2 ≥ 3

executed until J reaches K. The next step, Step 5, inserts ITEM into the array in the space just created. Before the exit from the algorithm, the number N of elements in LA is increased by 1 to account for the new element.

```
Algorithm 4.2: (Inserting into a Linear Array) INSERT(LA, N, K, ITEM)      some type data..
    Here LA is a linear array with N elements and K is a positive integer such that      not in points/decimal
    K ≤ N. This algorithm inserts an element ITEM into the Kth position in LA.
    1. [Initialize counter.] Set J := N. 5
    2. Repeat Steps 3 and 4 while J ≥ K: insert position   j = 5   4≥3   5≥3      Fold
    3.     [Move Jth element downward.] Set LA[J + 1] := LA[J].
    4.     [Decrease counter.] Set J := J − 1.   5-1 4   while; 5 mean 5 no. value on LA[5]] mean LA[0]
       [End of Step 2 loop.]
    5. [Insert element.] Set LA[K] := ITEM.   (Fold)      - total positions
    6. [Reset N.] Set N := N + 1.
                         5+1 := 6
    7. Exit.
```

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

```
Algorithm 4.3: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
    Here LA is a linear array with N elements and K is a positive integer such that      forward movement
    K ≤ N. This algorithm deletes the Kth element from LA.      insertion.
    1. Set ITEM := LA[K]. Display value from array.   ITEM = Davis
    2. Repeat for J = K to N − 1:   2 to 7-1 → 2 to 6      backward movement
           [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
       [End of loop.]      LA[2] (3)  2+1      in deletion.
    3. [Reset the number N of elements in LA.] Set N := N − 1. A[3]2
                         7-1 = 6
    4. Exit.
```

Remark: We emphasize that if many deletions and insertions are to be made in a collection of data elements, then a linear array may not be the most efficient way of storing the data.

4.6 SORTING; BUBBLE SORT

- sort number (n-1) time comparing.

Let A be a list of n numbers. *Sorting* A refers to the operation of rearranging the elements of A so they are in increasing order, i.e., so that

$$A[1] < A[2] < A[3] < \cdots < A[N]$$

For example, suppose A originally is the list

8, 4, 19, 2, 7, 13, 5, 16

After sorting, A is the list

2, 4, 5, 7, 8, 13, 16, 19

Sorting may seem to be a trivial task. Actually, sorting efficiently may be quite complicated. In fact, there are many, many different sorting algorithms; some of these algorithms are discussed in Chap. 9. Here we present and discuss a very simple sorting algorithm known as the *bubble sort*.

Remark: The above definition of sorting refers to arranging numerical data in increasing order; this restriction is only for notational convenience. Clearly, sorting may also mean arranging numerical

data in decreasing order or arranging nonnumerical data in alphabetical order. Actu
frequently a file of records, and sorting A refers to rearranging the records of A so that the v
given key are ordered.

Bubble Sort

Suppose the list of numbers A[1], A[2], . . . , A[N] is in memory. The bubble sort algorith
as follows:

Step 1. Compare A[1] and A[2] and arrange them in the desired order, so that A[1] <
Then compare A[2] and A[3] and arrange them so that A[2] < A[3]. Then co
A[3] and A[4] and arrange them so that A[3] < A[4]. Continue until we co
A[N − 1] with A[N] and arrange them so that A[N − 1] < A[N].

Observe that Step 1 involves $n - 1$ comparisons. (During Step 1, the largest element is "bubbl
to the nth position or "sinks" to the nth position.) When Step 1 is completed, A[N] will conta
largest element.

Step 2. Repeat Step 1 with one less comparison; that is, now we stop after we compare
possibly rearrange A[N − 2] and A[N − 1]. (Step 2 involves N − 2 comparisons
when Step 2 is completed, the second largest element will occupy A[N − 1]

Step 3. Repeat Step 1 with two fewer comparisons; that is, we stop after we compare
possibly rearrange A[N − 3] and A[N − 2].

. . . .

. . . .

. . . .

Step N − 1. Compare A[1] with A[2] and arrange them so that A[1] < A[2].

After $n - 1$ steps, the list will be sorted in increasing order.

The process of sequentially traversing through all or part of a list is frequently called a "pass," s
each of the above steps is called a pass. Accordingly, the bubble sort algorithm requires $n - 1$ passes
where n is the number of input items.

EXAMPLE 4.7

Suppose the following numbers are stored in an array A:

32, 51, 27, 85, 66, 23, 13, 57 ← Data

We apply the bubble sort to the array A. We discuss each pass separately.

Pass 1. We have the following comparisons:

(a) Compare A_1 and A_2. Since 32 < 51, the list is not altered.

(b) Compare A_2 and A_3. Since 51 > 27, interchange 51 and 27 as follows:

— check condition less than .

32, 27, 51, 85, 66, 23, 13, 57

(c) Compare A_3 and A_4. Since 51 < 85, the list is not altered.

smallest to l

(d) Compare A_4 and A_5. Since 85 > 66, interchange 85 and 86 as follows:

32, 27, 51, 66, 85, 23, 13, 57

(e) Compare A_5 and A_6. Since 85 > 23, interchange 85 and 23 as follows:

32, 27, 51, 66, 23, 85, 13, 57

(f) Compare A_6 and A_7. Since 85 > 13, interchange 85 and 13 to yield:

32, 27, 51, 66, 23, 13, 85, 57

(g) Compare A_7 and A_8. Since 85 > 57, interchange 85 and 57 to yield:

32, 27, 51, 66, 23, 13, 57, 85

Pass 6. 13, 23, 27, 33, 51, 57, 66, 85

Pass 6 actually has two comparisons, A_1 with A_2 and A_2 and A_3. The second comparison does not involve an interchange.

Pass 7. Finally, A_1 is compared with A_2. Since 13 < 23, no interchange takes place.

Since the list has 8 elements; it is sorted after the seventh pass. (Observe that in this example, the list was actually sorted after the sixth pass. This condition is discussed at the end of the section.)

We now formally state the bubble sort algorithm.

(name of array (place) where al this keep)

```
Algorithm 4.4: (Bubble Sort) BUBBLE(DATA, N)                      8
               Here DATA is an array with N elements. This algorithm sorts the elements in
               DATA.
                                                    1 to 8-1        1 TO 7
               1. Repeat Steps 2 and 3 for K = 1 to N - 1.          no. of element
               2.     Set PTR := 1. [Initializes pass pointer PTR.]
               3.     Repeat while PTR ≤ N - K: [Executes pass.]    1 ≤ 7(1) → 1 ≤ 7
                          (a) If DATA[PTR] > DATA[PTR + 1], then:
                                  Interchange DATA[PTR] and DATA[PTR + 1].
                              [End of If structure.]
                          (b) Set PTR := PTR + 1.        1     2
                      [End of inner loop.]                 32 > Data [1+1]
                  [End of Step 1 outer loop.]                        2
               4. Exit.  (return)                                    51
```

PTR (Path pointer)
PTR (1value)
PTR+1 (2value)

32
32 >

Observe that there is an inner loop which is controlled by the variable PTR, and the loop is contained in an outer loop which is controlled by an index K. Also observe that PTR is used as a subscript but K is not used as a subscript, but rather as a counter.

Complexity of the Bubble Sort Algorithm

Traditionally, the time for a sorting algorithm is measured in terms of the number of comparisons. The number $f(n)$ of comparisons in the bubble sort is easily computed. Specifically, there are $n - 1$ comparisons during the first pass, which places the largest element in the last position; there are $n - 2$ comparisons in the second step, which places the second largest element in the next-to-last position; and so on. Thus

$$\frac{n^2 - n}{2}$$

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

$$\frac{n^2}{2}$$

In other words, the time required to execute the bubble sort algorithm is proportional to n^2 the number of input items.

Remark: Some programmers use a bubble sort algorithm that contains a 1-bit variable a *logical* variable FLAG) to signal when no interchange takes place during a pass. If FLA any pass, then the list is already sorted and there is no need to continue. This may cut d number of passes. However, when using such a flag, one must initialize, change and test th FLAG during each pass. Hence the use of the flag is efficient only when the list originally is sorted order.

4.7 SEARCHING; LINEAR SEARCH

Let DATA be a collection of data elements in memory, and suppose a specific information is given. *Searching* refers to the operation of finding the location LOC of ITEM or printing some message that ITEM does not appear there. The search is said to be *successful* does appear in DATA and *unsuccessful* otherwise.

Frequently, one may want to add the element ITEM to DATA after an unsuccessful ITEM in DATA. One then uses a *search and insertion* algorithm, rather than simply algorithm; such search and insertion algorithms are discussed in the problem sections.

There are many different searching algorithms. The algorithm that one chooses generally on the way the information in DATA is organized. Searching is discussed in detail in Chap section discusses a simple algorithm called *linear search*, and the next section discusses the we algorithm called *binary search*.

The complexity of searching algorithms is measured in terms of the number $f(n)$ of com required to find ITEM in DATA where DATA contains n elements. We shall show that linear a linear time algorithm, but that binary search is a much more efficient algorithm, proportional to $\log_2 n$. On the other hand, we also discuss the drawback of relying only on the binary algorithm.

Linear Search

Suppose DATA is a linear array with n elements. Given no other information about DA most intuitive way to search for a given ITEM in DATA is to compare ITEM with each ele DATA one by one. That is, first we test whether DATA[1] = ITEM, and then we test DATA[2] = ITEM, and so on. This method, which traverses DATA sequentially to locate IT called *linear search* or *sequential search*.

To simplify the matter, we first assign ITEM to DATA[N + 1], the position following element of DATA. Then the outcome

$$\text{LOC} = \text{N} + 1$$

where LOC denotes the location where ITEM first occurs in DATA, signifies the se unsuccessful. The purpose of this initial assignment is to avoid repeatedly testing whether or have reached the end of the array DATA. This way, the search must eventually "succeed."

A formal presentation of linear search is shown in Algorithm 4.5.

Observe that Step 1 guarantees that the loop in Step 3 must terminate. Without Step Algorithm 2.4), the Repeat statement in Step 3 must be replaced by the following statement. involves two comparisons, not one:

$$\text{Repeat while LOC} \le \text{N and DATA[LOC]} \ne \text{ITEM:}$$

On the other hand, in order to use Step 1, one must guarantee that there is an unused memory lo comparisons. Thus, in the worst case, ... ion. (See Sec. 2

The running time of the average case uses the proba... Suppose p_k is the probability that ITEM appears in DATA[K], and suppose q is the probability ITEM does not appear in DATA. (Then $p_1 + p_2 + \cdots + p_n + q = 1$.) Since the algorithm u comparisons when ITEM appears in DATA[K], the average number of comparisons is given b

$$f(n) = 1 \cdot p_1 + 2 \cdot p_2 + \cdots + n \cdot p_n + (n+1) \cdot q$$

(item doesn't appear)

In particular, suppose q is very small and ITEM appears with equal probability in each eleme DATA. Then $q \approx 0$ and each $p_i = 1/n$. Accordingly,

$$f(n) = 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \cdots + n \cdot \frac{1}{n} + (n+1) \cdot 0 = (1 + 2 + \cdots + n) \cdot \frac{1}{n}$$

$$= \frac{n(n+1)}{2} \cdot \frac{1}{n} = \frac{n+1}{2}$$

That is, in this special case, the average number of comparisons required to find the location of IT is approximately equal to half the number of elements in the array.

4.8 BINARY SEARCH

Suppose DATA is an array which is sorted in increasing numerical order or, equivale alphabetically. Then there is an extremely efficient searching algorithm, called *binary search* w can be used to find the location LOC of a given ITEM of information in DATA. Before form discussing the algorithm, we indicate the general idea of this algorithm by means of an ideal version of a familiar everyday example.

Suppose one wants to find the location of some name in a telephone directory (or some word dictionary). Obviously, one does not perform a linear search. Rather, one opens the directory in middle to determine which half contains the name being sought. Then one opens that half in the mi to determine which quarter of the directory contains the name. Then one opens that quarter in middle to determine which eighth of the directory contains the name. And so on. Eventually, one f the location of the name, since one is reducing (very quickly) the number of possible locations for the directory.

The binary search algorithm applied to our array DATA works as follows. During each stag our algorithm, our search for ITEM is reduced to a *segment* of elements of DATA:

$$\text{DATA[BEG], DATA[BEG + 1], DATA[BEG + 2], \ldots, DATA[END]}$$

Note that the variables BEG and END denote, respectively, the beginning and end locations of segment under consideration. The algorithm compares ITEM with the middle element DATA[M of the segment, where MID is obtained by

$$\text{MID} = \text{INT}((\text{BEG} + \text{END})/2)$$

(We use INT(A) for the integer value of A.) If DATA[MID] = ITEM, then the search is successfu we set LOC := MID. Otherwise a new segment of DATA is obtained as follows:

(*a*) If ITEM < DATA[MID], then ITEM can appear only in the left half of the segment

$$\text{DATA[BEG], DATA[BEG + 1], \ldots, DATA[MID - 1]}$$

So we reset END := MID − 1 and begin searching again.

(*b*) If ITEM > DATA[MID], then ITEM can appear only in the right half of the segment:

DATA[MID + 1], DATA[MID + 2], . . . , DATA[END]

So we reset BEG := MID + 1 and begin searching again.

Initially, we begin with the entire array DATA; i.e., we begin with BEG = 1 and END = *n*, or, more generally, with BEG = LB and END = UB.

If ITEM is not in DATA, then eventually we obtain

END < BEG

This condition signals that the search is unsuccessful, and in such a case we assign LOC := NULL. Here NULL is a value that lies outside the set of indices of DATA. (In most cases, we can choose NULL = 0.)

We state the binary search algorithm formally.

(array) beg. end find.

```
Algorithm 4.6: (Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)
    Here DATA is a sorted array with lower bound LB and upper bound UB, and
    ITEM is a given item of information. The variables BEG, END and MID
    denote, respectively, the beginning, end and middle locations of a segment of
    elements of DATA. This algorithm finds the location LOC of ITEM in DATA or
    sets LOC = NULL.

    1. [Initialize segment variables.]
       Set BEG := LB, END := UB and MID = INT((BEG + END)/2).
    2. Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
    3.     If ITEM < DATA[MID], then:
               Set END := MID − 1.
           Else:
               Set BEG := MID + 1.
           [End of If structure.]
    4.     Set MID := INT((BEG + END)/2).
       [End of Step 2 loop.]
    5. If DATA[MID] = ITEM, then:
           Set LOC := MID.
       Else:
           Set LOC := NULL.
       [End of If structure.]
    6. Exit.
```

Beg = LB End = UB

MID = (Beg + End/2)

conditions :-

Beg ≤ end

Item ≠ [MID] Data

Remark: Whenever ITEM does not appear in DATA, the algorithm eventually arrives at the stage that BEG = END = MID. Then the next step yields END < BEG, and control transfers to Step 5 of the algorithm. This occurs in part (*b*) of the next example.

EXAMPLE 4.9

Let DATA be the following sorted 13-element array: 6 Mid N = 13

DATA: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

We apply the binary search to DATA for different values of ITEM.

(*a*) Suppose ITEM = 40. The search for ITEM in the array DATA is pictured in Fig. 4-6, where the values of DATA[BEG] and DATA[END] in each stage of the algorithm are indicated by circles and the value of

first index

last index

value

ght side larger value

6.5 QUICKSORT, AN APPLICATION OF STACKS

Let A be a list of n data items. "Sorting A" refers to the operation of rearranging the elements of A so that they are in some logical order, such as numerically ordered when A contains numerical data, or alphabetically ordered when A contains character data. The subject of sorting, including various sorting algorithms, is treated mainly in Chap. 9. This section gives only one sorting algorithm, called quicksort, in order to illustrate an application of stacks.

Quicksort is an algorithm of the divide-and-conquer type. That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets. We illustrate this "reduction step" by means of a specific example.

Suppose A is the following list of 12 numbers:

44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

compare element with right side

1 The reduction step of the quicksort algorithm finds the final position of one of the numbers; in this illustration, we use the first number 44. This is accomplished as follows. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list

22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66

2 (Observe that the numbers 88 and 66 to the right of 44 are each greater than 44.) Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list

22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

3 (Observe that the numbers 22, 33 and 11 to the left of 44 are each less than 44.) Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list

find larger find smallest

22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66

4 (Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

5 (Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such a number before meeting 44. This means all numbers have been scanned and compared with 44. Furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44, as shown below:

smaller than larger than

$\underbrace{22, \quad 33, \quad 11, \quad 40,}_{\text{First sublist}} \quad 44, \quad \underbrace{90, \quad 77, \quad 60, \quad 99, \quad 55, \quad 88, \quad 66}_{\text{Second sublist}}$

5th

Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to the task of sorting each of the above sublists.

The above reduction step is repeated with each sublist containing 2 or more elements. Since we can process only one sublist at a time, we must be able to keep track of some sublists for future processing. This is accomplished by using two stacks, called LOWER and UPPER, to temporarily "hold" such

22, 33, 11, 40 11, 22, 33, 40

11 33, 22, 40

sublists. That is, the addresses of the first and last elements of each sublist, called its *boundary values*, are pushed onto the stacks LOWER and UPPER, respectively; and the reduction step is applied to a sublist only after its boundary values are removed from the stacks. The following example illustrates the way the stacks LOWER and UPPER are used.

EXAMPLE 6.7

Consider the above list A with $n = 12$ elements. The algorithm begins by pushing the boundary values 1 and 12 of A onto the stacks to yield

LOWER: 1 UPPER: 12

In order to apply the reduction step, the algorithm first removes the top values 1 and 12 from the stacks, leaving

LOWER: (empty) UPPER: (empty)

and then applies the reduction step to the corresponding list A[1], A[2], . . . , A[12]. The reduction step, as executed above, finally places the first element, 44, in A[5]. Accordingly, the algorithm pushes the boundary values 1 and 4 of the first sublist and the boundary values 6 and 12 of the second sublist onto the stacks to yield

LOWER: 1, 6 UPPER: 4, 12 both

In order to apply the reduction step again, the algorithm removes the top values, 6 and 12, from the stacks, leaving

LOWER: 1 UPPER: 4

and then applies the reduction step to the corresponding sublist A[6], A[7], . . . , A[12]. The reduction step changes this list as in Fig. 6-9. Observe that the second sublist has only one element. Accordingly, the algorithm pushes only the boundary values 6 and 10 of the first sublist onto the stacks to yield

LOWER: 1, 6 UPPER: 4, 10

And so on. The algorithm ends when the stacks do not contain any sublist to be processed by the reduction step.

66, 77, 60, 88, 55
6 , 66, 55

A[6],	A[7],	A[8],	A[9],	A[10],	A[11],	A[12],
90,	77,	60,	99,	55,	88,	66
66,	77,	60,	99,	55,	88,	90
66,	77,	60,	90,	55,	88,	99
66,	77,	60,	88,	55,	90,	99

First sublist Second sublist

Fig. 6-9

The formal statement of our quicksort algorithm follows (on page 175). For notational convenience and pedagogical considerations, the algorithm is divided into two parts. The first part gives a procedure, called QUICK, which executes the above reduction step of the algorithm, and the second part uses QUICK to sort the entire list.

Observe that Step 2(*c*) (iii) is unnecessary. It has been added to emphasize the symmetry between Step 2 and Step 3. The procedure does not assume the elements of A are distinct. Otherwise, the condition LOC ≠ RIGHT in Step 2(*a*) and the condition LEFT ≠ LOC in Step 3(*a*) could be omitted.

The second part of the algorithm follows (on page 175). As noted above, LOWER and UPPER are stacks on which the boundary values of the sublists are stored. (As usual, we use NULL = 0.)

```
dure 6.5: QUICK(A, N, BEG, END, LOC)
    Here A is an array with N elements. Parameters BEG and END contain the
    boundary values of the sublist of A to which this procedure applies. LOC keeps
    track of the position of the first element A[BEG] of the sublist during the
    procedure. The local variables LEFT and RIGHT will contain the boundary
    values of the list of elements that have not been scanned.

    1. [Initialize.] Set LEFT := BEG, RIGHT := END and LOC := BEG.
    2. [Scan from right to left.]
       (a) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT:
               RIGHT := RIGHT − 1.
           [End of loop.]
       (b) If LOC = RIGHT, then: Return.
       (c) If A[LOC] > A[RIGHT], then:
               (i)   [Interchange A[LOC] and A[RIGHT].]
                         TEMP := A[LOC], A[LOC] := A[RIGHT],
                         A[RIGHT] := TEMP.
               (ii)  Set LOC := RIGHT.
               (iii) Go to Step 3.
           [End of If structure.]
    3. [Scan from left to right.]
       (a) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC:
               LEFT := LEFT + 1.
           [End of loop.]
       (b) If LOC = LEFT, then: Return.
       (c) If A[LEFT] > A[LOC], then
               (i)   [Interchange A[LEFT] and A[LOC].]
                         TEMP := A[LOC], A[LOC] := A[LEFT],
                         A[LEFT] := TEMP.
               (ii)  Set LOC := LEFT.
               (iii) Go to Step 2.
           [End of If structure.]
```

```
Algorithm 6.6: (Quicksort) This algorithm sorts an array A with N elements.

    1. [Initialize.] TOP := NULL.
    2. [Push boundary values of A onto stacks when A has 2 or more elements.]
       If N > 1, then: TOP := TOP + 1, LOWER[1] := 1, UPPER[1] := N.
    3. Repeat Steps 4 to 7 while TOP ≠ NULL.
    4.     [Pop sublist from stacks.]
           Set BEG := LOWER[TOP], END := UPPER[TOP],
           TOP := TOP − 1.
    5.     Call QUICK(A, N, BEG, END, LOC). [Procedure 6.5.]
    6.     [Push left sublist onto stacks when it has 2 or more elements.]
           If BEG < LOC − 1, then:
               TOP := TOP + 1, LOWER[TOP] := BEG,
               UPPER[TOP] = LOC − 1.
           [End of If structure.]
    7.     [Push right sublist onto stacks when it has 2 or more elements.]
           If LOC + 1 < END, then:
               TOP := TOP + 1, LOWER[TOP] := LOC + 1,
               UPPER[TOP] := END.
           [End of If structure.]
       [End of Step 3 loop.]
    8. Exit.
```