From Animation to Analysis in Introductory Computer Science

Richard Rasala, Viera K. Proulx, Harriet J. Fell
College of Computer Science
Northeastern University, Boston, MA 02115
E-mail: rasala@ccs.neu.edu, vkp@ccs.neu.edu, fell@ccs.neu.edu

(SIGCSE'94.

Introduction

At educational computer conferences and exhibits, one is overwhelmed by the extensive use of computers as learning tools in almost any subject. However, the one subject which stands out for its limited use of computers is computer science. There is a tendency in computer science education to focus on what goes on in the mind: design and analysis of algorithms, development of data structures, and use of abstraction and modularization in programming. The computer is seen only as the object of study but not as a tool in the educational process. The danger in such an approach to teaching computer science is that the student may become facile in the use of formalism and languages but may fail to deeply grasp what is really going on.

One important way in which computers have been used in computer science to assist the educational process is by the development of algorithm animations. Algorithm animations have been created by a number of computer science educators including ourselves [ ]. The primary goals of these algorithm animations have been:

- To permit students to observe an algorithmic animation rapidly in order to grasp the overall gestalt of the algorithm
- To permit students to examine an algorithmic process in a step-by-step fashion in order to understand the details of the algorithm

In our curriculum work, animations have been used in three additional ways. We use animation lab exercises to teach basic programming structures such as loops, branches, procedures, and arrays. We use more sophisticated algorithm animations as a rich collection of source code to illustrate abstraction and program design principles. Finally, we expect students at the conclusion of the freshman year to be able to create animations of their own for a variety of algorithms.

Recently, we have discovered how to integrate algorithm animations with the more traditional concerns of analysis of algorithms. Our process requires four steps:

- Students execute algorithm animations to understand how the algorithms work and to obtain a rough idea of how fast or slow each algorithm runs.
- Students acquire detailed performance data on all algorithms by running a separate "Time Trials" program which permits them to design and refine experiments with the algorithms and to collect comparative results.
- Students import their performance data into a spreadsheet program for analysis and the creation of charts.
- Students prepare a report which discusses the performance data and the charts in relation to the animations and to the theoretical "Big O" estimates. Agreements and anomalies must be explained in the report.

Our students carried out this process for the classical sorting algorithms. Their response to the assignment was the most enthusiastic of any exercise we gave in the freshman year. After the project, students felt that they really understood the algorithms deeply. We believe that the key to this success was the integration of animation, performance data, theory, and high

quality spreadsheet tools for analysis. Students were given the chance to act as scientists: to discover and confirm for themselves rather than to be told what is the right answer.

In this article, we will contrast the traditional approach to the explanation and analysis of algorithms with our methods which integrate animation, data analysis, and software tools into the process. We will also argue that this methodology has implications for other areas of computer science education.

Traditional Algorithm Analysis

Traditional courses on Algorithms and Data Structures spend significant amounts of time on analyzing different algorithms and explaining the "Big O" notation. With each new algorithm, students count the number of steps and do the prescribed formal manipulations until they arrive at the right result. Discussion points are made about average and worst cases and students dutifully learn the assigned tricks. Nevertheless, despite much talk about n^2 versus $n \cdot log(n)$, students often do not know *how much worse* a BubbleSort of 10000 objects is than a QuickSort. The students lack real experience with the algorithms and therefore have only a shallow sense of the meaning of the analysis.

The problems with the traditional approach to analysis of algorithms in the freshman computer science curriculum can be divided into two major categories: educational weaknesses in the traditional mathematical approach and experiential weaknesses due to the fact that the students have not worked with the algorithms for sufficient time and with adequate depth.

The educational problems related to the traditional mathematical approach may be summarized as follows:

- Each algorithm must be examined separately with attention to the particular details of its implementation. No analytical method handles all algorithms for a given task at once. Some simple algorithms are already too complex for complete analysis.
- The combinatorial mathematics used is complicated. Frequently, students must do discrete mathematics in computer science courses which is more messy and requires more special tricks than the mathematics taught in calculus courses being taken concurrently. Students do not feel confident that they can do the mathematics on their own.
- The simplifying assumption that all elementary operations take approximately the same amount of time may not always be justified.
- The results of an analysis are usually stated as formulas: "Algorithm X is O(…)". Students do not have a strong sense for the impact of such formulas in real life. Algebraic formulas are treated by students as collections of symbols to be memorized but not deeply understood.
- It is difficult to make detailed comparison of algorithms for the same task with the same "Big O" performance.
- Experimental results are undervalued. There is no detailed examination of the actual time differences in performance as algorithms change or data sizes vary.

The experiential weaknesses of the traditional approach raise more fundamental questions of what is the proper paradigm for the study of algorithms. Students who approach algorithms from a purely mathematical viewpoint often do not have sufficient experience with the algorithms to know their behavior and performance well. They may know that one algorithm is "fast" and another is "slow" but they do not know how long either algorithm takes on real data nor whether the difference matters in practice.

In contrast, an approach to algorithm analysis based on the methodology of science and engineering would require that students first acquire experimental data by actually testing algorithms and then make an assessment of that data. Such an approach should either verify or contradict estimates of performance made by theoretical analysis or by heuristic guesses. The resolution of contradictions between what happens and what was expected should be an important component of the learning process. Students must determine if the contradictions arise from mistakes in the analysis, bugs in the implementation, or subtle factors in the hardware and operating system environment.

The use of experimental methods of data analysis is actually a more widely applicable skill for students to learn than purely theoretical analysis. There are many problems and systems in computer science which can only be studied by acquiring benchmark data and then interpreting that data as the basis for further work. In the sciences, engineering, and social sciences, the primary method for finding out about things is to make hypotheses, design experiments, and acquire data that will confirm, modify, or deny the hypotheses. Theory is but one ingredient in a complete analysis. It is important for students to learn how to intelligently use the tools of data analysis: spreadsheets, statistical packages, visualization software, and simulation tools.

An Experiential Approach to Algorithm Analysis

We will now illustrate our integrated approach to algorithm analysis using sorting algorithms as the case study. Students generally work on this project about midway through the Spring quarter of the freshman year. They have already been introduced to simple sorting algorithms in an earlier course and have recently learned about QuickSort and HeapSort.

Students begin the study of sorting algorithms by running the animation program for sorting. They explore all of the algorithms and learn that the algorithms which are fastest *in animation* are SelectionSort, QuickSort, and MergeSort. HeapSort and ShellSort are about one-third as fast as these algorithms. InsertionSort and BubbleSort are much, much slower than the other algorithms. From the animations, students can already see why QuickSort is very fast and why InsertionSort and BubbleSort are so slow. An interesting anomaly is that SelectionSort is the fastest algorithm *in animation* although it is clearly an $O(n^2)$ algorithm from a theoretical standpoint. This anomaly is a consequence of the high performance cost of graphical swap operations in algorithm animation combined with the fact that SelectionSort happens to require few swaps. For students, this anomaly is a puzzle to be explored during the project.

When students are comfortable with how the algorithms work, they turn to a separate "Time Trials" program to study raw performance. The Time Trials program is designed to permit rapid and systematic accumulation of data about sorting performance. Students can control the following options:

- How many different trials will be performed for each algorithm.
- What size data sets will be used in each trial.
- How the data sets will be arranged: randomly, in order, or in inverted order.
- What collection of algorithms will be tested.

The choice of data set size is simplified in the following manner. If a student selects 10 trials with minimum data set size = 100 and maximum data set size = 1000, the program will make the 10 data sets have sizes 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000. This allows the student to obtain performance data for linearly increasing data set sizes. For the special case of the QuickSort algorithm, options are also provided to make a transition to InsertionSort or

SelectionSort when sorting small subranges of the data. Students learn that these variants run slightly faster than the standard QuickSort algorithm.

The Time Trials program executes the trials, prints the performance data to the screen, and asks if the student wishes to save the data to a file. If the student answers YES, the data is saved in tab-delimited text format for easy import into a spreadsheet program. The Time Trials program provides no analysis tools since it is expected that students will use a spreadsheet program to perform analysis and create charts. An important message to students is that they should learn to use high quality off-the-shelf software rather than reinvent-the-wheel. In Table 1, a sample of data is shown together with some subsequent analysis. Columns 1-2 were generated by the Time Trials program while Colums 3-5 were added in the spreadsheet. Note that: 1 tick = 1/60 of a second.

Data Size	10000			
Time	Ticks	Seconds	Time Ratio	
Bubble Sort	16625	277.08	536.29	Slowest
Double Bubble Sort	14597	243.28	470.87	
Insertion Sort	5911	98.52	190.68	
Selection Sort	6267	104.45	202.16	
Shell Sort	61	1.02	1.97	
Heap Sort	106	1.77	3.42	
Merge Sort	91	1.52	2.94	
Quick Sort #1	38	0.63	1.23	
Quick Sort #2	37	0.62	1.19	
Quick Sort #1 & Insertion (10)	31	0.52	1	Fastest
Quick Sort #2 & Insertion (10)	33	0.55	1.06	
Quick Sort #1 & Selection (10)	34	0.57	1.1	
Quick Sort #2 & Selection (10)	35	0.58	1.13	

Table 1: Time Trial Data With Analysis

The table shows that the slowest algorithm, BubbleSort, takes 4 minutes and 37 seconds to sort 10000 integers while the fastest sort, a variant of QuickSort, takes only slightly more than half a second for the same task. The performance ratio is 536 to 1. During the acquisition of this data, a student must wait impatiently for more than 12 minutes for BubbleSort and the other slow sort algorithms to finish. Then, suddenly, the fast sorts all finish in a few seconds combined. This experience drives home in an emphatic fashion the difference between an n^2 and an $n \cdot log(n)$ algorithm.

The table shows that SelectionSort is really not as good as the animations suggest and that the students must therefore explain the anomalous behavior in the animations. The table also shows that ShellSort is solidly in second place behind the variants of QuickSort.

The Time Trials program allows students to create much more elaborate data tables in which algorithms label the rows and the data sizes label the columns. Using such tables students can create graphs which illustrate the changes in performance as the data size increases.

The option to test sorts on data that is ordered or inverted allows students to examine some issues of best and worst case behavior. InsertionSort is extremely fast on ordered data. In contrast, one of the two QuickSort algorithms selects its pivot from the endpoint of the sorting range. This variant of QuickSort has terrible performance on ordered data.

When students have finished the data collection and spreadsheet analysis, they must make a report about what they have discovered. This effort forces them to explain in English how the design of the various algorithms relates to their performance. We provide a set of rather

open-ended questions which mark a starting point for the report but do not constrain how the student will develop the data and the analysis. Here is a sample of some of the questions:

> Experiment 1.
> Compare the different quicksort algorithms with varying data sizes, and determine which one is the best for a given size. Document your experimentation and your findings with appropriate charts. Try different subarray sizes for switching to selection or insertion sort and find the optimum.
>
> Experiment 2.
> Determine which of the algorithms are much worse than average, and which ones are much better, when the input data is sorted or inverted. Again, document your findings using charts.
>
> Experiment 3.
> Compare all (most) of the algorithms on small size arrays (100 to 200). Do the same for arrays of sizes in the range from 800 to 1000. Plot the two sets of results and observe the differences.
>
> Write a summary of your findings - describing the good and bad points about different algorithms, what you have learned, what was surprising, etc. Hand in your work typed, using word processor. You may make the charts an integral part of your document or just print them separately. This assignment will be graded for how professionally it is done, how much you tried to learn, and how well you designed your experiments. We have talked about the complexity of some of the algorithms - see if the experimental results correspond to our theoretical results.
>
> Do the best you can. The most important points we will be looking for are the following:
> Did you learn enough Excel to get meaningful charts.
> Did you run enough experiments to get some meaningful results?
> Did you learn anything about sorting algorithms from this lab?
> Make sure we see what you learned from this lab.

As can be seen, students were given the responsibility for deciding what data sets to use in the experiments and what data to collect. In addition, we intentionally did not provide careful instruction in using a spreadsheet. Students did have some brief experience with a spreadsheet earlier. They had to learn what they needed for this lab by experimentation and asking questions. There are a large number of application packages that users learn best by observing others and by asking questions. We want all of our students to feel that this can be done. Students who are timid about using a new application without prior formal instruction are at a great disadvantage in the technical world. By giving them encouragement to play around and learn by doing, we encourage them to become more assertive in their use of computers.

Student Results

The Sorting Lab was the most surprising among all that we tried last year. First of all, students spent much more time on the lab than we anticipated. We compensated for it in the grading, feeling the time was well spent. Students became very engaged, tried a much larger number of cases than we planned, figured out several different ways in which to represent the results in charts, and taught each other the different features of Excel. They were very serious about producing professionally written reports. When the reports were finally handed in (it took them about ten days to complete them) the results were beyond all expectations. To illustrate what the students learned, we include some quotes from the lab reports and some verbal comments made by students:

"I'll never use BubbleSort again!"

"But the HeapSort cannot be 200 times faster than the InsertionSort!"

"A surprising aspect of the lab was the fluctuation of the graph with small range data. With some thought, we soon determined that this phenomenon was caused by a lack of array size … Arrays of larger size produced data which was much more stable and dependable."

"One of the important aspects of the test of QuickSort is that when the switch size is small the combination algorithms perform best overall … QuickSort1 & InsertionSort with a switch size of 10 proved to be the best in these tests."

"Upon testing with sorted data, I found that the quickness of the algorithms changed drastically. BubbleSort was still the slowest, but DoubleBubbleSort became the fastest and InsertionSort became the second fastest. The three versions of QuickSort2 also changed drastically becoming three of the slowest algorithms."

"Anyway, here in this lab, I could learn Excel and Microsoft Word which I was afraid to use before even though I knew how to use other spreadsheets and word processors. This lab was pretty helpful for learning the usage of other software than we use for programming."

"An important thing to learn from this lab is that any task can probably be done by a computer in many different ways. Solving the problem is more than just getting a program to do its required task. Any programmer should be able to produce the required output. The good programmer analyzes the problem more thoroughly and finds the most efficient algorithm in terms of time and system resource use. A program may even have two different algorithms to solve the same problem, depending on the type of data."

To illustrate what the students accomplished, it is also informative to show an example of the kind of charts they prepared. See Figure 1.

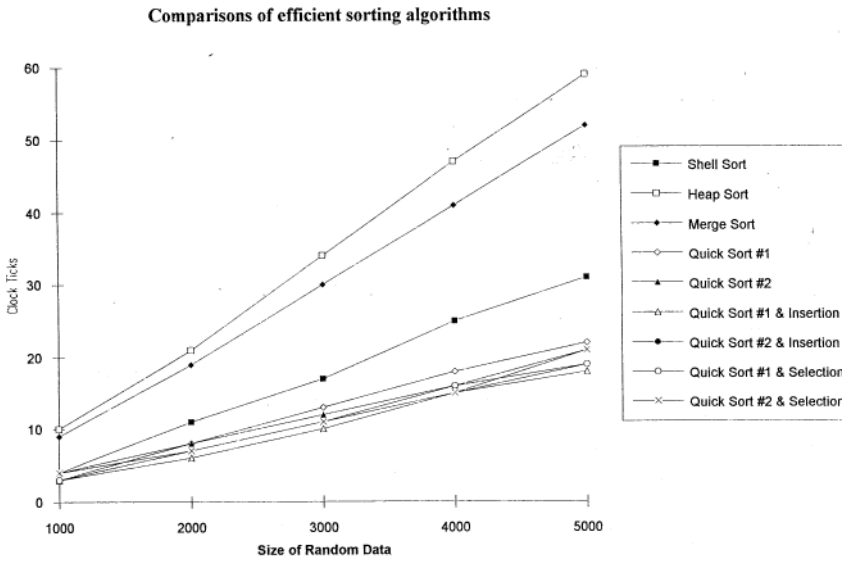

Figure 1: Student Analysis of QuickSort Variants

Observations

We want to gather together some observations about what the students did and learned in the Sorting Lab project and about why the structure of the project was really fruitful.

- The experimental approach to the analysis of the sorting algorithms permitted each algorithm to be treated in exactly the same way: Simply run the algorithm and record its performance.
- There was no need for the use of special mathematical tricks in the study of any of the algorithms.
- The comparison between different versions of the QuickSort algorithm could not have been done analytically. Students learned that it is often faster to run a series of performance tests and that such tests can provide more precise comparative information than a mathematical analysis.
- The comparison between small input data sets and large input data sets pointed out differences that students often do not see. It provided a basis for discussing in class the problem of determining when $O(n^2)$ is larger than $O(n \cdot log(n))$ and permitted students to see concretely that a simpler but slower algorithm may indeed be faster for smaller data sets. The success of the combination algorithm, QuickSort & InsertionSort, reinforced this message.
- The comment about HeapSort being 200 times faster than InsertionSort came from a smart student, the type that learns everything easily. He never stopped to think about what was the meaning behind all the analysis we had done. He could reproduce all of the results but he never understood them deeply. It took the visual jolt of the performance data and the graphs to make him see what we had been trying to say for several weeks.
- The students learned that it is quite easy to write a driver to perform a series of time comparisons for different algorithms. Although we supplied the Time Trials program, the students had access to the source code and could see that there was no particular magic in the code. The interface is clean, the timing of the algorithms is straightforward, and the recording of the results to a file is easy. They know that this is a method they can use in future work.

The discovery approach in this lab gave the responsibility for learning back to the students and they were delighted to have this opportunity. Each of the ingredients in the project added a new perspective towards an overall comprehension of the sorting algorithms. The algorithm animations gave students a concrete appreciation of how the algorithms work. The experience of collecting the performance data gave students a gut-level feeling for the relative performance of the various algorithms. The numerical data and the graphs created in Excel showed students that they could transform this gut-level understanding into something much more precise. Finally, the experience of writing the reports provided students with a time for synthesis and for a well-deserved sense of professional accomplishment.

Conclusions

The success of the Sorting Lab project suggests that such experiences should be much more common in the computer science curriculum. Students need the opportunity to do projects in which they can find out important things for themselves. They also need to know that they have the freedom to use whatever tools and methods are fruitful. To plan a project, they can use theory or mathematics or heuristics or past experience. If software is needed, they can write their own or, often more effectively, use off-the-shelf tools such as spreadsheets. Students also need to learn the necessity of adaption and modification since, as a project progresses, it is often necessary to change plans and ask new questions.

In the introductory curriculum, there are many topics besides sorting algorithms which lend themselves to this project approach: linked list manipulation, hashing methods, tree

balancing, graph traversal, and backtracking algorithms are all good examples. In the upper level courses, many similar possibilities can be easily found. The important theme is to permit students to get their minds, their eyes, and their hands focused on interesting problems where they have a good chance of learning something really exciting.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, Data Structures and Algorithms, Addison-Wesley, 1983.

[2] R. M. Baecker and D. Sherman, Sorting Out Sorting, 16mm color sound film, 30 minutes, 1981. (Shown at ACM SIGGRAPH '81 in Dallas, TX and excerpted in ACM SIGGRAPH Video Review No.7, 1983.)

[3] C. Brown, H. J. Fell, V. K. Proulx, R. Rasala, "Computer Science Learning Modules", 101 Success Stories of Information Technology in Higher Education: The Joe Wyatt Challenge, Judith V. Boettcher, ed. McGraw Hill, 1993, pp. 274 -280.

[4] C. Brown, H. J. Fell, V. K. Proulx, R. Rasala, "Instructional Frameworks: Toolkits and Abstractions in Introductory Computer Science", Proceedings of ACM Computer Science Conference, Indianapolis, IN, February 1993.

[5] C. Brown, H. J. Fell, V. K. Proulx, R. Rasala, "Using Visual Feedback and Model Programs in Introductory Computer Science", Journal of Computing in Higher Education, Fall 1992, Vol. 4(1), 3-26.

[6] C. Brown, H. J. Fell, V. K. Proulx, R. Rasala, "Programming by Example and Experimentation", Proceedings of the Fourth International Conference on Computers and Learning (4th ICCAL), Acadia University, Wolfville, Nova Scotia, June 1992.

[7] M. H. Brown, Perspectives on Algorithm Animation, Proc. ACM SIGCHI '88 Conf. on Human Factors in Computing Systems, April 1988, pp. 33-44.

[8] M. H. Brown and Robert Sedgewick, "Techniques for Algorithm Animation," IEEE Software, January 1985, Vol. 22, Vol. 2, No. 1, pp. 28-39.

[9] M. C. Linn and M. J. Clancy, The Case for Case Studies of Programming Problems, Communications of the ACM, March 1992, Vol. 35, No. 3, pp. 121-132.

[10] T. L. Naps, Algorithm Visualization in Computer Science Laboratories, SIGCSE Bulletin, February 1990, Vol. 22, No. 1, pp. 105-110.

[11] J. Robergé, Creating Programming Projects with Visual Impact, SIGCSE Bulletin, March 1992, Vol. 24, No. 1, pp. 230-234.

[12] J. T. Stasko, Tango: A Framework and System for Algorithm Animation, IEEE Computer, September 1990, Vol. 23, No. 9, pp. 27-39.

[13] A. B. Tucker, Fundamentals of Computing I: Logic, Problem Solving, Programs, & Computers, McGraw Hill, 1992.

[14] A. B. Tucker, et. al. (ed.), Computing Curricula 1991, Report of the ACM/IEEE-CS Joint Curriculum Task Force, ACM Press, 1991.