Chapter 3

DIVIDE-AND-CONQUER

3.1 GENERAL METHOD

Given a function to compute on n inputs the *divide-and-conquer* strategy suggests splitting the inputs into k distinct subsets, $1 < k \leq n$, yielding k subproblems. These subproblems must be solved, and then a method must be found to combine subsolutions into a solution of the whole. If the subproblems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the subproblems resulting from a divide-and-conquer design are of the *same* type as the original problem. For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm. Now smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting are produced.

To be more precise, suppose we consider the divide-and-conquer strategy when it splits the input into two subproblems of the same kind as the original problem. This splitting is typical of many of the problems we examine here. We can write a control abstraction that mirrors the way an algorithm based on divide-and-conquer will look. By a *control abstraction* we mean a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. DAndC (Algorithm 3.1) is initially invoked as DAndC(P), where P is the problem to be solved.

Small(P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting. If this is so, the function S is invoked. Otherwise the problem P is divided into smaller subproblems. These subproblems $P_1, P_2, \ldots, P_k$ are solved by recursive applications of DAndC. Combine is a function that determines the solution to P using the solutions to the k subproblems. If the size of P is n and the sizes of the k subproblems are $n_1, n_2, \ldots, n_k$, respectively, then the

```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
9       }
10  }
```

Algorithm 3.1 Control abstraction for divide-and-conquer

computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases} \tag{3.1}$$

where $T(n)$ is the time for DAndC on any input of size n and $g(n)$ is the time to compute the answer directly for small inputs. The function $f(n)$ is the time for dividing P and combining the solutions to subproblems. For divide-and-conquer-based algorithms that produce subproblems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases} \tag{3.2}$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

One of the methods for solving any such recurrence relation is called the *substitution method*. This method repeatedly makes substitution for each occurrence of the function T in the right-hand side until all such occurrences disappear.

Example 3.1 Consider the case in which $a = 2$ and $b = 2$. Let $T(1) = 2$ and $f(n) = n$. We have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \end{aligned}$$

In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In particular, then, $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$, corresponding to the choice of $i = \log_2 n$. Thus, $T(n) = nT(1) + n \log_2 n = n \log_2 n + 2n$. □

Beginning with the recurrence (3.2) and using the substitution method, it can be shown that

$$T(n) = n^{\log_b a}[T(1) + u(n)]$$

where $u(n) = \sum_{j=1}^{k} h(b^j)$ and $h(n) = f(n)/n^{\log_b a}$. Table 3.1 tabulates the asymptotic value of $u(n)$ for various values of $h(n)$. This table allows one to easily obtain the asymptotic value of $T(n)$ for many of the recurrences one encounters when analyzing divide-and-conquer algorithms. Let us consider some examples using this table.

$h(n)$	$u(n)$
$O(n^r),\ r < 0$	$O(1)$
$\Theta((\log n)^i),\ i \geq 0$	$\Theta((\log n)^{i+1}/(i+1))$
$\Omega(n^r),\ r > 0$	$\Theta(h(n))$

Table 3.1 $u(n)$ values for various $h(n)$ values

Example 3.2 Look at the following recurrence when n is a power of 2:

$$T(n) = \begin{cases} T(1) & n = 1 \\ T(n/2) + c & n > 1 \end{cases}$$

Comparing with (3.2), we see that $a = 1,\ b = 2$, and $f(n) = c$. So, $\log_b(a) = 0$ and $h(n) = f(n)/n^{\log_b a} = c = c(\log n)^0 = \Theta((\log n)^0)$. From Table 3.1, we obtain $u(n) = \Theta(\log n)$. So, $T(n) = n^{\log_b a}[c + \Theta(\log n)] = \Theta(\log n)$. □

Example 3.3 Next consider the case in which $a = 2,\ b = 2$, and $f(n) = cn$. For this recurrence, $\log_b a = 1$ and $h(n) = f(n)/n = c = \Theta((\log n)^0)$. Hence, $u(n) = \Theta(\log n)$ and $T(n) = n[T(1) + \Theta(\log n)] = \Theta(n \log n)$. □

Example 3.4 As another example, consider the recurrence $T(n) = 7T(n/2) + 18n^2,\ n \geq 2$ and a power of 2. We obtain $a = 7,\ b = 2$, and $f(n) = 18n^2$. So, $\log_b a = \log_2 7 \approx 2.81$ and $h(n) = 18n^2/n^{\log_2 7} = 18n^{2-\log_2 7} = O(n^r)$, where $r = 2 - \log_2 7 < 0$. So, $u(n) = O(1)$. The expression for $T(n)$ is

$$\begin{aligned} T(n) &= n^{\log_2 7}[T(1) + O(1)] \\ &= \Theta(n^{\log_2 7}) \end{aligned}$$

as $T(1)$ is assumed to be a constant. □

Example 3.5 As a final example, consider the recurrence $T(n) = 9T(n/3) + 4n^6,\ n \geq 3$ and a power of 3. Comparing with (3.2), we obtain $a = 9,\ b = 3$, and $f(n) = 4n^6$. So, $\log_b a = 2$ and $h(n) = 4n^6/n^2 = 4n^4 = \Omega(n^4)$. From Table 3.1, we see that $u(n) = \Theta(h(n)) = \Theta(n^4)$. So,

$$\begin{aligned} T(n) &= n^2[T(1) + \Theta(n^4)] \\ &= \Theta(n^6) \end{aligned}$$

as $T(1)$ can be assumed constant. □

EXERCISES

1. Solve the recurrence relation (3.2) for the following choices of a, b, and $f(n)$ (c being a constant):

   (a) $a = 1,\ b = 2$, and $f(n) = cn$

   (b) $a = 5,\ b = 4$, and $f(n) = cn^2$

   (c) $a = 28,\ b = 3$, and $f(n) = cn^3$

2. Solve the following recurrence relations using the substitution method:

   (a) All three recurrences of Exercise 1.

   (b)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ T(\sqrt{n}) + c & n > 4 \end{cases}$$

(c)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \log n & n > 4 \end{cases}$$

(d)

$$T(n) = \begin{cases} 1 & n \leq 4 \\ 2T(\sqrt{n}) + \frac{\log n}{\log \log n} & n > 4 \end{cases}$$

3.2 BINARY SEARCH

Let a_i, $1 \leq i \leq n$, be a list of elements that are sorted in nondecreasing order. Consider the problem of determining whether a given element x is present in the list. If x is present, we are to determine a value j such that $a_j = x$. If x is not in the list, then j is to be set to zero. Let $P = (n, a_i, \ldots, a_\ell, x)$ denote an arbitrary instance of this search problem (n is the number of elements in the list, $a_i, \ldots, a_\ell$ is the list of elements, and x is the element searched for).

Divide-and-conquer can be used to solve this problem. Let Small(P) be true if $n = 1$. In this case, S(P) will take the value i if $x = a_i$; otherwise it will take the value 0. Then $g(1) = \Theta(1)$. If P has more than one element, it can be divided (or reduced) into a new subproblem as follows. Pick an index q (in the range $[i, \ell]$) and compare x with a_q. There are three possibilities: (1) $x = a_q$: In this case the problem P is immediately solved. (2) $x < a_q$: In this case x has to be searched for only in the sublist $a_i, a_{i+1}, \ldots, a_{q-1}$. Therefore, P reduces to $(q - i, a_i, \ldots, a_{q-1}, x)$. (3) $x > a_q$: In this case the sublist to be searched is $a_{q+1}, \ldots, a_\ell$. P reduces to $(\ell - q, a_{q+1}, \ldots, a_\ell, x)$.

In this example, any given problem P gets divided (reduced) into one new subproblem. This division takes only $\Theta(1)$ time. After a comparison with a_q, the instance remaining to be solved (if any) can be solved by using this divide-and-conquer scheme again. If q is always chosen such that a_q is the middle element (that is, $q = \lfloor (n+1)/2 \rfloor$), then the resulting search algorithm is known as binary search. Note that the answer to the new subproblem is also the answer to the original problem P; there is no need for any combining. Algorithm 3.2 describes this binary search method, where BinSrch has four inputs $a[\,], i, l$, and x. It is initially invoked as BinSrch$(a, 1, n, x)$.

A nonrecursive version of BinSrch is given in Algorithm 3.3. BinSearch has three inputs a, n, and x. The **while** loop continues processing as long as there are more elements left to check. At the conclusion of the procedure 0 is returned if x is not present, or j is returned, such that $a[j] = x$.

Is BinSearch an algorithm? We must be sure that all of the operations such as comparisons between x and $a[mid]$ are well defined. The relational operators carry out the comparisons among elements of a correctly if these operators are appropriately defined. Does BinSearch terminate? We observe

```
1   Algorithm BinSrch(a, i, l, x)
2   // Given an array a[i : l] of elements in nondecreasing
3   // order, 1 ≤ i ≤ l, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       if (l = i) then  // If Small(P)
7       {
8           if (x = a[i]) then return i;
9           else return 0;
10      }
11      else
12      { // Reduce P into a smaller subproblem.
13          mid := ⌊(i + l)/2⌋;
14          if (x = a[mid]) then return mid;
15          else if (x < a[mid]) then
16                  return BinSrch(a, i, mid − 1, x);
17              else return BinSrch(a, mid + 1, l, x);
18      }
19  }
```

Algorithm 3.2 Recursive binary search

```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
10          if (x < a[mid]) then high := mid − 1;
11          else if (x > a[mid]) then low := mid + 1;
12              else return mid;
13      }
14      return 0;
15  }
```

Algorithm 3.3 Iterative binary search

that *low* and *high* are integer variables such that each time through the loop either x is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* becomes greater than *high* and causes termination in a finite number of steps if x is not present.

Example 3.6 Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

place them in $a[1:14]$, and simulate the steps that BinSearch goes through as it searches for different values of x. Only the variables *low*, *high*, and *mid* need to be traced as we simulate the algorithm. We try the following values for x: 151, −14, and 9 for two successful searches and one unsuccessful search. Table 3.2 shows the traces of BinSearch on these three inputs. □

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Table 3.2 Three examples of binary search on 14 elements

These examples may give us a little more confidence about Algorithm 3.3, but they by no means prove that it is correct. Proofs of algorithms are very useful because they establish the correctness of the algorithm for *all* possible inputs, whereas testing gives much less in the way of guarantees. Unfortunately, algorithm proving is a very difficult process and the complete proof of an algorithm can be many times longer than the algorithm itself. We content ourselves with an informal "proof" of BinSearch.

Theorem 3.1 Algorithm BinSearch(a, n, x) works correctly.

Proof: We assume that all statements work as expected and that comparisons such as $x > a[mid]$ are appropriately carried out. Initially $low = 1$, $high = n$, $n \geq 0$, and $a[1] \leq a[2] \leq \cdots \leq a[n]$. If $n = 0$, the **while** loop is

not entered and 0 is returned. Otherwise we observe that each time through the loop the possible elements to be checked for equality with x are $a[low]$, $a[low+1]$, ..., $a[mid]$, ..., $a[high]$. If $x = a[mid]$, then the algorithm terminates successfully. Otherwise the range is narrowed by either increasing low to $mid+1$ or decreasing $high$ to $mid-1$. Clearly this narrowing of the range does not affect the outcome of the search. If low becomes greater than $high$, then x is not present and hence the loop is exited. □

Notice that to fully test binary search, we need not concern ourselves with the values of $a[1:n]$. By varying x sufficiently, we can observe all possible computation sequences of BinSearch without devising different values for a. To test all successful searches, x must take on the n values in a. To test all unsuccessful searches, x need only take on $n+1$ different values. Thus the complexity of testing BinSearch is $2n+1$ for each n.

Now let's analyze the execution profile of BinSearch. The two relevant characteristics of this profile are the frequency counts and space required for the algorithm. For BinSearch, storage is required for the n elements of the array plus the variables $low, high, mid$, and x, or $n+4$ locations. As for the time, there are three possibilities to consider: the best, average, and worst cases.

Suppose we begin by determining the time for BinSearch on the previous data set. We observe that the only operations in the algorithm are comparisons and some arithmetic and data movements. We concentrate on comparisons between x and the elements in $a[\ ]$, recognizing that the frequency count of all other operations is of the same order as that for these comparisons. Comparisons between x and elements of $a[\ ]$ are referred to as *element comparisons*. We assume that only one comparison is needed to determine which of the three possibilities of the **if** statement holds. The number of element comparisons needed to find each of the 14 elements is

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

No element requires more than 4 comparisons to be found. The average is obtained by summing the comparisons needed to find all 14 items and dividing by 14; this yields 45/14, or approximately 3.21, comparisons per successful search on the average. There are 15 possible ways that an unsuccessful search may terminate depending on the value of x. If $x < a[1]$, the algorithm requires 3 element comparisons to determine that x is not present. For all the remaining possibilities, BinSearch requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is $(3 + 14 * 4)/15 = 59/15 \approx 3.93$.

The analysis just done applies to any sorted sequence containing 14 elements. But the result we would prefer is a formula for n elements. A good

way to derive such a formula plus a better way to understand the algorithm is to consider the sequence of values for *mid* that are produced by BinSearch for all possible values of x. These values are nicely described using a binary decision tree in which the value in each node is the value of *mid*. For example, if $n = 14$, then Figure 3.1 contains a binary decision tree that traces the way in which these values are produced by BinSearch.

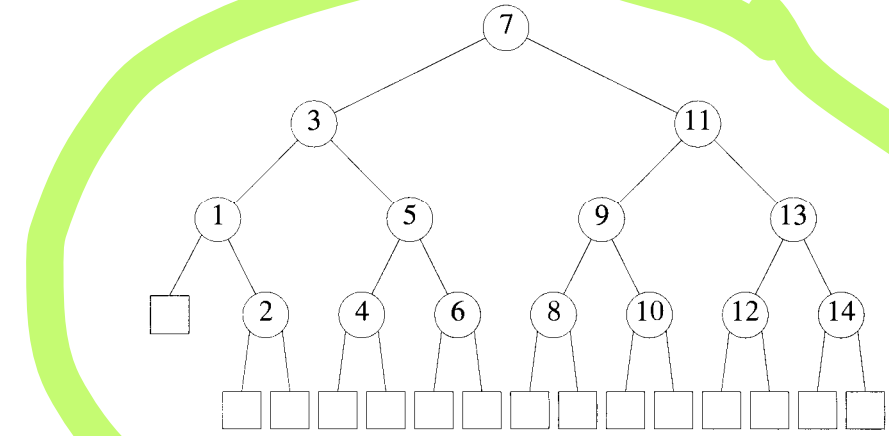

Figure 3.1 Binary decision tree for binary search, $n = 14$

The first comparison is x with $a[7]$. If $x < a[7]$, then the next comparison is with $a[3]$; similarly, if $x > a[7]$, then the next comparison is with $a[11]$. Each path through the tree represents a sequence of comparisons in the binary search method. If x is present, then the algorithm will end at one of the circular nodes that lists the index into the array where x was found. If x is not present, the algorithm will terminate at one of the square nodes. Circular nodes are called *internal* nodes, and square nodes are referred to as *external nodes*.

Theorem 3.2 If n is in the range $[2^{k-1}, 2^k)$, then BinSearch makes at most k element comparisons for a successful search and either $k-1$ or k comparisons for an unsuccessful search. (In other words the time for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$).

Proof: Consider the binary decision tree describing the action of BinSearch on n elements. All successful searches end at a circular node whereas all unsuccessful searches end at a square node. If $2^{k-1} \leq n < 2^k$, then all circular nodes are at levels $1, 2, \ldots, k$ whereas all square nodes are at levels

k and $k+1$ (note that the root is at level 1). The number of element comparisons needed to terminate at a circular node on level i is i whereas the number of element comparisons needed to terminate at a square node at level i is only $i-1$. The theorem follows. □

Theorem 3.2 states the worst-case time for binary search. To determine the average behavior, we need to look more closely at the binary decision tree and equate its size to the number of element comparisons in the algorithm. The *distance* of a node from the root is one less than its level. The *internal path length* I is the sum of the distances of all internal nodes from the root. Analogously, the *external path length* E is the sum of the distances of all external nodes from the root. It is easy to show by induction that for any binary tree with n internal nodes, E and I are related by the formula

$$E = I + 2n$$

It turns out that there is a simple relationship between E, I, and the average number of comparisons in binary search. Let $A_s(n)$ be the average number of comparisons in a successful search, and $A_u(n)$ the average number of comparisons in an unsuccessful search. The number of comparisons needed to find an element represented by an internal node is one more than the distance of this node from the root. Hence,

$$A_s(n) = 1 + I/n$$

The number of comparisons on any path from the root to an external node is equal to the distance between the root and the external node. Since every binary tree with n internal nodes has $n+1$ external nodes, it follows that

$$A_u(n) = E/(n+1)$$

Using these three formulas for $E, A_s(n)$, and $A_u(n)$, we find that

$$A_s(n) = (1 + 1/n)A_u(n) - 1$$

From this formula we see that $A_s(n)$ and $A_u(n)$ are directly related. The minimum value of $A_s(n)$ (and hence $A_u(n)$) is achieved by an algorithm whose binary decision tree has minimum external and internal path length. This minimum is achieved by the binary tree all of whose external nodes are on adjacent levels, and this is precisely the tree that is produced by binary search. From Theorem 3.2 it follows that E is proportional to $n \log n$. Using this in the preceding formulas, we conclude that $A_s(n)$ and $A_u(n)$ are both proportional to $\log n$. Thus we conclude that the average- and worst-case numbers of comparisons for binary search are the same to within a constant

factor. The best-case analysis is easy. For a successful search only one element comparison is needed. For an unsuccessful search, Theorem 3.2 states that $\lfloor \log n \rfloor$ element comparisons are needed in the best case.

In conclusion we are now able to completely describe the computing time of binary search by giving formulas that describe the best, average, and worst cases:

successful searches			**unsuccessful searches**
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

Can we expect another searching algorithm to be significantly better than binary search in the worst case? This question is pursued rigorously in Chapter 10. But we can anticipate the answer here, which is no. The method for proving such an assertion is to view the binary decision tree as a general model for any searching algorithm that depends on comparisons of entire elements. Viewed in this way, we observe that the *longest* path to discover any element is minimized by binary search, and so any alternative algorithm is no better from this point of view.

Before we end this section, there is an interesting variation of binary search that makes only one comparison per iteration of the **while** loop. This variation appears as Algorithm 3.4. The correctness proof of this variation is left as an exercise.

BinSearch will sometimes make twice as many element comparisons as BinSearch1 (for example, when $x > a[n]$). However, for successful searches BinSearch1 may make $(\log n)/2$ more element comparisons than BinSearch (for example, when $x = a[mid]$). The analysis of BinSearch1 is left as an exercise. It should be easy to see that the best-, average-, and worst-case times for BinSearch1 are $\Theta(\log n)$ for both successful and unsuccessful searches.

These two algorithms were run on a Sparc 10/30. The first two rows in Table 3.3 represent the average time for a successful search. The second set of two rows give the average times for all possible unsuccessful searches. For both successful and unsuccessful searches BinSearch1 did marginally better than BinSearch.

EXERCISES

1. Run the recursive and iterative versions of binary search and compare the times. For appropriate sizes of n, have each algorithm find every element in the set. Then try all $n + 1$ possible unsuccessful searches.

2. Prove by induction the relationship $E = I + 2n$ for a binary tree with n internal nodes. The variables E and I are the external and internal path length, respectively.

```
1   Algorithm BinSearch1(a, n, x)
2   // Same specifications as BinSearch except n > 0
3   {
4       low := 1; high := n + 1;
5       // high is one more than possible.
6       while (low < (high − 1)) do
7       {
8           mid := ⌊(low + high)/2⌋;
9           if (x < a[mid]) then high := mid;
10              // Only one comparison in the loop.
11          else low := mid; // x ≥ a[mid]
12      }
13      if (x = a[low]) then return low; // x is present.
14      else return 0; // x is not present.
15  }
```

Algorithm 3.4 Binary search using one comparison per cycle

Array sizes	5,000	10,000	15,000	20,000	25,000	30,000
successful searches						
BinSearch	51.30	67.95	67.72	73.85	76.77	73.40
BinSearch1	47.68	53.92	61.98	67.46	68.95	71.11
unsuccessful searches						
BinSearch	50.40	66.36	76.78	79.54	78.20	81.15
BinSearch1	41.93	52.65	63.33	66.86	69.22	72.26

Table 3.3 Computing times for two binary search algorithms; times are in microseconds

3. In an infinite array, the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. Present an algorithm that takes x as input and finds the position of x in the array in $\Theta(\log n)$ time. *You are not given the value of* n.

4. Devise a "binary" search algorithm that splits the set not into two sets of (almost) equal sizes but into two sets, one of which is twice the size of the other. How does this algorithm compare with binary search?

5. Devise a ternary search algorithm that first tests the element at position $n/3$ for equality with some value x, and then checks the element at $2n/3$ and either discovers x or reduces the set size to one-third the size of the original. Compare this with binary search.

6. (a) Prove that BinSearch1 works correctly.

   (b) Verify that the following algorithm segment functions correctly according to the specifications of binary search. Discuss its computing time.

```
low := 1; high := n;
repeat {
    mid := ⌊(low + high)/2⌋;
    if (x ≥ a[mid]) then low := mid;
    else high := mid;
} until ((low + 1) = high)
```

3.3 FINDING THE MAXIMUM AND MINIMUM

Let us consider another simple problem that can be solved by the divide-and-conquer technique. The problem is to find the maximum and minimum items in a set of n elements. Algorithm 3.5 is a straightforward algorithm to accomplish this.

In analyzing the time complexity of this algorithm, we once again concentrate on the number of element comparisons. The justification for this is that the frequency count for other operations in this algorithm is of the same order as that for element comparisons. More importantly, when the elements in $a[1:n]$ are polynomials, vectors, very large numbers, or strings of characters, the cost of an element comparison is much higher than the cost of the other operations. Hence the time is determined mainly by the total cost of the element comparisons.

StraightMaxMin requires $2(n-1)$ element comparisons in the best, average, and worst cases. An immediate improvement is possible by realizing

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

Algorithm 3.5 Straightforward maximum and minimum

that the comparison $a[i] < min$ is necessary only when $a[i] > max$ is false. Hence we can replace the contents of the **for** loop by

```
if (a[i] > max) then max := a[i];
else if (a[i] < min) then min := a[i];
```

Now the best case occurs when the elements are in increasing order. The number of element comparisons is $n - 1$. The worst case occurs when the elements are in decreasing order. In this case the number of element comparisons is $2(n-1)$. The average number of element comparisons is less than $2(n-1)$. On the average, $a[i]$ is greater than max half the time, and so the average number of comparisons is $3n/2 - 1$.

A divide-and-conquer algorithm for this problem would proceed as follows: Let $P = (n, a[i], \ldots, a[j])$ denote an arbitrary instance of the problem. Here n is the number of elements in the list $a[i], \ldots, a[j]$ and we are interested in finding the maximum and minimum of this list. Let Small(P) be true when $n \leq 2$. In this case, the maximum and minimum are $a[i]$ if $n = 1$. If $n = 2$, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances $P_1 = (\lfloor n/2 \rfloor, a[1], \ldots, a[\lfloor n/2 \rfloor])$ and $P_2 = (n - \lfloor n/2 \rfloor, a[\lfloor n/2 \rfloor + 1], \ldots, a[n])$. After having divided P into two smaller subproblems, we can solve them by recursively invoking the same divide-and-conquer algorithm. How can we combine the solutions for P_1 and P_2 to obtain a solution for P? If MAX(P) and MIN(P) are the maximum and minimum of the elements in P, then MAX(P) is the larger of MAX(P_1) and MAX(P_2). Also, MIN(P) is the smaller of MIN(P_1) and MIN(P_2).

Algorithm 3.6 results from applying the strategy just described. MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements $\{a(i), a(i+1), \ldots, a(j)\}$. The situation of set sizes one ($i = j$) and two ($i = j - 1$) are handled separately. For sets containing more than two elements, the midpoint is determined (just as in binary search) and two new subproblems are generated. When the maxima and minima of these subproblems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i = j) then max := min := a[i]; // Small(P)
7       else if (i = j − 1) then  // Another case of Small(P)
8           {
9               if (a[i] < a[j]) then
10              {
11                  max := a[j]; min := a[i];
12              }
13              else
14              {
15                  max := a[i]; min := a[j];
16              }
17          }
18          else
19          {   // If P is not small, divide P into subproblems.
20              // Find where to split the set.
21                  mid := ⌊(i + j)/2⌋;
22              // Solve the subproblems.
23                  MaxMin(i, mid, max, min);
24                  MaxMin(mid + 1, j, max1, min1);
25              // Combine the solutions.
26                  if (max < max1) then max := max1;
27                  if (min > min1) then min := min1;
28          }
29  }
```

Algorithm 3.6 Recursively finding the maximum and minimum

The procedure is initially invoked by the statement

$$\mathsf{MaxMin}(1, n, x, y)$$

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm each node has four items of information: i, j, max, and min. On the array $a[\,]$ above, the tree of Figure 3.2 is produced.

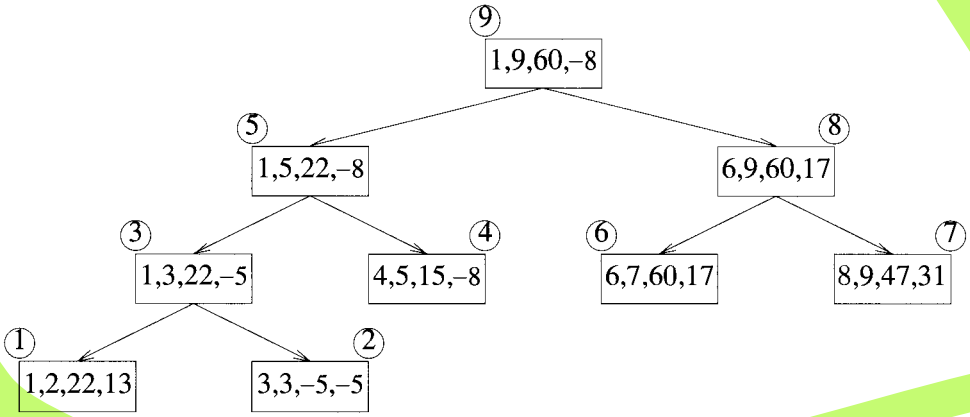

Figure 3.2 Trees of recursive calls of MaxMin

Examining Figure 3.2, we see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new calls to MaxMin, where i and j have the values 1, 5 and 6, 9, respectively, and thus split the set into two subsets of approximately the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call). The circled numbers in the upper left corner of each node represent the orders in which max and min are assigned values.

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \tag{3.3}$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

Compared with the $2n - 2$ comparisons for the straightforward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than $3n/2 - 2$ comparisons. So in this sense algorithm MaxMin is optimal (see Chapter 10 for more details). But does this imply that MaxMin is better in practice? Not necessarily. In terms of storage, MaxMin is worse than the straightforward algorithm because it requires stack space for i, j, $max, min, max1$, and $min1$. Given n elements, there will be $\lfloor \log_2 n \rfloor + 1$ levels of recursion and we need to save seven values for each recursive call (don't forget the return address is also needed).

Let us see what the count is when element comparisons have the same cost as comparisons between i and j. Let $C(n)$ be this number. First, we observe that lines 6 and 7 in Algorithm 3.6 can be replaced with

```
if (i ≥ j − 1) { // Small(P)
```

to achieve the same effect. Hence, a single comparison between i and $j - 1$ is adequate to implement the modified **if** statement. Assuming $n = 2^k$ for some positive integer k, we get

$$C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}$$

Solving this equation, we obtain

$$\begin{aligned} C(n) &= 2C(n/2) + 3 \\ &= 4C(n/4) + 6 + 3 \\ &\vdots \\ &= 2^{k-1}C(2) + 3\ \textstyle\sum_0^{k-2} 2^i \\ &= 2^k + 3 * 2^{k-1} - 3 \\ &= 5n/2 - 3 \end{aligned} \tag{3.4}$$

The comparative figure for StraightMaxMin is $3(n-1)$ (including the comparison needed to implement the **for** loop). This is larger than $5n/2 - 3$. Despite this, MaxMin will be slower than StraightMaxMin because of the overhead of stacking i, j, max, and min for the recursion.

Algorithm 3.6 makes several points. If comparisons among the elements of $a[\ ]$ are much more costly than comparisons of integer variables, then the divide-and-conquer technique has yielded a more efficient (actually an optimal) algorithm. On the other hand, if this assumption is not true, the technique yields a less-efficient algorithm. Thus the divide-and-conquer strategy is seen to be only a guide to better algorithm design which may not always succeed. Also we see that it is sometimes necessary to work out the constants associated with the computing time bound for an algorithm. Both MaxMin and StraightMaxMin are $\Theta(n)$, so the use of asymptotic notation is not enough of a discriminator in this situation. Finally, see the exercises for another way to find the maximum and minimum using only $3n/2 - 2$ comparisons.

Note: In the design of any divide-and-conquer algorithm, typically, it is a straightforward task to define Small(P) and S(P). So, from now on, we only discuss how to divide any given problem P and how to combine the solutions to subproblems.

EXERCISES

1. Translate algorithm MaxMin into a computationally equivalent procedure that uses no recursion.

2. Test your iterative version of MaxMin derived above against StraightMaxMin. Count all comparisons.

3. There is an iterative algorithm for finding the maximum and minimum which, though not a divide-and-conquer-based algorithm, is probably more efficient than MaxMin. It works by comparing consecutive pairs of elements and then comparing the larger one with the current maximum and the smaller one with the current minimum. Write out

the algorithm completely and analyze the number of comparisons it requires.

4. In Algorithm 3.6, what happens if lines 7 to 17 are dropped? Does the resultant function still compute the maximum and minimum elements correctly?

3.4 MERGE SORT

As another example of divide-and-conquer, we investigate a sorting algorithm that has the nice property that in the worst case its complexity is $O(n \log n)$. This algorithm is called *merge sort*. We assume throughout that the elements are to be sorted in nondecreasing order. Given a sequence of n elements (also called keys) $a[1], \ldots, a[n]$, the general idea is to imagine them split into two sets $a[1], \ldots, a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1], \ldots, a[n]$. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements. Thus we have another ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one.

MergeSort (Algorithm 3.7) describes this process very succinctly using recursion and a function Merge (Algorithm 3.8) which merges two sorted sets. Before executing MergeSort, the n elements should be placed in $a[1:n]$. Then MergeSort(1,n) causes the keys to be rearranged into nondecreasing order in a.

Example 3.7 Consider the array of ten elements $a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)$. Algorithm MergeSort begins by splitting $a[\ ]$ into two subarrays each of size five ($a[1:5]$ and $a[6:10]$). The elements in $a[1:5]$ are then split into two subarrays of size three ($a[1:3]$) and two ($a[4:5]$). Then the items in $a[1:3]$ are split into subarrays of size two ($a[1:2]$) and one ($a[3:3]$). The two values in $a[1:2]$ are split a final time into one-element subarrays, and now the merging begins. Note that no movement of data has yet taken place. A record of the subarrays is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as

(310 | 285 | 179 | 652, 351 | 423, 861, 254, 450, 520)

where vertical bars indicate the boundaries of subarrays. Elements $a[1]$ and $a[2]$ are merged to yield

(285, 310 | 179 | 652, 351 | 423, 861, 254, 450, 520)

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

Algorithm 3.7 Merge sort

Then $a[3]$ is merged with $a[1:2]$ and

(179, 285, 310 | 652, 351 | 423, 861, 254, 450, 520)

is produced. Next, elements $a[4]$ and $a[5]$ are merged:

(179, 285, 310 | 351, 652 | 423, 861, 254, 450, 520)

and then $a[1:3]$ and $a[4:5]$:

(179, 285, 310, 351, 652 | 423, 861, 254, 450, 520)

At this point the algorithm has returned to the first invocation of MergeSort and is about to process the second recursive call. Repeated recursive calls are invoked producing the following subarrays:

(179, 285, 310, 351, 652 | 423 | 861 | 254 | 450, 520)

Elements $a[6]$ and $a[7]$ are merged. Then $a[8]$ is merged with $a[6:7]$:

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

Algorithm 3.8 Merging two sorted subarrays using auxiliary storage

(179, 285, 310, 351, 652 | 254, 423, 861 | 450, 520)

Next $a[9]$ and $a[10]$ are merged, and then $a[6:8]$ and $a[9:10]$:

(179, 285, 310, 351, 652 | 254, 423, 450, 520, 861)

At this point there are two sorted subarrays and the final merge produces the fully sorted result

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

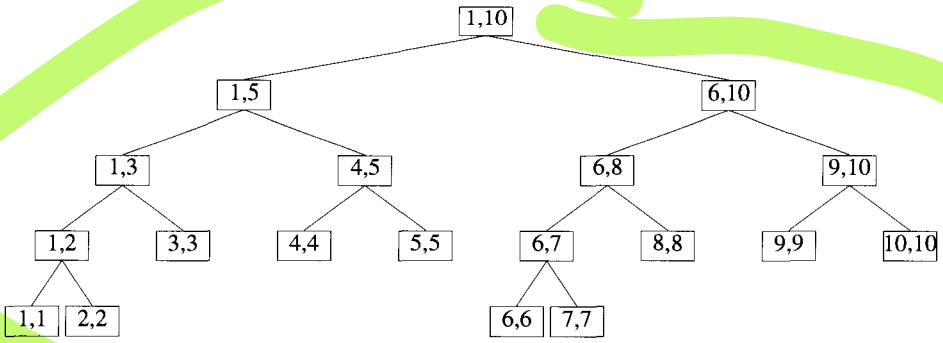

Figure 3.3 Tree of calls of MergeSort$(1, 10)$

Figure 3.3 is a tree that represents the sequence of recursive calls that are produced by MergeSort when it is applied to ten elements. The pair of values in each node are the values of the parameters *low* and *high*. Notice how the splitting continues until sets containing a single element are produced. Figure 3.4 is a tree representing the calls to procedure Merge by MergeSort. For example, the node containing 1, 2, and 3 represents the merging of $a[1:2]$ with $a[3]$. □

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

$$T(n) = O(n \log n)$$

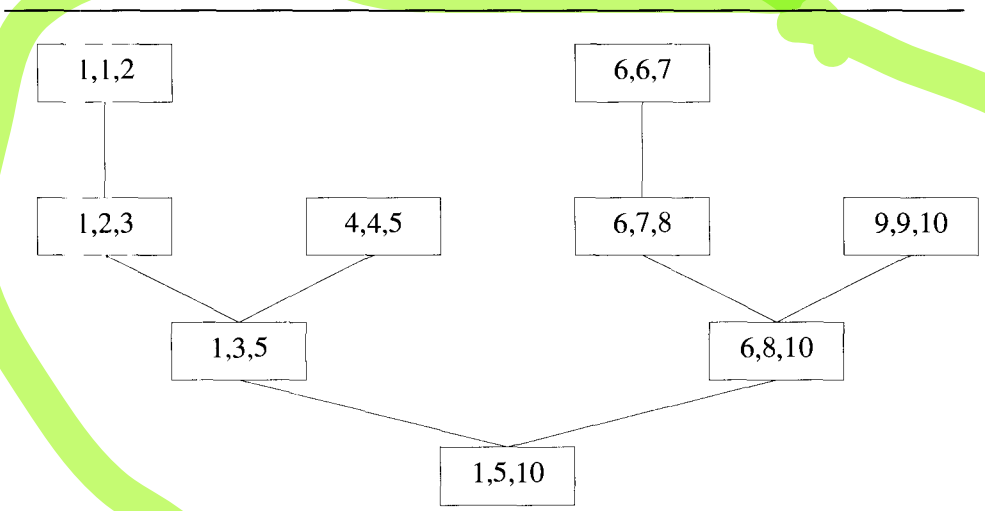

Figure 3.4 Tree of calls of Merge

Though Algorithm 3.7 nicely captures the divide-and-conquer nature of merge sort, there remain several inefficiencies that can and should be eliminated. We present these refinements in an attempt to produce a version of merge sort that is good enough to execute. Despite these improvements the algorithm's complexity remains $O(n \log n)$. We see in Chapter 10 that no sorting algorithm based on comparisons of entire keys can do better.

One complaint we might raise concerning merge sort is its use of $2n$ locations. The additional n locations were needed because we couldn't reasonably merge two sorted sets in place. But despite the use of this space the

algorithm must still work hard and copy the result placed into $b[low : high]$ back into $a[low : high]$ on each call of Merge. An alternative to this copying is to associate a new field of information with each key. (The elements in $a[\ ]$ are called *keys*.) This field is used to link the keys and any associated information together in a sorted list (keys and related information are called *records*). Then the merging of the sorted lists proceeds by changing the link values, and no records need be moved at all. A field that contains only a link will generally be smaller than an entire record, so less space will be used.

Along with the original array $a[\ ]$, we define an auxiliary array $link[1 : n]$ that contains integers in the range $[0, n]$. These integers are interpreted as pointers to elements of $a[\ ]$. A list is a sequence of pointers ending with a zero. Below is one set of values for $link$ that contains two lists: Q and R. The integer $Q = 2$ denotes the start of one list and $R = 5$ the start of the other.

$link$:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	6	4	7	1	3	0	8	0

The two lists are $Q = (2, 4, 1, 6)$ and $R = (5, 3, 7, 8)$. Interpreting these lists as describing sorted subsets of $a[1 : 8]$, we conclude that $a[2] \leq a[4] \leq a[1] \leq a[6]$ and $a[5] \leq a[3] \leq a[7] \leq a[8]$.

Another complaint we could raise about MergeSort is the stack space that is necessitated by the use of recursion. Since merge sort splits each set into two approximately equal-sized subsets, the maximum depth of the stack is proportional to $\log n$. The need for stack space seems indicated by the top-down manner in which this algorithm was devised. The need for stack space can be eliminated if we build an algorithm that works bottom-up; see the exercises for details.

As can be seen from function MergeSort and the previous example, even sets of size two will cause two recursive calls to be made. For small set sizes most of the time will be spent processing the recursion instead of sorting. This situation can be improved by not allowing the recursion to go to the lowest level. In terms of the divide-and-conquer control abstraction, we are suggesting that when Small is true for merge sort, more work should be done than simply returning with no action. We use a second sorting algorithm that works well on small-sized sets.

Insertion sort works exceedingly fast on arrays of less than, say, 16 elements, though for large n its computing time is $O(n^2)$. Its basic idea for sorting the items in $a[1 : n]$ is as follows:

```
for j := 2 to n do {
    place a[j] in its correct position in the sorted set a[1 : j − 1];
}
```

Though all the elements in $a[1 : j-1]$ may have to be moved to accommodate $a[j]$, for small values of n the algorithm works well. Algorithm 3.9 has the details.

```
1   Algorithm InsertionSort(a, n)
2   // Sort the array a[1 : n] into nondecreasing order, n ≥ 1.
3   {
4       for j := 2 to n do
5       {
6           // a[1 : j − 1] is already sorted.
7           item := a[j]; i := j − 1;
8           while ((i ≥ 1) and (item < a[i])) do
9           {
10              a[i + 1] := a[i]; i := i − 1;
11          }
12          a[i + 1] := item;
13      }
14  }
```

Algorithm 3.9 Insertion sort

The statements within the **while** loop can be executed zero up to a maximum of j times. Since j goes from 2 to n, the worst-case time of this procedure is bounded by

$$\sum_{2 \le j \le n} j = n(n+1)/2 - 1 = \Theta(n^2)$$

Its best-case computing time is $\Theta(n)$ under the assumption that the body of the **while** loop is never entered. This will be true when the data is already in sorted order.

We are now ready to present the revised version of merge sort with the inclusion of insertion sort and the links. Function MergeSort1 (Algorithm 3.10) is initially invoked by placing the keys of the records to be sorted in $a[1 : n]$ and setting $link[1 : n]$ to zero. Then one says MergeSort1$(1, n)$. A pointer to a list of indices that give the elements of $a[\ ]$ in sorted order is returned. Insertion sort is used whenever the number of items to be sorted is less than 16. The version of insertion sort as given by Algorithm 3.9 needs to be altered so that it sorts $a[low : high]$ into a linked list. Call the altered version InsertionSort1. The revised merging function, Merge1, is given in Algorithm 3.11.

```
1   Algorithm MergeSort1(low, high)
2   // The global array a[low : high] is sorted in nondecreasing order
3   // using the auxiliary array link[low : high]. The values in link
4   // represent a list of the indices low through high giving a[ ] in
5   // sorted order. A pointer to the beginning of the list is returned.
6   {
7       if ((high − low) < 15) then
8           return InsertionSort1(a, link, low, high);
9       else
10      {
11          mid := ⌊(low + high)/2⌋;
12          q := MergeSort1(low, mid);
13          r := MergeSort1(mid + 1, high);
14          return Merge1(q, r);
15      }
16  }
```

Algorithm 3.10 Merge sort using links

Example 3.8 As an aid to understanding this new version of merge sort, suppose we simulate the algorithm as it sorts the eight-element sequence (50, 10, 25, 30, 15, 70, 35, 55). We ignore the fact that less than 16 elements would normally be sorted using InsertionSort. The *link* array is initialized to zero. Table 3.4 shows how the *link* array changes after each call of MergeSort1 completes. On each row the value of p points to the list in *link* that was created by the last completion of Merge1. To the right are the subsets of sorted elements that are represented by these lists. For example, in the last row $p = 2$ which begins the list of links 2, 5, 3, 4, 7, 1, 8, and 6; this implies $a[2] \leq a[5] \leq a[3] \leq a[4] \leq a[7] \leq a[1] \leq a[8] \leq a[6]$. □

EXERCISES

1. Why is it necessary to have the auxiliary array $b[low : high]$ in function Merge? Give an example that shows why in-place merging is inefficient.

2. The worst-case time of procedure MergeSort is $O(n \log n)$. What is its best-case time? Can we say that the time for MergeSort is $\Theta(n \log n)$?

3. A sorting method is said to be *stable* if at the end of the method, identical elements occur in the same order as in the original unsorted

```
1   Algorithm Merge1(q, r)
2   // q and r are pointers to lists contained in the global array
3   // link[0 : n]. link[0] is introduced only for convenience and need
4   // not be initialized. The lists pointed at by q and r are merged
5   // and a pointer to the beginning of the merged list is returned.
6   {
7       i := q; j := r; k := 0;
8       // The new list starts at link[0].
9       while ((i ≠ 0) and (j ≠ 0)) do
10      { // While both lists are nonempty do
11          if (a[i] ≤ a[j]) then
12          { // Find the smaller key.
13              link[k] := i; k := i; i := link[i];
14              // Add a new key to the list.
15          }
16          else
17          {
18              link[k] := j; k := j; j := link[j];
19          }
20      }
21      if (i = 0) then link[k] := j;
22      else link[k] := i;
23      return link[0];
24  }
```

Algorithm 3.11 Merging linked lists of sorted elements

	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	
a:	-	50	10	25	30	15	70	35	55	
link:	0	0	0	0	0	0	0	0	0	
q r p										
1 2 2	2	0	1	0	0	0	0	0	0	(10, 50)
3 4 3	3	0	1	4	0	0	0	0	0	(10, 50), (25, 30)
2 3 2	2	0	3	4	1	0	0	0	0	(10, 25, 30, 50)
5 6 5	5	0	3	4	1	6	0	0	0	(10, 25, 30, 50), (15, 70)
7 8 7	7	0	3	4	1	6	0	8	0	(10, 25, 30, 50), (15, 70), (35, 55)
5 7 5	5	0	3	4	1	7	0	8	6	(10, 25, 30, 50) (15, 35, 55, 70)
2 5 2	2	8	5	4	7	3	0	1	6	(10, 15, 25, 30, 35, 50, 55, 70)

MergeSort1 applied to $a[1:8] = (50, 10, 25, 30, 15, 70, 35, 55)$

Table 3.4 Example of *link* array changes

set. Is merge sort a stable sorting method?

4. Suppose $a[1:m]$ and $b[1:n]$ both contain sorted elements in nondecreasing order. Write an algorithm that merges these items into $c[1:m+n]$. Your algorithm should be shorter than Algorithm 3.8 (Merge) since you can now place a large value in $a[m+1]$ and $b[n+1]$.

5. Given a file of n records that are partially sorted as $x_1 \le x_2 \le \cdots \le x_m$ and $x_{m+1} \le \cdots \le x_n$, is it possible to sort the entire file in time $O(n)$ using only a small fixed amount of additional storage?

6. Another way to sort a file of n records is to scan the file, merge consecutive pairs of size one, then merge pairs of size two, and so on. Write an algorithm that carries out this process. Show how your algorithm works on the data set (100, 300, 150, 450, 250, 350, 200, 400, 500).

7. A version of insertion sort is used by Algorithm 3.10 to sort small subarrays. However, its parameters and intent are slightly different from the procedure InsertionSort of Algorithm 3.9. Write a version of insertion sort that will work as Algorithm 3.10 expects.

8. The sequences $X_1, X_2, \ldots, X_\ell$ are sorted sequences such that $\sum_{i=1}^{\ell} |X_i| = n$. Show how to merge these ℓ sequences in time $O(n \log \ell)$.

3.5 QUICKSORT

The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort. In merge sort, the file $a[1:n]$ was divided

at its midpoint into subarrays which were independently sorted and later merged. In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in $a[1:n]$ such that $a[i] \leq a[j]$ for all i between 1 and m and all j between $m+1$ and n for some m, $1 \leq m \leq n$. Thus, the elements in $a[1:m]$ and $a[m+1:n]$ can be independently sorted. No merge is needed. The rearrangement of the elements is accomplished by picking some element of $a[\ ]$, say $t = a[s]$, and then reordering the other elements so that all elements appearing before t in $a[1:n]$ are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as *partitioning*.

Function Partition of Algorithm 3.12 (due to C. A. R. Hoare) accomplishes an in-place partitioning of the elements of $a[m:p-1]$. It is assumed that $a[p] \geq a[m]$ and that $a[m]$ is the partitioning element. If $m = 1$ and $p-1 = n$, then $a[n+1]$ must be defined and must be greater than or equal to all elements in $a[1:n]$. The assumption that $a[m]$ is the partition element is merely for convenience; other choices for the partitioning element than the first item in the set are better in practice. The function Interchange(a, i, j) exchanges $a[i]$ with $a[j]$.

Example 3.9 As an example of how Partition works, consider the following array of nine elements. The function is initially invoked as Partition$(a, 1, 10)$. The ends of the horizontal line indicate those elements which were interchanged to produce the next row. The element $a[1] = 65$ is the partitioning element and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about $a[5] = 65$. □

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Using Hoare's clever method of partitioning a set of elements about a chosen element, we can directly devise a divide-and-conquer method for completely sorting n elements. Following a call to the function Partition, two sets S_1 and S_2 are produced. All elements in S_1 are less than or equal

```
1   Algorithm Partition(a, m, p)
2   // Within a[m], a[m+1], ..., a[p-1] the elements are
3   // rearranged in such a manner that if initially t = a[m],
4   // then after completion a[q] = t for some q between m
5   // and p-1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6   // for q < k < p. q is returned. Set a[p] = ∞.
7   {
8       v := a[m]; i := m; j := p;
9       repeat
10      {
11          repeat
12              i := i + 1;
13          until (a[i] ≥ v);

14          repeat
15              j := j - 1;
16          until (a[j] ≤ v);

17          if (i < j) then Interchange(a, i, j);

18      } until (i ≥ j);

19      a[m] := a[j]; a[j] := v; return j;
20  }

1   Algorithm Interchange(a, i, j)
2   // Exchange a[i] with a[j].
3   {
4       p := a[i];
5       a[i] := a[j]; a[j] := p;
6   }
```

Algorithm 3.12 Partition the array $a[m : p-1]$ about $a[m]$

to the elements in S_2. Hence S_1 and S_2 can be sorted independently. Each set is sorted by reusing the function Partition. Algorithm 3.13 describes the complete process.

```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order; a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then  // If there are more than one element
7       {
8           // divide P into two subproblems.
9               j := Partition(a, p, q + 1);
10                  // j is the position of the partitioning element.
11          // Solve the subproblems.
12              QuickSort(p, j − 1);
13              QuickSort(j + 1, q);
14          // There is no need for combining solutions.
15      }
16  }
```

Algorithm 3.13 Sorting by partitioning

In analyzing QuickSort, we count only the number of element comparisons $C(n)$. It is easy to see that the frequency count of other operations is of the same order as $C(n)$. We make the following assumptions: the n elements to be sorted are distinct, and the input distribution is such that the partition element $v = a[m]$ in the call to Partition(a, m, p) has an equal probability of being the ith smallest element, $1 \leq i \leq p - m$, in $a[m : p - 1]$.

First, let us obtain the worst-case value $C_w(n)$ of $C(n)$. The number of element comparisons in each call of Partition is at most $p - m + 1$. Let r be the total number of elements in all the calls to Partition at any level of recursion. At level one only one call, Partition$(a, 1, n+1)$, is made and $r = n$; at level two at most two calls are made and $r = n - 1$; and so on. At each level of recursion, $O(r)$ element comparisons are made by Partition. At each level, r is at least one less than the r at the previous level as the partitioning elements of the previous level are eliminated. Hence $C_w(n)$ is the sum on r as r varies from 2 to n, or $O(n^2)$. Exercise 7 examines input data on which QuickSort uses $\Omega(n^2)$ comparisons.

The average value $C_A(n)$ of $C(n)$ is much less than $C_w(n)$. Under the assumptions made earlier, the partitioning element v has an equal probability

of being the ith-smallest element, $1 \leq i \leq p-m$, in $a[m:p-1]$. Hence the two subarrays remaining to be sorted are $a[m:j]$ and $a[j+1:p-1]$ with probability $1/(p-m), m \leq j < p$. From this we obtain the recurrence

$$C_A(n) = n+1+\frac{1}{n}\sum_{1\leq k\leq n}[C_A(k-1)) + C_A(n-k)] \tag{3.5}$$

The number of element comparisons required by Partition on its first call is $n+1$. Note that $C_A(0) = C_A(1) = 0$. Multiplying both sides of (3.5) by n, we obtain

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)] \tag{3.6}$$

Replacing n by $n-1$ in (3.6) gives

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

Subtracting this from (3.6), we get

$$\begin{aligned} nC_A(n) - (n-1)C_A(n-1) &= 2n + 2C_A(n-1) \\ \textit{or} & \\ C_A(n)/(n+1) &= C_A(n-1)/n + 2/(n+1) \end{aligned}$$

Repeatedly using this equation to substitute for $C_A(n-1), C_A(n-2), \ldots$, we get

$$\begin{aligned} \frac{C_A(n)}{n+1} &= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= \frac{C_A(1)}{2} + 2\sum_{3\leq k\leq n+1}\frac{1}{k} \\ &= 2\sum_{3\leq k\leq n+1}\frac{1}{k} \end{aligned} \tag{3.7}$$

Since

$$\sum_{3\leq k\leq n+1}\frac{1}{k} \leq \int_2^{n+1}\frac{1}{x}\,dx = \log_e(n+1) - \log_e 2$$

(3.7) yields

$$C_A(n) \leq 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

Even though the worst-case time is $O(n^2)$, the average time is only $O(n \log n)$. Let us now look at the stack space needed by the recursion. In the worst case the maximum depth of recursion may be $n-1$. This happens, for example, when the partition element on each call to Partition is the smallest value in $a[m : p-1]$. The amount of stack space needed can be reduced to $O(\log n)$ by using an iterative version of quicksort in which the smaller of the two subarrays $a[p : j-1]$ and $a[j+1 : q]$ is always sorted first. Also, the second recursive call can be replaced by some assignment statements and a jump to the beginning of the algorithm. With these changes, QuickSort takes the form of Algorithm 3.14.

We can now verify that the maximum stack space needed is $O(\log n)$. Let $S(n)$ be the maximum stack space needed. Then it follows that

$$S(n) \leq \begin{cases} 2 + S(\lfloor (n-1)/2 \rfloor) & n > 1 \\ 0 & n \leq 1 \end{cases}$$

which is less than $2 \log n$.

As remarked in Section 3.4, InsertionSort is exceedingly fast for n less than about 16. Hence InsertionSort can be used to speed up QuickSort2 whenever $q - p < 16$. The exercises explore various possibilities for selection of the partition element.

3.5.1 Performance Measurement

QuickSort and MergeSort were evaluated on a SUN workstation 10/30. In both cases the recursive versions were used. For QuickSort the Partition function was altered to carry out the median of three rule (i.e. the partitioning element was the median of $a[m]$, $a[\lfloor (m+p-1)/2 \rfloor]$ and $a[p-1]$). Each data set consisted of random integers in the range (0, 1000). Tables 3.5 and 3.6 record the actual computing times in milliseconds. Table 3.5 displays the average computing times. For each n, 50 random data sets were used. Table 3.6 shows the worst-case computing times for the 50 data sets.

Scanning the tables, we immediately see that QuickSort is faster than MergeSort for all values. Even though both algorithms require $O(n \log n)$ time on the average, QuickSort usually performs well in practice. The exercises discuss other tests that would make useful comparisons.

3.5.2 Randomized Sorting Algorithms

Though algorithm QuickSort has an average time of $O(n \log n)$ on n elements, its worst-case time is $O(n^2)$. On the other hand it does not make use of any

```
1   Algorithm QuickSort2(p, q)
2   // Sorts the elements in a[p : q].
3   {
4       // stack is a stack of size 2 log(n).
5       repeat
6       {
7           while (p < q) do
8           {
9               j := Partition(a, p, q + 1);
10              if ((j − p) < (q − j)) then
11              {
12                  Add(j + 1); // Add j + 1 to stack.
13                  Add(q); q := j − 1; // Add q to stack
14              }
15              else
16              {
17                  Add(p); // Add p to stack.
18                  Add(j − 1); p := j + 1; // Add j − 1 to stack
19              }
20          } // Sort the smaller subfile.
21          if stack is empty then return;
22          Delete(q); Delete(p); // Delete q and p from stack.
23      } until (false);
24  }
```

Algorithm 3.14 Iterative version of QuickSort

additional memory as does MergeSort. A possible input on which QuickSort displays worst-case behavior is one in which the elements are already in sorted order. In this case the partition will be such that there will be only one element in one part and the rest of the elements will fall in the other part. The performance of any divide-and-conquer algorithm will be good if the resultant subproblems are as evenly sized as possible. Can QuickSort be modified so that it performs well on every input? The answer is yes. Is the technique of using the median of the three elements $a[p]$, $a[\lfloor (q+p)/2 \rfloor]$, and $a[q]$ the solution? Unfortunately it is possible to construct inputs for which even this method will take $\Omega(n^2)$ time, as is explored in the exercises.

The solution is the use of a randomizer. While sorting the array $a[p : q]$, instead of picking $a[m]$, pick a random element (from among $a[p], \ldots, a[q]$) as the partition element. The resultant randomized algorithm (RQuickSort)

n	1000	2000	3000	4000	5000
MergeSort	72.8	167.2	275.1	378.5	500.6
QuickSort	36.6	85.1	138.9	205.7	269.0
n	6000	7000	8000	9000	10000
MergeSort	607.6	723.4	811.5	949.2	1073.6
QuickSort	339.4	411.0	487.7	556.3	645.2

Table 3.5 Average computing times for two sorting algorithms on random inputs

n	1000	2000	3000	4000	5000
MergeSort	105.7	206.4	335.2	422.1	589.9
QuickSort	41.6	97.1	158.6	244.9	397.8
n	6000	7000	8000	9000	10000
MergeSort	691.3	794.8	889.5	1067.2	1167.6
QuickSort	383.8	497.3	569.9	616.2	738.1

Table 3.6 Worst-case computing times for two sorting algorithms on random inputs

works on any input and runs in an expected $O(n \log n)$ time, where the expectation is over the space of all possible outcomes for the randomizer (rather than the space of all possible inputs). The code for RQuickSort is given in Algorithm 3.15. Note that this is a Las Vegas algorithm since it will always output the correct answer. Every call to the randomizer Random takes a certain amount of time. If there are only a very few elements to sort, the time taken by the randomizer may be comparable to the rest of the computation. For this reason, we invoke the randomizer only if $(q - p) > 5$. But 5 is not a magic number; in the machine employed, this seems to give the best results. In general this number should be determined empirically.

```
1   Algorithm RQuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order. a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then
7       {
8           if ((q − p) > 5) then
9               Interchange(a, Random() mod (q − p + 1) + p, p);
10          j := Partition(a, p, q + 1);
11              // j is the position of the partitioning element.
12          RQuickSort(p, j − 1);
13          RQuickSort(j + 1, q);
14      }
15  }
```

Algorithm 3.15 Randomized quick sort algorithm

The proof of the fact that RQuickSort has an expected $O(n \log n)$ time is the same as the proof of the average time of QuickSort. Let $A(n)$ be the average time of RQuickSort on *any input* of n elements. Then the number of elements in the second part will be $0, 1, 2, \ldots, n-2$, or $n-1$, all with an equal probability of $\frac{1}{n}$ (in the probability space of outcomes for the randomizer). Thus the recurrence relation for $A(n)$ will be

$$A(n) = \frac{1}{n} \sum_{1 \le k \le n} (A(k-1) + A(n-k)) + n + 1$$

This is the same as Equation 3.4, and hence its solution is $O(n \log n)$.

RQuickSort and QuickSort (without employing the median of three elements rule) were evaluated on a SUN 10/30 workstation. Table 3.7 displays

the times for the two algorithms in milliseconds averaged over 100 runs. For each n, the input considered was the sequence of numbers $1, 2, \ldots, n$. As we can see from the table, RQuickSort performs much better than QuickSort. Note that the times shown in this table for QuickSort are much more than the corresponding entries in Tables 3.5 and 3.6. The reason is that QuickSort makes $\Theta(n^2)$ comparisons on inputs that are already in sorted order. However, on random inputs its average performance is very good.

n	1000	2000	3000	4000	5000
QuickSort	195.5	759.2	1728	3165	4829
RQuickSort	9.4	21.0	30.5	41.6	52.8

Table 3.7 Comparison of QuickSort and RQuickSort on the input $a[i] = i$, $1 \leq i \leq n$; times are in milliseconds.

The performance of RQuickSort can be improved in various ways. For example, we could pick a small number (say 11) of the elements in the array $a[\,]$ randomly and use the median of these elements as the partition element. These randomly chosen elements form a random sample of the array elements. We would expect that the median of the sample would also be an approximate median of the array and hence result in an approximately even partitioning of the array.

An even more generalized version of the above random sampling technique is shown in Algorithm 3.16. Here we choose a random sample S of s elements (where s is a function of n) from the input sequence X and sort them using HeapSort, MergeSort, or any other sorting algorithm. Let $\ell_1, \ell_2, \ldots, \ell_s$ be the sorted sample. We partition X into $s+1$ parts using the sorted sample as partition keys. In particular $X_1 = \{x \in X | x \leq \ell_1\}$; $X_i = \{x \in X | \ell_{i-1} < x \leq \ell_i\}$, for $i = 2, 3, \ldots, s$; and $X_{s+1} = \{x \in X | x > \ell_s\}$. After having partitioned X into $s+1$ parts, we sort each part recursively. For a proper choice of s, the number of comparisons made in this algorithm is only $n \log n + \tilde{o}(n \log n)$. Note the constant 1 before $n \log n$. We see in Chapter 10 that this number is very close to the information theoretic lower bound for sorting.

Choose $s = \frac{n}{\log^2 n}$. The sample can be sorted in $O(s \log s) = O(\frac{n}{\log n})$ time and comparisons if we use HeapSort or MergeSort. If we store the sorted sample elements in an array, say $b[\,]$, for each $x \in X$, we can determine which part X_i it belongs to in $\leq \log n$ comparisons using binary search on $b[\,]$. Thus the partitioning process takes $n \log n + O(n)$ comparisons. In the exercises you are asked to show that with high probability the cardinality

```
1   Algorithm RSort(a, n)
2   // Sort the elements a[1 : n].
3   {
4       Randomly sample s elements from a[ ];
5       Sort this sample;
6       Partition the input using the sorted sample as partition keys;
7       Sort each part separately;
8   }
```

Algorithm 3.16 A randomized algorithm for sorting

of each X_i is no more than $\widetilde{O}(\frac{n}{s}\log n) = \widetilde{O}(\log^3 n)$. Using HeapSort or MergeSort to sort each of the X_i's (without employing recursion on any of them), the total cost of sorting the X_i's is

$$\sum_{i=1}^{s+1} O(|X_i|\log|X_i|) = \max_{1\le i\le s+1}\{\log|X_i|\}\sum_{i=1}^{s+1} O(|X_i|)$$

Since each $|X_i|$ is $\widetilde{O}(\log^3 n)$, the cost of sorting the $s+1$ parts is $\widetilde{O}(n\log\log n) = \widetilde{o}(n\log n)$. In summary, the number of comparisons made in this randomized sorting algorithm is $n\log n + \widetilde{o}(n\log n)$.

EXERCISES

1. Show how QuickSort sorts the following sequences of keys: 1, 1, 1, 1, 1, 1, 1 and 5, 5, 8, 3, 4, 3, 2.

2. QuickSort is not a stable sorting algorithm. However, if the key in $a[i]$ is changed to $a[i] * n + i - 1$, then the new keys are all distinct. After sorting, which transformation will restore the keys to their original values?

3. In the function Partition, Algorithm 3.12, discuss the merits or demerits of altering the statement **if** $(i < j)$ to **if** $(i \le j)$. Simulate both algorithms on the data set (5, 4, 3, 2, 5, 8, 9) to see the difference in how they work.

4. Function QuickSort uses the output of function Partition, which returns the position where the partition element is placed. If equal keys are present, then two elements can be properly placed instead of one. Show

how you might change Partition so that QuickSort can take advantage of this situation.

5. In addition to Partition, there are many other ways to partition a set. Consider modifying Partition so that i is incremented while $a[i] \leq v$ instead of $a[i] < v$. Rewrite Partition making all of the necessary changes to it and then compare the new version with the original.

6. Compare the sorting methods MergeSort1 and QuickSort2 (Algorithm 3.10 and 3.14, respectively). Devise data sets that compare both the average- and worst-case times for these two algorithms.

7. (a) On which input data does the algorithm QuickSort exhibit its worst-case behavior?

   (b) Answer part (a) for the case in which the partitioning element is selected according to the median of three rule.

8. With MergeSort we included insertion sorting to eliminate the bookkeeping for small merges. How would you use this technique to improve QuickSort?

9. Take the iterative versions of MergeSort and QuickSort and compare them for the same-size data sets as used in Section 3.5.1.

10. Let S be a sample of s elements from X. If X is partitioned into $s+1$ parts as in Algorithm 3.16, show that the size of each part is $\tilde{O}(\frac{n}{s}\log n)$.

3.6 SELECTION

The Partition algorithm of Section 3.5 can also be used to obtain an efficient solution for the selection problem. In this problem, we are given n elements $a[1:n]$ and are required to determine the kth-smallest element. If the partitioning element v is positioned at $a[j]$, then $j-1$ elements are less than or equal to $a[j]$ and $n-j$ elements are greater than or equal to $a[j]$. Hence if $k < j$, then the kth-smallest element is in $a[1:j-1]$; if $k = j$, then $a[j]$ is the kth-smallest element; and if $k > j$, then the kth-smallest element is the $(k-j)$th-smallest element in $a[j+1:n]$. The resulting algorithm is function Select1 (Algorithm 3.17). This function places the kth-smallest element into position $a[k]$ and partitions the remaining elements so that $a[i] \leq a[k]$, $1 \leq i < k$, and $a[i] \geq a[k]$, $k < i \leq n$.

Example 3.10 Let us simulate Select1 as it operates on the same array used to test Partition in Section 3.5. The array has the nine elements 65, 70,

```
1   Algorithm Select1(a, n, k)
2   // Selects the kth-smallest element in a[1 : n] and places it
3   // in the kth position of a[ ]. The remaining elements are
4   // rearranged such that a[m] ≤ a[k] for 1 ≤ m < k, and
5   // a[m] ≥ a[k] for k < m ≤ n.
6   {
7       low := 1; up := n + 1;
8       a[n + 1] := ∞; // a[n + 1] is set to infinity.
9       repeat
10      {
11          // Each time the loop is entered,
12          // 1 ≤ low ≤ k ≤ up ≤ n + 1.
13          j := Partition(a, low, up);
14                // j is such that a[j] is the jth-smallest value in a[ ].
15          if (k = j) then return;
16          else if (k < j) then up := j; // j is the new upper limit.
17                else low := j + 1; // j + 1 is the new lower limit.
18      } until (false);
19  }
```

Algorithm 3.17 Finding the kth-smallest element

75, 80, 85, 60, 55, 50, and 45, with $a[10] = \infty$. If $k = 5$, then the first call of Partition will be sufficient since 65 is placed into $a[5]$. Instead, assume that we are looking for the seventh-smallest element of a, that is, $k = 7$. The next invocation of Partition is Partition$(6, 10)$.

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	85	80	75	70	$+\infty$
	65	70	80	75	85	$+\infty$

This last call of Partition has uncovered the ninth-smallest element of a. The next invocation is Partition$(6, 9)$.

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	80	75	85	$+\infty$

This time, the sixth element has been found. Since $k \neq j$, another call to Partition is made, Partition$(7, 9)$.

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	75	80	85	$+\infty$

Now 80 is the partition value and is correctly placed at $a[8]$. However, Select1 has still not found the seventh-smallest element. It needs one more call to Partition, which is Partition$(7, 8)$. This performs only an interchange between $a[7]$ and $a[8]$ and returns, having found the correct value. □

In analyzing Select1, we make the same assumptions that were made for QuickSort:

1. The n elements are distinct.
2. The input distribution is such that the partition element can be the ith-smallest element of $a[m : p-1]$ with an equal probability for each i, $1 \leq i \leq p-m$.

Partition requires $O(p-m)$ time. On each successive call to Partition, either m increases by at least one or j decreases by at least one. Initially $m = 1$ and $j = n+1$. Hence, at most n calls to Partition can be made. Thus, the worst-case complexity of Select1 is $O(n^2)$. The time is $\Omega(n^2)$, for example, when the input $a[1 : n]$ is such that the partitioning element on the ith call to Partition is the ith-smallest element and $k = n$. In this case, m increases by one following each call to Partition and j remains unchanged. Hence, n calls are made for a total cost of $O(\sum_1^n i) = O(n^2)$. The average computing time of Select1 is, however, only $O(n)$. Before proving this fact, we specify more precisely what we mean by the average time.

Let $T_A^k(n)$ be the average time to find the kth-smallest element in $a[1 : n]$. This average is taken over all $n!$ different permutations of n distinct elements. Now define $T_A(n)$ and $R(n)$ as follows:

$$T_A(n) = \frac{1}{n} \sum_{1 \leq k \leq n} T_A^k(n)$$

and

$$R(n) = \max_k \{T_A^k(n)\}$$

$T_A(n)$ is the average computing time of Select1. It is easy to see that $T_A(n) \leq R(n)$. We are now ready to show that $T_A(n) = O(n)$.

Theorem 3.3 The average computing time $T_A(n)$ of Select1 is $O(n)$.

Proof: On the first call to Partition, the partitioning element v is the ith-smallest element with probability $\frac{1}{n}, 1 \leq i \leq n$ (this follows from the assumption on the input distribution). The time required by Partition and the **if** statement in Select1 is $O(n)$. Hence, there is a constant $c, c > 0$, such that

$$\begin{aligned} T_A^k(n) &\leq cn + \frac{1}{n}[\sum_{1\leq i<k} T_A^{k-1}(n-i) + \sum_{k<i\leq n} T_A^k(i-1)], \quad n \geq 2 \\ \text{So, } R(n) &\leq cn + \frac{1}{n}\max_k \{\sum_{1\leq i<k} R(n-i) + \sum_{k<i\leq n} R(i-1)\} \\ R(n) &\leq cn + \frac{1}{n}\max_k \{\sum_{n-k+1}^{n-1} R(i) + \sum_{k}^{n-1} R(i)\}, \quad n \geq 2 \end{aligned} \tag{3.8}$$

We assume that c is chosen such that $R(1) \leq c$ and show, by induction on n, that $R(n) \leq 4cn$.

Induction Base: For $n = 2$, (3.8) gives

$$\begin{aligned} R(n) &\leq 2c + \frac{1}{2}\max \{R(1), R(1)\} \\ &\leq 2.5c < 4cn \end{aligned}$$

Induction Hypothesis: Assume $R(n) \leq 4cn$ for all $n, 2 \leq n < m$.

Induction Step: For $n = m$, (3.8) gives

$$R(m) \leq cm + \frac{1}{m}\max_k \left\{ \sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i) \right\}$$

Since we know that $R(n)$ is a nondecreasing function of n, it follows that

$$\sum_{m-k+1}^{m-1} R(i) + \sum_{k}^{m-1} R(i)$$

is maximized if $k = \frac{m}{2}$ when m is even and $k = \frac{m+1}{2}$ when m is odd. Thus, if m is even, we obtain

$$R(m) \leq cm + \frac{2}{m}\sum_{m/2}^{m-1} R(i)$$

$$\begin{aligned} & \leq cm + \frac{8c}{m}\sum_{m/2}^{m-1} i \\ & < 4cm \\ \text{If } m \text{ is odd, } R(m) & \leq cm + \frac{2}{m}\sum_{(m+1)/2}^{m-1} R(i) \\ & \leq cm + \frac{8c}{m}\sum_{(m+1)/2}^{m-1} \\ & < 4cm \end{aligned}$$

Since $T_A(n) \leq R(n)$, it follows that $T_A(n) \leq 4cn$, and so $T_A(n)$ is $O(n)$. □

The space needed by Select1 is $O(1)$.

Algorithm 3.15 is a randomized version of QuickSort in which the partition element is chosen from the array elements randomly with equal probability. The same technique can be applied to Select1 and the partition element can be chosen to be a random array element. The resulting randomized Las Vegas algorithm (call it RSelect) has an expected time of $O(n)$ (where the expectation is over the space of randomizer outputs) on *any input.* The proof of this expected time is the same as in Theorem 3.3.

3.6.1 A Worst-Case Optimal Algorithm

By choosing the partitioning element v more carefully, we can obtain a selection algorithm with worst-case complexity $O(n)$. To obtain such an algorithm, v must be chosen so that at least some fraction of the elements is smaller than v and at least some (other) fraction of elements is greater than v. Such a selection of v can be made using the median of medians (mm) rule. In this rule the n elements are divided into $\lfloor n/r \rfloor$ groups of r elements each (for some $r, r > 1$). The remaining $n - r\lfloor n/r \rfloor$ elements are not used. The median m_i of each of these $\lfloor n/r \rfloor$ groups is found. Then, the median mm of the m_i's, $1 \leq i \leq \lfloor n/r \rfloor$, is found. The median mm is used as the partitioning element. Figure 3.5 illustrates the m_i's and mm when $n = 35$ and $r = 7$. The five groups of elements are $B_i, 1 \leq i \leq 5$. The seven elements in each group have been arranged into nondecreasing order down the column. The middle elements are the m_i's. The columns have been arranged in nondecreasing order of m_i. Hence, the m_i corresponding to column 3 is mm.

Since the median of r elements is the $\lceil r/2 \rceil$th-smallest element, it follows (see Figure 3.5) that at least $\lceil \lfloor n/r \rfloor /2 \rceil$ of the m_i's are less than or equal to mm and at least $\lfloor n/r \rfloor - \lceil \lfloor n/r \rfloor /2 \rceil + 1 \geq \lceil \lfloor n/r \rfloor /2 \rceil$ of the m_i's are greater than or equal to mm. Hence, at least $\lceil r/2 \rceil \lceil \lfloor n/r \rfloor /2 \rceil$ elements are less than

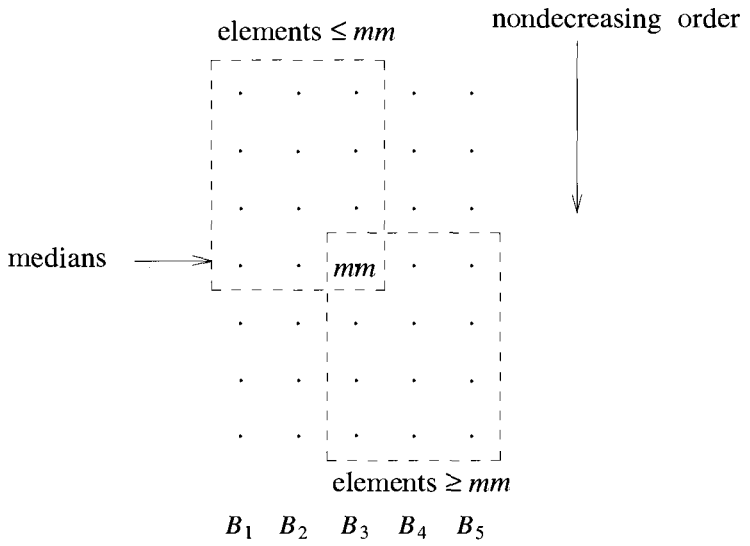

Figure 3.5 The median of medians when $r = 7$, $n = 35$

or equal to (or greater than or equal to) mm. When $r = 5$, this quantity is at least $1.5\lfloor n/5 \rfloor$. Thus, if we use the median of medians rule with $r = 5$ to select $v = mm$, we are assured that at least $1.5\lfloor n/5 \rfloor$ elements will be greater than or equal to v. This in turn implies that at most $n - 1.5\lfloor n/5 \rfloor \leq .7n + 1.2$ elements are less than v. Also, at most $.7n + 1.2$ elements are greater than v. Thus, the median of medians rule satisfies our earlier requirement on v.

The algorithm to select the kth-smallest element uses the median of medians rule to determine a partitioning element. This element is computed by a recursive application of the selection algorithm. A high-level description of the new selection algorithm appears as Select2 (Algorithm 3.18). Select2 can now be analyzed for any given r. First, let us consider the case in which $r = 5$ and all elements in $a[\ ]$ are distinct. Let $T(n)$ be the worst-case time requirement of Select2 when invoked with $up - low + 1 = n$. Lines 4 to 9 and 11 to 12 require at most $O(n)$ time (note that since $r = 5$ is fixed, each $m[i]$ (lines 8 and 9) can be found in $O(1)$ time). The time for line 10 is $T(n/5)$. Let S and R, respectively, denote the elements $a[low : j-1]$ and $a[j+1 : up]$. We see that $|S|$ and $|R|$ are at most $.7n + 1.2$, which is no more than $3n/4$ for $n \geq 24$. So, the time for lines 13 to 16 is at most $T(3n/4)$ when $n \geq 24$. Hence, for $n \geq 24$, we obtain

```
1   Algorithm Select2(a, k, low, up)
2   // Find the k-th smallest in a[low : up].
3   {
4       n := up − low + 1;
5       if (n ≤ r) then sort a[low : up] and return the k-th element;
6       Divide a[low : up] into n/r subsets of size r each;
7       Ignore excess elements;
8       Let m[i], 1 ≤ i ≤ (n/r) be the set of medians of
9       the above n/r subsets.
10      v := Select2(m, ⌈(n/r)/2⌉, 1, n/r);
11      Partition a[low : up] using v as the partition element;
12      Assume that v is at position j;
13      if (k = (j − low + 1)) then return v;
14      elseif (k < (j − low + 1)) then
15              return Select2(a, k, low, j − 1);
16          else return Select2(a, k − (j − low + 1), j + 1, up);
17  }
```

Algorithm 3.18 Selection pseudocode using the median of medians rule

$$T(n) \leq T(n/5) + T(3n/4) + cn \tag{3.9}$$

where c is chosen sufficiently large that

$$T(n) \leq cn \quad \text{for } n \leq 24$$

A proof by induction easily establishes that $T(n) \leq 20cn$ for $n \geq 1$. Algorithm Select2 with $r = 5$ is a linear time algorithm for the selection problem on distinct elements! The exercises examine other values of r that also yield this behavior. Let us now see what happens when the elements of $a[\,]$ are not all distinct. In this case, following a use of Partition (line 11), the size of S or R may be more than $.7n + 1.2$ as some elements equal to v may appear in both S and R. One way to handle the situation is to partition $a[\,]$ into three sets U, S, and R such that U contains all elements equal to v, S has all elements smaller than v, and R has the remainder. Lines 11 to 16 become:

```
Partition a[ ] into U, S, and R as above.
if (|S| ≥ k) then return Select2(a, k, low, low + |S| − 1);
else if ((|S| + |U|) ≥ k) then return v;
     else return Select2(a, k − |S| − |U|, low + |S| + |U|, up);
```

When this is done, the recurrence (3.9) is still valid as $|S|$ and $|R|$ are $\leq .7n + 1.2$. Hence, the new Select2 will be of linear complexity even when elements are not distinct.

Another way to handle the case of nondistinct elements is to use a different r. To see why a different r is needed, let us analyze Select2 with $r = 5$ and nondistinct elements. Consider the case when $.7n+1.2$ elements are less than v and the remaining elements are equal to v. An examination of Partition reveals that at most half the remaining elements may be in S. We can verify that this is the worst case. Hence, $|S| \leq .7n + 1.2 + (.3n - 1.2)/2 = .85n + .6$. Similarly, $|R| \leq .85n + .6$. Since, the total number of elements involved in the two recursive calls (in lines 10 and 15 or 16) is now $1.05n + .6 \geq n$, the complexity of Select2 is not $O(n)$. If we try $r = 9$, then at least $2.5\lfloor n/9 \rfloor$ elements will be less than or equal to v and at least this many will be greater than or equal to v. Hence, the size of S and R will be at most $n - 2.5\lfloor n/9 \rfloor + 1/2(2.5\lfloor n/9 \rfloor) = n - 1.25\lfloor n/9 \rfloor \leq 31/36n + 1.25 \leq 63n/72$ for $n \geq 90$. Hence, we obtain the recurrence

$$T(n) \leq \begin{cases} T(n/9) + T(63n/72) + c_1 n & n \geq 90 \\ c_1 n & n < 90 \end{cases}$$

where c_1 is a suitable constant. An inductive argument shows that $T(n) \leq 72c_1 n, n \geq 1$. Other suitable values of r are obtained in the exercises.

As far as the additional space needed by Select2 is concerned, we see that space is needed for the recursion stack. The recursive call from line 15 or 16 is easily eliminated as this call is the last statement executed in Select2. Hence, stack space is needed only for the recursion from line 10. The maximum depth of recursion is log n. The recursion stack should be capable of handling this depth. In addition to this stack space, space is needed only for some simple variables.

3.6.2 Implementation of Select2

Before attempting to write a pseudocode algorithm implementing Select2, we need to decide how the median of a set of size r is to be found and where we are going to store the $\lfloor n/r \rfloor$ medians of lines 8 and 9. Since, we expect to be using a small r (say $r = 5$ or 9), an efficient way to find the median of r elements is to sort them using InsertionSort(a, i, j). This algorithm is a modification of Algorithm 3.9 to sort $a[i : j]$. The median is now the middle element in $a[i : j]$. A convenient place to store these medians is at the front of the array. Thus, if we are finding the kth-smallest element in $a[low : up]$, then the elements can be rearranged so that the medians are $a[low]$, $a[low+1]$, $a[low+2]$, and so on. This makes it easy to implement line 10 as a selection on consecutive elements of $a[\,]$. Function Select2 (Algorithm 3.19) results from the above discussion and the replacement of the recursive calls of lines 15 and 16 by equivalent code to restart the algorithm.

```
1   Algorithm Select2(a, k, low, up)
2   // Return i such that a[i] is the kth-smallest element in
3   // a[low : up]; r is a global variable as described in the text.
4   {
5       repeat
6       {
7           n := up − low + 1; // Number of elements
8           if (n ≤ r) then
9           {
10              InsertionSort(a, low, up);
11              return low + k − 1;
12          }
13          for i := 1 to ⌊n/r⌋ do
14          {
15              InsertionSort(a, low + (i − 1) ∗ r, low + i ∗ r − 1);
16              // Collect medians in the front part of a[low : up].
17              Interchange(a, low + i − 1,
18                  low + (i − 1) ∗ r + ⌈r/2⌉ − 1);
19          }
20          j := Select2(a, ⌈⌊n/r⌋/2⌉, low, low + ⌊n/r⌋ − 1); // mm
21          Interchange(a, low, j);
22          j := Partition(a, low, up + 1);
23          if (k = (j − low + 1)) then return j;
24          else if (k < (j − low + 1)) then up := j − 1;
25              else
26              {
27                  k := k − (j − low + 1); low := j + 1;
28              }
29      } until (false);
30  }
```

Algorithm 3.19 Algorithm Select2

An alternative to moving the medians to the front of the array $a[low : up]$ (as in the Interchange statement within the **for** loop) is to delete this statement and use the fact that the medians are located at $low + (i-1)r + \lceil r/2 \rceil - 1, 1 \leq i \leq \lfloor n/r \rfloor$. Hence, Select2, Partition, and InsertionSort need to be rewritten to work on arrays for which the interelement distance is $b, b \geq 1$. At the start of the algorithm, all elements are a distance of one apart, i.e., $a[1], a[2], \ldots, a[n]$. On the first call of Select2 we wish to use only elements that are r apart starting with $a[\lceil r/2 \rceil]$. At the next level of recursion, the elements will be r^2 apart and so on. This idea is developed further in the exercises. We refer to arrays with an interelement distance of b as *b-spaced arrays.*

Algorithms Select1 (Algorithm 3.17) and Select2 (Algorithm 3.19) were implemented and run on a SUN Sparcstation 10/30. Table 3.8 summarizes the experimental results obtained. Times shown are in milliseconds. These algorithms were tested on random integers in the range [0, 1000] and the average execution times (over 500 input sets) were computed. Select1 outperforms Select2 on random inputs. But if the input is already sorted (or nearly sorted), Select2 can be expected to be superior to Select1.

n	1,000	2,000	3,000	4,000	5,000
Select1	7.42	23.50	30.44	39.24	52.36
Select2	49.54	104.02	174.54	233.56	288.64
n	6,000	7,000	8,000	9,000	10,000
Select1	70.88	83.14	95.00	101.32	111.92
Select2	341.34	414.06	476.98	532.30	604.40

Table 3.8 Comparison of Select1 and Select2 on random inputs

EXERCISES

1. Rewrite Select2, Partition, and InsertionSort using b-spaced arrays.

2. (a) Assume that Select2 is to be used only when all elements in a are distinct. Which of the following values of r guarantee $O(n)$ worst-case performance: $r = 3, 5, 7, 9$, and 11? Prove your answers.

   (b) Do you expect the computing time of Select2 to increase or decrease if a larger (but still eligible) choice for r is made? Why?

3. Do Exercise 2 for the case in which a is not restricted to distinct elements. Let $r = 7, 9, 11, 13$, and 15 in part (a).

4. Section 3.6 describes an alternative way to handle the situation when $a[\,]$ is not restricted to distinct elements. Using the partitioning element v, $a[\,]$ is divided into three subsets. Write algorithms corresponding to Select1 and Select2 using this idea. Using your new version of Select2 show that the worst-case computing time is $O(n)$ even when $r = 5$.

5. Determine optimal r values for worst-case and average performances of function Select2.

6. [Shamos] Let $x[1 : n]$ and $y[1 : n]$ contain two sets of integers, each sorted in nondecreasing order. Write an algorithm that finds the median of the $2n$ combined elements. What is the time complexity of your algorithm? (*Hint*: Use binary search.)

7. Let S be a (not necessarily sorted) sequence of n keys. A key k in S is said to be an *approximate median* of S if $|\{k' \in S : \ k' < k\}| \geq \frac{n}{4}$ and $|\{k' \in S : \ k' > k\}| \geq \frac{n}{4}$. Devise an $O(n)$ time algorithm to find all the approximate medians of S.

8. Input are a sequence S of n distinct keys, not necessarily in sorted order, and two integers m_1 and m_2 $(1 \leq m_1, m_2 \leq n)$. For any x in S, we define the *rank* of x in S to be $|\{k \in S : \ k \leq x\}|$. Show how to output all the keys of S whose ranks fall in the interval $[m_1, m_2]$ in $O(n)$ time.

9. The kth *quantiles* of an n-element set are the $k-1$ elements from the set that divide the sorted set into k equal-sized sets. Give an $O(n \log k)$ time algorithm to list the kth quantiles of a set.

10. Input is a (not necessarily sorted) sequence $S = k_1, k_2, \ldots, k_n$ of n arbitrary numbers. Consider the collection C of n^2 numbers of the form $\min\{k_i, k_j\}$, for $1 < i, j < n$. Present an $O(n)$-time and $O(n)$-space algorithm to find the median of C.

11. Given two vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$, $X < Y$ if there exists an $i, 1 \leq i \leq n$, such that $x_j = y_j$ for $1 \leq j < i$ and $x_i < y_i$. Given m vectors each of size n, write an algorithm that determines the minimum vector. Analyze the time complexity of your algorithm.

12. Present an $O(1)$ time Monte Carlo algorithm to find the median of an array of n numbers. The answer output should be correct with probability $\geq \frac{1}{n}$.

13. Input is an array $a[\ ]$ of n numbers. Present an $O(\log n)$ time Monte Carlo algorithm to output any member of $a[\ ]$ that is greater than or equal to the median. The answer should be correct with high probability. Provide a probability analysis.

14. Given a set X of n numbers, how will you find an element of X whose rank in X is at most $\frac{n}{f(n)}$, using a Monte Carlo algorithm? Your algorithm should run in time $O(f(n)\log n)$. Prove that the output will be correct with high probability.

15. In addition to Select1 and Select2, we can think of at least two more selection algorithms. The first of these is very straightforward and appears as Algorithm 3.20 (Algorithm Select3). The time complexity of Select3 is

$$O(n \min\{k, n-k+1\})$$

Hence, it is very fast for values of k close to 1 or close to n. In the worst case, its complexity is $O(n^2)$. Its average complexity is also $O(n^2)$.

Another selection algorithm proceeds by first sorting the n elements into nondecreasing order and then picking out the kth element. A complete sort can be avoided by using a minheap. Now, only k elements need to be removed from the heap. The time to set up the heap is $O(n)$. An additional $O(k \log n)$ time is needed to make k deletions. The total complexity is $O(n + k \log n)$. This basic algorithm can be improved further by using a maxheap when $k > n/2$ and deleting $n-k+1$ elements. The complexity is now $O(n + \log n \min\{k, n-k+1\})$. Call the resulting algorithm Select4. Now that we have four plausible selection algorithms, we would like to know which is best. On the basis of the asymptotic analyses of the four selection algorithms, we can make the following qualitative statements about our expectations on the relative performance of the four algorithms.

- Because of the overhead involved in Select1, Select2, and Select4 and the relative simplicity of Select3, Select3 will be fastest both on the average and in the worst case for small values of n. It will also be fastest for large n and very small or very large k, for example, $k = 1, 2, n$, or $n-1$.
- For larger values of n, Select1 will have the best behavior on the average.
- As far as worst-case behavior is concerned, Select2 will out-perform the others when n is suitably large. However, there will probably be a range of n for which Select4 will be faster than both Select2 and Select3. We expect this because of the relatively large

```
1   Algorithm Select3(a, n, k)
2   // Rearrange a[ ] such that a[k] is the k-th smallest.
3   {
4       if (k ≤ ⌊n/2⌋) then
5           for i := 1 to k do
6           {
7               q := i; min := a[i];
8               for j := i + 1 to n do
9                   if (a[j] < min) then
10                  {
11                      q := j; min := a[j];
12                  }
13              Interchange(a, q, i);
14          }
15      else
16          for i := n to k  step −1 do
17          {
18              q := i; max := a[i];
19              for j := (i − 1) to 1  step −1 do
20                  if (a[j] > max) then
21                  {
22                      q := j; max := a[j];
23                  }
24              Interchange(a, q, i);
25          }
26  }
```

Algorithm 3.20 Straightforward selection algorithm

overhead in Select2 (i.e., the constant term in $O(n)$ is relatively large).

- As a result of the above assertions, it is desirable to obtain composite algorithms for good average and worst-case performances. The composite algorithm for good worst-case performance will have the form of function Select2 but will include the following after the first **if** statement.

```
if (n < c1) then return Select3(a, m, p, k);
else if (n < c2) then return Select4(a, m, p, k);
```

Since the overheads in Select1 and Select4 are about the same, the constants associated with the average computing times will be about

the same. Hence, Select1 may always be better than Select4 or there may be a small c_3 such that Select4 is better than Select1 for $n < c_3$. In any case, we expect there is a $c_4, c_4 > 0$, such that Select3 is faster than Select1 on the average for $n < c_4$.

To verify the preceding statements and determine c_1, c_2, c_3, and c_4, it is necessary to program the four algorithms in some programming language and run the four corresponding programs on a computer. Once the programs have been written, test data are needed to determine average and worst-case computing times. So, let us now say something about the data needed to obtain computing times from which $c_i, 1 \leq i \leq 4$, can be determined. Since we would also like information regarding the average and worst-case computing times of the resulting composite algorithms, we need test data for this too. We limit our testing to the case of distinct elements.

To obtain worst-case computing times for Select1, we change the algorithm slightly. This change will not affect its worst-case computing time but will enable us to use a rather simple data set to determine this time for various values of n. We dispense with the random selection rule for Partition and instead use $a[m]$ as the partitioning element. It is easy to see that the worst-case time is obtained with $a[i] = i$, $1 \leq i \leq n$, and $k = n$. As far as the average time for any given n is concerned, it is not easy to arrive at one data set and a k that exhibits this time. On the other hand, trying out all $n!$ different input permutations and $k = 1, 2, \ldots, n$ for each of these is not a feasible way to find the average. An approximation to the average computing time can be obtained by trying out a few (say ten) random permutations of the numbers $1, 2, \ldots, n$ and for each of these using a few (say five) random values of k. The average of the times obtained can be used as an approximation to the average computing time. Of course, using more permutations and more k values results in a better approximation. However, the number of permutations and k values we can use is limited by the amount of computational resources (in terms of time) we have available.

For Select2, the average time can be obtained in the same way as for Select1. For the worst-case time we can either try to figure out an input permutation for which the number of elements less than the median of medians is always as large as possible and then use $k = 1$. A simpler approach is to find just an approximation to the worst-case time. This can be done by taking the max of the computing times for all the tests used to obtain the average computing time. Since the computing times for Select2 vary with r, it is first necessary to determine an r that yields optimum behavior. Note that the r's for optimum average and worst-case behaviors may be different.

We can verify that the worst-case data for Select3 are $a[i] = n + 1 - i$, for $1 \leq i \leq n$, and $k = \frac{n}{2}$. The computing time for Select3 is relatively insensitive to the input permutation. This permutation affects only the number of times the second **if** statement of Algorithm 3.20 is executed. On the average, this will be done about half the time. This can be achieved by using $a[i] = n + 1 - i$, $1 \leq i \leq n/2$, and $a[i] = n + 1$, $n/2 < i \leq n$. The k value needed to obtain the average computing time is readily seen to be $n/4$.

(a) What test data would you use to determine worst-case and average times for Select4?

(b) Use the ideas above to obtain a table of worst-case and average times for Select1, Select2, Select3, and Select4.

16. Program Select1 and Select3. Determine when algorithm Select1 becomes better than Select3 on the average and also when Select2 better than Select3 for worst-case performance.

17. [Project] Program the algorithms of Exercise 4 as well as Select3 and Select4. Carry out a complete test along the lines discussed in Exercise 15. Write a detailed report together with graphs explaining the data sets, test strategies, and determination of $c_1, \ldots, c_4$. Write the final composite algorithms and give tables of computing times for these algorithms.

3.7 STRASSEN'S MATRIX MULTIPLICATION

Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix whose i, jth element is formed by taking the elements in the ith row of A and the jth column of B and multiplying them to get

$$C(i,j) = \sum_{1 \leq k \leq n} A(i,k)B(k,j) \tag{3.10}$$

for all i and j between 1 and n. To compute $C(i,j)$ using this formula, we need n multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional method is $\Theta(n^3)$.

The divide-and-conquer strategy suggests another way to compute the product of two $n \times n$ matrices. For simplicity we assume that n is a power of 2, that is, that there exists a nonnegative integer k such that $n = 2^k$. In case n is not a power of two, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two

(see the exercises for more on this subject). Imagine that A and B are each partitioned into four square submatrices, each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$. Then the product AB can be computed by using the above formula for the product of 2×2 matrices: if AB is

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \tag{3.11}$$

then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{3.12}$$

If $n = 2$, then formulas (3.11) and (3.12) are computed using a multiplication operation for the elements of A and B. These elements are typically floating point numbers. For $n > 2$, the elements of C can be computed using *matrix* multiplication and addition operations applied to matrices of size $n/2 \times n/2$. Since n is a power of 2, these matrix products can be recursively computed by the same algorithm we are using for the $n \times n$ case. This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitably small ($n = 2$) so that the product is computed directly.

To compute AB using (3.12), we need to perform eight multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Since two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c, the overall computing time $T(n)$ of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

This recurrence can be solved in the same way as earlier recurrences to obtain $T(n) = O(n^3)$. Hence no improvement over the conventional method has been made. Since matrix multiplications are more expensive than matrix additions ($O(n^3)$ versus $O(n^2)$), we can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the C_{ij}'s of (3.12) using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven $n/2 \times n/2$ matrices P, Q, R, S, T, U, and V as in (3.13). Then the C_{ij}'s are computed using the formulas in (3.14). As can be seen, P, Q, R, S, T, U, and V can be computed using 7 matrix multiplications and 10 matrix additions or subtractions. The C_{ij}'s require an additional 8 additions or subtractions.

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned} \tag{3.13}$$

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned} \tag{3.14}$$

The resulting recurrence relation for $T(n)$ is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases} \tag{3.15}$$

where a and b are constants. Working with this formula, we get

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n}, \ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

EXERCISES

1. Verify by hand that Equations 3.13 and 3.14 yield the correct values for C_{11}, C_{12}, C_{21}, and C_{22}.
2. Write an algorithm that multiplies two $n \times n$ matrices using $O(n^3)$ operations. Determine the precise number of multiplications, additions, and array element accesses.
3. If k is a nonnegative constant, then prove that the recurrence

$$T(n) = \begin{cases} k & n = 1 \\ 3T(n/2) + kn & n > 1 \end{cases} \tag{3.16}$$

has the following solution (for n a power of 2):

$$T(n) = 3kn^{\log_2 3} - 2kn \tag{3.17}$$

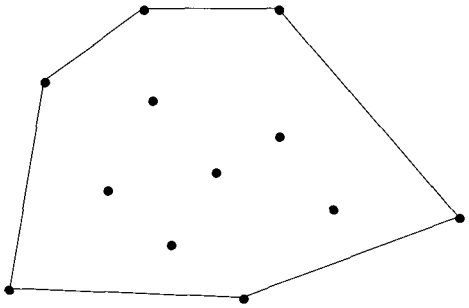

Figure 3.6 Convex hull: an example

(1) obtain the vertices of the convex hull (these vertices are also called *extreme points*), and (2) obtain the vertices of the convex hull in some order (clockwise, for example).

Here is a simple algorithm for obtaining the extreme points of a given set S of points in the plane. To check whether a particular point $p \in S$ is extreme, look at each possible triplet of points and see whether p lies in the triangle formed by these three points. If p lies in any such triangle, it is not extreme; otherwise it is. Testing whether p lies in a given triangle can be done in $\Theta(1)$ time (using the methods described in Section 3.8.1). Since there are $\Theta(n^3)$ possible triangles, it takes $\Theta(n^3)$ time to determine whether a given point is an extreme point or not. Since there are n points, this algorithm runs in a total of $\Theta(n^4)$ time.

Using divide-and-conquer, we can solve both versions of the convex hull problem in $\Theta(n \log n)$ time. We develop three algorithms for the convex hull in this section. The first has a worst-case time of $\Theta(n^2)$ whereas its average time is $\Theta(n \log n)$. This algorithm has a divide-and-conquer structure similar to that of QuickSort. The second has a worst-case time complexity of $\Theta(n \log n)$ and is not based on divide-and-conquer. The third algorithm is based on divide-and-conquer and has a time complexity of $\Theta(n \log n)$ in the worst case. Before giving further details, we digress to discuss some primitive geometric methods that are used in the convex hull algorithms.

3.8.1 Some Geometric Primitives

Let A be an $n \times n$ matrix whose elements are $\{a_{ij}\}$, $1 \leq i, j \leq n$. The ijth *minor* of A, denoted as A_{ij}, is defined to be the submatrix of A obtained by deleting the ith row and jth column. The *determinant* of A, denoted

$\det(A)$, is given by

$$\det(A) = \begin{cases} a_{11} & n = 1 \\ a_{11}\ \det(A_{11}) - a_{12}\ \det(A_{12}) + \cdots + (-1)^{n-1}\ \det(A_{1n}) & n > 1 \end{cases}$$

Consider the directed line segment $\langle p_1, p_2 \rangle$ from some point $p_1 = (x_1, y_1)$ to some other point $p_2 = (x_2, y_2)$. If $q = (x_3, y_3)$ is another point, we say q *is to the left (right) of* $\langle p_1, p_2 \rangle$ if the angle $p_1 p_2 q$ is a left (right) turn. [An angle is said to be a left (right) turn if it is less than or equal to (greater than or equal to) 180°.] We can check whether q is to the left (right) of $\langle p_1, p_2 \rangle$ by evaluating the determinant of the following matrix:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}$$

If this determinant is positive (negative), then q is to the left (right) of $\langle p_1, p_2 \rangle$. If this determinant is zero, the three points are colinear. This test can be used, for example, to check whether a given point p is within a triangle formed by three points, say p_1, p_2, and p_3 (in clockwise order). The point p is within the triangle iff p is to the right of the line segments $\langle p_1, p_2 \rangle$, $\langle p_2, p_3 \rangle$, and $\langle p_3, p_1 \rangle$.

Also, for any three points $(x_1, y_1), (x_2, y_2)$, and (x_3, y_3), the *signed area* formed by the corresponding triangle is given by one-half of the above determinant.

Let $p_1, p_2, \ldots, p_n$ be the vertices of the convex polygon Q in clockwise order. Let p be any other point. It is desired to check whether p lies in the interior of Q or outside. Consider a horizontal line h that extends from $-\infty$ to ∞ and goes through p. There are two possibilities: (1) h does not intersect any of the edges of Q, (2) h intersects some of the edges of Q. If case (1) is true, then, p is outside Q. In case (2), there can be at most two points of intersection. If h intersects Q at a single point, it is counted as two. Count the number of points of intersections that are to the left of p. If this number is even, then p is external to Q; otherwise it is internal to Q. This method of checking whether p is interior to Q takes $\Theta(n)$ time.

3.8.2 The QuickHull Algorithm

An algorithm that is similar to QuickSort can be devised to compute the convex hull of a set X of n points in the plane. This algorithm, called QuickHull, first identifies the two points (call them p_1 and p_2) of X with the smallest and largest x-coordinate values. Assume now that there are no ties. Later we see how to handle ties. Both p_1 and p_2 are extreme points and part of the convex hull. The set X is divided into X_1 and X_2 so that

X_1 has all the points to the left of the line segment $\langle p_1, p_2 \rangle$ and X_2 has all the points to the right of $\langle p_1, p_2 \rangle$. Both X_1 and X_2 include the two points p_1 and p_2. Then, the convex hulls of X_1 and X_2 (called the *upper hull* and *lower hull*, respectively) are computed using a divide-and-conquer algorithm called Hull. The union of these two convex hulls is the overall convex hull.

If there is more than one point with the smallest x-coordinate, let p_1' and p_1'' be the points from among these with the least and largest y-coordinates, respectively. Similarly define p_2' and p_2'' for the points with the largest x-coordinate values. Now X_1 will be all the points to the left of $\langle p_1'', p_2'' \rangle$ (including p_1'' and p_2'') and X_2 will be all the points to the right of $\langle p_1', p_2' \rangle$ (including p_1' and p_2'). In the rest of the discussion we assume for simplicity that there are no ties for p_1 and p_2. Appropriate modifications are needed in the event of ties.

We now describe how Hull computes the convex hull of X_1. We determine a point of X_1 that belongs to the convex hull of X_1 and use it to partition the problem into two independent subproblems. Such a point is obtained by computing the area formed by p_1, p, and p_2 for each p in X_1 and picking the one with the largest (absolute) area. Ties are broken by picking the point p for which the angle pp_1p_2 is maximum. Let p_3 be that point.

Now X_1 is divided into two parts; the first part contains all the points of X_1 that are to the left of $\langle p_1, p_3 \rangle$ (including p_1 and p_3), and the second part contains all the points of X_1 that are to the left of $\langle p_3, p_2 \rangle$ (including p_3 and p_2) (see Figure 3.7). There cannot be any point of X_1 that is to the left of both $\langle p_1, p_3 \rangle$ and $\langle p_3, p_2 \rangle$. Also, all the other points are interior points and can be dropped from future consideration. The convex hull of each part is computed recursively, and the two convex hulls are merged easily by placing one next to the other in the right order.

If there are m points in X_1, we can identify the point of division p_3 in time $O(m)$. Partitioning X_1 into two parts can also be done in $O(m)$ time. Merging the two convex hulls can be done in time $O(1)$. Let $T(m)$ stand for the run time of Hull on a list of m points and let m_1 and m_2 denote the sizes of the two resultant parts. Note that $m_1 + m_2 \leq m$. The recurrence relation for $T(m)$ is $T(m) = T(m_1) + T(m_2) + O(m)$, which is similar to the one for the run time of QuickSort. The worst-case run time is thus $O(m^2)$ on an input of m points. This happens when the partitioning at each level of recursion is highly uneven.

If the partitioning is nearly even at each level of recursion, then the run time will equal $O(m \log m)$ as in the case of QuickSort. Thus the average run time of QuickHull is $O(n \log n)$, on an input of size n, under appropriate assumptions on the input distribution.

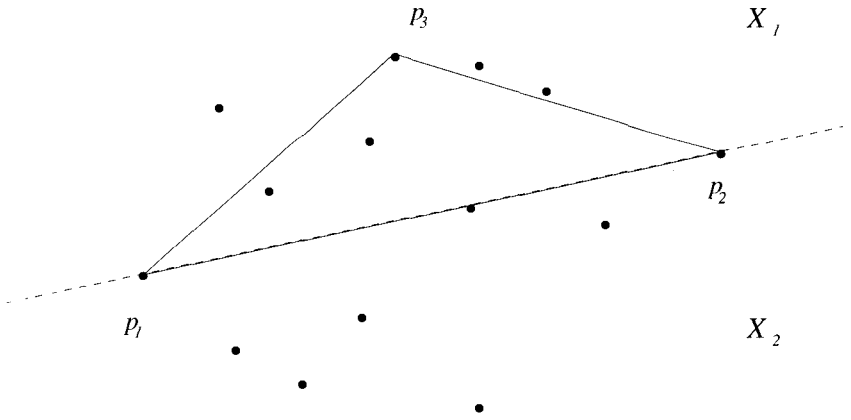

Figure 3.7 Identifying a point on the convex hull of X_1

3.8.3 Graham's Scan

If S is a set of points in the plane, Graham's scan algorithm identifies the point p from S with the lowest y-coordinate value (ties are broken by picking the leftmost among these). It then sorts the points of S according to the angle subtended by the points and p with the positive x-axis. Figure 3.8 gives an example. After having sorted the points, if we scan through the sorted list starting at p, every three successive points will form a left turn if all of these points lie on the hull. On the other hand if there are three successive points, say p_1, p_2, and p_3, that form a right turn, then we can immediately eliminate p_2 since it cannot lie on the convex hull. Notice that it will be an internal point because it lies within the triangle formed by p, p_1, and p_3.

We can eliminate all the interior points using the above procedure. Starting from p, we consider three successive points p_1, p_2, and p_3 at a time. To begin with, $p_1 = p$. If these points form a left turn, we move to the next point in the list (that is, we set $p_1 = p_2$, and so on). If these three points form a right turn, then p_2 is deleted since it is an interior point. We move one point behind in the list by setting p_1 equal to its predecessor. This process of scanning ends when we reach the point p again.

Example 3.11 In Figure 3.8, the first three points looked at are $p, 1$, and 2. Since these form a left turn, we move to $1, 2$, and 3. These form a right turn and hence 2 is deleted. Next, the three points $p, 1$, and 3 are considered. These form a left turn and hence the pointer is moved to point 1. The points

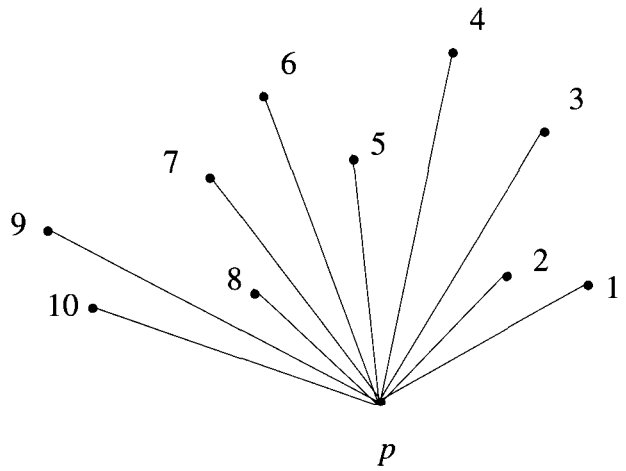

Figure 3.8 Graham's scan algorithm sorts the points first

$1, 3$, and 4 also form a left turn, and the scan proceeds to $3, 4$, and 5 and then to $4, 5$, and 6. Now point 5 gets deleted. The triplets $3, 4, 6$; $4, 6, 7$; and $6, 7, 8$ form left turns whereas the next triplet $7, 8, 9$ forms a right turn. Therefore, 8 gets deleted and in the next round 7 also gets eliminated. The next three triplets examined are $4, 6, 9$; $6, 9, 10$; and $9, 10, p$, all of which are left turns. The final hull obtained is $p, 1, 3, 4, 6, 9$, and 10, which are points on the hull in counterclockwise (ccw) order. □

This scan process is given in Algorithm 3.21. In this algorithm the set of points is realized as a doubly linked list *ptslist*. Function Scan runs in $O(n)$ time since for each triplet examined, either the scan moves one node ahead or one point gets removed. In the latter case, the scan moves one node back. Also note that for each triplet, the test as to whether a left or right turn is formed can be done in $O(1)$ time. Function Area computes the signed area formed by three points. The major work in the algorithm is in sorting the points. Since sorting takes $O(n \log n)$ time, the total time of Graham's scan algorithm is $O(n \log n)$.

3.8.4 An $O(n \log n)$ Divide-and-Conquer Algorithm

In this section we present a simple divide-and-conquer algorithm, called DCHull, which also takes $O(n \log n)$ time and computes the convex hull in clockwise order.

```
point = record{
    float x; float y;
    point *prev; point *next;
};

1   Algorithm Scan(list)
2   // list is a pointer to the first node in the input list.
3   {
4       *p := list; *p1 := list;
5       repeat
6       {
7           p2 := (p1 → next);
8           if ((p2 → next) ≠ 0) then p3 := (p2 → next);
9           else return; // End of the list
10          temp := Area((p1 → x), (p1 → y), (p2 → x),
11              (p2 → y), (p3 → x), (p3 → y));
12          if (temp ≥ 0.0) then p1 := (p1 → next);
13          // If p1, p2, p3 form a left turn, move one point ahead;
14          // If not, delete p2 and move back.
15          else
16          {
17              (p1 → next) := p3; (p3 → prev) := p1; delete p2;
18              p1 := (p1 → prev);
19          }
20      } until (false);
21  }

1   Algorithm ConvexHull(ptslist)
2   {
3       // ptslist is a pointer to the first item of the input list. Find
4       // the point p in ptslist of lowest y-coordinate. Sort the
5       // points according to the angle made with p and the x-axis.

6       Sort(ptslist); Scan(ptslist); PrintList(ptslist);
7   }
```

Algorithm 3.21 Graham's scan algorithm

Given a set X of n points, like that in the case of QuickHull, the problem is reduced to finding the upper hull and the lower hull separately and then putting them together. Since the computations of the upper and lower hulls are very similar, we restrict our discussion to computing the upper hull. The divide-and-conquer algorithm for computing the upper hull partitions X into two nearly equal halves. Partitioning is done according to the x-coordinate values of points using the median x-coordinate as the splitter (see Section 3.6 for a discussion on median finding). Upper hulls are recursively computed for the two halves. These two hulls are then merged by finding the *line of tangent* (i.e., a straight line connecting a point each from the two halves, such that all the points of X are on one side of the line) (see Figure 3.9).

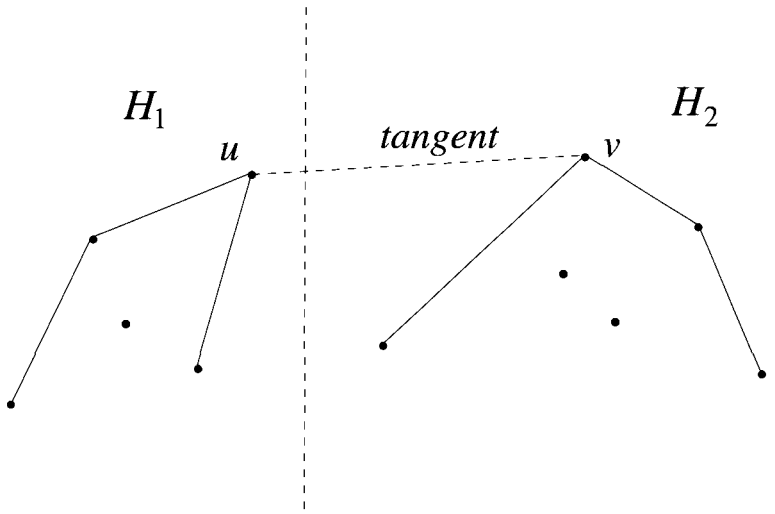

Figure 3.9 Divide and conquer to compute the convex hull

To begin with, the points p_1 and p_2 are identified [where p_1 (p_2) is the point with the least (largest) x-coordinate value]. This can be done in $O(n)$ time. Ties can be handled in exactly the same manner as in QuickHull. So, assume that there are no ties. All the points that are to the left of the line segment $\langle p_1, p_2 \rangle$ are separated from those that are to the right. This separation also can be done in $O(n)$ time. From here on, by "input" and "X" we mean all the points that are to the left of the line segment $\langle p_1, p_2 \rangle$. Also let $|X| = N$.

Sort the input points according to their x-coordinate values. Sorting can be done in $O(N \log N)$ time. This sorting is done only once in the computation of the upper hull. Let $q_1, q_2, \ldots, q_N$ be the sorted order of these

points. Now partition the input into two equal halves with $q_1, q_2, \ldots, q_{N/2}$ in the first half and $q_{N/2+1}, q_{N/2+2}, \ldots, q_N$ in the second half. The upper hull of each half is computed recursively. Let H_1 and H_2 be the upper hulls. Upper hulls are maintained as linked lists in clockwise order. We refer to the first element in the list as the leftmost point and the last element as the rightmost point.

The line of tangent is then found in $O(\log^2 N)$ time. If (u, v) is the line of tangent, then all the points of H_1 that are to the right of u are dropped. Similarly, all the points that are to the left of v in H_2 are dropped. The remaining part of H_1, the line of tangent, and the remaining part of H_2 form the upper hull of the given input set.

If $T(N)$ is the run time of the above recursive algorithm for the upper hull on an input of N points, then we have

$$T(N) = 2T(N/2) + O(\log^2 N)$$

which solves to $T(N) = O(N)$. Thus the run time is dominated by the initial sorting step.

The only part of the algorithm that remains to be specified is how to find the line of tangent $\langle u, v\rangle$ in $O(\log^2 N)$ time. The way to find the tangent is to start from the middle point, call it p, of H_1. Here the middle point refers to the middle element of the corresponding list. Find the tangent of p with H_2. Let $\langle p, q\rangle$ be the tangent. Using $\langle p, q\rangle$, we can determine whether u is to the left of, equal to, or to the right of p in H_1. A binary search in this fashion on the points of H_1 reveals u. Use a similar procedure to isolate v.

Lemma 3.1 Let H_1 and H_2 be two upper hulls with at most m points each. If p is any point of H_1, its point q of tangency with H_2 can be found in $O(\log m)$ time.

Proof. If q' is any point in H_2, we can check whether q' is to the left of, equal to, or to the right of q in $O(1)$ time (see Figure 3.10). In Figure 3.10, x and y are the left and right neighbors of q' in H_2, respectively. If $\angle pq'x$ is a right turn and $\angle pq'y$ is a left turn, then q is to the right of q' (see case 1 of Figure 3.10). If $\angle pq'x$ and $\angle pq'y$ are both right turns, then $q' = q$ (see case 2 of Figure 3.10); otherwise q is to the left of q' (see case 3 of Figure 3.10). Thus we can perform a binary search on the points of H_2 and identify q in $O(\log m)$ time. □

Lemma 3.2 If H_1 and H_2 are two upper hulls with at most m points each, their common tangent can be computed in $O(\log^2 m)$ time.

Proof. Let $u \in H_1$ and $v \in H_2$ be such that $\langle u, v\rangle$ is the line of tangent. Also let p be an arbitrary point of H_1 and let $q \in H_2$ be such that $\langle p, q\rangle$ is a

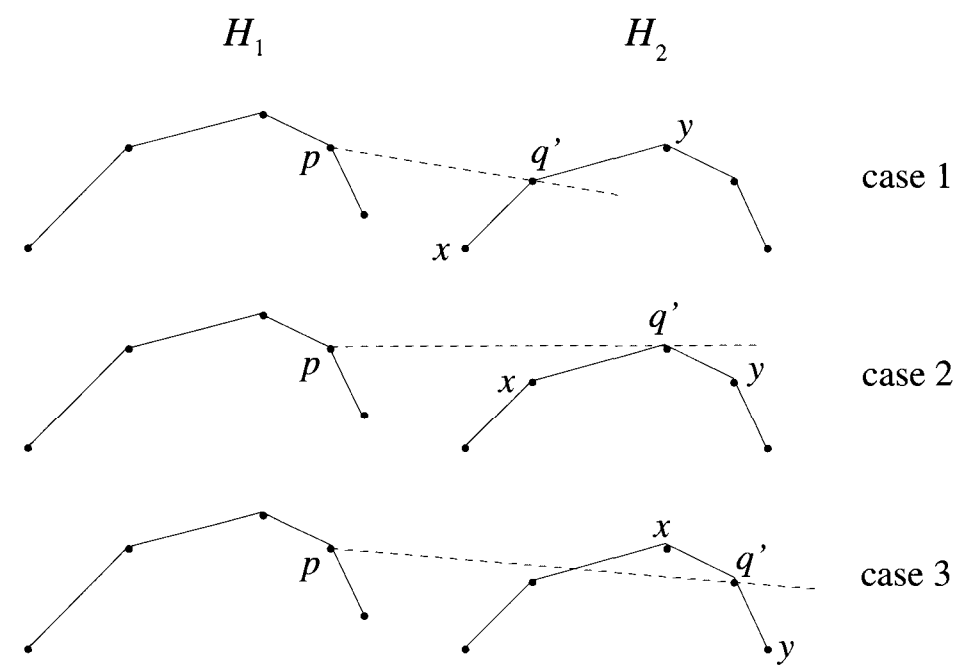

Figure 3.10 Proof of Lemma 3.1

tangent of H_2. Given p and q, we can check in $O(1)$ time whether u is to the left of, equal to, or to the right of p (see Figure 3.11). Here x and y are left and right neighbors, respectively, of p in H_1. If $\langle p, q \rangle$ is also tangential to H_1, then $p = u$. If $\angle xpq$ is a left turn, then u is to the left of p; otherwise u is to the right of p. This suggests a binary search for u. For each point p of H_1 chosen, we have to determine the tangent from p to H_2 and then decide the relative positioning of p with respect to u. We can do this computation in $O(\log m \times \log m) = O(\log^2 m)$ time. □

In summary, given two upper hulls with $\frac{N}{2}$ points each, the line of tangent can be computed in $O(\log^2 N)$ time.

Theorem 3.4 A convex hull of n points in the plane can be computed in $O(n \log n)$ time. □

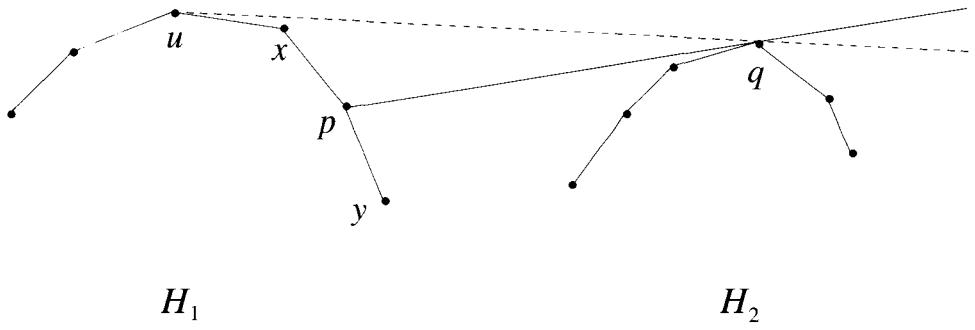

Figure 3.11 Proof of Lemma 3.2

EXERCISES

1. Write an algorithm in pseudocode that implements QuickHull and test it using suitable data.

2. Code the divide-and-conquer algorithm DCHull and test it using appropriate data.

3. Run the three algorithms for convex hull discussed in this section on various random inputs and compare their performances.

4. Algorithm DCHull can be modified as follows: Instead of using the median as the splitter, we could use a randomly chosen point as the splitter. Then X is partitioned into two around this point. The rest of the function DCHull is the same. Write code for this modified algorithm and compare it with DCHull empirically.

5. Let S be a set of n points in the plane. It is given that there is only a constant (say c) number of points on the hull of S. Can you devise a convex hull algorithm for S that runs in time $o(n \log n)$? Conceive of special algorithms for $c = 3$ and $c = 4$ first and then generalize.

3.9 REFERENCES AND READINGS

Algorithm MaxMin (Algorithm 3.6) is due to I. Pohl and the quicksort algorithm (Algorithm 3.13) is due to C. A. R. Haore. The randomized sorting algorithm in Algorithm 3.16 is due to W. D. Frazer and A. C. McKeller and

the selection algorithm of Algorithm 3.19 is due to M. Blum, R. Floyd, V. Pratt, R. Rivest and R. E. Tarjan.

For more on randomized sorting and selection see:

"Expected time bounds for selection," by R. Floyd and R. Rivest, *Communications of the ACM* 18, no. 3 (1975): 165–172.

"Samplesort: A Sampling Approach to Minimal Storage Tree Sorting," by W. D. Frazer and A. C. McKellar, *Journal of the ACM* 17, no. 3 (1970): 496–507.

"Derivation of Randomized Sorting and Selection Algorithms," by S. Rajasekaran and J. H. Reif, in *Parallel Algorithm Derivation and Program Transformation*, edited by R. Paige, J. H. Reif, and R. Wachter, Kluwer Academic Publishers, 1993, pp. 187–205.

The matrix multiplication algorithm in Section 3.7 is due to V. Strassen. For more information on the matrix multiplication problem see "Matrix multiplication via arithmetic progressions," by D. Coppersmith and S. Winograd, *Journal of Symbolic Computation* 9 (1990): 251–280. A complex $O(n^{2.376})$ time algorithm for multiplying two $n \times n$ matrices is given in this paper.

For more applications of divide-and-conquer see:

Computational Geometry, by F. Preparata and M. I. Shamos, Springer-Verlag, 1985.

Computational Geometry: An Introduction Through Randomized Algorithms by K. Mulmuley, Prentice-Hall, 1994.

Introduction to Algorithms: A Creative Approach, by U. Manber, Addison-Wesley, 1989.

3.10 ADDITIONAL EXERCISES

1. What happens to the worst-case run time of quicksort if we use the median of the given keys as the splitter key? (Assume that the selection algorithm of Section 3.6 is employed to determine the median).

2. The sets A and B have n elements each given in the form of sorted arrays. Present an $O(n)$ time algorithm to compute $A \cup B$ and $A \cap B$.

3. The sets A and B have m and n elements (respectively) from a linear order. These sets are not necessarily sorted. Also assume that $m \leq n$. Show how to compute $A \cup B$ and $A \cap B$ in $O(n \log m)$ time.

4. Consider the problem of sorting a sequence X of n keys where each key is either zero or one (i.e., each key is a bit). One way of sorting

X is to start with two empty lists L_0 and L_1. Let $X = k_1, k_2, \ldots, k_n$. For each $1 \leq i \leq n$ do: If $k_i = 0$, then append k_i to L_0. If $k_i = 1$, then append k_i to L_1. After processing all the keys of X in this manner, output the list L_0 followed by the list L_1.

The above idea of sorting can be extended to the case in which each key is of length more than one bit. In particular, if the keys are integers in the range $[0, m-1]$, then we start with m empty lists, $L_0, L_1, \ldots, L_{m-1}$, one list (or *bucket*) for each possible value that a key can take. Then the keys are processed in a similar fashion. In particular, if a key has a value ℓ, then it will be appended to the ℓth list.

Write an algorithm that employs this idea to sort n keys assuming that each key is in the range $[0, m-1]$. Show that the run time of your algorithm is $O(n+m)$. This algorithm is known as the *bucket sort*.

5. Consider the problem of sorting n two-digit integers. The idea of *radix sort* can be employed. We first sort the numbers only with respect to their least significant digits (LSDs). Followed by this, we apply a sort with respect to their second LSDs. More generally, d-digit numbers can be sorted in d phases, where in the ith phase ($1 \leq i \leq d$) we sort the keys only with respect to their ith LSDs. Will this algorithm always work?

   As an example, let the input be $k_1 = 12, k_2 = 45, k_3 = 23, k_4 = 14, k_5 = 32$, and $k_6 = 57$. After sorting these keys with respect to their LSDs, we end up with: $k_5 = 32, k_1 = 12, k_3 = 23, k_4 = 14, k_2 = 45$, and $k_6 = 57$. When we sort the resultant sequence with respect to the keys' second LSDs (i.e., the next-most significant digits), we get $k_1 = 12, k_4 = 14, k_3 = 23, k_5 = 32, k_2 = 45$, and $k_6 = 57$, which is the correct answer!

   But note that in the second phase of the algorithm, $k_4 = 14, k_1 = 12, k_3 = 23, k_5 = 32, k_2 = 45, k_6 = 57$ is also a valid sort with respect to the second LSDs. The result in any phase of radix sorting can be forced to be correct by enforcing the following condition on the sorting algorithm to be used. "Keys with equal values should remain in the same relative order in the output as they were in the input." Any sorting algorithm that satisfies this is called a *stable sort*.

   Note that in the above example, if the algorithm used to sort the keys in the second phase is stable, then the output will be correct. In summary, radix sort can be employed to sort d-digit numbers in d phases such that the sort applied in each phase (except the first phase) is stable.

   More generally, radix sort can be used to sort integers of arbitrary length. As usual, the algorithm will consist of phases in each of which the keys are sorted only with respect to certain parts of their keys.

The parts used in each phase could be single bits, single digits, or more generally, ℓ bits, for some appropriate ℓ.

In Exercise 4, you showed that n integers in the range $[0, m-1]$ can be sorted in $O(n+m)$ time. Is your algorithm stable? If not, make it stable. As a special case, your algorithm can sort n integers in the range $[0, n-1]$ in $O(n)$ time. Use this algorithm together with the idea of radix sorting to develop an algorithm that can sort n integers in the range $[0, n^c - 1]$ (for any fixed c) in $O(n)$ time.

6. Two sets A and B have n elements each. Assume that each element is an integer in the range $[0, n^{100}]$. These sets are not necessarily sorted. Show how to check whether these two sets are disjoint in $O(n)$ time. Your algorithm should use $O(n)$ space.

7. Input are the sets $S_1, S_2, \ldots,$ and S_ℓ (where $\ell \leq n$). Elements of these sets are integers in the range $[0, n^c - 1]$ (for some fixed c). Also let $\sum_{i=1}^{\ell} |S_i| = n$. The goal is to output S_1 in sorted order, then S_2 in sorted order, and so on. Present an $O(n)$ time algorithm for this problem.

8. Input is an array of n numbers where each number is an integer in the range $[0, N]$ (for some $N >> n$). Present an algorithm that runs in the worst case in time $O\left(n \frac{\log N}{\log n}\right)$ and checks whether all these n numbers are distinct. Your algorithm should use only $O(n)$ space.

9. Let S be a sequence of n^2 integers in the range $[1, n]$. Let $R(i)$ be the number of i's in the sequence (for $i = 1, 2, \ldots, n$). Given S, we have to compute an approximate value of $R(i)$ for each i. If $N(i)$ is an approximation to $R(i), i = 1, \ldots, n$, it should be the case that (with high probability) $N(i) \geq R(i)$ for each i and $\sum_{i=1}^{n} N(i) = O(n^2)$. Of course we can do this computation in deterministic $O(n^2)$ time. Design a randomized algorithm for this problem that runs in time $O(n \log^{O(1)} n)$.