Appendix B

Searching and Sorting Arrays

Introduction to Search Algorithms

- A **search algorithm** is a method of locating a specific item of information in a larger collection of data (array).

- This section discusses two algorithms for searching the contents of an array:
  - **Linear Search**
  - **Binary Search**

The **Linear** Search

- This is a very simple algorithm.
- It uses a **loop** to sequentially step through an array, starting with the first element.
- It **<u>compares each element with the value being searched for</u>** and stops when that value is found or the end of the array is reached.

Linear Search Algorithm:

```
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
```

Program 1

```
// This program demonstrates a searchList function; it
// performs a linear search on an integer array.

#include <iostream>
using namespace std;

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

const int ARR_SIZE = 5;

int main( )
{
   int tests[ARR_SIZE] = {87, 75, 98, 100, 82};
   int results;
```

Program continues:

```
results = searchList(tests, ARR_SIZE, 100);
        // tests    -->  list
        // ARR_SIZE -->  number-of-elements
        // 100      -->  search-value
        // results  <--  position

 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;
}
```

Program continues:

```
// The searchList function performs a linear search on an
// integer array. The array list, which has a maximum of numElems
// elements, is searched for the number stored in value. If the
// number is found, its array subscript is returned. Otherwise,
// -1 is returned indicating the value was not in the array.

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

   while (index < numElements && !found)
   {
         if ( list[index] == value )
         {
            found = true;
            position = index;
          }

          index++;
    }

    return position;
}
```

Program Output:

You earned 100 points on test 4

Linear Search – Tradeoffs / Efficiency

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order
- Disadvantages:
  - Inefficient (slow)
  - for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

Binary Search

- The binary search is much more efficient than the linear search.
- **It requires the list to be in order.**
- The algorithm starts searching with the **middle element**.
  - If the item is less than the middle element, it starts over searching the <u>first half</u> of the list.
  - If the item is greater than the middle element, the search starts over starting with the middle element in the <u>second half</u> of the list.
  - It then continues halving the list until the item is found.

Binary Search - Example

- Array **numlist2** contains:

2	3	5	11	17	23	29

- Searching for the the value **11**, binary search examines **11** and stops

- Searching for the the value **7**, binary search examines **11, 3, 5,** and stops

Program 2

```
// This program demonstrates the binarySearch function, which
// performs a binary search on an integer array.

#include <iostream>
using namespace std;

// Function prototype
int binarySearch( int [], int, int );

const int arrSize = 20;

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

Program continues:

```
    int results, empID;

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

    results = binarySearch(tests, arrSize, 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;
}
```

Program continues:

```
// The binarySearch function performs a binary search on an integer array. Array,
// which has a maximum of numElems elements, is searched for the number
// stored in value. If the number is found, its array subscript is returned.
// Otherwise, -1 is returned indicating the value was not in the array.

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

    while ( first <= last && !found )
    {
        middle = (first + last) / 2;     // Calculate mid point

        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 Output with Example Input:

Enter the Employee ID you wish to search for: **199**

That ID is found at element 4 in the array.

Efficiency of the Binary Search

- Much more efficient than the linear search.
- The **minimum** number of comparisons that the binary search will perform is 1
- The **maximum** number of comparisons that the binary search will perform is x, where:

$2^x > N$ where N is the number of elements in the array

[$\log_2 N$ comparisons where N = array size] check performed:

if (array[middle] == value)

true -> done

array[middle] < value -> f = m + 1

array[middle] > value -> l = m - 1

Binary Search Example searching for a value of **10**

index	0	1	2	3	4	5	6	7	8	9			
value	2	3	5	6	7	9	11	12	13	15	first	last	middle
					7						0	9	4
								12			M+1 5	9	7
						9					5	M-1 6	5
							11				M+1 6	6	6
											6	M-1 5	stop

The search value 10 was not found.

Binary Search Example searching for a value of 13

index	0	1	2	3	4	5	6	7	8	9			
value	2	3	5	6	7	9	11	12	13	15	first	last	middle
					7						0	9	4
								12			5	9	7
									13		8	9	8

The search value 13 was found at element 8

Binary Search Example searching for a value of 2

index	0	1	2	3	4	5	6	7	8	9			
value	2	3	5	6	7	9	11	12	13	15	first	last	middle
					7						0	9	4
		3									0	3	1
	2										0	0	0

The search value 2 was found at element 0

Introduction to Sorting Algorithms

- Sort: **arrange values into an order**
  - Alphabetical
  - Ascending numeric
  - Descending numeric
- Two algorithms considered here:
  - **Bubble** sort
  - **Selection** sort

Bubble Sort

Concept:

- Compare 1st two elements
  - If out of order, swap them to put them in order
- Move to next element, compare 2nd and 3rd elements, swap them if necessary. Continue until end of array.
- Pass through array again, swapping as necessary
- Repeat until pass made with no swaps
- passes / compares

The Bubble Sort

- An easy way to arrange data in ascending or descending order.
- Pseudocode [sort in ascending order]

```
Do
    Set swap flag to 0

    For count is set to each subscript in Array from 0 to the next-to-last subscript

        If   array[count]   is greater than   array[count+1]
                swap them
                set swap flag to 1
        End if

     End for

While any elements have been swapped.
```

Bubble Sort Example (pass 1)

Array **`numlist3`** contains:

17	23	5	11

compare values **`17`** and **`23`** – in correct order, so no exchange

compare values **`23`** and **`5`** – not in correct order, so exchange them

compare values **`23`** and **`11`** – not in correct order, so exchange them

Bubble Sort Example (pass 2)

After first pass, array **`numlist3`** contains:

17	5	11	23

compare values **17** and **5** – not in correct order, so exchange them

compare values **17** and **11** – not in correct order, so exchange them

compare values **17** and **23** – in correct order, so no exchange

Bubble Sort Example (pass 3)

After second pass, array **`numlist3`** contains:

5	**11**	**17**	**23**

compare values **`5`** and **`11`** – in correct order, so no exchange

compare values **`11`** and `17` – in correct order, so no exchange

compare values **`17`** and **`23`** – in correct order, so no exchange

No exchanges, so array is in order

Program 4

```
// This program uses the bubble sort algorithm to sort an
// array in ascending order.
#include <iostream.h>

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

int main( )
{
   int values[6] = {7, 2, 3, 8, 9, 1};

   cout << "The unsorted values are:\n";
   showArray(values, 6);

   sortArray(values, 6);

   cout << "The sorted values are:\n";
   showArray(values, 6);

   return 0;
}
```

Program continues:

```
// Definition of function sortArray. This function performs an ascending
// order bubble sort on Array. elems is the number of elements in the array.

void sortArray(int array[], int elems)
{
   int swap, temp;
   do
   {
       swap = 0;
       for (int count = 0; count < (elems - 1); count++)
       {
          if (array[count] > array[count + 1])
          {
              temp = array[count];
              array[count] = array[count + 1];
              array[count + 1] = temp;
              swap = 1;
          }
       }
   } while (swap != 0);
}
```

Program continues:

```
// Definition of function showArray.
// This function displays the contents of array. elems is the
// number of elements.

void showArray(int array[], int elems)
{
   for (int count = 0; count < elems; count++)
       cout << array[count] << " ";

   cout << endl;
}
```

Program Output:

The unsorted values are:
7 2 3 8 9 1

The sorted values are:

1 2 3 7 8 9

Bubble Sort - Tradeoffs

- Benefit:
  - Easy to understand and implement

- Disadvantage:
  - Inefficient: slow for large arrays

The **Selection** Sort

- The **bubble sort** is inefficient for large arrays because items only move by one element at a time.

- The **selection sort** moves items immediately to their final position in the array so it makes fewer exchanges (swaps)

Selection Sort

- Concept for sort in ascending order:
  - Locate smallest element in array. Exchange it with element in position 0
  - Locate next smallest element in array. Exchange it with element in position 1.
  - Continue until all elements are arranged in order

Selection Sort – pass 1

Array **`numlist`** contains:

11	2	29	3

1. Smallest element is **2**. Exchange **2** with element in **1st** position in array:

2	11	29	3

Selection Sort – pass 2, 3

2. Next smallest element is **3**. Exchange **3** with element in **2nd** position in array:

2	3	29	11

3. Next smallest element is **11**. Exchange **11** with element in **3rd** position in array:

2	3	11	29

Selection Sort Pseudocode:

```
For Start is set to each subscript in Array from 0 through the next-to-last subscript
   Set Index variable to Start
   Set minIndex variable to Start
   Set minValue variable to array[Start]

   For Index is set to each subscript in Array from Start+1 through the next-to-last subscript
      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[Start]
   Set array[Start] to minValue

End For
```

Program 5

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

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

int main( )
{
   int values[6] = {5, 7, 2, 8, 9, 1};

   cout << "The unsorted values are\n";
   showArray(values, 6);

   selectionSort(values, 6);

   cout << "The sorted values are\n";
   showArray(values, 6);

   return 0;
}
```

Program continues:

```
// Definition of function selectionSort. This function performs an
// ascending order selection sort on Array. elems is the number of
// elements in the array.

void selectionSort(int array[], int elems)
{
    int startScan, minIndex, minValue;
    for (startScan = 0; startScan < (elems - 1); startScan++)
    {
          minIndex = startScan;
          minValue = array[startScan];
          for(int index = startScan + 1; index < elems; index++)
          {
             if (array[index] < minValue)
             {
                 minValue = array[index];
                 minIndex = index;
             }
          }
          array[minIndex] = array[startScan];
          array[startScan] = minValue;
    }
}
```

Program continues:

```
// Definition of function showArray.
// This function displays the contents of Array.
// elems is the number of elements.

void showArray(int array[], int elems)
{
   for (int count = 0; count < elems; count++)
       cout << array[count] << " ";

   cout << endl;
}
```

Program Output:

The unsorted values are
5 7 2 8 9 1

The sorted values are

1 2 5 7 8 9

Sorting and Searching Vectors

- Sorting and searching algorithms can be applied to vectors as well as arrays

- Need slight modifications to functions to use vector arguments:
  - `vector <type> &` used in prototype
    - **Because by default, vectors are passed by value**
  - No need to indicate vector size – functions can use `size` member function to calculate

Program 7

```
// This program produces a sales report for the Demetris
// Leadership Center. This version of the program uses vectors.

#include <iostream>
#include <iomanip>
#include <vector>                    // Needed to declare vectors
using namespace std;                 // vectors are in the std namespace

// Function prototypes
void initVectors(vector<int> &, vector<int> &, vector<double> &);
void calcSales(vector<int>, vector<double>, vector<double> &);
void showOrder(vector<double>, vector<int>);
void dualSort(vector<int> &, vector<double> &);
void showTotals(vector<double>, vector<int>);

void main(void)
{
        vector<int> id;
        vector<int> units;
        vector<double> prices;
        vector<double> sales;
```

Program 7 (continued)

```
    // Must provide an initialization routine.
    initVectors(id, units, prices);

    // Calculate and sort the sales totals,
    // and display the results.

    calcSales(units, prices, sales);
    dualSort(id, sales);

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

    showOrder(sales, id);
    showTotals(sales, units);
}
```

Program 7 (continued)

```
//*********************************************************************
// Definition of initVectors. Accepts id, units, and prices          *
// vectors as reference arguments. This function initializes each   *
// vector to a set of starting values.                               *
//*********************************************************************
void initVectors(vector<int> &id, vector<int> &units,
                 vector<double> &prices)
{
       // Initialize the id vector
       for (int value = 914; value <= 922; value++)
              id.push_back(value);

       // Initialize the units vector
       units.push_back(842);    units.push_back(416);
       units.push_back(127);    units.push_back(514);
       units.push_back(437);    units.push_back(269);
       units.push_back( 97);    units.push_back(492);
       units.push_back(212);
```

Program 7 (continued)

```
    // Initialize the prices vector
    prices.push_back(12.95);
    prices.push_back(14.95);
    prices.push_back(18.95);
    prices.push_back(16.95);
    prices.push_back(21.95);
    prices.push_back(31.95);
    prices.push_back(14.95);
    prices.push_back(14.95);
    prices.push_back(16.95);
}

//****************************************************************
// Definition of calcSales. Accepts units, prices, and sales     *
// vectors as arguments. The sales vector is passed into a       *
// reference parameter. This function calculates each product's  *
// sales by multiplying its units sold by each unit's price. The *
// result is stored in the sales vector.                         *
//****************************************************************
```

Program 7 (continued)

```
void calcSales(vector<int> units, vector<double> prices,
               vector<double> &sales)
{
        for (int index = 0; index < units.size(); index++)
                sales.push_back(units[index] * prices[index]);
}

//******************************************************************
// Definition of function dualSort. Accepts id and sales vectors *
// as reference arguments. This function performs a descending   *
// order selection sort on the sales vector. The elements of the *
// id vector are exchanged identically as those of the sales     *
// vector.                                                        *
//******************************************************************

void dualSort(vector<int> &id, vector<double> &sales)
{
        int startScan, maxIndex, tempid, elems;
        float maxValue;
```

Program 7 (continued)

```
        elems = id.size();
        for (startScan = 0; startScan < (elems - 1); startScan++)
        {
            maxIndex = startScan;
            maxValue = sales[startScan];
            tempid   = id[startScan];
            for(int index = startScan + 1; index < elems; index++)
            {
                if (sales[index] > maxValue)
                {
                    maxValue = sales[index];
                    tempid   = id[index];
                    maxIndex = index;
                }
        }
        sales[maxIndex]  = sales[startScan];
        id[maxIndex]     = id[startScan];
        sales[startScan] = maxValue;
        id[startScan]    = tempid;
    }
}
```

Program 7 (continued)

```
//*******************************************************************
// Definition of showOrder function. Accepts sales and id vectors *
// as arguments. The function first displays a heading, then the  *
// sorted list of product numbers and sales.                      *
//*******************************************************************

void showOrder(vector<double> sales, vector<int> id)
{
        cout << "Product number\tsales\n";
        cout << "----------------------------------\n";

        for (int index = 0; index < id.size(); index++)
        {
                cout << id[index] << "\t\t$";
                cout << setw(8) << sales[index] << endl;
        }

        cout << endl;
}
```

Program 7 (continued)

```
//*******************************************************************
// Definition of showTotals function. Accepts sales and id vectors  *
// as arguments. The function first calculates the total units (of  *
// all products) sold and the total sales. It then displays these   *
// amounts.                                                         *
//*******************************************************************

void showTotals(vector<double> sales, vector<int> units)
{
	int totalUnits = 0;
	float totalSales = 0.0;

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

Program 7 (continued)

Program Output

```
Product number  sales
----------------------------------
914             $10903.90
918             $ 9592.15
917             $ 8712.30
919             $ 8594.55
921             $ 7355.40
915             $ 6219.20
922             $ 3593.40
916             $ 2406.65
920             $ 1450.15

Total units Sold:  3406
Total sales:      $58827.70
```