Assignment #5: Sorting Lab

Thanks to Eric Roberts and Dave Levine for ideas and prose for Part A.

Due: Mon, Feb 25 2:15pm

Sorting is one of the most fundamental algorithmic problems within computer science. It has been claimed that as many as 25% of all CPU cycles are spent sorting, which provides a lot of incentive for further study and optimization of sorting. In addition, there are many other tasks (searching, calculating the median, finding the mode, remove duplicates, etc.) that can be implemented much more efficiently once the data is sorted. Sorting is spectacular in that there are many varied approaches (divide-and-conquer, partitioning, pigeonhole, etc.) that lead to useful sorting algorithms, and those techniques often can be repurposed to other algorithmic tasks. The wide variety of algorithms gives us a lot of richness to explore, especially when considering the tradeoffs offered in terms of efficiency, operation mix, code complexity, best/worse case inputs, and so on. (In case it wasn't totally obvious in lecture, I love sorting…) Your next task is to do some exploration into sorting algorithms in a playful, experimental way— I hope you enjoy it!

Part A — Sort Detective

Back in the day, students in high school chemistry class were often given an unknown compound and asked to figure out what it was in a process called *qualitative analysis*. In this problem, your job is to take several unknown algorithms and figure out what they are by experimenting with their computational properties in much the same way.

We give you a compiled library containing five sorting routines, named **`MysterySort1`**, **`MysterySort2`**, and so on. Each function is designed to take a vector of numbers and order the elements into ascending order. What's different about them is that each uses a different sorting algorithm. More specifically, the algorithms used are (in alphabetical order):

- Bubble sort
- Insertion sort
- Merge sort
- Quicksort, where first element of subvector is used as pivot
- Selection sort

We talked about four of these algorithms in class and they are covered in Chapter 7 of the reader (Insertion Sort appears as exercise 7-2). The only missing algorithm is bubble sort. Bubble sort is an interesting bit of computer science history. It is a simple $O(N^2)$ algorithm that works by repeatedly stepping through the elements to be sorted, comparing two items at a time, swapping them if they are out of order. The pass through the elements is repeated until no swaps are needed, which means the data is sorted. The algorithm gets its name from the way elements "bubble" to the correct position. Here is pseudo-code for the bubble sort algorithm:

```
loop doing passes over the elements until sorted
   a pass goes from start to end
       compare each adjacent pair of elements
        if the two are out of order, swap
   if no pairs were exchanged on this pass, you're done
```

Bubble sort is a rather weak algorithm that for some strange reason dominates introductory courses (despite the fact that it has just enough subtlety to often be demonstrated with bugs). Owen Astrachan, a friend and colleague of mine at Duke, once presented a fascinating "archaeological excavation" into the history of bubble sort, trying to find out why this rather unexceptional quadratic

performer was so popular. His efforts turned up little more than tradition, which I guess makes bubble sort mostly a hazing ritual for computer scientists. After this assignment, you can consider yourself a participant in this long and venerable tradition!

Your job as sort detective is to figure out which algorithm is implemented by each mystery sort. In addition to identifying the algorithm, your submission must include an explanation of the evidence you used to make your decision.

To a certain extent, you're on your own in terms of figuring out how to run these experiments. However it should be clear that timing provides a very important clue. The difference in an algorithm that runs in $O(N^2)$ time versus $O(N \log N)$ must certainly be reflected in a significant difference in how the running time changes as the size of the input increases. Moreover, some of these algorithms work more or less quickly depending on the arrangement of elements in the starting configuration, so carefully preparing different inputs and comparing the resulting performance could provide useful information. Our mystery sorting routines also include a feature that allows you to specify a maximum amount of time for the sort to run. By choosing an appropriate value, you can stop the sort mid-stream, and then use the debugger (or print statements) to examine the partially sorted data and try to determine the pattern with which the elements are being rearranged. Through a combination of these types of experiments, you should be able to figure out which algorithm is which and assert your conclusions with confidence.

Timing an operation

Gauging performance by measuring elapsed time (e.g. using your watch) is rather imprecise and can be affected by other concurrent tasks on your computer. A better way to measure elapsed system time for programs is to use the standard **`clock`** function, which is exported by the standard **`ctime`** interface. The **`clock`** function takes no arguments and returns the amount of time the processing unit of the computer has used in the execution of the program. The unit of measurement and even the type used to store the result of **`clock`** differ depending on the type of machine, but you can always convert the system-dependent clock units into seconds by using the following expression:

```
double(clock()) / CLOCKS_PER_SEC
```

If you record the starting and finishing times in the variables **`start`** and **`finish`**, you can use the following code to compute the time required by a calculation:

```
#include <ctime>

int main()
{
    double start = double(clock()) / CLOCKS_PER_SEC;
    . . . Perform some calculation . . .
    double finish = double(clock()) / CLOCKS_PER_SEC;
    double elapsed = finish - start;
}
```

Unfortunately, calculating the time requirements for a program that runs quickly requires some subtlety because there is no guarantee that the system clock unit is precise enough to measure the elapsed time. For example, if you used this strategy to time the process of sorting 10 integers, the odds are good that the time value of **`elapsed`** at the end of the code fragment would be 0. The reason is that the processing unit on most machines can execute many instructions in the space of a single clock tick—almost certainly enough to get the entire sorting process done for 10 elements. Because the system's internal clock may not tick in the interim, the values recorded for **`start`** and **`finish`** are likely to be the same.

One way to get around this problem is to repeat the calculation many times between the two calls to the `clock` function. For example, if you want to determine how long it takes to sort 10 numbers, you can perform the sort-10-numbers experiment 1000 times in a row and then divide the total elapsed time by 1000. This strategy gives you a timing measurement that is much more accurate. Part of your experimentation is figuring out how many times you need to perform the sort operation for different-sized inputs in order to get reasonably accurate results.

The write-up

You are to identify the mystery sorts and justify your conclusions. Your write-up should demonstrate that you understand the differences in the sorting algorithms and can identify how those differences are reflected when observing the algorithms at work. Please keep things low-key (i.e. no need for multi-color graphs with dozens of data points), however, you should write in clear and complete English sentences and include details of the experimental data that substantiates your claims. If your experiments are well chosen, this write-up can be quite succinct.

References

Astrachan, Owen. Bubble Sort: An Archaeological Algorithmic Analysis. *Technical Symposium on Computer Science Education,* 2003.

Part B — Write your own generic sort

In lecture and in the reader, the sort functions operate on vectors of integers. In practice, we also need to sort other types of data. Writing a new sorting routine for each type is clearly not desirable. C++ templates allow you to write a single polymorphic function capable of handling different data types. Such a template would take a comparison function as an argument, which allows the client to provide a suitable method of comparison or use the built-in relational operators if appropriate.

For this part, you are to write a fully generic sorting function. It should take two parameters, a vecto of elements and an optional function pointer to compare elements. If the client doesn't supply a comparison function, the elements should be compared using the built-in relational operators via our standard comparison function. The prototype for this function will thus be

```
template <typename Type>
    void Sort(Vector<Type> &v, int (cmpFn)(Type, Type) = OperatorCmp)
```

(An aside about quirky C++ syntax: the default argument, i.e. the `= expr` part, should not be repeated on the definition for a function that has a separate prototype. If there is no separate prototype, then the definition *does* include the default argument.)

Here's the twist: we want you to implement a different algorithm than the ones we've already discussed. There are many other sorting algorithms and rather than deciding for you which one to implement, we are giving you the latitude to do a little research and choose one that interests you. Living in the wondrous Information Age, you have many great resources you can draw upon to further explore the wide world of sorting. Here are a dozen names to start your investigation:

Bingo sort
Bogo sort (ok, this one is super-dumb, but has a cute name :-)
Comb sort
Gnome sort
Heap sort
Introsort
Library sort (relative newcomer, published in 2004)
Fancy Mergesort (variants include in-place, k-way, bottom-up, natural, etc.)
Shaker sort (usually means bi-directional Bubble, but also sometimes bi-directional Selection)
Shell sort (or variants Brick sort and Shake sort)

Strand sort (could be implemented using our container classes to manage the strands)
Fancy Quicksort (i.e. improvements in pivot choice/partitioning strategy/hybrid crossover/etc.)
Or another of your own choosing or something you designed yourself… surprise us!

Here are a few useful web sites on algorithms that might help you get started:

Wikipedia http://en.wikipedia.org/wiki/Sort_algorithms
NIST http://www.nist.gov/dads/HTML/sort.html
Google Directory
http://directory.google.com/Top/Computers/Algorithms/Sorting_and_Searching/
Open Source Software Educational Society
http://www.softpanorama.org/Algorithms/sorting.shtml

On the web and in books, you will find descriptions of algorithms, animations, pseudo-code, and even implemented code in all variety of programming languages. You can make use of any of these, as long as you properly cite your references. However, we strongly recommend implementing the algorithm yourself (starting from pseudocode or a conceptual description) rather than copying some existing code line-for-line. A word of caution: **don't believe everything you read**! The web isn't known for its fact-checking and even textbooks and published articles (and Stanford lectures :-) have been known to have errors in their code. As the CS legend Don Knuth says "An algorithm must be seen to be believed". Implement and test carefully!

We recommend that you first implement your sort to operate on integers. That is an easier context in which to get the algorithm written and debugged. Once that's done, go back and make the necessary modifications to transform it into a generic template version that uses a comparison function. Be sure to test the generic version on several different data types using the default and client-supplied callback functions, so you are confident it can handle anything you throw at it.

Lastly, you are to implement two comparison functions to practice writing callbacks. One is a string comparison function that can be passed to your sort function to sort a vector of strings in order of increasing string length, breaking ties by alphabetical ordering. The second is a comparison function that can be passed to your sort function to sort a Vector of Set<int> in order of increasing sum (i.e. a Set containing {1, 2, 3} is ordered before one containing {2, 8}).

Once you have your code finished, you will write a paragraph or two describing its features to us. What it is Big-O running time? How does it compare to other sorts within its Big-O class? Does it have any best/worst-case inputs? What is the mix of operations (compare/swap/function calls) it uses? Did you make any optimizations or improvements when implementing the algorithm? How complicated is the code? What do you see as the overall strengths and weaknesses of this algorithm?

Special note on the Honor Code
For this assignment, we are encouraging you to use outside sources. Some of the information you find may be more helpful than others, so you may feel unsure on what is fair to use. Here's our take on it: we expect you to do your own independent inquiry, to clearly cite what resources you used to complete your work, and to understand and be able to explain all of the code that you submit. We do think you will learn the most if you write your own implementation starting from an algorithm description or pseudo-code. If you do find pre-written code that you choose to adapt instead, you should clearly indicate that in your citation and carefully work through the code to ensure you thoroughly understand it. When you're done, what you submit should feel like "your" code, no matter what its origins.

Extensions
With midterms in the air, we figure you won't have many spare cycles, but we wouldn't want you to go away empty-handed if you come seeking an extra challenge. Here are a few ideas just in case:

- Optimize the sort you implemented. Try to knock our socks off with how fast you can make it! Take inspiration from Jon Bentley (hero to programmers worldwide) about optimizing Quicksort ("Engineering a sort function" http://portal.acm.org/citation.cfm?id=172710). Many standard C/C++ library sorting routines derive from Jon's version. Although not all of these techniques he discusses are applicable to all sorting algorithms, it's a fascinating investigation into how to pursue optimization.
- Writing a sort routine to sort a linked list. We have focused on sorting arrays/vectors, which both benefit and suffer from the arrangement of elements in contiguous memory. The dynamic pointed-based arrangement of a linked list makes for a different context in which to evaluate the tradeoffs among sorting algorithms. Many of the standard sorting algorithms can be adapted to operate on a linked list, but some are certainly better suited (i.e. easier to implement, result in better efficiency, and so on). Before you choose, think carefully about which of the algorithms are a good fit for the particular strengths and weaknesses of a singly-linked list.
- A *stable* sorting algorithm preserves the relative order of elements with equal values. For example, if a list of students is sorted by student ID number and then re-sorted by last name, those students with the same last name would remain in order of ID number if the sort were stable. Of the sort algorithms we have studied, which can be implemented to be stable? Can you devise an experiment that would determine if our mystery implementation is stable?
- Although an $O(N \log N)$ algorithm is certainly more efficient than an $O(N^2)$ for large inputs, the overhead of the more complex algorithm can dominate the cost for when the input is small, causing it to take more time than a simpler quadratic approach. The c*rossover point* is the size at which the more efficient algorithm starts to outperform. Find the crossover point between the $O(N \log N)$ and $O(N^2)$ algorithms from Part A. A *hybrid* algorithm is one that combines two or more algorithms to exploit the best features of each. The crossover point would be useful in writing a hybrid sort that delegates to several algorithms behind the scenes, depending on which operates most effectively on the input size.
- Doug McIlroy, former head of Bell Labs research, wrote this entertaining paper ("A Killer Adversary for Quicksort" http://www.cs.dartmouth.edu/~doug/mdmspe.pdf) about manipulating the input to force almost any Quicksort implementation into its degenerate behavior. Very interesting reading! The setup is in straight C code, so you have to deal with some **`void*`** goop, but try running the adversary against the qsort in your C environment and report how this mischievous code does relative to an ordinary input.

Accessing files http://see.stanford.edu/see/materials/icspacs106b/assignments.aspx
On the class web site, you'll find a starter folder containing these files:

mysterysort.lib/h	Interface/implementation for MysterySort functions

When working on Part A, be sure to add **`mysterysort.lib`** file to your project so that the linker will find the mystery functions.

Deliverables
You are to submit a text file for the written part and a C++ source file of your code. For Part A, identify each the mystery sorts and include the evidence for your determination. For Part B, introduce your chosen algorithm and describe its characteristics. The source file should contain your **`Sort`** template and the two comparison callback functions. Your source file can also contain any testing code you used. Hand in a printed version in lecture, and submit all files electronically via ftp. Submissions are due before the **beginning of lecture** on the due date.

"If 25% of the world's CPU cycles are used for sorting, I guess that makes it second only to screen-savers." — Paul Lu