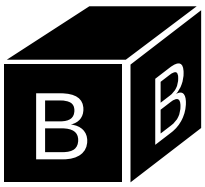


Chapter *16*

Iterative Searching and Sorting

Probably the most important algorithms in all of computer science are the searching and sorting algorithms. They are important because they are so common. To search for an item is to know a key value and to look it up in a list. For example, when you look up a word in a dictionary or look up a phone number in a phone book based on a person's name you are performing a search. Putting data in order is performing a sort. For example, table entries in many business reports are in some kind of order. The post office wants bulk mailings to be in order by zip code.

Searching

This section presents two basic search algorithms, the sequential search and the binary search. In a search problem, you are given

- An array of values
- The number of values in the array
- A search value

The algorithm must determine whether the array contains a value equal to the search value. If it does, the algorithm must compute the index of the array where the value is located.

For example, suppose you declare the following procedure:

```
PROCEDURE Search (IN v: ARRAY OF INTEGER; numItm, srchNum: INTEGER;
      OUT i: INTEGER; OUT fnd: BOOLEAN);
```

Also, suppose that numItm has the value 4, and the first four values in v are

50 20 70 60

If srchNum has the value 70, then the search algorithm should set fnd to TRUE and i to 2, because v[2] has the value 70. If srchNum has the value 40, the algorithm should set fnd to FALSE. It does not matter what value it gives to i, because v does not contain the value 40.

Parameter v is called by constant reference, because the purpose of Search is not to change the values of v, but to use the values of the actual parameter. It is called by constant reference instead of by value because it could be a long array. Parameters

numItm and srchNum are called by value, because the calling procedure gives the values to Search, with information flowing from the calling to the called procedure. Parameters i and fnd are called by result, because their initial values can be considered undefined when the procedure is first called, and Search will change the values of the corresponding actual parameters.

The sequential search

The sequential search algorithm starts at the first of the list. It compares srchNum with v[0], then v[1], and so on until it either finds srchNum in v or it gets to the end of the list. One version of the algorithm follows.

```
PROCEDURE Search (IN v: ARRAY OF INTEGER; numItm, srchNum: INTEGER;
      OUT i: INTEGER; OUT fnd: BOOLEAN);
BEGIN
   ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
   i := 0;
   WHILE (i < numItm) & (v[i] # srchNum) DO
      INC(i)
   END;
   fnd := i < numItm
END Search;
```

Figure 16.1
The first version of the sequential search algorithm. This is not the most efficient version.

If you trace this algorithm with a value of 70 for srchNum, you will see that i first gets 0. The WHILE expression is true because 0 is less than 4, and 50 is not equal to srchNum. When i gets 1, the WHILE expression is true again, because 1 is less than 4 and 20 is not equal to srchNum.

When i gets 2, however, the WHILE expression is false, because v[2] equals srchNum. The loop terminates, and fnd gets true. The value of i is the index of v where srchNum was found.

If you trace the algorithm with 40 for the value of srchNum, i will get 0, then 1, then 2, then 3. When i is 3, the WHILE expression will still be true, so i will get 4. Then the expression

```
(i < numItm) & (v[i] # srchNum)
```

will be evaluated with i having a value of 4.

At this point in the execution, an interesting feature of Component Pascal comes into play. The first part of the expression is false because i is equal to numItm. The second part of the expression does not need to be evaluated. Regardless of whether it is true or false, the entire expression will be false because of the first part.

There are two evaluation techniques in this situation. One is called full evaluation, and the other is called short-circuit evaluation. The steps of the full evaluation technique are

Full evaluation of AND expressions

- Evaluate the first part.
- Evaluate the second part.
- Perform the AND operation.

The steps of the short-circuit evaluation technique are

Short-circuit evaluation of AND expressions

- Evaluate the first part.
- If it is false, skip the second part.
- Otherwise, evaluate the second part.

With full evaluation, both parts are evaluated, regardless of whether the first part is true or false.

Fortunately, Component Pascal uses the short-circuit evaluation technique. In this example, the second part of the expression

```
(v[i] # srchNum)
```

will not be evaluated, with i having the value 4. If it were evaluated, the algorithm would be comparing v[4] with 40. If the array in the actual parameter were an ARRAY 4 OF INTEGER, the comparison would generate a trap. Because Component Pascal uses the short-circuit evaluation technique, such a trap will not occur with this algorithm.

The short-circuit evaluation technique also works with expressions that contain the OR boolean operator. When Component Pascal evaluates the boolean expression in the WHILE statement

```
WHILE p OR q DO
```

if it finds that *p* is true it does not evaluate *q*, because the entire boolean expression will be true regardless of whether expression *q* is true or false.

How fast is this algorithm? That depends on several things, namely how many items are in the list, whether the list contains the value of srchNum, and if it does contain srchNum, where it is located. Because the performance depends on these various factors, three categories of performance are commonly specified. They are

- Best-case performance
- Worst-case performance
- Average performance

In this algorithm, the best case is when v[0] contains the same value as srchNum. The worst case is when the value of srchNum is not in the list at all. The average case is somewhat difficult to define because it depends on the probability that srchNum will be in the list. We will not pursue the problem of determining the average performance of the search algorithms.

Search algorithms are usually evaluated by counting the number of comparisons necessary to find the search item. This algorithm makes two comparisons each time it evaluates the while expression. They are

```
i < numItm
```

and

v[i] # srchNum

In the best case, the WHILE expression is false the first time, the body of the loop never executes, and the algorithm makes two comparisons. Let *n* equal the value of numItm. In the worst case, the algorithm searches the entire list, evaluating the WHILE expression *n* times plus one additional comparison to detect the end of the list. So it makes 2*n* + 1 comparisons.

Therefore, the performance of this algorithm is

- Best case: 2 comparisons
- Worst case: 2*n* + 1 comparisons

If you search a list of 1000 items, then in the best case you will make two comparisons and in the worst case you will make 2001 comparisons.

With a little thought, you can improve this sequential search algorithm substantially. Think about the WHILE expression. Why do you need the comparison of i with numItm? Because, if srchNum is not in the list, i would keep getting bigger and the loop would not terminate. You would not need the first comparison if you knew that srchNum was in the list. You can guarantee that it will be in the list if you put it there yourself, before starting the loop.

```
PROCEDURE Search (VAR v: ARRAY OF INTEGER; numItm, srchNum: INTEGER;
      OUT i: INTEGER; OUT fnd: BOOLEAN);
BEGIN
   ASSERT((0 <= numItm) & (numItm < LEN(v)), 20);
   v[numItm] := srchNum;
   i := 0;
   WHILE v[i] # srchNum DO
      INC(i)
   END;
   fnd := i < numItm
END Search;
```

Figure 16.2
An efficient version of the sequential search algorithm.

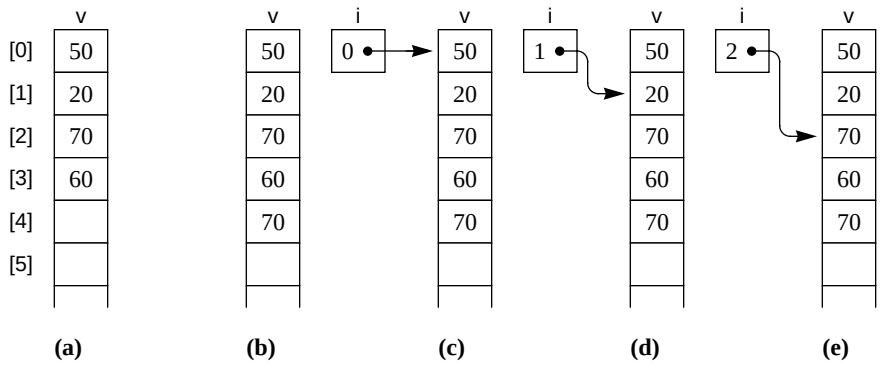


Figure 16.3
A trace of the sequential search algorithm when the value of srchNum, 70, is in the list.

Figure 16.2 shows a better version of the sequential search algorithm. The algorithm puts the value of srchNum at the end of the list to act as a sentinel, if necessary. Formal parameter v must be called by reference instead of by constant reference, because the algorithm modifies v. Also, the precondition requires numItm to be strictly less than LEN(v). It cannot be equal to LEN(v) because the algorithm reserves the last spot in the array for the sentinel value.

Figure 16.3 is a trace of the algorithm when the item searched is in the list. i never reaches numItm. Figure 16.4 is a trace when the item is not in the list. This time, i reaches numItm, and the value of srchNum acts like a sentinel.

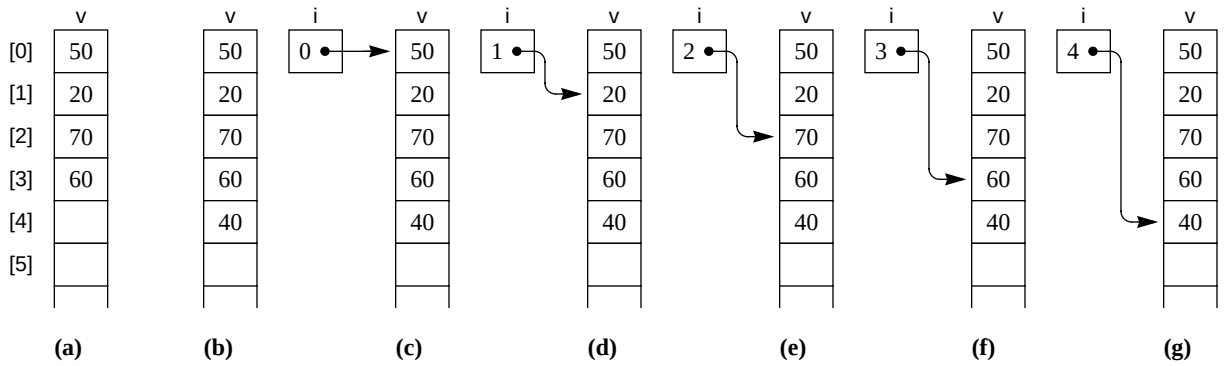


Figure 16.4
A trace of the sequential search algorithm when the value of srchNum, 40, is not in the list.

How much better is this version of the sequential search? Counting the comparisons as in the analysis of the previous version gives the following performance figures:

- Best case: 1 comparison
- Worst case: $n + 1$ comparisons

If you search a list of 1000 items, in the best case you will make 1 comparison instead of 2 and in the worst case you will make 1001 comparisons instead of 2001. This version is substantially better than the first and is the one you should use whenever you need to do a sequential search.

Tool dialog boxes

In practice, you rarely will have to search for a numerical value in a single list of numbers. A more common need is to look up someone's name to retrieve additional information about the person. For example, you may want to search a list of names for a particular name to find the corresponding telephone number.

Figure 16.5 shows a dialog box for such a problem. It consists of two groups of controls, one to load the telephone book and one to perform a query. The figure shows how to load a phone book. A window contains the phone book, which consists of a list of names followed by phone numbers. Each name and number is enclosed in double quotes, but single quotes could be used consistently as well. To load the phone book, the user must first make the phone book window the focus window, then make the dialog box the focus

window, the user clicks the Load Phone Book button. A procedure then scans the names and numbers from the phone book window into two arrays, one for the names and one for the numbers. The dialog box displays how many entries from the book have been scanned into the phone book.

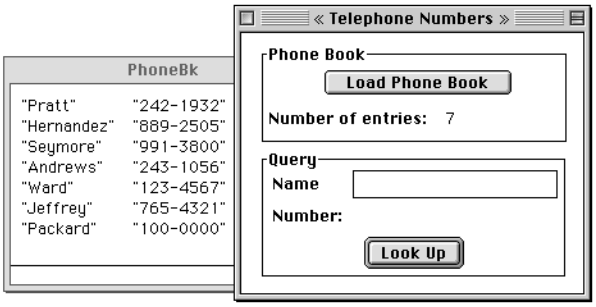


Figure 16.5
A tool dialog box.

You can tell from Figure 16.5 that the dialog box is the focus window because it overlaps with the phone book window and appears on top of it. When the user clicks the Load Phone Book button, how does the BlackBox framework know which window to scan? It cannot scan from the focus window, because when the user clicks the button the dialog box is the focus window. Somehow, the framework must remember which window was the focus window just *before* the dialog box became the focus window.

The framework does not consider a tool dialog box to be the focus window.

Tool dialog boxes have the property that they are not considered by the framework to be the focus window. Instead, the framework considers the window that was focused immediately before the tool dialog box is activated to be the focus window. You construct a tool dialog box the same way to construct any dialog box. But instead of opening it as an auxiliary dialog box with procedure StdCmds.OpenAuxDialog, you open it as a tool dialog box with procedure StdCmds.OpenToolDialog.

One possibility is to have the documentation section provide a commander that when clicked will activate the tool dialog box. Although this technique is convenient for users of the BlackBox framework, a more conventional GUI technique is to provide the user with a menu selection to activate the tool dialog box.

The tool dialog box in Figure 16.5 was opened with the menu script

```
MENU "Pbox17"
   "A..." "" "StdCmds.OpenToolDialog('Pbox17/Rsrc/DlgA', 'Telephone Numbers')" ""
END
```

When you provide a menu selection to activate a dialog box, you should include the ellipsis … to give the user a cue that when the menu item is selected a dialog box will appear.

Figure 16.6 shows how the user enters a query by typing a name and clicking the Look Up button. In case the name is not found in the phone book an appropriate message appears in the dialog box.

It is clear from the dialog box that the module will need to export five items for the five controls—a procedure to load the phone book, an output integer for the

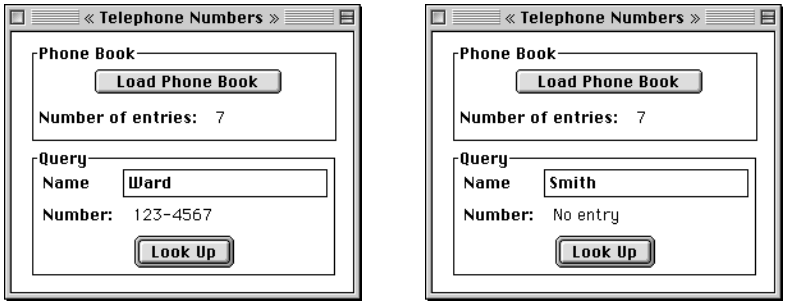


Figure 16.6
Performing a query.

number of items loaded, an input field for the name, an output field for the number, and a procedure for performing the search. In addition, a global data structure is needed for the phone book, which consists of an array of names and an array of numbers. Figure 16.7 shows the array of names and the corresponding array of numbers. They are known as parallel arrays because the number numberList[i] goes with the name nameList[i]. Figure 16.8 shows the module that implements the dialog box.

nameList

[0]	Pratt
[1]	Hernandez
[2]	Seymore

numberList

[0]	242-1932
[1]	889-2505
[2]	991-3800

Figure 16.7
The parallel arrays for the phone book in Figure 16.8.

```
MODULE Pbox16A;
   IMPORT Dialog, TextModels, TextControllers, PboxMappers;
   TYPE
      Name = ARRAY 32 OF CHAR;
      Number = ARRAY 16 OF CHAR;
   VAR
      d*: RECORD
         numEntries-: INTEGER;
         name*: Name;
         number-: Number
      END;
   CONST
      maxItems = 1024;
   VAR
      nameList: ARRAY maxItems OF Name;
      numberList: ARRAY maxItems OF Number;
```

Figure 16.8
Using the sequential search to look up a phone number.

Figure 16.8
Continued.

```
    PROCEDURE LoadBook*;
        VAR
            md: TextModels.Model;
            cn: TextControllers.Controller;
            sc: PboxMappers.Scanner;
            i: INTEGER;
    BEGIN
        cn := TextControllers.Focus();
        IF cn # NIL THEN
            md := cn.text; sc.ConnectTo(md);
            i := 0;
            sc.ScanString(nameList[i]); sc.ScanString(numberList[i]);
            WHILE ~sc.eot DO
                INC(i);
                sc.ScanString(nameList[i]); sc.ScanString(numberList[i])
            END;
            d.numEntries := i
        END;
        d.number := "";
        Dialog.Update(d)
    END LoadBook;

    PROCEDURE Search (VAR v: ARRAY OF Name; numItm: INTEGER; IN srchName: Name;
            OUT i: INTEGER; OUT fnd: BOOLEAN);
    BEGIN
        ASSERT((0 <= numItm) & (numItm < LEN(v)), 20);
        v[numItm] := srchName$;
        i := 0;
        WHILE v[i] # srchName DO
            INC(i)
        END;
        fnd := i < numItm
    END Search;

    PROCEDURE LookUp*;
        VAR
            j: INTEGER;
            found: BOOLEAN;
    BEGIN
        Search(nameList, d.numEntries, d.name, j, found);
        IF found THEN
            d.number := numberList[j]$
        ELSE
            d.number := "No entry"
        END;
        Dialog.Update(d)
    END LookUp;

BEGIN
    d.numEntries := 0; d.name := ""; d.number := ""
END Pbox16A.
```

The binary search

When you look up a word in a dictionary, you do not use the sequential search. If you want to look up the word walrus, you do not start at the front of the book and look sequentially from the first entry on. Instead, you use the fact that the words are in alphabetical order. You open the book to an arbitrary place and look at a word. If walrus is less than the word you opened to, you know that walrus lies in the section before your place in the book. Otherwise it is in the back part.

This common idea is the basis of the binary search. To perform a binary search the list must be in order. The algorithm makes the initial selection at the midpoint of the list. After the first comparison, the algorithm knows which half of the list the item must be in.

The second comparison is at the midpoint of the proper half. After this comparison the algorithm knows which quarter of the list the item must be in. The algorithm continues to split the known region in half until it finds the value or determines that it is not in the list. It gets the name "binary" from the fact that it divides the list into two equal parts with each comparison.

Figure 16.9 shows the binary search algorithm. The variables for this algorithm are the same as those for the sequential search, except that three local indices are necessary—first, mid, and last—instead of one index, i.

```
PROCEDURE Search (IN v: ARRAY OF INTEGER; numItm, srchNum: INTEGER;
      OUT i: INTEGER; OUT fnd: BOOLEAN);
   VAR
      first, mid, last: INTEGER;
BEGIN
   ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
   first := 0;
   last := numItm - 1;
   WHILE first <= last DO
      mid := (first + last) DIV 2;
      IF srchNum < v[mid] THEN
         last := mid - 1
      ELSIF srchNum > v[mid] THEN
         first := mid + 1
      ELSE
         fnd := TRUE; i := mid;
         RETURN
      END
   END;
   fnd := FALSE
END Search;
```

Figure 16.9
The binary search algorithm.

The variables first and last will keep track of the boundaries of the list within which the search value must lie, if it is in the list at all. The algorithm initializes first to zero and last to numItm - 1. At the beginning of the loop, if srchNum is in v then

```
(v[first] <= srchNum) & (srchNum <= v[last])
```

will be true. This assertion is the loop invariant. The algorithm works by keeping the loop invariant true each time the loop executes.

The variable mid is the midpoint between first and last. The algorithm compares v[mid] with srchNum. Depending on the test, it updates either first or last such that the value is still between v[first] and v[last]. When the algorithm terminates, mid is the index of the cell of v that contains srchNum. Figure 16.10 is a trace of the algorithm when the values of v are

10 30 40 50 60 70 90

and srchNum is 40, a value in the list. The algorithm initializes first to 0 and last to 6. mid gets (0 + 6) DIV 2, which is 3, the midpoint between 0 and 6.

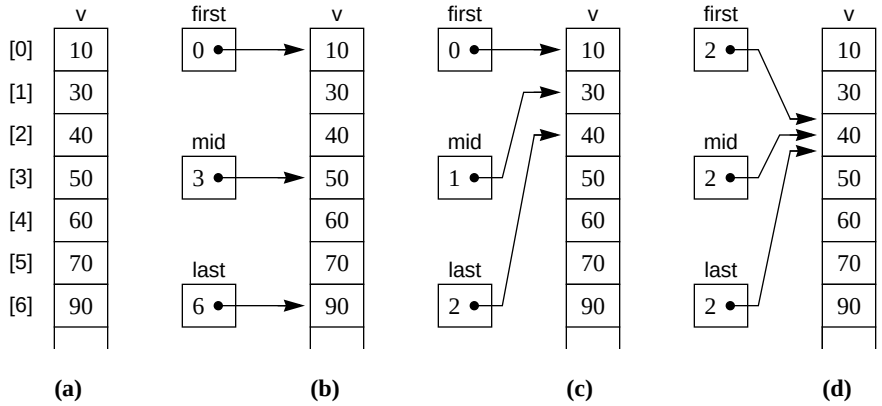


Figure 16.10
A trace of the binary search algorithm when the value of srchNum, 40, is in the list.

The IF statement compares srchNum, 40, with v[mid], 50. Because srchNum is less than v[mid], the algorithm knows that the value cannot be in the bottom half of the list. So it updates last to 1 less than mid, which is 2. Notice how the loop invariant is still true. The search value, if it is in the list, must be between v[0] and v[2].

The next time through the loop mid gets 1, which is the midpoint between 0 and 2. After the comparison, first gets 2. The next time through the loop, mid gets 2 also, and the loop terminates because v[2] has the same value as srchNum.

This algorithm is the first one we have encountered that has the RETURN statement in a proper procedure. Previous examples have always used the RETURN statement in function procedures, where RETURN is followed by an expression for the value to be returned. Because procedure Search in Listing 16.9 is not a function procedure, there is no value to be returned. Therefore, there is no expression following the RETURN statement. The RETURN statement serves to exit the procedure immediately, in this case before reaching the end of the procedure. If the loop terminates because first <= last then fnd will be set to false. On the other hand, if srchNum = v[mid] then fnd will be set to true, and the loop and the procedure will terminate immediately.

The RETURN statement in a proper procedure

Figure 16.11 is a trace with the same values as Figure 16.10 for v, but with a value of 80 for srchNum, which is not in the list. This time first, mid, and last eventually all get 6. Because srchNum, 80, is less than v[mid], 90, the algorithm sets last to

mid - 1, which is 5. In the figure, that causes first and last to cross. Mathematically, first > last, which causes the loop to terminate.

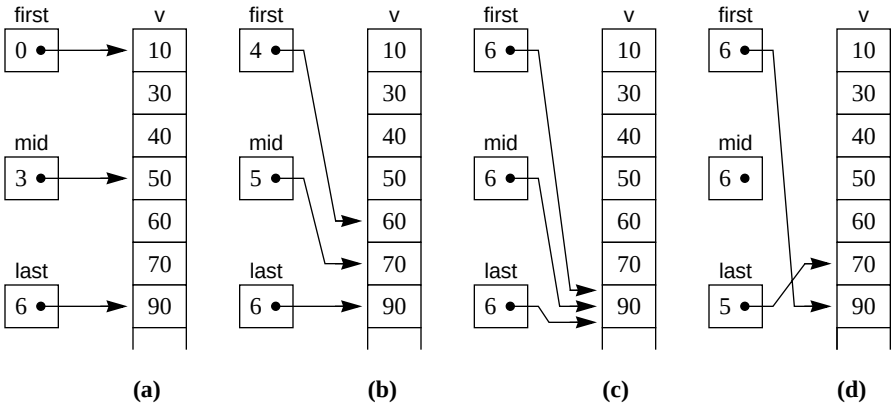


Figure 16.11
A trace of the binary search algorithm when the value of srchNum, 80, is not in the list.

Does this algorithm look familiar? The bisection algorithm in Chapter 10 for finding the root of an equation was essentially a binary search. Figure 10.9 shows the variables left, mid, and right, which correspond directly to first, mid, and last.

The difference is that the bisection algorithm searches for a real value on the continuous number line, while the binary search algorithm searches for a discrete value in an ordered array. The bisection algorithm updates the real variable mid with

```
mid := (left + right) / 2.0
```

while the binary search algorithm updates the integer variable mid with

```
mid := (first + last) DIV 2
```

How fast is this algorithm? As with the sequential search, that depends on several factors. The loop will terminate as soon as srchNum equals v[mid]. The best case is when they are equal the first time, which happens if the value is exactly in the middle of the list. The number of comparisons will be 3.

The worst case is when the value is not in the list, the comparison in the test of the WHILE loop is made every time the body executes, and both comparisons in the IF statement are made every time the body executes. That happens if srchNum is always greater than v[mid]. The worst case with the values of v that were just presented occurs with a value of 95 for srchNum.

Each time the loop executes, it makes three comparisons. So the total number of comparisons is three times the number of times the loop executes. If numItm has the value *n*, how many times will the loop execute? The answer to that question is the same as the answer to the question, How many times must you cut an integer, *n*, in half to get to one? After all, each time the loop executes it eliminates half the possible locations for the value.

If you do not know the answer, you can use the problem-solving technique of going from the specific to the general. Let *t* equal the number of times. Here are some specific values of *n* and *t*:

n = 16, t = 4 times: 16 8 4 2 1
n = 32, t = 5 times: 32 16 8 4 2 1
n = 64, t = 6 times: 64 32 16 8 4 2 1

You can see that the general relationship between n and t is $n = 2^t$. This relationship is approximately true even if n is not an exact power of 2. For example, if n is 40, the number of times you must halve it in order to get to 1 is either 5 or 6.

In mathematics, the logarithm to the base 10 is denoted log, and the logarithm to the base e is denoted ln. In computer science, logarithms are usually to the base 2. The logarithm to the base 2 is denoted lg. In mathematical notation, $\log_2 n = \lg n$

lg is the logarithm to the base 2.

The relationship between t and n can be written $t = \lg n$ by the definition of the logarithm. This is the number of times the loop executes in the worst case. Each time the loop executes, it makes three comparisons. So the number of comparisons is $3\lg n$.

Summarizing, the binary search has the following performance figures:

- Best case: 3 comparisons
- Worst case: $3\lg n$ comparisons

The average case, however you define it, is somewhere between the best case and the worst case.

The selection sort

Sort algorithms put lists of values in order. For example, if numItm is 9 and the first 9 values of v are

7 3 8 2 1 4 9 5 6

then after the sort, the nine values should be

1 2 3 4 5 6 7 8 9

The selection sort is based on the idea of finding the largest item in the list and putting it at the end. Then it finds the next largest and puts it next to the end. It continues finding the largest item in the top part of the list and putting it at the end of the top part until it gets to the top. Here is the first outline of the algorithm.

```
FOR k := numItm - 1 TO 1 BY -1 DO
   Select the largest element from the top part of the list between v[0] and v[k].
   Exchange it with v[k].
END
```

The first time the loop executes, k gets numItm - 1. So the largest element in the entire list is exchanged with the element at the bottom of the list. After this first execution of the loop, the largest element is in its correct position in the sorted list.

The second time the loop executes, k gets numItm - 2. So the largest element between v[0] and v[numItm - 2] is exchanged with v[numItm - 2]. At this point in the

execution, the bottom two elements will be at their correct positions in the list.

Similarly, after the third execution of the loop the bottom three elements will be in their correct positions. The last time, k gets 1, after which the bottom k - 1 elements will be in their correct order. Therefore, the top one must be in its correct order also.

How do you put the largest element between v[0] and v[k] into v[k]? This is precisely the problem solved by procedure LargestLast in Figure 15.12. Inserting the code for that processing into the outer loop yields the algorithm in Figure 16.12 for the selection sort.

```
PROCEDURE Sort (VAR v: ARRAY OF INTEGER; numItm: INTEGER);
   VAR
      i, k: INTEGER;
      maxIndex: INTEGER;
      temp: INTEGER;
BEGIN
   ASSERT((0 <= numItm) & (numItm <= LEN(v)), 20);
   FOR k := numItm - 1 TO 1 BY -1 DO
      maxIndex := 0;
      FOR i := 1 TO k DO
         IF v[i] > v[maxIndex] THEN
            maxIndex := i
         END
      END;
      temp := v[k];
      v[k] := v[maxIndex];
      v[maxIndex] := temp
   END
END Sort;
```

Figure 16.12
The selection sort.

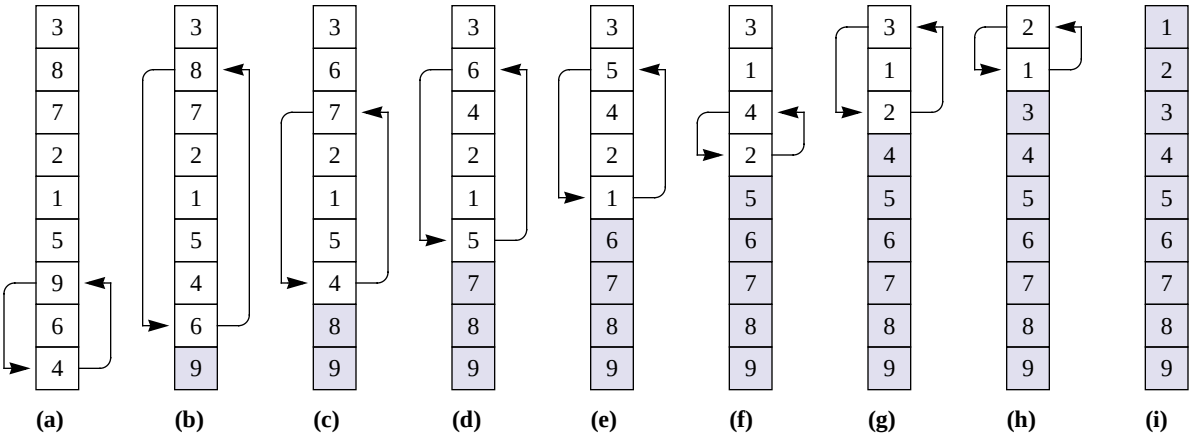
Figure 16.13 is a trace of the outer loop of the selection sort algorithm. Figure 16.13(a) shows the original list. When k has the value 8 the first time the outer loop executes, the inner loop computes maxIndex as 6, the index of the largest element between v[0] and v[8]. The algorithm exchanges v[8] with v[6].

Figure 16.13(b) shows the list after the first exchange. The second time the outer loop executes, maxIndex is computed as 1, after which the algorithm exchanges v[1] with v[7]. At this point in the execution, the last two values are in order. The outer loop executes eight times, after which the entire list is in order.

How fast is the selection sort algorithm? Two criteria are common in the analysis of sort algorithms. One criterion counts the number of comparisons performed by the algorithm. The other counts the number of exchanges. In the selection sort, it is easier to count the number of comparisons.

The inner loop makes one comparison each time it executes. So the number of comparisons is the number of times the inner loop executes. Let n be the value of numItm. The first time the outer loop executes, the inner loop executes $n-1$ times. The second time the outer loop executes, the inner loop executes one less time, which is $n-2$. The third time the outer loop executes, the inner loop executes $n-3$ times, and so on.

Figure 16.13
Eight executions of the outer loop of the selection sort. The shaded cells contain the values of the array that are in order.



So, the number of comparisons is

$$(n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$$

which is the sum of the first $n-1$ integers. You should recognize this problem from the statement execution count for the algorithm that prints a triangle. Remember that the sum of the first m integers is $m(m+1)/2$. So the sum of the first $n-1$ integers is $(n-1)(n-1+1)/2$ or $n(n-1)/2$, which is the number of comparisons the selection sort makes to sort n items.

Example 16.1 To sort 100 items requires $100(99)/2 = 4950$ comparisons. If you double the number of items to 200, the algorithm makes $200(199)/2 =$ 19,900 comparisons. ∎

So, doubling the number of items more than doubles the number of comparisons. Why? Because the number of comparisons is not linear in n. It is quadratic in n, because $n(n-1)/2$ is the same as $(n^2-n)/2$. That means doubling the number of items will approximately quadruple the number of comparisons. Four times 4950 is approximately 19,900.

Exercises

1. Determine **(a)** the best case, and **(b)** the worst case execution count of each statement in procedure Search of Figure 16.1 assuming numItm has value n. Calculate the total statement execution count as a polynomial in n. Because there are two tests in the WHILE statement, count that statement as executing twice each time it executes.

2. Determine **(a)** the best case, and **(b)** the worst case execution count of each statement in procedure Search of Figure 16.2 assuming numItm has value n. Calculate the total statement execution count as a polynomial in n.

3. Determine **(a)** the best case, and **(b)** the worst case execution count of each statement in procedure Sort of Figure 16.12 assuming numItm has value *n*. Calculate the total statement execution count as a polynomial in *n*.

4. **(a)** For the search algorithm of Figure 16.1, translate the statements

```
i := 0;
WHILE (i < numItm) & (v[i] # srchNum) DO
    INC(i)
END;
fnd := i < numItm
```

from CP to GCL. **(b)** Write a formal specification for the code fragment. Use the fact that if there exists a j in the proper range such that $v[j] = s$ then $s = v[i]$, where s is the search number.

5. **(a)** For the search algorithm of Figure 16.2, translate the statements

```
v[numItm] := srchNum;
i := 0;
WHILE v[i] # srchNum DO
    INC(i)
END;
fnd := i < numItm
```

from CP to GCL. **(b)** Write a formal specification for the code fragment. Use the fact that if there exists a j in the proper range such that $v[j] = s$ then $s = v[i]$, where s is the search number.

6. **(a)** For the binary search algorithm of Figure 16.8, translate the statements

```
first := 0;
last := numItm - 1;
WHILE first <= last DO
    mid := (first + last) DIV 2;
    IF srchNum < v[mid] THEN
        last := mid - 1
    ELSIF srchNum > v[mid] THEN
        first := mid + 1
    ELSE
        fnd := TRUE; i := mid;
        RETURN
    END
END;
fnd := FALSE
```

from CP to GCL. You may assume that GCL has a **return** statement. **(b)** Write a formal specification for the code fragment. Use the fact that if there exists a j in the proper range such that $v[j] = s$ then $s = v[i]$, where s is the search number.

7. **(a)** For the selection sort algorithm of Figure 16.12, translate the statements

```
FOR n := numItm - 1 TO 1 BY -1 DO
    maxIndex := 0;
    FOR i := 1 TO n DO
        IF v[i] > v[maxIndex] THEN
            maxIndex := i
        END
    END;
    temp := v[n];
    v[n] := v[maxIndex];
    v[maxIndex] := temp
END
```

from CP to GCL. **(b)** Write a formal specification for the code fragment. You may use the predicate $perm(a,b,numItm)$ to specify that the final values of *v* are a rearrangement of thed initial values of *v.*

Problems

8. Write a procedure

OddFirstSort (VAR v: ARRAY OF INTEGER; numItm: INTEGER)

that rearranges the elements of a list of integers so all the odd integers are before all the even integers. To test it, write a program similar to that in Figure 15.7, with input from the focus window using ScanIntVector from module PboxMappers and OddFirstSort taking the place of RotateLeft. The new window should contain the original list followed by the rearranged list. Activate your procedure with a menu selection.

9. Write a program to input values from a focus window similar to the one titled PhoneBk in Figure 16.5 into the parallel arrays nameList and numberList as defined in Figure 16.8. Declare a procedure

SortBook (VAR nameLst: ARRAY OF Name; VAR numLst: ARRAY OF Number; numItm: INTEGER)

that sorts the parallel arrays based on the values in nameLst. For example, after the sort nameList[0] should be Andrews, and numList[0] should be 243-1056. Output the sorted names with the corresponding numbers next to them on the Log. Activate your procedure with a menu selection.

10. Modify Figure 16.8 so that it uses the binary search instead of the sequential search. Assume the names are in order in the phone book window.

11. The input for this problem is in a focus window that contains a list of up to 1024 integers. Write a program that inputs the list, then outputs to the Log a list of those integers that occur more than once and the number of times each occurs. Activate your procedure with a menu selection. Hint: First sort the list. Also consider the possibility that the last item in the list might be duplicated. Considering that case, you may be able to simplify your code by appending an extra value after the last item of the list, similar to the technique of the sequential search in Figure 16.2.

Sample focus window:

```
 33  -2  25  25   3   7  -2  17  12  25  33   8  17   2  17  20  25
```

Sample output to the Log:
-2 occurs 2 times
17 occurs 3 times
25 occurs 4 times
33 occurs 2 times

12. Modify procedure Sort in Figure 16.12 so that it sorts with the largest element first. To test it, write a program similar to that in Figure 15.7, with input from the focus window using ScanIntVector from module PboxMappers and Sort taking the place of RotateLeft. The new window should contain the original list followed by the sorted list. Activate your procedure with a menu selection.

13. Modify procedure Sort in Figure 16.12 so that it moves the smallest element to v[0] on the first pass, the next larger to v[1] on the second pass, and so on. To test it, write a program similar to that in Figure 15.7, with input from the focus window using ScanIntVector from module PboxMappers and Sort taking the place of RotateLeft. The new window should contain the original list followed by the sorted list. Activate your procedure with a menu selection.

14. Declare

```
PROCEDURE Compress (VAR v: ARRAY OF INTEGER; VAR numItm)
```

which removes duplicate integers from a sorted list of integers. For example, if numItm is 12 and v is

1 4 4 5 9 9 9 14 19 19 19 19

then Compress should change v to

1 4 5 9 14 19

and numItm to 6. To test it, write a program similar to that in Figure 15.7, with input from the focus window using ScanIntVector from module PboxMappers and Compress taking the place of RotateLeft. The new window should contain the original list followed by the compressed list. Activate your procedure with a menu selection. Do not include any output statements in Compress.

15. Anagrams are words that use the same letters. For example, races and acres are anagrams, but car and cat are not. Declare

```
PROCEDURE IsAnagram (wd1, wd2: ARRAY OF CHAR): BOOLEAN
```

which determines if wd1 and wd2 are anagrams. Test your function with a dialog box that has two input fields for the words and one output field for a message stating whether the two words are anagrams.