Intro to Computing Lab 10 October 28, 2009

Objectives: You will gain experience using C++:

- array searching
- array sorting
- timing of functions

Download the following file to your desktop: http://www.cs.uni.edu/~fienup/cs051f09/labs/lab10.zip
Extract this file by right-clicking on lab10.zip icon and selecting Extract All.

Part A: Previously in lecture we looked at the textbook's linear search code which assumed an unsorted array, but in Lab 9 we developed a linear search function that assumed a sorted array.

The lab10.zip file you downloaded and extracted contains a `linearSearch` folder with a Visual Studio C++ project file: `linearSearch.sln` inside. Double-click on it to open this project in Visual Studio. Start the program running by Debug | Start Without Debugging (Ctrl F5). NOTE: This program which takes a minute or two. While its executing, study the code. Obverse that it creates an array, evenList, that holds 100,000 sorted, even values (e.g., evenList = {0, 2, 4, 6, 8, ..., 199996, 199998}). It then measures the execution time (i.e., *times*) several searching algorithms. For each searching algorithm, it first searches for all target values from 0, 2, 4, ..., 199998 so all of the searches are successful, then it searches for all target values from 1, 3, 5, ..., 199999 so all of the searches are unsuccessful. The search algorithms are:

- linearSearchList (the textbook's code from section8.1) that performs a linear search on a possibly unsorted array
- linearSearchSortedList that performs a linear search on a **sorted** array, so it can stop when it sees an element value bigger than the target.

Both algorithms return a -1 result to indicate an unsuccessful search, or the location of the target if successful.

a) Compete the following table with the timings:

Algorithm	Timing of 100,000 Searches (seconds)	
	Successful searches for 0, 2, 4, ...	Unsuccessful searches for 1, 3, 5, ...
linearSearchList		
linearSearchSortedList		

b) For the linearSearchList algorithm, why did the unsuccessful searches take about twice as long as the successful searches?

c) For the linearSearchSortedList algorithm, why did the unsuccessful searches take the same time as the successful searches?

d) Why did the successful searches of linearSearchList take less time than the successful searches of linearSearchSortedList? Modify the linearSearchSortedList code to improve its performance.

After you have answered the above questions, raise your hand and explain your answers.

Part B: Previously in lecture we looked at the textbook's binary search code for a sorted array.

The lab10.zip file also contains a `binarySearch` folder with a Visual Studio C++ project file: `binarySearch.sln` inside. Double-click on it to open this project in Visual Studio. Start the program running by Debug | Start Without Debugging (Ctrl F5). Again, the main function creates an array, evenList, that holds 100,000 sorted, even values (e.g., evenList = {0, 2, 4, 6, 8, ..., 199996, 199998}). It then times the binary search algorithm by first searches for all target values from 0, 2, 4, ..., 199998 so all of the searches are successful, then it searches for all target values from 1, 3, 5, ..., 199999 so all of the searches are unsuccessful. .

a) Compete the following table with the timings:

Algorithm	Timing of 100,000 Searches (seconds)	
	Successful searches for 0, 2, 4, ...	Unsuccessful searches for 1, 3, 5, ...
binarySearch		

b) Why did the binarySearch searches take so much less time than the linearSearchSortedList searches?

After you have answered the above question, raise your hand and explain your answer.

Part C: The lab10.zip file you downloaded and extracted contains a `simpleSorts` folder with a Visual Studio C++ project file: `simpleSorts.sln` inside. Double-click on it to open this project in Visual Studio. Start the program running by Debug | Start Without Debugging (Ctrl F5). NOTE: This program which takes a minute or two. While its executing, study the code. We'll study the following sorting algorithms:

- bubbleSortTextbook - the textbook's original bubble sort code
- bubbleSort - the bubble sort code we developed in class
- selectionSort - the selection sort code we developed in class

For each sorting algorithm:

1. The array randomList is filled with randomly generated integers. NOTE: Since the same seedValue is used to seed the random number generator, each sorting algorithm sorts the same sequence of random numbers.
2. The array containing the random numbers is sorted.
3. The sorting algorithm is called with the sorted array as input.

Initial, the ARRAY_SIZE is set at 1000 items. Complete the following timings by changing the ARRAY_SIZE and rerunning the program.

Timings on Sorting Algorithm (seconds)						
	bubbleSortTextbook		bubbleSort		selectionSort	
listSize	Random items	Sorted items	Random items	Sorted items	Random items	Sorted items
1,000						
2,000						
4,000						
8,000						
16,000						

Study the code and answer the following questions about the sorting algorithms:

a) Why does the bubbleSort algorithm take less time than the bubbleSortTextbook algorithm on random items?

b) Why does the bubbleSort algorithm take A LOT less time on sorted items?

c) Why does the selectionSort algorithm take more time on sorted items?

After you have answered the above questions, raise your hand and explain your answers.

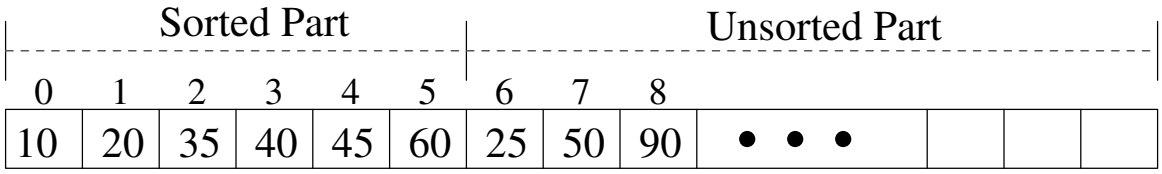
Nothing needs to turned in for this lab. Make sure that you log off the computer before you leave.

EXTRA CREDIT:

If you have time to kill and/or want some extra credit, write the code for another simple sort called insertion sort. Insertion sort is a *simple sort* so:

- 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.

After several iterations of the outer loop, an array might look like:



Insertion sort takes the "first unsorted element" (25 at index 6 in the above example) and "inserts" it into the sorted part of the list "at the correct spot." After 25 is inserted into the sorted part, the array would look like: