Assignment 4: Sort Detective

Kudos to Julie Zelenski and Eric Roberts for the handout.

Sorting algorithms are about as fundamental to computer science as the computer itself. It has been claimed that computers spend about a quarter of their time sorting, so it makes sense that we spend some time understanding how that time is spent. There are many other algorithms (searching and order statistics, to just name two of them) that can be implemented much more efficiently on sorted data. And sorting is just plain neat, because there are many different techniques (divide-and-conquer, partitioning, etc.) that lead to novel algorithms, and those techniques often can be repurposed to other tasks. The large numbers of algorithms gives us lots to study, especially when considering the tradeoffs offered in terms of efficiency, operation mix, code complexity, best/worse case inputs, and so on. Your next task allows you to study a set of sorting algorithms to gain insight as to how they operate.

Due: Friday, November 2nd at 3:00 p.m.

Part A — Sort Detective

Back in our secondary school days, chemistry students were often given an unknown compound and asked to figure out what it was via qualitative analysis. For this assignment, your job is to take several mystery algorithms and figure out what they are by analyzing 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 sort elements into ascending order. What's different about them is that each is backed by a different algorithm. Specifically, the algorithms used are (in alphabetical order):

- bubble sort
- insertion sort
- mergesort
- quicksort
- selection sort

We talked about four of these in class, and they are covered in Chapter 10 of your reader (insertion sort appears as Exercise 10-2). Bubble sort—the only one we've not discussed—has an interesting story. It's a simple **O(n^2)** algorithm that works by repeatedly stepping through the elements to be sorted, comparing two neighboring items at a time and swapping them if they are out of order. The pass through the elements is repeated until nothing moves. The algorithm gets its name from the way elements bubble left and right 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 pretty weak as far as sorting algorithms go, but for some strange reason introductory courses like talking about it anyway. Owen Astrachan, a friend and colleague of mine at Duke, once presented a fascinating "archaeological excavation" of the bubble sort, trying to find out why this unspectacular approach to sorting is so popular. His research turned up little more than tradition, which I guess makes bubble sort mostly a hazing ritual for computer scientists.

Your job for the next week: figure out which algorithm backs each mystery sort. Your submission should simply state what algorithm is used behind each of **`mysterySort1`** through **`mysterySort5`**, and you must also support your work by explaining why you know what you know.

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 between an **O(n^2)** algorithm and **O(n log n)** one will certainly be evident for very large input sizes. Moreover, some of these algorithms work more quickly depending on the sequence's initial configuration, so carefully preparing different inputs and comparing the results will provide some good 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 midstream, 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 support your conclusions with confidence.

Timing an operation

Gauging performance by measuring elapsed time (e.g. using your watch) is pretty bad and can be impacted by other applications running 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 the system clock unit isn't 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 sort 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 job 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 an understanding of the sorting algorithms and should also identify how their differences are realized by your experiments. Please keep things low-key (i.e. no need for multi-color graphs with dozens of data points). But you should write in clear English sentences and include a reasonable amount of detail. If your experiments are well chosen, your write-up can still be pretty short.

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 numbers. 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.

For this part, you are to write a fully generic sorting function. It should take a reference to a **`Vector<Type>`** and it should just assume that **`Type`** plays well with **<** (though you shouldn't assume it plays well with **>**, **==**, **!=**, and so forth).

```
template <typename Type>
void sort(Vector<Type>& v)
```

Here's the twist: we want you to implement a different algorithm than the ones we've already discussed. There are many sorting algorithms out there, so rather than deciding which one you should implement, we are granting you the **authority** to do a little research and choose one that interests you. Living in the Information Age, you have many great resources you can draw from. Here are a several names to help you get starter:

- bingo sort
- comb sort
- gnome sort
- heap sort
- 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`
Open Directory Project
  `http://www.dmoz.org/Computers/Algorithms/Sorting_and_Searching/`
Open Source Software Educational Society
  `http://www.softpanorama.org/Algorithms/sorting.shtml`

As you search da Internets, you will come across algorithms, animations, pseudo-code, and even full implementations in a 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 pseudo-code or a high-level description) rather than copying some existing code line-for-line. A word of caution: **don't believe everything you read**! The web isn't exactly known for its fact checking and even textbooks and published articles (and Stanford lectures :-) have been known to have errors in them. As 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 **`Vector`**s of integers. Once that's done, go back and make the necessary modifications to transform it into a generic template version that relies on nothing more than **<** to compare elements. Be sure to test the generic version on several different data types known to support **`operator<`**, so you are confident it can handle anything you throw at it.

Once you've finished, cut and paste the code into your write-up and include a short paragraph describing it. What is the 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 mix of operations (compare/swap/function calls) does it use? Did you make any optimizations or improvements when implementing it? How complicated is it? What are its strengths and weaknesses?

Special Note: 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 be unsure what's fair and what's not fair. 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 think you'll learn more if you write your own implementation starting from a high-level description or from pseudo-code. If you do find fully operational code and choose to adapt it, you should clearly indicate that in your citation and carefully work through the code to ensure you understand it. When you're done, what you submit should feel like "your" code, no matter how much of it was already written for you.