...ng records can easily be done. ... to appear in successive memory ... and the binary

9.3 INSERTION SORT

Suppose an array A with n elements A[1], A[2], . . . , A[N] is in memory. The insertion sort algorithm scans A from A[1] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[1], A[2], . . . , A[K−1]. That is:

Pass 1. A[1] by itself is trivially sorted.
Pass 2. A[2] is inserted either before or after A[1] so that: A[1], A[2] is sorted.
Pass 3. A[3] is inserted into its proper place in A[1], A[2], that is, before A[1], between A[1] and A[2], or after A[2], so that: A[1], A[2], A[3] is sorted.
Pass 4. A[4] is inserted into its proper place in A[1], A[2], A[3] so that:
A[1], A[2], A[3], A[4] is sorted.
. . . .
Pass N. A[N] is inserted into its proper place in A[1], A[2], . . . , A[N − 1] so that:
A[1], A[2], . . . , A[N] is sorted.

This sorting algorithm is frequently used when n is small. For example, this algorithm is very popular with bridge players when they are first sorting their cards.

There remains only the problem of deciding how to insert A[K] in its proper place in the sorted subarray A[1], A[2], . . . , A[K−1]. This can be accomplished by comparing A[K] with A[K−1], comparing A[K] with A[K−2], comparing A[K] with A[K−3], and so on, until first meeting an element A[J] such that A[J] ≤ A[K]. Then each of the elements A[K−1], A[K−2], . . . , A[J+1] is moved forward one location, and A[K] is then inserted in the J+1st position in the array.

The algorithm is simplified if there always is an element A[J] such that A[J] ≤ A[K]; otherwise we must constantly check to see if we are comparing A[K] with A[1]. This condition can be accomplished by introducing a sentinel element A[0] = −∞ (or a very small number).

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1:	−∞	77	33	44	11	88	22	66	55
K = 2:	−∞	77	33	44	11	88	22	66	55
K = 3:	−∞	33	77	44	11	88	22	66	55
K = 4:	−∞	33	44	77	11	88	22	66	55
K = 5:	−∞	11	33	44	77	88	22	66	55
K = 6:	−∞	11	33	44	77	88	22	66	55
K = 7:	−∞	11	22	33	44	77	88	66	55
K = 8:	−∞	11	22	33	44	66	77	88	55
Sorted:	−∞	11	22	33	44	55	66	77	88

Fig. 9-3 Insertion sort for $n = 8$ items.

9.4 SELECTION SORT

Suppose an array A with n elements A[1], A[2], . . . , A[N] is in memory. The selection sort algorithm for sorting A works as follows. First find the smallest element in the list and put it in the first position. Then find the second smallest element in the list and put it in the second position. And so on. More precisely:

Pass 1.	Find the location LOC of the smallest in the list of N elements A[1], A[2], . . . , A[N], and then interchange A[LOC] and A[1]. Then: A[1] is sorted.
Pass 2.	Find the location LOC of the smallest in the sublist of N − 1 elements A[2], A[3], . . . , A[N], and then interchange A[LOC] and A[2]. Then: A[1], A[2] is sorted, since A[1] ≤ A[2].
Pass 3.	Find the location LOC of the smallest in the sublist of N − 2 elements A[3], A[4], . . . , A[N], and then interchange A[LOC] and A[3]. Then: A[1], A[2], . . . , A[3] is sorted, since A[2] ≤ A[3].
. . .	. . . .
. . .	. . . .
Pass N − 1.	Find the location LOC of the smaller of the elements A[N − 1], A[N], and then interchange A[LOC] and A[N − 1]. Then: A[1], A[2], . . . , A[N] is sorted, since A[N − 1] ≤ A[N].

Thus A is sorted after N − 1 passes.

EXAMPLE 9.5

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

Applying the selection sort algorithm to A yields the data in Fig. 9-4. Observe that LOC gives the location of the smallest among A[K], A[K + 1], . . . , A[N] during Pass K. The circled elements indicate the elements which are to be interchanged.

There remains only the problem of finding, during the Kth pass, the location LOC of the smallest among the elements A[K], A[K + 1], . . . , A[N]. This may be accomplished by using a variable MIN to hold the current smallest value while scanning the subarray from A[K] to A[N]. Specifically, first set MIN := A[K] and LOC := K, and then traverse the list, comparing MIN with each other element A[J] as follows:

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1, LOC = 4	$\textcircled{77}$	33	44	$\textcircled{11}$	88	22	66	55
K = 2, LOC = 6	11	$\textcircled{33}$	44	77	88	$\textcircled{22}$	66	55
K = 3, LOC = 6	11	22	$\textcircled{44}$	77	88	$\textcircled{33}$	66	55
K = 4, LOC = 6	11	22	33	$\textcircled{77}$	88	$\textcircled{44}$	66	55
K = 5, LOC = 8	11	22	33	44	$\textcircled{88}$	77	66	$\textcircled{55}$
K = 6, LOC = 7	11	22	33	44	55	$\textcircled{77}$	$\textcircled{66}$	88
K = 7, LOC = 7	11	22	33	44	55	66	$\textcircled{77}$	88
Sorted:	11	22	33	44	55	66	77	88

Fig. 9-4 Selection sort for $n = 8$ items.

EXAMPLE 9.6

Suppose A has 5 elements and suppose B has 100 elements. Then merging A and B by Algorithm 9.4 uses approximately 100 comparisons. On the other hand, only approximately log 100 = 7 comparisons are needed to find the proper place to insert an element of A into B using a binary search. Hence only approximately 5 · 7 = 35 comparisons are need to merge A and B using the binary search and insertion algorithm.

The binary search and insertion algorithm does not take into account the fact that A is sorted. Accordingly, the algorithm may be improved in two ways as follows. (Here we assume that A has 5 elements and B has 100 elements.)

(1) *Reducing the target set*. Suppose after the first search we find that A[1] is to be inserted after B[16]. Then we need only use a binary search on B[17], . . . , B[100] to find the proper location to insert A[2]. And so on.

(2) *Tabbing*. The expected location for inserting A[1] in B is near B[20] (that is, B[s/r]), not near B[50]. Hence we first use a linear search on B[20], B[40], B[60], B[80] and B[100] to find B[K] such that A[1] ≤ B[K], and then we use a binary search on B[K − 20], B[K − 19], . . . , B[K]. (This is analogous to using the tabs in a dictionary which indicate the location of all words with the same first letter.)

The details of the revised algorithm are left to the reader.

9.6 MERGE-SORT

Suppose an array A with n elements A[1], A[2], . . . , A[N] is in memory. The merge-sort algorithm which sorts A will first be described by means of a specific example.

EXAMPLE 9.7

Suppose the array A contains 14 elements as follows:

66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30

Each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows:

Pass 1. Merge each pair of elements to obtain the following list of sorted pairs:

$$\underbrace{33,\ 66} \qquad \underbrace{22,\ 40} \qquad \underbrace{55,\ 88} \qquad \underbrace{11,\ 60} \qquad \underbrace{20,\ 80} \qquad \underbrace{44,\ 50} \qquad \underbrace{30,\ 77}$$

Pass 2. Merge each pair of pairs to obtain the following list of sorted quadruplets:

$$\underbrace{22,\ 33,\ 40,\ 66} \qquad \underbrace{11,\ 55,\ 60,\ 88} \qquad \underbrace{20,\ 44,\ 50,\ 80} \qquad \underbrace{30,\ 77}$$

Pass 3. Merge each pair of sorted quadruplets to obtain the following two sorted subarrays:

$$\underbrace{11,\ 22,\ 33,\ 40,\ 55,\ 60,\ 66,\ 88} \qquad \underbrace{20,\ 30,\ 44,\ 50,\ 77,\ 80}$$

Pass 4. Merge the two sorted subarrays to obtain the single sorted array

11, 20, 22, 30, 33, 40, 44, 50, 55, 60, 66, 77, 80, 88

original array A is now sorted.

The above merge-sort algorithm for sorting an array A has the following important property. After ss K, the array A will be partitioned into sorted subarrays where each subarray, except possibly the st, will contain exactly $L = 2^K$ elements. Hence the algorithm requires at most log n passes to sort an element array A.

The above informal description of merge-sort will now be translated into a formal algorithm which be divided into two parts. The first part will be a procedure MERGEPASS, which uses Procedure to execute a single pass of the algorithm; and the second part will repeatedly apply MERGEPASS l A is sorted.

The MERGEPASS procedure applies to an n-element array A which consists of a sequence of d subarrays. Moreover, each subarray consists of L elements except that the last subarray may fewer than L elements. Dividing n by 2 * L, we obtain the quotient Q, which tells the number of of L-element sorted subarrays; that is,

$$Q = \text{INT}(N/(2*L))$$

se INT(X) to denote the integer value of X.) Setting S = 2*L*Q, we get the total number S of nts in the Q pairs of subarrays. Hence R = N − S denotes the number of remaining elements. rocedure first merges the initial Q pairs of L-element subarrays. Then the procedure takes care of e where there is an odd number of subarrays (when R ≤ L) or where the last subarray has fewer L elements.

e formal statement of MERGEPASS and the merge-sort algorithm follow:

```
edure 9.6: MERGEPASS(A, N, L, B)
    The N-element array A is composed of sorted subarrays where each subarray has
    L elements except possibly the last subarray, which may have fewer than L
    elements. The procedure merges the pairs of subarrays of A and assigns them to
    the array B.
    1. Set Q := INT(N/(2*L)), S := 2*L*Q and R := N - S.
    2. [Use Procedure 9.5 to merge the Q pairs of subarrays.]
       Repeat for J = 1, 2, . . . , Q:
           (a) Set LB := 1 + (2*J - 2)*L. [Finds lower bound of first array.]
           (b) Call MERGE(A, L, LB, A, L, LB + L, B, LB).
       [End of loop.]
    3. [Only one subarray left?]
       If R ≤ L, then:
           Repeat for J = 1, 2, . . . , R:
               Set B(S + J) := A(S + J).
           [End of loop.]
       Else:
           Call MERGE(A, L, S + 1, A, R, L + S + 1, B, S + 1).
       [End of If structure.]
    4. Return.
```

Complexity of the Merge

Let $f(n)$ denote the number of comparisons needed to sort an n-element array A using the merge-sort algorithm. Recall that the algorithm requires at most $\log n$ passes. Moreover, each pass merges a total of n elements, and by the discussion on the complexity of merging, each pass will require at most n comparisons. Accordingly, for both the worst case and average case,

$$f(n) \le n \log n$$

Observe that this algorithm has the same order as heapsort and the same average order as quicksort. The main drawback of merge-sort is that it requires an auxiliary array with n elements. Each of the other sorting algorithms we have studied requires only a finite number of extra locations, which is independent of n.

The above results are summarized in the following table:

Algorithm	Worst Case	Average Case	Extra Memory
Merge-Sort	$n \log n = O(n \log n)$	$n \log n = O(n \log n)$	$O(n)$

9.7 RADIX SORT

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of the alphabet.) Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labeled as follows:

$$9,\ 8,\ 7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1,\ 0,\ 11,\ 12,\ \text{R (reject)}$$

Each pocket other than R corresonds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the units digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit. We illustrate with an example.

EXAMPLE 9.8

Suppose 9 cards are punched as follows:

$$348,\ 143,\ 361,\ 423,\ 538,\ 128,\ 321,\ 543,\ 366$$

Given to a card sorter, the numbers would be sorted in three phases, as pictured in Fig. 9-6:

(*a*) In the first pass, the units digits are sorted into pockets. (The pockets are pictured upside down, so 348 is at the bottom of pocket 8.) The cards are collected pocket by pocket, from pocket 9 to pocket 0. (Note that 361 will now be at the bottom of the pile and 128 at the top of the pile.) The cards are now reinput to the sorter.

(*b*) In the second pass, the tens digits are sorted into pockets. Again the cards are collected pocket by pocket and reinput to the sorter.

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(*a*) First pass.

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(*b*) Second pass.

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(*c*) Third pass.

Fig. 9-6

(*c*) In the third and final pass, the hundreds digits are sorted into pockets.

When the cards are collected after the third pass, the numbers are in the following order:

128, 143, 321, 348, 361, 366, 423, 538, 543

Thus the cards are now sorted.

The number C of comparisons needed to sort nine such 3-digit numbers is bounded as follows:

$$C \le 9 * 3 * 10 \qquad C \le n * 3 \times d$$

The 9 comes from the nine cards, the 3 comes from the three digits in each number, and the 10 comes from radix $d = 10$ digits.

Complexity of Radix Sort

Suppose a list A of n items $A_1, A_2, \ldots, A_n$ is given. Let d denote the radix (e.g., $d = 10$ for decimal digits, $d = 26$ for letters and $d = 2$ for bits), and suppose each item A_i is represented by means of s of the digits:

$$A_i = d_{i1} d_{i2} \cdots d_{is}$$

The radix sort algorithm will require s passes, the number of digits in each item. Pass K will compare each d_{iK} with each of the d digits. Hence the number $C(n)$ of comparisons for the algorithm is bounded as follows:

$$C(n) \le d * s * n$$

Although d is independent of n, the number s does depend on n. In the worst case, $s = n$, so $C(n) = O(n^2)$. In the best case, $s = \log_d n$, so $C(n) = O(n \log n)$. In other words, radix sort performs well only when the number s of digits in the representation of the A_i's is small.

Another drawback of radix sort is that one may need $d * n$ memory locations. This comes from the fact that all the items may be "sent to the same pocket" during a given pass. This drawback may be minimized by using linked lists rather than arrays to store the items during a given pass. However, one will still require $2 * n$ memory locations.