Chapter 9

Sorting and Searching Arrays

A portion of this lab is to be done during the scheduled lab time. The take-home programming assignments are to be completed on your own; see the lab website for due dates. The in-lab portion is worth 20% of the lab credit; the programming assignments are worth the other 80%. See the website for details on how the homework programming assignments will be graded. You are not responsible for user errors in input unless specified in the assignment. Feedback will be provided explaining your grade on each assignment.

It is important that you complete each step of this lab before going on to the next, as the exercises build upon one another. You will find it helpful to diagram the action of each method or function as you go along. If you have difficulty in some step, DO NOT proceed before resolving it; seek help from the lab assistants. You will not be able to fully appreciate the remaining content of the lab and you are likely to compound the problem.

Contents

Introduction

Sorting and searching are two fundamental problems in computer science. This lab introduces an elementary means of sorting information. This lab also examines the linear search and the binary search for a set of parallel arrays.

Topics Covered in this Lab:

- bubble sort
- linear search
- binary search
- analysis for best, worst, and average cases

Questions Answered in this Lab:

- How does the bubble sort put data in order?
- When is a linear search appropriate?
- When is a binary search appropriate?

Demonstrable Skills Acquired in this Lab:

- ability to use files for input and output
- ability to manage input and output streams
- comprehension of basic sequential file access
- understanding of how to efficiently bubble sort data
- ability to implement a linear search
- ability to implement a binary search
- understanding of circumstances appropriate to each type of search

Sorting with the Bubble Sort

Sorting is one of the most fundamental problems of Computer Science. The *bubble sort* uses an elementary algorithm to accomplish this task. The basic algorithm will be presented and then refinements will be added. The bubble sort to be developed will put a set of exam scores in descending order.

Obtain the *sp09iL.cpp* and *dataInLab.txt* resource files from the server.

Add the following prototype for the first version of the algorithm to the program.

Code Illustration

```
int descendingBubbleSortV1 (int finalExams[], int numberOfExams);
```

Add the following to function `main` to invoke the bubble sort algorithm; the number of comparisons performed is returned by the function and displayed for inspection.

Code Illustration

```
// demonstrate the basic bubble sort
   cout << "number of compares: "
        << descendingBubbleSortV1(finalExams, numberOfExams) << endl;
   write ("outputInLab.txt", ios::out|ios::app, "post V1 sort", finalExams,
          numberOfExams);
```

The essence of the sorting algorithm is that it looks at successive *pairs* of values, putting them in order if they are not already so. If there are five values to be sorted, there will be four successive pairs to examine (elements 0 and 1, elements 1 and 2, elements 2 and 3, and elements 3 and 4); these pairs will be numbered from 0 to 3. If there are ten values to be sorted, there will be nine successive pairs, and these will be numbered from 0 to 8. In general, for `numberOfExams`, there are `numberOfExams`-1 pairs and these are counted from 0 to `numberOfExams`-2. The following pseudocode illustrates this part of the algorithm.

Pseudocode

```
for // pair = 0 to numberOfExams-2
      if (finalExams[pair] < finalExams[pair+1])
         // swap finalExams[pair] and finalExams[pair+1]
```

Once all successive pairs have been examined, the smallest value must have "sunk" to the last position.

FOR IN-LAB CREDIT: Explain why to a lab proctor.

A second pass over the values with this loop would then "sink" the second smallest value into position. If there are five values to be sorted, four passes would sink all but the largest element into position; the largest element floats (or *bubbles*) up into the correct position by process of elimination. The four sink passes required for five values would be numbered from 0 to 3. If there are ten values to be sorted, nine sink passes will be required, and these will be numbered from 0 to 8. For `numberOfExams`, there are `numberOfExams`-1 sink passes required and these are counted from 0 to `numberOfExams`-2. Note that the numbering for this outer "sink" loop is the same as for the previous inner "pair" loop, but the reasons are very different. The following pseudocode adds the outer loop to the algorithm.

Pseudocode

```
for // sink = 0 to numberOfExams-2
      for // pair = 0 to numberOfExams-2
         if (finalExams[pair] < finalExams[pair+1])
           // swap finalExams[pair] and finalExams[pair+1])
```

In order to compare this version of the algorithm with subsequent revisions, it is helpful to keep track of how many comparisons are performed. A counter can be added to the algorithm for this purpose.

Pseudocode

```
int compares = 0;
   for // sink = 0 to numberOfExams-2
      for // pair = 0 to numberOfExams-2
      {
         compares++;
         if (finalExams[pair] < finalExams[pair+1])
           // swap finalExams[pair] and finalExams[pair+1])
      }  // end inner for (pair)
   return compares;
```

Consider now the swap of two elements within an array. The logic for this is that since both variables are filled with information that must not be lost, a third storage location is required to temporarily save one of the values:

Code Illustration

```
int temp           = finalExams[pair];
   finalExams[pair]   = finalExams[pair+1];
   finalExams[pair+1] = temp;
```

Put the pieces of this first version of the bubble sort algorithm together as `descendingBubbleSortV1`. Execute the program and inspect *outputInLab.txt* to verify that the sort worked correctly.

Eliminating Redundant Comparisons

Since the first pass through the values sinks the smallest value to the last position, there is no need to examine this value on the next pass. In fact, since each pass sinks one value into its final position, the number of values to look at on each subsequent pass is reduced by one. For example, ten values would require nine pairs be looked at on the first sink pass, then eight on the second sink pass, seven on the third, and so on. Since sink passes are numbered from zero, the number of pairs can simply be reduced by the number of the current sink pass.

Pseudocode

```
for // pair = 0 to numberOfExams-2-sink
```

Create a second version of the algorithm `descendingBubbleSortV2` and implement this change. Add a prototype for this second version of the loop algorithm and add the following to function `main`.

Code Illustration

```
// demonstrate bubble sort with redundant comparisons eliminated
   numberOfExams = read("dataInLab.txt", finalExams, MAX_SCORES);
   cout << "number of compares: "
        << descendingBubbleSortV2(finalExams, numberOfExams) << endl;
   write ("outputInLab.txt", ios::out|ios::app, "post V2 sort", finalExams,
          numberOfExams);
```

Note that the exam scores are read from the file into the array again so that this version of the sort starts under the same conditions. Execute the program and verify that the number of comparisons has indeed dropped.

FOR IN-LAB CREDIT: Demonstrate this drop in comparisons for a lab proctor.

Early Termination

How would the bubble sort discussed thus far deal with a list of numbers which were nearly or even completely in sorted order? It would require the same number of comparisons even in these cases. A smarter sort would be able to conclude that no further work need be done if a pass required no swaps. The outer count-controlled loop is not appropriate to this end; in its place, an event-controlled loop is used to check for the occurrence of any swap during a pass. Further, this new loop should be of the post-test variety; the values must always be examined once, even if they are completely in order. Revise the bubble sort in a third version of the algorithm.

Pseudocode

```
int sink     = 0;
   int compares = 0;

   bool swapOccurred;

   do
   {
      swapOccurred = false;
      for // pair = 0 to numberOfExams-2-sink
      {
         compares++;

         if (finalExams[pair] < finalExams[pair+1])
         {
            // swap finalExams[pair] and finalExams[pair+1])
            swapOccurred = true;
         }  // end if
      } // end for

      sink++;
   }  // end do-while
   while (swapOccurred);

   return compares;
```

Add a prototype for this third version of the loop algorithm and add the following to function `main`.

Code Illustration

```
// demonstrate bubble sort with early termination
   numberOfExams = read("dataInLab.txt", finalExams, MAX_SCORES);
   cout << "number of compares: "
        << descendingBubbleSortV3 (finalExams, numberOfExams) << endl;
   write ("outputInLab.txt", ios::out|ios::app, "post V3 sort", finalExams,
          numberOfExams);
```

Execute the program and verify that the number of comparisons has dropped further still.

<u>FOR IN-LAB CREDIT</u>: Demonstrate your program for a lab proctor.

The Linear Search

As an example with which to visualize a *search*, imagine a telephone book of perhaps thousands of pages. Like any ordinary book, the organization is alphabetical by name. The typical use of a telephone book retrieves a number for a particular name; in finding a name, one flips rapidly through the book until names alphabetically close to the one in question are encountered. A *linear search* of the list of numbers does not employ intelligence of this nature; it simply starts on the first page every time information is to be looked up. A linear search can be performed for either a name or a number. The difference between these searches is apparent when a request is made for information that is not in the book. In the case of a non-existent name, the search can stop early when an alphabetically-subsequent name is encountered; the search for a non-existent number must continue to the end of the book because the number could be anywhere.

Linear Search of Unordered Data

To demonstrate an unordered linear search, consider the final exams scores found in *dataInLab.txt*. Add the following prototype for a linear search function to the program.

Code Illustration

```
int unorderedLinearSearch (int searchValue, const int finalExams[], int numberOfExams);
```

Add the following to function `main` to invoke such a search.

Code Illustration

```
// demonstrate a linear search of unordered data
   numberOfExams = read("dataInLab.txt", finalExams, MAX_SCORES);
   int searchScore;
   cout << "Enter score for searching: ";
   cin >> searchScore;

   int location = unorderedLinearSearch (searchScore, finalExams, numberOfExams);
   cout << "unordered linear search results for " << searchScore << endl;
   if (location >= 0)
   {
      cout << "score " << searchScore << " found in array location ["
           << location << "]\n";
   }  // end if
   else // location < 0
      cout << "score " << searchScore << " not found by linear search\n";
```

The search itself will return the location where the desired score is found. Comparisons are made with each score, starting with the first, until a match is found. Complete function `unorderedLinearSearch` now.

Pseudocode

```
// initialize location to -1 (as -1 will indicate not found)
   for // index = 0 to numberOfExams-1
      if // searchValue is finalExams[index]
         location = index;
   return location;
```

Execute the program and verify that the search works correctly for scores in the data and not in the data. Close inspection of the data shows that an exam score may be seen more than once as evidenced by the exam score `80`. Execute the program and search for this score; note its location reported by the program.

This search can be improved by providing a way to exit the loop once the score has been located. Modify the `unorderedLinearSearch` function as follows.

Pseudocode

```
// initialize location to -1 (as -1 will indicate not found)
   bool found = false;
   for // index = 0 to numberOfExams-1 and not found
      if // searchValue is finalExams[index]
      {
         location = index;
         found = true;
      }
   return location;
```
Execute the program and verify that the search works correctly for scores in the data and not in the data. Repeat the search for `80` and note its location reported by the program.

FOR IN-LAB CREDIT: Demonstrate one successful search and one unsuccessful search for a lab proctor. Then,explain why the program reports a different location for the exam score `80`.

The Binary Search

Some intelligence can be applied to the search to make it more human-like in its execution. When The binary search is so named because it cuts the list in two on each comparison. Add the following prototype.

Code Illustration

```
int binarySearch (int searchValue, const int finalExams[], int numberOfExams);
```
Add the following to function `main` to invoke a binary search; note that this sequence is identical in form to that used to invoke the linear search.

Code Illustration

```
// demonstrate the use of a binary search
   // note:  array must be in sorted order
   descendingBubbleSortV3 (finalExams, numberOfExams);
   location = binarySearch (searchScore, finalExams, numberOfExams);
   cout << "binary search results for " << searchScore << endl;
   if (location >= 0)
      cout << "score " << searchScore << " found in array location ["
           << location << "]\n";
   else // location < 0
      cout << "score " << searchScore << " not found by binary search\n";
```
The binary search assumes *left* and *right* bounds at the limits of the array, 0 and `numberOfExams-1`, respectively. The search proceeds by examining the score at the middle of the current range and subsequently shrinking the range to one side or the other of the middle, assuming the score was not found at the middle. For example, consider a list of 100 integers, the first of which is indexed by 0 and the last by 99. The search will begin at the middle, (0+99)/2 = 49. If the integer to be found comes before that at position 49, the search should continue with the integers between 0 and 48; if the integer to be found comes after that at 49, the search should continue between 50 and 99. If neither of these conditions is true, the integer must match that at position 49 and so the search has found its target.

Under what circumstances can it be concluded that the score is not in the list? Each time the score is not found, either `left` or `right` is modified. If the score is not present, these modifications will eventually result in inconsistent values for `left` and `right`. Given their definitions, inconsistency means `left` becomes greater than `right`. Experiment with an example to illustrate what happens when a score to be found is not in the list.

FOR IN-LAB CREDIT: Explain to a lab proctor the steps in this example which lead to an inconsistency.

Use a test for this inconsistency to determine when the score might still be in the list and complete the body of function `binarySearch`.

Pseudocode

```
int mid;
   int left = 0;
   int right = numberOfExams - 1;
   int location = -1; //assume won't be found
   bool found = false;

   while // not found and score might still be in list
   {
      mid = // center between left & right
      if // searchValue might be found to the left of finalExams[mid]
         right = mid - 1;
      else if // searchValue might be found to the right finalExams[mid]
         left = mid + 1;
      else // searchValue is finalExams[mid]
         found = true;
   }  // end while

   if //found score
   {
       location = mid;
   }
   return location;
```
Execute the program and verify that the search works correctly for score both in the data and not in the data.

FOR IN-LAB CREDIT: Demonstrate one successful search and one unsuccessful search for a lab proctor.

<u>FOR IN-LAB CREDIT</u>: Upload the solution for each of the above lab steps in a file named **`sp09iL.cpp`**.