	(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 non-decreasing 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 $x_1 \leq x_2 \leq \cdots \leq x_m$ and $x_{m+1} \leq \cdots \leq x_n$, 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

```
Algorithm Partition($a, m, p$)
// Within $a[m], a[m+1], \ldots, a[p-1]$ the elements are
// rearranged in such a manner that if initially $t = a[m]$,
// then after completion $a[q] = t$ for some $q$ between $m$
// and $p-1$, $a[k] \leq t$ for $m \leq k < q$, and $a[k] \geq t$
// for $q < k < p$. $q$ is returned. Set $a[p] = \infty$.
{
    $v := a[m]$; $i := m$; $j := p$;
    repeat
    {
        repeat
            $i := i + 1$;
        until ($a[i] \geq v$);

        repeat
            $j := j - 1$;
        until ($a[j] \leq v$);

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

    } until ($i \geq j$);

    $a[m] := a[j]$; $a[j] := v$; return $j$;
}

Algorithm Interchange($a, i, j$)
// Exchange $a[i]$ with $a[j]$.
{
    $p := a[i]$;
    $a[i] := a[j]$; $a[j] := p$;
}
```

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.

```
Algorithm QuickSort($p, q$)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1 : n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1 : n]$.
{
    if ($p < q$) then  // If there are more than one element
    {
        // divide $P$ into two subproblems.
            $j$ := Partition($a, p, q + 1$);
                // $j$ is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort($p, j - 1$);
            QuickSort($j + 1, q$);
        // There is no need for combining solutions.
    }
}
```

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, $1 \le i \le p - m$, 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 $1/(p - m), m \le j < p$. From this we obtain the recurrence

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

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

$$\begin{array}{rcl} nC_A(n) - (n-1)C_A(n-1) & = & 2n + 2C_A(n-1) \\ & \mathit{or} & \\ C_A(n)/(n+1) & = & C_A(n-1)/n + 2/(n+1) \end{array}$$

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

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

Since

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

(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

```
Algorithm QuickSort2($p, q$)
// Sorts the elements in $a[p : q]$.
{
    // stack is a stack of size $2\log(n)$.
    repeat
    {
        while ($p < q$) do
        {
            $j$ := Partition($a, p, q + 1$);
            if ($(j - p) < (q - j)$) then
            {
                Add($j + 1$); // Add $j + 1$ to stack.
                Add($q$); $q := j - 1$; // Add $q$ to stack
            }
            else
            {
                Add($p$); // Add $p$ to stack.
                Add($j - 1$); $p := j + 1$; // Add $j - 1$ to stack
            }
        } // Sort the smaller subfile.
        if stack is empty then return;
        Delete($q$); Delete($p$); // Delete $q$ and $p$ from stack.
    } until (false);
}
```

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 $a[p]$, ... ,$a[q]$) 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.

```
Algorithm RQuickSort($p, q$)
// Sorts the elements $a[p], \dots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order. $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if ($p < q$) then
    {
        if (($q - p$) > 5) then
            Interchange($a$, Random() mod ($q - p + 1$) + $p, p$);
        $j$ := Partition($a, p, q + 1$);
            // $j$ is the position of the partitioning element.
        RQuickSort($p, j - 1$);
        RQuickSort($j + 1, q$);
    }
}
```

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 $0, 1, 2, \dots, n-2$, 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

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

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

```
Algorithm RSort($a, n$)
// Sort the elements $a[1 : n]$.
{
    Randomly sample $s$ elements from $a[\,]$;
    Sort this sample;
    Partition the input using the sorted sample as partition keys;
    Sort each part separately;
}
```

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