CHAPTER 9

Searching, Sorting, and Algorithm Analysis

TOPICS

9.1 Introduction to Search Algorithms

CONCEPT: **A search algorithm is a method of locating a specific item in a collection of data.**

It's very common for programs not only to store and process data stored in arrays, but to search arrays for specific items. This section will show you two methods of searching an array: the linear search and the binary search. Each has its advantages and disadvantages.

The Linear Search

The *linear search* is a very simple algorithm. Sometimes called a *sequential search*, it uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm will search to the end of the array.

Here is the pseudocode for a function that performs the linear search:

```
Set found to false
Set position to -1
Set index to 0
While index < number of elements and found is false
   If list[index] is equal to search value
      found = true
      position = index
   End If
   Add 1 to index
End While
Return position
```

The function `searchList`, which follows, is an example of C++ code used to perform a linear search on an integer array. The array `list`, which has a maximum of `size` elements, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, –1 is returned, indicating the value did not appear in the array.

```
int searchList(const int list[], int size, int value)
{
   int index = 0;          // Used as a subscript to search array
   int position = -1;      // Used to record position of search value
   bool found = false;     // Flag to indicate if the value was found

   while (index < size && !found)
   {
      if (list[index] == value)     // If the value is found
      {
          found = true;             // Set the flag
          position = index;         // Record the value's subscript
      }
      index++;                      // Go to the next element
   }
   return position;                 // Return the position, or -1
}
```

NOTE: The reason –1 is chosen to indicate that the search value was not found in the array is that –1 is not a valid subscript. Any other nonvalid subscript value could also have been used to signal this.

Program 9-1 is a complete program that uses the `searchList` function. It searches the five-element `tests` array to find a score of 100.

Program 9-1

```
1 // This program demonstrates the searchList function,
2 // which performs a linear search on an integer array.
3 #include <iostream>
4 using namespace std;
5
```

(program continues)

Program 9-1 *(continued)*

```
// Function prototype
int searchList(const int [], int, int);

const int SIZE = 5;

int main()
{
   int tests[SIZE] = {87, 75, 98, 100, 82};
   int results;          // Holds the search results

   // Search the array for the value 100
   results = searchList(tests, SIZE, 100);

   // If searchList returned -1, 100 was not found
   if (results == -1)
      cout << "You did not earn 100 points on any test.\n";
   else
   {  // Otherwise results contains the subscript of
      // the first 100 found in the array
      cout << "You earned 100 points on test ";
      cout << (results + 1) << ".\n";
   }
   return 0;
}

/**************************************************************
 *                          searchList                        *
 * This function performs a linear search on an integer array.   *
 * The list array, which has size elements, is searched for      *
 * the number stored in value. If the number is found, its array *
 * subscript is returned. Otherwise, -1 is returned.             *
 **************************************************************/
int searchList(const int list[], int size, int value)
{
   int index = 0;           // Used as a subscript to search array
   int position = -1;       // Used to record position of search value
   bool found = false;      // Flag to indicate if the value was found

   while (index < size && !found)
   {
      if (list[index] == value)  // If the value is found
      {
         found = true;           // Set the flag
         position = index;       // Record the value's subscript
      }
      index++;                   // Go to the next element
   }
   return position;              // Return the position, or -1
}
```

Program Output

```
You earned 100 points on test 4.
```

Inefficiency of the Linear Search

The advantage of the linear search is its simplicity. It is very easy to understand and implement. Furthermore, it doesn't require the data in the array to be stored in any particular order. Its disadvantage, however, is its inefficiency. If the array being searched contained 20,000 elements, the algorithm would have to look at all 20,000 elements in order to find a value stored in the last element or to determine that a desired element was not in the array.

In a typical case, an item is just as likely to be found near the beginning of the array as near the end. On average, for an array of N items, the linear search will locate an item in N/2 attempts. If an array has 20,000 elements, the linear search will make a comparison with 10,000 of them on average. This is assuming, of course, that the search item is consistently found in the array. (N/2 is the average number of comparisons. The maximum number of comparisons is always N.)

When the linear search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. When it can be avoided the linear search should not be used on large arrays if speed is important.

The Binary Search

Performing a Binary Search

The *binary search* is a clever algorithm that is much more efficient than the linear search. Its only requirement is that the values in the array be in order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater than the desired value then the value (if it is in the list) will be found somewhere in the first half of the array. If it is less than the desired value then the value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

If the desired value wasn't found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm immediately tests *its* middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until the value being searched for is either found or there are no more elements to test.

Here is the pseudocode for a function that performs a binary search on an array whose elements are stored in ascending order.

```
Set first to 0
Set last to the last subscript in the array
Set found to false
Set position to -1
```

```
While found is not true and first is less than or equal to last
   Set middle to the subscript halfway between first and last
   If array[middle] equals the desired value
       Set found to true
       Set position to middle
   Else If array[middle] is greater than the desired value
       Set last to middle - 1
   Else
       Set first to middle + 1
   End If
End While
Return position
```

This algorithm uses three index variables: `first`, `last`, and `middle`. The `first` and `last` variables mark the boundaries of the portion of the array currently being searched. They are initialized with the subscripts of the array's first and last elements. The subscript of the element approximately halfway between `first` and `last` is calculated and stored in the `middle` variable. If there is no precisely central element, the integer division used to calculate `middle` will select the element immediately preceding the midpoint. If the element in the middle of the array does not contain the search value, the `first` or `last` variables are adjusted so that only the top or bottom half of the array is searched during the next iteration. This cuts the portion of the array being searched in half each time the loop fails to locate the search value.

The function `binarySearch` in the following example C++ code is used to perform a binary search on an integer array. The first parameter, `array`, which has `size` elements, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, –1 is returned indicating the value did not appear in the array.

```
int binarySearch(const int array[], int size, int value)
{
   int  first = 0,                    // First array element
        last = size - 1,              // Last array element
        middle,                       // Midpoint of search
        position = -1;                // Position of search value
   bool found = false;                // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;    // Calculate midpoint
      if (array[middle] == value)     // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;          // If value is in upper half
   }
   return position;
}
```

Program 9-2 is a complete program using the `binarySearch` function. It searches an array of employee ID numbers for a specific value.

Program 9-2

```
1  // This program performs a binary search on an integer
2  // array whose elements are in ascending order.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  int binarySearch(const int [], int, int);
8
9  const int SIZE = 20;
10
11 int main()
12 {
13    // Create an array of ID numbers sorted in ascending order
14    int IDnums[SIZE] = {101, 142, 147, 189, 199, 207, 222,
15                        234, 289, 296, 310, 319, 388, 394,
16                        417, 429, 447, 521, 536, 600 };
17
18    int empID,          // Holds the ID to search for
19        results;        // Holds the search results
20
21    // Get an employee ID to search for
22    cout << "Enter the employee ID you wish to search for: ";
23    cin  >> empID;
24
25    // Search for the ID
26    results = binarySearch(IDnums, SIZE, empID);
27
28    // If binarySearch returned -1, the ID was not found
29    if (results == -1)
30       cout << "That number does not exist in the array.\n";
31    else
32    {  // Otherwise results contains the subscript of
33       // the specified employee ID in the array
34       cout << "ID " << empID << " was found in element "
35            << results << " of the array.\n";
36    }
37    return 0;
38 }
39
```

(program continues)

Program 9-2 *(continued)*

```
40 /**************************************************************
41  *                       binarySearch                         *
42  * This function performs a binary search on an integer array    *
43  * with size elements whose values are stored in ascending       *
44  * order. The array is searched for the number stored in the     *
45  * value parameter. If the number is found, its array subscript  *
46  * is returned. Otherwise, -1 is returned.                       *
47  **************************************************************/
48 int binarySearch(const int array[], int size, int value)
49 {
50    int  first = 0,                    // First array element
51         last = size - 1,              // Last array element
52         middle,                       // Midpoint of search
53         position = -1;                // Position of search value
54    bool found = false;                // Flag
55
56    while (!found && first <= last)
57    {
58       middle = (first + last) / 2;    // Calculate midpoint
59       if (array[middle] == value)     // If value is found at mid
60       {
61          found = true;
62          position = middle;
63       }
64       else if (array[middle] > value) // If value is in lower half
65          last = middle - 1;
66       else
67          first = middle + 1;          // If value is in upper half
68    }
69    return position;
70 }
```

Program Output with Example Input Shown in Bold

```
Enter the employee ID you wish to search for: 199[Enter]
ID 199 was found in element 4 of the array.
```

The Efficiency of the Binary Search

Obviously, the binary search is much more efficient than the linear search. Every time it makes a comparison and fails to find the desired item, it eliminates half of the remaining portion of the array that must be searched. For example, consider an array with 20,000 elements. If the binary search fails to find an item on the first attempt, the number of elements that remains to be searched is 10,000. If the item is not found on the second attempt, the number of elements that remains to be searched is 5,000. This process continues until the binary search locates the desired value or determines that it is not in the array. With 20,000 elements in the array, this takes a maximum of 15 comparisons. (Compare this to the linear search, which would make an average of 10,000 comparisons!)

Powers of 2 are used to calculate the maximum number of comparisons the binary search will make on an array of any size. (A power of 2 is 2 raised to some integer exponent.) Simply find the smallest power of 2 that is greater than the number of elements in the array. That will tell you the maximum number of comparisons needed to find an element, or to determine that it is not present. For example, a maximum of 16 comparisons will be made to find an item in an array of 50,000 elements ($2^{16} = 65{,}536$), and a maximum of 20 comparisons will be made to find an item in an array of 1,000,000 elements ($2^{20} = 1{,}048{,}576$).

9.2 Searching an Array of Objects

CONCEPT: Linear and binary searches can also be used to search for a specific entry in an array of objects or structures.

In Programs 9-1 and 9-2 we searched for a particular value in an array of integers. We can just as easily search through an array holding values of some other data type, such as `double` or `string`. We can even search an array of objects or structures. In this case, however, the search value is not the entire object or structure we are looking for, but rather a value in a particular member variable of that object or structure. The member variable being examined by the search is sometimes called the *key field*, and the particular value being looked for is called the *search key*.

Assume we have a class named `Inventory` that includes the following member variables

```
string itemCode;
string description;
double price;
```

as well as methods to *set* and *get* the value of each of these. Assume also that we have set up an array of `Inventory` objects. We might want to search for a particular object in the array, say the object whose `itemCode` is K33, so that we can then call the `getPrice` method for that object. Program 9-3 illustrates how to do this. It searches the array of `Inventory` objects using a `search` function similar to the `searchList` function we used earlier in this chapter. However, it has been modified to work with an array of `Inventory` objects.

Program 9-3

```
1  // This program searches an array of Inventory objects to get
2  // the price of a particular object. It demonstrates how to
3  // perform a linear search using an array of objects.
4  #include <iostream>
5  #include <string>
6  using namespace std;
7
8  // Inventory class declaration
9  class Inventory
10 {   private:
11       string itemCode;
12       string description;
13       double price;
14
```

(program continues)

Program 9-3 *(continued)*

```
15   public:
16      Inventory()                              // Default constructor
17      {  itemCode = "XXX";  description = " ";  price = 0.0; }
18
19      Inventory(string c, string d, double p) // 3 argument constructor
20      {  itemCode = c;
21         description = d;
22         price = p;
23      }
24
25      // Add methods setCode, setDescription, and setPrice here.
26
27      // Get functions to retrieve member variable values
28      string getCode() const
29      {  string code = itemCode;
30         return code;
31      }
32
33      string getDescription() const
34      {  string d = description;
35         return d;
36      }
37
38      double getPrice() const
39      {  return price;
40      }
41
42 }; // End Inventory class declaration
43
44 // Program that uses the Inventory class
45
46 // Function prototype
47 int search(const Inventory[], int, string);
48
49 /*****************************************************
50  *                      main                         *
51  *****************************************************/
52 int main()
53 {
54    const int SIZE = 6;
55
56    // Create and initialize the array of Inventory objects
57    Inventory silverware[SIZE] =
58                              { Inventory("S15", "soup spoon",  2.35),
59                                Inventory("S12", "teaspoon",    2.19),
60                                Inventory("F15", "dinner fork", 3.19),
61                                Inventory("F09", "salad fork" , 2.25),
62                                Inventory("K33", "knife",       2.35),
63                                Inventory("K41", "steak knife", 4.15) };
64
65    string desiredCode;     // The itemCode to search for
66    int pos;                // Position of desired object in the array
67    char doAgain;           // Look up another price (Y/N)?
```

(program continues)

Program 9-3 *(continued)*

```
68
69    do
70    {   // Get the itemCode to search for
71        cout << "\nEnter an item code: ";
72        cin  >> desiredCode;
73
74        // Search for the object
75        pos = search(silverware, SIZE, desiredCode);
76
77        // If pos = -1, the code was not found
78        if (pos == -1)
79           cout << "That code does not exist in the array\n";
80        else
81        {  // The object was found, so use pos to get the
82           // description and price
83           cout << "This "    << silverware[pos].getDescription()
84                << " costs $" << silverware[pos].getPrice() << endl;
85        }
86
87        // Does the user want to look up another price?
88        cout << "\nLook up another price (Y/N)? ";
89        cin  >> doAgain;
90
91    } while (doAgain == 'Y' || doAgain == 'y');
92    return 0;
93 }// End main
94
95 /*************************************************************
96  *                          search                           *
97  * This function performs a linear search on an array of     *
98  * Inventory objects, using itemCode as the key field.       *
99  * If the desired code is found, its array subscript is      *
100 * returned. Otherwise, -1 is returned.                      *
101 *************************************************************/
102 int search(const Inventory object[], int size, string value)
103 {
104    int index = 0;          // Used as a subscript to search array
105    int position = -1;      // Used to record position of search value
106    bool found = false;     // Flag to indicate if the value was found
107
108    while (index < size && !found)
109    {
110       if (object[index].getCode() == value) // If the value is found
111       {
112          found = true;           // Set the flag
113          position = index;       // Record the value's subscript
114       }
115       index++;                   // Go to the next element
116    }
117    return position;              // Return the position, or -1
118 }// End search
```

(program continues)

Program 9-3 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter an item code: F15[Enter]
This dinner fork costs $3.19

Look up another price (Y/N)? n[Enter]
```

Recall from Chapter 7 that when an object is passed to a function as a constant reference, any of the object's member functions that the receiving function will call must also be defined with the key word `const`. This is also the case when an array of objects is passed to a function. In Program 9-3 the `search` function uses a `const` array parameter to receive the array of `Inventory` objects in order to safeguard it from any changes being made to it. Therefore, the `Inventory` class member functions it calls are also declared to be `const`.

Checkpoint

9.1 Describe the difference between the linear search and the binary search.

9.2 On average, with an array of 20,000 elements, how many comparisons will the linear search perform? (Assume the items being search for are consistently found in the array.)

9.3 With an array of 20,000 elements, what is the maximum number of comparisons the binary search will perform?

9.4 If a linear search is performed on an array, and it is known that some items are searched for more frequently than others, how can the contents of the array be reordered to improve the average performance of the search?

9.3 Introduction to Sorting Algorithms

CONCEPT: Sorting algorithms are used to arrange data into some order.

VideoNote
Sorting a Set of Data

Often the data in an array must be sorted in some order. Customer lists, for instance, are commonly sorted in alphabetical order. Student grades might be sorted from highest to lowest. Mailing label records could be sorted by ZIP code. To sort the data in an array, the programmer must use an appropriate *sorting algorithm*. A sorting algorithm is a technique for scanning through an array and rearranging its contents in some specific order. This section will introduce two simple sorting algorithms: the *bubble sort* and the *selection sort*.

The Bubble Sort

The bubble sort is an easy way to arrange data in *ascending* or *descending order*. Sorting data in ascending order means placing the values in order from lowest to highest. Sorting in descending order means placing them in order from highest to lowest. Bubble sort works by comparing each element in the array with its neighbor and swapping them if they are not in the desired order. Let's see how it arranges the following array's elements in ascending order:

7	2	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The bubble sort starts by comparing the first two elements in the array. If element 0 is greater than element 1, they are exchanged. After the exchange, the array appears as

2	7	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

This process is repeated with elements 1 and 2. If element 1 is greater than element 2, they are exchanged. The array now appears as

2	3	7	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Next, elements 2 and 3 are compared. However, in this array, these two elements are already in the proper order (element 2 is less than element 3), so no exchange takes place.

As the cycle continues, elements 3 and 4 are compared. Once again, because they are already in the proper order, no exchange is necessary. When elements 4 and 5 are compared, however, an exchange must take place because element 4 is greater than element 5. The array now appears as

2	3	7	8	1	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

At this point, the entire array has been scanned. This is called the first *pass* of the sort. Notice that the largest value is now correctly placed in the last array element. However, the rest of the array is not yet sorted. So the sort starts over again with elements 0 and 1. Because they are in the proper order, no exchange takes place. Elements 1 and 2 are compared next, but once again, no exchange takes place. This continues until elements 3 and 4 are compared. Because element 3 is greater than element 4, they are exchanged. The array now appears as

2	3	7	1	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Notice that this second pass over the array elements has placed the second largest number in the next to the last array element. This process will continue, with the sort repeatedly passing through the array and placing at least one number in order on each pass, until the array is fully sorted. Ultimately, the array will appear as

1	2	3	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Here is the bubble sort in pseudocode. Notice that it uses a pair of nested loops. The outer loop, a `do-while` loop, iterates once for each pass of the sort. The inner loop, a `for` loop, holds the code that does all the comparisons and needed swaps during a pass. If two elements are exchanged, the `swap` flag variable is set to `true`. The outer loop continues iterating, causing additional passes to be made, until it finds the `swap` flag `false`, meaning that no elements were swapped on the previous pass. This indicates that the array is now fully sorted.

```
Do
   Set swap flag to false
   For count = 0 to the next-to-last array subscript
      If array[count] is greater than array[count + 1]
         Swap the contents of array[count] and array[count + 1]
         Set swap flag to true
      End If
   End For
While the swap flag is true   // A swap ocurred on the previous pass.
```

The following C++ code implements the bubble sort as a function. The parameter `array` references an integer array to be sorted. The parameter `size` contains the number of elements in `array`.

```
void sortArray(int array[], int size)
{
   int temp;
   bool swap;

   do
   {  swap = false;
      for (int count = 0; count < (size - 1); count++)
      {
         if (array[count] > array[count + 1])
         {
            temp = array[count];
            array[count] = array[count + 1];
            array[count + 1] = temp;
            swap = true;
         }
      }
   } while (swap);    // Loop again if a swap occurred on this pass.
}
```

Let's look more closely at the `for` loop that handles the comparisons and exchanges during a pass. Here is its starting line:

```
for (int count = 0; count < (size - 1); count++)
```

The variable `count` holds the array subscripts. It starts at zero and is incremented as long as it is less than `size - 1`. The value of `size` is the number of elements in the array, and `count` stops just short of reaching this value because the following line compares each element with the one after it:

```
if (array[count] > array[count + 1])
```

When `array[count]` is the next-to-last element, it will be compared to the last element. If the `for` loop were allowed to increment `count` past `size - 1`, the last element in the array would be compared to a value outside the array.

Here is the `if` statement in its entirety:

```
if (array[count] > array[count + 1])
{
   temp = array[count];
   array[count] = array[count + 1];
   array[count + 1] = temp;
   swap = true;
}
```

If `array[count]` is greater than `array[count + 1]`, the two elements must be exchanged. First, the contents of `array[count]` is copied into the variable `temp`. Then the contents of `array[count + 1]` is copied into `array[count]`. The exchange is made complete when `temp` (which holds the previous contents of `array[count]`) is copied to `array[count + 1]`. Last, the `swap` flag variable is set to `true`. This indicates that an exchange has been made.

Program 9-4 demonstrates the bubble sort function in a complete program.

Program 9-4

```
1  // This program uses the bubble sort algorithm to sort an array
2  // of integers in ascending order.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void sortArray(int [], int);
8  void showArray(const int [], int);
9
10 int main()
11 {
12    const int SIZE = 6;
13
14    // Array of unsorted values
15    int values[SIZE] = {7, 2, 3, 8, 9, 1};
16
17    // Display the values
18    cout << "The unsorted values are:\n";
19    showArray(values, SIZE);
20
21    // Sort the values
22    sortArray(values, SIZE);
23
24    // Display them again
25    cout << "The sorted values are:\n";
26    showArray(values, SIZE);
27    return 0;
28 }
29
```

(program continues)

Program 9-4 *(continued)*

```
30 /**********************************************************
31  *                        sortArray                       *
32  * This function performs an ascending-order bubble sort on *
33  * array. The parameter size holds the number of elements   *
34  * in the array.                                            *
35  **********************************************************/
36 void sortArray(int array[], int size)
37 {
38    int  temp;
39    bool swap;
40
41    do
42    {   swap = false;
43       for (int count = 0; count < (size - 1); count++)
44       {
45          if (array[count] > array[count + 1])
46          {
47             temp = array[count];
48             array[count] = array[count + 1];
49             array[count + 1] = temp;
50             swap = true;
51          }
52       }
53    } while (swap);    // Loop again if a swap occurred on this pass.
54 }
55
56 /**********************************************************
57  *                        showArray                       *
58  * This function displays the contents of array. The        *
59  * parameter size holds the number of elements in the array. *
60  **********************************************************/
61 void showArray(const int array[], int size)
62 {
63    for (int count = 0; count < size; count++)
64       cout << array[count] << " ";
65    cout << endl;
66 }
```

Program Output

```
The unsorted values are:
7 2 3 8 9 1
The sorted values are:
1 2 3 7 8 9
```

The Selection Sort

The bubble sort is inefficient for large arrays because repeated data swaps are often required to place a single item in its correct position. The selection sort, like the bubble sort, places just one item in its correct position on each pass. However, it usually performs fewer exchanges because it moves items immediately to their correct position in the array.

Like any sort, it can be modified to sort in either ascending or descending order. An ascending sort works like this: The smallest value in the array is located and moved to element 0. Then the next smallest value is located and moved to element 1. This process continues until all of the elements have been placed in their proper order.

Let's see how the selection sort works when arranging the elements of the following array:

5	7	2	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The selection sort scans the array, starting at element 0, and locates the element with the smallest value. The contents of this element are then swapped with the contents of element 0. In this example, the 1 stored in element 5 is the smallest value, so it is swapped with the 5 stored in element 0. This completes the first pass and the array now appears as

1	7	2	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The algorithm then repeats the process, but because element 0 already contains the smallest value in the array, it can be left out of the procedure. For the second pass, the algorithm begins the scan at element 1. It locates the smallest value in the unsorted part of the array, which is the 2 in element 2. Therefore, element 2 is exchanged with element 1. The array now appears as

1	2	7	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Once again the process is repeated, but this time the scan begins at element 2. The algorithm will find that element 5 contains the next smallest value and will exchange this element's contents with that of element 2, causing the array to appear as

1	2	5	8	9	7
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Next, the scanning begins at element 3. Its contents is exchanged with that of element 5, causing the array to appear as

1	2	5	7	9	8
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

At this point there are only two elements left to sort. The algorithm finds that the value in element 5 is smaller than that of element 4, so the two are swapped. This puts the array in its final arrangement:

1	2	5	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Here is the selection sort algorithm in pseudocode:

```
For startScan = 0 to the next-to-last array subscript
   Set index to startScan
   Set minIndex to startScan
   Set minValue to array[startScan]
   For index = (startScan + 1) to the last subscript in the array
      If array[index] is less than minValue
         Set minValue to array[index]
         Set minIndex to index
      End If
      Increment index
   End For
   Set array[minIndex] to array[startScan]
   Set array[startScan] to minValue
End For
```

The following function uses the selection sort to arrange the values in an integer array in ascending order. It accepts two arguments. The first parameter, `array`, receives the array to be sorted and the second, `size`, indicates how many values are stored in the array.

```
void selectionSort(int array[], int size)
{
    int startScan, minIndex, minValue;

    for (startScan = 0; startScan < (size - 1); startScan++)
    {
       minIndex = startScan;
       minValue = array[startScan];

       for (int index = startScan + 1; index < size; index++)
       {
          if (array[index] < minValue)
          {
             minValue = array[index];
             minIndex = index;
          }
       }
       array[minIndex] = array[startScan];
       array[startScan] = minValue;
    }
}
```

As with bubble sort, selection sort uses a pair of nested loops, in this case two `for` loops. The inner loop sequences through the array, starting at `array[startScan + 1]`, searching for the element with the smallest value. When the element is found, its subscript is stored in the variable `minIndex`, and its value is stored in `minValue`. The outer loop then exchanges the contents of this element with `array[startScan]` and increments `startScan`. This procedure repeats until the contents of every element have been moved to their proper location. For N pieces of data this requires N-1 passes.

Program 9-5 demonstrates the selection sort function in a complete program.

Program 9-5

```
// This program uses the selection sort algorithm to sort
// an array in ascending order.
#include <iostream>
using namespace std;

// Function prototypes
void selectionSort(int [], int);
void showArray(const int [], int);

int main()
{
   const int SIZE = 6;

   // Array of unsorted values
   int values[SIZE] = {5, 7, 2, 8, 9, 1};

   // Display the values
   cout << "The unsorted values are\n";
   showArray(values, SIZE);

   // Sort the array
   selectionSort(values, SIZE);

   // Display the values again
   cout << "The sorted values are\n";
   showArray(values, SIZE);
   return 0;
}

/**********************************************************
 *                      selectionSort                     *
 * This function performs an ascending-order selection sort  *
 * on array. The parameter size holds the number of elements  *
 * in the array.                                          *
 **********************************************************/
void selectionSort(int array[], int size)
{
   int startScan, minIndex, minValue;

```

(program continues)

Program 9-5 *(continued)*

```
40    for (startScan = 0; startScan < (size - 1); startScan++)
41    {
42       minIndex = startScan;
43       minValue = array[startScan];
44       for(int index = startScan + 1; index < size; index++)
45       {
46          if (array[index] < minValue)
47          {
48             minValue = array[index];
49             minIndex = index;
50          }
51       }
52       array[minIndex] = array[startScan];
53       array[startScan] = minValue;
54    }
55 }
56
57 /**********************************************************
58  *                        showArray                       *
59  * This function displays the contents of array. The      *
60  * parameter size holds the number of elements in the array. *
61  **********************************************************/
62 void showArray(const int array[], int size)
63 {
64    for (int count = 0; count < size; count++)
65       cout << array[count] << " ";
66    cout << endl;
67 }
```

Program Output

```
The unsorted values are
5 7 2 8 9 1
The sorted values are
1 2 5 7 8 9
```

Checkpoint

9.5 True or false: Any sort can be modified to sort in either ascending or descending order.

9.6 What one line of code would need to be modified in the bubble sort to make it sort in descending, rather than ascending order? How would the revised line be written?

9.7 After one pass of bubble sort, which value is in order?

9.8 After one pass of selection sort, which value is in order?

9.9 Which sort usually requires fewer data values to be swapped, bubble sort or selection sort?