# *Part II Sorting and Order Statistics*

# Introduction

This part presents several algorithms that solve the following ***sorting problem***:

**Input:** A sequence of $n$ numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

**Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The input sequence is usually an $n$-element array, although it may be represented in some other fashion, such as a linked list.

## The structure of the data

In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a ***record***. Each record contains a ***key***, which is the value to be sorted. The remainder of the record consists of ***satellite data***, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, it often pays to permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

In a sense, it is these implementation details that distinguish an algorithm from a full-blown program. A sorting algorithm describes the *method* to determine the sorted order, regardless of whether what's being sorted are individual numbers or large records containing many bytes of satellite data. Thus, when focusing on the problem of sorting, we typically assume that the input consists only of numbers. Translating an algorithm for sorting numbers into a program for sorting records is conceptually straightforward, although in a given engineering situation other subtleties may make the actual programming task a challenge.

### Why sorting?

Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms. There are several reasons:

- Sometimes an application inherently needs to sort information. For example, in order to prepare customer statements, banks need to sort checks by check number.
- Algorithms often use sorting as a key subroutine. For example, a program that renders graphical objects which are layered on top of each other might have to sort the objects according to an "above" relation so that it can draw these objects from bottom to top. We will see numerous algorithms in this text that use sorting as a subroutine.
- We can draw from among a wide variety of sorting algorithms, and they employ a rich set of techniques. In fact, many important techniques used throughout algorithm design appear in sorting algorithms that have been developed over the years. In this way, sorting is also a problem of historical interest.
- We can prove a nontrivial lower bound for sorting (as we'll do in Chapter 8). Since the best upper bounds match the lower bound asymptotically, we can conclude that certain of our sorting algorithms are asymptotically optimal. Moreover, we can use the lower bound for sorting to prove lower bounds for various other problems.
- Many engineering issues come to the fore when implementing sorting algorithms. The fastest sorting program for a particular situation may depend on many factors, such as prior knowledge about the keys and satellite data, the memory hierarchy (caches and virtual memory) of the host computer, and the software environment. Many of these issues are best dealt with at the algorithmic level, rather than by "tweaking" the code.

### Sorting algorithms

We introduced two algorithms that sort $n$ real numbers in Chapter 2. Insertion sort takes $\Theta(n^2)$ time in the worst case. Because its inner loops are tight, however, it is a fast sorting algorithm for small input sizes. Moreover, unlike merge sort, it sorts ***in place***, meaning that at most a constant number of elements of the input array are ever stored outside the array, which can be advantageous for space efficiency. Merge sort has a better asymptotic running time, $\Theta(n \lg n)$, but the MERGE procedure it uses does not operate in place. (We'll see a parallelized version of merge sort in Section 26.3.)

This part introduces two more algorithms that sort arbitrary real numbers. Heapsort, presented in Chapter 6, sorts $n$ numbers in place in $O(n \lg n)$ time. It uses an important data structure, called a heap, which can also implement a priority queue.

Quicksort, in Chapter 7, also sorts $n$ numbers in place, but its worst-case running time is $\Theta(n^2)$. Its expected running time is $\Theta(n \lg n)$, however, and it generally outperforms heapsort in practice. Like insertion sort, quicksort has tight code, and so the hidden constant factor in its running time is small. It is a popular algorithm for sorting large arrays.

Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts: they determine the sorted order of an input array by comparing elements. Chapter 8 begins by introducing the decision-tree model in order to study the performance limitations of comparison sorts. Using this model, we prove a lower bound of $\Omega(n \lg n)$ on the worst-case running time of any comparison sort on $n$ inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

Chapter 8 then goes on to show that we might be able to beat this lower bound of $\Omega(n \lg n)$ if an algorithm can gather information about the sorted order of the input by means other than comparing elements. The counting sort algorithm, for example, assumes that the input numbers belong to the set $\{0, 1, \ldots, k\}$. By using array indexing as a tool for determining relative order, counting sort can sort $n$ numbers in $\Theta(k + n)$ time. Thus, when $k = O(n)$, counting sort runs in time that is linear in the size of the input array. A related algorithm, radix sort, can be used to extend the range of counting sort. If there are $n$ integers to sort, each integer has $d$ digits, and each digit can take on up to $k$ possible values, then radix sort can sort the numbers in $\Theta(d(n + k))$ time. When $d$ is a constant and $k$ is $O(n)$, radix sort runs in linear time. A third algorithm, bucket sort, requires knowledge of the probabilistic distribution of numbers in the input array. It can sort $n$ real numbers uniformly distributed in the half-open interval $[0, 1)$ in average-case $O(n)$ time.

The table on the following page summarizes the running times of the sorting algorithms from Chapters 2 and 6–8. As usual, $n$ denotes the number of items to sort. For counting sort, the items to sort are integers in the set $\{0, 1, \ldots, k\}$. For radix sort, each item is a $d$-digit number, where each digit takes on $k$ possible values. For bucket sort, we assume that the keys are real numbers uniformly distributed in the half-open interval $[0, 1)$. The rightmost column gives the average-case or expected running time, indicating which one it gives when it differs from the worst-case running time. We omit the average-case running time of heapsort because we do not analyze it in this book.

| Algorithm | Worst-case running time | Average-case/expected running time |
|---|---|---|
| Insertion sort | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Merge sort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ |
| Heapsort | $O(n \lg n)$ | — |
| Quicksort | $\Theta(n^2)$ | $\Theta(n \lg n)$ (expected) |
| Counting sort | $\Theta(k + n)$ | $\Theta(k + n)$ |
| Radix sort | $\Theta(d(n + k))$ | $\Theta(d(n + k))$ |
| Bucket sort | $\Theta(n^2)$ | $\Theta(n)$ (average-case) |

**Order statistics**

The $i$th order statistic of a set of $n$ numbers is the $i$th smallest number in the set. You can, of course, select the $i$th order statistic by sorting the input and indexing the $i$th element of the output. With no assumptions about the input distribution, this method runs in $\Omega(n \lg n)$ time, as the lower bound proved in Chapter 8 shows.

Chapter 9 shows how to find the $i$th smallest element in $O(n)$ time, even when the elements are arbitrary real numbers. We present a randomized algorithm with tight pseudocode that runs in $\Theta(n^2)$ time in the worst case, but whose expected running time is $O(n)$. We also give a more complicated algorithm that runs in $O(n)$ worst-case time.

**Background**

Although most of this part does not rely on difficult mathematics, some sections do require mathematical sophistication. In particular, analyses of quicksort, bucket sort, and the order-statistic algorithm use probability, which is reviewed in Appendix C, and the material on probabilistic analysis and randomized algorithms in Chapter 5.

# 6 Heapsort

This chapter introduces another sorting algorithm: heapsort. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \lg n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a "heap," to manage information. Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue. The heap data structure will reappear in algorithms in later chapters.

The term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage," such as the programming languages Java and Python provide. Please don't be confused. The heap data structure is *not* garbage-collected storage. This book is consistent in using the term "heap" to refer to the data structure, not the storage class.

## 6.1 Heaps

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree (see Section B.5.3), as shown in Figure 6.1. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array $A[1:n]$ that represents a heap is an object with an attribute $A.\mathit{heap\text{-}size}$, which represents how many elements in the heap are stored within array $A$. That is, although $A[1:n]$ may contain numbers, only the elements in $A[1:A.\mathit{heap\text{-}size}]$, where $0 \le A.\mathit{heap\text{-}size} \le n$, are valid elements of the heap. If $A.\mathit{heap\text{-}size} = 0$, then the heap is empty. The root of the tree is $A[1]$, and given the index $i$ of a node,

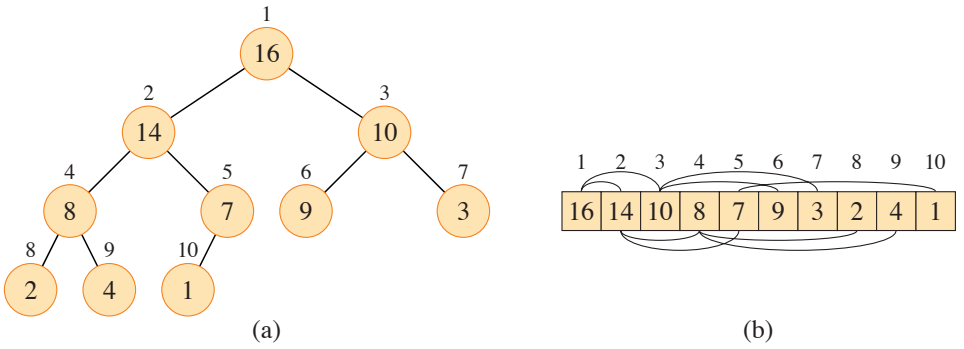


**Figure 6.1** A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships, with parents always to the left of their children. The tree has height 3, and the node at index 4 (with value 8) has height 1.

there's a simple way to compute the indices of its parent, left child, and right child with the one-line procedures PARENT, LEFT, and RIGHT.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

On most computers, the LEFT procedure can compute $2i$ in one instruction by simply shifting the binary representation of $i$ left by one bit position. Similarly, the RIGHT procedure can quickly compute $2i + 1$ by shifting the binary representation of $i$ left by one bit position and then adding 1. The PARENT procedure can compute $\lfloor i/2 \rfloor$ by shifting $i$ right one bit position. Good implementations of heapsort often implement these procedures as macros or inline procedures.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a ***heap property***, the specifics of which depend on the kind of heap. In a ***max-heap***, the ***max-heap property*** is that for every node $i$ other than the root,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself. A ***min-heap*** is organized in the opposite way: the ***min-heap property*** is that for every node $i$ other than the root,

$$A[\text{PARENT}(i)] \leq A[i] .$$

The smallest element in a min-heap is at the root.

The heapsort algorithm uses max-heaps. Min-heaps commonly implement priority queues, which we discuss in Section 6.5. We'll be precise in specifying whether we need a max-heap or a min-heap for any particular application, and when properties apply to either max-heaps or min-heaps, we just use the term "heap."

Viewing a heap as a tree, we define the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of $n$ elements is based on a complete binary tree, its height is $\Theta(\lg n)$ (see Exercise 6.1-2). As we'll see, the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(\lg n)$ time. The remainder of this chapter presents some basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The procedures MAX-HEAP-INSERT, MAX-HEAP-EXTRACT-MAX, MAX-HEAP-INCREASE-KEY, and MAX-HEAP-MAXIMUM allow the heap data structure to implement a priority queue. They run in $O(\lg n)$ time plus the time for mapping between objects being inserted into the priority queue and indices in the heap.

## Exercises

### *6.1-1*

What are the minimum and maximum numbers of elements in a heap of height $h$?

### *6.1-2*

Show that an $n$-element heap has height $\lfloor \lg n \rfloor$.

***6.1-3***
Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

***6.1-4***
Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

***6.1-5***
At which levels in a max-heap might the $k$th largest element reside, for $2 \leq k \leq \lfloor n/2 \rfloor$, assuming that all elements are distinct?

***6.1-6***
Is an array that is in sorted order a min-heap?

***6.1-7***
Is the array with values $\langle 33, 19, 20, 15, 13, 10, 2, 13, 16, 12 \rangle$ a max-heap?

***6.1-8***
Show that, with the array representation for storing an $n$-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

## 6.2 Maintaining the heap property

The procedure MAX-HEAPIFY on the facing page maintains the max-heap property. Its inputs are an array $A$ with the *heap-size* attribute and an index $i$ into the array. When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT$(i)$ and RIGHT$(i)$ are max-heaps, but that $A[i]$ might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index $i$ obeys the max-heap property.

Figure 6.2 illustrates the action of MAX-HEAPIFY. Each step determines the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$ and stores the index of the largest element in *largest*. If $A[i]$ is largest, then the subtree rooted at node $i$ is already a max-heap and nothing else needs to be done. Otherwise, one of the two children contains the largest element. Positions $i$ and *largest* swap their contents, which causes node $i$ and its children to satisfy the max-heap property. The node indexed by *largest*, however, just had its value decreased, and thus the subtree rooted at *largest* might violate the max-heap property. Consequently, MAX-HEAPIFY calls itself recursively on that subtree.

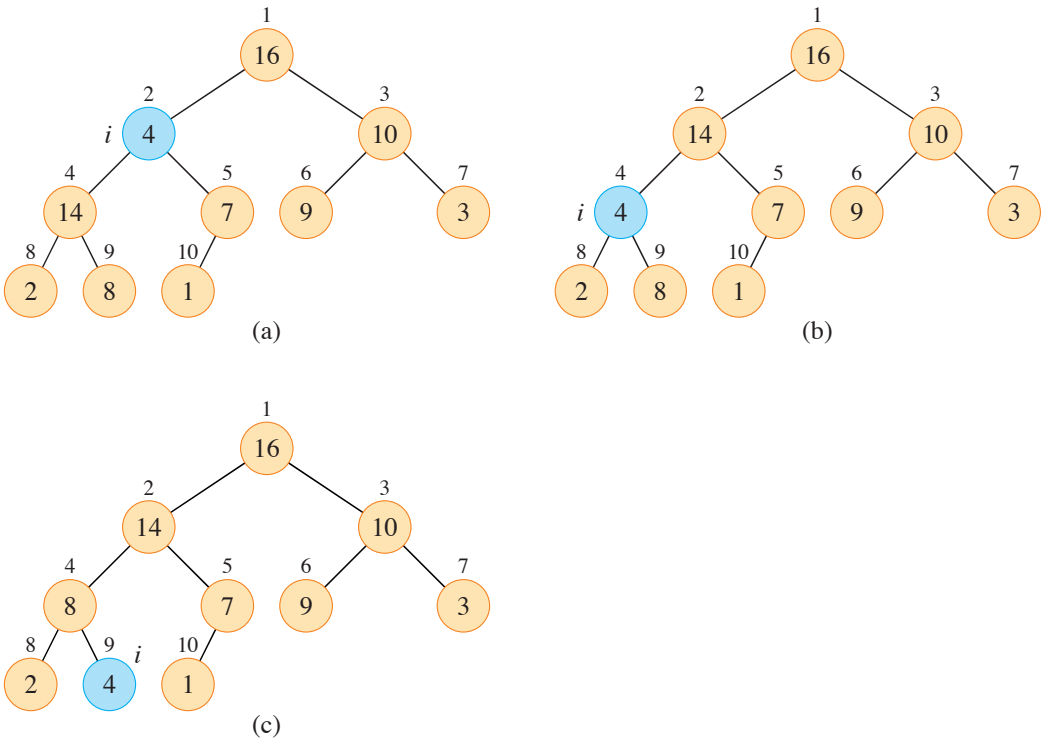


**Figure 6.2** The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. The node that potentially violates the max-heap property is shown in blue. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After $A[4]$ and $A[9]$ are swapped, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

To analyze MAX-HEAPIFY, let $T(n)$ be the worst-case running time that the procedure takes on a subtree of size at most $n$. For a tree rooted at a given node $i$, the running time is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node $i$ (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$ (see Exercise 6.2-2), and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1) . \tag{6.1}$$

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1 on page 102), is $T(n) = O(\lg n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height $h$ as $O(h)$.

## Exercises

### *6.2-1*

Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY$(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

### *6.2-2*

Show that each child of the root of an $n$-node heap is the root of a subtree containing at most $2n/3$ nodes. What is the smallest constant $\alpha$ such that each subtree has at most $\alpha n$ nodes? How does that affect the recurrence (6.1) and its solution?

### *6.2-3*

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY$(A, i)$, which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare with that of MAX-HEAPIFY?

### *6.2-4*

What is the effect of calling MAX-HEAPIFY$(A, i)$ when the element $A[i]$ is larger than its children?

### *6.2-5*

What is the effect of calling MAX-HEAPIFY$(A, i)$ for $i > A.\textit{heap-size}/2$?

### *6.2-6*

The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, for which some compilers might produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

***6.2-7***

Show that the worst-case running time of MAX-HEAPIFY on a heap of size $n$ is $\Omega(\lg n)$. (*Hint:* For a heap with $n$ nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

## 6.3 Building a heap

The procedure BUILD-MAX-HEAP converts an array $A[1:n]$ into a max-heap by calling MAX-HEAPIFY in a bottom-up manner. Exercise 6.1-8 says that the elements in the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are all leaves of the tree, and so each is a 1-element heap to begin with. BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one. Figure 6.3 shows an example of the action of BUILD-MAX-HEAP.

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–3, each node $i + 1$, $i + 2, \ldots, n$ is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that the invariant provides a useful property to show correctness when the loop terminates.

**Initialization:** Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

**Maintenance:** To see that each iteration maintains the loop invariant, observe that the children of node $i$ are numbered higher than $i$. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY$(A, i)$ to make node $i$ a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1$, $i + 2, \ldots, n$ are all roots of max-heaps. Decrementing $i$ in the **for** loop update reestablishes the loop invariant for the next iteration.

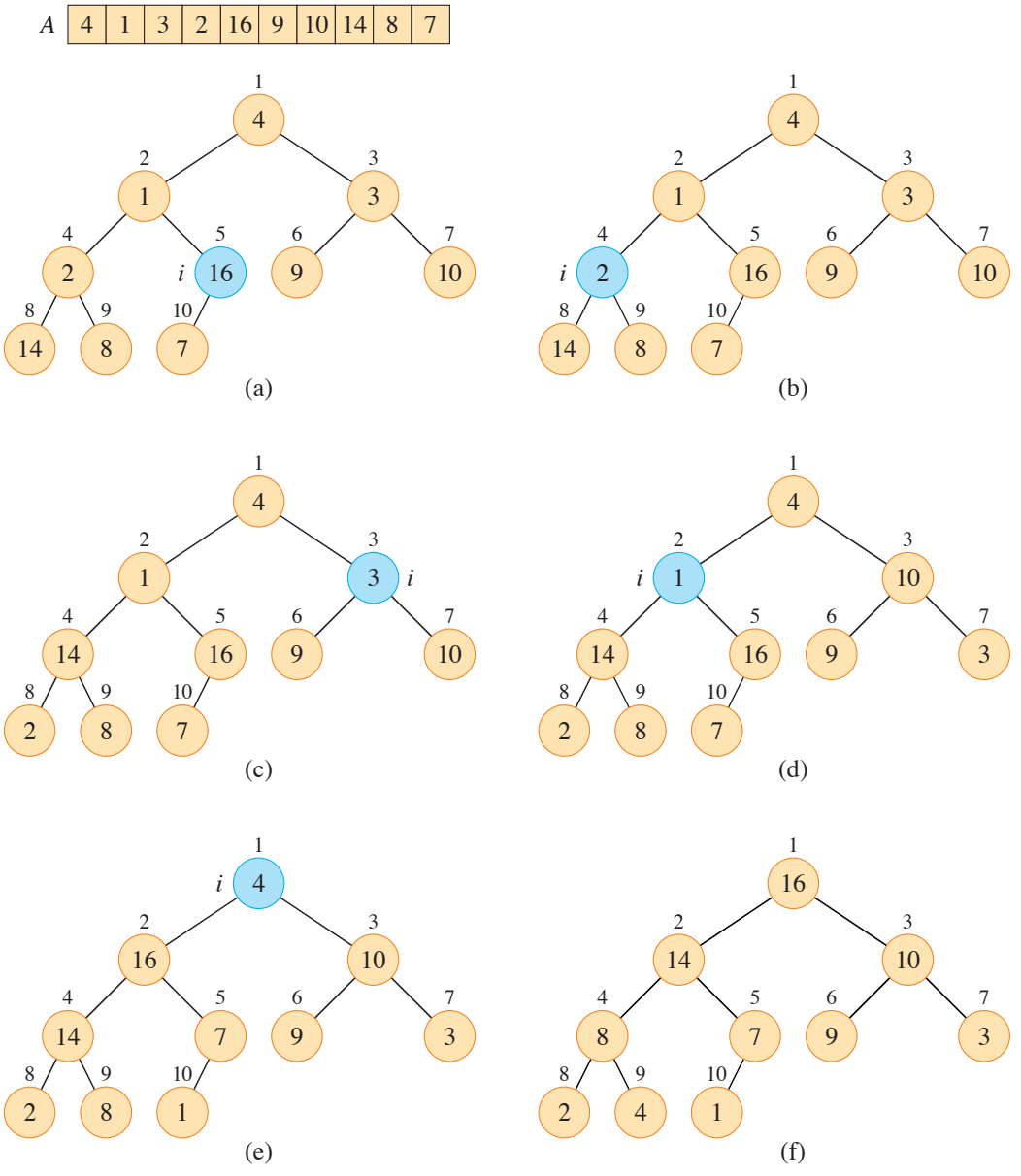


**Figure 6.3** The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. The node indexed by $i$ in each iteration is shown in blue. **(a)** A 10-element input array $A$ and the binary tree it represents. The loop index $i$ refers to node 5 before the call MAX-HEAPIFY$(A, i)$. **(b)** The data structure that results. The loop index $i$ for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

**Termination:** The loop makes exactly $\lfloor n/2 \rfloor$ iterations, and so it terminates. At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

We can compute a simple upper bound on the running time of BUILD-MAX-HEAP as follows. Each call to MAX-HEAPIFY costs $O(\lg n)$ time, and BUILD-MAX-HEAP makes $O(n)$ such calls. Thus, the running time is $O(n \lg n)$. This upper bound, though correct, is not as tight as it can be.

We can derive a tighter asymptotic bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and that the heights of most nodes are small. Our tighter analysis relies on the properties that an $n$-element heap has height $\lfloor \lg n \rfloor$ (see Exercise 6.1-2) and at most $\lceil n/2^{h+1} \rceil$ nodes of any height $h$ (see Exercise 6.3-4).

The time required by MAX-HEAPIFY when called on a node of height $h$ is $O(h)$. Letting $c$ be the constant implicit in the asymptotic notation, we can express the total cost of BUILD-MAX-HEAP as being bounded from above by $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil ch$. As Exercise 6.3-2 shows, we have $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$. Since $\lceil x \rceil \leq 2x$ for any $x \geq 1/2$, we have $\lceil n/2^{h+1} \rceil \leq n/2^h$. We thus obtain

$$
\begin{aligned}
&\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil ch \\
&\qquad \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \\
&\qquad = cn \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \\
&\qquad \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} \\
&\qquad \leq cn \cdot \frac{1/2}{(1-1/2)^2} \quad \text{(by equation (A.11) on page 1142 with } x = 1/2) \\
&\qquad = O(n) \,.
\end{aligned}
$$

Hence, we can build a max-heap from an unordered array in linear time.

To build a min-heap, use the procedure BUILD-MIN-HEAP, which is the same as BUILD-MAX-HEAP but with the call to MAX-HEAPIFY in line 3 replaced by a call to MIN-HEAPIFY (see Exercise 6.2-3). BUILD-MIN-HEAP produces a min-heap from an unordered linear array in linear time.

### Exercises

***6.3-1***
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

***6.3-2***
Show that $\lceil n/2^{h+1} \rceil \geq 1/2$ for $0 \leq h \leq \lfloor \lg n \rfloor$.

***6.3-3***
Why does the loop index $i$ in line 2 of BUILD-MAX-HEAP decrease from $\lfloor n/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor n/2 \rfloor$?

***6.3-4***
Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$ in any $n$-element heap.

## 6.4 The heapsort algorithm

The heapsort algorithm, given by the procedure HEAPSORT, starts by calling the BUILD-MAX-HEAP procedure to build a max-heap on the input array $A[1:n]$. Since the maximum element of the array is stored at the root $A[1]$, HEAPSORT can place it into its correct final position by exchanging it with $A[n]$. If the procedure then discards node $n$ from the heap—and it can do so by simply decrementing $A.heap\text{-}size$—the children of the root remain max-heaps, but the new root element might violate the max-heap property. To restore the max-heap property, the procedure just calls MAX-HEAPIFY$(A, 1)$, which leaves a max-heap in $A[1:n-1]$. The HEAPSORT procedure then repeats this process for the max-heap of size $n-1$ down to a heap of size 2. (See Exercise 6.4-2 for a precise loop invariant.)

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Figure 6.4 shows an example of the operation of HEAPSORT after line 1 has built the initial max-heap. The figure shows the max-heap before the first iteration of the **for** loop of lines 2–5 and after each iteration.

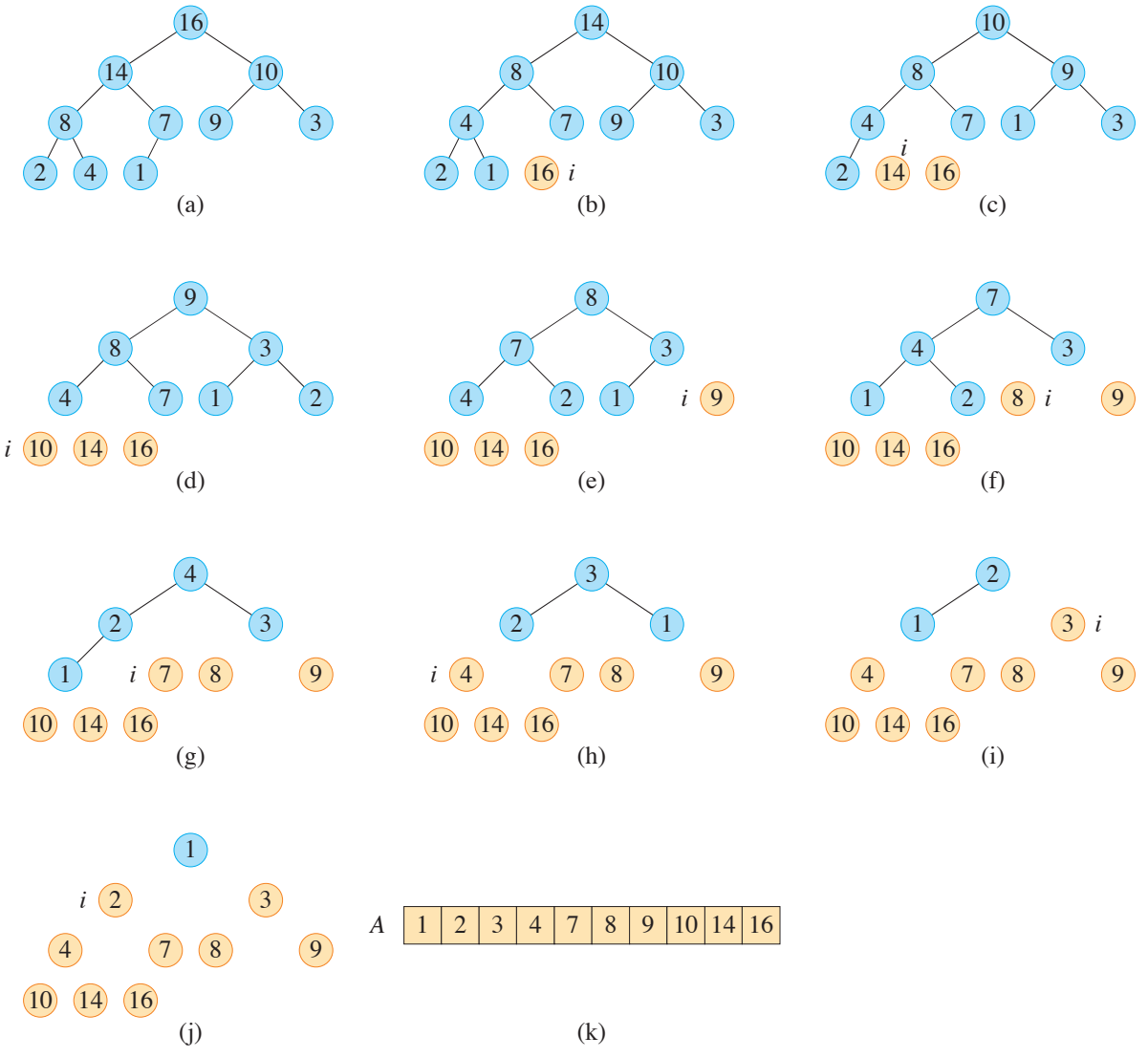


**Figure 6.4** The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of $i$ at that time. Only blue nodes remain in the heap. Tan nodes contain the largest values in the array, in sorted order. **(k)** The resulting sorted array $A$.

The HEAPSORT procedure takes $O(n \lg n)$ time, since the call to BUILD-MAX-HEAP takes $O(n)$ time and each of the $n - 1$ calls to MAX-HEAPIFY takes $O(\lg n)$ time.

### Exercises

***6.4-1***
Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

***6.4-2***
Argue the correctness of HEAPSORT using the following loop invariant:

> At the start of each iteration of the **for** loop of lines 2–5, the subarray $A[1 : i]$ is a max-heap containing the $i$ smallest elements of $A[1 : n]$, and the subarray $A[i + 1 : n]$ contains the $n - i$ largest elements of $A[1 : n]$, sorted.

***6.4-3***
What is the running time of HEAPSORT on an array $A$ of length $n$ that is already sorted in increasing order? How about if the array is already sorted in decreasing order?

***6.4-4***
Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

★ ***6.4-5***
Show that when all the elements of $A$ are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

## 6.5 Priority queues

In Chapter 8, we will see that any comparison-based sorting algorithm requires $\Omega(n \lg n)$ comparisons and hence $\Omega(n \lg n)$ time. Therefore, heapsort is asymptotically optimal among comparison-based sorting algorithms. Yet, a good implementation of quicksort, presented in Chapter 7, usually beats it in practice. Nevertheless, the heap data structure itself has many uses. In this section, we present one of the most popular applications of a heap: as an efficient priority queue. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We'll focus here on how to implement max-priority queues, which are in turn based on max-heaps. Exercise 6.5-3 asks you to write the procedures for min-priority queues.

A ***priority queue*** is a data structure for maintaining a set $S$ of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

INSERT$(S, x, k)$ inserts the element $x$ with key $k$ into the set $S$, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM$(S)$ returns the element of $S$ with the largest key.

EXTRACT-MAX$(S)$ removes and returns the element of $S$ with the largest key.

INCREASE-KEY$(S, x, k)$ increases the value of element $x$'s key to the new value $k$, which is assumed to be at least as large as $x$'s current key value.

Among their other applications, you can use max-priority queues to schedule jobs on a computer shared among multiple users. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling EXTRACT-MAX. The scheduler can add a new job to the queue at any time by calling INSERT.

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program calls EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, the simulator inserts them into the min-priority queue by calling INSERT. We'll see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 21 and 22.

When you use a heap to implement a priority queue within a given application, elements of the priority queue correspond to objects in the application. Each object contains a key. If the priority queue is implemented by a heap, you need to determine which application object corresponds to a given heap element, and vice versa. Because the heap elements are stored in an array, you need a way to map application objects to and from array indices.

One way to map between application objects and heap elements uses ***handles***, which are additional information stored in the objects and heap elements that give enough information to perform the mapping. Handles are often implemented to be opaque to the surrounding code, thereby maintaining an abstraction barrier between the application and the priority queue. For example, the handle within an application object might contain the corresponding index into the heap array. But since only the code for the priority queue accesses this index, the index is entirely hidden from the application code. Because heap elements change locations within

the array during heap operations, an actual implementation of the priority queue, upon relocating a heap element, must also update the array indices in the corresponding handles. Conversely, each element in the heap might contain a pointer to the corresponding application object, but the heap element knows this pointer as only an opaque handle and the application maps this handle to an application object. Typically, the worst-case overhead for maintaining handles is $O(1)$ per access.

As an alternative to incorporating handles in application objects, you can store within the priority queue a mapping from application objects to array indices in the heap. The advantage of doing so is that the mapping is contained entirely within the priority queue, so that the application objects need no further embellishment. The disadvantage lies in the additional cost of establishing and maintaining the mapping. One option for the mapping is a hash table (see Chapter 11).[1] The added expected time for a hash table to map an object to an array index is just $O(1)$, though the worst-case time can be as bad as $\Theta(n)$.

Let's see how to implement the operations of a max-priority queue using a max-heap. In the previous sections, we treated the array elements as the keys to be sorted, implicitly assuming that any satellite data moved with the corresponding keys. When a heap implements a priority queue, we instead treat each array element as a pointer to an object in the priority queue, so that the object is analogous to the satellite data when sorting. We further assume that each such object has an attribute *key*, which determines where in the heap the object belongs. For a heap implemented by an array $A$, we refer to $A[i].key$.

The procedure MAX-HEAP-MAXIMUM on the facing page implements the MAXIMUM operation in $\Theta(1)$ time, and MAX-HEAP-EXTRACT-MAX implements the operation EXTRACT-MAX. MAX-HEAP-EXTRACT-MAX is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure. We implicitly assume that MAX-HEAPIFY compares priority-queue objects based on their *key* attributes. We also assume that when MAX-HEAPIFY exchanges elements in the array, it is exchanging pointers and also that it updates the mapping between objects and array indices. The running time of MAX-HEAP-EXTRACT-MAX is $O(\lg n)$, since it performs only a constant amount of work on top of the $O(\lg n)$ time for MAX-HEAPIFY, plus whatever overhead is incurred within MAX-HEAPIFY for mapping priority-queue objects to array indices.

The procedure MAX-HEAP-INCREASE-KEY on page 176 implements the INCREASE-KEY operation. It first verifies that the new key $k$ will not cause the key in the object $x$ to decrease, and if there is no problem, it gives $x$ the new key value. The procedure then finds the index $i$ in the array corresponding to object $x$,

---

[1] In Python, dictionaries are implemented with hash tables.

```
MAX-HEAP-MAXIMUM(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  return A[1]

MAX-HEAP-EXTRACT-MAX(A)
1  max = MAX-HEAP-MAXIMUM(A)
2  A[1] = A[A.heap-size]
3  A.heap-size = A.heap-size − 1
4  MAX-HEAPIFY(A, 1)
5  return max
```

so that $A[i]$ is $x$. Because increasing the key of $A[i]$ might violate the max-heap property, the procedure then, in a manner reminiscent of the insertion loop (lines 5–7) of INSERTION-SORT on page 19, traverses a simple path from this node toward the root to find a proper place for the newly increased key. As MAX-HEAP-INCREASE-KEY traverses this path, it repeatedly compares an element's key to that of its parent, exchanging pointers and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds. (See Exercise 6.5-7 for a precise loop invariant.) Like MAX-HEAPIFY when used in a priority queue, MAX-HEAP-INCREASE-KEY updates the information that maps objects to array indices when array elements are exchanged. Figure 6.5 shows an example of a MAX-HEAP-INCREASE-KEY operation. In addition to the overhead for mapping priority queue objects to array indices, the running time of MAX-HEAP-INCREASE-KEY on an $n$-element heap is $O(\lg n)$, since the path traced from the node updated in line 3 to the root has length $O(\lg n)$.

The procedure MAX-HEAP-INSERT on the next page implements the INSERT operation. It takes as inputs the array $A$ implementing the max-heap, the new object $x$ to be inserted into the max-heap, and the size $n$ of array $A$. The procedure first verifies that the array has room for the new element. It then expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls MAX-HEAP-INCREASE-KEY to set the key of this new element to its correct value and maintain the max-heap property. The running time of MAX-HEAP-INSERT on an $n$-element heap is $O(\lg n)$ plus the overhead for mapping priority queue objects to indices.

In summary, a heap can support any priority-queue operation on a set of size $n$ in $O(\lg n)$ time, plus the overhead for mapping priority queue objects to array indices.

```
MAX-HEAP-INCREASE-KEY(A, x, k)
1  if k < x.key
2       error "new key is smaller than current key"
3  x.key = k
4  find the index i in array A where object x occurs
5  while i > 1 and A[PARENT(i)].key < A[i].key
6       exchange A[i] with A[PARENT(i)], updating the information that maps
            priority queue objects to array indices
7       i = PARENT(i)

MAX-HEAP-INSERT(A, x, n)
1  if A.heap-size == n
2       error "heap overflow"
3  A.heap-size = A.heap-size + 1
4  k = x.key
5  x.key = −∞
6  A[A.heap-size] = x
7  map x to index heap-size in the array
8  MAX-HEAP-INCREASE-KEY(A, x, k)
```

## Exercises

***6.5-1***

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-EXTRACT-MAX on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

***6.5-2***

Suppose that the objects in a max-priority queue are just keys. Illustrate the operation of MAX-HEAP-INSERT$(A, 10)$ on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

***6.5-3***

Write pseudocode to implement a min-priority queue with a min-heap by writing the procedures MIN-HEAP-MINIMUM, MIN-HEAP-EXTRACT-MIN, MIN-HEAP-DECREASE-KEY, and MIN-HEAP-INSERT.

***6.5-4***

Write pseudocode for the procedure MAX-HEAP-DECREASE-KEY$(A, x, k)$ in a max-heap. What is the running time of your procedure?

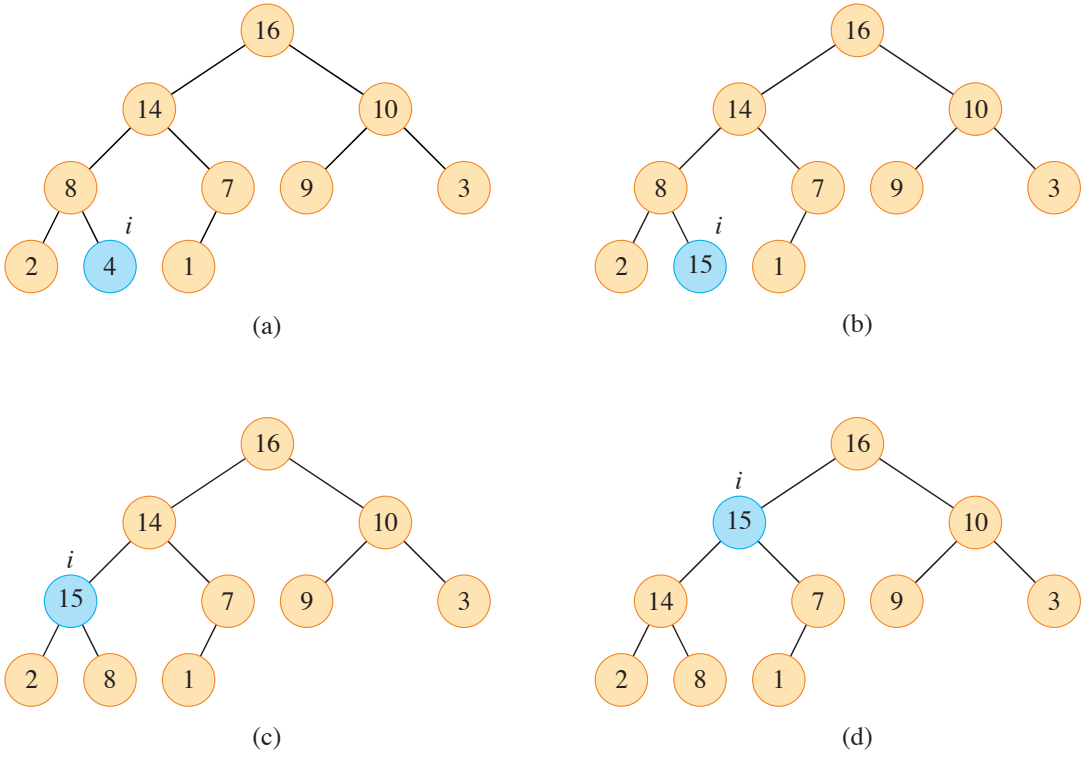


**Figure 6.5** The operation of MAX-HEAP-INCREASE-KEY. Only the key of each element in the priority queue is shown. The node indexed by $i$ in each iteration is shown in blue. **(a)** The max-heap of Figure 6.4(a) with $i$ indexing the node whose key is about to be increased. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 5–7, the node and its parent have exchanged keys, and the index $i$ moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

### *6.5-5*

Why does MAX-HEAP-INSERT bother setting the key of the inserted object to $-\infty$ in line 5 given that line 8 will set the object's key to the desired value?

### *6.5-6*

Professor Uriah suggests replacing the **while** loop of lines 5–7 in MAX-HEAP-INCREASE-KEY by a call to MAX-HEAPIFY. Explain the flaw in the professor's idea.

### *6.5-7*

Argue the correctness of MAX-HEAP-INCREASE-KEY using the following loop invariant:

At the start of each iteration of the **while** loop of lines 5–7:

a. If both nodes PARENT($i$) and LEFT($i$) exist, then $A[\text{PARENT}(i)].key \geq A[\text{LEFT}(i)].key$.
b. If both nodes PARENT($i$) and RIGHT($i$) exist, then $A[\text{PARENT}(i)].key \geq A[\text{RIGHT}(i)].key$.
c. The subarray $A[1 : A.heap\text{-}size]$ satisfies the max-heap property, except that there may be one violation, which is that $A[i].key$ may be greater than $A[\text{PARENT}(i)].key$.

You may assume that the subarray $A[1 : A.heap\text{-}size]$ satisfies the max-heap property at the time MAX-HEAP-INCREASE-KEY is called.

***6.5-8***

Each exchange operation on line 6 of MAX-HEAP-INCREASE-KEY typically requires three assignments, not counting the updating of the mapping from objects to array indices. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments to just one assignment.

***6.5-9***

Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.3.)

***6.5-10***

The operation MAX-HEAP-DELETE$(A, x)$ deletes the object $x$ from max-heap $A$. Give an implementation of MAX-HEAP-DELETE for an $n$-element max-heap that runs in $O(\lg n)$ time plus the overhead for mapping priority queue objects to array indices.

***6.5-11***

Give an $O(n \lg k)$-time algorithm to merge $k$ sorted lists into one sorted list, where $n$ is the total number of elements in all the input lists. (*Hint:* Use a min-heap for $k$-way merging.)

## Problems

***6-1 Building a heap using insertion***

One way to build a heap is by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the procedure BUILD-MAX-HEAP′ on the facing page. It assumes that the objects being inserted are just the heap elements.

```
BUILD-MAX-HEAP′(A, n)
1  A.heap-size = 1
2  for i = 2 to n
3      MAX-HEAP-INSERT(A, A[i], n)
```

***a.*** Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

***b.*** Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an $n$-element heap.

### *6-2 Analysis of d-ary heaps*

A ***d-ary heap*** is like a binary heap, but (with one possible exception) nonleaf nodes have $d$ children instead of two children. In all parts of this problem, assume that the time to maintain the mapping between objects and heap elements is $O(1)$ per operation.

***a.*** Describe how to represent a $d$-ary heap in an array.

***b.*** Using $\Theta$-notation, express the height of a $d$-ary heap of $n$ elements in terms of $n$ and $d$.

***c.*** Give an efficient implementation of EXTRACT-MAX in a $d$-ary max-heap. Analyze its running time in terms of $d$ and $n$.

***d.*** Give an efficient implementation of INCREASE-KEY in a $d$-ary max-heap. Analyze its running time in terms of $d$ and $n$.

***e.*** Give an efficient implementation of INSERT in a $d$-ary max-heap. Analyze its running time in terms of $d$ and $n$.

### *6-3 Young tableaus*

An $m \times n$ ***Young tableau*** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be $\infty$, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \le mn$ finite numbers.

***a.*** Draw a $4 \times 4$ Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

***b.*** Argue that an $m \times n$ Young tableau $Y$ is empty if $Y[1,1] = \infty$. Argue that $Y$ is full (contains $mn$ elements) if $Y[m,n] < \infty$.

***c.*** Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m+n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. (*Hint:* Think about MAX-HEAPIFY.) Explain why your implementation of EXTRACT-MIN runs in $O(m+n)$ time.

***d.*** Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m+n)$ time.

***e.*** Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort $n^2$ numbers in $O(n^3)$ time.

***f.*** Give an $O(m+n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

## Chapter notes

The heapsort algorithm was invented by Williams [456], who also described how to implement a priority queue with a heap. The BUILD-MAX-HEAP procedure was suggested by Floyd [145]. Schaffer and Sedgewick [395] showed that in the best case, the number of times elements move in the heap during heapsort is approximately $(n/2) \lg n$ and that the average number of moves is approximately $n \lg n$.

We use min-heaps to implement min-priority queues in Chapters 15, 21, and 22. Other, more complicated, data structures give better time bounds for certain min-priority queue operations. Fredman and Tarjan [156] developed Fibonacci heaps, which support INSERT and DECREASE-KEY in $O(1)$ amortized time (see Chapter 16). That is, the average worst-case running time for these operations is $O(1)$. Brodal, Lagogiannis, and Tarjan [73] subsequently devised strict Fibonacci heaps, which make these time bounds the actual running times. If the keys are unique and drawn from the set $\{0, 1, \ldots, n-1\}$ of nonnegative integers, van Emde Boas trees [440, 441] support the operations INSERT, DELETE, SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR in $O(\lg \lg n)$ time.

If the data are $b$-bit integers, and the computer memory consists of addressable $b$-bit words, Fredman and Willard [157] showed how to implement MINIMUM in $O(1)$ time and INSERT and EXTRACT-MIN in $O(\sqrt{\lg n})$ time. Thorup [436] has

improved the $O(\sqrt{\lg n})$ bound to $O(\lg \lg n)$ time by using randomized hashing, requiring only linear space.

An important special case of priority queues occurs when the sequence of EXTRACT-MIN operations is ***monotone***, that is, the values returned by successive EXTRACT-MIN operations are monotonically increasing over time. This case arises in several important applications, such as Dijkstra's single-source shortest-paths algorithm, which we discuss in Chapter 22, and in discrete-event simulation. For Dijkstra's algorithm it is particularly important that the DECREASE-KEY operation be implemented efficiently. For the monotone case, if the data are integers in the range $1, 2, \ldots, C$, Ahuja, Mehlhorn, Orlin, and Tarjan [8] describe how to implement EXTRACT-MIN and INSERT in $O(\lg C)$ amortized time (Chapter 16 presents amortized analysis) and DECREASE-KEY in $O(1)$ time, using a data structure called a radix heap. The $O(\lg C)$ bound can be improved to $O(\sqrt{\lg C})$ using Fibonacci heaps in conjunction with radix heaps. Cherkassky, Goldberg, and Silverstein [90] further improved the bound to $O(\lg^{1/3+\epsilon} C)$ expected time by combining the multilevel bucketing structure of Denardo and Fox [112] with the heap of Thorup mentioned earlier. Raman [375] further improved these results to obtain a bound of $O\left(\min\left\{\lg^{1/4+\epsilon} C, \lg^{1/3+\epsilon} n\right\}\right)$, for any fixed $\epsilon > 0$.

Many other variants of heaps have been proposed. Brodal [72] surveys some of these developments.

# 7 Quicksort

The quicksort algorithm has a worst-case running time of $\Theta(n^2)$ on an input array of $n$ numbers. Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on average: its expected running time is $\Theta(n \lg n)$ when all numbers are distinct, and the constant factors hidden in the $\Theta(n \lg n)$ notation are small. Unlike merge sort, it also has the advantage of sorting in place (see page 158), and it works well even in virtual-memory environments.

Our study of quicksort is broken into four sections. Section 7.1 describes the algorithm and an important subroutine used by quicksort for partitioning. Because the behavior of quicksort is complex, we'll start with an intuitive discussion of its performance in Section 7.2 and analyze it precisely at the end of the chapter. Section 7.3 presents a randomized version of quicksort. When all elements are distinct,[1] this randomized algorithm has a good expected running time and no particular input elicits its worst-case behavior. (See Problem 7-2 for the case in which elements may be equal.) Section 7.4 analyzes the randomized algorithm, showing that it runs in $\Theta(n^2)$ time in the worst case and, assuming distinct elements, in expected $O(n \lg n)$ time.

---

[1] You can enforce the assumption that the values in an array $A$ are distinct at the cost of $\Theta(n)$ additional space and only constant overhead in running time by converting each input value $A[i]$ to an ordered pair $(A[i], i)$ with $(A[i], i) < (A[j], j)$ if $A[i] < A[j]$ or if $A[i] = A[j]$ and $i < j$. There are also more practical variants of quicksort that work well when elements are not distinct.

## 7.1 Description of quicksort

Quicksort, like merge sort, applies the divide-and-conquer method introduced in Section 2.3.1. Here is the three-step divide-and-conquer process for sorting a subarray $A[p:r]$:

**Divide** by partitioning (rearranging) the array $A[p:r]$ into two (possibly empty) subarrays $A[p:q-1]$ (the ***low side***) and $A[q+1:r]$ (the ***high side***) such that each element in the low side of the partition is less than or equal to the ***pivot*** $A[q]$, which is, in turn, less than or equal to each element in the high side. Compute the index $q$ of the pivot as part of this partitioning procedure.

**Conquer** by calling quicksort recursively to sort each of the subarrays $A[p:q-1]$ and $A[q+1:r]$.

**Combine** by doing nothing: because the two subarrays are already sorted, no work is needed to combine them. All elements in $A[p:q-1]$ are sorted and less than or equal to $A[q]$, and all elements in $A[q+1:r]$ are sorted and greater than or equal to the pivot $A[q]$. The entire subarray $A[p:r]$ cannot help but be sorted!

The QUICKSORT procedure implements quicksort. To sort an entire $n$-element array $A[1:n]$, the initial call is QUICKSORT$(A, 1, n)$.

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

### Partitioning the array

The key to the algorithm is the PARTITION procedure on the next page, which rearranges the subarray $A[p:r]$ in place, returning the index of the dividing point between the two sides of the partition.

Figure 7.1 shows how PARTITION works on an 8-element array. PARTITION always selects the element $x = A[r]$ as the pivot. As the procedure runs, each element falls into exactly one of four regions, some of which may be empty. At the start of each iteration of the **for** loop in lines 3–6, the regions satisfy certain properties, shown in Figure 7.2. We state these properties as a loop invariant:

```
PARTITION(A, p, r)
1  x = A[r]                              // the pivot
2  i = p − 1                             // highest index into the low side
3  for j = p to r − 1                    // process each element other than the pivot
4      if A[j] ≤ x                       // does this element belong on the low side?
5          i = i + 1                         // index of a new slot in the low side
6          exchange A[i] with A[j]       // put this element there
7  exchange A[i + 1] with A[r]           // pivot goes just to the right of the low side
8  return i + 1                          // new index of the pivot
```

At the beginning of each iteration of the loop of lines 3–6, for any array index $k$, the following conditions hold:

1. if $p \le k \le i$, then $A[k] \le x$ (the tan region of Figure 7.2);
2. if $i + 1 \le k \le j - 1$, then $A[k] > x$ (the blue region);
3. if $k = r$, then $A[k] = x$ (the yellow region).

We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that correctness follows from the invariant when the loop terminates.

**Initialization:** Prior to the first iteration of the loop, we have $i = p - 1$ and $j = p$. Because no values lie between $p$ and $i$ and no values lie between $i + 1$ and $j - 1$, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition.

**Maintenance:** As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when $A[j] > x$: the only action in the loop is to increment $j$. After $j$ has been incremented, the second condition holds for $A[j - 1]$ and all other entries remain unchanged. Figure 7.3(b) shows what happens when $A[j] \le x$: the loop increments $i$, swaps $A[i]$ and $A[j]$, and then increments $j$. Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j - 1] > x$, since the item that was swapped into $A[j - 1]$ is, by the loop invariant, greater than $x$.

**Termination:** Since the loop makes exactly $r - p$ iterations, it terminates, whereupon $j = r$. At that point, the unexamined subarray $A[j : r - 1]$ is empty, and every entry in the array belongs to one of the other three sets described by the invariant. Thus, the values in the array have been partitioned into three sets: those less than or equal to $x$ (the low side), those greater than $x$ (the high side), and a singleton set containing $x$ (the pivot).

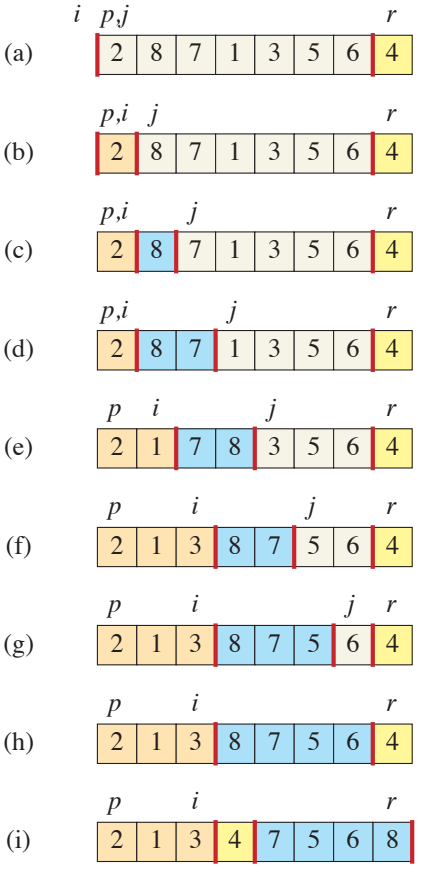


**Figure 7.1** The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element $x$. Tan array elements all belong to the low side of the partition, with values at most $x$. Blue elements belong to the high side, with values greater than $x$. White elements have not yet been put into either side of the partition, and the yellow element is the pivot $x$. **(a)** The initial array and variable settings. None of the elements have been placed into either side of the partition. **(b)** The value 2 is "swapped with itself" and put into the low side. **(c)–(d)** The values 8 and 7 are placed into to high side. **(e)** The values 1 and 8 are swapped, and the low side grows. **(f)** The values 3 and 7 are swapped, and the low side grows. **(g)–(h)** The high side of the partition grows to include 5 and 6, and the loop terminates. **(i)** Line 7 swaps the pivot element so that it lies between the two sides of the partition, and line 8 returns the pivot's new index.

The final two lines of PARTITION finish up by swapping the pivot with the left-most element greater than $x$, thereby moving the pivot into its correct place in the partitioned array, and then returning the pivot's new index. The output of PARTITION now satisfies the specifications given for the divide step. In fact, it satisfies a slightly stronger condition: after line 3 of QUICKSORT, $A[q]$ is strictly less than every element of $A[q+1:r]$.

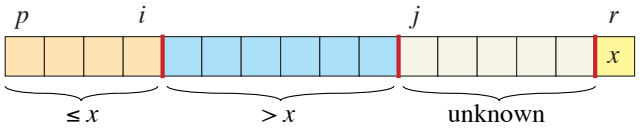


**Figure 7.2** The four regions maintained by the procedure PARTITION on a subarray $A[p:r]$. The tan values in $A[p:i]$ are all less than or equal to $x$, the blue values in $A[i+1:j-1]$ are all greater than $x$, the white values in $A[j:r-1]$ have unknown relationships to $x$, and $A[r] = x$.

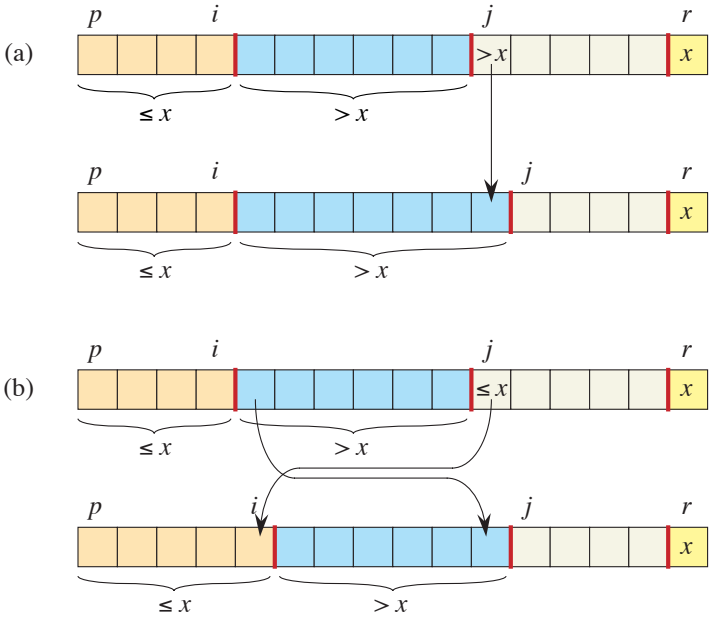


**Figure 7.3** The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment $j$, which maintains the loop invariant. **(b)** If $A[j] \le x$, index $i$ is incremented, $A[i]$ and $A[j]$ are swapped, and then $j$ is incremented. Again, the loop invariant is maintained.

Exercise 7.1-3 asks you to show that the running time of PARTITION on a subarray $A[p:r]$ of $n = r - p + 1$ elements is $\Theta(n)$.

## Exercises

### *7.1-1*

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

***7.1-2***
What value of $q$ does PARTITION return when all elements in the subarray $A[p:r]$ have the same value? Modify PARTITION so that $q = \lfloor (p+r)/2 \rfloor$ when all elements in the subarray $A[p:r]$ have the same value.

***7.1-3***
Give a brief argument that the running time of PARTITION on a subarray of size $n$ is $\Theta(n)$.

***7.1-4***
Modify QUICKSORT to sort into monotonically decreasing order.

## 7.2 Performance of quicksort

The running time of quicksort depends on how balanced each partitioning is, which in turn depends on which elements are used as pivots. If the two sides of a partition are about the same size—the partitioning is balanced—then the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. To allow you to gain some intuition before diving into a formal analysis, this section informally investigates how quicksort performs under the assumptions of balanced versus unbalanced partitioning.

But first, let's briefly look at the maximum amount of memory that quicksort requires. Although quicksort sorts in place according to the definition on page 158, the amount of memory it uses—aside from the array being sorted—is not constant. Since each recursive call requires a constant amount of space on the runtime stack, outside of the array being sorted, quicksort requires space proportional to the maximum depth of the recursion. As we'll see now, that could be as bad as $\Theta(n)$ in the worst case.

### Worst-case partitioning

The worst-case behavior for quicksort occurs when the partitioning produces one subproblem with $n - 1$ elements and one with 0 elements. (See Section 7.4.1.) Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns without doing anything, $T(0) = \Theta(1)$, and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \,. \end{aligned}$$

By summing the costs incurred at each level of the recursion, we obtain an arithmetic series (equation (A.3) on page 1141), which evaluates to $\Theta(n^2)$. Indeed, the substitution method can be used to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$. (See Exercise 7.2-1.)

Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$. The worst-case running time of quicksort is therefore no better than that of insertion sort. Moreover, the $\Theta(n^2)$ running time occurs when the input array is already completely sorted—a situation in which insertion sort runs in $O(n)$ time.

### Best-case partitioning

In the most even possible split, PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor (n-1)/2 \rfloor \le n/2$ and one of size $\lceil (n-1)/2 \rceil - 1 \le n/2$. In this case, quicksort runs much faster. An upper bound on the running time can then be described by the recurrence

$$T(n) = 2T(n/2) + \Theta(n) \,.$$

By case 2 of the master theorem (Theorem 4.1 on page 102), this recurrence has the solution $T(n) = \Theta(n \lg n)$. Thus, if the partitioning is equally balanced at every level of the recursion, an asymptotically faster algorithm results.

### Balanced partitioning

As the analyses in Section 7.4 will show, the average-case running time of quicksort is much closer to the best case than to the worst case. By appreciating how the balance of the partitioning affects the recurrence describing the running time, we can gain an understanding of why.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) = T(9n/10) + T(n/10) + \Theta(n) \,,$$

on the running time of quicksort. Figure 7.4 shows the recursion tree for this recurrence, where for simplicity the $\Theta(n)$ driving function has been replaced by $n$, which won't affect the asymptotic solution of the recurrence (as Exercise 4.7-1 on page 118 justifies). Every level of the tree has cost $n$, until the recursion bottoms out in a base case at depth $\log_{10} n = \Theta(\lg n)$, and then the levels have cost

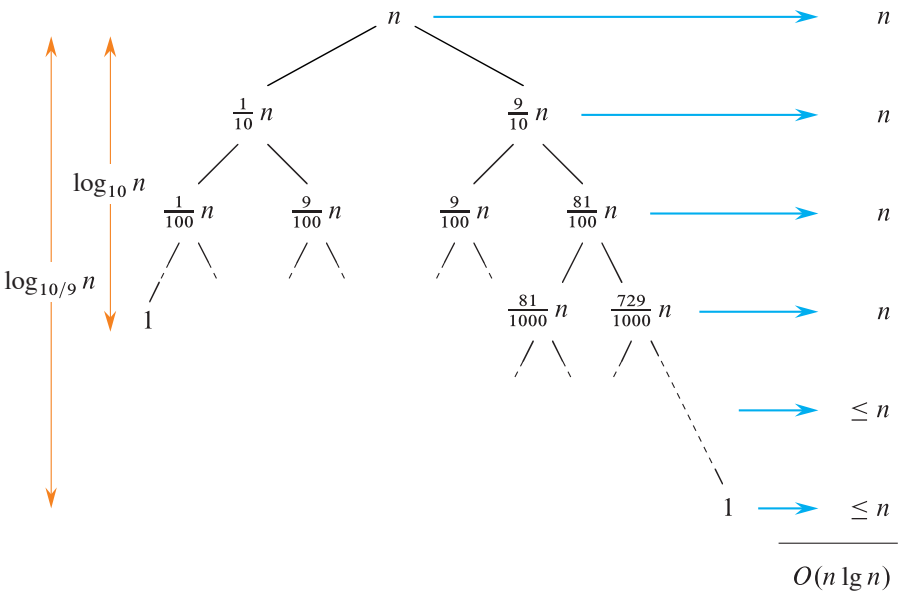


**Figure 7.4** A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right.

at most $n$. The recursion terminates at depth $\log_{10/9} n = \Theta(\lg n)$. Thus, with a 9-to-1 proportional split at every level of recursion, which intuitively seems highly unbalanced, quicksort runs in $O(n \lg n)$ time—asymptotically the same as if the split were right down the middle. Indeed, even a 99-to-1 split yields an $O(n \lg n)$ running time. In fact, any split of *constant* proportionality yields a recursion tree of depth $\Theta(\lg n)$, where the cost at each level is $O(n)$. The running time is therefore $O(n \lg n)$ whenever the split has constant proportionality. The ratio of the split affects only the constant hidden in the $O$-notation.

### Intuition for the average case

To develop a clear notion of the expected behavior of quicksort, we must assume something about how its inputs are distributed. Because quicksort determines the sorted order using only comparisons between input elements, its behavior depends on the relative ordering of the values in the array elements given as the input, not on the particular values in the array. As in the probabilistic analysis of the hiring problem in Section 5.2, assume that all permutations of the input numbers are equally likely and that the elements are distinct.

When quicksort runs on a random input array, the partitioning is highly unlikely to happen in the same way at every level, as our informal analysis has assumed.

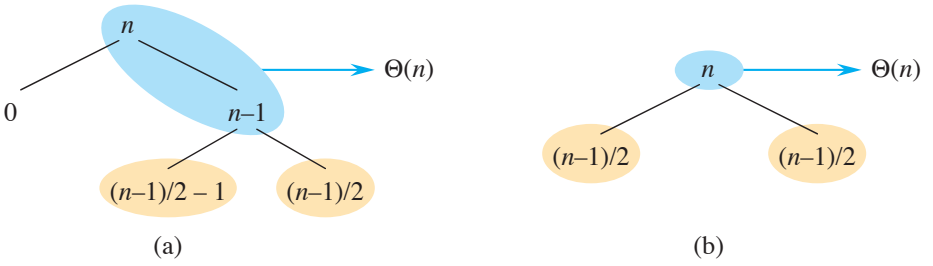


**Figure 7.5** **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs $n$ and produces a "bad" split: two subarrays of sizes 0 and $n-1$. The partitioning of the subarray of size $n-1$ costs $n-1$ and produces a "good" split: subarrays of size $(n-1)/2-1$ and $(n-1)/2$. **(b)** A single level of a recursion tree that is well balanced. In both parts, the partitioning cost for the subproblems shown with blue shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with tan shading, are no larger than the corresponding subproblems remaining to be solved in (b).

We expect that some of the splits will be reasonably well balanced and that some will be fairly unbalanced. For example, Exercise 7.2-6 asks you to show that about 80% of the time PARTITION produces a split that is at least as balanced as 9 to 1, and about 20% of the time it produces a split that is less balanced than 9 to 1.

In the average case, PARTITION produces a mix of "good" and "bad" splits. In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree. Suppose for the sake of intuition that the good and bad splits alternate levels in the tree, and that the good splits are best-case splits and the bad splits are worst-case splits. Figure 7.5(a) shows the splits at two consecutive levels in the recursion tree. At the root of the tree, the cost is $n$ for partitioning, and the subarrays produced have sizes $n-1$ and 0: the worst case. At the next level, the subarray of size $n-1$ undergoes best-case partitioning into subarrays of size $(n-1)/2-1$ and $(n-1)/2$. Let's assume that the base-case cost is 1 for the subarray of size 0.

The combination of the bad split followed by the good split produces three subarrays of sizes 0, $(n-1)/2-1$, and $(n-1)/2$ at a combined partitioning cost of $\Theta(n)+\Theta(n-1)=\Theta(n)$. This situation is at most a constant factor worse than that in Figure 7.5(b), namely, where a single level of partitioning produces two subarrays of size $(n-1)/2$, at a cost of $\Theta(n)$. Yet this latter situation is balanced! Intuitively, the $\Theta(n-1)$ cost of the bad split in Figure 7.5(a) can be absorbed into the $\Theta(n)$ cost of the good split, and the resulting split is good. Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: still $O(n \lg n)$, but with a slightly larger constant hidden by the $O$-notation. We'll analyze the expected running time of a randomized version of quicksort rigorously in Section 7.4.2.

### Exercises

***7.2-1***
Use the substitution method to prove that the recurrence $T(n) = T(n-1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

***7.2-2***
What is the running time of QUICKSORT when all elements of array $A$ have the same value?

***7.2-3***
Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array $A$ contains distinct elements and is sorted in decreasing order.

***7.2-4***
Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Explain persuasively why the procedure INSERTION-SORT might tend to beat the procedure QUICKSORT on this problem.

***7.2-5***
Suppose that the splits at every level of quicksort are in the constant proportion $\alpha$ to $\beta$, where $\alpha + \beta = 1$ and $0 < \alpha \leq \beta < 1$. Show that the minimum depth of a leaf in the recursion tree is approximately $\log_{1/\alpha} n$ and that the maximum depth is approximately $\log_{1/\beta} n$. (Don't worry about integer round-off.)

***7.2-6***
Consider an array with distinct elements and for which all permutations of the elements are equally likely. Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that PARTITION produces a split at least as balanced as $1 - \alpha$ to $\alpha$.

## 7.3 A randomized version of quicksort

In exploring the average-case behavior of quicksort, we have assumed that all permutations of the input numbers are equally likely. This assumption does not always hold, however, as, for example, in the situation laid out in the premise for

Exercise 7.2-4. Section 5.3 showed that judicious randomization can sometimes be added to an algorithm to obtain good expected performance over all inputs. For quicksort, randomization yields a fast and practical algorithm. Many software libraries provide a randomized version of quicksort as their algorithm of choice for sorting large data sets.

In Section 5.3, the RANDOMIZED-HIRE-ASSISTANT procedure explicitly permutes its input and then runs the deterministic HIRE-ASSISTANT procedure. We could do the same for quicksort as well, but a different randomization technique yields a simpler analysis. Instead of always using $A[r]$ as the pivot, a randomized version randomly chooses the pivot from the subarray $A[p:r]$, where each element in $A[p:r]$ has an equal probability of being chosen. It then exchanges that element with $A[r]$ before partitioning. Because the pivot is chosen randomly, we expect the split of the input array to be reasonably well balanced on average.

The changes to PARTITION and QUICKSORT are small. The new partitioning procedure, RANDOMIZED-PARTITION, simply swaps before performing the partitioning. The new quicksort procedure, RANDOMIZED-QUICKSORT, calls RANDOMIZED-PARTITION instead of PARTITION. We'll analyze this algorithm in the next section.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

## Exercises

***7.3-1***

Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

***7.3-2***
When RANDOMIZED-QUICKSORT runs, how many calls are made to the random-number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of $\Theta$-notation.

## 7.4 Analysis of quicksort

Section 7.2 gave some intuition for the worst-case behavior of quicksort and for why we expect the algorithm to run quickly. This section analyzes the behavior of quicksort more rigorously. We begin with a worst-case analysis, which applies to either QUICKSORT or RANDOMIZED-QUICKSORT, and conclude with an analysis of the expected running time of RANDOMIZED-QUICKSORT.

### 7.4.1 Worst-case analysis

We saw in Section 7.2 that a worst-case split at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which, intuitively, is the worst-case running time of the algorithm. We now prove this assertion.

We'll use the substitution method (see Section 4.3) to show that the running time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size $n$. Because the procedure PARTITION produces two subproblems with total size $n-1$, we obtain the recurrence

$$T(n) = \max\{T(q) + T(n-1-q) : 0 \le q \le n-1\} + \Theta(n)\,, \tag{7.1}$$

We guess that $T(n) \le cn^2$ for some constant $c > 0$. Substituting this guess into recurrence (7.1) yields

$$\begin{aligned} T(n) &\le \max\left\{cq^2 + c(n-1-q)^2 : 0 \le q \le n-1\right\} + \Theta(n) \\ &= c \cdot \max\left\{q^2 + (n-1-q)^2 : 0 \le q \le n-1\right\} + \Theta(n)\,. \end{aligned}$$

Let's focus our attention on the maximization. For $q = 0, 1, \ldots, n-1$, we have

$$\begin{aligned} q^2 + (n-1-q)^2 &= q^2 + (n-1)^2 - 2q(n-1) + q^2 \\ &= (n-1)^2 + 2q(q-(n-1)) \\ &\le (n-1)^2 \end{aligned}$$

because $q \le n-1$ implies that $2q(q-(n-1)) \le 0$. Thus every term in the maximization is bounded by $(n-1)^2$.

Continuing with our analysis of $T(n)$, we obtain

$$\begin{aligned} T(n) &\le c(n-1)^2 + \Theta(n) \\ &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

by picking the constant $c$ large enough that the $c(2n-1)$ term dominates the $\Theta(n)$ term. Thus $T(n) = O(n^2)$. Section 7.2 showed a specific case where quicksort takes $\Omega(n^2)$ time: when partitioning is maximally unbalanced. Thus, the worst-case running time of quicksort is $\Theta(n^2)$.

### 7.4.2 Expected running time

We have already seen the intuition behind why the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$: if, in each level of recursion, the split induced by RANDOMIZED-PARTITION puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth $\Theta(\lg n)$ and $O(n)$ work is performed at each level. Even if we add a few new levels with the most unbalanced split possible between these levels, the total time remains $O(n \lg n)$. We can analyze the expected running time of RANDOMIZED-QUICKSORT precisely by first understanding how the partitioning procedure operates and then using this understanding to derive an $O(n \lg n)$ bound on the expected running time. This upper bound on the expected running time, combined with the $\Theta(n \lg n)$ best-case bound we saw in Section 7.2, yields a $\Theta(n \lg n)$ expected running time. We assume throughout that the values of the elements being sorted are distinct.

#### Running time and comparisons

The QUICKSORT and RANDOMIZED-QUICKSORT procedures differ only in how they select pivot elements. They are the same in all other respects. We can therefore analyze RANDOMIZED-QUICKSORT by considering the QUICKSORT and PARTITION procedures, but with the assumption that pivot elements are selected randomly from the subarray passed to RANDOMIZED-PARTITION. Let's start by relating the asymptotic running time of QUICKSORT to the number of times elements are compared (all in line 4 of PARTITION), understanding that this analysis also applies to RANDOMIZED-QUICKSORT. Note that we are counting the number of times that *array elements* are compared, not comparisons of indices.

***Lemma 7.1***

The running time of QUICKSORT on an $n$-element array is $O(n + X)$, where $X$ is the number of element comparisons performed.

***Proof*** The running time of QUICKSORT is dominated by the time spent in the PARTITION procedure. Each time PARTITION is called, it selects a pivot element, which is never included in any future recursive calls to QUICKSORT and PARTITION. Thus, there can be at most $n$ calls to PARTITION over the entire execution of the quicksort algorithm. Each time QUICKSORT calls PARTITION, it also recursively calls itself twice, so there are at most $2n$ calls to the QUICKSORT procedure itself.

One call to PARTITION takes $O(1)$ time plus an amount of time that is proportional to the number of iterations of the **for** loop in lines 3–6. Each iteration of this **for** loop performs one comparison in line 4, comparing the pivot element to another element of the array $A$. Therefore, the total time spent in the **for** loop across all executions is proportional to $X$. Since there are at most $n$ calls to PARTITION and the time spent outside the **for** loop is $O(1)$ for each call, the total time spent in PARTITION outside of the **for** loop is $O(n)$. Thus the total time for quicksort is $O(n + X)$. ■

Our goal for analyzing RANDOMIZED-QUICKSORT, therefore, is to compute the expected value $\mathrm{E}[X]$ of the random variable $X$ denoting the total number of comparisons performed in all calls to PARTITION. To do so, we must understand when the quicksort algorithm compares two elements of the array and when it does not. For ease of analysis, let's index the elements of the array $A$ by their position in the sorted output, rather than their position in the input. That is, although the elements in $A$ may start out in any order, we'll refer to them by $z_1, z_2, \ldots, z_n$, where $z_1 < z_2 < \cdots < z_n$, with strict inequality because we assume that all elements are distinct. We denote the set $\{z_i, z_{i+1}, \ldots, z_j\}$ by $Z_{ij}$.

The next lemma characterizes when two elements are compared.

***Lemma 7.2***

During the execution of RANDOMIZED-QUICKSORT on an array of $n$ distinct elements $z_1 < z_2 < \cdots < z_n$, an element $z_i$ is compared with an element $z_j$, where $i < j$, if and only if one of them is chosen as a pivot before any other element in the set $Z_{ij}$. Moreover, no two elements are ever compared twice.

***Proof*** Let's look at the first time that an element $x \in Z_{ij}$ is chosen as a pivot during the execution of the algorithm. There are three cases to consider. If $x$ is neither $z_i$ nor $z_j$—that is, $z_i < x < z_j$—then $z_i$ and $z_j$ are not compared at any subsequent time, because they fall into different sides of the partition around $x$. If $x = z_i$, then PARTITION compares $z_i$ with every other item in $Z_{ij}$. Similarly, if $x = z_j$, then PARTITION compares $z_j$ with every other item in $Z_{ij}$. Thus, $z_i$ and $z_j$ are compared if and only if the first element to be chosen as a pivot from $Z_{ij}$ is either $z_i$ or $z_j$. In the latter two cases, where one of $z_i$ and $z_j$ is chosen

as a pivot, since the pivot is removed from future comparisons, it is never compared again with the other element. ■

As an example of this lemma, consider an input to quicksort of the numbers 1 through 10 in some arbitrary order. Suppose that the first pivot element is 7. Then the first call to PARTITION separates the numbers into two sets: $\{1, 2, 3, 4, 5, 6\}$ and $\{8, 9, 10\}$. In the process, the pivot element 7 is compared with all other elements, but no number from the first set (e.g., 2) is or ever will be compared with any number from the second set (e.g., 9). The values 7 and 9 are compared because 7 is the first item from $Z_{7,9}$ to be chosen as a pivot. In contrast, 2 and 9 are never compared because the first pivot element chosen from $Z_{2,9}$ is 7.

The next lemma gives the probability that two elements are compared.

***Lemma 7.3***

Consider an execution of the procedure RANDOMIZED-QUICKSORT on an array of $n$ distinct elements $z_1 < z_2 < \cdots < z_n$. Given two arbitrary elements $z_i$ and $z_j$ where $i < j$, the probability that they are compared is $2/(j - i + 1)$.

***Proof*** Let's look at the tree of recursive calls that RANDOMIZED-QUICKSORT makes, and consider the sets of elements provided as input to each call. Initially, the root set contains all the elements of $Z_{ij}$, since the root set contains every element in $A$. The elements belonging to $Z_{ij}$ all stay together for each recursive call of RANDOMIZED-QUICKSORT until PARTITION chooses some element $x \in Z_{ij}$ as a pivot. From that point on, the pivot $x$ appears in no subsequent input set. The first time that RANDOMIZED-SELECT chooses a pivot $x \in Z_{ij}$ from a set containing all the elements of $Z_{ij}$, each element in $Z_{ij}$ is equally likely to be $x$ because the pivot is chosen uniformly at random. Since $|Z_{ij}| = j - i + 1$, the probability is $1/(j - i + 1)$ that any given element in $Z_{ij}$ is the first pivot chosen from $Z_{ij}$. Thus, by Lemma 7.2, we have

$$\begin{aligned}
\Pr\{z_i \text{ is compared with } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is the first pivot chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is the first pivot chosen from } Z_{ij}\} \\
&= \frac{2}{j - i + 1},
\end{aligned}$$

where the second line follows from the first because the two events are mutually exclusive. ■

We can now complete the analysis of randomized quicksort.

***Theorem 7.4***
The expected running time of RANDOMIZED-QUICKSORT on an input of $n$ distinct elements is $O(n \lg n)$.

***Proof*** The analysis uses indicator random variables (see Section 5.2). Let the $n$ distinct elements be $z_1 < z_2 < \cdots < z_n$, and for $1 \le i < j \le n$, define the indicator random variable $X_{ij} = \mathrm{I}\{z_i \text{ is compared with } z_j\}$. From Lemma 7.2, each pair is compared at most once, and so we can express $X$ as follows:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ .$$

By taking expectations of both sides and using linearity of expectation (equation (C.24) on page 1192) and Lemma 5.1 on page 130, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared with } z_j\} && \text{(by Lemma 5.1)} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} && \text{(by Lemma 7.3) .}
\end{aligned}$$

We can evaluate this sum using a change of variables ($k = j - i$) and the bound on the harmonic series in equation (A.9) on page 1142:

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) \ .
\end{aligned}$$

This bound and Lemma 7.1 allow us to conclude that the expected running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$ (assuming that the element values are distinct). ■

### Exercises

***7.4-1***
Show that the recurrence

$$T(n) = \max\{T(q) + T(n-q-1) : 0 \le q \le n-1\} + \Theta(n)$$

has a lower bound of $T(n) = \Omega(n^2)$.

***7.4-2***
Show that quicksort's best-case running time is $\Omega(n \lg n)$.

***7.4-3***
Show that the expression $q^2 + (n-q-1)^2$ achieves its maximum value over $q = 0, 1, \ldots, n-1$ when $q = 0$ or $q = n-1$.

***7.4-4***
Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

***7.4-5***
Coarsening the recursion, as we did in Problem 2-1 for merge sort, is a common way to improve the running time of quicksort in practice. We modify the base case of the recursion so that if the array has fewer than $k$ elements, the subarray is sorted by insertion sort, rather than by continued recursive calls to quicksort. Argue that the randomized version of this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should you pick $k$, both in theory and in practice?

★ ***7.4-6***
Consider modifying the PARTITION procedure by randomly picking three elements from subarray $A[p:r]$ and partitioning about their median (the middle value of the three elements). Approximate the probability of getting worse than an $\alpha$-to-$(1-\alpha)$ split, as a function of $\alpha$ in the range $0 < \alpha < 1/2$.

## Problems

### *7-1 Hoare partition correctness*

The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partitioning algorithm, which is due to C. A. R. Hoare.

```
HOARE-PARTITION(A, p, r)
 1  x = A[p]
 2  i = p − 1
 3  j = r + 1
 4  while TRUE
 5      repeat
 6          j = j − 1
 7      until A[j] ≤ x
 8      repeat
 9          i = i + 1
10      until A[i] ≥ x
11      if i < j
12          exchange A[i] with A[j]
13      else return j
```

***a.*** Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and the indices $i$ and $j$ after each iteration of the **while** loop in lines 4–13.

***b.*** Describe how the PARTITION procedure in Section 7.1 differs from HOARE-PARTITION when all elements in $A[p:r]$ are equal. Describe a practical advantage of HOARE-PARTITION over PARTITION for use in quicksort.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p:r]$ contains at least two elements, prove the following:

***c.*** The indices $i$ and $j$ are such that the procedure never accesses an element of $A$ outside the subarray $A[p:r]$.

***d.*** When HOARE-PARTITION terminates, it returns a value $j$ such that $p \le j < r$.

***e.*** Every element of $A[p:j]$ is less than or equal to every element of $A[j+1:r]$ when HOARE-PARTITION terminates.

The PARTITION procedure in Section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two partitions $A[p:j]$ and $A[j+1:r]$. Since $p \le j < r$, neither partition is empty.

***f.*** Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

### *7-2 Quicksort with equal element values*

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. This problem examines what happens when they are not.

***a.*** Suppose that all element values are equal. What is randomized quicksort's running time in this case?

***b.*** The PARTITION procedure returns an index $q$ such that each element of $A[p:q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1:r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p:r]$ and returns two indices $q$ and $t$, where $p \le q \le t \le r$, such that

- all elements of $A[q:t]$ are equal,
- each element of $A[p:q-1]$ is less than $A[q]$, and
- each element of $A[t+1:r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r-p)$ time.

***c.*** Modify the RANDOMIZED-PARTITION procedure to call PARTITION$'$, and name the new procedure RANDOMIZED-PARTITION$'$. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT$'(A, p, r)$ that calls RANDOMIZED-PARTITION$'$ and recurses only on partitions where elements are not known to be equal to each other.

***d.*** Using QUICKSORT$'$, adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct.

### *7-3 Alternative quicksort analysis*

An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to RANDOMIZED-QUICKSORT, rather than on the number of comparisons performed. As in the analysis of Section 7.4.2, assume that the values of the elements are distinct.

***a.*** Argue that, given an array of size $n$, the probability that any particular element is chosen as the pivot is $1/n$. Use this probability to define indicator random variables $X_i = \mathrm{I}\{i\text{th smallest element is chosen as the pivot}\}$. What is $\mathrm{E}[X_i]$?

***b.*** Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size $n$. Argue that

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{q=1}^{n} X_q \left(T(q-1) + T(n-q) + \Theta(n)\right)\right] . \tag{7.2}$$

***c.*** Show how to rewrite equation (7.2) as

$$\mathrm{E}[T(n)] = \frac{2}{n}\sum_{q=1}^{n-1} \mathrm{E}[T(q)] + \Theta(n) . \tag{7.3}$$

***d.*** Show that

$$\sum_{q=1}^{n-1} q \lg q \le \frac{n^2}{2}\lg n - \frac{n^2}{8} \tag{7.4}$$

for $n \ge 2$. (*Hint:* Split the summation into two parts, one summation for $q = 1, 2, \ldots, \lceil n/2 \rceil - 1$ and one summation for $q = \lceil n/2 \rceil, \ldots, n-1$.)

***e.*** Using the bound from equation (7.4), show that the recurrence in equation (7.3) has the solution $\mathrm{E}[T(n)] = O(n \lg n)$. (*Hint:* Show, by substitution, that $\mathrm{E}[T(n)] \le an \lg n$ for sufficiently large $n$ and for some positive constant $a$.)

### *7-4 Stooge sort*

Professors Howard, Fine, and Howard have proposed a deceptively simple sorting algorithm, named stooge sort in their honor, appearing on the following page.

***a.*** Argue that the call STOOGE-SORT$(A, 1, n)$ correctly sorts the array $A[1:n]$.

***b.*** Give a recurrence for the worst-case running time of STOOGE-SORT and a tight asymptotic ($\Theta$-notation) bound on the worst-case running time.

***c.*** Compare the worst-case running time of STOOGE-SORT with that of insertion sort, merge sort, heapsort, and quicksort. Do the professors deserve tenure?

```
STOOGE-SORT(A, p, r)
1  if A[p] > A[r]
2      exchange A[p] with A[r]
3  if p + 1 < r
4      k = ⌊(r − p + 1)/3⌋          // round down
5      STOOGE-SORT(A, p, r − k)     // first two-thirds
6      STOOGE-SORT(A, p + k, r)     // last two-thirds
7      STOOGE-SORT(A, p, r − k)     // first two-thirds again
```

### 7-5 *Stack depth for quicksort*

The QUICKSORT procedure of Section 7.1 makes two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the low side of the partition and then it recursively sorts the high side of the partition. The second recursive call in QUICKSORT is not really necessary, because the procedure can instead use an iterative control structure. This transformation technique, called ***tail-recursion elimination***, is provided automatically by good compilers. Applying tail-recursion elimination transforms QUICKSORT into the TRE-QUICKSORT procedure.

```
TRE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and then sort the low side.
3      q = PARTITION(A, p, r)
4      TRE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

***a.*** Argue that TRE-QUICKSORT$(A, 1, n)$ correctly sorts the array $A[1:n]$.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is called, its information is ***pushed*** onto the stack, and when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

***b.*** Describe a scenario in which TRE-QUICKSORT's stack depth is $\Theta(n)$ on an $n$-element input array.

***c.*** Modify TRE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

### *7-6 Median-of-3 partition*

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. A common approach is the ***median-of-3*** method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See Exercise 7.4-6.) For this problem, assume that the $n$ elements in the input subarray $A[p:r]$ are distinct and that $n \geq 3$. Denote the sorted version of $A[p:r]$ by $z_1, z_2, \ldots, z_n$. Using the median-of-3 method to choose the pivot element $x$, define $p_i = \Pr\{x = z_i\}$.

***a.*** Give an exact formula for $p_i$ as a function of $n$ and $i$ for $i = 2, 3, \ldots, n-1$. (Observe that $p_1 = p_n = 0$.)

***b.*** By what amount does the median-of-3 method increase the likelihood of choosing the pivot to be $x = z_{\lfloor (n+1)/2 \rfloor}$, the median of $A[p:r]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

***c.*** Suppose that we define a "good" split to mean choosing the pivot as $x = z_i$, where $n/3 \leq i \leq 2n/3$. By what amount does the median-of-3 method increase the likelihood of getting a good split compared with the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

***d.*** Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

### *7-7 Fuzzy sorting of intervals*

Consider a sorting problem in which you do not know the numbers exactly. Instead, for each number, you know an interval on the real line to which it belongs. That is, you are given $n$ closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. The goal is to ***fuzzy-sort*** these intervals: to produce a permutation $\langle i_1, i_2, \ldots, i_n \rangle$ of the intervals such that for $j = 1, 2, \ldots, n$, there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \cdots \leq c_n$.

***a.*** Design a randomized algorithm for fuzzy-sorting $n$ intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the $a_i$ values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the prob-

lem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

***b.*** Argue that your algorithm runs in $\Theta(n \lg n)$ expected time in general, but runs in $\Theta(n)$ expected time when all of the intervals overlap (i.e., when there exists a value $x$ such that $x \in [a_i, b_i]$ for all $i$). Your algorithm should not be checking for this case explicitly, but rather, its performance should naturally improve as the amount of overlap increases.

## Chapter notes

Quicksort was invented by Hoare [219], and his version of PARTITION appears in Problem 7-1. Bentley [51, p. 117] attributes the PARTITION procedure given in Section 7.1 to N. Lomuto. The analysis in Section 7.4 based on an analysis due to Motwani and Raghavan [336]. Sedgewick [401] and Bentley [51] provide good references on the details of implementation and how they matter.

McIlroy [323] shows how to engineer a "killer adversary" that produces an array on which virtually any implementation of quicksort takes $\Theta(n^2)$ time.

# 8 Sorting in Linear Time

We have now seen a handful of algorithms that can sort $n$ numbers in $O(n \lg n)$ time. Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of $n$ input numbers that causes the algorithm to run in $\Omega(n \lg n)$ time.

These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements*. We call such sorting algorithms ***comparison sorts***. All the sorting algorithms introduced thus far are comparison sorts.

In Section 8.1, we'll prove that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort $n$ elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Sections 8.2, 8.3, and 8.4 examine three sorting algorithms—counting sort, radix sort, and bucket sort—that run in linear time on certain types of inputs. Of course, these algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Omega(n \lg n)$ lower bound does not apply to them.

## 8.1 Lower bounds for sorting

A comparison sort uses only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements $a_i$ and $a_j$, it performs one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i = a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order. It may not inspect the values of the elements or gain order information about them in any other way.

Since we are proving a lower bound, we assume without loss of generality in this section that all the input elements are distinct. After all, a lower bound for distinct elements applies when elements may or may not be distinct. Consequently,

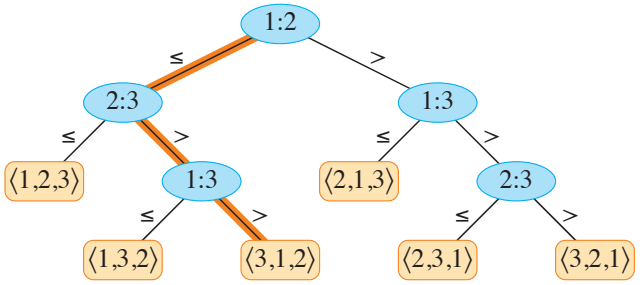


**Figure 8.1** The decision tree for insertion sort operating on three elements. An internal node (shown in blue) annotated by $i{:}j$ indicates a comparison between $a_i$ and $a_j$. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The highlighted path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$. Going left from the root node, labeled 1:2, indicates that $a_1 \le a_2$. Going right from the node labeled 2:3 indicates that $a_2 > a_3$. Going right from the node labeled 1:3 indicates that $a_1 > a_3$. Therefore, we have the ordering $a_3 \le a_1 \le a_2$, as indicated in the leaf labeled $\langle 3, 1, 2 \rangle$. Because the three input elements have $3! = 6$ possible permutations, the decision tree must have at least 6 leaves.

comparisons of the form $a_i = a_j$ are useless, which means that we can assume that no comparisons for exact equality occur. Moreover, the comparisons $a_i \le a_j$, $a_i \ge a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent in that they yield identical information about the relative order of $a_i$ and $a_j$. We therefore assume that all comparisons have the form $a_i \le a_j$.

### The decision-tree model

We can view comparison sorts abstractly in terms of decision trees. A ***decision tree*** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored. Figure 8.1 shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

A decision tree has each internal node annotated by $i{:}j$ for some $i$ and $j$ in the range $1 \le i, j \le n$, where $n$ is the number of elements in the input sequence. We also annotate each leaf by a permutation $\langle \pi(1), \pi(2), \pi(n) \rangle$. (See Section C.1 for background on permutations.) Indices in the internal nodes and the leaves always refer to the original positions of the array elements at the start of the sorting algorithm. The execution of the comparison sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \le a_j$. The left subtree then dictates sub-

sequent comparisons once we know that $a_i \le a_j$, and the right subtree dictates subsequent comparisons when $a_i > a_j$. Arriving at a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on $n$ elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct. Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort. (We call such leaves "reachable.") Thus, we consider only decision trees in which each permutation appears as a reachable leaf.

### A lower bound for the worst case

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

***Theorem 8.1***
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

***Proof*** From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height $h$ with $l$ reachable leaves corresponding to a comparison sort on $n$ elements. Because each of the $n!$ permutations of the input appears as one or more leaves, we have $n! \le l$. Since a binary tree of height $h$ has no more than $2^h$ leaves, we have

$$n! \le l \le 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\ge \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.28) on page 67) .} \end{aligned}$$

■

***Corollary 8.2***
Heapsort and merge sort are asymptotically optimal comparison sorts.

***Proof*** The $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound from Theorem 8.1. ■

### Exercises

***8.1-1***
What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

***8.1-2***
Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

***8.1-3***
Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length $n$. What about a fraction of $1/n$ of the inputs of length $n$? What about a fraction $1/2^n$?

***8.1-4***
You are given an $n$-element input sequence, and you know in advance that it is partly sorted in the following sense. Each element initially in position $i$ such that $i \bmod 4 = 0$ is either already in its correct position, or it is one place away from its correct position. For example, you know that after sorting, the element initially in position 12 belongs in position 11, 12, or 13. You have no advance information about the other elements, in positions $i$ where $i \bmod 4 \neq 0$. Show that an $\Omega(n \lg n)$ lower bound on comparison-based sorting still holds in this case.

## 8.2 Counting sort

***Counting sort*** assumes that each of the $n$ input elements is an integer in the range 0 to $k$, for some integer $k$. It runs in $\Theta(n + k)$ time, so that when $k = O(n)$, counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element $x$, the number of elements less than or equal to $x$. It then uses this information to place element $x$ directly into its position in the output array. For example, if 17 elements are less than or equal to $x$, then $x$ belongs in output position 17. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

The COUNTING-SORT procedure on the facing page takes as input an array $A[1:n]$, the size $n$ of this array, and the limit $k$ on the nonnegative integer values in $A$. It returns its sorted output in the array $B[1:n]$ and uses an array $C[0:k]$ for temporary working storage.

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Figure 8.2 illustrates counting sort. After the **for** loop of lines 2–3 initializes the array $C$ to all zeros, the **for** loop of lines 4–5 makes a pass over the array $A$ to inspect each input element. Each time it finds an input element whose value is $i$, it increments $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to $i$ for each integer $i = 0, 1, \ldots, k$. Lines 7–8 determine for each $i = 0, 1, \ldots, k$ how many input elements are less than or equal to $i$ by keeping a running sum of the array $C$.

Finally, the **for** loop of lines 11–13 makes another pass over $A$, but in reverse, to place each element $A[j]$ into its correct sorted position in the output array $B$. If all $n$ elements are distinct, then when line 11 is first entered, for each $A[j]$, the value $C[A[j]]$ is the correct final position of $A[j]$ in the output array, since there are $C[A[j]]$ elements less than or equal to $A[j]$. Because the elements might not be distinct, the loop decrements $C[A[j]]$ each time it places a value $A[j]$ into $B$. Decrementing $C[A[j]]$ causes the previous element in $A$ with a value equal to $A[j]$, if one exists, to go to the position immediately before $A[j]$ in the output array $B$.

How much time does counting sort require? The **for** loop of lines 2–3 takes $\Theta(k)$ time, the **for** loop of lines 4–5 takes $\Theta(n)$ time, the **for** loop of lines 7–8 takes $\Theta(k)$ time, and the **for** loop of lines 11–13 takes $\Theta(n)$ time. Thus, the overall time is $\Theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Counting sort can beat the lower bound of $\Omega(n \lg n)$ proved in Section 8.1 because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the

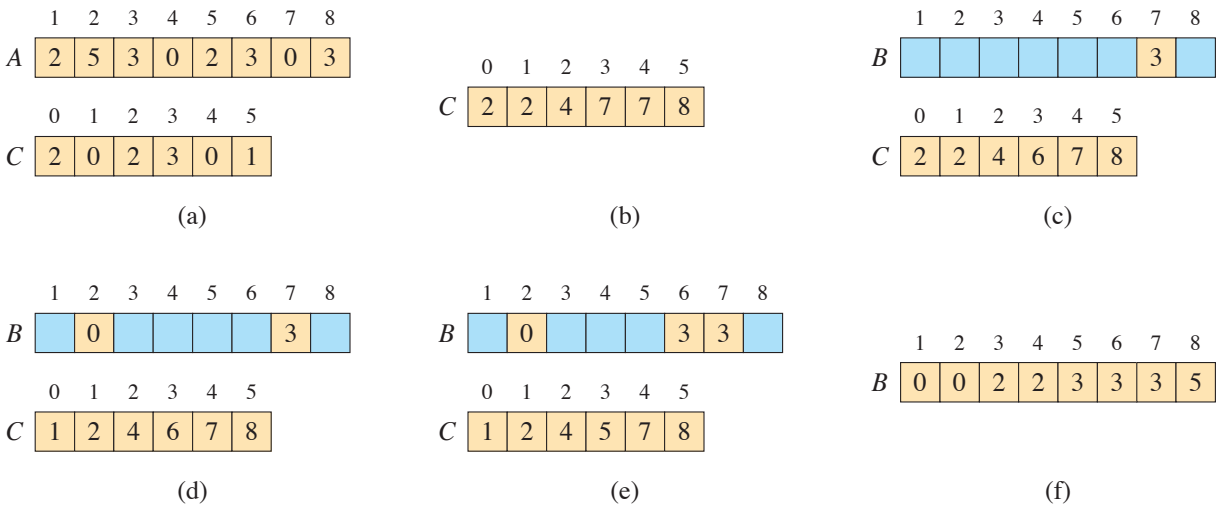


**Figure 8.2** The operation of COUNTING-SORT on an input array $A[1:8]$, where each element of $A$ is a nonnegative integer no larger than $k = 5$. **(a)** The array $A$ and the auxiliary array $C$ after line 5. **(b)** The array $C$ after line 8. **(c)–(e)** The output array $B$ and the auxiliary array $C$ after one, two, and three iterations of the loop in lines 11–13, respectively. Only the tan elements of array $B$ have been filled in. **(f)** The final sorted output array $B$.

elements to index into an array. The $\Omega(n \lg n)$ lower bound for sorting does not apply when we depart from the comparison sort model.

An important property of counting sort is that it is ***stable***: elements with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

## Exercises

### *8.2-1*

Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$.

### *8.2-2*

Prove that COUNTING-SORT is stable.

***8.2-3***
Suppose that we were to rewrite the **for** loop header in line 11 of the COUNTING-SORT as

11 **for** $j = 1$ **to** $n$

Show that the algorithm still works properly, but that it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

***8.2-4***
Prove the following loop invariant for COUNTING-SORT:

> At the start of each iteration of the **for** loop of lines 11–13, the last element in $A$ with value $i$ that has not yet been copied into $B$ belongs in $B[C[i]]$.

***8.2-5***
Suppose that the array being sorted contains only integers in the range 0 to $k$ and that there are no satellite data to move with those keys. Modify counting sort to use just the arrays $A$ and $C$, putting the sorted result back into array $A$ instead of into a new array $B$.

***8.2-6***
Describe an algorithm that, given $n$ integers in the range 0 to $k$, preprocesses its input and then answers any query about how many of the $n$ integers fall into a range $[a : b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

***8.2-7***
Counting sort can also work efficiently if the input values have fractional parts, but the number of digits in the fractional part is small. Suppose that you are given $n$ numbers in the range 0 to $k$, each with at most $d$ decimal (base 10) digits to the right of the decimal point. Modify counting sort to run in $\Theta(n + 10^d k)$ time.

## 8.3 Radix sort

***Radix sort*** is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one

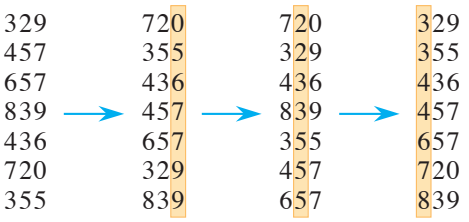


**Figure 8.3** The operation of radix sort on seven 3-digit numbers. The leftmost column is the input. The remaining columns show the numbers after successive sorts on increasingly significant digit positions. Tan shading indicates the digit position sorted on to produce each list from the previous one.

of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

For decimal digits, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.) A $d$-digit number occupies a field of $d$ columns. Since the card sorter can look at only one column at a time, the problem of sorting $n$ cards on a $d$-digit number requires a sorting algorithm.

Intuitively, you might sort numbers on their *most significant* (leftmost) digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that you would have to keep track of. (See Exercise 8.3-6.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on the *least significant* digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner. The process continues until the cards have been sorted on all $d$ digits. Remarkably, at that point the cards are fully sorted on the $d$-digit number. Thus, only $d$ passes through the deck are required to sort. Figure 8.3 shows how radix sort operates on a "deck" of seven 3-digit numbers.

In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator must be careful not to change the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column.

In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates,

compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day (the "least significant" part), next on month, and finally on year.

The code for radix sort is straightforward. The RADIX-SORT procedure assumes that each element in array $A[1:n]$ has $d$ digits, where digit 1 is the lowest-order digit and digit $d$ is the highest-order digit.

```
RADIX-SORT(A, n, d)
1  for i = 1 to d
2      use a stable sort to sort array A[1:n] on digit i
```

Although the pseudocode for RADIX-SORT does not specify which stable sort to use, COUNTING-SORT is commonly used. If you use COUNTING-SORT as the stable sort, you can make RADIX-SORT a little more efficient by revising COUNTING-SORT to take a pointer to the output array as a parameter, having RADIX-SORT preallocate this array, and alternating input and output between the two arrays in successive iterations of the **for** loop in RADIX-SORT.

***Lemma 8.3***
Given $n$ $d$-digit numbers in which each digit can take on up to $k$ possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time.

***Proof*** The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to $k-1$ (so that it can take on $k$ possible values), and $k$ is not too large, counting sort is the obvious choice. Each pass over $n$ $d$-digit numbers then takes $\Theta(n+k)$ time. There are $d$ passes, and so the total time for radix sort is $\Theta(d(n+k))$. ■

When $d$ is constant and $k = O(n)$, we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits.

***Lemma 8.4***
Given $n$ $b$-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2^r))$ time if the stable sort it uses takes $\Theta(n+k)$ time for inputs in the range 0 to $k$.

***Proof*** For a value $r \leq b$, view each key as having $d = \lceil b/r \rceil$ digits of $r$ bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32, r = 8, k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes $\Theta(n + k) = \Theta(n + 2^r)$ time and there are $d$ passes, for a total running time of $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r))$. ■

Given $n$ and $b$, what value of $r \leq b$ minimizes the expression $(b/r)(n + 2^r)$? As $r$ decreases, the factor $b/r$ increases, but as $r$ increases so does $2^r$. The answer depends on whether $b < \lfloor \lg n \rfloor$. If $b < \lfloor \lg n \rfloor$, then $r \leq b$ implies $(n+2^r) = \Theta(n)$. Thus, choosing $r = b$ yields a running time of $(b/b)(n + 2^b) = \Theta(n)$, which is asymptotically optimal. If $b \geq \lfloor \lg n \rfloor$, then choosing $r = \lfloor \lg n \rfloor$ gives the best running time to within a constant factor, which we can see as follows.[1] Choosing $r = \lfloor \lg n \rfloor$ yields a running time of $\Theta(bn/\lg n)$. As $r$ increases above $\lfloor \lg n \rfloor$, the $2^r$ term in the numerator increases faster than the $r$ term in the denominator, and so increasing $r$ above $\lfloor \lg n \rfloor$ yields a running time of $\Omega(bn/\lg n)$. If instead $r$ were to decrease below $\lfloor \lg n \rfloor$, then the $b/r$ term increases and the $n + 2^r$ term remains at $\Theta(n)$.

Is radix sort preferable to a comparison-based sorting algorithm, such as quicksort? If $b = O(\lg n)$, as is often the case, and $r \approx \lg n$, then radix sort's running time is $\Theta(n)$, which appears to be better than quicksort's expected running time of $\Theta(n \lg n)$. The constant factors hidden in the $\Theta$-notation differ, however. Although radix sort may make fewer passes than quicksort over the $n$ keys, each pass of radix sort may take significantly longer. Which sorting algorithm to prefer depends on the characteristics of the implementations, of the underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort), and of the input data. Moreover, the version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the $\Theta(n \lg n)$-time comparison sorts do. Thus, when primary memory storage is at a premium, an in-place algorithm such as quicksort could be the better choice.

## Exercises

### *8.3-1*

Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

---

[1] The choice of $r = \lfloor \lg n \rfloor$ assumes that $n > 1$. If $n \leq 1$, there is nothing to sort.

***8.3-2***
Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any comparison sort stable. How much additional time and space does your scheme entail?

***8.3-3***
Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

***8.3-4***
Suppose that COUNTING-SORT is used as the stable sort within RADIX-SORT. If RADIX-SORT calls COUNTING-SORT $d$ times, then since each call of COUNTING-SORT makes two passes over the data (lines 4–5 and 11–13), altogether $2d$ passes over the data occur. Describe how to reduce the total number of passes to $d + 1$.

***8.3-5***
Show how to sort $n$ integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

★ ***8.3-6***
In the first card-sorting algorithm in this section, which sorts on the most significant digit first, exactly how many sorting passes are needed to sort $d$-digit decimal numbers in the worst case? How many piles of cards does an operator need to keep track of in the worst case?

## 8.4 Bucket sort

***Bucket sort*** assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$. Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval $[0, 1)$. (See Section C.2 for a definition of a uniform distribution.)

Bucket sort divides the interval $[0, 1)$ into $n$ equal-sized subintervals, or ***buckets***, and then distributes the $n$ input numbers into the buckets. Since the inputs are uniformly and independently distributed over $[0, 1)$, we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

The BUCKET-SORT procedure on the next page assumes that the input is an array $A[1 : n]$ and that each element $A[i]$ in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array $B[0 : n - 1]$ of linked lists (buckets) and assumes

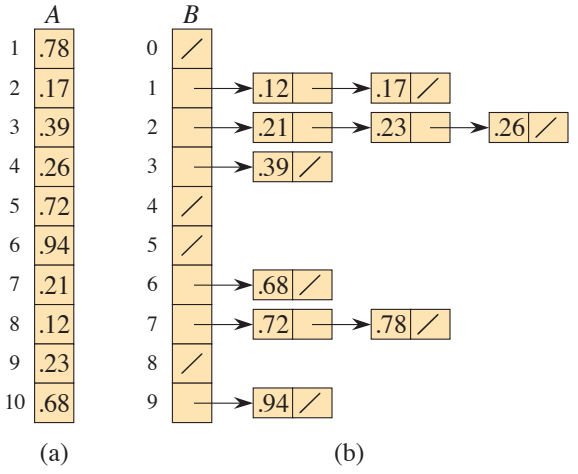


**Figure 8.4** The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1:10]$. **(b)** The array $B[0:9]$ of sorted lists (buckets) after line 7 of the algorithm, with slashes indicating the end of each bucket. Bucket $i$ holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation of the lists $B[0], B[1], \ldots, B[9]$ in order.

that there is a mechanism for maintaining such lists. (Section 10.2 describes how to implement basic operations on linked lists.) Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3       make B[i] an empty list
4  for i = 1 to n
5       insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7       sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

To see that this algorithm works, consider two elements $A[i]$ and $A[j]$. Assume without loss of generality that $A[i] \le A[j]$. Since $\lfloor n \cdot A[i] \rfloor \le \lfloor n \cdot A[j] \rfloor$, either element $A[i]$ goes into the same bucket as $A[j]$ or it goes into a bucket with a lower index. If $A[i]$ and $A[j]$ go into the same bucket, then the **for** loop of lines 6–7 puts them into the proper order. If $A[i]$ and $A[j]$ go into different buckets, then line 8 puts them into the proper order. Therefore, bucket sort works correctly.

To analyze the running time, observe that, together, all lines except line 7 take $O(n)$ time in the worst case. We need to analyze the total time taken by the $n$ calls to insertion sort in line 7.

To analyze the cost of the calls to insertion sort, let $n_i$ be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time (see Section 2.2), the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \; . \tag{8.1}$$

We now analyze the average-case running time of bucket sort, by computing the expected value of the running time, where we take the expectation over the input distribution. Taking expectations of both sides and using linearity of expectation (equation (C.24) on page 1192), we have

$$\begin{aligned} \mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O(n_i^2)\right] \quad \text{(by linearity of expectation)} \\ &= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right) \quad \text{(by equation (C.25) on page 1193)} \; . \end{aligned} \tag{8.2}$$

We claim that

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n \tag{8.3}$$

for $i = 0, 1, \ldots, n-1$. It is no surprise that each bucket $i$ has the same value of $\mathrm{E}[n_i^2]$, since each value in the input array $A$ is equally likely to fall in any bucket.

To prove equation (8.3), view each random variable $n_i$ as the number of successes in $n$ Bernoulli trials (see Section C.4). Success in a trial occurs when an element goes into bucket $B[i]$, with a probability $p = 1/n$ of success and $q = 1 - 1/n$ of failure. A binomial distribution counts $n_i$, the number of successes, in the $n$ trials. By equations (C.41) and (C.44) on pages 1199–1200, we have $\mathrm{E}[n_i] = np = n(1/n) = 1$ and $\mathrm{Var}[n_i] = npq = 1 - 1/n$. Equation (C.31) on page 1194 gives

$$\begin{aligned} \mathrm{E}\left[n_i^2\right] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] \\ &= (1 - 1/n) + 1^2 \\ &= 2 - 1/n \; , \end{aligned}$$

which proves equation (8.3). Using this expected value in equation (8.2), we get that the average-case running time for bucket sort is $\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$.

Even if the input is not drawn from a uniform distribution, bucket sort may still run in linear time. As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, equation (8.1) tells us that bucket sort runs in linear time.

## Exercises

### *8.4-1*

Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

### *8.4-2*

Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

### *8.4-3*

Let $X$ be a random variable that is equal to the number of heads in two flips of a fair coin. What is $E[X^2]$? What is $E^2[X]$?

### *8.4-4*

An array $A$ of size $n > 10$ is filled in the following way. For each element $A[i]$, choose two random variables $x_i$ and $y_i$ uniformly and independently from $[0, 1)$. Then set

$$A[i] = \frac{\lfloor 10x_i \rfloor}{10} + \frac{y_i}{n} .$$

Modify bucket sort so that it sorts the array $A$ in $O(n)$ expected time.

### ★ *8.4-5*

You are given $n$ points in the unit disk, $p_i = (x_i, y_i)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \ldots, n$. Suppose that the points are uniformly distributed, that is, the probability of finding a point in any region of the disk is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the $n$ points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin. (*Hint:* Design the bucket sizes in BUCKET-SORT to reflect the uniform distribution of the points in the unit disk.)

### ★ *8.4-6*

A ***probability distribution function*** $P(x)$ for a random variable $X$ is defined by $P(x) = \Pr\{X \leq x\}$. Suppose that you draw a list of $n$ random variables

$X_1, X_2, \ldots, X_n$ from a continuous probability distribution function $P$ that is computable in $O(1)$ time (given $y$ you can find $x$ such that $P(x) = y$ in $O(1)$ time). Give an algorithm that sorts these numbers in linear average-case time.

## Problems

***8-1 Probabilistic lower bounds on comparison sorting***

In this problem, you will prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on $n$ distinct input elements. You'll begin by examining a deterministic comparison sort $A$ with decision tree $T_A$. Assume that every permutation of $A$'s inputs is equally likely.

***a.*** Suppose that each leaf of $T_A$ is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

***b.*** Let $D(T)$ denote the external path length of a decision tree $T$—the sum of the depths of all the leaves of $T$. Let $T$ be a decision tree with $k > 1$ leaves, and let $LT$ and $RT$ be the left and right subtrees of $T$. Show that $D(T) = D(LT) + D(RT) + k$.

***c.*** Let $d(k)$ be the minimum value of $D(T)$ over all decision trees $T$ with $k > 1$ leaves. Show that $d(k) = \min\{d(i) + d(k-i) + k : 1 \le i \le k-1\}$. (*Hint:* Consider a decision tree $T$ with $k$ leaves that achieves the minimum. Let $i_0$ be the number of leaves in $LT$ and $k - i_0$ the number of leaves in $RT$.)

***d.*** Prove that for a given value of $k > 1$ and $i$ in the range $1 \le i \le k-1$, the function $i \lg i + (k-i)\lg(k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

***e.*** Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the average-case time to sort $n$ elements is $\Omega(n \lg n)$.

Now consider a *randomized* comparison sort $B$. We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form $\text{RANDOM}(1, r)$ made by algorithm $B$. The node has $r$ children, each of which is equally likely to be chosen during an execution of the algorithm.

***f.*** Show that for any randomized comparison sort $B$, there exists a deterministic comparison sort $A$ whose expected number of comparisons is no more than those made by $B$.

### 8-2 Sorting in place in linear time

You have an array of $n$ data records to sort, each with a key of 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

***a.*** Give an algorithm that satisfies criteria 1 and 2 above.

***b.*** Give an algorithm that satisfies criteria 1 and 3 above.

***c.*** Give an algorithm that satisfies criteria 2 and 3 above.

***d.*** Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts $n$ records with $b$-bit keys in $O(bn)$ time? Explain how or why not.

***e.*** Suppose that the $n$ records have keys in the range from 1 to $k$. Show how to modify counting sort so that it sorts the records in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable?

### 8-3 Sorting variable-length items

***a.*** You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is $n$. Show how to sort the array in $O(n)$ time.

***b.*** You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is $n$. Show how to sort the strings in $O(n)$ time. (The desired order is the standard alphabetical order: for example, `a` < `ab` < `b`.)

### 8-4 Water jugs

You are given $n$ red and $n$ blue water jugs, all of different shapes and sizes. All the red jugs hold different amounts of water, as do all the blue jugs, and you cannot tell from the size of a jug how much water it holds. Moreover, for every jug of one color, there is a jug of the other color that holds the same amount of water.

Your task is to group the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair

of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation tells you whether the red jug or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

***a.*** Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

***b.*** Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.

***c.*** Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

### *8-5 Average sorting*

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an $n$-element array $A$ ***k-sorted*** if, for all $i = 1, 2, \ldots, n - k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} .$$

***a.*** What does it mean for an array to be 1-sorted?

***b.*** Give a permutation of the numbers $1, 2, \ldots, 10$ that is 2-sorted, but not sorted.

***c.*** Prove that an $n$-element array is $k$-sorted if and only if $A[i] \leq A[i + k]$ for all $i = 1, 2, \ldots, n - k$.

***d.*** Give an algorithm that $k$-sorts an $n$-element array in $O(n \lg(n/k))$ time.

We can also show a lower bound on the time to produce a $k$-sorted array, when $k$ is a constant.

***e.*** Show how to sort a $k$-sorted array of length $n$ in $O(n \lg k)$ time. (*Hint:* Use the solution to Exercise 6.5-11.)

***f.*** Show that when $k$ is a constant, $k$-sorting an $n$-element array requires $\Omega(n \lg n)$ time. (*Hint:* Use the solution to part (e) along with the lower bound on comparison sorts.)

### 8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, you will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing $n$ items. First, you will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

***a.*** Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with $n$ numbers.

***b.*** Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Now you will show a slightly tighter $2n - 1$ bound.

***c.*** Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

***d.*** Use your answer to part (c) to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

### 8-7 The 0-1 sorting lemma and columnsort

A ***compare-exchange*** operation on two array elements $A[i]$ and $A[j]$, where $i < j$, has the form

```
COMPARE-EXCHANGE(A, i, j)
1  if A[i] > A[j]
2        exchange A[i] with A[j]
```

After the compare-exchange operation, we know that $A[i] \le A[j]$.

An ***oblivious compare-exchange algorithm*** operates solely by a sequence of prespecified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare-exchange operation. For example, the COMPARE-EXCHANGE-INSERTION-SORT procedure on the facing page shows a variation of insertion sort as an oblivious compare-exchange algorithm. (Unlike the INSERTION-SORT procedure on page 19, the oblivious version runs in $\Theta(n^2)$ time in all cases.)

The ***0-1 sorting lemma*** provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

```
COMPARE-EXCHANGE-INSERTION-SORT(A, n)
1  for i = 2 to n
2      for j = i − 1 downto 1
3          COMPARE-EXCHANGE(A, j, j + 1)
```

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange algorithm X fails to correctly sort the array $A[1:n]$. Let $A[p]$ be the smallest value in $A$ that algorithm X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1:n]$ of 0s and 1s as follows:

$$B[i] = \begin{cases} 0 & \text{if } A[i] \le A[p] \,, \\ 1 & \text{if } A[i] > A[p] \,. \end{cases}$$

***a.*** Argue that $A[q] > A[p]$, so that $B[p] = 0$ and $B[q] = 1$.

***b.*** To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort array $B$ correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm, ***columnsort***, works on a rectangular array of $n$ elements. The array has $r$ rows and $s$ columns (so that $n = rs$), subject to three restrictions:

- $r$ must be even,
- $s$ must be a divisor of $r$, and
- $r \ge 2s^2$.

When columnsort completes, the array is sorted in ***column-major order***: reading down each column in turn, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of $n$. The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

1. Sort each column.
2. Transpose the array, but reshape it back to $r$ rows and $s$ columns. In other words, turn the leftmost column into the top $r/s$ rows, in order; turn the next column into the next $r/s$ rows, in order; and so on.

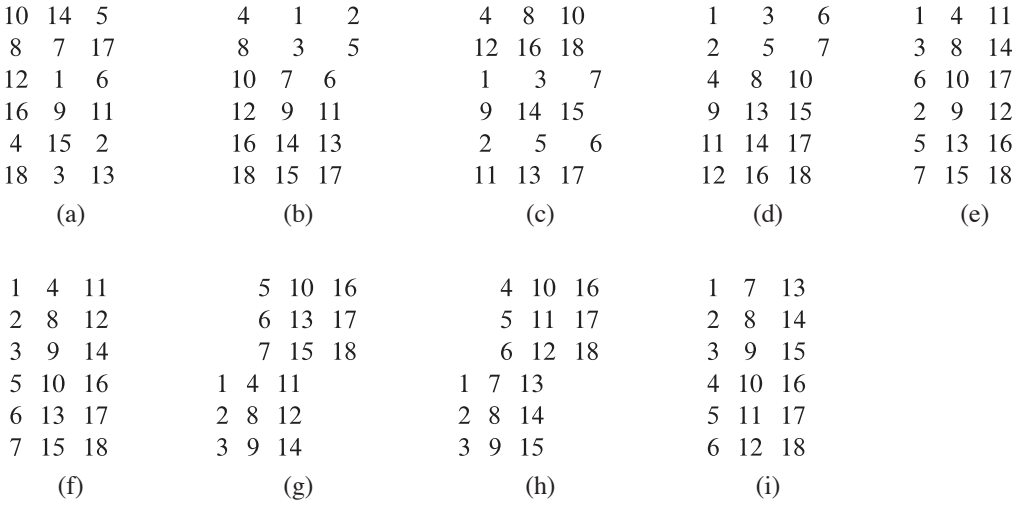


**Figure 8.5** The steps of columnsort. **(a)** The input array with 6 rows and 3 columns. (This example does not obey the $r \geq 2s^2$ requirement, but it works.) **(b)** After sorting each column in step 1. **(c)** After transposing and reshaping in step 2. **(d)** After sorting each column in step 3. **(e)** After performing step 4, which inverts the permutation from step 2. **(f)** After sorting each column in step 5. **(g)** After shifting by half a column in step 6. **(h)** After sorting each column in step 7. **(i)** After performing step 8, which inverts the permutation from step 6. Steps 6–8 sort the bottom half of each column with the top half of the next column. After step 8, the array is sorted in column-major order.

3. Sort each column.
4. Perform the inverse of the permutation performed in step 2.
5. Sort each column.
6. Shift the top half of each column into the bottom half of the same column, and shift the bottom half of each column into the top half of the next column to the right. Leave the top half of the leftmost column empty. Shift the bottom half of the last column into the top half of a new rightmost column, and leave the bottom half of this new column empty.
7. Sort each column.
8. Perform the inverse of the permutation performed in step 6.

You can think of steps 6–8 as a single step that sorts the bottom half of each column and the top half of the next column. Figure 8.5 shows an example of the steps of columnsort with $r = 6$ and $s = 3$. (Even though this example violates the requirement that $r \geq 2s^2$, it happens to work.)

***c.*** Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.

Although it might seem hard to believe that columnsort actually sorts, you will use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies because we can treat columnsort as an oblivious compare-exchange algorithm. A couple of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is ***clean*** if we know that it contains either all 0s or all 1s or if it is empty. Otherwise, the area might contain mixed 0s and 1s, and it is ***dirty***. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with $r$ rows and $s$ columns.

***d.*** Prove that after steps 1–3, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most $s$ dirty rows between them. (One of the clean rows could be empty.)

***e.*** Prove that after step 4, the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most $s^2$ elements in the middle. (Again, one of the clean areas could be empty.)

***f.*** Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

***g.*** Now suppose that $s$ does not divide $r$. Prove that after steps 1–3, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. (Once again, one of the clean areas could be empty.) How large must $r$ be, compared with $s$, for columnsort to correctly sort when $s$ does not divide $r$?

***h.*** Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when $s$ does not divide $r$, and prove that with your change, columnsort correctly sorts.

## Chapter notes

The decision-tree model for studying comparison sorts was introduced by Ford and Johnson [150]. Knuth's comprehensive treatise on sorting [261] covers many variations on the sorting problem, including the information-theoretic lower bound on the complexity of sorting given here. Ben-Or [46] studied lower bounds for sorting using generalizations of the decision-tree model.

Knuth credits H. H. Seward with inventing counting sort in 1954, as well as with the idea of combining counting sort with radix sort. Radix sorting starting with the least significant digit appears to be a folk algorithm widely used by operators of

mechanical card-sorting machines. According to Knuth, the first published reference to the method is a 1929 document by L. J. Comrie describing punched-card equipment. Bucket sorting has been in use since 1956, when the basic idea was proposed by Isaac and Singleton [235].

Munro and Raman [338] give a stable sorting algorithm that performs $O(n^{1+\epsilon})$ comparisons in the worst case, where $0 < \epsilon \leq 1$ is any fixed constant. Although any of the $O(n \lg n)$-time algorithms make fewer comparisons, the algorithm by Munro and Raman moves data only $O(n)$ times and operates in place.

The case of sorting $n$ $b$-bit integers in $o(n \lg n)$ time has been considered by many researchers. Several positive results have been obtained, each under slightly different assumptions about the model of computation and the restrictions placed on the algorithm. All the results assume that the computer memory is divided into addressable $b$-bit words. Fredman and Willard [157] introduced the fusion tree data structure and used it to sort $n$ integers in $O(n \lg n / \lg \lg n)$ time. This bound was later improved to $O(n \sqrt{\lg n})$ time by Andersson [17]. These algorithms require the use of multiplication and several precomputed constants. Andersson, Hagerup, Nilsson, and Raman [18] have shown how to sort $n$ integers in $O(n \lg \lg n)$ time without using multiplication, but their method requires storage that can be unbounded in terms of $n$. Using multiplicative hashing, we can reduce the storage needed to $O(n)$, but then the $O(n \lg \lg n)$ worst-case bound on the running time becomes an expected-time bound. Generalizing the exponential search trees of Andersson [17], Thorup [434] gave an $O(n(\lg \lg n)^2)$-time sorting algorithm that does not use multiplication or randomization, and it uses linear space. Combining these techniques with some new ideas, Han [207] improved the bound for sorting to $O(n \lg \lg n \lg \lg \lg n)$ time. Although these algorithms are important theoretical breakthroughs, they are all fairly complicated and at the present time seem unlikely to compete with existing sorting algorithms in practice.

The columnsort algorithm in Problem 8-7 is by Leighton [286].

# 9 Medians and Order Statistics

The $i$th ***order statistic*** of a set of $n$ elements is the $i$th smallest element. For example, the ***minimum*** of a set of elements is the first order statistic ($i = 1$), and the ***maximum*** is the $n$th order statistic ($i = n$). A ***median***, informally, is the "halfway point" of the set. When $n$ is odd, the median is unique, occurring at $i = (n+1)/2$. When $n$ is even, there are two medians, the ***lower median*** occurring at $i = n/2$ and the ***upper median*** occurring at $i = n/2 + 1$. Thus, regardless of the parity of $n$, medians occur at $i = \lfloor (n+1)/2 \rfloor$ and $i = \lceil (n+1)/2 \rceil$. For simplicity in this text, however, we consistently use the phrase "the median" to refer to the lower median.

This chapter addresses the problem of selecting the $i$th order statistic from a set of $n$ distinct numbers. We assume for convenience that the set contains distinct numbers, although virtually everything that we do extends to the situation in which a set contains repeated values. We formally specify the ***selection problem*** as follows:

**Input:** A set $A$ of $n$ distinct numbers[1] and an integer $i$, with $1 \le i \le n$.

**Output:** The element $x \in A$ that is larger than exactly $i - 1$ other elements of $A$.

We can solve the selection problem in $O(n \lg n)$ time simply by sorting the numbers using heapsort or merge sort and then outputting the $i$th element in the sorted array. This chapter presents asymptotically faster algorithms.

Section 9.1 examines the problem of selecting the minimum and maximum of a set of elements. More interesting is the general selection problem, which we investigate in the subsequent two sections. Section 9.2 analyzes a practical randomized algorithm that achieves an $O(n)$ expected running time, assuming dis-

[1] As in the footnote on page 182, you can enforce the assumption that the numbers are distinct by converting each input value $A[i]$ to an ordered pair $(A[i], i)$ with $(A[i], i) < (A[j], j)$ if either $A[i] < A[j]$ or $A[i] = A[j]$ and $i < j$.

tinct elements. Section 9.3 contains an algorithm of more theoretical interest that achieves the $O(n)$ running time in the worst case.

## 9.1 Minimum and maximum

How many comparisons are necessary to determine the minimum of a set of $n$ elements? To obtain an upper bound of $n - 1$ comparisons, just examine each element of the set in turn and keep track of the smallest element seen so far. The MINIMUM procedure assumes that the set resides in array $A[1:n]$.

```
MINIMUM(A, n)
1  min = A[1]
2  for i = 2 to n
3      if min > A[i]
4          min = A[i]
5  return min
```

It's no more difficult to find the maximum with $n - 1$ comparisons.

Is this algorithm for minimum the best we can do? Yes, because it turns out that there's a lower bound of $n - 1$ comparisons for the problem of determining the minimum. Think of any algorithm that determines the minimum as a tournament among the elements. Each comparison is a match in the tournament in which the smaller of the two elements wins. Since every element except the winner must lose at least one match, we can conclude that $n - 1$ comparisons are necessary to determine the minimum. Hence the algorithm MINIMUM is optimal with respect to the number of comparisons performed.

### Simultaneous minimum and maximum

Some applications need to find both the minimum and the maximum of a set of $n$ elements. For example, a graphics program may need to scale a set of $(x, y)$ data to fit onto a rectangular display screen or other graphical output device. To do so, the program must first determine the minimum and maximum value of each coordinate.

Of course, we can determine both the minimum and the maximum of $n$ elements using $\Theta(n)$ comparisons. We simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2 = \Theta(n)$ comparisons.

Although $2n-2$ comparisons is asymptotically optimal, it is possible to improve the leading constant. We can find both the minimum and the maximum using at most $3\lfloor n/2 \rfloor$ comparisons. The trick is to maintain both the minimum and maximum elements seen thus far. Rather than processing each element of the input by comparing it against the current minimum and maximum, at a cost of 2 comparisons per element, process elements in pairs. Compare pairs of elements from the input first *with each other*, and then compare the smaller with the current minimum and the larger to the current maximum, at a cost of 3 comparisons for every 2 elements.

How you set up initial values for the current minimum and maximum depends on whether $n$ is odd or even. If $n$ is odd, set both the minimum and maximum to the value of the first element, and then process the rest of the elements in pairs. If $n$ is even, perform 1 comparison on the first 2 elements to determine the initial values of the minimum and maximum, and then process the rest of the elements in pairs as in the case for odd $n$.

Let's count the total number of comparisons. If $n$ is odd, then $3\lfloor n/2 \rfloor$ comparisons occur. If $n$ is even, 1 initial comparison occurs, followed by another $3(n-2)/2$ comparisons, for a total of $3n/2-2$. Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$.

## Exercises

### *9.1-1*

Show that the second smallest of $n$ elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

### *9.1-2*

Given $n > 2$ distinct numbers, you want to find a number that is neither the minimum nor the maximum. What is the smallest number of comparisons that you need to perform?

### *9.1-3*

A racetrack can run races with five horses at a time to determine their relative speeds. For 25 horses, it takes six races to determine the fastest horse, assuming transitivity (see page 1159). What's the minimum number of races it takes to determine the fastest three horses out of 25?

### ★ *9.1-4*

Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of $n$ numbers. (*Hint:* Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects these counts.)

## 9.2 Selection in expected linear time

The general selection problem—finding the $i$th order statistic for any value of $i$—appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: $\Theta(n)$. This section presents a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 7. Like quicksort it partitions the input array recursively. But unlike quicksort, which recursively processes both sides of the partition, RANDOMIZED-SELECT works on only one side of the partition. This difference shows up in the analysis: whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected running time of RANDOMIZED-SELECT is $\Theta(n)$, assuming that the elements are distinct.

RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION introduced in Section 7.3. Like RANDOMIZED-QUICKSORT, it is a randomized algorithm, since its behavior is determined in part by the output of a random-number generator. The RANDOMIZED-SELECT procedure returns the $i$th smallest element of the array $A[p:r]$, where $1 \le i \le r - p + 1$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6       return A[q]        // the pivot value is the answer
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Figure 9.1 illustrates how the RANDOMIZED-SELECT procedure works. Line 1 checks for the base case of the recursion, in which the subarray $A[p:r]$ consists of just one element. In this case, $i$ must equal 1, and line 2 simply returns $A[p]$ as the $i$th smallest element. Otherwise, the call to RANDOMIZED-PARTITION in line 3 partitions the array $A[p:r]$ into two (possibly empty) subarrays $A[p:q-1]$ and $A[q+1:r]$ such that each element of $A[p:q-1]$ is less than or equal to $A[q]$, which in turn is less than each element of $A[q+1:r]$. (Although our analysis assumes that the elements are distinct, the procedure still yields the correct result even if equal elements are present.) As in quicksort, we'll refer to $A[q]$ as the ***pivot*** element. Line 4 computes the number $k$ of elements in the subarray $A[p:q]$, that is,

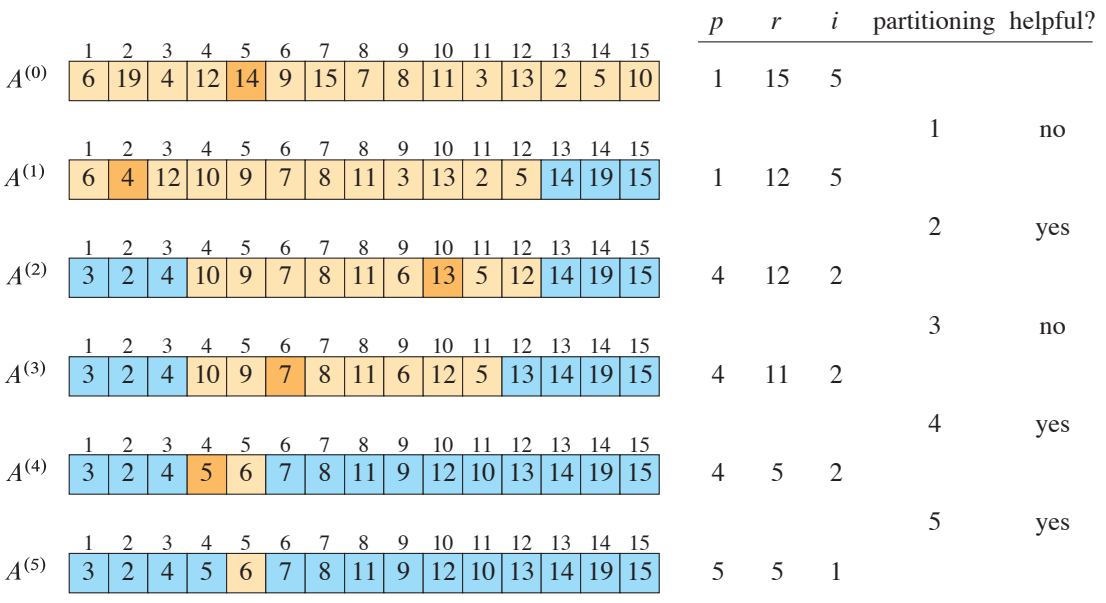


**Figure 9.1** The action of RANDOMIZED-SELECT as successive partitionings narrow the subarray $A[p:r]$, showing the values of the parameters $p$, $r$, and $i$ at each recursive call. The subarray $A[p:r]$ in each recursive step is shown in tan, with the dark tan element selected as the pivot for the next partitioning. Blue elements are outside $A[p:r]$. The answer is the tan element in the bottom array, where $p = r = 5$ and $i = 1$. The array designations $A^{(0)}, A^{(1)}, \ldots, A^{(5)}$, the partitioning numbers, and whether the partitioning is helpful are explained on the following page.

the number of elements in the low side of the partition, plus 1 for the pivot element. Line 5 then checks whether $A[q]$ is the $i$th smallest element. If it is, then line 6 returns $A[q]$. Otherwise, the algorithm determines in which of the two subarrays $A[p:q-1]$ and $A[q+1:r]$ the $i$th smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and line 8 recursively selects it from the subarray. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know $k$ values that are smaller than the $i$th smallest element of $A[p:r]$—namely, the elements of $A[p:q]$—the desired element is the $(i-k)$th smallest element of $A[q+1:r]$, which line 9 finds recursively. The code appears to allow recursive calls to subarrays with 0 elements, but Exercise 9.2-1 asks you to show that this situation cannot happen.

The worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum, because it could be extremely unlucky and always partition around the largest remaining element before identifying the $i$th smallest when only one element remains. In this worst case, each recursive step removes only the pivot from consideration. Because partitioning $n$ elements takes $\Theta(n)$ time, the recurrence for the worst-case running time is the same as for QUICKSORT:

$T(n) = T(n-1) + \Theta(n)$, with the solution $T(n) = \Theta(n^2)$. We'll see that the algorithm has a linear expected running time, however, and because it is randomized, no particular input elicits the worst-case behavior.

To see the intuition behind the linear expected running time, suppose that each time the algorithm randomly selects a pivot element, the pivot lies somewhere within the second and third quartiles—the "middle half"—of the remaining elements in sorted order. If the $i$th smallest element is less than the pivot, then all the elements greater than the pivot are ignored in all future recursive calls. These ignored elements include at least the uppermost quartile, and possibly more. Likewise, if the $i$th smallest element is greater than the pivot, then all the elements less than the pivot—at least the first quartile—are ignored in all future recursive calls. Either way, therefore, at least $1/4$ of the remaining elements are ignored in all future recursive calls, leaving at most $3/4$ of the remaining elements ***in play***: residing in the subarray $A[p:r]$. Since RANDOMIZED-PARTITION takes $\Theta(n)$ time on a subarray of $n$ elements, the recurrence for the worst-case running time is $T(n) = T(3n/4) + \Theta(n)$. By case 3 of the master method (Theorem 4.1 on page 102), this recurrence has solution $T(n) = \Theta(n)$.

Of course, the pivot does not necessarily fall into the middle half every time. Since the pivot is selected at random, the probability that it falls into the middle half is about $1/2$ each time. We can view the process of selecting the pivot as a Bernoulli trial (see Section C.4) with success equating to the pivot residing in the middle half. Thus the expected number of trials needed for success is given by a geometric distribution: just two trials on average (equation (C.36) on page 1197). In other words, we expect that half of the partitionings reduce the number of elements still in play by at least $3/4$ and that half of the partitionings do not help as much. Consequently, the expected number of partitionings at most doubles from the case when the pivot always falls into the middle half. The cost of each extra partitioning is less than the one that preceded it, so that the expected running time is still $\Theta(n)$.

To make the above argument rigorous, we start by defining the random variable $A^{(j)}$ as the set of elements of $A$ that are still in play after $j$ partitionings (that is, within the subarray $A[p:r]$ after $j$ calls of RANDOMIZED-SELECT), so that $A^{(0)}$ consists of all the elements in $A$. Since each partitioning removes at least one element—the pivot—from being in play, the sequence $|A^{(0)}|, |A^{(1)}|, |A^{(2)}|, \ldots$ strictly decreases. Set $A^{(j-1)}$ is in play before the $j$th partitioning, and set $A^{(j)}$ remains in play afterward. For convenience, assume that the initial set $A^{(0)}$ is the result of a 0th "dummy" partitioning.

Let's call the $j$th partitioning ***helpful*** if $|A^{(j)}| \le (3/4)|A^{(j-1)}|$. Figure 9.1 shows the sets $A^{(j)}$ and whether partitionings are helpful for an example array. A helpful partitioning corresponds to a successful Bernoulli trial. The following lemma shows that a partitioning is at least as likely to be helpful as not.

***Lemma 9.1***
A partitioning is helpful with probability at least $1/2$.

***Proof*** Whether a partitioning is helpful depends on the randomly chosen pivot. We discussed the "middle half" in the informal argument above. Let's more precisely define the middle half of an $n$-element subarray as all but the smallest $\lceil n/4 \rceil - 1$ and greatest $\lceil n/4 \rceil - 1$ elements (that is, all but the first $\lceil n/4 \rceil - 1$ and last $\lceil n/4 \rceil - 1$ elements if the subarray were sorted). We'll prove that if the pivot falls into the middle half, then the pivot leads to a helpful partitioning, and we'll also prove that the probability of the pivot falling into the middle half is at least $1/2$.

Regardless of where the pivot falls, either all the elements greater than it or all the elements less than it, along with the pivot itself, will no longer be in play after partitioning. If the pivot falls into the middle half, therefore, at least $\lceil n/4 \rceil - 1$ elements less than the pivot or $\lceil n/4 \rceil - 1$ elements greater than the pivot, plus the pivot, will no longer be in play after partitioning. That is, at least $\lceil n/4 \rceil$ elements will no longer be in play. The number of elements remaining in play will be at most $n - \lceil n/4 \rceil$, which equals $\lfloor 3n/4 \rfloor$ by Exercise 3.3-2 on page 70. Since $\lfloor 3n/4 \rfloor \le 3n/4$, the partitioning is helpful.

To determine a lower bound on the probability that a randomly chosen pivot falls into the middle half, we determine an upper bound on the probability that it does not. That probability is

$$\begin{aligned}
\frac{2(\lceil n/4 \rceil - 1)}{n} &\le \frac{2((n/4 + 1) - 1)}{n} \quad \text{(by inequality (3.2) on page 64)} \\
&= \frac{n/2}{n} \\
&= 1/2 \, .
\end{aligned}$$

Thus, the pivot has a probability of at least $1/2$ of falling into the middle half, and so the probability is at least $1/2$ that a partitioning is helpful. ■

We can now bound the expected running time of RANDOMIZED-SELECT.

***Theorem 9.2***
The procedure RANDOMIZED-SELECT on an input array of $n$ distinct elements has an expected running time of $\Theta(n)$.

***Proof*** Since not every partitioning is necessarily helpful, let's give each partitioning an index starting at 0 and denote by $\langle h_0, h_1, h_2, \ldots, h_m \rangle$ the sequence of partitionings that are helpful, so that the $h_k$th partitioning is helpful for $k = 0, 1, 2, \ldots, m$. Although the number $m$ of helpful partitionings is a random vari-

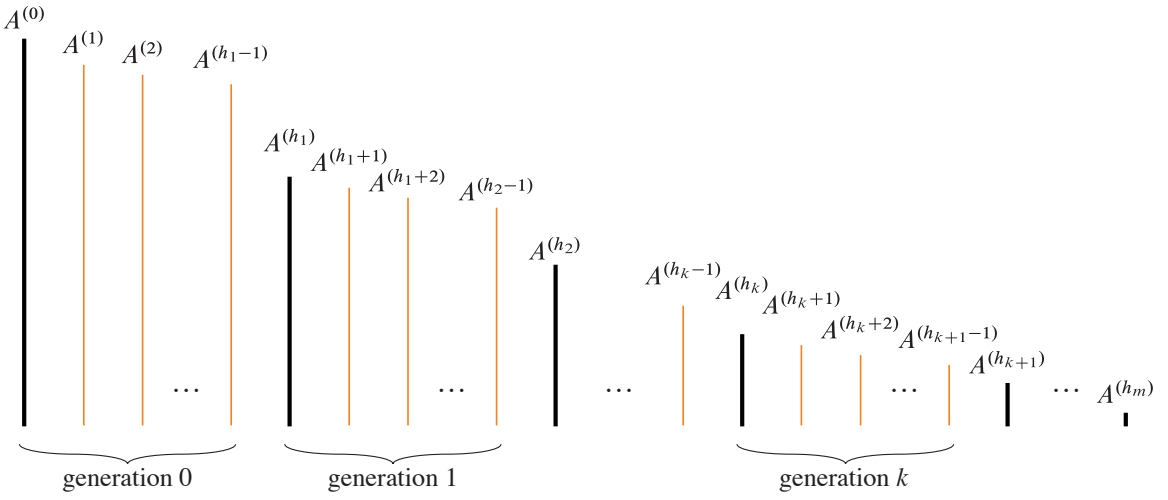


**Figure 9.2** The sets within each generation in the proof of Theorem 9.2. Vertical lines represent the sets, with the height of each line indicating the size of the set, which equals the number of elements in play. Each generation starts with a set $A^{(h_k)}$, which is the result of a helpful partitioning. These sets are drawn in black and are at most 3/4 the size of the sets to their immediate left. Sets drawn in orange are not the first within a generation. A generation may contain just one set. The sets in generation $k$ are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_{k+1}-1)}$. The sets $A^{(h_k)}$ are defined so that $|A^{(h_k)}| \le (3/4)|A^{(h_k-1)}|$. If the partitioning gets all the way to generation $h_m$, set $A^{(h_m)}$ has at most one element in play.

able, we can bound it, since after at most $\lceil \log_{4/3} n \rceil$ helpful partitionings, only one element remains in play. Consider the dummy 0th partitioning as helpful, so that $h_0 = 0$. Denote $|A^{(h_k)}|$ by $n_k$, where $n_0 = |A^{(0)}|$ is the original problem size. Since the $h_k$th partitioning is helpful and the sizes of the sets $A^{(j)}$ strictly decrease, we have $n_k = |A^{(h_k)}| \le (3/4)|A^{(h_k-1)}| = (3/4)\,n_{k-1}$ for $k = 1, 2, \ldots, m$. By iterating $n_k \le (3/4)\,n_{k-1}$, we have that $n_k \le (3/4)^k n_0$ for $k = 0, 1, 2, \ldots, m$.

As Figure 9.2 depicts, we break up the sequence of sets $A^{(j)}$ into $m$ ***generations*** consisting of consecutively partitioned sets, starting with the result $A^{(h_k)}$ of a helpful partitioning and ending with the last set $A^{(h_{k+1}-1)}$ before the next helpful partitioning, so that the sets in generation $k$ are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_{k+1}-1)}$. Then for each set of elements $A^{(j)}$in the $k$th generation, we have that $|A^{(j)}| \le |A^{(h_k)}| = n_k \le (3/4)^k n_0$.

Next, we define the random variable

$$X_k = h_{k+1} - h_k$$

for $k = 0, 1, 2, \ldots, m-1$. That is, $X_k$ is the number of sets in the $k$th generation, so that the sets in the $k$th generation are $A^{(h_k)}, A^{(h_k+1)}, \ldots, A^{(h_k+X_k-1)}$.

By Lemma 9.1, the probability that a partitioning is helpful is at least 1/2. The probability is actually even higher, since a partitioning is helpful even if the pivot

does not fall into the middle half but the $i$th smallest element happens to lie in the smaller side of the partitioning. We'll just use the lower bound of $1/2$, however, and then equation (C.36) gives that $\mathrm{E}[X_k] \le 2$ for $k = 0, 1, 2, \ldots, m-1$.

Let's derive an upper bound on how many comparisons are made altogether during partitioning, since the running time is dominated by the comparisons. Since we are calculating an upper bound, assume that the recursion goes all the way until only one element remains in play. The $j$th partitioning takes the set $A^{(j-1)}$ of elements in play, and it compares the randomly chosen pivot with all the other $|A^{(j-1)}| - 1$ elements, so that the $j$th partitioning makes fewer than $|A^{(j-1)}|$ comparisons. The sets in the $k$th generation have sizes $|A^{(h_k)}|, |A^{(h_k+1)}|, \ldots, |A^{(h_k+X_k-1)}|$. Thus, the total number of comparisons during partitioning is less than

$$\begin{aligned}
\sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(j)}| &\le \sum_{k=0}^{m-1} \sum_{j=h_k}^{h_k+X_k-1} |A^{(h_k)}| \\
&= \sum_{k=0}^{m-1} X_k \, |A^{(h_k)}| \\
&\le \sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0 \, .
\end{aligned}$$

Since $\mathrm{E}[X_k] \le 2$, we have that the expected total number of comparisons during partitioning is less than

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=0}^{m-1} X_k \left(\frac{3}{4}\right)^k n_0\right] &= \sum_{k=0}^{m-1} \mathrm{E}\left[X_k \left(\frac{3}{4}\right)^k n_0\right] \quad \text{(by linearity of expectation)} \\
&= n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \mathrm{E}[X_k] \\
&\le 2n_0 \sum_{k=0}^{m-1} \left(\frac{3}{4}\right)^k \\
&< 2n_0 \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\
&= 8n_0 \qquad \text{(by equation (A.7) on page 1142)} \, .
\end{aligned}$$

Since $n_0$ is the size of the original array $A$, we conclude that the expected number of comparisons, and thus the expected running time, for RANDOMIZED-SELECT is $O(n)$. All $n$ elements are examined in the first call of RANDOMIZED-

PARTITION, giving a lower bound of $\Omega(n)$. Hence the expected running time is $\Theta(n)$. ■

### Exercises

***9.2-1***
Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

***9.2-2***
Write an iterative version of RANDOMIZED-SELECT.

***9.2-3***
Suppose that RANDOMIZED-SELECT is used to select the minimum element of the array $A = \langle 2, 3, 0, 5, 7, 9, 1, 8, 6, 4 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

***9.2-4***
Argue that the expected running time of RANDOMIZED-SELECT does not depend on the order of the elements in its input array $A[p:r]$. That is, the expected running time is the same for any permutation of the input array $A[p:r]$. (*Hint:* Argue by induction on the length $n$ of the input array.)

## 9.3 Selection in worst-case linear time

We'll now examine a remarkable and theoretically interesting selection algorithm whose running time is $\Theta(n)$ in the worst case. Although the RANDOMIZED-SELECT algorithm from Section 9.2 achieves linear expected time, we saw that its running time in the worst case was quadratic. The selection algorithm presented in this section achieves linear time in the worst case, but it is not nearly as practical as RANDOMIZED-SELECT. It is mostly of theoretical interest.

Like the expected linear-time RANDOMIZED-SELECT, the worst-case linear-time algorithm SELECT finds the desired element by recursively partitioning the input array. Unlike RANDOMIZED-SELECT, however, SELECT *guarantees* a good split by choosing a provably good pivot when partitioning the array. The cleverness in the algorithm is that it finds the pivot recursively. Thus, there are two invocations of SELECT: one to find a good pivot, and a second to recursively find the desired order statistic.

The partitioning algorithm used by SELECT is like the deterministic partitioning algorithm PARTITION from quicksort (see Section 7.1), but modified to take the element to partition around as an additional input parameter. Like PARTITION, the

PARTITION-AROUND algorithm returns the index of the pivot. Since it's so similar to PARTITION, the pseudocode for PARTITION-AROUND is omitted.

The SELECT procedure takes as input a subarray $A[p:r]$ of $n = r - p + 1$ elements and an integer $i$ in the range $1 \le i \le n$. It returns the $i$th smallest element of $A$. The pseudocode is actually more understandable than it might appear at first.

```
SELECT(A, p, r, i)
 1  while (r − p + 1) mod 5 ≠ 0
 2      for j = p + 1 to r                    // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p:r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1:r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                         // number of 5-element groups
12  for j = p to p + g − 1                    // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p:r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)          // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                           // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

The pseudocode starts by executing the **while** loop in lines 1–10 to reduce the number $r - p + 1$ of elements in the subarray until it is divisible by 5. The **while** loop executes 0 to 4 times, each time rearranging the elements of $A[p:r]$ so that $A[p]$ contains the minimum element. If $i = 1$, which means that we actually want the minimum element, then the procedure simply returns it in line 7. Otherwise, SELECT eliminates the minimum from the subarray $A[p:r]$ and iterates to find the $(i-1)$st element in $A[p+1:r]$. Lines 9–10 do so by incrementing $p$ and decrementing $i$. If the **while** loop completes all of its iterations without returning a

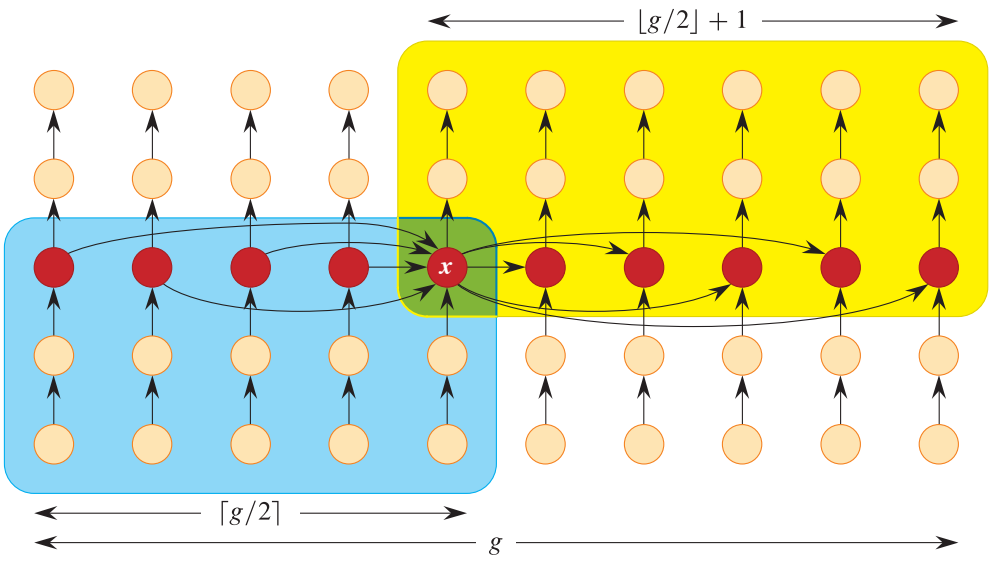


**Figure 9.3** The relationships between elements (shown as circles) immediately after line 17 of the selection algorithm SELECT. There are $g = (r - p + 1)/5$ groups of 5 elements, each of which occupies a column. For example, the leftmost column contains elements $A[p]$, $A[p + g]$, $A[p + 2g]$, $A[p + 3g]$, $A[p + 4g]$, and the next column contains $A[p + 1]$, $A[p + g + 1]$, $A[p + 2g + 1]$, $A[p + 3g + 1]$, $A[p + 4g + 1]$. The medians of the groups are red, and the pivot $x$ is labeled. Arrows go from smaller elements to larger. The elements on the blue background are all known to be less than or equal to $x$ and cannot fall into the high side of the partition around $x$. The elements on the yellow background are known to be greater than or equal to $x$ and cannot fall into the low side of the partition around $x$. The pivot $x$ belongs to both the blue and yellow regions and is shown on a green background. The elements on the white background could lie on either side of the partition.

result, the procedure executes the core of the algorithm in lines 11–24, assured that the number $r - p + 1$ of elements in $A[p : r]$ is evenly divisible by 5.

The next part of the algorithm implements the following idea, illustrated in Figure 9.3. Divide the elements in $A[p : r]$ into $g = (r-p+1)/5$ groups of 5 elements each. The first 5-element group is

$$\langle A[p], A[p + g], A[p + 2g], A[p + 3g], A[p + 4g]\rangle ,$$

the second is

$$\langle A[p + 1], A[p + g + 1], A[p + 2g + 1], A[p + 3g + 1], A[p + 4g + 1]\rangle ,$$

and so forth until the last, which is

$$\langle A[p + g - 1], A[p + 2g - 1], A[p + 3g - 1], A[p + 4g - 1], A[r]\rangle .$$

(Note that $r = p + 5g - 1$.) Line 13 puts each group in order using, for example, insertion sort (Section 2.1), so that for $j = p, p + 1, \ldots, p + g - 1$, we have

$$A[j] \le A[j+g] \le A[j+2g] \le A[j+3g] \le A[j+4g] \,.$$

Each vertical column in Figure 9.3 depicts a sorted group of 5 elements. The median of each 5-element group is $A[j+2g]$, and thus all the 5-element medians, shown in red, lie in the range $A[p+2g : p+3g-1]$.

Next, line 16 determines the pivot $x$ by recursively calling SELECT to find the median (specifically, the $\lceil g/2 \rceil$th smallest) of the $g$ group medians. Line 17 uses the modified PARTITION-AROUND algorithm to partition the elements of $A[p:r]$ around $x$, returning the index $q$ of $x$, so that $A[q] = x$, elements in $A[p:q]$ are all at most $x$, and elements in $A[q:r]$ are greater than or equal to $x$.

The remainder of the code mirrors that of RANDOMIZED-SELECT. If the pivot $x$ is the $i$th largest, the procedure returns it. Otherwise, the procedure recursively calls itself on either $A[p:q-1]$ or $A[q+1:r]$, depending on the value of $i$.

Let's analyze the running time of SELECT and see how the judicious choice of the pivot $x$ plays into a guarantee on its worst-case running time.

***Theorem 9.3***
The running time of SELECT on an input of $n$ elements is $\Theta(n)$.

***Proof*** Define $T(n)$ as the worst-case time to run SELECT on any input subarray $A[p:r]$ of size at most $n$, that is, for which $r - p + 1 \le n$. By this definition, $T(n)$ is monotonically increasing.

We first determine an upper bound on the time spent outside the recursive calls in lines 16, 23, and 24. The **while** loop in lines 1–10 executes 0 to 4 times, which is $O(1)$ times. Since the dominant time within the loop is the computation of the minimum in lines 2–4, which takes $\Theta(n)$ time, lines 1–10 execute in $O(1) \cdot \Theta(n) = O(n)$ time. The sorting of the 5-element groups in lines 12–13 takes $\Theta(n)$ time because each 5-element group takes $\Theta(1)$ time to sort (even using an asymptotically inefficient sorting algorithm such as insertion sort), and there are $g$ elements to sort, where $n/5 - 1 < g \le n/5$. Finally, the time to partition in line 17 is $\Theta(n)$, as Exercise 7.1-3 on page 187 asks you to show. Because the remaining bookkeeping only costs $\Theta(1)$ time, the total amount of time spent outside of the recursive calls is $O(n) + \Theta(n) + \Theta(n) + \Theta(1) = \Theta(n)$.

Now let's determine the running time for the recursive calls. The recursive call to find the pivot in line 16 takes $T(g) \le T(n/5)$ time, since $g \le n/5$ and $T(n)$ monotonically increases. Of the two recursive calls in lines 23 and 24, at most one is executed. But we'll see that no matter which of these two recursive calls to SELECT actually executes, the number of elements in the recursive call turns out to be at most $7n/10$, and hence the worst-case cost for lines 23 and 24 is at most $T(7n/10)$. Let's now show that the machinations with group medians and the choice of the pivot $x$ as the median of the group medians guarantees this property.

Figure 9.3 helps to visualize what's going on. There are $g \le n/5$ groups of 5 elements, with each group shown as a column sorted from bottom to top. The arrows show the ordering of elements within the columns. The columns are ordered from left to right with groups to the left of $x$'s group having a group median less than $x$ and those to the right of $x$'s group having a group median greater than $x$. Although the relative order within each group matters, the relative order among groups to the left of $x$'s column doesn't really matter, and neither does the relative order among groups to the right of $x$'s column. The important thing is that the groups to the left have group medians less than $x$ (shown by the horizontal arrows entering $x$), and that the groups to the right have group medians greater than $x$ (shown by the horizontal arrows leaving $x$). Thus, the yellow region contains elements that we know are greater than or equal to $x$, and the blue region contains elements that we know are less than or equal to $x$.

These two regions each contain at least $3g/2$ elements. The number of group medians in the yellow region is $\lfloor g/2 \rfloor + 1$, and for each group median, two additional elements are greater than it, making a total of $3(\lfloor g/2 \rfloor + 1) \ge 3g/2$ elements. Similarly, the number of group medians in the blue region is $\lceil g/2 \rceil$, and for each group median, two additional elements are less than it, making a total of $3\lceil g/2 \rceil \ge 3g/2$.

The elements in the yellow region cannot fall into the low side of the partition around $x$, and those in the blue region cannot fall into the high side. The elements in neither region—those lying on a white background—could fall into either side of the partition. But since the low side of the partition excludes the elements in the yellow region, and there are a total of $5g$ elements, we know that the low side of the partition can contain at most $5g - 3g/2 = 7g/2 \le 7n/10$ elements. Likewise, the high side of the partition excludes the elements in the blue region, and a similar calculation shows that it also contains at most $7n/10$ elements.

All of which leads to the following recurrence for the worst-case running time of SELECT:

$$T(n) \le T(n/5) + T(7n/10) + \Theta(n) \; . \tag{9.1}$$

We can show that $T(n) = O(n)$ by substitution.[2] More specifically, we'll prove that $T(n) \le cn$ for some suitably large constant $c > 0$ and all $n > 0$. Substituting this inductive hypothesis into the right-hand side of recurrence (9.1) and assuming that $n \ge 5$ yields

---

[2] We could also use the Akra-Bazzi method from Section 4.7, which involves calculus, to solve this recurrence. Indeed, a similar recurrence (4.24) on page 117 was used to illustrate that method.

$$\begin{aligned} T(n) &\le c(n/5) + c(7n/10) + \Theta(n) \\ &\le 9cn/10 + \Theta(n) \\ &= cn - cn/10 + \Theta(n) \\ &\le cn \end{aligned}$$

if $c$ is chosen large enough that $c/10$ dominates the upper-bound constant hidden by the $\Theta(n)$. In addition to this constraint, we can pick $c$ large enough that $T(n) \le cn$ for all $n \le 4$, which is the base case of the recursion within SELECT. The running time of SELECT is therefore $O(n)$ in the worst case, and because line 13 alone takes $\Theta(n)$ time, the total time is $\Theta(n)$. ■

As in a comparison sort (see Section 8.1), SELECT and RANDOMIZED-SELECT determine information about the relative order of elements only by comparing elements. Recall from Chapter 8 that sorting requires $\Omega(n \lg n)$ time in the comparison model, even on average (see Problem 8-1). The linear-time sorting algorithms in Chapter 8 make assumptions about the type of the input. In contrast, the linear-time selection algorithms in this chapter do not require any assumptions about the input's type, only that the elements are distinct and can be pairwise compared according to a linear order. The algorithms in this chapter are not subject to the $\Omega(n \lg n)$ lower bound, because they manage to solve the selection problem without sorting all the elements. Thus, solving the selection problem by sorting and indexing, as presented in the introduction to this chapter, is asymptotically inefficient in the comparison model.

## Exercises

### *9.3-1*

In the algorithm SELECT, the input elements are divided into groups of 5. Show that the algorithm works in linear time if the input elements are divided into groups of 7 instead of 5.

### *9.3-2*

Suppose that the preprocessing in lines 1–10 of SELECT is replaced by a base case for $n \ge n_0$, where $n_0$ is a suitable constant; that $g$ is chosen as $\lfloor r - p + 1)/5 \rfloor$; and that the elements in $A[5g : n]$ belong to no group. Show that although the recurrence for the running time becomes messier, it still solves to $\Theta(n)$.

### *9.3-3*

Show how to use SELECT as a subroutine to make quicksort run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

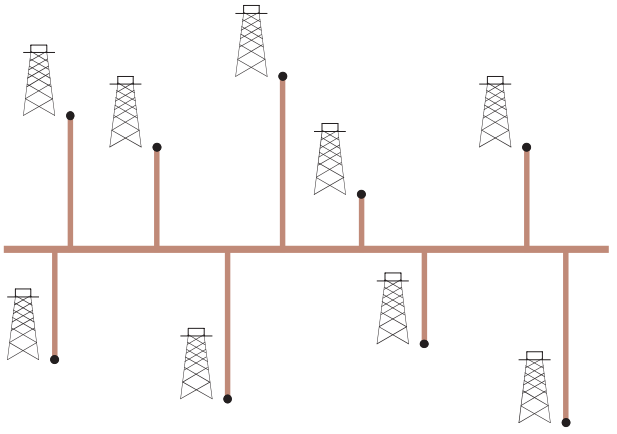

**Figure 9.4** Professor Olay needs to determine the position of the east-west oil pipeline that minimizes the total length of the north-south spurs.

★ ***9.3-4***

Suppose that an algorithm uses only comparisons to find the $i$th smallest element in a set of $n$ elements. Show that it can also find the $i - 1$ smaller elements and the $n - i$ larger elements without performing any additional comparisons.

***9.3-5***

Show how to determine the median of a 5-element set using only 6 comparisons.

***9.3-6***

You have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

***9.3-7***

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of $n$ wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.4. Given the $x$- and $y$-coordinates of the wells, how should the professor pick an optimal location of the main pipeline to minimize the total length of the spurs? Show how to determine an optimal location in linear time.

***9.3-8***

The $k$th ***quantiles*** of an $n$-element set are the $k - 1$ order statistics that divide the sorted set into $k$ equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the $k$th quantiles of a set.

***9.3-9***
Describe an $O(n)$-time algorithm that, given a set $S$ of $n$ distinct numbers and a positive integer $k \leq n$, determines the $k$ numbers in $S$ that are closest to the median of $S$.

***9.3-10***
Let $X[1:n]$ and $Y[1:n]$ be two arrays, each containing $n$ numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays $X$ and $Y$. Assume that all $2n$ numbers are distinct.

## Problems

***9-1 Largest $i$ numbers in sorted order***
You are given a set of $n$ numbers, and you wish to find the $i$ largest in sorted order using a comparison-based algorithm. Describe the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of $n$ and $i$.

***a.*** Sort the numbers, and list the $i$ largest.

***b.*** Build a max-priority queue from the numbers, and call EXTRACT-MAX $i$ times.

***c.*** Use an order-statistic algorithm to find the $i$th largest number, partition around that number, and sort the $i$ largest numbers.

***9-2 Variant of randomized selection***
Professor Mendel has proposed simplifying RANDOMIZED-SELECT by eliminating the check for whether $i$ and $k$ are equal. The simplified procedure is SIMPLER-RANDOMIZED-SELECT.

```
SIMPLER-RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]        // 1 ≤ i ≤ r − p + 1 means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i ≤ k
6       return SIMPLER-RANDOMIZED-SELECT(A, p, q, i)
7  else return SIMPLER-RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

***a.*** Argue that in the worst case, SIMPLER-RANDOMIZED-SELECT never terminates.

***b.*** Prove that the expected running time of SIMPLER-RANDOMIZED-SELECT is still $O(n)$.

### *9-3 Weighted median*

Consider $n$ elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$. The ***weighted (lower) median*** is an element $x_k$ satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \le \frac{1}{2} .$$

For example, consider the following elements $x_i$ and weights $w_i$:

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $x_i$ | 3 | 8 | 2 | 5 | 4 | 1 | 6 |
| $w_i$ | 0.12 | 0.35 | 0.025 | 0.08 | 0.15 | 0.075 | 0.2 |

For these elements, the median is $x_5 = 4$, but the weighted median is $x_7 = 6$. To see why the weighted median is $x_7$, observe that the elements less than $x_7$ are $x_1$, $x_3$, $x_4$, $x_5$, and $x_6$, and the sum $w_1 + w_3 + w_4 + w_5 + w_6 = 0.45$, which is less than $1/2$. Furthermore, only element $x_2$ is greater than $x_7$, and $w_2 = 0.35$, which is no greater than $1/2$.

***a.*** Argue that the median of $x_1, x_2, \ldots, x_n$ is the weighted median of the $x_i$ with weights $w_i = 1/n$ for $i = 1, 2, \ldots, n$.

***b.*** Show how to compute the weighted median of $n$ elements in $O(n \lg n)$ worst-case time using sorting.

***c.*** Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

The ***post-office location problem*** is defined as follows. The input is $n$ points $p_1, p_2, \ldots, p_n$ with associated weights $w_1, w_2, \ldots, w_n$. A solution is a point $p$ (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i \, d(p, p_i)$, where $d(a, b)$ is the distance between points $a$ and $b$.

***d.*** Argue that the weighted median is a best solution for the one-dimensional post-office location problem, in which points are simply real numbers and the distance between points $a$ and $b$ is $d(a,b) = |a - b|$.

***e.*** Find the best solution for the two-dimensional post-office location problem, in which the points are $(x, y)$ coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the ***Manhattan distance*** given by $d(a,b) = |x_1 - x_2| + |y_1 - y_2|$.

### *9-4 Small order statistics*

Let's denote by $S(n)$ the worst-case number of comparisons used by SELECT to select the $i$th order statistic from $n$ numbers. Although $S(n) = \Theta(n)$, the constant hidden by the $\Theta$-notation is rather large. When $i$ is small relative to $n$, there is an algorithm that uses SELECT as a subroutine but makes fewer comparisons in the worst case.

***a.*** Describe an algorithm that uses $U_i(n)$ comparisons to find the $i$th smallest of $n$ elements, where

$$U_i(n) = \begin{cases} S(n) & \text{if } i \geq n/2\,, \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + S(2i) & \text{otherwise}\,. \end{cases}$$

(*Hint:* Begin with $\lfloor n/2 \rfloor$ disjoint pairwise comparisons, and recurse on the set containing the smaller element from each pair.)

***b.*** Show that, if $i < n/2$, then $U_i(n) = n + O(S(2i)\lg(n/i))$.

***c.*** Show that if $i$ is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.

***d.*** Show that if $i = n/k$ for $k \geq 2$, then $U_i(n) = n + O(S(2n/k)\lg k)$.

### *9-5 Alternative analysis of randomized selection*

In this problem, you will use indicator random variables to analyze the procedure RANDOMIZED-SELECT in a manner akin to our analysis of RANDOMIZED-QUICKSORT in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array $A$ as $z_1, z_2, \ldots, z_n$, where $z_i$ is the $i$th smallest element. Thus the call RANDOMIZED-SELECT$(A, 1, n, i)$ returns $z_i$.

For $1 \leq j < k \leq n$, let

$$X_{ijk} = \mathrm{I}\{z_j \text{ is compared with } z_k \text{ sometime during the execution of the algorithm to find } z_i\}\,.$$

***a.*** Give an exact expression for $\mathrm{E}[X_{ijk}]$. (*Hint:* Your expression may have different values, depending on the values of $i$, $j$, and $k$.)

***b.*** Let $X_i$ denote the total number of comparisons between elements of array $A$ when finding $z_i$. Show that

$$\mathrm{E}[X_i] \le 2\left(\sum_{j=1}^{i}\sum_{k=i}^{n}\frac{1}{k-j+1} + \sum_{k=i+1}^{n}\frac{k-i-1}{k-i+1} + \sum_{j=1}^{i-2}\frac{i-j-1}{i-j+1}\right) .$$

***c.*** Show that $\mathrm{E}[X_i] \le 4n$.

***d.*** Conclude that, assuming all elements of array $A$ are distinct, RANDOMIZED-SELECT runs in $O(n)$ expected time.

### *9-6 Select with groups of 3*

Exercise 9.3-1 asks you to show that the SELECT algorithm still runs in linear time if the elements are divided into groups of 7. This problem asks about dividing into groups of 3.

***a.*** Show that SELECT runs in linear time if you divide the elements into groups whose size is any odd constant greater than 3.

***b.*** Show that SELECT runs in $O(n \lg n)$ time if you divide the elements into groups of size 3.

Because the bound in part (b) is just an upper bound, we do not know whether the groups-of-3 strategy actually runs in $O(n)$ time. But by repeating the groups-of-3 idea on the middle group of medians, we can pick a pivot that guarantees $O(n)$ time. The SELECT3 algorithm on the next page determines the $i$th smallest of an input array of $n > 1$ distinct elements.

***c.*** Describe in English how the SELECT3 algorithm works. Include in your description one or more suitable diagrams.

***d.*** Show that SELECT3 runs in $O(n)$ time in the worst case.

## Chapter notes

The worst-case linear-time median-finding algorithm was devised by Blum, Floyd, Pratt, Rivest, and Tarjan [62]. The fast randomized version is due to Hoare [218]. Floyd and Rivest [147] have developed an improved randomized version that partitions around an element recursively selected from a small sample of the elements.

```
SELECT3(A, p, r, i)
 1  while (r − p + 1) mod 9 ≠ 0
 2      for j = p + 1 to r                    // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p : r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/3                         // number of 3-element groups
12  for j = p to p + g − 1                    // run through the groups
13      sort ⟨A[j], A[j + g], A[j + 2g]⟩ in place
14  // All group medians now lie in the middle third of A[p : r].
15  g′ = g/3                                  // number of 3-element subgroups
16  for j = p + g to p + g + g′ − 1           // sort the subgroups
17      sort ⟨A[j], A[j + g′]A[j  + 2g′]⟩ in place
18  // All subgroup medians now lie in the middle ninth of A[p : r].
19  // Find the pivot x recursively as the median of the subgroup medians.
20  x = SELECT3(A, p + 4g′, p + 5g′ − 1, ⌈g′/2⌉)
21  q = PARTITION-AROUND(A, p, r, x)          // partition around the pivot
22  // The rest is just like lines 19–24 of SELECT.
23  k = q − p + 1
24  if i == k
25      return A[q]                           // the pivot value is the answer
26  elseif i < k
27      return SELECT3(A, p, q − 1, i)
28  else return SELECT3(A, q + 1, r, i − k)
```

It is still unknown exactly how many comparisons are needed to determine the median. Bent and John [48] gave a lower bound of $2n$ comparisons for median finding, and Schönhage, Paterson, and Pippenger [397] gave an upper bound of $3n$. Dor and Zwick have improved on both of these bounds. Their upper bound [123] is slightly less than $2.95n$, and their lower bound [124] is $(2 + \epsilon)n$, for a small positive constant $\epsilon$, thereby improving slightly on related work by Dor et al. [122]. Paterson [354] describes some of these results along with other related work.

Problem 9-6 was inspired by a paper by Chen and Dumitrescu [84].