CSC 2111 **Spring 2013**

Name(s): _____

Lab 13: Sorting

Reference: *C++ Plus Data Structures*
Chapter 10.1 - "Sorting"

Objective: To work with and understand various sorting algorithms.

Files: *See Below*

Introduction

Sorting is a very important problem in processing data, especially for large data sets. The fact that the data is ordered allows a program to find elements quickly. But before that can happen, the data has to be sorted in the first place. In this lab exercise, you will investigate some of the sorting algorithms described in Chapter 10 of your book.

In the first part of the exercise you will investigate how these sorting schemes work, and in the second part you will measure their computing times so you can compare their efficiencies:

Note: You will be using several files that you can get from the class iLearn web-site:

- *SortLibrary.h*: containing function templates for various sorting methods
- *sortdriverl.cpp*: used in the first part of the lab exercise to test sorting methods
- *Timer.h*: used to time code in the second part of the lab exercise
- *sortdriver2.cpp*: used in the second part of the lab exercise to time sorting methods
- Several data files that vary in size from about 15 words to 13,400 words. These files are:
  - *bach.txt*,
  - *frost.txt*,
  - *gettysburg.txt*,
  - *moby-ch1.txt*,
  - *confessions1.txt*,
  - *confessions2.txt*, and
  - *confessions3.txt*.

Output Operator

In this lab, it will be useful to have a generic output operator to display the elements in a vector regardless of the type of elements it contains. Adding that operator in *sortdriverl.cpp* will be your first step. Note: without this operator your program won't compile.

Add a prototype and a definition of a function template for `operator<<()` in *sortdriverl.cpp* where indicated.

Use a template declaration with a type parameter `ElementType`.

`operator<<`'s signature should be:

```
ostream &operator<<(ostream &out, const vector<ElementType>
&v)
```

Test your function by compiling and executing *sortdriverl.cpp*. It should display an unsorted list and then a sorted list (using bubblesort).

NOTE: *If working in Unix, don't name your binary executable sort—"sort" is a Unix command.*

Does you program compile and work as expected (above)? Y/N

An Exchange Sort: Bubblesort *(See Chapter 10.1, pages 633-637 in the text)*

Bubblesort is a sorting algorithm that is found in many texts that include a discussion of sorting. It is often coded as follows. (Note that `bubbleSort()` is a function template, so it can be used to sort any type of values for which it is defined.)

```
template <class ElementType>
void bubbleSort(vector<ElementType> & x)
{
  int numElements = x.size();
  for (int listEnd = numElements-1; listEnd > 0; listEnd--)
     for (int j = 0; j < listEnd; j++)
        if (x[j] > x[j+1])
           interchange(x[j], x[j+1]);
}
```

It makes repeated passes through the list of elements or a part of the list, **exchanging** pairs that are out of order. Because the largest element is guaranteed to move to the end of the list, the list size can be reduced by one after each pass. The following diagrams

illustrate the action of *bubblesort* on the list 88, 55, 11, 44 (inserted into the list in this order). The parentheses indicate items that were properly positioned on the preceding pass and thus don't need to be considered on the current pass. Note that the largest value *sinks* to the end of the list on each pass, but smaller values *bubble up* toward the beginning.

Pass 1	Pass 2	Pass 3
88 55 11 44	55 11 44 (88)	11 44 (55 88)
55 88 11 44	11 55 44 (88)	11 44 (55 88)
55 11 88 44	11 44 55 (88)	
55 11 44 88		

Draw similar diagrams for the following lists:

List in reverse order: 88, 55, 44, 11

Pass 1	Pass 2	Pass 3

List already in order: 11, 44, 55, 88

Pass 1	Pass 2	Pass 3

Compile and execute *sortdriverl.cpp* to verify that the version of `bubbleSort()` in *SortLibrary.h* correctly sorts the list of integers 10, 7, 2, *5,* 11.

Does you program compile and work as expected? Y/N

Bubblesort with Sort Detection

The preceding version of *bubblesort* works, but we can modify it slightly to make it more efficient. From the examples, it should be clear that for a list with *n* elements, *it will always make n-1 passes through the list, even if the list is sorted!* It can be improved by checking on each pass if the list is sorted, and if it is, stop processing it. An algorithm for this version is implemented in *SortLibrary.h* in the section labeled SECOND VERSION OF BUBBLE SORT BEGINS HERE.

Make diagrams like those in Step 1 to show the interchanges that take place on each pass through the while loop in "bubblesorting" the list 70, 30, 20, 80, 40, 50, 60 with this improved version.

Pass 1	Pass 2	Pass 3

How many passes through the loop did it take to complete the sorting? _____

How many passes through the loop would the first version of bubblesort require? ____

Comment out (or remove) the first version of `bubbleSort()` in *SortLibrary.h* and comment out with // (or remove) the comment delimiters that enclose the improved version. Compile and execute *sortdriverl.cpp* to test that it works.

Does you program compile and work as expected? Y/N