Chapter 8 Searching and Sorting Arrays

8.1 Focus on Software Engineering: Introduction to Search Algorithms

Concept:

- A search algorithm is a method for finding a specific item within a larger collection of data.
- This section covers two algorithms for searching array contents.

- It is a common task for programs to search arrays for specific items.
- This section introduces two search methods: the linear search and the binary search, each with distinct advantages and disadvantages.

The Linear Search

- The **linear search**, also known as the **sequential search**, is a straightforward algorithm.
- It uses a loop to step through an array sequentially from the first element.
- It compares each element with the search value and stops when the value is found or the end of the array is reached.
- If the value is not in the array, the algorithm will search to the end without success.

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

- The `linearSearch` function below is a C++ implementation for searching an integer array.
- It searches the array `arr` of size `size` for the given `value`.
- If the value is found, its array subscript is returned; otherwise, -1 is returned.

```cpp
int linearSearch (const int arr[], int size, int value)
{
  int index = 0;
  int position = -1;
  bool found = false;
  while (index < size && !found)
  {
     if (arr[index] == value)
     {
        found = true;
        position = index;
     }
     index++;
  }
  return position;
}
```

Note:

The reason −1 is returned when the search value is not found in the array is because −1 is not a valid subscript.

- Program 8-1 is a complete program that uses the `linearSearch` function to find a score of 100 in a five-element array named `tests`.

Program 8-1

```cpp
#include <iostream>
using namespace std;

int linearSearch(const int[], int, int);

int main()
{
   const int SIZE = 5;
   int tests[SIZE] = { 87, 75, 98, 100, 82 };
```

```cpp
    int results;

    results = linearSearch(tests, SIZE, 100);

    if (results == -1)
       cout << "You did not earn 100 points on any test\n";
    else
    {
       cout << "You earned 100 points on test ";
       cout << (results + 1) << endl;
    }
    return 0;
}

int linearSearch(const int arr[], int size, int value)
{
    int index = 0;
    int position = -1;
    bool found = false;

    while (index < size && !found)
    {
       if (arr[index] == value)
       {
          found = true;
          position = index;
       }
       index++;
    }
    return position;
}
```

Program Output

Inefficiency of the Linear Search

- **Advantage**: The linear search is simple to understand and implement, and it does not require the data to be sorted.
- **Disadvantage**: It is inefficient. For a 20,000-element array, finding the last item requires looking at all 20,000 elements.

- In an average case, for an array of N items, the linear search will find an item in N/2 attempts.
- For an array of 50,000 elements, this means an average of 25,000 comparisons.
- The maximum number of comparisons is always N.
- When a search fails, the linear search must compare every element in the array.
- The linear search is not recommended for large arrays if speed is a priority.

The Binary Search

- The **binary search** is a much more efficient algorithm than the linear search.
- Its only requirement is that the array values must be sorted.
- It begins by checking the middle element of the array.
- If the middle element is not the desired value, the algorithm determines if the value is in the first or second half of the array.
- In either case, half of the array's elements are eliminated from the search.

The Binary Search

- If the value is not found in the middle, the search procedure is repeated on the half of the array that could contain the value.
- This process of halving the search area continues until the value is found or there are no elements left to test.

Here is the pseudocode for a function that performs a binary search on an array:

```
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 array[first]
   and array[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`.
- `first` and `last` mark the boundaries of the portion of the array being searched.
- `middle` is the calculated subscript halfway between `first` and `last`.
- If the middle element is not the search value, `first` or `last` is adjusted to narrow the search to one half of the current portion.

The function `binarySearch` shown in the following example is used to perform a binary search on an integer array. The first parameter, `array`, which has a maximum of `numElems` 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.

```cpp
int binarySearch(const int array[], int numElems, int value)
{
   int first = 0,
       last = numElems - 1,
       middle,
       position = -1;
   bool found = false;
   while (!found && first <= last)
   {
       middle = (first + last) / 2;
       if (array[middle] == value)
       {
          found = true;
          position = middle;
       }
       else if (array[middle] > value)
          last = middle - 1;
       else
           first = middle + 1;
   }
```

```
    return position;
}
```

- Program 8-2 is a complete program using the `binarySearch` function to search for an employee ID number.

Program 8-2

```
#include <iostream>
using namespace std;

int binarySearch(const int [], int, int);
const int SIZE = 20;

int main()
{
    int idNums[SIZE] = {101, 142, 147, 189, 199, 207, 222,
                        234, 289, 296, 310, 319, 388, 394,
                        417, 429, 447, 521, 536, 600};
    int results;
    int empID;

    cout << "Enter the employee ID you wish to search for: ";
    cin >> empID;

    results = binarySearch(idNums, SIZE, empID);

    if (results == -1)
       cout << "That number does not exist in the array. \n";
    else
    {
          cout << "That ID is found at element " << results;
          cout << " in the array.\n";
    }
    return 0;
}


int binarySearch(const int array[], int size, int value)
{
       int first = 0,
          last = size - 1,
          middle,
          position = -1;
```

```
        bool found = false;

    while (!found && first <= last)
    {
        middle = (first + last) / 2;
        if (array[middle] == value)
        {
            found = true;
            position = middle;
        }
        else if (array[middle] > value)
            last = middle - 1;
        else
            first = middle + 1;
    }
    return position;
}
```

Program Output

Warning!

Notice the array in Program 8-2 is initialized with its values already sorted in ascending order. The binary search algorithm will not work properly unless the values in the array are sorted.

The Efficiency of the Binary Search

- The binary search is much more efficient than the linear search because it eliminates half of the remaining search area with each comparison.
- For an array with 1,000 elements, a binary search takes no more than 10 comparisons, while a linear search would average 500 comparisons.
- The maximum number of comparisons for a binary search can be determined using powers of 2.
- Find the smallest power of 2 that is greater than or equal to the number of elements in the array.

- For an array of 50,000 elements, a maximum of 16 comparisons will be made ($2^{16} = 65,536$).
- For an array of 1,000,000 elements, a maximum of 20 comparisons will be made ($2^{20} = 1,048,576$).

8.2 Focus on Problem Solving and Program Design: A Case Study

- This case study involves creating a program for the Demetris Leadership Center (DLC, Inc.) to look up product prices.
- The program will prompt a user to enter a product number and then display the product's title, description, and price from Table 8-1.

Table 8-1 Products

PRODUCT TITLE	PRODUCT DESCRIPTION	PRODUCT NUMBER	UNIT PRICE
Six Steps to Leadership	Book	914	$12.95
Six Steps to Leadership	Audio CD	915	$14.95
The Road to Excellence	DVD	916	$18.95
Seven Lessons of Quality	Book	917	$16.95
Seven Lessons of Quality	Audio CD	918	$21.95
Seven Lessons of Quality	DVD	919	$31.95
Teams Are Made, Not Born	Book	920	$14.95
Leadership for the Future	Book	921	$14.95
Leadership for the Future	Audio CD	922	$16.95

Variables

Table 8-2 lists the variables needed:

Table 8-2 Variables

VARIABLE	DESCRIPTION
`NUM_PRODS`	A constant integer initialized with the number of products the Demetris Leadership Center sells. This value will be used in the definition of the program's array.
`MIN_PRODNUM`	A constant integer initialized with the lowest product number.
`MAX_PRODNUM`	A constant integer initialized with the highest product number.
`id`	Array of integers. Holds each product's number.
`title`	Array of strings, initialized with the titles of products.
`description`	Array of strings, initialized with the descriptions of each product.
`prices`	Array of `double`s. Holds each product's price.

Modules

The program will consist of the functions listed in Table 8-3.

Table 8-3 Functions

FUNCTION	DESCRIPTION
`main`	The program's `main` function. It calls the program's other functions.
`getProdNum`	Prompts the user to enter a product number. The function validates input and rejects any value outside the range of correct product numbers.

FUNCTION	DESCRIPTION
`binarySearch`	A standard binary search routine. Searches an array for a specified value. If the value is found, its subscript is returned. If the value is not found, –1 is returned.
`displayProd`	Uses a common subscript into the `title`, `description`, and `prices` arrays to display the title, description, and price of a product.

Function `main`

- The `main` function defines variables and calls the other functions in the program. Here is its pseudocode:

```
do
   Call getProdNum
   Call binarySearch
   If binarySearch returned -1
      Inform the user that the product number was not found
   else
      Call displayProd
   End If
   Ask the user if the program should repeat
While the user wants to repeat the program
```

- The C++ code is shown below.
- The global constant `NUM_PRODS` is set to 9.
- The `id`, `title`, `description`, and `prices` arrays are initialized with data.

```
do
{
   prodNum = getProdNum();
   index = binarySearch(id, NUM_PRODS, prodNum);
   if (index == -1)
      cout << "That product number was not found.\n";
   else
      displayProd(title, description, prices, index);
   cout << "Would you like to look up another product? (y/n) ";
   cin >> again;
} while (again == 'y' || again == 'Y');
```

The `getProdNum` Function

- The `getProdNum` function prompts the user for a product number.
- It validates the input to ensure it is within the valid range (914–922).
- If the input is invalid, it re-prompts the user until a valid number is entered, which is then returned.

```
Display a prompt to enter a product number
Read prodNum
While prodNum is invalid
   Display an error message
   Read prodNum
End While
Return prodNum
```

Here is the actual C++ code:

```cpp
int getProdNum()
{
   int prodNum;
   cout << "Enter the item's product number: ";
   cin >> prodNum;
   while (prodNum < MIN_PRODNUM || prodNum > MAX_PRODNUM)
   {
      cout << "Enter a number in the range of " << MIN_PRODNUM;
      cout <<" through " << MAX_PRODNUM << ".\n";
      cin >> prodNum;
   }
   return prodNum;
}
```

The `binarySearch` Function

- This function is identical to the `binarySearch` function discussed earlier in this chapter.

The `displayProd` Function

- The `displayProd` function accepts the `title`, `description`, and `price` arrays, along with a subscript value `index`, as arguments.
- It displays the product information stored at the given subscript in each array.

```
void displayProd(const string title[], const string desc[],
                 const double price[], int index)
{
   cout << "Title: " << title[index] << endl;
   cout << "Description: " << desc[index] << endl;
   cout << "Price: $" << price[index] << endl;
}
```

The Entire Program

- Program 8-3 shows the entire source code for this program.

Program 8-3

```
#include <iostream>
#include <string>
using namespace std;

const int NUM_PRODS = 9;
const int MIN_PRODNUM = 914;
const int MAX_PRODNUM = 922;

int getProdNum();
int binarySearch(const int [], int, int);
void displayProd(const string [], const string [], const double [], int);

int main()
{
    int id[NUM_PRODS] = {914, 915, 916, 917, 918, 919, 920,
                         921, 922};

    string title[NUM_PRODS] =
        { "Six Steps to Leadership",
          "Six Steps to Leadership",
          "The Road to Excellence",
          "Seven Lessons of Quality",
          "Seven Lessons of Quality",
          "Seven Lessons of Quality",
```

```cpp
              "Teams Are Made, Not Born",
              "Leadership for the Future",
              "Leadership for the Future"
            };

    string description[NUM_PRODS] =
        { "Book", "Audio CD", "DVD",
          "Book", "Audio CD", "DVD",
          "Book", "Book", "Audio CD"
        };

    double prices[NUM_PRODS] = {12.95, 14.95, 18.95, 16.95, 21.95,
                                31.95, 14.95, 14.95, 16.95};

    int prodNum;
    int index;
    char again;

    do
    {
        prodNum = getProdNum();

        index = binarySearch(id, NUM_PRODS, prodNum);

        if (index == -1)
            cout << "That product number was not found.\n";
        else
            displayProd(title, description, prices, index);

        cout << "Would you like to look up another product? (y/n) ";
        cin >> again;
    } while (again == 'y' || again == 'Y');
    return 0;
}


int getProdNum()
{
    int prodNum;

    cout << "Enter the item's product number: ";
    cin >> prodNum;
    while (prodNum < MIN_PRODNUM || prodNum > MAX_PRODNUM)
    {
        cout << "Enter a number in the range of " << MIN_PRODNUM;
        cout <<" through " << MAX_PRODNUM << ".\n";
        cin >> prodNum;
    }
    return prodNum;
}


int binarySearch(const int array[], int numElems, int value)
{
    int first = 0,
        last = numElems - 1,
        middle,
        position = -1;
    bool found = false;

    while (!found && first <= last)
    {
        middle = (first + last) / 2;
        if (array[middle] == value)
        {
            found = true;
            position = middle;
        }
        else if (array[middle] > value)
            last = middle - 1;
        else
            first = middle + 1;
    }
    return position;
}


void displayProd(const string title[], const string desc[],
                const double price[], int index)
{
    cout << "Title: " << title[index] << endl;
    cout << "Description: " << desc[index] << endl;
    cout << "Price: $" << price[index] << endl;
}
```

Program Output

Checkpoint

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

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

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

8.4 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?

8.3 Focus on Software Engineering: Introduction to Sorting Algorithms

Concept:

- Sorting algorithms are used to arrange data into some order.

Sorting Algorithms

- Many programming tasks, such as creating alphabetical customer lists or ordering student grades, require data in an array to be sorted.
- A **sorting algorithm** is a technique for stepping through an array and rearranging its contents into a specific order.
- Data can be sorted in **ascending order** (lowest to highest) or **descending order** (highest to lowest).

- This section introduces two sorting algorithms: the **bubble sort** and the **selection sort**.

The Bubble Sort

- The **bubble sort** is a simple algorithm for arranging data.
- It is named “bubble sort” because values “bubble” toward their correct position with each pass through the array.
- In an ascending sort, larger values move toward the end of the array.
- In a descending sort, smaller values move toward the end.
- This section will demonstrate how to sort an array in ascending order.
- Let’s consider arranging the elements of the array in Figure 8-1 in ascending order.
- The bubble sort starts by comparing the first two elements. If element 0 is greater than element 1, they are swapped.
- This process of comparing adjacent elements and swapping them if they are out of order is repeated for the entire array.
- After the first full pass through the array, the largest value will have “bubbled” to the last position.
- The algorithm then makes another pass, but it can ignore the last element, which is already in place.
- In the second pass, the second-largest value will move to the second-to-last position.
- This continues for subsequent passes, with the sorted portion of the array growing from the end.
- After all passes are complete, the array will be fully sorted.

Here is the bubble sort algorithm in pseudocode:

```
For maxElement = each subscript in the array, from the last to the first
        For index = 0 To maxElement - 1
                If array[index] > array[index + 1]
                        swap array[index] with array[index + 1]
                End If
        End For
End For
```

- The following C++ function implements the bubble sort algorithm. It accepts an array and its size as arguments.

```cpp
void bubbleSort(int array[], int size)
{
   int maxElement;
   int index;

   for (maxElement = size - 1; maxElement > 0; maxElement--)
   {
      for (index = 0; index < maxElement; index++)
      {
         if (array[index] > array[index + 1])
         {
            swap(array[index], array[index + 1]);
         }
      }
   }
}
```

- The function uses local variables `maxElement` to track the last unsorted element's subscript and `index` for the inner loop.
- An outer `for` loop controls the number of passes.
- A nested inner `for` loop iterates through the unsorted portion, comparing `array[index]` with `array[index + 1]`.
- If the elements are out of order, the `swap` function is called to exchange them.

Swapping Array Elements

- Sorting algorithms often require swapping the values of two variables in memory.

Assume we have the following variable declarations:

```cpp
int a = 1;
int b = 9;
```

- A common error is to attempt a swap with direct assignment, like `a = b; b = a;`.

- This doesn't work because the original value of `a` is lost when `b` is assigned to it.
- To swap correctly, a third temporary variable is needed.
- The process is:
    1. Assign the value of `a` to `temp`.
    2. Assign the value of `b` to `a`.
    3. Assign the value of `temp` to `b`.
- The following `swap` function encapsulates this logic.

```
void swap(int &a, int &b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

Note:

It is critical that we use reference parameters in the `swap` function, because the function must be able to change the values of the items that are passed to it as arguments.

- Program 8-4 demonstrates the `bubbleSort` function in a complete program.

Program 8-4

```
#include <iostream>
using namespace std;

void bubbleSort(int[], int);
void swap(int &, int &);

int main()
{
   const int SIZE = 6;

   int values[SIZE] = { 6, 1, 5, 2, 4, 3 };

   cout << "The unsorted values:\n";
   for (auto element : values)
     cout << element << " ";
```

```cpp
   cout << endl;

   bubbleSort(values, SIZE);

   cout << "The sorted values:\n";
   for (auto element : values)
      cout << element << " ";
   cout << endl;

   return 0;
}

void bubbleSort(int array[], int size)
{
   int maxElement;
   int index;

   for (maxElement = size - 1; maxElement > 0; maxElement--)
   {
      for (index = 0; index < maxElement; index++)
      {
         if (array[index] > array[index + 1])
         {
            swap(array[index], array[index + 1]);
         }
      }
   }
}

void swap(int &a, int &b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

Program Output

The Selection Sort Algorithm

- The **selection sort** is generally more efficient than the bubble sort as it performs fewer swaps.
- It works by repeatedly finding the minimum element from the unsorted part of the array and moving it to the beginning of the sorted part.
- The process is:
    1. Find the smallest value in the array and swap it with the element at index 0.
    2. Find the next smallest value (from index 1 to the end) and swap it with the element at index 1.
    3. Continue this until the entire array is sorted.
- Let's trace this process with the array in Figure 8-8.

The Selection Sort

- In the first pass, the algorithm finds the smallest value (1) and swaps it with the element at index 0.
- In the second pass, it scans from element 1, finds the next smallest value (2), and swaps it with the element at index 1.
- This process continues, with each pass extending the sorted portion of the array by one element.
- The scan area for the next minimum value shrinks with each pass.
- After N-1 passes, the array is fully sorted.

Here is the selection sort algorithm in pseudocode:

```
For start = each array subscript, from the first to the next-to-last
        minIndex = start
        minValue = array[start]
        For index = start + 1 To size - 1
                If array[index] < minValue
                        minValue = array[index]
                        minIndex = index
                End If
        End For
        swap array[minIndex] with array[start]
End For
```

- The following C++ function implements the selection sort, taking an integer array and its size to sort it in ascending order.

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

  for (int start = 0; start < (size - 1); start++)
  {
     minIndex = start;
     minValue = array[start];
     for (int index = start + 1; index < size; index++)
     {
        if (array[index] < minValue)
        {
           minValue = array[index];
           minIndex = index;
        }
     }
     swap(array[minIndex], array[start]);
  }
}
```

- The function uses nested `for` loops.
- The outer loop iterates through the array to place each element correctly.
- The inner loop scans the unsorted portion of the array to find the element with the minimum value.
- After the inner loop completes, the minimum value found is swapped with the element at the beginning of the unsorted portion.
- Program 8-5 demonstrates the `selectionSort` function in a complete program.

Program 8-5

```cpp
#include <iostream>
using namespace std;

void selectionSort(int[], int);
void swap(int &, int &);

int main()
{
   const int SIZE = 6;

   int values[SIZE] = { 6, 1, 5, 2, 4, 3 };

   cout << "The unsorted values:\n";
   for (auto element : values)
      cout << element << " ";
   cout << endl;

   selectionSort(values, SIZE);

   cout << "The sorted values:\n";
   for (auto element : values)
      cout << element << " ";
   cout << endl;

   return 0;
}

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

   for (int start = 0; start < (size − 1); start++)
   {
      minIndex = start;
      minValue = array[start];
      for (int index = start + 1; index < size; index++)
      {
         if (array[index] < minValue)
         {
            minValue = array[index];
            minIndex = index;
         }
      }
      swap(array[minIndex], array[start]);
   }
}

void swap(int &a, int &b)
```

```
    {
        int temp = a;
        a = b;
        b = temp;
    }
```

Program Output

8.4 Focus on Problem Solving and Program Design: A Case Study

- This case study also pertains to the Demetris Leadership Center and involves creating a sales-reporting program.
- Using the units sold data from Table 8-4, the program must display:
    - A list of products sorted by sales dollars, from highest to lowest.
    - The total number of all units sold.
    - The total sales for the period.

Table 8-4 Units Sold

PRODUCT NUMBER	UNITS SOLD
914	842
915	416
916	127
917	514
918	437
919	269
920	97

PRODUCT NUMBER	UNITS SOLD
921	492
922	212

Variables

Table 8-5 Variables

VARIABLE	DESCRIPTION
`NUM_PRODS`	A constant integer initialized with the number of products that DLC, Inc., sells. This value will be used in the definition of the program's array.
`prodNum`	Array of `int`s. Holds each product's number.
`units`	Array of `int`s. Holds each product's number of units sold.
`prices`	Array of `double`s. Holds each product's price.
`sales`	Array of `double`s. Holds the computed sales amounts (in dollars) of each product.

- The arrays `prodNum`, `units`, `prices`, and `sales` are parallel arrays, meaning elements at the same index across these arrays correspond to the same product.

Modules

The program will consist of the functions listed in Table 8-6.

Table 8-6 Functions

FUNCTION	DESCRIPTION
`main`	The program's `main` function. It calls the program's other functions.

FUNCTION	DESCRIPTION
`calcSales`	Calculates each product's sales.
`dualSort`	Sorts the `sales` array so the elements are ordered from highest to lowest. The `prodNum` array is ordered so the product numbers correspond with the correct sales figures in the sorted `sales` array.
`swap`	Swaps the values of two `doubles` that are passed by reference (overloaded).
`swap`	Swaps the values of two `ints` that are passed by reference (overloaded).
`showOrder`	Displays a list of the product numbers and sales amounts from the sorted `sales` and `prodNum` arrays.
`showTotals`	Displays the total number of units sold and the total sales amount for the period.

Function `main`

- The `main` function simply defines variables and calls the other program modules.

```
Call calcSales
Call dualSort
Set display mode to fixed point with two decimal places of precision
Call showOrder
Call showTotals
```

Here is its actual C++ code:

```
calcSales(units, prices, sales, NUM_PRODS);
dualSort(id, sales, NUM_PRODS);
cout << setprecision(2) << fixed << showpoint;
showOrder(sales, id, NUM_PRODS);
showTotals(sales, units, NUM_PRODS);
```

The `calcSales` Function

- The `calcSales` function computes the sales for each product by multiplying its units sold by its price and stores the result in the corresponding element of the `sales` array.

```
For index = each array subscript from 0 through the last subscript
        sales[index] = units[index] * prices[index]
End For
```

And here is the function's actual C++ code:

```cpp
void calcSales(const int units[], const double prices[],
              double sales[], int num)
{
   for (int index = 0; index < num; index++)
     sales[index] = units[index] * prices[index];
}
```

The `dualSort` Function

- The `dualSort` function is a modified selection sort that sorts the `sales` array in descending order.
- When it swaps two elements in the `sales` array, it performs the same swap on the corresponding elements in the `id` array to maintain their parallel relationship.

```
For start = each array subscript, from the first to the next-to-last
        index = start
        maxIndex = start
        tempId = id[start]
        maxValue = sales[start]
        For index = start + 1 To size - 1
                If sales[index] > maxValue
                        maxValue = sales[index]
                        tempId = id[index]
                        maxIndex = index
                End If
        End For
        swap sales[maxIndex] with sales[start]
        swap id[maxIndex] with id[start]
End For
```

Here is the actual C++ code for the `dualSort` function:

```
void dualSort(int id[], double sales[], int size)
{
   int start, maxIndex, tempid;
   double maxValue;
   for (start = 0; start < (size - 1); start++)
   {
      maxIndex = start;
      maxValue = sales[start];
      tempid = id[start];
      for (int index = start + 1; index < size; index++)
      {
         if (sales[index] > maxValue)
         {
            maxValue = sales[index];
            tempid = id[index];
            maxIndex = index;
         }
      }
      swap(sales[maxIndex], sales[start]);
      swap(id[maxIndex], id[start]);
   }
}
```

Note:

Once the `dualSort` function is called, the `id` and `sales` arrays are no longer synchronized with the `units` and `prices` arrays. Because this program doesn't use `units` and `prices` together with `id` and `sales` after this point, it will not be noticed in the final output. However, it is never a good programming practice to sort parallel arrays in such a way that they are out of synchronization. It will be left as an exercise for you to modify the program so all the arrays are synchronized and used in the final output of the program.

The Overloaded `swap` Functions

- This program requires two overloaded versions of the `swap` function.
- One version swaps `double` values (for the `sales` array).
- Another version swaps `int` values (for the `id` array).

```
void swap(double &a, double &b)
{
   double temp = a;
   a = b;
   b = temp;
}
void swap(int &a, int &b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

The `showOrder` Function

- The `showOrder` function displays a formatted header and then lists the sorted product numbers and their sales amounts.

```
Display heading
For index = each subscript of the arrays from 0 through the last subscript
        Display id[index]
        Display sales[index]
End For
```

Here is the function's actual C++ code:

```
void showOrder(const double sales[], const int id[], int num)
{
    cout << "Product Number\tSales\n";
    cout << "----------------------------------\n";
    for (int index = 0; index < num; index++)
    {
        cout << id[index] << "\t\t$";
        cout << setw(8) << sales[index] << endl;
    }
    cout << endl;
}
```

The `showTotals` Function

- The `showTotals` function calculates and displays the total number of units sold and the total sales for the period.

```
totalUnits = 0
totalSales = 0.0
For index = each array subscript from 0 through the last subscript
        Add units[index] to totalUnits[index]
        Add sales[index] to totalSales
End For
Display totalUnits with appropriate heading
Display totalSales with appropriate heading
```

Here is the function’s actual C++ code:

```cpp
void showTotals(const double sales[], const int units[], int num)
{
   int totalUnits = 0;
   double totalSales = 0.0;
   for (int index = 0; index < num; index++)
   {
       totalUnits += units[index];
       totalSales += sales[index];
   }
   cout << "Total Units Sold: " << totalUnits << endl;
   cout << "Total Sales:     $" << totalSales << endl;
}
```

The Entire Program

- Program 8-6 shows the entire program’s source code.

Program 8-6

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

void calcSales(const int[], const double[], double[], int);
void showOrder(const double[], const int[], int);
void dualSort(int[], double[], int);
void showTotals(const double[], const int[], int);
void swap(double&, double&);
void swap(int&, int&);

int main()
{
   const int NUM_PRODS = 9;

   int id[NUM_PRODS] = { 914, 915, 916, 917, 918,
                         919, 920, 921, 922 };

   int units[NUM_PRODS] = { 842, 416, 127, 514, 437,
                            269, 97,  492, 212 };

   double prices[NUM_PRODS] = { 12.95, 14.95, 18.95, 16.95, 21.95,
                                31.95, 14.95, 14.95, 16.95 };

   double sales[NUM_PRODS];

   calcSales(units, prices, sales, NUM_PRODS);

   dualSort(id, sales, NUM_PRODS);

   cout << setprecision(2) << fixed << showpoint;

   showOrder(sales, id, NUM_PRODS);

   showTotals(sales, units, NUM_PRODS);
   return 0;
}


void calcSales(const int units[], const double prices[], double sales[], in
{
   for (int index = 0; index < num; index++)
      sales[index] = units[index] * prices[index];
}


void dualSort(int id[], double sales[], int size)
{
   int start, maxIndex, tempid;
   double maxValue;

   for (start = 0; start < (size - 1); start++)
   {
      maxIndex = start;
      maxValue = sales[start];
      tempid = id[start];
      for (int index = start + 1; index < size; index++)
      {
         if (sales[index] > maxValue)
         {
            maxValue = sales[index];
            tempid = id[index];
            maxIndex = index;
         }
      }
      swap(sales[maxIndex], sales[start]);
      swap(id[maxIndex], id[start]);
   }
}

void swap(double &a, double &b)
{
   double temp = a;
   a = b;
   b = temp;
}

void swap(int &a, int &b)
{
   int temp = a;
   a = b;
   b = temp;
}


void showOrder(const double sales[], const int id[], int num)
{
   cout << "Product Number\tSales\n";
   cout << "----------------------------------\n";
   for (int index = 0; index < num; index++)
   {
      cout << id[index] << "\t\t$";
      cout << setw(8) << sales[index] << endl;
   }
   cout << endl;
}


void showTotals(const double sales[], const int units[], int num)
{
   int totalUnits = 0;
```

```
    double totalSales = 0.0;

    for (int index = 0; index < num; index++)
    {
       totalUnits += units[index];
       totalSales += sales[index];
    }
    cout << "Total units Sold:  " << totalUnits << endl;
    cout << "Total sales:      $" << totalSales << endl;
 }
```

Program Output

8.5 Sorting and Searching `vector`s (Continued from Section 7.11)

Concept:

- The sorting and searching algorithms discussed in this chapter can be applied to STL `vector`s just as they are to arrays.

- Once an STL `vector` is defined and populated, you can use the algorithms from this chapter to sort and search it.
- You simply need to substitute `vector` syntax (e.g., `v.size()`, `v[index]`) for array syntax.

- Program 8-7 demonstrates using the selection sort and binary search algorithms with a `vector` of strings.

Program 8-7

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void selectionSort(vector<string>&);
void swap(string &, string &);
int binarySearch(const vector<string>&, string);

int main()
{
   string searchValue;
   int position;

   vector<string> names{ "Lopez", "Smith", "Pike", "Jones",
                         "Abernathy", "Hall", "Wilson", "Kimura",
                         "Alvarado", "Harrison", "Geddes", "Irvine" };

   selectionSort(names);

   cout << "Here are the sorted names:\n";
   for (auto element : names)
      cout << element << endl;
   cout << endl;

   cout << "Enter a name to search for: ";
   getline(cin, searchValue);
   position = binarySearch(names, searchValue);

   if (position != -1)
      cout << "That name is found at position " << position << endl;
   else
      cout << "That name is not found.\n";

   return 0;
}

void selectionSort(vector<string> &v)
{
   int minIndex;
   string minValue;

   for (int start = 0; start < (v.size() - 1); start++)
   {
      minIndex = start;
      minValue = v[start];
      for (int index = start + 1; index < v.size(); index++)
      {
         if (v[index] < minValue)
         {
            minValue = v[index];
            minIndex = index;
         }
      }
      swap(v[minIndex], v[start]);
   }
}

void swap(string &a, string &b)
{
   string temp = a;
   a = b;
   b = temp;
}


int binarySearch(const vector<string> &v, string str)
{
   int first = 0,
       last = v.size() - 1,
       middle,
       position = -1;
   bool found = false;

   while (!found && first <= last)
   {
      middle = (first + last) / 2;
      if (v[middle] == str)
      {
         found = true;
         position = middle;
      }
      else if (v[middle] > str)
```

```
            last = middle - 1;
        else
            first = middle + 1;
    }
    return position;
}
```

Program Output

Program Output

Review Questions and Exercises

Short Answer

1. Why is the linear search also called “sequential search”?
2. If a linear search function is searching for a value that is stored in the last element of a 10,000-element array, how many elements will the search code have to read to locate the value?
3. In an average case involving an array of N elements, how many times will a linear search function have to read the array to locate a specific value?
4. A binary search function is searching for a value that is stored in the middle element of an array. How many times will the function read an element in the array before finding the value?
5. What is the maximum number of comparisons that a binary search function will make when searching for a value in a 1,000-element array?
6. Why is the bubble sort inefficient for large arrays?
7. Why is the selection sort more efficient than the bubble sort on large arrays?

Fill-in-the-Blank

1. The _______ search algorithm steps sequentially through an array, comparing each item with the search value.
2. The _______ search algorithm repeatedly divides the portion of an array being searched in half.
3. The _______ search algorithm is adequate for small arrays but not large arrays.
4. The _______ search algorithm requires that the array’s contents be sorted.
5. If an array is sorted in _______ order, the values are stored from lowest to highest.
6. If an array is sorted in _______ order, the values are stored from highest to lowest.

True or False

1. T F If data are sorted in ascending order, it means they are ordered from lowest value to highest value.

2. T F If data are sorted in descending order, it means they are ordered from lowest value to highest value.

3. T F The *average* number of comparisons performed by the linear search on an array of N elements is N/2 (assuming the search values are consistently found).

4. T F The *maximum* number of comparisons performed by the linear search on an array of N elements is N/2 (assuming the search values are consistently found).

5. Complete the following table calculating the average and maximum number of comparisons the linear search will perform, and the maximum number of comparisons the binary search will perform.

ARRAY SIZE ➞	**50 ELEMENTS**	**500 ELEMENTS**	**10,000 ELEMENTS**	**100,000 ELEMENTS**	**10,000,000 ELEMENTS**
Linear Search (Average Comparisons)					
Linear Search (Maximum Comparisons)					
Binary Search (Maximum Comparisons)					

Programming Challenges

1. Charge Account Validation

   Write a program that lets the user enter a charge account number. The program should determine if the number is valid by checking for it in the following list:

```
5658845  4520125  7895122  8777541  8451277  1302850
8080152  4562555  5552012  5050552  7825877  1250255
```

```
1005231  6545231  3852085  7576651  7881200  4581002
```

The list of numbers above should be initialized in a single-dimensional array. A simple linear search should be used to locate the number entered by the user. If the user enters a number that is in the array, the program should display a message saying the number is valid. If the user enters a number that is not in the array, the program should display a message indicating the number is invalid.

2. Lottery Winners

   A lottery ticket buyer purchases ten tickets a week, always playing the same ten 5-digit “lucky” combinations. Write a program that initializes an array or a `vector` with these numbers, then lets the player enter this week’s winning 5-digit number. The program should perform a linear search through the list of the player’s numbers and report whether or not one of the tickets is a winner this week. Here are the numbers:

   ```
   13579  26791  26792  33445  55555
   62483  77777  79422  85647  93121
   ```

3. Lottery Winners Modification

   Modify the program you wrote for Programming Challenge 2 (Lottery Winners) so it performs a binary search instead of a linear search.

   VideoNote

   Solving the Charge Account Validation Modification Problem

4. Charge Account Validation Modification

   Modify the program you wrote for Problem 1 (Charge Account Validation) so it performs a binary search to locate valid account numbers. Use the selection sort algorithm to sort the array before the binary search is performed.

5. Rainfall Statistics Modification

   Modify the Rainfall Statistics program you wrote for Programming Challenge 2 of Chapter 7 (Rainfall Statistics). The program should display a list of months, sorted in order of rainfall, from highest to lowest.

6. String Selection Sort

Modify the `selectionSort` function presented in this chapter so it sorts an array of strings instead of an array of `int`s. Test the function with a driver program. Use Program 8-8 as a skeleton to complete.

Program 8-8

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    const int NUM_NAMES = 20;
    string names[NUM_NAMES] = {"Collins, Bill", "Smith, Bart", "Allen, Jim",
                                  "Griffin, Jim", "Stamey, Marty", "Rose, Geri
                             "Taylor, Terri", "Johnson, Jill",
                                  "Allison, Jeff", "Looney, Joe", "Wolfe, Bill
                            "James, Jean", "Weaver, Jim", "Pore, Bob",
                             "Rutherford, Greg", "Javens, Renee",
                            "Harrison, Rose", "Setzer, Cathy",
                            "Pike, Gordon", "Holland, Beth" };
    return 0;
}
```

7. Binary String Search

   Modify the `binarySearch` function presented in this chapter so it searches an array of strings instead of an array of `int`s. Test the function with a driver program. Use Program 8-8 as a skeleton to complete. (The array must be sorted before the binary search will work.)

8. Search Benchmarks

   Write a program that has an array of at least 20 integers. It should call a function that uses the linear search algorithm to locate one of the values. The function should keep a count of the number of comparisons it makes until it finds the value. The program then should call a function that uses the binary search algorithm to locate the same value. It should also keep count of the number of comparisons it makes. Display these values on the screen.

9. Sorting Benchmarks

Write a program that uses two identical arrays of at least 20 integers. It should call a function that uses the bubble sort algorithm to sort one of the arrays in ascending order. The function should keep a count of the number of exchanges it makes. The program then should call a function that uses the selection sort algorithm to sort the other array. It should also keep count of the number of exchanges it makes. Display these values on the screen.

10. Sorting Orders

    Write a program that uses two identical arrays of just eight integers. It should display the contents of the first array, then call a function to sort the array using an ascending order bubble sort modified to print out the array contents after each pass of the sort. Next, the program should display the contents of the second array, then call a function to sort the array using an ascending order selection sort modified to print out the array contents after each pass of the sort.

11. Using Files—String Selection Sort Modification

    Modify the program you wrote for Programming Challenge 6 (String Selection Sort) so it reads in 20 strings from a file. The data can be found in the names.txt file.

12. Sorted List of 1994 Gas Prices

    In the student sample programs for this book, you will find a text file named 1994_Weekly_Gas_Averages.txt. The file contains the average gas price for each week in the year 1994. (There are 52 lines in the file. Line 1 contains the average price for week 1; line 2 contains the average price for week 2, and so forth.) Write a program that reads the gas prices from the file, and calculates the average gas price for each month. (To get the average price for a given month, calculate the average of the average weekly prices for that month.) Then, the program should create another file that lists the names of the months, along with each month's average gas price, sorted from lowest to highest.