1. The following is the textbook's code for a linear search on an unsorted array.

```c
//*****************************************************************
// 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 < numElems && !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
    } // end while

    return position; // Return the position, or -1
} // end searchList
```

a) Trace the `searchList` code using the following data.

```
<table>
<thead>
<tr>
<th>index</th>
<th>list:</th>
<th>numElems:</th>
<th>value:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3 9 5 7 8 2 4</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
```

b) If numElems is 1,000,000, then how many comparisons of array items would you expect on a successful search?

c) If numElems is 1,000,000, then what is the maximum number of comparisons of array items on a successful search?

d) If numElems is 1,000,000, then how many comparisons of array items would you expect on an unsuccessful search?

e) If there are 1,000,000 names in a phone book, what algorithm do you use to search for a name, say "Smith, John".
2. The following is the textbook's code for a *binary search* on an array sorted in *ascending order* (items get bigger as we go from left to right).

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

    while (!found && first <= last) { // Calculate mid point
        middle = (first + last) / 2;
        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 // If value is in upper half
            first = middle + 1;
    } // end while
    return position; // end binarySearch
}
```

a) Trace the *binarySearch* code using the following data.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>(MAX-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

array: 2 3 4 5 7 8 9 size: 7 value: 2

b) If size is 1,000,000, then what is the maximum number of comparisons of array items on a *successful search*?

c) If size is 1,000,000, then how many comparisons of array items would you expect on an *unsuccessful search*?
All simple sorts consist of two nested loops where:
- the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
- the inner loop's job is to do the work to extend the sorted part's size by one.

Initially, the sorted part is typically empty. The simple sorts differ in how their inner loops perform their job.

3. Selection sort is an example of a simple sort. Selection sort’s inner loop scans the unsorted part of the array to find the minimum item. The minimum item in the unsorted part is then exchanged with the first unsorted item to extend the sorted part by one item.

At the start of the first iteration of the outer loop, initial array is completely unsorted:

```
<table>
<thead>
<tr>
<th>Empty Sorted Part</th>
<th></th>
<th>Unsorted Part</th>
<th></th>
<th>size: 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>25 35 20 40 90 60 10 50 45</td>
<td>● ● ●</td>
<td></td>
</tr>
</tbody>
</table>
```

The inner loop scans the unsorted part and determines that the index of the minimum item, minIndex = 6.

```
<table>
<thead>
<tr>
<th>Sorted Part</th>
<th></th>
<th>Unsorted Part</th>
<th></th>
<th>size: 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>25 35 20 40 90 60 10 50 45</td>
<td>● ● ●</td>
<td></td>
</tr>
</tbody>
</table>
```

firstUnsortedIndex = 0      minIndex = 6

After the inner loop (but still inside the outer loop), the item at minIndex is exchanged with the item at firstUnsortedIndex. Thus, extending the Sorted Part of the array by one item.

```
<table>
<thead>
<tr>
<th>Sorted Part</th>
<th></th>
<th>Unsorted Part</th>
<th></th>
<th>size: 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 1 2 3 4 5 6 7 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 35 20 40 90 60 25 50 45</td>
<td>● ● ●</td>
<td></td>
</tr>
</tbody>
</table>
```

firstUnsortedIndex = 0      minIndex = 6

a) Write the code for the inner loop

b) Write the code to exchange the items
4. **Bubble sort** is another example of a simple sort. Bubble sort’s inner loop scans the unsorted part of the array comparing adjacent items. If it finds adjacent items out of order, then it exchanges them. This causes the largest item to “bubble” up to the “top” of the unsorted part of the array.

**At the start of the first iteration** of the outer loop, initial array is completely unsorted:

![Diagram of an initial array](image)

The inner loop scans the unsorted part and determines that the index of the minimum item, `minIndex = 6`.

![Diagram showing the process of bubble sort](image)

After the inner loop (but still inside the outer loop), there is nothing to do since the exchanges occurred inside the inner loop.

a) What would be true if we scanned the unsorted part and didn’t need to do any exchanges?