Double Sorting: Testing Their Sorting Skills

Darrah P. Chavey
Beloit College; Dept. of Mathematics & Computer Science
700 College St.; Beloit, WI 53511
chavey@beloit.edu

ABSTRACT

You're teaching elementary sorting techniques, and you would like your students to do a programming assignment that tests their understanding of the ideas. But all of the code for elementary sorting techniques are in the textbook, easily found on the Web, etc. We suggest the use of two "Double Sorting" techniques whose solution is not standardly available, are fairly straightforward to code, and offer speed improvements over the "straight" sorts. Double Sorting, the idea of processing two chosen elements simultaneously, applies to both Insertion Sort and Selection Sort, with speedups of 33% and 25%, hence are good enough to justify coding, but not good enough to be in Web collections. Code for this can be written in as little as a dozen lines of C++/Java code, and is easily within the reach of introductory students who understand the basic algorithms. In addition, the ideas used for double sorting are natural first steps in understanding how the N^2 sorts can be improved towards the $N \log N$ sorts that they will study later. For more advanced students, these double sorts also generate good exercises in the analysis of algorithms.

Categories and Subject Descriptors

A.1 [**Introductory and Survey**]: CS1/CS2, Arrays, Sorting.

General Terms

Algorithms, Performance.

Keywords

Sorting, Insertion, Selection, Quicksort, CS1.

1. INTRODUCTION

When first teaching students about working with arrays, an excellent programming exercise would be to explain the Insertion Sort, or Selection Sort, algorithm to the students in class, and then require that they convert the algorithm into code on their own. It might even be possible to use such an assignment in a closed lab, where the lab instructor was careful to watch for students who grabbed the code off the web, or from the text, instead of writing it themselves. In most cases, though, we need to resign ourselves to the fact that this is not an effective programming assignment. At least as good, though, is to demonstrate the sorting algorithm *and* how to convert the algorithm into code, then demonstrate a modification of that algorithm, and finally to assign the coding of that modification as a programming exercise. This paper suggests two such simple modifications of these sorting algorithms: Double Insertion Sort, and Double Selection Sort.

SIGCSE'10, March 10–13, 2010, Milwaukee, Wisconsin, USA.

A good programming exercise should, ideally, give the students both programming practice, and help them understand significant concepts. We will argue that each of the double sorts introduces students to concepts that are important in other contexts. Because the double sorting code is substantially different in the cases where the array size is odd or even, these two sorting algorithms provide examples of four programming assignments that are roughly equivalent in difficulty, but almost completely independent of each other, e.g. assign "Double Insertion Sort, odd size arrays" one semester, etc. Consequently, they could provide four semesters of "the same, but different" assignments before recycling a previous assignment, reducing potential plagiarism issues.

Double Insertion Sort does not currently (as of Dec., 2009) have solutions on the Web, and the two available solutions for Double Selection Sort are either less than helpful, or are probably identifiable as copied. Since these sorts do not offer enough substantial improvements over other sorting algorithms, we would not expect that solution code would appear on the Web – assuming that readers of this article do not post their own program solutions on publicly accessible pages (and we ask that they avoid doing this). Since the double sorts do offer substantial speed improvements (33% and 25%) over the standard "single" sorts, we can motivate such an assignment as an improvement on the code that they probably have in their textbooks. In addition, since both QuickSort and MergeSort, done properly, will use recursion on large array sizes, but will have a threshold at which they will switch to one of the N^2 sorts, one can argue that the use of a double sort for these small array partitions will provide a (modest) improvement to these two critical sorting algorithms.

Finally, while these sorting algorithms are aimed primarily at CS 1–CS 2 courses, the analysis of these algorithms can provide useful exercises for a more advanced class doing analysis of algorithms, and we will discuss this later.

2. RELATED WORK

Papers such as [4, 5] have worked to address related problems that we have when trying to construct reasonable assignments involving the $N \log N$ sorts. These assignments suffer from the same difficulty of students Googling their results instead of writing their own, and these authors have interesting suggestions on ways to have students focus on the ideas and concepts of these algorithms, while still having to develop their own code.

In [3], Juhlstrom suggest an interesting approach to getting students to understand these N^2 sorts. In the only paper in the ACM Digital Library with the word "Whimsical" in the title, he suggests looking at N^3 sorts! In particular, he limits candidate algorithms to those which "compare and move values [and where] each operation must plausibly advance the sorting process." By constructing algorithms that create N^3 versions of Bubble Sort, Selection Sort, and Insertion Sort, one can give students

assignments whose solutions they surely will not find with Google, but which require an understanding of the algorithm. It seems likely that class discussion of the differences between the "textbook code" and their assigned algorithm would do much to illuminate both the algorithms and the need to plan, and think, before coding.

The Double Insertion Sort problem, as an "Assignment to Use Next Week," was included as part of the tutorial [1], although individual assignments were not published in that journal, and the assignment distributed did not include most of the additional material of this paper. The Double Selection Sort problem is a somewhat well-known problem, as will be discussed in section 6.

3. DOUBLE INSERTION SORT

One problem with Insertion Sort is that we always insert a new element from the top of the array, even if will end up much closer to the bottom. Wouldn't it be faster to insert large elements from the top and small elements from the bottom? An implementation of this idea can be built by sorting an array from the center out. Assume, for simplicity, that the array we're sorting has an even number of elements. Sort the center two elements. Take the two elements on the "fringe" of the sorted portion of the array (the ones immediately before and after the sorted portion); insert the larger element of this pair from the top of the array, moving larger elements up one by one as we would do in regular Insertion Sort, until we find the right location for this new element. Finally, insert the smaller one of this pair from the bottom of the array, moving smaller elements down one by one.

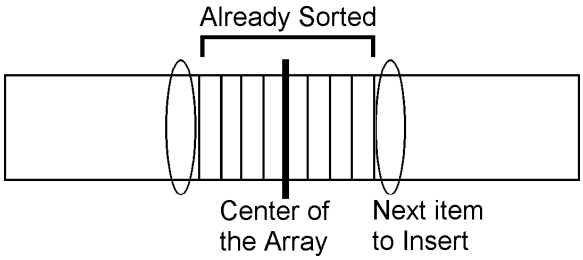

Figure 1. Double inserting the fifth pair of elements.

Pictorially, imagine that we are at the point where 4 pairs of items have been double insertion sorted. Then the middle 8 items of the array are fully sorted, with the next pair drawn as ovals in figure 1. We first interchange the two elements of this pair if necessary. We insert the top item of this pair, exactly as if inserting the 9-th element in Insertion Sort (but with a different "bottom element" then in Insertion Sort). When that is done, we insertion sort the bottom element, moving it up, essentially reversing the code for Insertion Sort. We should note that because of the initial sort of the two "outside" elements, we are guaranteed to have sentinels at both ends of our region of activity, so we are guaranteed never to fall off the array, or to leave the sorted portion of the array. This is both a good teaching example of sentinels, and one way in which the code is somewhat nicer than for straight insertion sort.

The coding requirements for this assignment are not particularly difficult, although it can be simplified by assuming that the size of the array is even (or that it is odd). The insertion sort from one end of the array should follow directly from, say, code in your text or your class. The student needs to infer the insertion sort in the other direction, then spend some time thinking about where the center is and how to move out from there. (With many options for "off by 1" errors.) A useful extra credit option is to ask students to run Double Insertion Sort against Insertion Sort to empirically compare their performance. While we normally recommend assigning only the "odd size" or "even size" array problem for students to program, an interesting paired project (which we have not tried) is to have two students work together, one primarily responsible for the odd case and the other primarily for the even case.

This algorithm generally succeeds in the goal that the larger element tends to be inserted from the large side of the array (e.g., the right side), and the smaller element from the small side (the left). Thus the initial swap of elements, if necessary, tends to move *both* items a large step towards their final destination. This tends to noticeably increase the speed of the algorithm as compared to Insertion Sort. This algorithm can thus be viewed as a conceptual step towards some of the faster exchange sorting algorithms, such as Shellsort and Quicksort. These faster algorithms also begin with the premise that we should often exchange elements that are far apart in the array, so that both are moved a large step closer to their final positions. The principal that speeds up Double Insertion Sort leads naturally to this more general sorting principal.

It is easy, even at this level, for students to understand the improvement in the worst-case time. If we were to insert items k and k+1 using straight Insertion Sort, the worst case number of comparisons will be $2k-1$. However, in Double Insertion Sort the smaller element can never move past the final location of the larger one. For example, if the larger item of a pair must be moved 90% of the way down the currently sorted array, then we are guaranteed that the smaller item will be moved no more than 10% of the way up. With the original comparison, inserting two items thus requires at most k+1 comparisons, for a 50% worst-case speed-up. Overall, Insertion Sort has a worst-case complexity of $N^2/2 + O(N)$, while Double Insertion Sort has a worst-case complexity of $N^2/4 + O(N)$. We might note also that the worst-case performance for Insertion Sort occurs with the fairly "natural" example of a reverse sorted list, while worst-case performance for Double Insertion Sort does not seem to occur for any such "natural" example. In particular, it runs in linear time on a reverse sorted list.

Although it is more complicated (see section 5), analysis shows that Insertion Sort has an average-case complexity, for both assignments and comparisons, of $N^2/4 + O(N)$. On the other hand, Double Insertion Sort can be shown to have an average case complexity (either measure) of $N^2/6 + O(N)$. Thus the average-case is 33% faster, and the worst-case is 50% faster, than Insertion Sort.

Sample Java code for this problem is shown below. This code works for odd size arrays; code for even size arrays is very similar, and can fairly easily be combined with this into a single method. There are a variety of ways to re-write the six lines of the "if" statement. For example, one could replace this by two conditional assignments, giving an 11 line solution to the problem. This "if" statement involves a certain optimization that a student solution would not likely include: Double insertion requires sorting a pair of elements; Insertion sort pulls out the element before moving others in the array; the code here combines those into a single step. This optimization affects the $O(N)$ constant in a full analysis, but does not change the constant of the N^2 term.

```
public void doubleInsertionOddSize( long toSort[] ) {
    long smallerItem, largerItem;
    int arraySize = toSort.length;
    int toMove, middle = arraySize / 2;

    for (int fringe = 0; fringe < arraySize/2; fringe++ ) {

        // Put the pair being inserted into "smallerItem" and "largerItem"
        if (toSort[ middle - fringe - 1 ] > toSort[ middle + fringe + 1 ]) {
            smallerItem = toSort[ middle + fringe + 1 ];
            largerItem  = toSort[ middle - fringe - 1 ];
        } else {
            largerItem  = toSort[ middle + fringe + 1 ];
            smallerItem = toSort[ middle - fringe - 1 ];
        }

        // Now insert the larger item from the top
        for (toMove = middle+fringe; toSort[toMove] > largerItem; toMove--)
            toSort[ toMove+1 ] = toSort[ toMove ];
        toSort[ toMove+1 ] = largerItem;

        // And insert the smaller item from the bottom
        for (toMove = middle-fringe; (toSort[toMove] < smallerItem); toMove++)
            toSort[ toMove-1 ] = toSort[ toMove ];
        toSort[ toMove-1 ] = smallerItem;
}
```

Figure 2. Java Code for Double Insertion Sort on Odd-Length Arrays

4. DOUBLE INSERTION SORT & THE N log N SORTS

The author prefers to give students programming assignments that have at least some sense of "realistic value," as opposed to being too apparently "make work" assignments. Of course one advantage of double sorting as an assignment is that it *isn't* important enough to appear in places the students will easily find. Better students will realize, however, that by itself double insertion sort is not particularly important: The advantage of any N^2 sort is in the simplicity of the code; if we were willing to put up with more complicated code, we would probably go with an N log N sort. Even at the CS 1/CS 2 level, we have some students who know this. What most of them don't know, though, is that sorts such as Quicksort and Mergesort have break-even thresholds, below which the N^2 sorts are superior, and that better implementations of N log N sorts will use the recursive techniques on large array sizes, but switch to an N^2 sort for small array partitions. For example, Quicksort is often used down to a threshold of size 15 (or some number close to that), then Insertion Sort is used for partitions of that size or less. Based on the previous discussion, one would expect that if we replaced Insertion Sort with Double Insertion Sort, we would get some modest amount of speed-up. That does, in fact, happen. We ran 100 trials on lists of size 10,000; 100,000; and 1,000,000, using a threshold of 15, and running Insertion Sort in one case and Double Insertion Sort in another case. (We used identical arrays for these two cases to ensure comparability.) If we scale the number of assignments and comparisons executed by Quicksort + Insertion sort to 100%, the results for Quicksort + Double Insertion Sort are shown in table 1:

Table 1. Quicksort Speedup Using Double Insertion Sort

N	10,000	100,000	1,000,000
Compares	97.5%	98.1%	98.4%
Assigns	89.1%	91.0%	92.3%

Of course as N gets larger, we expect these numbers to increase, because the N log N component must (asymptotically) overwhelm the N^2 component for these small partitions. While these improvements are quite modest, given the importance of Quicksort and the difficulty in improving it, it is still somewhat impressive that it is possible to get an 8-9% speed-up in it using a introductory student level program.

5. AVERAGE-CASE ANALYSIS OF DOUBLE INSERTION SORT

The average case analysis of Double Insertion Sort is beyond what we would normally expect to present to a CS1/CS2 class. We have, however, used it in an upper-level algorithms class, and it might be appropriate for a Discrete Math course in the CS curriculum as well, so we outline the analysis here, focusing on the number of comparisons only.

Inserting the larger number from the top is symmetrical with inserting the smaller number from the bottom, so we need only consider the cost of the former operation, and then double it. We cannot calculate the insertion cost as we would with Insertion Sort, because the distribution of the final location of this element is not evenly distributed — it is much more likely to be on the right side of the array than on the left. Instead, we begin with the state where we have just completed the sorting of elements k–1 and k, so that we now have a sub-array of k sorted items, which we number from 1 to k. We now ask how many comparisons did it take us to get the larger of the last two items into place? In particular, what is the probability that the top item moved i–1 steps down this part of the array, and landed in position k–i? This probability is given by the number of ways of choosing two

elements from this array (i.e. the last pair inserted) with the larger item being in position *k*–*i*, divided by the number of ways of choosing any two elements from this array. Since the larger one must have been in position *k*–*i*, and we have *k*–*i*–1 options for where the smaller element was (see figure 3), this probability is $(k–i–1) / \binom{k}{2}$.

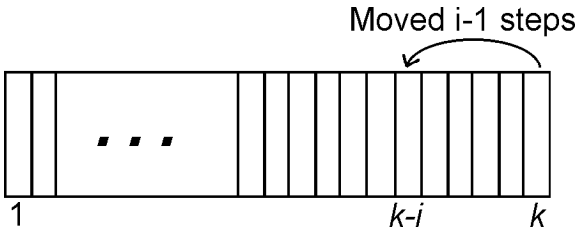

Figure 3. The cost of inserting an item from the top.

It will have taken us *i* comparisons for this insertion, so summing up the expected value of all such comparisons tells us that the average number of comparisons will be $\sum_{i=1}^{k-1} i \cdot (k-i-1) / \binom{k}{2}$. Standard summation techniques reduce this to $k/3$. Doubling this (for inserting the lower item) and summing for every other value of *k* (since we add two items at a time) can be written as: $\sum_{k=1}^{N-1} (k/3 + \mathrm{O}(1))$ which gives $N(N–1)/6 + \mathrm{O}(N)$.

6. DOUBLE SELECTION SORT

Unlike Double Insertion Sort, Double Selection Sort has been described before, and can be found in two sources on the Web (and, as best as I can find, only those two sources). One of these sources, [6], has it available, but somewhat difficult to locate, written in Visual Basic, and probably offers more translation problems than writing it in Java or C++ in the first place. The second source, [2], has it in C++, but probably in a style from which plagiarism would be easily detectable. Thus while this as a program assignment is slightly more problematic than Double Insertion Sort, it is still probably usable, and introduces another important algorithm that we often show our students at this level.

Selection Sort finds the minimum, or maximum, of *N* elements in *N*–1 comparisons. The MaxMin algorithm lets you find the minimum *and* maximum of *N* elements in 3*N*/2 comparisons (keep the current min and max; take 2 more elements, compare them against each other; then compare the two largest and the two smallest). To convert this to a sorting algorithm, we reverse the "build direction" of the Double Insertion Sort algorithm by sorting our array from the outside in. Find the Max & Min in array elements [0 … *N*–1], place them in the two outside positions, then iterate this algorithm with the remaining array elements [1 … *N*–2]. While Selection Sort requires $N^2/2 + \mathrm{O}(N)$ comparisons, Double Selection Sort will require $3N^2/8 + \mathrm{O}(N)$ comparisons, for a 25% speedup on this measure. Unfortunately, Selection Sort is used primarily when comparisons are much cheaper than assignments, since it uses only $N + \mathrm{O}(1)$ assignments. Consequently, this improvement does not have as much of the "real-life value" that Double Insertion Sort has. Nevertheless, it could still be a useful assignment, or for a class that's seen Double Insertion Sort, it would make a useful comparison and an opportunity to demonstrate the MaxMin algorithm.

7. REFERENCES

[1] Cutter, P. and Schultz, K. 2004. Assignments to Use Next Week: Tutorial Presentation. J. of Computing Sciences in Colleges 20(1), Oct. 2004, p. 114.

[2] Gordon, S. 2007. Benchmark of Sorting Algorithms on Arrays. http://pr.stewartsplace.org.uk/d/sortbench.d

[3] Julstrom, J., 1992. Slow Sorting: A Whimsical Inquiry. ACM SIGCSE Bulletin 24(3), Sept. 1992, pp. 11-13.

[4] Merritt, S., and Nauch, C., 1990. Inventing A New Sorting Algorithm: A Case Study. ACM SIGCSE Bulletin 22(1), Feb. 1990, pp. 181-185.

[5] Rolfe, T., 2005. List processing: sort again, naturally. ACM SIGCSE Bulletin 37(2), June 2005, pp. 46-48.

[6] VBExplorer, 2004. Sorting Viewer – Sorting Methods Discussed. http://www.vbexplorer.com/VBExplorer/vb_feature/august2000/sv4.asp