

GOODRICH / TAMASSIA

ESTRUCTURAS DE DATOS Y ALGORITMOS EN JAVA

**2A.
EDICIÓN**

CECSA

Contenido

10.1	Ordenamiento <i>merge</i>	448
10.1.1	Divide y vencerás	449
10.1.2	Implementación del ordenamiento <i>merge</i> en Java ..	458
10.1.3	Ordenamiento <i>merge</i> y relaciones de recurrencia ★	460
10.2	Tipo de dato abstracto conjunto	461
10.2.1	Implementación sencilla de un conjunto	462
10.3	<i>Quick sort</i>	467
10.3.1	Ordenamiento <i>Quick-sort</i> en el lugar	472
10.3.2	Ordenamiento <i>Quick-sort</i> aleatorizado	476
10.4	Cota inferior de ordenamiento basada en comparación ...	478
10.5	Ordenamiento por cubetas y ordenamiento radix	480
10.5.1	Ordenamiento por cubetas	480
10.5.2	Ordenamiento radix	481
10.6	Comparación de algoritmos de ordenamiento	483
10.7	Selección	484
10.7.1	Poda y búsqueda	485
10.7.2	Selección rápida aleatorizada	485
10.7.3	Análisis de la selección rápida aleatorizada ★	486
10.8	Ejercicios	488

La segunda ley de la termodinámica parece indicar que la naturaleza tiende al desorden, y el hombre, por otra parte, prefiere el orden. En realidad hay varias ventajas al mantener datos ordenados. Por ejemplo, el algoritmo de búsqueda binaria, descrito en la sección 8.2, funciona bien sólo para un vector ordenado. Como las computadoras pretenden ser herramientas para el hombre, se dedica este capítulo al estudio de algoritmos de ordenamiento y sus aplicaciones. Recuérdese que el problema de ordenamiento se define como sigue: sea S una secuencia de n elementos que se pueden comparar entre sí según una relación de orden total, esto es, siempre es posible comparar dos elementos de S y ver cuál es mayor o menor, o si los dos son iguales. Se trata de reorganizar S de modo que los elementos aparezcan en orden creciente (o en orden no decreciente si hay elementos iguales en S).

Ya se presentaron varios algoritmos de ordenamiento en los capítulos anteriores. En particular, en la sección 7.1.2 se presentó un esquema sencillo de ordenamiento, llamado `PriorityQueueSort`, que consiste en insertar elementos en una cola de prioridad para después extraerlos en orden no decreciente, por medio de una serie de operaciones `removeMin`. Si se implementa la cola de prioridad mediante una secuencia, `PriorityQueueSort` se ejecuta en el tiempo $O(n^2)$, y corresponde al método de ordenamiento llamado ordenamiento por inserción, u ordenamiento por selección, dependiendo de si se mantiene ordenada la secuencia que forme la cola de prioridad (sección 7.2.3). Si en lugar de ello la cola de prioridad se implementa mediante un *heap* (sección 7.3), `PriorityQueueSort` se ejecuta en el tiempo $O(n \log n)$, y corresponde al método llamado ordenamiento *heap* (sección 7.3.4).

En este capítulo se presentan otros cuatro algoritmos de ordenamiento, llamados *ordenamiento merge*, *ordenamiento rápido*, *ordenamiento por cubetas* y *ordenamiento radix*. También se presenta el tipo de dato abstracto *conjunto* y se muestra el modo en que la técnica *merge* usada en el algoritmo de ordenamiento *merge* se puede emplear para implementar sus métodos. En todo el capítulo se supone que hay una relación de orden total definida sobre los elementos por ordenar. Si esta relación se induce por un comparador (sección 7.1.4), se supone que una prueba de comparación ocupa el tiempo $O(1)$.

10.1 Ordenamiento *merge*

Ya se ha visto el poder de la recursión para describir algoritmos en forma elegante. Véanse, por ejemplo, las técnicas de recorrido de árbol que se presentaron en el capítulo 6. En esta sección se presenta una técnica de ordenamiento llamada *ordenamiento merge*, que se puede describir en una forma sencilla y compacta mediante la recursión. Este algoritmo también se basa en el uso de un patrón de diseño algorítmico llamado *divide y vencerás*, que es poderoso y general al mismo tiempo.

10.1.1 Divide y vencerás

El ordenamiento *merge* se basa en el patrón de diseño algorítmico llamado **divide y vencerás**. Este paradigma se puede describir en términos generales formado de los tres pasos siguientes:

1. **Dividir:** Si el tamaño de la entrada es menor que determinado umbral (por ejemplo, uno o dos elementos), resolver el problema en forma directa con un método directo, y regresar la solución que así se obtuvo. En caso contrario, dividir los datos de entrada en dos o más subconjuntos ajenos (o subconjuntos “disjuntos”).
2. **Recursión:** Resolver recursivamente los subproblemas asociados con los subconjuntos.
3. **Vencer:** Tomar las soluciones de los subproblemas y reunir las en una solución del problema original.

El algoritmo de ordenamiento *merge* aplica la técnica de divide y vencerás al problema del ordenamiento.

Aplicación de divide y vencerás en ordenamientos

Recuerde que en el problema de ordenamiento el dato es un conjunto de n objetos, que normalmente están guardados en una lista, vector, arreglo o secuencia, junto con algún comparador que defina un orden total en esos objetos, y lo que nos piden es preparar una representación ordenada de estos objetos. Para fines de generalización, nos concentramos en la versión del problema de ordenamiento que toma una secuencia S de objetos como entrada y regresa S ordenada. Las especializaciones a otras estructuras lineales, como listas, vectores o arreglos son directas, y se dejan como ejercicios R-10.3 y R-10.12. Para el problema de ordenar una secuencia S con n elementos, los tres pasos de divide y vencerás son los siguientes:

1. **Dividir:** Si S tiene cero o un elemento, regresar S de inmediato; ya está ordenado. En caso contrario (S tiene al menos dos elementos), eliminar todos los elementos de S y ponerlos en dos secuencias, S_1 y S_2 , cada uno con más o menos la mitad de los elementos de S ; esto es, S_1 contiene los primeros $\lceil n/2 \rceil$ elementos de S y S_2 contiene los restantes $\lfloor n/2 \rfloor$ elementos.
2. **Recursión:** Ordenar recursivamente las secuencias S_1 y S_2 .
3. **Vencer:** Regresar los elementos a S , reuniendo las secuencias ordenadas S_1 y S_2 en una sola secuencia ordenada.

Respecto al paso de dividir, recuérdese que la notación $\lceil x \rceil$ indica el **techo** de x , esto es, el menor entero m tal que $x \leq m$. De igual modo, la notación $\lfloor x \rfloor$ indica el **piso** de x , que es el mayor entero k tal que $k \leq x$.

Se puede visualizar una ejecución del algoritmo de ordenamiento *merge* por medio de un árbol binario T , llamado **árbol de ordenamiento *merge***. Cada nodo de T representa una invocación o llamada recursiva al algoritmo de ordenamiento *merge*. Asociamos con cada nodo v de T la secuencia S que es procesada por la invocación asociada con v . Los hijos del nodo v se asocian con las llamadas recursivas que procesan las subsecuencias S_1 y S_2 de S . Los nodos externos de T se asocian con los elementos individuales de S , que corresponden a las instancias del algoritmo que no hacen llamadas recursivas.

La figura 10.1 es un resumen de una ejecución del algoritmo de ordenamiento *merge*, y muestra las secuencias de entrada y salida procesadas en cada nodo del árbol de ordenamiento *merge*. En las figuras 10.2 a 10.5 se muestra la evolución, paso a paso, del árbol de ordenamiento *merge*.

Esta visualización del algoritmo en términos del árbol de ordenamiento *merge* ayuda a analizar el tiempo de ejecución del algoritmo. En particular, como el tamaño de la secuencia de entrada se reduce más o menos a la mitad en cada llamada recursiva de ordenamiento *merge*, la altura aproximada del árbol de ordenamiento *merge* es $\log n$; recuérdese que la base del logaritmo es 2 si se ha omitido.

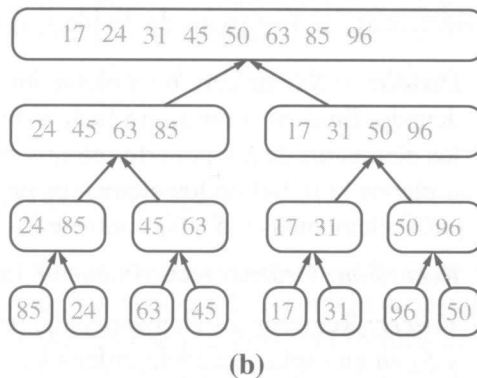
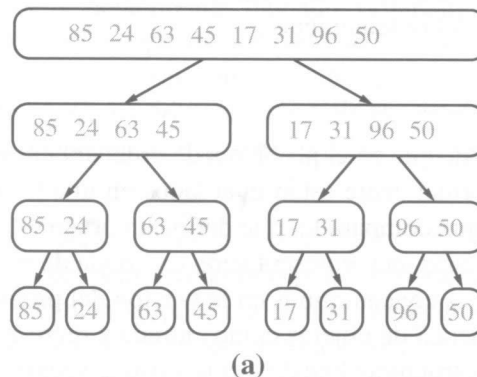


Figura 10.1: Árbol T de ordenamiento *merge*, para ejecutar el algoritmo de ordenamiento *merge* en una secuencia con 8 elementos: (a) secuencias de entrada procesadas en cada nodo de T ; (b) secuencias de salida generadas en cada nodo de T .

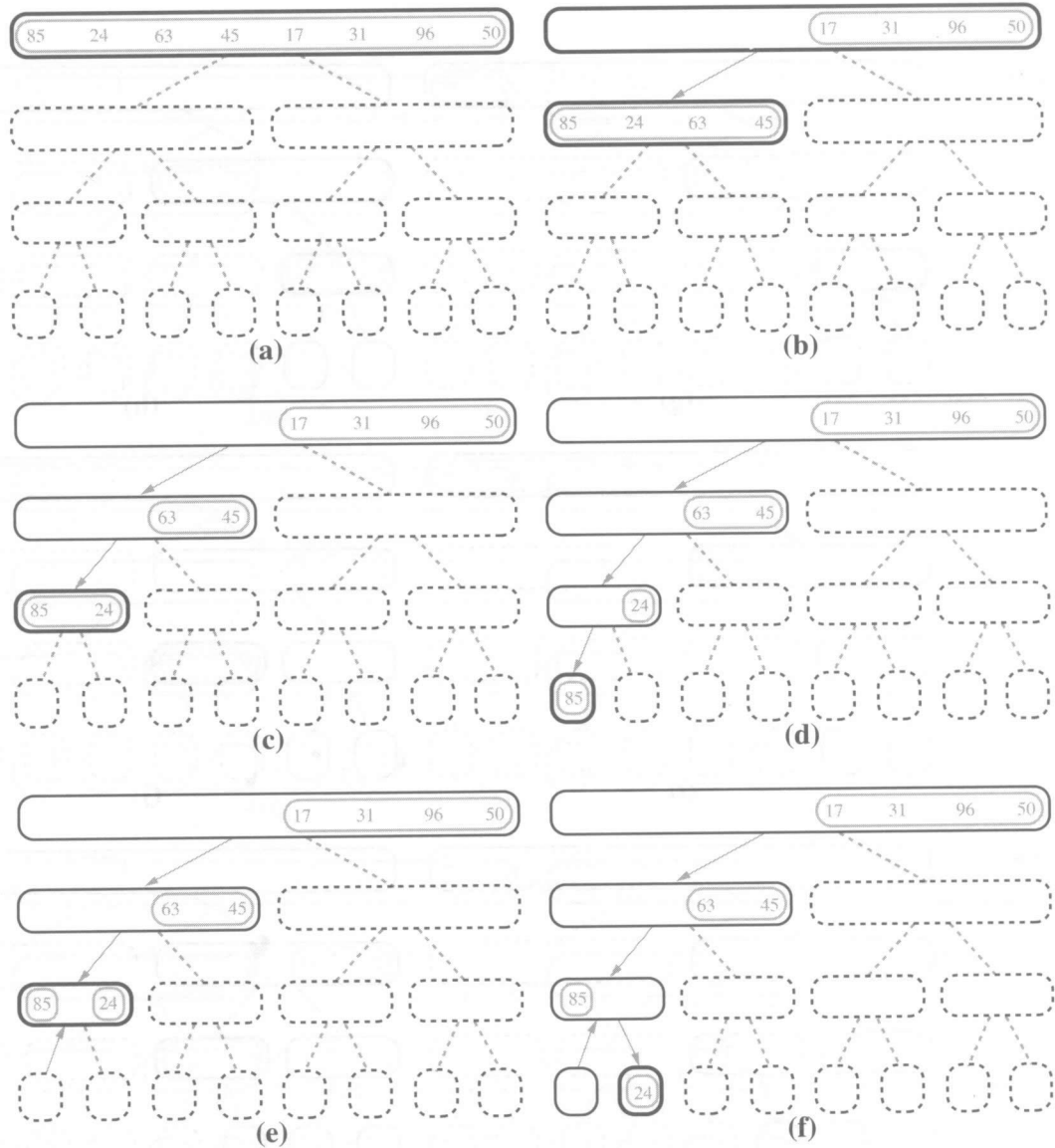


Figura 10.2: Visualización de una ejecución del ordenamiento *merge*. Cada nodo del árbol representa una llamada recursiva del ordenamiento *merge*. Los nodos que se trazan con líneas interrumpidas representan llamadas que todavía no se han hecho. El nodo con líneas gruesas representa la llamada actual. Los nodos vacíos con línea delgada representan llamadas completadas. Los nodos restantes (trazados con líneas delgadas y que no están vacíos) representan llamadas que esperan el regreso de la invocación de un hijo.

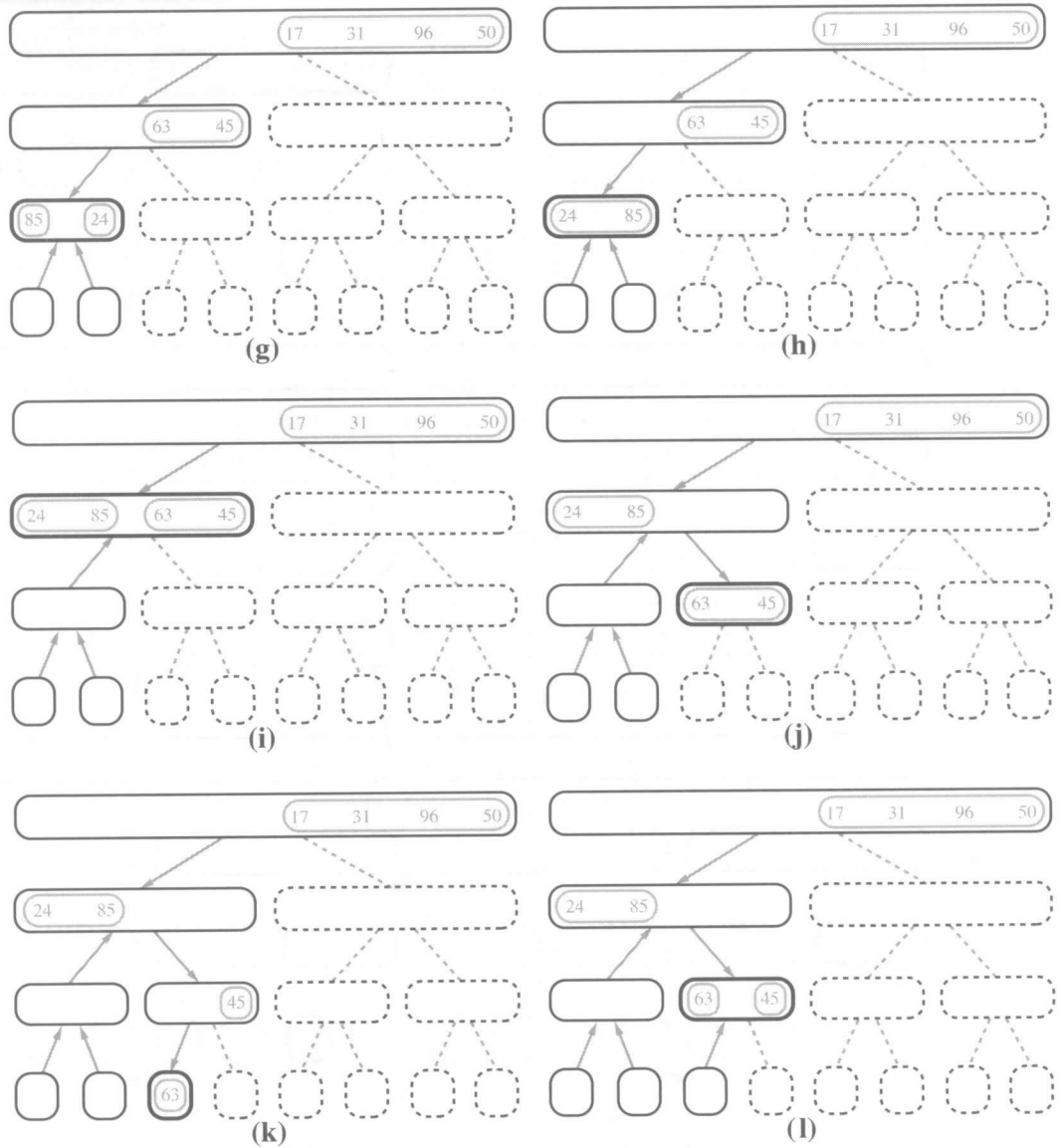


Figura 10.3: (Continuación de la figura 10.2.) Visualización de una ejecución del ordenamiento *merge*. Nótese el paso vencer ejecutado en (h).

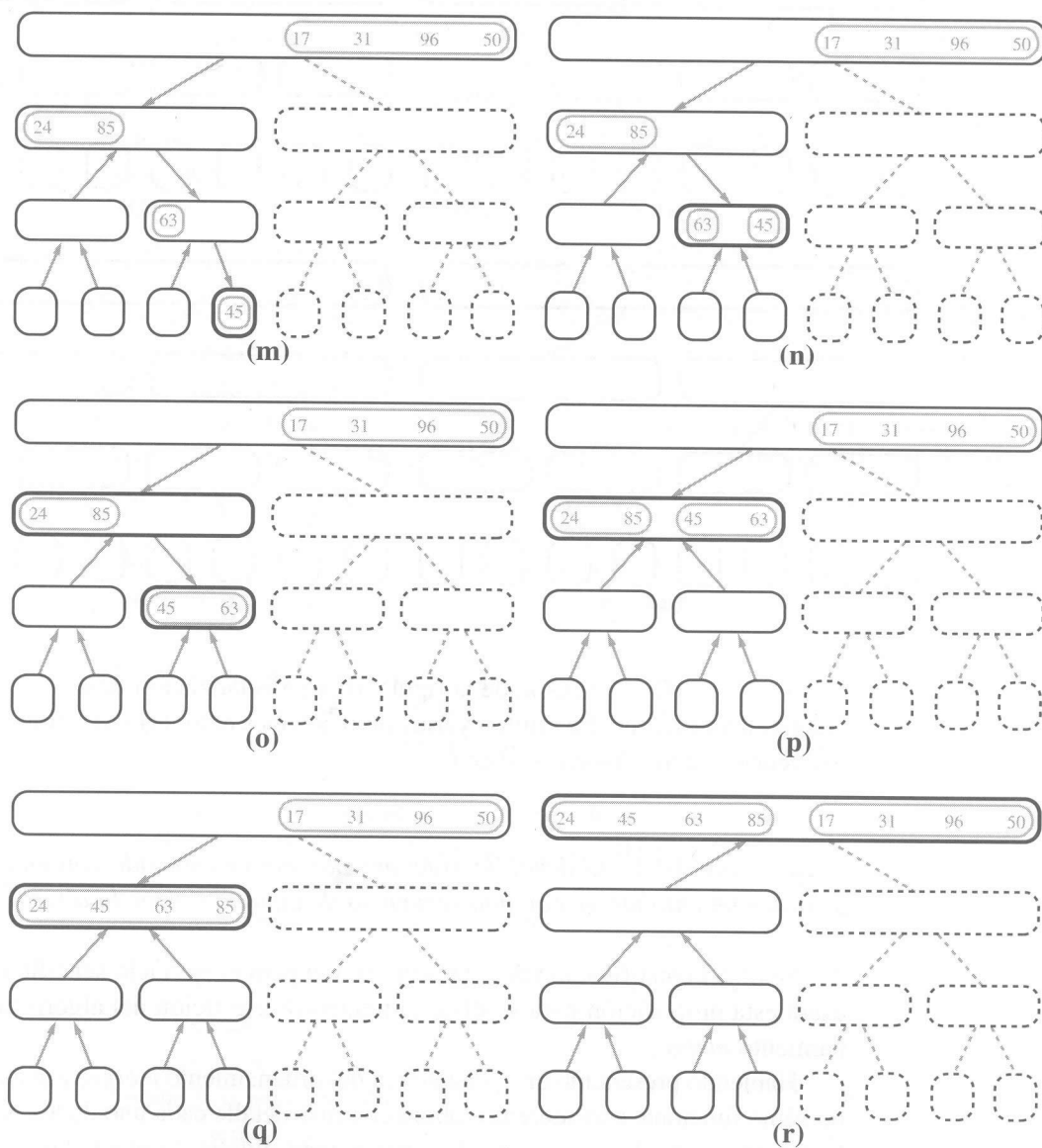


Figura 10.4: (Continuación de la figura 10.3.) Visualización de una ejecución del ordenamiento *merge*. Nótese los pasos vencer ejecutados en (o) y en (q).

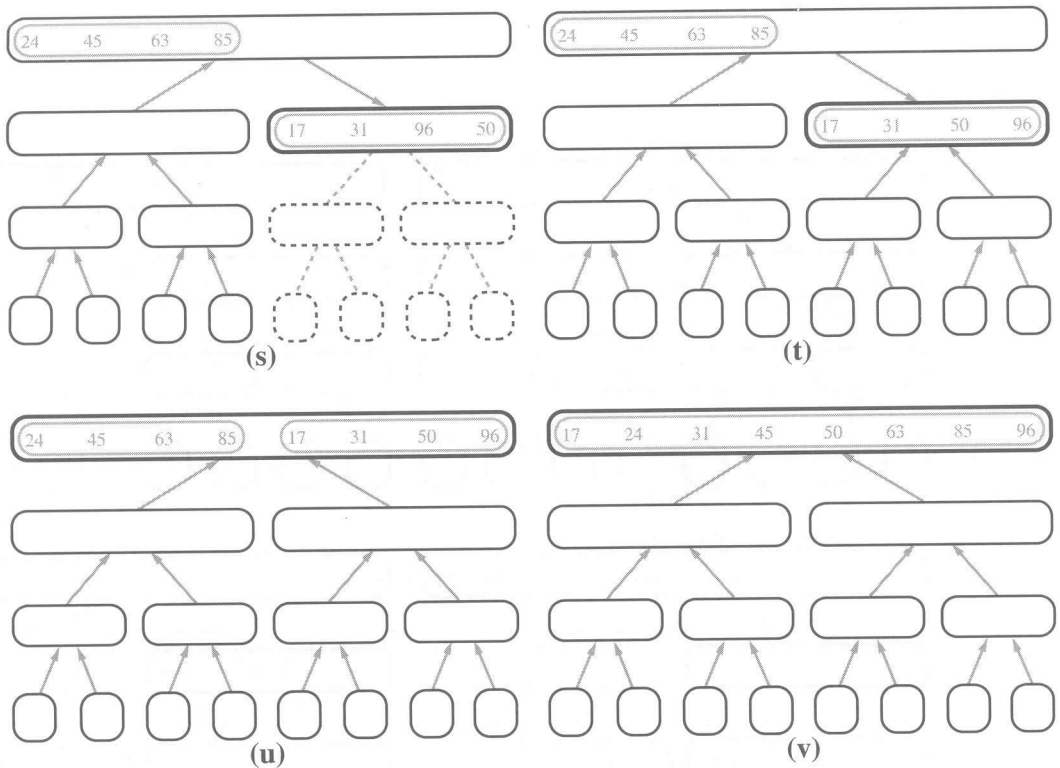


Figura 10.5: (Continuación de la figura 10.4.) Visualización de una ejecución del ordenamiento merge. Se omiten varias invocaciones entre (s) y (t). Nótese los pasos vencer ejecutados en (t) y en (v).

Proposición 10.1: *El árbol de ordenamiento merge asociado con una ejecución de ordenamiento merge con una secuencia de tamaño n tiene la altura $\lceil \log n \rceil$.*

Se deja la justificación de esta proposición para el ejercicio sencillo R-10.1. Se usará esta proposición para analizar el tiempo de ejecución del algoritmo de ordenamiento merge.

Habiendo presentado una perspectiva del ordenamiento merge, y una ilustración de cómo funciona, consideremos ahora con más detalle cada uno de los pasos de este algoritmo divide y vencerás. Los pasos para dividir y repetir del algoritmo de ordenamiento merge son sencillos; la división de una secuencia de tamaño n implica separarla en el elemento con rango $\lceil n/2 \rceil$ y las llamadas recursivas sólo implican pasar estas secuencias menores como parámetros. El paso difícil es el de conquistar, que reúne dos secuencias ordenadas en una sola secuencia ordenada. Así, antes de presentar el análisis del ordenamiento merge, se necesita describir más cómo se hace.

Reunión de dos secuencias ordenadas

El algoritmo *merge*, en el fragmento de programa 10.1, reúne dos secuencias ordenadas, S_1 y S_2 quitando en forma iterativa el elemento mínimo de las dos secuencias y agregándolo al final de la secuencia de salida, S , hasta que una de las dos secuencias queda vacía, y en ese punto se copia el resto de la otra secuencia en la secuencia de salida.

Se analiza el tiempo de ejecución del algoritmo *merge* después de unas observaciones sencillas. Sean n_1 y n_2 las cantidades de elementos de S_1 y S_2 , respectivamente. También, supóngase que las secuencias S_1 , S_2 y S se implementan para que el acceso, la inserción y la eliminación de sus posiciones primera y última tarden el tiempo $O(1)$, cada una. Éste es el caso de las implementaciones basadas en arreglos circulares, o en listas doblemente enlazadas (sección 5.3). El algoritmo *merge* tiene tres ciclos **while**. Debido a las hipótesis, las operaciones ejecutadas en el interior de cada ciclo ocupan el tiempo $O(1)$ cada una. La observación clave es que durante cada iteración de uno de los ciclos, se quita un elemento de S_1 o de S_2 . Como no se hacen inserciones en S_1 o S_2 , esta observación implica que la cantidad total de iteraciones de los tres ciclos es $n_1 + n_2$. Por consiguiente, el tiempo de ejecución del algoritmo *merge* es $O(n_1 + n_2)$, y el resumen es:

Proposición 10.2: *La reunión de dos secuencias ordenadas S_1 y S_2 tarda el tiempo $O(n_1 + n_2)$, siendo n_1 el tamaño de S_1 y n_2 el tamaño de S_2 .*

Algoritmo merge(S_1, S_2, S):

Entrada: Secuencias S_1 y S_2 en orden no decreciente, y una secuencia vacía S

Salida: Secuencia S que entiende los elementos de S_1 y S_2 en orden no decreciente, con las secuencias S_1 y S_2 vacías

while S_1 is not empty **and** S_2 is not empty **do**

if S_1 .first().element() \leq S_2 .first().element() **then**

 { pasar el primer elemento de S_1 al final de S }

S .insertLast(S_1 .remove(S_1 .first()))

else

 { pasar el primer elemento de S_2 al final de S }

S .insertLast(S_2 .remove(S_2 .first()))

 { pasar los elementos restantes de S_1 a S }

while S_1 is not empty **do**

S .insertLast(S_1 .remove(S_1 .first()))

 { pasar los elementos restantes de S_2 a S }

while S_2 is not empty **do**

S .insertLast(S_2 .remove(S_2 .first()))

Fragmento de programa 10.1: Algoritmo *merge* para reunir a dos secuencias ordenadas.

En la figura 10.6 se muestra un ejemplo de ejecución del algoritmo *merge*.

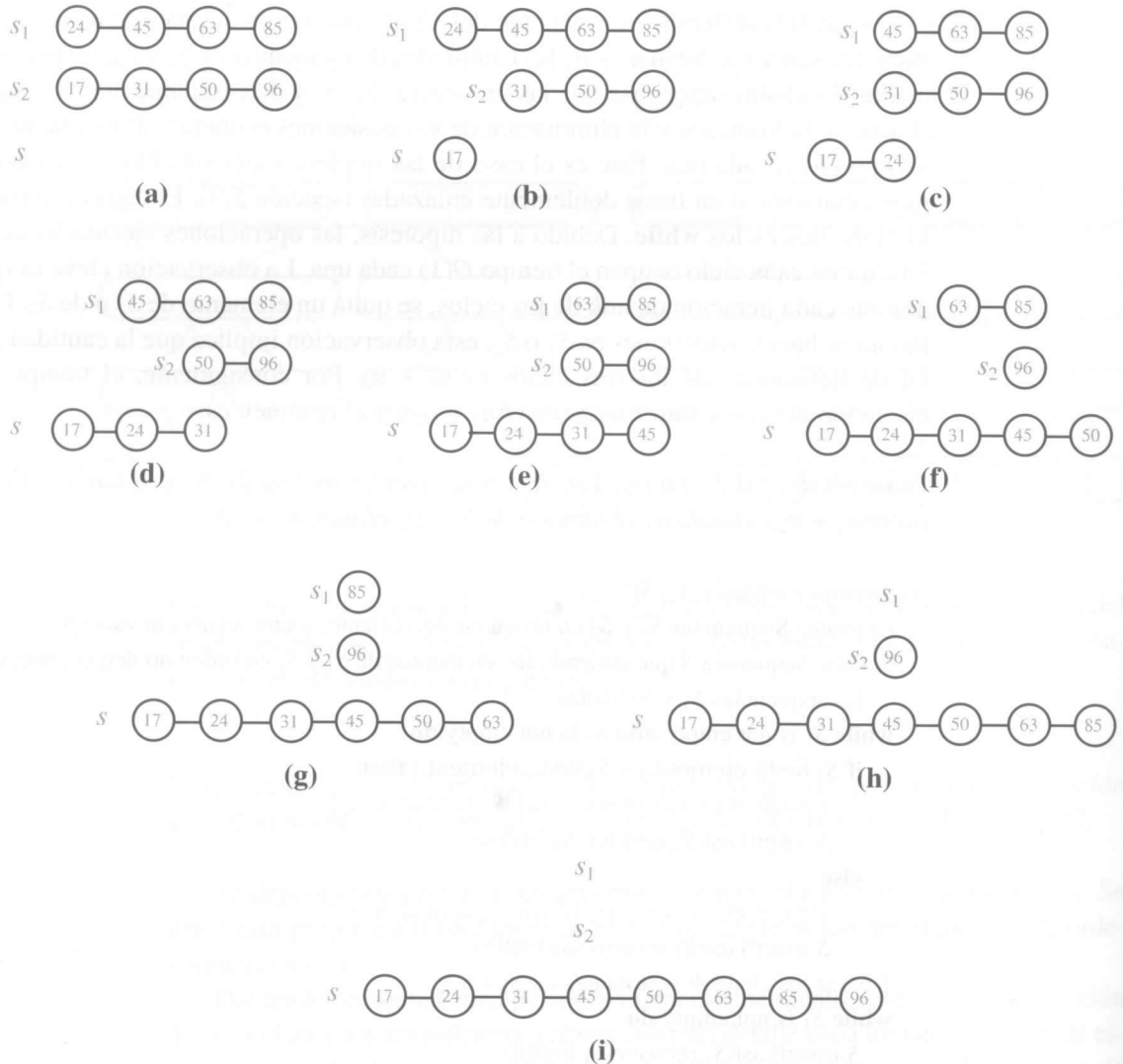


Figura 10.6: Ejemplo de la ejecución del algoritmo *merge* del fragmento de programa 10.1.

El tiempo de ejecución del ordenamiento *merge*

Ahora que se han explicado los detalles del algoritmo de ordenamiento *merge*, y se ha analizado el tiempo de ejecución del fundamental algoritmo *merge* que se usa en el paso de vencer, analicemos el tiempo de ejecución de todo el algoritmo de ordenamiento *merge*, suponiendo que el dato es una secuencia de n elementos. Para simplificar, se restringe la atención al caso en que n es una potencia de 2. Se deja para el ejercicio R-10.4 mostrar que el resultado de este análisis también es válido cuando n no es una potencia de 2.

Como se hizo en el análisis del algoritmo *merge*, suponemos que la secuencia de entrada S , y las secuencias auxiliares S_1 y S_2 , creadas por cada llamada recursiva al ordenamiento *merge*, se implementan de tal manera que el acceso, la inserción y la eliminación de las posiciones primera y última de la secuencia ocupan cada una el tiempo $O(1)$. Es el caso de las implementaciones basadas en un arreglo circular y una lista doblemente enlazada (sección 5.3) o bien, cuando se usa una de estas estructuras de datos en forma directa.

Como se mencionó antes, se analiza el procedimiento de ordenamiento *merge* con base en el árbol de ordenamiento *merge* T (recuérdense las figuras 10.2 a 10.5). Se llama **tiempo ocupado en un nodo** v de T al tiempo de ejecución de la llamada recursiva asociada con v , excluyendo el tiempo ocupado en la espera de las llamadas recursivas asociadas con los hijos de v para terminar. En otras palabras, el tiempo ocupado en el nodo v incluye los tiempos de ejecución de los pasos de dividir y vencer, pero excluye el tiempo de ejecución del paso de recursión. Ya se ha observado que los detalles del paso de dividir son directos; este paso se ejecuta en el tiempo proporcional al tamaño de la secuencia para v . También, de acuerdo con la proposición 10.2, el paso de conquista que consiste en reunir dos subsecuencias ordenadas ocupa también un tiempo lineal. Esto es, si i representa la profundidad del nodo v , el tiempo ocupado en el nodo v es $O(n/2^i)$ porque el tamaño de la secuencia manejada por la llamada recursiva asociada con v es igual a $n/2^i$.

En un examen global del árbol T , como se muestra en la figura 10.7, se aprecia que, dada la definición de “tiempo ocupado en un nodo”, el tiempo de ejecución del ordenamiento *merge* es igual a la suma de los tiempos ocupados en los nodos de T . Obsérvese que T tiene exactamente 2^i nodos en la profundidad i . Esta sencilla observación tiene una consecuencia importante porque implica que el tiempo total ocupado en todos los nodos de T en la profundidad i es $O(2^i \cdot n/2^i)$, que es $O(n)$. De acuerdo con la proposición 10.1, la altura de T es $\lceil \log n \rceil$. Así, como el tiempo ocupado en cada uno de los $\lceil \log n \rceil + 1$ niveles de T es $O(n)$, se ha llegado al siguiente resultado:

Proposición 10.3: *El algoritmo de ordenamiento merge ordena una secuencia de tamaño n en el tiempo $O(n \log n)$, en el peor de los casos.*

En otras palabras, el algoritmo de ordenamiento *merge* coincide asintóticamente con el rápido tiempo de ejecución del algoritmo de ordenamiento *heap*.

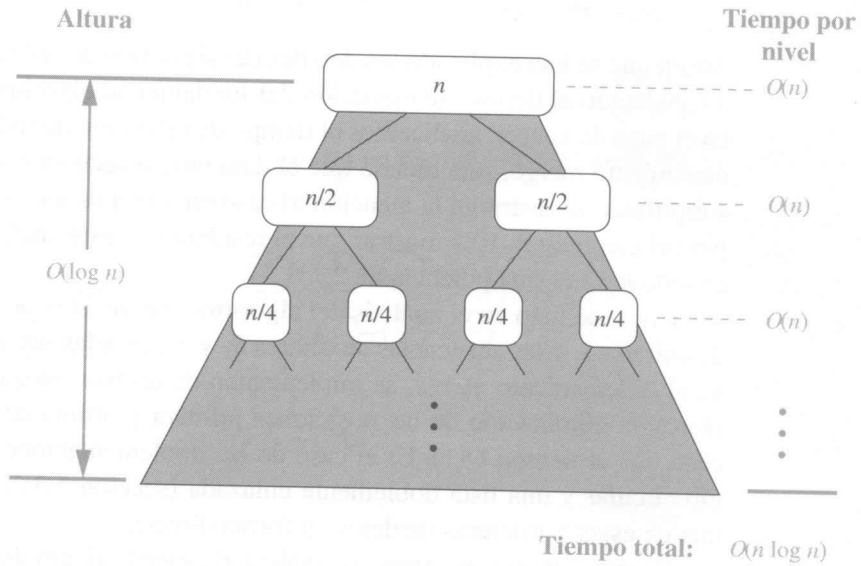


Figura 10.7: Análisis visual del tiempo del árbol T de ordenamiento *merge*. Se muestra cada nodo identificado con el tamaño del subproblema.

10.1.2 Implementación del ordenamiento *merge* en Java

En el fragmento de programa 10.2 se muestra una implementación completa del algoritmo de ordenamiento *merge* en Java. El método `mergeSort` es recursivo y llama al método `merge` para hacer el paso de conquista. Se usa un comparador (véase la sección 7.1.4) para decidir el orden relativo de dos elementos en el método `merge`.

En esta implementación, la secuencia de entrada S y las secuencias auxiliares $S1$ y $S2$ se modifican con las inserciones y eliminaciones sólo en la cabeza y en la cola. Por consiguiente, cada una de esas actualizaciones tarda el tiempo $O(1)$ en una secuencia implementada con una lista doblemente enlazada, o con un arreglo en forma circular (véase la tabla 5.4). Es el caso de la clase `NodeSequence` (véase el fragmento de programa 5.12) que se usa para las secuencias auxiliares.

Por lo tanto, para una secuencia S de tamaño n , el método `mergeSort(S, c)` se ejecuta en el tiempo $O(n \log n)$, siempre que se satisfagan las dos condiciones siguientes:

1. Que se implemente S con una lista doblemente enlazada o con un arreglo usado en forma circular.
2. Que el comparador c pueda comparar dos elementos de S en el tiempo $O(1)$.


```

/**
 * Ordenar los elementos de la secuencia S en orden no decreciente de acuerdo
 * con el comparador c, usando el algoritmo de ordenamiento merge.
 */
public static void mergeSort (Sequence S, Comparator c) {
    int n = S.size();
    // una secuencia con 0 o 1 elementos ya está ordenada
    if (n < 2) return;
    // dividir
    int i = n;
    Sequence S1 = new NodeSequence();
    do { // pasar a S1 los primeros n/2 elementos
        S1.insertLast(S.remove(S.first()));
        i--;
    } while (i > n/2);
    Sequence S2 = new NodeSequence();
    do { // pasar los n/2 elementos restantes a S2
        S2.insertLast(S.remove(S.first()));
        i--;
    } while (i > 0); // la secuencia S ya está vacía
    // recursión
    mergeSort(S1,c);
    mergeSort(S2,c);
    // vencer
    merge(S1,S2,c,S);
}

/**
 * Reunir dos secuencias ordenadas, S1 y S2, en una secuencia ordenada S.
 */
public static void merge(Sequence S1, Sequence S2, Comparator c, Sequence S) {
    while(!S1.isEmpty() && !S2.isEmpty())
        if(c.isLessThanOrEqualTo(S1.first().element(), S2.first().element()))
            S.insertLast(S1.remove(S1.first()));
        else
            S.insertLast(S2.remove(S2.first()));
    while(!S1.isEmpty()) // pasar los elementos restantes de S1
        S.insertLast(S1.remove(S1.first()));
    while(!S2.isEmpty()) // pasar los elementos restantes de S2
        S.insertLast(S2.remove(S2.first()));
}

```

Fragmento de programa 10.2: Métodos `mergeSort` y `merge` para implementar el algoritmo de ordenamiento *merge*.

10.1.3 Ordenamiento *merge* y relaciones de recurrencia ★

Hay otra forma de justificar que el tiempo de ejecución del algoritmo de ordenamiento *merge* es $O(n \log n)$ (proposición 10.3). Es decir, hay una justificación que maneja en forma más directa la naturaleza recursiva de ese algoritmo. En esta sección se presenta ese análisis del tiempo de ejecución y al hacerlo se introduce el concepto matemático de **relación de recurrencia**.

Sea la función $t(n)$ que representa el tiempo de ejecución del ordenamiento *merge* en el peor de los casos, con una secuencia de entrada de tamaño n . Como el ordenamiento *merge* es recursivo, se puede caracterizar la función $t(n)$ mediante las siguientes igualdades, en las que la función $t(n)$ se expresa en forma recursiva en términos de sí misma, como sigue:

$$t(n) = \begin{cases} b & \text{si } n \leq 1 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{en cualquier otro caso} \end{cases}$$

en donde $b \geq 1$ y $c \geq 1$ son constantes. A una caracterización de una función como la de arriba se le llama **relación de recurrencia**, porque la función aparece tanto a la izquierda como a la derecha del signo igual. Aunque esa caracterización es correcta y exacta, lo que realmente se quiere es una caracterización de $t(n)$ que sea del tipo de O mayúscula, que no implique a la misma función $t(n)$ (esto es, se desea tener una caracterización de $t(n)$ **en forma cerrada**).

Para llegar a una caracterización de $t(n)$ en forma cerrada, se restringe la atención al caso en que n es una potencia de 2, y se deja como ejercicio el problema de mostrar que la caracterización asintótica obtenida sigue siendo válida en el caso general. Cuando n es una potencia de 2, se puede simplificar la definición de $t(n)$ a

$$t(n) = \begin{cases} b & \text{si } n \leq 1 \\ 2t(n/2) + cn & \text{en cualquier otro caso} \end{cases}$$

pero todavía así se debe caracterizar esta ecuación de recurrencia en una forma cerrada. Una forma de hacerlo es aplicar esta ecuación en forma iterativa, suponiendo que n es relativamente grande. Por ejemplo, después de una aplicación más de esta ecuación se puede escribir una nueva recurrencia para $t(n)$ como sigue:

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2t(n/2^2) + 2(cn/2) + cn \\ &= 2^2t(n/2^2) + 2cn \end{aligned}$$

Si se aplica de nuevo la ecuación, se obtiene

$$t(n) = 2^3t(n/2^3) + 3cn$$

En este momento se observa un patrón en el comportamiento tal que, después de aplicar la ecuación i veces, se obtiene:

$$t(n) = 2^i t(n/2^i) + icn$$

El asunto que queda entonces es determinar cuándo detener este proceso. Para entenderlo, recuérdese que se cambia a la forma cerrada $t(n) = b$ cuando $n \leq 1$, y eso sucederá cuando $2^i = n$. En otras palabras, eso ocurrirá cuando $i = \log n$. Se hace esta sustitución y se llega a

$$\begin{aligned} t(n) &= 2^{\log n} t(n/2^{\log n}) + (\log n)cn \\ &= nt(1) + cn \log n \\ &= nb + cn \log n \end{aligned}$$

Esto es, se obtiene una justificación alternativa del hecho que $t(n)$ es $O(n \log n)$.

En la siguiente sección se muestra cómo se pueden usar el ordenamiento y el algoritmo *merge* para implementar un tipo de dato abstracto para conjuntos.

10.2 Tipo de dato abstracto conjunto

En esta sección se introduce el TDA *conjunto*. Un *conjunto* es un contenedor de objetos distintos. Esto es, no hay elementos duplicados en un conjunto y no hay una noción explícita de claves, ni siquiera de orden. Aun así, se incluye esta descripción de conjuntos aquí, en un capítulo dedicado al ordenamiento, porque el ordenamiento puede desempeñar un papel básico en las implementaciones eficientes de las operaciones del TDA conjunto.

Primero, recuérdense las definiciones matemáticas de *unión*, *intersección* y *diferencia* de dos conjuntos A y B :

$$A \cup B = \{x: x \in A \text{ o } x \in B\}$$

$$A \cap B = \{x: x \in A \text{ y } x \in B\}$$

$$A - B = \{x: x \in A \text{ y } x \notin B\}$$

Ejemplo 10.4: La mayor parte de los motores de búsqueda en Internet guardan, para cada palabra x en su base de datos de diccionario, un conjunto $W(x)$ de páginas Web que contienen a x , y cada página Web se identifica con una dirección única de Internet. Cuando se les hace una consulta sobre una palabra x , el motor de búsqueda sólo tiene que regresar las páginas Web en el conjunto $W(x)$, ordenadas de acuerdo con una clasificación patentada de prioridad de la "importancia" de la página. Pero cuando la consulta es de dos palabras, x y y , ese motor de búsqueda debe calcular primero la intersección $W(x) \cap W(y)$, para regresar entonces las páginas Web en el conjunto resultante ordenado por prioridades. Muchos motores de búsqueda usan, para efectuar esta intersección, el algoritmo que se describe en esta sección.

Los métodos fundamentales del TDA conjunto, que actúan sobre el conjunto A , son los siguientes:

- union(B):** Reemplaza a A con la unión de A y B , esto es, ejecutar $A \leftarrow A \cup B$.
Entrada: conjunto; **Salida:** ninguna.
- intersect(B):** Reemplaza a A con la intersección de A y B , esto es, ejecutar $A \leftarrow A \cap B$.
Entrada: conjunto; **Salida:** ninguna.
- subtract(B):** Reemplaza a A con la diferencia de A y B , esto es, ejecutar $A \leftarrow A - B$.
Entrada: conjunto; **Salida:** ninguna.

Se han definido las operaciones union, intersect y subtract arriba, de tal modo que modifican el contenido del conjunto A implicado. También se podrían haber definido esos métodos de modo que no se modifica A , sino que se regresa un conjunto nuevo.

El paquete java.util incluye una interfaz Set. Mantiene la propiedad de que un conjunto no puede contener elementos duplicados, e incluye métodos parecidos a los que se definieron arriba. Se muestra en la tabla 10.1, la correspondencia entre los métodos del minimalista TDA conjunto, de arriba, y los métodos de la interfaz java.util.set.

Métodos de TDA conjunto	Métodos de java.util.Set
union(B)	addAll(Collection B)
intersect(B)	retainAll(Collection B)
subtract(B)	removeAll(Collection B)

Tabla 10.1: Correspondencia entre los métodos del TDA conjunto y los de la interfaz java.util.Set. La última también soporta algunos otros métodos.

10.2.1 Implementación sencilla de un conjunto

Una de las formas más sencillas de implementar un conjunto es guardar sus elementos en una secuencia ordenada. En realidad, esta preferencia de implementación está implícita en la interfaz java.util.SortedSet, que extiende a java.util.Set e impone un ordenamiento del contenido del conjunto. Aun cuando no se proporciona un comparador, los elementos mismos pueden tener una relación natural de orden definida en ellos. En general, siempre debería ser posible imponer un orden en los elementos. Por ejemplo, se podrían usar las direcciones de los objetos en la memoria. Por consiguiente, se describe la implementación del TDA conjunto con una secuencia ordenada, y otras implementaciones se examinan en varios ejercicios.

Se implementa cada una de las tres operaciones fundamentales de los conjuntos usando una versión genérica del algoritmo *merge* que toma, como entrada, dos secuencias ordenadas que representan los conjuntos de entrada, y forma una secuencia que representa el conjunto de salida, sea éste la unión, la intersección o la diferencia de los conjuntos de entrada. El ordenamiento de las dos secuencias se puede basar en cualquier regla consistente de ordenamiento (esto es, un orden total), siempre que se use la misma regla de ordenamiento para los dos conjuntos con los que se desea efectuar la unión, intersección o diferencia. En el fragmento de programa 10.3 se describe con detalle el algoritmo genérico *merge*.

Algoritmo genericMerge(*A*,*B*):

Entrada: Sets represented by sorted sequences *A* and *B*

Salida: Set represented by a sorted sequence *C*

{No se destruirán *A* y *B*}

let *A'* be a copy of *A*

let *B'* be a copy of *B*

while *A'* and *B'* are not empty **do**

$a \leftarrow A'.\text{first}()$

$b \leftarrow B'.\text{first}()$

if $a < b$ **then**

 alsLess(a, C)

$A'.\text{removeFirst}()$

else if $a = b$ **then**

 bothAreEqual(a, b, C)

$A'.\text{removeFirst}()$

$B'.\text{removeFirst}()$

else

 blsLess(b, C)

$B'.\text{removeFirst}()$

while *A'* is not empty **do**

$a \leftarrow A'.\text{first}()$

 alsLess(a, C)

$A'.\text{removeFirst}()$

while *B'* is not empty **do**

$b \leftarrow B'.\text{first}()$

 blsLess(b, C)

$B'.\text{removeFirst}()$

Fragmento de programa 10.3: Algoritmo *merge* genérico, parametrizado por los métodos alsLess, bothAreEqual y blsLess, que determinan la composición del conjunto *C*. Nótese que este algoritmo no modifica las secuencias de entrada.

El método *merge* genérico examina y compara, en forma iterativa, los elementos actuales a y b de las secuencias A y B , respectivamente, y determina si $a < b$, $a = b$ o $a > b$. A continuación, según el resultado de esa comparación, determina si debe copiar uno o ninguno de los elementos a y b del extremo de la secuencia C de salida. Esta determinación se hace con base en la operación particular que se esté efectuando, sea unión, intersección o diferencia.

Por ejemplo, en una operación de unión se copia el menor de a y b en la secuencia de salida; si son iguales sólo se copia uno (por ejemplo, a). Este procedimiento asegura que se copiará cada elemento que haya en uno de los dos conjuntos, sin crear elementos duplicados. Se especifican las acciones de copia por hacer con los **métodos auxiliares** *alsLess*, *bothAreEqual* y *blsLess*, que se definen dependiendo de la operación que se desea efectuar.

Merge genérico como patrón con método de plantilla

El algoritmo genérico *merge* se basa en el **patrón con método de plantilla** (véase la sección 6.3.5). El patrón con método de plantilla es un patrón de diseño de ingeniería de programación que describe a un mecanismo genérico de cómputo que se puede especializar redefiniendo ciertos pasos. En este caso se describe un método que reúne a dos secuencias en una, y que se puede especializar de acuerdo con el comportamiento de tres métodos abstractos.

El fragmento de programa 10.4 muestra la clase *Merger* (combinador) que proporciona una implementación del algoritmo *merge* genérico en Java. Para convertir la clase genérica *Merger* en clases útiles, se debe extender con clases que redefinan los tres métodos auxiliares, *alsLess*, *bothAreEqual* y *blsLess*. Se muestra cómo se pueden describir con facilidad cada una de las operaciones de unión, intersección y diferencia, en términos de esos métodos, en el fragmento de programa 10.5. Los métodos auxiliares se redefinen de tal modo que el método plantilla *merge* haga las siguientes acciones:

- En la clase *UnionMerger*, *merge* copia a todo elemento de A y B en C , sin duplicar a ningún elemento.
- En la clase *IntersectMerger*, *merge* copia en C todos los elementos que están en A y en B , pero “descarta” cualquier elemento que esté en un conjunto pero no en el otro. En la clase *SubtractMerger*, *merge* copia en C a todos los elementos que están en A pero no están en B .

También se podría extender el TDA conjunto agregando métodos como *insert* y *remove*, para insertar y quitar elementos de un conjunto. Esos métodos se podrían implementar en el sistema de secuencia ordenada, con barridos sencillos de la secuencia que representa a un conjunto, lo que se puede hacer con facilidad en el tiempo $O(n)$ como se determina en el ejercicio R-10.7.

```

/** Merge genérico para secuencias ordenadas. */
public abstract class Merger {
    private Object a, b;      // elementos actuales en A y en B
    private ObjectIterator iterA, iterB;    // iteradores para A y B
    /** Método de plantilla de merge genérico */
    public void merge(Sequence A, Sequence B, Comparator comp, Sequence C) {
        iterA = A.elements();
        iterB = B.elements();
        boolean aExists = advanceA(); // Prueba booleana para ver si hay una a actual
        boolean bExists = advanceB(); // Prueba booleana para ver si hay una b actual
        while (aExists && bExists) {    // Ciclo principal para reunir a y b
            if (comp.isLessThan(a, b)) {
                alsLess(a, C); aExists = advanceA();
            }
            else if (comp.isEqualTo(a,b)) {
                bothAreEqual(a, b, C);
                aExists = advanceA(); bExists = advanceB();
            }
            else {
                blsLess(b, C); bExists = advanceB();
            }
        }
        while (aExists) { alsLess(a, C); aExists = advanceA(); }
        while (bExists) { blsLess(b, C); bExists = advanceB(); }
    }
    // métodos auxiliares para ser especializados por subclases
    protected void alsLess(Object a, Sequence C) { }
    protected void bothAreEqual(Object a, Object b, Sequence C) { }
    protected void blsLess(Object b, Sequence C) { }
    // métodos auxiliares
    private boolean advanceA() {
        if (iterA.hasNext()) {
            a = iterA.nextObject();
            return true;
        }
        return false;
    }
    private boolean advanceB() {
        if (iterB.hasNext()) {
            b = iterB.nextObject();
            return true;
        }
        return false;
    }
}

```

Fragmento de programa 10.4: Clase Merger que implementa al algoritmo *merge* genérico.

```

/** Clase que especializa la plantilla generic merge para la unión de dos conjuntos */
public class UnionMerger extends Merger {
    protected void alsLess(Object a, Sequence C) {
        C.insertLast(a);    // agregar a
    }
    protected void bothAreEqual(Object a, Object b, Sequence C) {
        C.insertLast(a);    // agregar a (pero no su duplicado b)
    }
    protected void blsLess(Object b, Sequence C) {
        C.insertLast(b);    // agregar b
    }
}

/** Clase para especializar la plantilla generic merge para intersección de dos conjuntos */
public class IntersectMerger extends Merger {
    protected void alsLess(Object a, Sequence C) {}
    protected void bothAreEqual(Object a, Object b, Sequence C) {
        C.insertLast(a);    // agregar a (pero no su duplicado b)
    }
    protected void blsLess(Object b, Sequence C) {}
}

/** Clase que especializa la plantilla generic merge para diferencia de dos conjuntos */
public class SubtractMerger extends Merger {
    protected void alsLess(Object a, Sequence C) {
        C.insertLast(a);    // agregar a
    }
    protected void bothAreEqual(Object a, Object b, Sequence C) {}
    protected void blsLess(Object b, Sequence C) {}
}

```

Fragmento de programa 10.5: Clases que extienden la clase Merger especializando los métodos auxiliares para que ejecuten la unión, intersección y diferencia de conjuntos, respectivamente.

Desempeño del *merge* genérico

A continuación se analiza el tiempo de ejecución del algoritmo *merge* genérico. En cada iteración de uno de los ciclos, se quita un elemento de A , de B o de ambos. Suponiendo que las comparaciones tardan el tiempo $O(1)$ y que cada llamada a un método auxiliar tarde el tiempo $O(1)$, el tiempo total de ejecución de $\text{genericMerge}(A, B)$ es $O(n_A + n_B)$, siendo n_A el tamaño de A y n_B el tamaño de B , esto es, *merge* ocupa el tiempo proporcional a la cantidad de elementos que intervienen. Por consiguiente:

Proposición 10.5: *El TDA conjunto se puede implementar con una secuencia ordenada y un esquema de merge genérico que soporte las operaciones union, intersect y subtract en el tiempo $O(n)$, siendo n la suma de los tamaños de los conjuntos que intervienen.*

10.3 Quick-Sort

El siguiente algoritmo de ordenamiento que se describe es el llamado **ordenamiento rápido (quick-sort)**. Al igual que el ordenamiento *merge*, se basa en el paradigma de *divide y vencerás*, pero usa esta técnica en forma algo contraria porque todo el trabajo duro se hace *antes* de las llamadas recursivas.

Descripción del ordenamiento rápido en alto nivel

El algoritmo de ordenamiento rápido ordena una secuencia S mediante un sistema recursivo sencillo. El concepto principal es aplicar la técnica *divide y vencerás*, en la que se divide S en subsecuencias, se hace recursión para ordenar cada subsecuencia y a continuación se combinan las secuencias ordenadas mediante una simple concatenación. En particular, el algoritmo de ordenamiento rápido consiste en los tres pasos siguientes (véase la figura 10.8):

1. **Dividir:** Si S tiene al menos dos elementos (nada hay que hacer si S tiene cero o un elemento), seleccionar un elemento específico x de S , que se llama **pivote**. Lo que se acostumbra es escoger al pivote x como el último elemento de S . Se quitan todos los elementos de S y se ponen en tres secuencias:

- L , que guarda los elementos en S menores que x
- E , que guarda los elementos en S iguales a x
- G , que guarda los elementos en S mayores que x

Es claro que si todos los elementos de S son distintos, entonces E guarda sólo un elemento: el pivote mismo.

2. **Recursión:** Clasificar las secuencias L y G recursivamente.
3. **Conquistar:** Regresar los elementos a S , en orden, insertando primero los elementos de L , después los de E y por último los de G .

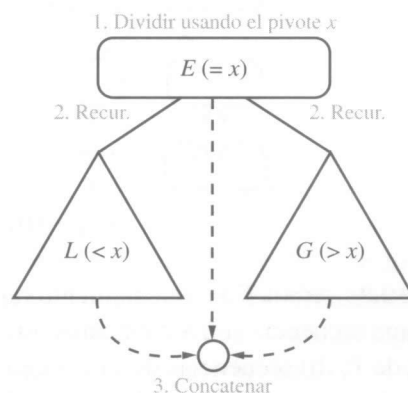
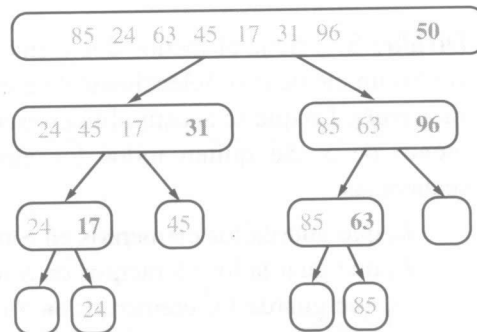


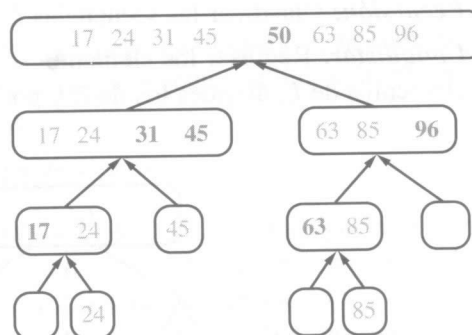
Figura 10.8: Esquema del algoritmo de *quick-sort*.

Al igual que el ordenamiento *merge*, la ejecución del ordenamiento rápido se puede visualizar por medio de un árbol binario de recursión, llamado **árbol de ordenamiento rápido**. La figura 10.9 resume una ejecución del algoritmo de ordenamiento rápido, mostrando las secuencias de entrada y salida procesadas en cada nodo del árbol de ordenamiento rápido. La evolución, paso a paso, del árbol de ordenamiento rápido se muestra en las figuras 10.10, 10.11 y 10.12.

Sin embargo, a diferencia del ordenamiento *merge*, la altura del árbol de ordenamiento rápido asociado con una ejecución del ordenamiento rápido es lineal, en el peor de los casos. Eso sucede, por ejemplo, si la secuencia consiste en n elementos distintos y ya está ordenada. En realidad, en este caso, la opción normal del pivote como elemento mayor produce una subsecuencia L de tamaño $n - 1$, mientras que la subsecuencia E tiene el tamaño 1, y la subsecuencia G tiene el tamaño 0. En cada invocación del ordenamiento rápido en la subsecuencia L , el tamaño disminuye en 1. Por lo anterior, la altura del árbol de ordenamiento rápido es $n - 1$.



(a)



(b)

Figura 10.9: Árbol T de ordenamiento rápido para ejecutar el algoritmo de *quick-sort* en una secuencia con 8 elementos: (a) secuencias de entrada procesadas en cada nodo de T ; (b) secuencias de salida generadas en cada nodo de T . Se muestra en negritas el pivote usado en cada nivel de la recursión.

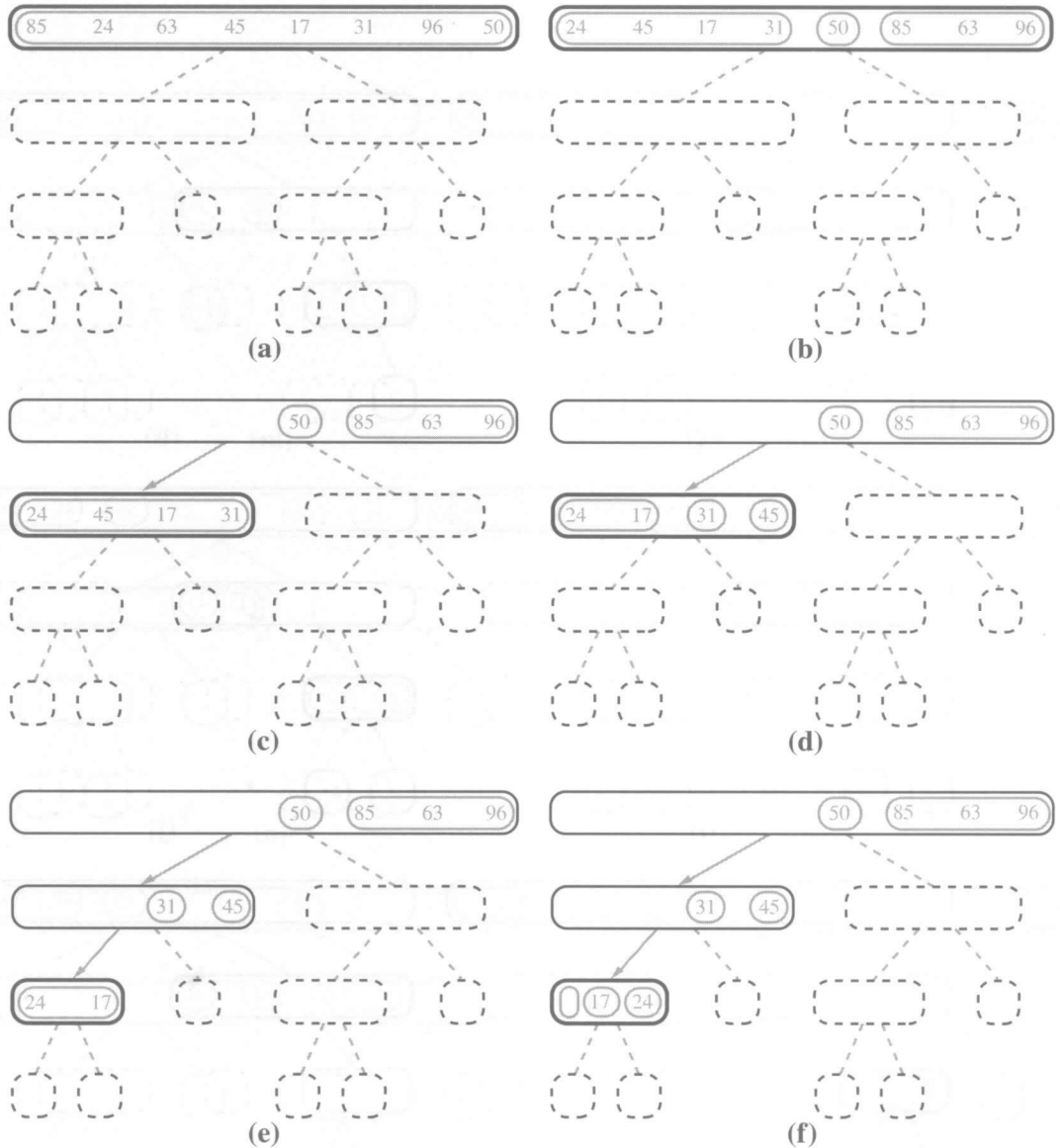


Figura 10.10: Visualización del algoritmo *quick-sort*. Cada nodo del árbol representa una llamada recursiva. Los nodos con líneas interrumpidas representan llamadas que todavía no se han hecho. El nodo con línea gruesa representa la invocación de ejecución. Los nodos vacíos trazados con líneas delgadas representan llamadas terminadas. Los nodos restantes representan llamadas suspendidas (esto es, invocaciones activas que están esperando el regreso de una invocación del hijo). Nótese los pasos dividir ejecutados en (b), (d) y (f). (Continúa en la figura 10.11.)

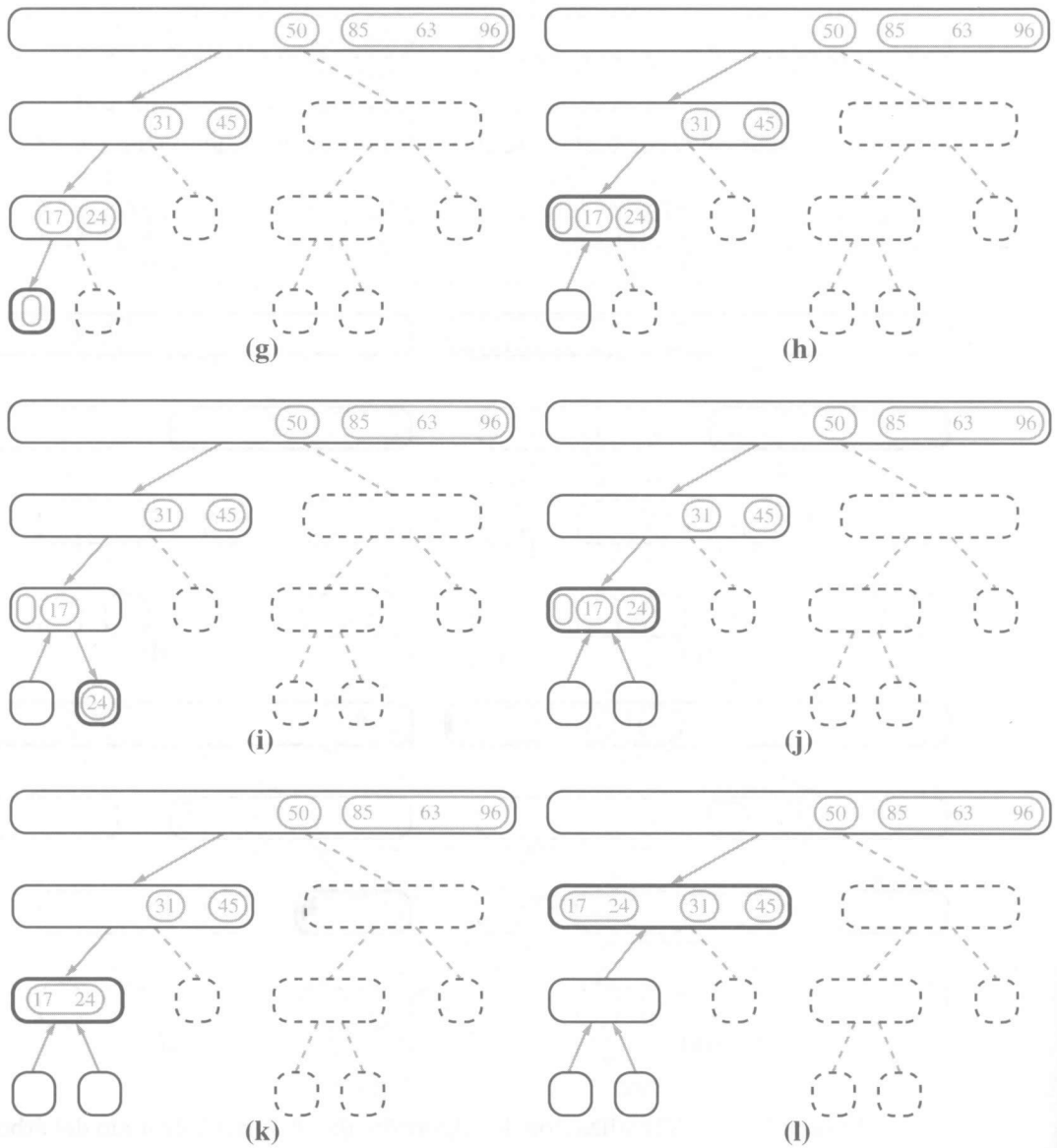


Figura 10.11: (Continuación de la figura 10.10.) Visualización de la ejecución del ordenamiento *quick-sort*. Nótese el paso vencer ejecutado en (k).

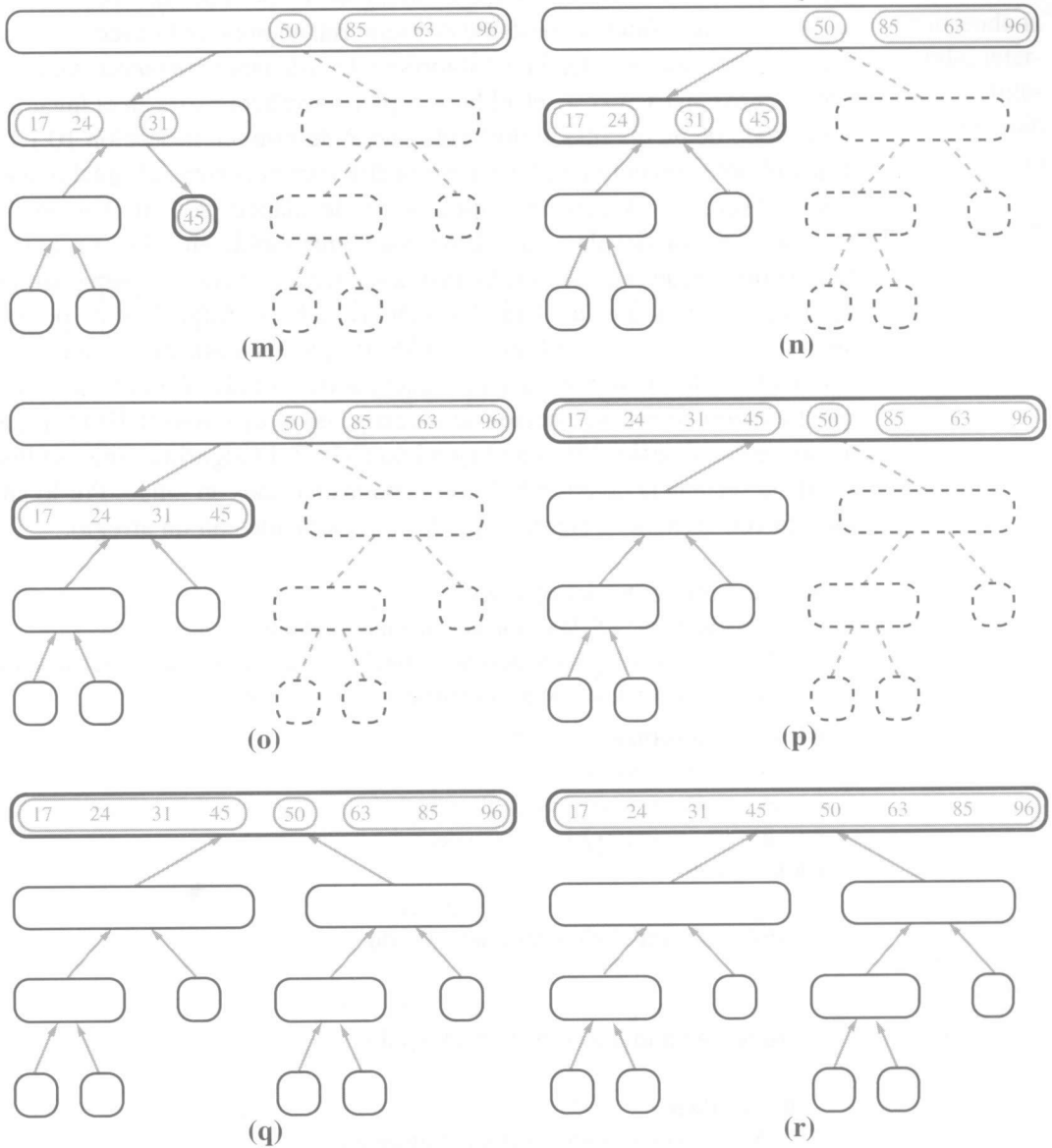


Figura 10.12: (Continuación de la figura 10.11.) Visualización de una ejecución del algoritmo *quick-sort*. Se han omitido varias invocaciones entre (p) y (q). Nótese los pasos vencer ejecutados en (o) y en (r).

10.3.1 Ordenamiento *Quick-Sort* en el lugar

Recuérdese, de la sección 7.3.4, que un algoritmo de ordenamiento está *en el lugar* si sólo usa una cantidad constante de memoria, además de la necesaria para los objetos mismos que se ordenan. El algoritmo de ordenamiento *merge*, tal como se describió líneas antes, no está en el lugar y para hacerlo que esté en el lugar se requiere un método de *merge* más complicado que el descrito en la sección 10.1.1. No obstante, el ordenamiento en el lugar no es difícil en sí porque, al igual que el ordenamiento *heap*, el ordenamiento rápido se puede adaptar para que esté en el lugar.

La ejecución del algoritmo de ordenamiento rápido en el lugar requiere bastante ingenio porque se debe usar la misma secuencia de entrada para guardar las subsecuencias para todas las llamadas recursivas. En el fragmento de programa 10.6 se muestra el algoritmo `inPlaceQuickSort`, que hace esta ejecución. El algoritmo `inPlaceQuickSort` supone que la secuencia de entrada, S , tiene elementos distintos. La razón de esta restricción se investiga en el ejercicio R-10.11. La extensión al caso general se describe en el ejercicio C-10.7. El algoritmo ingresa los elementos de la secuencia de entrada S con métodos basados en rango. Por lo anterior, se ejecuta con eficiencia, siempre que S se implemente con un arreglo.

Algoritmo `inPlaceQuickSort(S, a, b)`:

Entrada: Secuencia S de elementos distintos; enteros a y b

Salida: Secuencia S con elementos originalmente en rangos de a a b , inclusive, en orden no decreciente de los rangos a a b .

```

if  $a \geq b$  then return      {subrango vacío}
 $p \leftarrow S.\text{elemAtRank}(b)$     {pivote}
 $l \leftarrow a$       {barre hacia la derecha}
 $r \leftarrow b - 1$     {barre hacia la izquierda}
while  $l \leq r$  do
    {encontrar un elemento mayor que el pivote}
    while  $l \leq r$  and  $S.\text{elemAtRank}(l) \leq p$  do
         $l \leftarrow l + 1$ 
    {encontrar un elemento menor que el pivote}
    while  $r \geq l$  and  $S.\text{elemAtRank}(r) \geq p$  do
         $r \leftarrow r - 1$ 
    if  $l < r$  then
         $S.\text{swapElements}(S.\text{atRank}(l), S.\text{atRank}(r))$ 
    {poner el pivote en su lugar definitivo}
     $S.\text{swapElements}(S.\text{atRank}(l), S.\text{atRank}(b))$ 
    {llamadas recursivas}
    inPlaceQuickSort( $S, a, l - 1$ )
    inPlaceQuickSort( $S, l + 1, b$ )

```

Fragmento de programa 10.6: Ordenamiento rápido en el lugar para una secuencia implementada con un arreglo.

El ordenamiento rápido en el lugar modifica la secuencia de entrada usando operaciones de `swapElements` y no crea subsecuencias en forma explícita. En realidad, una subsecuencia de la secuencia de entrada se representa en forma implícita mediante un intervalo de posiciones definido por un rango l de extrema izquierda, y un rango r de extrema derecha. El paso de dividir se hace barriendo la secuencia en forma simultánea desde l , avanzando y desde r retrocediendo, intercambiando pares de elementos que están en orden inverso, como se ve en la figura 10.13. Cuando se “encuentran” estos dos índices, las subsecuencias L y G están en los lados opuestos del punto de encuentro. El algoritmo se termina haciendo la recursión en esas dos subsecuencias.

El ordenamiento rápido en el lugar reduce el tiempo de ejecución causado por la creación de nuevas secuencias y el movimiento de elementos entre ellas, por un factor constante. En el fragmento de programa 10.7 se muestra una versión en Java del ordenamiento rápido en el lugar.

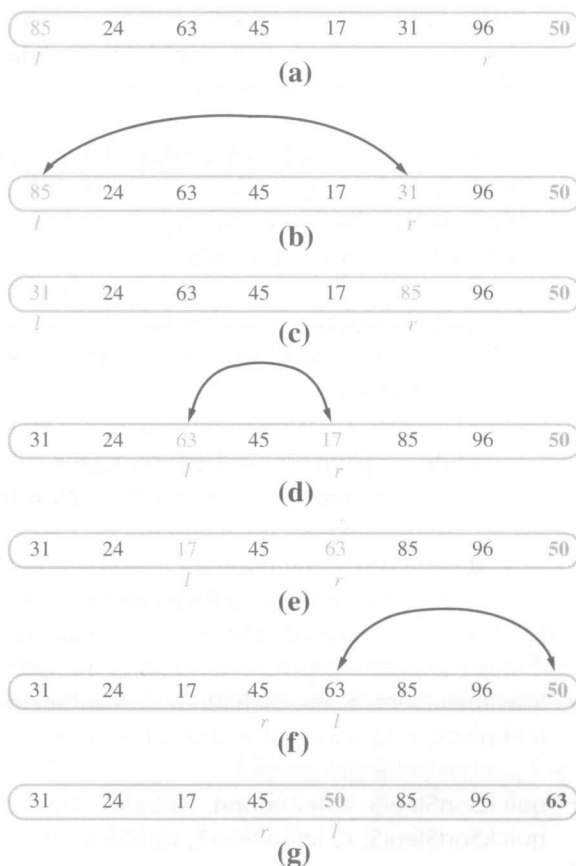


Figura 10.13: Paso dividir del algoritmo *quick-sort* en el lugar. El índice l recorre la secuencia de izquierda a derecha, y el índice r lo hace de derecha a izquierda. Se ejecuta un intercambio cuando l está a un elemento mayor que el pivote, y r a un elemento menor que el pivote. Un intercambio final con el pivote completa el paso dividir.


```

/**
 * Ordenar los elementos de la secuencia S en orden no decreciente, de acuerdo
 * con el comparador c, con el algoritmo de quick-sort. La mayor parte
 * del trabajo lo hace el método recursivo auxiliar quickSortStep.
 */
public static void quickSort (Sequence S, Comparator c) {
    if (S.size() < 2)
        return; // una secuencia con 0 o 1 elemento ya está ordenada
    quickSortStep(S, c, 0, S.size()-1); // método recursivo de ordenamiento
}

/**
 * Poner en orden no decreciente los elementos de la secuencia S entre
 * los rangos leftBound y rightBound, con una implementación recursiva,
 * en el lugar, del algoritmo de quick-sort.
 */
private static void quickSortStep (Sequence S, Comparator c,
                                   int leftBound, int rightBound ) {
    if (leftBound >= rightBound)
        return;
    Object pivot = S.atRank(rightBound).element();
    int leftIndex = leftBound; // barre hacia la derecha
    int rightIndex = rightBound-1; // barre hacia la izquierda
    while (leftIndex <= rightIndex) {
        // barrer hacia la derecha para encontrar un elemento mayor que el pivote
        while (leftIndex <= rightIndex) &&
            c.isLessThanOrEqualTo(S.atRank(leftIndex).element(), pivot) )
            leftIndex++;
        // barrer hacia la izquierda para encontrar un elemento menor que el pivote
        while ( rightIndex >= leftIndex) &&
            c.isGreaterThanOrEqualTo(S.atRank(rightIndex).element(), pivot) )
            rightIndex--;
        if (leftIndex < rightIndex) // se encontraron ambos elementos
            S.swapElements(S.atRank(leftIndex), S.atRank(rightIndex));
    } // el ciclo continúa hasta que los índices se cruzan
    // poner el pivote intercambiándolo con el elemento del índice izquierdo
    S.swapElements(S.atRank(leftIndex), S.atRank(rightBound));
    // el pivote está ahora en el índice izquierdo, por consiguiente hacer recursión en
    // ambos lados del mismo
    quickSortStep(S, c, leftBound, leftIndex-1);
    quickSortStep(S, c, leftIndex+1, rightBound);
}

```

Fragmento de programa 10.7: Implementación, en Java, del algoritmo *quick-sort* en el lugar. Se supone que se implementa la secuencia de entrada con un arreglo y que tiene elementos distintos.

Por desgracia, la implementación anterior del ordenamiento rápido no es, hablando en términos técnicos, totalmente en el lugar porque sigue requiriendo más que una cantidad constante de espacio adicional. Naturalmente que no se usa espacio adicional para las subsecuencias y sólo se usa una cantidad constante de espacio adicional para las variables locales (como l y r). Así, ¿de dónde viene ese espacio adicional? Proviene de la recursión porque, si se recuerda la sección 4.1.3, se observa que el espacio que se necesita para una pila es proporcional a la profundidad del árbol de recursión para el ordenamiento rápido, que es cuando menos $\log n$ y cuando mucho $n - 1$. Para hacer que el ordenamiento rápido sea realmente en el lugar, se debe implementar en forma no recursiva (y no usar una pila). El detalle clave en esa implementación es que se necesita una forma, en el lugar, para determinar las cotas de los límites izquierdo y derecho de la subsecuencia “actual”. Ese esquema no es muy difícil y se dejan los detalles de su implementación para el ejercicio C-10.5.

Tiempo de ejecución del ordenamiento *Quick-Sort*

Se puede analizar el tiempo de ejecución del ordenamiento rápido con la misma técnica que se usó para el ordenamiento *merge* en la sección 10.1.1. Es decir, se identifica el tiempo ocupado en cada nodo del árbol T de ordenamiento rápido (figuras 10.10, 10.11 y 10.12) y se suman los tiempos de ejecución para todos los nodos.

Son fáciles de implementar en tiempo lineal el paso de dividir y el de conquistar del ordenamiento rápido. El tiempo ocupado en un nodo v de T es proporcional al **tamaño de la entrada** $s(v)$ de v , que se define como el tamaño de la secuencia manejada por la invocación del ordenamiento rápido asociada con el nodo v . Como la subsecuencia E tiene al menos un elemento (el pivote), la suma de los tamaños de entrada de los hijos de v es $s(v) - 1$, cuando mucho.

Dado un árbol de ordenamiento rápido T , s_i denota la suma de los tamaños de entrada de los nodos a la profundidad i en T . Es claro que $s_0 = n$ porque la raíz r de T se asocia con toda la secuencia. También, $s_1 \leq n - 1$, porque el pivote no se propaga a los hijos de r . A continuación examínese a s_2 . Si los dos hijos de r tienen tamaño de entrada no cero, entonces $s_2 = n - 3$. En caso contrario (un hijo de la raíz tiene tamaño cero y el otro tiene tamaño $n - 1$), $s_2 = n - 2$. Así, $s_2 \leq n - 2$. Se continúa este razonamiento y se obtiene que $s_i \leq n - i$. Como se observó en la sección 10.3, la altura de T es $n - 1$ en el peor de los casos. Por consiguiente, el tiempo de ejecución del ordenamiento rápido, en el peor de los casos, es

$$O\left(\sum_{i=0}^{n-1} s_i\right), \quad \text{que es} \quad O\left(\sum_{i=0}^{n-1} (n-i)\right) \quad \text{esto es,} \quad O\left(\sum_{i=1}^n i\right)$$

De acuerdo con la proposición 3.4, $\sum_{i=1}^n i$ es $O(n^2)$. Por ello el ordenamiento rápido se ejecuta en el tiempo $O(n^2)$ en el peor de los casos.

De acuerdo con su nombre, cabe esperar que el ordenamiento rápido se ejecute con rapidez. Sin embargo, la cota cuadrática anterior indica que en el peor de los casos el ordenamiento rápido es lento. Paradójicamente, este comportamiento en el peor de los casos se presenta en los problemas en los que el ordenamiento debería ser fácil,

cuando la secuencia ya está ordenada. Más aún, se puede demostrar que el ordenamiento rápido se desempeña mal aun cuando la secuencia esté “casi” ordenada.

Regresando al análisis, obsérvese que el mejor de los casos para el ordenamiento rápido de una secuencia de elementos distintos se presenta cuando las subsecuencias L y G tienen más o menos el mismo tamaño. En realidad, en este caso se guarda un pivote en cada nodo interno y se hacen dos llamadas de igual tamaño para sus hijos. Por consiguiente se ahorra un pivote en la raíz, 2 en el nivel 1, 2^2 en el nivel 2, y así sucesivamente. Esto es, en el mejor de los casos

$$\begin{aligned} s_0 &= n \\ s_1 &= n - 1 \\ s_2 &= n - (1 + 2) = n - 3 \\ &\vdots \\ s_i &= n - (1 + 2 + 2^2 + \cdots + 2^{i-1}) = n - (2^i - 1) \\ &\vdots \end{aligned}$$

En el mejor de los casos T tiene la altura $O(\log n)$ y el ordenamiento rápido se ejecuta en el tiempo $O(n \log n)$; se deja la justificación de este hecho para el ejercicio R-10.10.

La intuición informal acerca del comportamiento esperado del ordenamiento rápido es que en cada invocación es probable que el pivote divida la secuencia de entrada en partes más o menos iguales. Así, es de esperar que el tiempo promedio de ejecución del ordenamiento rápido sea parecido al tiempo del mejor de los casos, es decir, a $O(n \log n)$. Se verá en la siguiente sección que al introducir la aleatorización se hace que el ordenamiento rápido se comporte exactamente como se ha descrito.

10.3.2 Ordenamiento *Quick-Sort* aleatorizado

Un método frecuente para analizar el ordenamiento rápido es suponer que el pivote siempre divide la secuencia en partes casi iguales. Se aprecia que esa hipótesis presupone cierto conocimiento sobre la distribución de la entrada, del cual normalmente no se dispone. Por ejemplo, habría que suponer que casi nunca se tendrían secuencias “casi” ordenadas para ordenarlas, y eso es lo que sucede en realidad en muchas aplicaciones. Por fortuna, no se necesita esta hipótesis para hacer que la intuición coincida con el comportamiento del ordenamiento rápido.

Ya que el objetivo del paso de partición en el método de ordenamiento rápido es dividir la secuencia S en partes casi iguales, se introduce aleatorización en el algoritmo y se escoge un *elemento aleatorio* de la secuencia de entrada como pivote. A esta variación del ordenamiento rápido se le llama *ordenamiento rápido aleatorizado*. La siguiente proposición indica que el tiempo esperado de ejecución del ordenamiento rápido aleatorizado en una secuencia con n elementos es $O(n \log n)$. Esta expectativa es respecto a todas las elecciones aleatorias posibles que hace el algoritmo, y es independiente de cualquier hipótesis acerca de la distribución de las secuencias posibles de entrada que deba procesar probablemente el algoritmo.

Proposición 10.6: El tiempo esperado de ejecución del ordenamiento rápido aleatorizado en una secuencia de tamaño n es $O(n \log n)$.

Justificación: Se aplica un hecho sencillo de la teoría de las probabilidades:

La cantidad esperada de veces que se debe lanzar al aire una moneda imparcial, para que salgan k “caras”, es $2k$.

Examinemos ahora una sola invocación recursiva del ordenamiento rápido aleatorizado; sea m el tamaño de la secuencia de entrada para esta invocación. Se dice que esta invocación es “buena” si el pivote se elige de tal manera que las subsecuencias L y G tienen $m/4$ elementos cuando menos, y $3m/4$ cuando mucho, cada una. Así, como el pivote se elige uniformemente al azar, y hay $m/2$ pivotes para los que esta invocación es buena, la probabilidad de que la invocación sea buena es $1/2$, igual que la misma de sacar cara al arrojar una moneda.

Si se asocia un nodo v del árbol de ordenamiento rápido T , como se muestra en la figura 10.14, con una “buena” llamada recursiva, los tamaños de entrada de los hijos de v son $3s(v)/4$ cuando mucho, que es igual a $(s(v)/(4/3))$. Si se sigue cualquier camino en T , desde la raíz hasta un nodo externo, la longitud de esta trayectoria es cuando mucho la cantidad de invocaciones que se deben hacer (en cada nodo de este camino) hasta lograr $\log_{4/3} n$ invocaciones buenas. De acuerdo con el hecho de probabilidad reseñado líneas antes, la cantidad esperada de invocaciones que se deben hacer hasta que eso ocurra es $2\log_{4/3} n$. Por lo anterior, la longitud esperada de cualquier trayectoria de la raíz a un nodo externo en T es $O(\log n)$. Como el tiempo ocupado en cada nivel de T es $O(n)$, el tiempo esperado de ejecución del ordenamiento rápido aleatorizado es $O(n \log n)$. ■

En realidad, al aplicar poderosos resultados de la teoría de las probabilidades se puede demostrar que el tiempo de ejecución del ordenamiento rápido aleatorizado es $O(n \log n)$ con gran probabilidad. Se deja esto para el ejercicio C-10.7, para los lectores con grandes inclinaciones matemáticas.

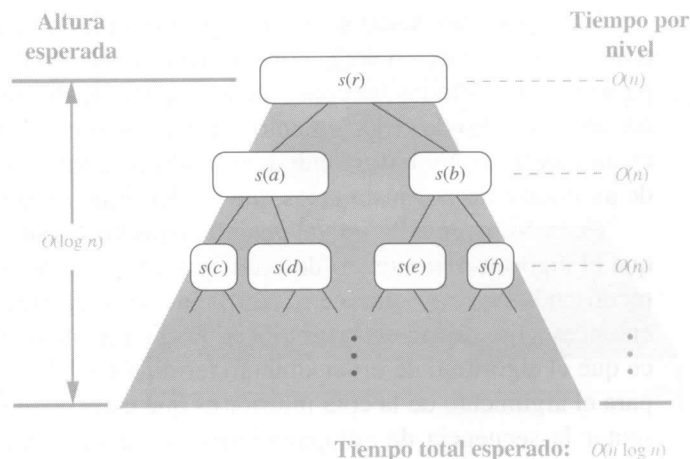


Figura 10.14: Análisis visual del tiempo del árbol T , del algoritmo *quick-sort*.

10.4 Cota inferior de ordenamiento basada en comparación

Recapitulando las descripciones de ordenamientos presentadas hasta ahora, se han descrito varios métodos, ya sea con un tiempo esperado de ejecución $O(n \log n)$ en el peor de los casos o con una secuencia de entrada de tamaño n . Esos métodos son el ordenamiento *merge* y el ordenamiento rápido, descritos en este capítulo, y también el ordenamiento *heap*, de la sección 7.3.4. Una duda natural es si sería posible ordenar con más rapidez que en el tiempo $O(n \log n)$.

En esta sección mostramos que si el primitivo informático, o computacional, usado por un algoritmo de ordenamiento es la comparación de dos elementos, ese tiempo es el mejor que se puede obtener; el ordenamiento basado en comparación tiene una cota inferior, en el peor de los casos, de $\Omega(n \log n)$ para su tiempo de ejecución. (Recuérdese la notación $\Omega(\cdot)$ de la sección 3.6.2.) Para enfocar el costo principal del ordenamiento basado en comparación, sólo se contarán las comparaciones que hace un algoritmo de ordenamiento. Como se trata de obtener una cota inferior, eso bastará.

Supóngase que se tiene una secuencia $S = (x_0, x_1, \dots, x_{n-1})$ que se desea ordenar, y que todos los elementos de S son distintos (lo cual no es una restricción porque se está deduciendo una cota inferior). Cada vez que un algoritmo de ordenamiento compara dos elementos x_i y x_j , esto es, que pregunta “¿es $x_i < x_j$?”, hay dos resultados posibles: “sí” o “no”. Con base en el resultado de esta comparación, el algoritmo de ordenamiento puede hacer algunos cálculos internos (que aquí no se cuentan) y al final hará otra comparación entre otros dos elementos de S ; habrá de nuevo dos resultados posibles. En consecuencia, se puede representar un algoritmo de ordenamiento basado en comparación con un árbol de decisión T (recuérdese el ejemplo 6.4). Esto es, cada nodo interno v en T corresponde a una comparación, y las aristas del nodo v a sus hijos corresponden a los cálculos que resultan de una respuesta “sí” o “no” (véase la figura 10.15).

Es importante observar que el algoritmo hipotético de ordenamiento en cuestión probablemente no tenga conocimiento explícito del árbol T . Tan sólo se usa T para representar todas las consecuencias posibles de comparaciones que podría hacer un algoritmo de ordenamiento, comenzando con la primera comparación (asociada con la raíz) y terminando con la última comparación (asociada con el padre de un nodo externo) justo antes que el algoritmo termine su ejecución.

Cada ordenamiento inicial posible, o **permutación**, de los elementos en S hará que el algoritmo hipotético de ordenamiento ejecute una serie de comparaciones, recorriendo una trayectoria en T desde la raíz hasta algún nodo externo. Asociemos entonces, con cada nodo externo v en T , el conjunto de permutaciones de S que hace que el algoritmo de ordenamiento termine en v . La observación más importante para el argumento de la cota inferior es que cada nodo externo v en T puede representar la secuencia de comparaciones de cuando mucho una permutación de S . La justificación de esta afirmación es sencilla: si dos permutaciones distintas P_1 y P_2 de S se asocian con el mismo nodo externo, entonces al menos hay dos objetos,

x_i y x_j , tales que x_i está antes que x_j en P_1 , pero x_i está después de x_j en P_2 . Al mismo tiempo, la salida asociada con v debe ser un reordenamiento específico de S , en el que x_i o x_j aparezcan primero que el otro. Pero si P_1 y P_2 causan que el algoritmo de ordenamiento produzca los elementos de S en este orden, eso implica que hay una forma de engañar al algoritmo para que produzca x_i y x_j en el orden incorrecto. Como un algoritmo de ordenamiento correcto no puede permitir eso, cada nodo externo de T se debe asociar exactamente con una permutación de S . Se usa esta propiedad del árbol de decisión asociado con un algoritmo de ordenamiento para demostrar el siguiente resultado:

Proposición 10.7: *El tiempo de ejecución de cualquier algoritmo basado en comparación, para ordenar una secuencia de n elementos, es $\Omega(n \log n)$ en el peor de los casos.*

Justificación: El tiempo de ejecución de un algoritmo de ordenamiento basado en comparación debe ser mayor o igual a la altura del árbol de decisión T asociado con él, tal como se describió antes (véase la figura 10.15). De acuerdo con este argumento, cada nodo externo en T debe asociarse con una permutación de S . Además, cada permutación de S debe resultar en un nodo externo distinto de T . La cantidad de permutaciones de n objetos es $n! = n(n-1)(n-2) \cdots 2 \cdot 1$. Así, T debe tener al menos $n!$ nodos externos. De acuerdo con la proposición 6.10, la altura mínima de T es $\log(n!)$. Esto justifica de inmediato la proposición, porque al menos hay $n/2$ términos que son mayores o iguales a $n/2$ en el producto $n!$; por consiguiente

$$\log(n!) \geq \log \left(\frac{n}{2} \right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2}$$

que es $\Omega(n \log n)$. ■

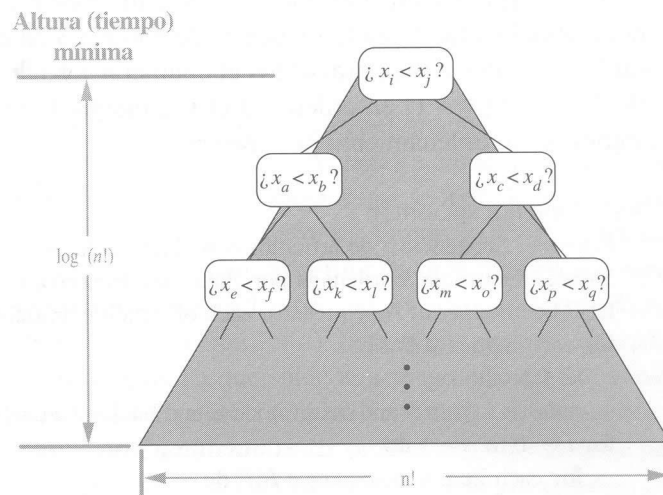


Figura 10.15: Visualización de la cota inferior para ordenamientos basados en comparación.

10.5 Ordenamiento por cubetas y ordenamiento radix

En la sección anterior se mostró que en el peor de los casos es necesario un tiempo $\Omega(n \log n)$ para ordenar una secuencia de n elementos con un algoritmo basado en comparación. Por consiguiente, es natural preguntar si hay otras clases de algoritmos de ordenamiento que se puedan diseñar para que trabajen asintóticamente más rápido que $O(n \log n)$. Es interesante saber que sí existen esos algoritmos, pero requieren hipótesis especiales acerca de la secuencia de entrada por ordenar. Aun así, esos casos se presentan con frecuencia en la práctica, por lo que vale la pena describirlos. En esta sección se describe el problema de ordenar una secuencia de artículos, siendo cada uno un par clave-elemento.

10.5.1 Ordenamiento por cubetas

Consideremos una secuencia S de n artículos cuyas claves son enteros entre los límites $[0, N - 1]$ para un entero $N \geq 2$, y supongamos que se debe ordenar S de acuerdo con las claves de los artículos. En este caso es posible ordenar S en el tiempo $O(n + N)$. Parece sorprendente, pero esto implica, por ejemplo, que si N es $O(n)$, entonces se puede ordenar S en el tiempo $O(n)$. Naturalmente lo determinante es que, debido a la hipótesis restrictiva sobre el formato de los elementos, se puede evitar el uso de comparaciones.

El concepto principal es usar un algoritmo llamado *bucket-sort*, que no se basa en comparaciones, sino en el uso de claves como índices en un arreglo de cubetas B cuyos elementos son de 0 a $N - 1$. Un artículo con clave k se coloca en la “cubeta” $B[k]$, que en sí es una secuencia (de elementos con clave k). Después de insertar cada artículo en la secuencia de entrada S en esa cubeta, se pueden regresar los artículos a S ya ordenados al enumerar los contenidos de las cubetas $B[0]$, $B[1]$, \dots , $B[N - 1]$ en orden. En el fragmento de programa 10.8 se describe el algoritmo de ordenamiento por cubetas.

Algoritmo bucketSort(S):

Entrada: Secuencia S de artículos con claves enteras en el intervalo $[0, N - 1]$

Salida: Secuencia S ordenada por claves no decrecientes

let B be an array of N sequences, each of which is initially empty

for each item x in S **do**

 let k be the key of x

 remove x from S and insert it at the end of bucket (sequence) $B[k]$

for $i \leftarrow 0$ to $N - 1$ **do**

for each item x in sequence $B[i]$ **do**

 remove x from $B[i]$ and insert it at the end of S

Fragmento de programa 10.8: Bucket-sort.

Es fácil apreciar que el ordenamiento por cubetas se ejecuta en el tiempo $O(n + N)$ y usa el espacio $O(n + N)$. Por consiguiente, el ordenamiento por cubetas es eficiente cuando el intervalo N de valores para las claves es pequeño en comparación con el tamaño n de la secuencia; por ejemplo $N = O(n)$ o $N = O(n \log n)$, pero esta eficiencia se deteriora a medida que N aumenta en comparación con n .

Una propiedad importante del algoritmo de ordenamiento por cubetas es que funciona en forma correcta, aunque haya muchos elementos distintos con la misma clave. En realidad, se ha descrito en una forma en que se anticipan esos casos.

Ordenamiento estable

Cuando se ordenan artículos de clave y elemento, un asunto importante es la forma en que se manejan claves iguales. Sea $S = ((k_0, e_0), \dots, (k_{n-1}, e_{n-1}))$ una secuencia de artículos. Se dice que un algoritmo de ordenamiento es **estable** si, para dos items cualesquiera, (k_i, e_i) y (k_j, e_j) de S , tales que $k_i = k_j$ y que (k_i, e_i) antecede a (k_j, e_j) en S antes del ordenamiento (esto es, que $i < j$), el item (k_i, e_i) también antecede al item (k_j, e_j) después del ordenamiento. La estabilidad es importante para un algoritmo de ordenamiento, porque en las aplicaciones se puede tratar de preservar el ordenamiento inicial de los elementos con la misma clave.

La descripción informal del ordenamiento por cubetas, en el fragmento de programa 10.8, no garantiza la estabilidad pero eso no es inherente al método mismo de ordenamiento por cubetas, porque se puede modificar con facilidad la descripción para hacer estable a este ordenamiento y además seguir conservando su tiempo de ejecución $O(n + N)$. En realidad se puede obtener un algoritmo estable de ordenamiento por cubetas, quitando siempre el **primer** elemento de la secuencia S y de las secuencias $B[i]$ durante la ejecución del algoritmo.

10.5.2 Ordenamiento radix

Una de las razones de la gran importancia del ordenamiento estable es que permite aplicar la técnica de ordenamiento por cubetas a contextos más generales que el de ordenar artículos. Por ejemplo, supóngase que se trata de ordenar artículos que son pares (k, l) donde k y l son enteros entre los límites $[0, N - 1]$, para un entero $N \geq 2$. En un contexto como éste, es natural definir un ordenamiento de esos artículos mediante una convención **lexicográfica** (de diccionario), en la que $(k_1, l_1) < (k_2, l_2)$ si $k_1 < k_2$, o si $k_1 = k_2$ y $l_1 < l_2$ (sección 7.1.4). Ésta es una versión apareada de la función de comparación lexicográfica que suele aplicarse a cadenas de caracteres de igual longitud, y que se generaliza con facilidad a múltiplos de d números para $d > 2$.

El algoritmo de **ordenamiento radix** ordena una secuencia de pares, por ejemplo S , aplicándole dos veces un ordenamiento estable por cubetas; primero usando un componente del par como clave de ordenamiento y después el segundo componente. Pero, ¿cuál orden es el correcto? ¿Se debe ordenar primero con las k (el primer componente) y después con las l (el segundo componente), o al revés?

Antes de contestar, examínese el siguiente ejemplo.

Ejemplo 10.8: *Se tiene la siguiente secuencia S :*

$$S = ((3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2))$$

Si se ordena S en forma estable por el primer componente, se obtiene la secuencia

$$S_1 = ((1,5),(1,2),(1,7),(2,5),(2,3),(2,2),(3,3),(3,2))$$

Si a continuación se ordena en forma estable esta secuencia S_1 por el segundo componente, se obtiene la secuencia

$$S_{1,2} = ((1,2),(2,2),(3,2),(2,3),(3,3),(1,5),(2,5),(1,7))$$

que no es exactamente una secuencia ordenada. Por otra parte, si primero se ordena a S en forma estable de acuerdo con el segundo componente, se obtiene la secuencia

$$S_2 = ((1,2),(3,2),(2,2),(3,3),(2,3),(1,5),(2,5),(1,7))$$

Si a continuación se ordena la secuencia S_2 en forma estable con el primer componente, se obtiene la secuencia

$$S_{2,1} = ((1,2),(1,5),(1,7),(2,2),(2,3),(2,5),(3,2),(3,3))$$

que es en realidad la secuencia S ordenada lexicográficamente.

Así, de acuerdo con este ejemplo, parece que primero se debe ordenar por el segundo componente y después ordenar de nuevo de acuerdo con el primer componente. Esta intuición es exactamente la correcta. Si primero se ordena en forma estable por el segundo componente y después de nuevo por el primero, se garantiza que si dos elementos son iguales en el segundo ordenamiento (por el primer componente), se preserva su orden relativo en la secuencia inicial (que se ordena por el segundo componente). Entonces se garantiza que siempre se ordena lexicográficamente la secuencia resultante. Se deja para el sencillo ejercicio R-10.14 la determinación de cómo se puede extender esta técnica a tercias y a otros d -tuplos de números. Se puede resumir como sigue esta sección:

Proposición 10.9: *Sea S una secuencia de n artículos clave-elemento, cada uno de los cuales tiene una clave (k_1, k_2, \dots, k_d) , siendo k_i un entero entre los límites $[0, N-1]$ para un entero $N \geq 2$. Se puede ordenar S lexicográficamente en el tiempo $O(d(n+N))$ usando ordenamiento radix.*

Con toda su importancia, el ordenamiento no es el único problema interesante que maneja una relación de orden total en un conjunto de elementos. Hay otras aplicaciones, por ejemplo, que no requieren una lista ordenada de todo un conjunto, pero que requieren cierta cantidad de información de ordenamiento del conjunto. Antes de estudiar ese problema (que se llama “selección”), regresemos brevemente para comparar todos los algoritmos de ordenamiento que se han estudiado hasta aquí.

10.6 Comparación de algoritmos de ordenamiento

En este punto es útil tomar un respiro y examinar todos los algoritmos que se han estudiado en este libro para ordenar una secuencia de n elementos. Se han estudiado varios métodos, por ejemplo el ordenamiento de burbuja, por inserción y por selección, cuyo comportamiento en el tiempo es $O(n^2)$, en los casos promedio y peor. También se estudiaron varios métodos con tiempo requerido $O(n \log n)$, incluyendo al ordenamiento *heap*, ordenamiento *merge* y ordenamiento rápido. Por último, se estudió una clase especial de algoritmos de ordenamiento, el ordenamiento por cubetas y el ordenamiento radix que se ejecutan en tiempo lineal para ciertas clases de claves. Con certeza, los algoritmos de ordenamiento de burbuja y ordenamiento por selección son malas opciones en cualquier aplicación porque se ejecutan en el tiempo $\Theta(n^2)$, aun en el mejor de los casos. Pero, entre los algoritmos restantes de ordenamiento, ¿cuál es el mejor?

Al igual que en muchas cosas de la vida, no hay un algoritmo de ordenamiento que sea claramente “mejor”. El algoritmo más adecuado para determinada aplicación depende de varias propiedades de esa aplicación. Se pueden establecer algunos lineamientos y observaciones basados en las propiedades conocidas de los “buenos” algoritmos de ordenamiento.

- Si se implementa bien, el tiempo de ejecución del **ordenamiento por inserción** es $O(n + k)$, siendo k la cantidad de inversiones (esto es, la cantidad de pares de elementos fuera de orden). Así, el ordenamiento por inserción es un algoritmo excelente para secuencias pequeñas, por ejemplo, de menos de 50 elementos, porque es fácil de programar, y las secuencias pequeñas tienen necesariamente pocas inversiones. También, el ordenamiento por inserción es muy efectivo para ordenar secuencias que están ya “casi” ordenadas. Por “casi” se entiende que la cantidad de inversiones es pequeña. Pero la eficiencia $O(n^2)$ en tiempo del ordenamiento por inserción hace que sea mala opción fuera de estos contextos especiales.
- Por otra parte, el **ordenamiento merge** se ejecuta en el tiempo $O(n \log n)$ en el peor de los casos, y es óptimo para métodos de ordenamiento basados en comparación. Sin embargo, con estudios experimentales se ha demostrado que como es difícil hacer que el ordenamiento *merge* se ejecute en el lugar, los *overheads* necesarios para implementarlo lo hacen menos atractivo que las implementaciones, en el lugar, del ordenamiento *heap* y el ordenamiento rápido para secuencias que puedan caber totalmente en el área de memoria principal de una computadora. Aun así, el ordenamiento *merge* es un algoritmo excelente para los casos en los que no puede caber la entrada en la memoria principal, sino que se tiene que guardar en bloques de un dispositivo de memoria externa, por ejemplo en un disco. En estos contextos, la forma en la que el ordenamiento *merge* procesa las corridas de datos formando largas corrientes *merge* hace el uso óptimo de todos los datos que pasan a la memoria principal de un bloque de disco. Así, para ordenamientos con memoria externa, el algoritmo de ordenamiento *merge* tiende a minimizar la cantidad total de lecturas y escrituras de disco necesarias, lo cual lo hace superior en esos contextos.

- Se ha demostrado con estudios experimentales que si una secuencia de entrada puede caber totalmente en la memoria principal, las versiones en el lugar del ordenamiento rápido y el ordenamiento *heap* se ejecutan más rápido que el ordenamiento *merge*. De hecho, el ordenamiento rápido tiende en promedio a superar al ordenamiento *heap* en esas pruebas. Por lo anterior, el **ordenamiento rápido** es una opción excelente como algoritmo de ordenamiento de propósito general en memoria. En realidad se incluye en la utilidad *qsort* de ordenamiento que se proporciona en las bibliotecas del lenguaje C. Sin embargo, su eficiencia de tiempo $O(n^2)$ en el peor de los casos hace que el ordenamiento rápido sea mala opción, para aplicaciones en tiempo real en las que se deben tener garantías, para el tiempo necesario para completar una operación de ordenamiento.
- En escenarios de tiempo real, donde se tiene una cantidad fija de tiempo para ejecutar una operación de ordenamiento, y los datos de entrada pueden caber en la memoria principal, es probable que la mejor opción de algoritmo sea el **ordenamiento heap**. Se ejecuta en el tiempo $O(n \log n)$ en el peor de los casos, y con facilidad se puede hacer que se ejecute en el lugar.
- Por último, si en la aplicación interviene el ordenamiento por claves, o múltiples d de claves enteras, las opciones excelentes son el **ordenamiento por cubetas** o el **ordenamiento radix**, porque se ejecutan en el tiempo $O(d(n + N))$, siendo $[0, N - 1]$ el intervalo de claves enteras (y $d = 1$ para el ordenamiento por cubetas). Así, si $d(n + N)$ está “por debajo” de $n \log n$ (formalmente, $d(n + N)$ es $o(n \log n)$, con la notación o minúscula de la sección 3.6.2), este método de ordenamiento debe ser más rápido todavía que el ordenamiento *quick-sort* o el ordenamiento *heap*.

Por lo anterior, el estudio de todos estos algoritmos de ordenamiento nos ha proporcionado un conjunto adaptable de métodos para la “caja de herramientas” de ingeniería de algoritmos.

10.7 Selección

Hay varias aplicaciones en las que interesa identificar un solo elemento en términos de su rango relativo en el ordenamiento de todo un conjunto. Entre los ejemplos está identificar los elementos mínimo y máximo, aunque también se puede tratar de identificar al elemento **mediano**, que es el elemento tal que la mitad de los demás elementos son menores y la otra mitad son mayores. En general, las consultas que buscan un elemento con determinado rango se llaman **estadísticos de orden**.

En esta sección se describe el problema general de los estadísticos de orden, que es seleccionar el k -ésimo elemento más pequeño de una colección no ordenada de n elementos comparables. A esto se le llama problema de **selección**. Naturalmente, este problema se puede resolver ordenando la colección para después indizarla en la secuencia ordenada en el rango k . Aplicando los mejores algoritmos de ordenamiento basados en comparación, ese método necesitaría el tiempo $O(n \log n)$ que naturalmente es demasiado para los casos en los que $k = 1$ o $k = n$ (o hasta $k = 2$, $k = 3$, $k = n - 1$ o $k = n - 5$), porque se puede resolver con facilidad el problema de

selección para esos valores de k en el tiempo $O(n)$. Así, una pregunta natural es si se puede lograr un tiempo de ejecución de $O(n)$ para todos los valores de k , incluyendo el caso interesante de encontrar la mediana, donde $k = \lfloor n/2 \rfloor$.

10.7.1 Poda y búsqueda

Es algo sorprendente, pero sí se puede resolver el problema de selección en el tiempo $O(n)$ para cualquier valor de k . Además, la técnica que se usa para alcanzar este resultado implica un patrón interesante de diseño de algoritmos. A este patrón de diseño se le llama *poda y búsqueda*, o *disminuye y vencerás*. Para aplicarlo se resuelve un problema dado, que se define en una colección de n objetos, podando y descartando una parte de los n objetos y resolviendo el problema menor en forma recursiva. Cuando al final el problema se ha reducido a uno definido en una colección de objetos de tamaño constante, se resuelve incluso con un método poco elegante. La construcción se completa regresando de todas las llamadas recursivas. En algunos casos se puede evitar el uso de la recursión y entonces sólo se itera el paso de reducción de podar y buscar, hasta que se pueda aplicar un método poco elegante, y detenerse. Por cierto, el método de búsqueda binaria que se describió en la sección 8.5.1 es un ejemplo del patrón de diseño de poda y búsqueda.

10.7.2 Selección rápida aleatorizada

Para aplicar el patrón de poda y búsqueda al problema de la selección, se puede diseñar un método sencillo y práctico llamado *selección rápida aleatorizada*, para encontrar el k -ésimo elemento más pequeño en una secuencia no ordenada de n elementos en la que esté definida una relación de orden total. La selección rápida aleatorizada se ejecuta en el tiempo *esperado* $O(n)$, calculado con todas las opciones aleatorias posibles que hace el algoritmo, y esta expectativa no depende en grado alguno de hipótesis de aleatoriedad acerca de la distribución de la entrada. Sin embargo, se observa que la selección rápida aleatorizada se ejecuta en el tiempo $O(n^2)$ en el *peor de los casos*, cuya demostración se deja para el ejercicio R-10.17. También hay un ejercicio, el C-10.23, para modificar una selección rápida aleatorizada y obtener un algoritmo de selección *determinista* que se ejecute en el tiempo $O(n)$ en el *peor de los casos*. La existencia de este algoritmo determinista tiene más que nada un interés teórico, porque en este caso el factor constante oculto por la notación de O mayúscula es relativamente grande.

Supóngase que se tiene una secuencia no ordenada S , de n elementos comparables, junto con un entero $k \in [1, n]$. En un nivel alto, el algoritmo de selección rápida para encontrar al k -ésimo elemento más pequeño de S tiene una estructura parecida a la del algoritmo de ordenamiento rápido aleatorizado que se describió en la sección 10.3.2. Se escoge un elemento x de S , al azar, y se usa como “pivote” para subdividir S en tres subsecuencias, L , E y G , que guardan los elementos de S menores que x , iguales a x y mayores que x , respectivamente. Éste es el paso de podar. A continuación, con base en el valor de k , se determina en cuál de esos

conjuntos se debe hacer la recursión. La selección rápida aleatorizada se describe en el fragmento de programa 10.9.

Algoritmo quickSelect(S, k):

Entrada: Secuencia S de n elementos comparables, y un entero $k \in [1, n]$

Salida: Ele k -ésimo elemento más pequeño de S

if $n = 1$ **then**

return the (first) element of S

 pick a random element x of S

 remove all the elements from S and put them into three sequence:

- L , que guarde los elementos en S menores que x
- E , que guarde los elementos en S iguales a x
- G , que guarde los elementos en S mayores que x .

if $k \leq |L|$ **then**

 quickSelect(L, k)

else if $k \leq |L| + |E|$ **then**

return x {cada elemento de E es igual a x }

else

 quickSelect($G, k - |L| - |E|$) {nótese el nuevo parámetro de selección}

Fragmento de programa 10.9: Algoritmo de quick-select aleatorizada.

10.7.3 Análisis de la selección rápida aleatorizada ★

Se mencionó líneas antes que el algoritmo de selección rápida aleatorizada se ejecuta en el tiempo esperado $O(n)$. Por fortuna, para justificar esta afirmación sólo se requieren los más sencillos argumentos de probabilidad. El hecho principal que se usa es la *linealidad de la expectativa*. Dice que si X y Y son variables aleatorias y si c es un número, entonces

$$E(X + Y) = E(X) + E(Y)$$

y

$$E(cX) = cE(x)$$

donde $E(Z)$ representa el valor esperado de la expresión Z .

Sea $t(n)$ el tiempo de ejecución de la selección rápida aleatorizada para una secuencia de tamaño n . Como el algoritmo de selección rápida aleatorizada depende del resultado de eventos aleatorios, su tiempo de ejecución, $t(n)$, es una variable aleatoria. Se quiere acotar a $E(t(n))$, el valor esperado de $t(n)$. Se dice que una invocación recursiva de la selección rápida aleatorizada es “buena” si parte a S de tal modo que el tamaño de L y G es, cuando mucho, $3n/4$. Es claro que una llamada recursiva tiene la probabilidad $1/2$ de ser buena. Sea $g(n)$ la cantidad de

invocaciones recursivas, incluyendo la actual, para que salga una buena invocación. Entonces

$$t(n) \leq bn \cdot g(n) + t(3n/4)$$

donde $b \geq 1$ es una constante (que tiene en cuenta el *overhead* de cada llamada). Es claro que se está enfocando el caso en el que n es mayor que 1, porque con facilidad se puede caracterizar, en una forma cerrada, que $t(1) = b$. Se aplica la propiedad de linealidad de expectativa al caso general, y se obtiene

$$E(t(n)) \leq E(bn \cdot g(n) + t(3n/4)) = bn \cdot E(g(n)) + E(t(3n/4))$$

Puesto que una llamada recursiva tiene la probabilidad $1/2$ de ser buena, y el que sea buena o no una llamada recursiva no depende de que su llamada padre sea buena, el valor esperado de $g(n)$ es igual a la cantidad esperada de veces que se debe arrojar al aire una moneda imparcial para que salga “cara”. Esto implica que $E(g(n)) = 2$. Así, si $T(n)$ es una notación taquigráfica que representa $E(t(n))$ (el tiempo esperado de ejecución del algoritmo de selección rápida aleatorizada), para el caso $n > 1$ se puede escribir

$$T(n) \leq T(3n/4) + 2bn$$

Al igual que en la relación de recurrencia del ordenamiento *merge*, se trata de convertir esta relación en una forma cerrada. Para ello se aplica de nuevo iterativamente esta ecuación, suponiendo que n es grande. Entonces, por ejemplo, después de dos aplicaciones iterativas, se obtiene

$$T(n) \leq T((3/4)^2 n) + 2b(3/4)n + 2bn$$

Se ve aquí que el caso general es

$$T(n) \leq 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i$$

En otras palabras, el tiempo esperado de ejecución de la selección rápida aleatorizada es $2bn$ multiplicado por la suma de una progresión geométrica cuya base es un número positivo menor que 1. Así, de acuerdo con la proposición 3.3, acerca de sumas geométricas, se llega al resultado que $T(n)$ es $O(n)$. En resumen:

Proposición 10.10: *El tiempo esperado de ejecución de la selección rápida aleatorizada en una secuencia de tamaño n es $O(n)$.*

Como se mencionó antes, hay una variación de la selección rápida que no usa aleatorización, y que se ejecuta en el tiempo $O(n)$ en el peor de los casos. En el ejercicio C-10.23 el lector interesado puede apreciar el diseño de este algoritmo y su análisis.

10.8 Ejercicios

Refuerzo

- R-10.1** Presente una justificación completa de la proposición 10.1.
- R-10.2** En el árbol de ordenamiento *merge*, que se ve en las figuras 10.2 a 10.5, algunas aristas se representan con flechas. ¿Qué significa una flecha hacia abajo? ¿Y una flecha hacia arriba?
- R-10.3** Presente una descripción, en pseudocódigo, del ordenamiento *merge* que tome un arreglo como entrada y salida, y no una secuencia general. (Nota: es probable que se necesite usar un arreglo auxiliar como “memoria intermedia”).
- R-10.4** Muestre que el tiempo de ejecución del algoritmo de ordenamiento *merge*, de una secuencia de n elementos, es $O(n \log n)$, aun cuando n no sea una potencia de 2.
- R-10.5** Suponga que hay dos secuencias ordenadas, A y B , cada una de n elementos, que no se deben considerar conjuntos (esto es, que A y B pueden contener elementos duplicados). Describa un método para calcular una secuencia que represente al conjunto $A \cup B$, sin duplicados, en un tiempo $O(n)$.
- R-10.6** Muestre que $(X - A) \cup (X - B) = X - (A \cap B)$, para tres conjuntos cualesquiera X , A y B .
- R-10.7** Realice, en pseudocódigo, descripciones para hacer los métodos *insert* y *remove* del TDA conjunto, suponiendo que se usan secuencias ordenadas para implementar conjuntos.
- R-10.8** Suponga que se modifica la versión determinista del algoritmo de ordenamiento rápido para que, en vez de seleccionar al último elemento de una secuencia de n elementos como pivote, se escoja al elemento de rango $\lfloor n/2 \rfloor$. ¿Cuál es el tiempo de ejecución de esta versión del ordenamiento rápido, en una secuencia que ya esté ordenada?
- R-10.9** De nuevo se tiene la modificación de la versión determinista del algoritmo de ordenamiento rápido para que, en lugar de seleccionar al último elemento de una secuencia de n elementos como pivote, se seleccione el elemento de rango $\lfloor n/2 \rfloor$. Describa la clase de secuencia que hace que esta versión del ordenamiento rápido se ejecute en el tiempo $\Theta(n^2)$.
- R-10.10** Muestre que el tiempo de ejecución, en el mejor de los casos, del ordenamiento rápido de una secuencia de tamaño n con elementos distintos es $O(n \log n)$.