	NAME	SSN	SEX	SALARY
1				
2	Davis	192-38-7282	Female	22 800
3	Kelly	165-64-3351	Male	19 000
4	Green	175-56-2251	Male	27 200
5				
6	Brown	178-52-1065	Female	14 700
7	Lewis	181-58-9939	Female	16 400
8				
9	Cohen	177-44-4557	Male	19 000
10	Rubin	135-46-6262	Female	15 500
11				
12	Evans	168-56-8113	Male	34 200
13				
14	Harris	208-56-1654	Female	22 800

(a)

	POINT	PTRNAME	PTRSSN
1	2	6	10
2	3	9	3
3	4	2	12
4	6	12	4
5	7	4	9
6	9	14	6
7	10	3	7
8	12	7	2
9	14	10	14

(b)

Fig. 9.2

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).

Example 9.4

Suppose an array A contains 8 elements as follows:

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

Figure 9.3 illustrates the insertion sort algorithm. The circled element indicates the A[K] in each pass of the algorithm, and the arrow indicates the proper place for inserting A[K].

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*

The formal statement of our insertion sort algorithm follows.

Algorithm 9.1: (Insertion Sort) INSERTION(A, N).
This algorithm sorts the array A with N elements.

1. Set A[0] := –∞. [Initializes sentinel element.]
2. Repeat Steps 3 to 5 for K = 2, 3, …, N:
3. Set TEMP := A[K] and PTR := K – 1.
4. Repeat while TEMP < A[PTR]:
   (a) Set A[PTR + 1] := A[PTR]. [Moves element forward.]
   (b) Set PTR := PTR – 1.
   [End of loop.]
5. Set A[PTR + 1] := TEMP. [Inserts element in proper place.]
   [End of Step 2 loop.]
6. Return.

Observe that there is an inner loop which is essentially controlled by the variable PTR, and there is an outer loop which uses K as an index.

Complexity of Insertion Sort

The number $f(n)$ of comparisons in the insertion sort algorithm can be easily computed. First of all, the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number K – 1 of comparisons. Hence

$$f(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

Furthermore, one can show that, on the average, there will be approximately (K – 1)/2 comparisons in the inner loop. Accordingly, for the average case,

$$f(n) = \frac{1}{2} + \frac{2}{2} + \cdots + \frac{n-1}{2} = \frac{n(n-1)}{4} = O(n^2)$$

Thus the insertion sort algorithm is a very slow algorithm when n is very large.

The above results are summarized in the following table:

Algorithm	*Worst Case*	*Average Case*
Insertion Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{4} = O(n^2)$

Remark: Time may be saved by performing a binary search, rather than a linear search, to find the location in which to insert A[K] in the subarray A[1], A[2], ..., A[K – 1]. This requires, on the average, log K comparisons rather than (K – 1)/2 comparisons. However, one still needs to move (K – 1)/2 elements forward. Thus the order of complexity is not changed. Furthermore, insertion sort is usually used only when n is small, and in such a case, the linear search is about as efficient as the binary search.

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.

Pass	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
K = 1, LOC = 4	(77)	33	44	(11)	88	22	66	55
K = 2, LOC = 6	11	(33)	44	77	88	(22)	66	55
K = 3, LOC = 6	11	22	(44)	77	88	(33)	66	55
K = 4, LOC = 6	11	22	33	(77)	88	(44)	66	55
K = 5, LOC = 8	11	22	33	44	(88)	77	66	(55)
K = 6, LOC = 7	11	22	33	44	55	(77)	(66)	88
K = 7, LOC = 7	11	22	33	44	55	66	(77)	88
Sorted:	11	22	33	44	55	66	77	88

Fig. 9.4 *Selection Sort for n = 8 Items*

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:

(a) If MIN ≤ A[J], then simply move to the next element.

(b) If MIN > A[J], then update MIN and LOC by setting MIN := A[J] and LOC := J.

After comparing MIN with the last element A[N], MIN will contain the smallest among the elements A[K], A[K + 1], ..., A[N] and LOC will contain its location.

The above process will be stated separately as a procedure.

Procedure 9.2: MIN(A, K, N, LOC)
An array A is in memory. This procedure finds the location LOC of the smallest element among A[K], A[K + 1], ..., A[N].

1. Set MIN := A[K] and LOC := K. [Initializes pointers.]
2. Repeat for J = K + 1, K + 2, ..., N:
   If MIN > A[J], then: Set MIN := A[J] and LOC := A[J] and LOC := J.
   [End of loop.]
3. Return.

The selection sort algorithm can now be easily stated:

Algorithm 9.3: (Selection Sort) SELECTION(A, N)
This algorithm sorts the array A with N elements.

1. Repeat Steps 2 and 3 for K = 1, 2, ..., N – 1:
2. Call MIN(A, K, N, LOC).
3. [Interchange A[K] and A[LOC].]
   Set TEMP := A[K], A[K] := A[LOC] and A[LOC] := TEMP.
   [End of Step 1 loop.]
4. Exit.

Complexity of the Selection Sort Algorithm

First note that the number $f(n)$ of comparisons in the selection sort algorithm is independent of the original order of the elements. Observe that MIN(A, K, N, LOC) requires $n - K$ comparisons. That is, there are $n - 1$ comparisons during Pass 1 to find the smallest element, there are $n - 2$ comparisons during Pass 2 to find the second smallest element, and so on. Accordingly,

$$f(n) = (n - 1) + (n - 2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

The above result is summarized in the following table:

Algorithm	*Worst Case*	*Average Case*
Selection Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n^2)$

Remark: The number of interchanges and assignments does depend on the original order of the elements in the array A, but the sum of these operations does not exceed a factor of n^2.

9.5 MERGING

Suppose A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the elements of A and B into a single sorted list C with $n = r + s$ elements is called *merging*. One simple way to merge is to place the elements of B after the elements of A and then use some sorting algorithm on the entire list. This method does not take advantage of the fact that A and B are individually sorted. A much more efficient algorithm is Algorithm 9.4 in this section. First, however, we indicate the general idea of the algorithm by means of two examples.

Suppose one is given two sorted decks of cards. The decks are merged as in Fig. 9.5. That is, at each step, the two front cards are compared and the smaller one is placed in the combined deck. When one of the decks is empty, all of the remaining cards in the other deck are put at the end of the combined deck. Similarly, suppose we have two lines of students sorted by increasing heights, and suppose we want to merge them into a single sorted line. The new line is formed by choosing, at each step, the shorter of the two students who are at the head of their respective lines. When one of the lines has no more students, the remaining students line up at the end of the combined line.

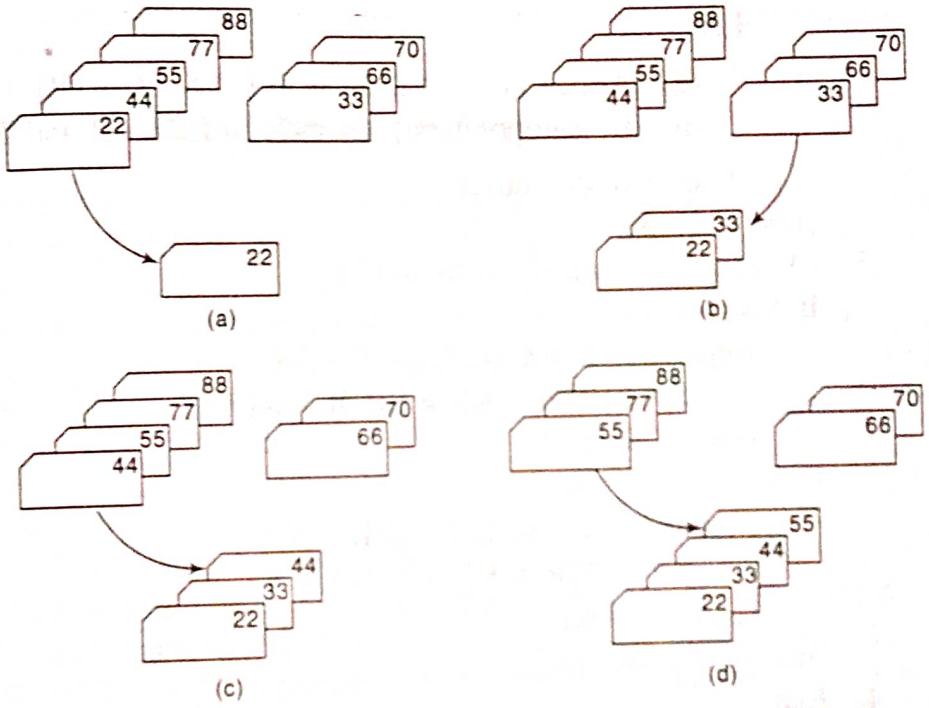

Fig. 9.5

The above discussion will now be translated into a formal algorithm which merges a sorted r-element array A and a sorted s-element array B into a sorted array C, with $n = r + s$ elements. First of all, we must always keep track of the locations of the smallest element of A and the smallest element of B which have not yet been placed in C. Let NA and NB denote these locations, respectively. Also, let PTR denote the location in C to be filled. Thus, initially, we set NA := 1, NB := 1 and PTR := 1. At each step of the algorithm, we compare

A[NA] and B[NB]

and assign the smaller element to C[PTR]. Then we increment PTR by setting PTR := PTR + 1, and we either increment NA by setting NA := NA + 1 or increment NB by setting NB := NB + 1, according to whether the new element in C has come from A or from B. Furthermore, if NA > r, then the remaining elements of B are assigned to C; or if NB > s, then the remaining elements of A are assigned to C.

The formal statement of the algorithm follows.

Algorithm 9.4: MERGING(A, R, B, S, C)

Let A and B be sorted arrays with R and S elements, respectively. This algorithm merges A and B into an array C with N = R + S elements.

1. [Initialize.] Set NA := 1, NB := 1 and PTR := 1.
2. [Compare.] Repeat while NA ≤ R and NB ≤ S:
   If A[NA] < B[NB], then:
   (a) [Assign element from A to C.] Set C[PTR] := A[NA].
   (b) [Update pointers.] Set PTR := PTR + 1 and NA := NA + 1.
   Else:
   (a) [Assign element from B to C.] Set C[PTR] := B[NB].
   (b) [Update pointers.] Set PTR := PTR + 1 and NB := NB + 1.
   [End of If structure.]
   [End of loop.]
3. [Assign remaining elements to C.]
   If NA > R, then:
   Repeat for K = 0, 1, 2, ..., S – NB:
   Set C[PTR + K] := B[NB + K].
   [End of loop.]
   Else:
   Repeat for K = 0, 1, 2, ..., R – NA:
   Set C[PTR + K] := A[NA + K].
   [End of loop.]
   [End of If structure.]
4. Exit.

Complexity of the Merging Algorithm

The input consists of the total number $n = r + s$ of elements in A and B. Each comparison assigns an element to the array C, which eventually has n elements. Accordingly, the number $f(n)$ of comparisons cannot exceed n:

$$f(n) \leq n = O(n)$$

In other words, the merging algorithm can be run in linear time.

Nonregular Matrices

Suppose A, B and C are matrices, but not necessarily regular matrices. Assume A is sorted, with r elements and lower bound LBA; B is sorted, with s elements and lower bound LBB; and C has lower bound LBC. Then UBA = LBA + r − 1 and UBB = LBB + s − 1 are, respectively, the upper bounds of A and B. Merging A and B now may be accomplished by modifying the above algorithm as follows.

Procedure 9.5: MERGE(A, R, LBA, S, LBB, C, LBC)
This procedure merges the sorted arrays A and B into the array C.

1. Set NA := LBA, NB := LBB, PTR := LBC, UBA := LBA + R − 1, UBB := LBB + S − 1.
2. Same as Algorithm 9.4 except R is replaced by UBA and S by UBB.
3. Same as Algorithm 9.4 except R is replaced by UBA and S by UBB.
4. Return.

Observe that this procedure is called MERGE, whereas Algorithm 9.4 is called MERGING. The reason for stating this special case is that this procedure will be used in the next section, on merge-sort.

Binary Search and Insertion Algorithm

Suppose the number r of elements in a sorted array A is much smaller than the number s of elements in a sorted array B. One can merge A with B as follows. For each element A[K] of A, use a binary search on B to find the proper location to insert A[K] into B. Each such search requires at most log s comparisons; hence this binary search and insertion algorithm to merge A and B requires at most r log s comparisons. We emphasize that this algorithm is more efficient than the usual merging Algorithm 9.4 only when $r << s$, that is, when r is much less than s.

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 needed 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:

33, 66 22, 40 55, 88 11, 60 20, 80 44, 50 30, 70

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

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

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

11, 22, 33, 40, 55, 60, 66, 88 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

The original array A is now sorted.

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

The above informal description of merge-sort will now be translated into a formal algorithm which will be divided into two parts. The first part will be a procedure MERGEPASS, which uses Procedure 9.5 to execute a single pass of the algorithm; and the second part will repeatedly apply MERGEPASS until A is sorted.

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

$$Q = INT(N/(2*L))$$

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

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

Procedure 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.

Algorithm 9.7: MERGESORT(A, N)

This algorithm sorts the N-element array A using an auxiliary array B.

1. Set L := 1. [Initializes the number of elements in the subarrays.]
2. Repeat Steps 3 to 6 while L < N:
3. Call MERGEPASS(A, N, L, B).
4. Call MERGEPASS(B, N, 2 * L, A).
5. Set L := 4 * L.

   [End of Step 2 loop.]
6. Exit.

Since we want the sorted array to finally appear in the original array A, we must execute the procedure MERGEPASS an even number of times.

Complexity of the Merge-Sort Algorithm

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,

$$f(n) \leq n \log n$$

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, R (reject)

Each pocket other than R corresponds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the