 16 88
division:    53 40 28 85 77
             53 40 28
                     85 77
division:    53 40 28
             53 40
                   28

division:    53 40
             53
                40

combinacion:     53
                    40
          40 53

combinacion:     40 53
                       28
          28 40 53

division:              85 77
                       85
                          77

combinacion:              85
                             77
                   77 85

combinacion:     28 40 53
                          77 85
                 28 40 53 77 85

division:                    63 76 77 16 88
                             63 76 77
                                      16 88

division:                    63 76 77
                             63 76
                                   77

division:                    63 76
                             63
                                76

combinacion:                 63
                                76
                             63 76

combinacion:                 63 76
                                   77
                             63 76 77

division:                             16 88
                                      16
                                         88
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 4 de 5).

```
combinacion:                              16
                                             88
                                          16 88

combinacion:                      63 76 77
                                             16 88
                                  16 63 76 77 88

combinacion:     28 40 53 77 85
                                  16 63 76 77 88
                 16 28 40 53 63 76 77 77 85 88

Arreglo ordenado:
[16, 28, 40, 53, 63, 76, 77, 77, 85, 88]
```

Fig. 19.6 | Cómo ordenar un arreglo mediante el ordenamiento por combinación (parte 5 de 5).

El método `ordenamientoCombinacion`

En las líneas 9 a 12 de la figura 19.6 se declara el método `ordenamientoCombinacion`. En la línea 11 se hace una llamada al método `ordenarArreglo` con `0` y `datos.length - 1` como los argumentos (que corresponden a los índices inicial y final, respectivamente, del arreglo que se ordenará). Estos valores indican al método `ordenarArreglo` que debe operar en todo el arreglo completo.

El método `ordenarArreglo`

Este método recursivo (líneas 15 a 38) ejecuta el algoritmo de ordenamiento por combinación recursivo. En la línea 18 se evalúa el caso base. Si el tamaño del arreglo es 1, ya está ordenado, por lo que el método regresa de inmediato. Si el tamaño del arreglo es mayor que 1, el método divide el arreglo en dos, llama en forma recursiva al método `ordenarArreglo` para ordenar los dos subarreglos y después los combina. En la línea 32 se hace una llamada recursiva al método `ordenarArreglo` en la primera mitad del arreglo, mientras que en la línea 33 se hace una llamada recursiva al método `ordenarArreglo` en la segunda mitad del mismo. Cuando regresan estas dos llamadas al método, cada mitad del arreglo se ha ordenado. En la línea 36 se hace una llamada al método `combinar` (líneas 41 a 83) con las dos mitades del arreglo, para combinar los dos arreglos ordenados en un arreglo ordenado más grande.

El método `combinar`

En las líneas 41 a 83 se declara el método `combinar`. En las líneas 56 a 64 se itera hasta llegar al final de cualquiera de los subarreglos. En la línea 60 se evalúa cuál elemento al principio de los arreglos es más pequeño. Si el elemento en el arreglo izquierdo es más pequeño, en la línea 61 se coloca el elemento en su posición en el arreglo combinado. Si el elemento en el arreglo derecho es más pequeño, en la línea 63 se coloca en su posición en el arreglo combinado. Cuando el ciclo `while` termina, un subarreglo completo se coloca en el arreglo combinado, pero el otro subarreglo aún contiene datos. En la línea 67 se evalúa si el arreglo izquierdo ha llegado al final. De ser así, en las líneas 69 y 70 se llena el arreglo combinado con los elementos del arreglo derecho. Si el arreglo izquierdo no ha llegado al final, entonces el arreglo derecho debe haber llegado, por lo que en las líneas 73 y 74 se llena el arreglo combinado con los elementos del arreglo izquierdo. Por último, en las líneas 77 y 78 se copia el arreglo combinado en el arreglo original.

19.8.2 Eficiencia del ordenamiento por combinación

El ordenamiento por combinación es un algoritmo *mucho más eficiente* que el de inserción o el de selección. Considere la primera llamada (no recursiva) al método `ordenarArreglo`. Esto produce dos llamadas recursivas al método `ordenarArreglo` con subarreglos, cada uno de los cuales tiene un tamaño aproximado de la mitad del arreglo original, y una sola llamada al método `combinar`, que requiere a lo más, $n - 1$ comparaciones para llenar el arreglo original, que es $O(n)$. (Recuerde que se puede elegir cada elemento

en el arreglo mediante la comparación de un elemento de cada uno de los subarreglos). Las dos llamadas al método `ordenarArreglo` producen cuatro llamadas recursivas más al método `ordenarArreglo`, cada una con un subarreglo de un tamaño aproximado a una cuarta parte del tamaño del arreglo original, junto con dos llamadas al método `combinar` que requieren, a lo más, $n/2 - 1$ comparaciones para un número total de $O(n)$ comparaciones. Este proceso continúa, y cada llamada a `ordenarArreglo` genera dos llamadas adicionales a `ordenarArreglo` y una llamada a `combinar`, hasta que el algoritmo *divide* el arreglo en subarreglos de un elemento. En cada nivel, se requieren $O(n)$ comparaciones para *combinar* los subarreglos. Cada nivel divide los arreglos a la mitad, por lo que al duplicar el tamaño del arreglo se requiere un nivel más. Si se cuadruplica el tamaño del arreglo, se requieren dos niveles más. Este patrón es logarítmico, y produce $\log_2 n$ niveles. Esto resulta en una eficiencia total de ***O*(*n* log *n*)**.

19.9 Resumen de Big O para los algoritmos de búsqueda y ordenamiento de este capítulo

En la figura 19.17 se sintetizan los algoritmos de búsqueda y ordenamiento que cubrimos en este capítulo, junto con el valor de Big O para cada uno de ellos. En la figura 19.8 se enlistan los valores de Big O que hemos cubierto en este capítulo, junto con cierto número de valores para n, de modo que se resalten las diferencias en las proporciones de crecimiento.

Algoritmo	Ubicación	Big O
Algoritmos de búsqueda:		
Búsqueda lineal	Sección 19.2	$O(n)$
Búsqueda binaria	Sección 19.4	$O(\log n)$
Búsqueda lineal recursiva	Ejercicio 19.8	$O(n)$
Búsqueda binaria recursiva	Ejercicio 19.9	$O(\log n)$
Algoritmos de ordenamiento:		
Ordenamiento por selección	Sección 19.6	$O(n^2)$
Ordenamiento por inserción	Sección 19.7	$O(n^2)$
Ordenamiento por combinación	Ejercicio 19.8	$O(n \log n)$
Ordenamiento de burbuja	Ejercicios 19.5 y 19.6	$O(n^2)$

Fig. 19.7 | Algoritmos de búsqueda y ordenamiento con valores de Big O.

n =	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	16
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10,000
1000	3	1000	3000	10^6
1,000,000	6	1,000,000	6,000,000	10^{12}
1,000,000,000	9	1,000,000,000	9,000,000,000	10^{18}

Fig. 19.8 | Número de comparaciones para las notaciones comunes de Big O.

19.10 Conclusión

En este capítulo se presentaron las técnicas de ordenamiento y búsqueda. Hablamos sobre dos algoritmos de búsqueda (búsqueda lineal y búsqueda binaria) y tres algoritmos de ordenamiento (ordenamiento por selección, por inserción y por combinación). Presentamos la notación Big O, la cual nos ayuda a analizar la eficiencia de un algoritmo. En los siguientes dos capítulos continuaremos nuestro análisis sobre las estructuras de datos dinámicas, que pueden aumentar o reducir su tamaño en tiempo de ejecución. En el capítulo 20 demostraremos cómo usar las herramientas de genéricos de Java para implementar métodos genéricos y clases genéricas. En el capítulo 21 (en inglés, en el sitio web del libro) veremos los detalles sobre la implementación de estructuras de datos genéricas. Cada uno de los algoritmos en este capítulo es de "un solo hilo"; en el capítulo 23 "*Concurrency*" hablaremos sobre la tecnología multihilo y cómo puede ayudarle a su programa a obtener un mejor rendimiento en los sistemas multinúcleo de la actualidad.

Resumen

Sección 19.1 Introducción

- La búsqueda (pág. 811) implica determinar si una clave de búsqueda está presente en los datos y de ser así, encontrar su ubicación.
- El ordenamiento (pág. 811) implica poner los datos en orden.

Sección 19.2 Búsqueda lineal

- El algoritmo de búsqueda lineal (pág. 812) busca secuencialmente cada elemento en el arreglo hasta que encuentra el elemento correcto o hasta llegar al final del mismo sin encontrar ese elemento.

Sección 19.3 Notación Big O

- Una de las principales diferencias entre los algoritmos de búsqueda es la cantidad de esfuerzo que requieren para poder devolver un resultado.
- La notación Big O (pág. 814) describe la eficiencia de un algoritmo en términos del trabajo requerido para resolver un problema. Para los algoritmos de búsqueda y ordenamiento, por lo general depende del número de elementos en los datos.
- Un algoritmo que es $O(1)$ no necesariamente requiere sólo una comparación (pág. 815). Sólo significa que el número de comparaciones no aumenta a medida que se incrementa el tamaño del arreglo.
- Se considera que un algoritmo $O(n)$ tiene un tiempo de ejecución lineal (pág. 815).
- La notación Big O está diseñada para resaltar los factores dominantes e ignorar los términos que pierden importancia con valores altos de n.
- La notación Big O se enfoca en la proporción de crecimiento de los tiempos de ejecución de los algoritmos, por lo que se ignoran las constantes.
- El algoritmo de búsqueda lineal se ejecuta en un tiempo $O(n)$.
- El peor caso en la búsqueda lineal es que se debe comprobar cada elemento (pág. 816) para determinar si la clave de búsqueda existe (lo que ocurre si la clave de búsqueda es el último elemento en el arreglo) o si no está presente.

Sección 19.4 Búsqueda binaria

- El algoritmo de búsqueda binaria (pág. 816) es más eficiente que el algoritmo de búsqueda lineal, pero requiere que el arreglo esté ordenado.
- La primera iteración de la búsqueda binaria evalúa el elemento medio del arreglo. Si es igual a la clave de búsqueda, el algoritmo devuelve su ubicación. Si la clave de búsqueda es menor que el elemento medio, la búsqueda continúa con la primera mitad del arreglo. Si la clave de búsqueda es mayor que el elemento medio, la búsqueda binaria continúa con la segunda mitad del arreglo. En cada iteración se evalúa el valor medio del resto del arreglo y si no se encuentra el elemento, se elimina la mitad de los elementos restantes.
- La búsqueda binaria es un algoritmo de búsqueda más eficiente que la búsqueda lineal, ya que cada comparación elimina la mitad de los elementos del arreglo a considerar.

- La búsqueda binaria se ejecuta en un tiempo $O(\log n)$, ya que cada paso elimina la mitad de los elementos restantes; esto también se conoce como tiempo de ejecución logarítmico (pág. 820).
- Si el tamaño del arreglo se duplica, la búsqueda binaria sólo requiere una comparación adicional.

Sección 19.6 Ordenamiento por selección

- El ordenamiento por selección (pág. 821) es un algoritmo de ordenamiento simple, pero ineficiente.
- El ordenamiento empieza seleccionando el elemento más pequeño en el arreglo y lo intercambia con el primer elemento. En la segunda iteración se selecciona el segundo elemento más pequeño (que viene siendo el elemento restante más pequeño) y se intercambia con el segundo elemento. El ordenamiento continúa hasta que en la última iteración se selecciona el segundo elemento más grande, y se intercambia con el antepenúltimo elemento, dejando el elemento más grande en el último índice. En la i-ésima iteración del ordenamiento por selección, los i elementos más pequeños de todo el arreglo se ordenan en los primeros i índices.
- El algoritmo de ordenamiento por selección se ejecuta en un tiempo $O(n^2)$ (pág. 824).

Sección 19.7 Ordenamiento por inserción

- En la primera iteración del ordenamiento por inserción (pág. 824) se toma el segundo elemento en el arreglo y si es menor que el primer elemento, éstos se intercambian. En la segunda iteración se analiza el tercer elemento y se inserta en la posición correcta con respecto a los primeros dos elementos. Después de la i-ésima iteración del ordenamiento por inserción, quedan ordenados los primeros i elementos del arreglo original.
- El algoritmo de ordenamiento por inserción se ejecuta en un tiempo $O(n^2)$ (pág. 827).

Sección 19.8 Ordenamiento por combinación

- El ordenamiento por combinación (pág. 827) es un algoritmo de ordenamiento que es más rápido pero más complejo de implementar que el ordenamiento por selección y el ordenamiento por inserción. Para ordenar un arreglo, el algoritmo de ordenamiento por combinación lo divide en dos subarreglos de igual tamaño, ordena cada subarreglo en forma recursiva y combina los subarreglos en un arreglo más grande.
- El caso base del ordenamiento por combinación es un arreglo con un elemento. Un arreglo de un elemento ya está ordenado, por lo que el ordenamiento por combinación regresa de inmediato cuando se llama con un arreglo de un elemento. La parte de este algoritmo que corresponde al proceso de combinar recibe dos arreglos ordenados y los combina en un arreglo ordenado más grande.
- Para realizar la combinación, el ordenamiento por combinación analiza el primer elemento en cada arreglo, que también es el elemento más pequeño en el arreglo. El ordenamiento por combinación recibe el más pequeño de estos elementos y lo coloca en el primer elemento del arreglo más grande. Si aún hay elementos en el subarreglo, el ordenamiento por combinación analiza el segundo elemento en el subarreglo (que ahora es el elemento más pequeño restante) y lo compara con el primer elemento en el otro subarreglo. El ordenamiento por combinación continúa con este proceso hasta que se llena el arreglo más grande.
- En el peor caso, la primera llamada al ordenamiento por combinación tiene que realizar $O(n)$ comparaciones para llenar las n posiciones en el arreglo final.
- La porción del algoritmo de ordenamiento por combinación que corresponde al proceso de combinar se realiza en dos subarreglos, cada uno de un tamaño aproximado a $n/2$. Para crear cada uno de estos subarreglos, se requieren $n/2 - 1$ comparaciones para cada subarreglo, o un total de $O(n)$ comparaciones. Este patrón continúa a medida que cada nivel trabaja hasta en el doble de esa cantidad de arreglos, pero cada uno equivale a la mitad del tamaño del arreglo anterior.
- De manera similar a la búsqueda binaria, esta acción de partir los subarreglos a la mitad produce un total de $\log n$ niveles, para una eficiencia total de $O(n \log n)$ (pág. 833).

Ejercicios de autoevaluación

19.1 Complete los siguientes enunciados:

a) Una aplicación de ordenamiento por selección debe requerir un tiempo aproximado de _____ veces más para ejecutarse en un arreglo de 128 elementos, en comparación con un arreglo de 32 elementos.

b) La eficiencia del ordenamiento por combinación es de _____.

19.2 ¿Qué aspecto clave de la búsqueda binaria y del ordenamiento por combinación es responsable de la parte logarítmica de sus respectivos valores Big O?

19.3 ¿En qué sentido es superior el ordenamiento por inserción al ordenamiento por combinación? ¿En qué sentido es superior el ordenamiento por combinación al ordenamiento por inserción?

19.4 En el texto decimos que una vez que el ordenamiento por combinación divide el arreglo en dos subarreglos, después ordena estos dos subarreglos y los combina. ¿Por qué alguien podría confundirse cuando decimos que "después ordena estos dos subarreglos"?

Respuestas a los ejercicios de autoevaluación

19.1 a) 16, ya que un algoritmo $O(n^2)$ requiere 16 veces más de tiempo para ordenar hasta cuatro veces más información. b) $O(n \log n)$.

19.2 Ambos algoritmos incorporan la acción de "dividir a la mitad" (reducir algo de cierta forma a la mitad). La búsqueda binaria elimina del proceso una mitad del arreglo después de cada comparación. El ordenamiento por combinación divide el arreglo a la mitad, cada vez que se llama.

19.3 El ordenamiento por inserción es más fácil de comprender y de programar que el ordenamiento por combinación. El ordenamiento por combinación es mucho más eficiente [$O(n \log n)$] que el ordenamiento por inserción [$O(n^2)$].

19.4 En cierto sentido, en realidad no ordena estos dos subarreglos. Simplemente sigue dividiendo el arreglo original a la mitad, hasta que obtiene un subarreglo de un elemento, que desde luego está ordenado. Después construye los dos subarreglos originales al combinar estos arreglos de un elemento para formar subarreglos más grandes, los cuales se mezclan, y así en lo sucesivo.

Ejercicios

19.5 ***(Ordenamiento de burbuja)*** Implemente el ordenamiento de burbuja (otra técnica de ordenamiento simple pero ineficiente). Se le llama ordenamiento de burbuja o de hundimiento, debido a que los valores más pequeños van "subiendo como burbujas" en forma gradual hasta llegar a la parte superior del arreglo (es decir, hacia el primer elemento) como las burbujas de aire que se elevan en el agua, mientras que los valores más grandes se hunden en el fondo (final) del arreglo. Esta técnica utiliza ciclos anidados para realizar varias pasadas a través del arreglo. Cada pasada compara pares sucesivos de elementos. Si un par se encuentra en orden ascendente (o los valores son iguales), el ordenamiento de burbuja deja los valores como están. Si un par se encuentra en orden descendente, el ordenamiento de burbuja intercambia sus valores en el arreglo. En la primera pasada se comparan los primeros dos elementos del arreglo y se intercambian sus valores si es necesario. Después se comparan los elementos segundo y tercero en el arreglo. Al final de esta pasada se comparan los últimos dos elementos en el arreglo y se intercambian, en caso de ser necesario. Después de una pasada el elemento más grande estará en el último índice. Después de dos pasadas, los dos elementos más grandes se encontrarán en los últimos dos índices. Explique por qué el ordenamiento de burbuja es un algoritmo $O(n^2)$.

19.6 ***(Ordenamiento de burbuja mejorado)*** Realice las siguientes modificaciones simples para mejorar el rendimiento del ordenamiento de burbuja que desarrolló en el ejercicio 19.5:

a) Después de la primera pasada, se garantiza que el número más grande estará en el elemento con la numeración más alta del arreglo; después de la segunda pasada, los dos números más altos estarán "acomodados"; y así en lo sucesivo. En vez de realizar nueve comparaciones en cada pasada para un elemento con diez arreglos, modifique el ordenamiento de burbuja para que realice ocho comparaciones en la segunda pasada, siete en la tercera, y así en lo sucesivo.

b) Los datos en el arreglo tal vez se encuentren ya en el orden apropiado, o casi apropiado, así que ¿para qué realizar nueve pasadas, si basta con menos? Modifique el ordenamiento para comprobar al final de cada pasada si se han realizado intercambios. Si no se ha realizado ninguno, los datos ya deben estar en el orden apropiado, por lo que el programa debe terminar. Si se han realizado intercambios, por lo menos se necesita una pasada más.

19.7 ***(Ordenamiento de cubeta)*** Un ordenamiento de cubeta comienza con un arreglo unidimensional de enteros positivos que se deben ordenar, y un arreglo bidimensional de enteros, en el que las filas están indexadas de 0 a 9

y las columnas de 0 a $n - 1$, en donde n es el número de valores a ordenar. Cada fila del arreglo bidimensional se conoce como una *cubeta*. Escriba una clase llamada `OrdenamientoCubeta` que contenga un método llamado `ordenar` y que opere de la siguiente manera:

a) Coloque cada valor del arreglo unidimensional en una fila del arreglo de cubeta, con base en el dígito de las unidades (el de más a la derecha) del valor. Por ejemplo, el número 97 se coloca en la fila 7, el 3 se coloca en la fila 3 y el 100 se coloca en la fila 0. A este procedimiento se le llama *pasada de distribución*.

b) Itere a través del arreglo de cubeta fila por fila, y copie los valores de vuelta al arreglo original. A este procedimiento se le llama *pasada de recopilación*. El nuevo orden de los valores anteriores en el arreglo unidimensional es 100, 3 y 97.

c) Repita este proceso para cada posición de dígito subsiguiente (decenas, centenas, miles, etcétera). En la segunda pasada (el dígito de las decenas) se coloca el 100 en la fila 0, el 3 en la fila 0 (ya que 3 no tiene dígito de decenas) y el 97 en la fila 9. Después de la pasada de recopilación, el orden de los valores en el arreglo unidimensional es 100, 3 y 97. En la tercera pasada (dígito de las centenas), el 100 se coloca en la fila 1, el 3 en la fila 0 y el 97 en la fila 0 (después del 3). Después de esta última pasada de recopilación, el arreglo original se encuentra en orden.

El arreglo bidimensional de cubetas es 10 veces la longitud del arreglo entero que se está ordenando. Esta técnica de ordenamiento proporciona un mejor rendimiento que el ordenamiento de burbuja, pero requiere mucha más memoria; el ordenamiento de burbuja requiere espacio sólo para un elemento adicional de datos. Esta comparación es un ejemplo de la concesión entre espacio y tiempo, ya que el ordenamiento de cubeta utiliza más memoria que el ordenamiento de burbuja, pero su rendimiento es mejor. Esta versión del ordenamiento de cubeta requiere copiar todos los datos de vuelta al arreglo original en cada pasada. Otra posibilidad es crear un segundo arreglo de cubeta bidimensional e intercambiar en forma repetida los datos entre los dos arreglos de cubeta.

19.8 ***(Búsqueda lineal recursiva)*** Modifique la figura 19.2 de modo que se utilice el método recursivo `busquedaLinealRecursiva` para realizar una búsqueda lineal en el arreglo. El método debe recibir la clave de búsqueda y el índice inicial como argumentos. Si se encuentra la clave de búsqueda, se devuelve su índice en el arreglo; en caso contrario, se devuelve -1. Cada llamada al método recursivo debe comprobar un índice en el arreglo.

19.9 ***(Búsqueda binaria recursiva)*** Modifique la figura 19.3 de modo que se utilice el método recursivo `busquedaBinariaRecursiva` para realizar una búsqueda binaria en el arreglo. El método debe recibir la clave de búsqueda, el índice inicial y el índice final como argumentos. Si se encuentra la clave de búsqueda, se devuelve su índice en el arreglo. Si no se encuentra, se devuelve `-1`.

19.10 ***(Quicksort)*** La técnica de ordenamiento recursiva llamada "quicksort" utiliza el siguiente algoritmo básico para un arreglo unidimensional de valores:

a) *Paso de particionamiento*: tomar el primer elemento del arreglo desordenado y determinar su ubicación final en el arreglo ordenado (es decir, todos los valores a la izquierda del elemento en el arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores; a continuación le mostraremos cómo hacer esto). Ahora tenemos un elemento en su ubicación apropiada y dos subarreglos desordenados.

b) *Paso recursivo*: llevar a cabo el *paso 1* en cada subarreglo desordenado. Cada vez que se realiza el *paso 1* en un subarreglo, se coloca otro elemento en su ubicación final en el arreglo ordenado y se crean dos subarreglos desordenados. Cuando un subarreglo consiste en un elemento, ese elemento se encuentra en su ubicación final (debido a que un arreglo de un elemento ya está ordenado).

El algoritmo básico parece bastante simple, pero ¿cómo determinamos la posición final del primer elemento de cada subarreglo? Como ejemplo, considere el siguiente conjunto de valores (el elemento en negritas es el elemento de particionamiento; se colocará en su ubicación final en el arreglo ordenado):

37 2 6 4 89 8 10 12 68 45

Empezando desde el elemento de más a la derecha del arreglo, se compara cada elemento con **37** hasta que se encuentra un elemento menor que **37**; después se intercambian el **37** y ese elemento. El primer elemento menor que **37** es 12, por lo que se intercambian el **37** y el 12. El nuevo arreglo es

12 2 6 4 89 8 10 37 68 45

El elemento 12 está en cursivas, para indicar que se acaba de intercambiar con el **37**.

Empezando desde la parte izquierda del arreglo, pero con el elemento que está después de 12, compare cada elemento con **37** hasta encontrar un elemento mayor que **37**; después intercambie el **37** y ese elemento. El primer elemento mayor que **37** es 89, por lo que se intercambian el **37** y el 89. El nuevo arreglo es

12 2 6 4 **37** 8 10 *89* 68 45

Empezando desde la derecha, pero con el elemento antes del 89, compare cada elemento con **37** hasta encontrar un elemento menor que **37**; después se intercambian el **37** y ese elemento. El primer elemento menor que **37** es 10, por lo que se intercambian **37** y 10. El nuevo arreglo es

12 2 6 4 *10* 8 **37** 89 68 45

Empezando desde la izquierda, pero con el elemento que está después de 10, compare cada elemento con **37** hasta encontrar un elemento mayor que **37**; después intercambie el **37** y ese elemento. No hay más elementos mayores que **37**, por lo que al comparar el **37** consigo mismo, sabemos que se ha colocado en su ubicación final en el arreglo ordenado. Cada valor a la izquierda de **37** es más pequeño, y cada valor a la derecha de **37** es más grande.

Una vez que se ha aplicado la partición en el arreglo anterior, hay dos subarreglos desordenados. El subarreglo con valores menores que 37 contiene 12, 2, 6, 4, 10 y 8. El subarreglo con valores mayores que 37 contiene 89, 68 y 45. El ordenamiento continúa en forma recursiva, y ambos subarreglos se particionan de la misma forma que el arreglo original.

Con base en la explicación anterior, escriba el método recursivo `ayudanteQuicksort` para ordenar un arreglo entero unidimensional. El método debe recibir como argumentos un índice inicial y un índice final en el arreglo original que se está ordenando.