Goal
Build a program that will read an arbitrary collection of integers into an array and then sort them into ascending order using a bubble sort.

Information on Sorting
• Sorting is a very common and useful operation in computing. A sorting algorithm will take a set of values in an arbitrary order and put them into either ascending or descending order.
• Example:

  Unsorted array

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

  Sorted array

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

• There are many well-known sorting algorithms.
  o Bubble Sort
  o Insertion Sort
  o Selection Sort
  o Merge Sort
  o Quicksort

• In this lab, you will be implementing a bubble sort, which is one of the simplest sorting algorithms.
  o The basic strategy on which it relies is to swap adjacent elements that are out of order until all values are in order.
  o A common sorting function employs an algorithm that uses a nested for() loop.
    • The outer for() loop will start in the same manner, determining how many passes the inner loop will make.
    • The inner for() loop will start at 1 and end at the last index, making comparisons and swaps.
      • If the value at index j is less than the value at index j−1, swap the two values.
      • If sorting in ascending order, this moves the largest unsorted value to its proper location (it “bubbles up” to its proper location).
      • An advanced way to speed up the algorithm is to steadily reduce the number of iterations in the inner loop per each iteration of the outer loop.
  o Thus, if we had an array of 50 elements, and we applied the above strategy 49 times, we would be guaranteed that the array would end up with no elements out of order.
Assignment

Write a program called `lab9.c` that contains two functions as follows:

1. Create a `main()` function, that contains an array of capacity of 100 ints: `int numArray[100]` and a variable that will contain the actual number of values in the array: `int counter`
   a. Your `main()` function should use a while loop to read integer values from the standard input (or from an input file using input redirection) into adjacent elements in the array, and set the counter to the total number of integers read.
   b. Use the following command to copy a sample input file of integers (one on each line) to your directory:
      
      ```
      ```
   c. In a separate loop after the read loop completes, print all the values using “%d
   d. Before moving on to the second step, show your ta you have this much working.

2. Create a function called `sort(int array[], int count)`
   a. This function will be passed an integer array along with the actual number of values in the array.
   b. The function will use a loop for each pass through the array.
   c. The function should return the number of swaps (not comparisons) that it performed (a swap is made when two values change places).
   d. The function must NOT contain any instances of `fscanf()` or `fprintf()`.

3. Now return to the `main()` function and insert a third loop after the reading loop and before the printing loop. This should iterate exactly `counter` times; the body of this loop should invoke the `sort()` function, passing it `numArray` and `counter`. This is the loop that controls the number of passes through the array.
   a. Before you move on to the improvements, show your ta that you have this much working.

4. Improvement #1: Passing `counter` every time actually does more work than is necessary in `sort()`. After the first call to `sort()`, the largest element is guaranteed to be in `numArray[counter-1]`. After the second call to `sort()`, the last two elements of the array will be correct. Eliminate the unnecessary work by passing decreasing values to `sort()`. Add code to print out the total number of swaps that occurred after all the sorting is finished.

5. Improvement #2: Suppose the integers are already in sorted order at the time they are read in. Then there is nothing to do, but our program will still call `sort()` `counter` times. If you ever call `sort()` and it doesn’t swap anything, then the values are in proper order and the sort is finished. Modify the loop so that it will terminate if `sort()` doesn’t swap any values. Add code to print out the number of times that the `sort()` function was called.

Turn In Work

Show your TA that you completed the assignment. Then turn in your lab9.c program using the handin page at [http://handin.cs.clemson.edu](http://handin.cs.clemson.edu)