# CLASSROOM VIGNETTES

Henry M. Walker

# Sorting Algorithms: When the Internet Gives You Lemons, Organize a Course Festival

**YEARS AGO**, when I was writing my first books, my series editor, Gerald Weinberg, put forward the principle that any example for class or a textbook should be valid on its own merits. Of course material introduced early may be refined, but students should not learn a technique initially, then discover the approach was wrong. With so much for students to learn, students should not be devoting time to examples that don't work or are not valid in any context. Students should not have to unlearn an early example and then learn something to replace it. Rather, every example should be valid as presented—at least in some context within current understandings of the discipline. To illustrate, an insertion sort is a fine choice for the first example of a simple sorting algorithm: it is easily understood and is extremely efficient when the data set is almost ordered. Of course, much better (but more complicated) algorithms are available for random data or data in descending order—but the insertion sort is an algorithm of choice in at least some situations. (In contrast, the bubble sort is never the algorithm of choice in any context, so should never be used or taught.)

Traditionally, sorting algorithms have been part of introductory courses for decades. Often, the pedagogy has involved some of the following elements:

- trace the algorithm when applied to one or more data sets;
- code the algorithm in whatever language is used in the course;
- time the algorithm on data sets of different types (e.g., ascending, random, descending) and different sizes (e.g., 40,000, 80,000, or 160,000 elements).

For example, at the end of a unit on sorting, a common assignment might be to write code for several algorithms, run them on various data sets to determine execution time, and graph the run times. In principle, this approach gives practice working through the algorithms and provides experimental data to support algorithmic analysis of efficiency. For those interested, simple Java code to time an algorithm might have the form

```
// time and check insertion sort without swapping
start_time = System.currentTimeMillis();
call_algorithm_method (data_set);
end_time = System.currentTimeMillis();
System.out.print (end_time - start_time);
```

## Enter the Internet: Lemons

In recent years, however, numerous versions of standard sorting algorithms can easily be found on the web. Unfortunately, many of these examples violate Gerald Weinberg's principle—the code is simply awful. I would be embarrassed to have my name associated with many examples found on the internet, but apparently the writers of these solutions have other perspectives. Some common difficulties include:

- unnecessary memory allocation;
- excessive data movements;
- calls to procedures that perform very little work;
- use of overly-generalized mechanisms to perform specific, simple tasks.

Three examples may illustrate the widespread difficulties. The first two of these come from class notes available on the web, and the third comes from web-based material to support a published textbook. In an insertion sort, instead of taking an element out of an array, sliding elements up, and putting the element in its place, code on the web swaps the element down one position at a time. As shown in Table 1, column 2 shows time required for extraction, sliding up, and reinsertion of successive items in an insertion sort. Column

**TABLE 1: TIMINGS OF TWO IMPLEMENTATIONS OF AN INSERTION SORT**

| | Insertion Sort | | |
|---|---|---|---|
| Array Size | NoSwaps | WithSwaps | SwapFunc |
| Ascending Data | | | |
| 10000: | 0 | 1 | 1 |
| 20000: | 1 | 1 | 3 |
| 40000: | 0 | 0 | 0 |
| 80000: | 0 | 0 | 1 |
| 160000: | 1 | 0 | 0 |
| Random Data | | | |
| 10000: | 30 | 30 | 197 |
| 20000: | 116 | 155 | 779 |
| 40000: | 466 | 702 | 3125 |
| 80000: | 1863 | 2961 | 12490 |
| 160000: | 7418 | 12048 | 49801 |
| Descending Data | | | |
| 10000: | 59 | 72 | 389 |
| 20000: | 232 | 368 | 1557 |
| 40000: | 928 | 1552 | 6228 |
| 80000: | 3749 | 6271 | 24908 |
| 160000: | 14886 | 24749 | 99670 |

3 swaps elements down one position at a time, and column 4 implements the swap by separate procedure calls. Altogether, the swap-function version ran 7-8 times slower than a simply-coded insertion sort.

In a merge sort, each recursive step may involve creating subarrays, copying data from an original array to the subarrays, and then merging the subarrays back into the original array. In some cases, the merge may put data into a third array, and then the results may be copied back to the original array. In many cases, the online code may contain separate functions to compare values in sorting—but for integers in Java this requires additional autoboxing and adds calls for a simple "less than" comparison. (Generality may be wonderful in some circumstances, but it may come at a price.) From timings comparing two versions, the multiple-copy-with-comparator approach ran 3-5 times slower than a simply-coded merge sort.

In a radix sort of integers, using Java and an int array based on decimal digits, one approach on the web creates an array of 10 linked lists for each int digit. With autoboxing and autounboxing, new objects are created for every decimal digit of every data element. As a conservative estimate, for an initial int array of n elements, where the integers contain 8 decimal digits, memory allocation is required at least 16n times. Further, the Math.pow function is used to compute the power of 10 needed to extract each decimal digit from an integer. (Math.pow is wonderful for fractional exponents, but often inefficient for small positive integer powers that do not need to be recomputed at each step.) Timings indicated that the online code ran 10 times longer than a simply-coded radix sort.

## Organize a Course Gala

With such awful code easily available, many traditional coding assignments have limited usefulness. When asked to write a specific sorting algorithm, students can draw upon hundreds (thousands?) of sources. When asked to time and compare algorithms, bad implementations may indicate that some algorithms work relatively well, when a better implementation might highlight shortcomings.

An alternative approach is to utilize online examples as starting points to highlight algorithms, implementation inefficiencies, and timing issues. The basic idea is that one can begin with awful implementations and organize an assignment gala (or festive celebration), in which students consider how to turn misguided code into efficient and effective implementations. Here are several examples I have used in the recent semesters. In each case, initial code is given, possibly limiting students trying to find solutions on the Web—student work must be based on the code given.

- After highlighting common inefficiencies, give students one or two bad implementations and ask them to make improvements. Then students can time versions of the code on various data sets to determine what, if any, speed up has been achieved.
- Give students several implementations of the same sorting algorithm and ask them to compare and contrast. Based on these different versions, students might write a new version that builds on the strengths of the given versions, but avoids the weaknesses.
- Give students a list of potential inefficiencies in code, as well as several implementations of a sorting algorithm. Then ask the students to find which, if any, of those difficulties are present in several implementations
- Start with a bad implementation, and ask the students to time it on various data sets. Then the exercise could identify 3-6 improvements, and students could time the resulting code when each adjustment was made. In addition to the code, students might produce a table similar to Table 1.
- Give two or three versions of a merge sort (perhaps changing what happens in a merge when two values are equal), and ask students to analyze which version(s) are stable, which shows timings (in milliseconds) of several sorting algorithms using comparably-coded Java methods.
- Give students a specific sorting implementation and ask them to adjust it so that one version uses a comparator for determining order and another version compares elements directly (no comparator function parameter). Then ask students to time the two versions to determine the extent to which a comparator adds noticeable overhead.
- After reviewing several sorting algorithms, ask students how testing might be automated, so that a user will know that an implementation is working correctly.
- In the spirit of a gala or festival, ask students to examine implementations of a specified sorting algorithm from the web. Awards might be given to students who find the most efficient or the least efficient implementations.

All of these activities openly acknowledge that a simple Web search will generate numerous implementations of various algorithms. Rather than ignore these sources, these approaches ask students to focus on specific algorithms, analyze available code, make improvements, and time results. Not only may these activities help students learn about specific algorithms, but students also may gain insights on the quality or lack of quality found on the Internet.

Altogether, each of these activities allows students to examine examples at an early stage, by contrasting inefficient code with well-designed code. Further, students gain direct experience with qualities that separate well-constructed code from misguided code. In this context, even awful examples can provide insights—although not necessarily about the problems the examples might have been trying to address. **Ir**

**Acknowledgments**
Some of the ideas for the exercises mentioned in this column build upon approaches suggested by my colleague, Samuel Rebelsky. Thanks also to Margie Coahan for her suggestions on this column.

**Henry M. Walker**
Computer Science, Grinnell College
Noyce Science Center, 1116 Eighth Avenue, Grinnell, Iowa 50112 USA
*walker@cs.grinnell.edu*

**DOI:** 10.1145/2727125