Putting the *Science* into Computer Science: Treating Introductory Computer Science as the Study of Algorithms

Justin Solomon
Computer Science Department
Stanford University
Stanford, California 94305 USA
Justin.Solomon@stanford.edu

Abstract

This paper describes why the study of algorithms should be a fundamental component of the standard introductory computer science (CS) curriculum. By shifting the focus of basic CS classes from *implementation* to *concept*, educators can greatly enhance student understanding and course relevance despite rapidly-changing paradigms, technologies, and programming languages. Teaching algorithms also encourages the development of other more generalized skills, including the scientific method, problem-solving, modeling, and technical communication.

Keywords: CS I, algorithms, introductory computer science, programming, science

1. Introduction

Despite its title, the "Introductory Computer Science" course taught in high schools and colleges nationwide is losing focus. As the fundamental computer science (CS) curriculum has evolved, emphasis has been placed on achieving a breadth of miscellaneous skills and knowledge necessary to succeed in computer *engineering* at the expense of developing a mastery of concepts basic to computer *science*, the most important of which are rooted in the scientific method and the algorithmic thought process. This approach has led to coursework characterized by rote memorization and other repetitive exercises designed to help students absorb the specific aspects of a particular programming language or paradigm, neglecting problem-solving skills or concepts fundamental to understanding the "science" of computers. Absent this focus, introductory CS courses miss an important opportunity to reinforce generalized technological or scientific literacy skills that would benefit students regardless of career choice or new developments in technology. One possible solution to these problems is to redirect the CS curriculum and make the study of algorithmic problem solving a fundamental component of the standard introductory computer science course.

2. Situation

Students in introductory CS courses often are overwhelmed by minute details that lack a unifying theme. Bogged down in the specifics of a particular language or paradigm, they fail to see the larger implications of each unit within the computer science curriculum as a whole. Because many courses have followed industry trends in choosing more and more advanced languages for introductory instruction, from BASIC to Pascal to C++ to Java, the details and specifics of writing and compiling a simple program have become more complicated than ever. With these new languages also come newer and more complex programming paradigms that students must learn in parallel with basic skills. For instance, some CS textbooks introduce Object-Oriented Programming (OOP) in the first few chapters without motivation or a notion of simpler data structures. Instruction covering these details of a programming environment or language may be useful for students entering the workforce immediately after taking an introductory course. If they enter the field in a more reasonable amount of time, however, the particular paradigms or constructs they learned are likely to have evolved while more basic skills remain the same.

This approach to computer science education has led to increased disinterest in computer science as a career or research field. In his 1996 study, Richard O'Lander of St. John's University found that the strongest factors affecting students' "apprehension about majoring in Computer Science" were their perception of their computing ability and enthusiasm towards computing [4]. Given that neither of these criteria was directly related to the quality or accessibility of computer science instruction available to the students, one might predict that ten years later, as

teaching methods for computer science were developed and the technology became available to make instruction more appealing, student interest and confidence might increase. Yet, in a 2006 survey, students still had little idea of what a computer science major entails. In fact, young men looking toward computer science listed an interest in developing games as their primary reason for their choice of major, while young women desired to use their computer skills in another field [2]. Neither of these reasons expressed any understanding of the vast majority of careers available in the computing field, nor did they reveal the value of an introductory course in presenting the basic skill set necessary for success in the area.

To understand both the breadth and depth of a field as new as computer science, students must achieve an understanding of the study of computers as a theoretical and scientific endeavor. While computer engineering certainly is a valuable career goal, it differs from computer science in that CS explores wholly unsolved *problems* while CE works within the context of an existing system for solutions to specific *needs*. Thus, a skill fundamental to computer science courses that may be overlooked in engineering curricula is the study of algorithms, or computational methods for solving problems. These methods can be expressed *independently* of a particular programming language, making them useful for a wide variety of applications. A student who learns a programming language or paradigm can survive until the industry switches to another language or paradigm; a student who has a thorough knowledge of the formation and expression of algorithms has a usable skill throughout shifts in specific technologies or languages.

As a fundamental component of an introductory CS course, the study of algorithms encourages generalized scientific skills to be used in any number of fields. Whereas the particulars of most programming languages can be counterintuitive, algorithms, like most other scientific or mathematical processes, tend to be straightforward (although not necessarily easy) to express and analyze. This allows students to understand the important distinction between *concept* and *implementation*, one that often eludes first-year programmers. The fundamental speed of a computer program lies not within the particular sequence of commands outlined in a program, but rather the algorithm it uses; a "Quick Sort" will run faster that a "Bubble Sort" in sorting long lists of numbers regardless of whether it was written in C++ or Java or the fact that the Quick Sort takes more lines.

Forming a computer science class based on the study of algorithms emphasizes the scientific and analytical aspects of the field above the nitty-gritty details that can obfuscate a greater meaning. Although it may take serious adjustments to a programming curriculum, this change will lead to greater student achievement and understanding and perhaps to student retention in computer science.

3. Introducing Algorithms

Shifting focus from implementation to algorithms will take a corresponding shift in curriculum structure. To this end, perhaps the most fundamental skill in the study of algorithms is the development of a link between the *intuitive* and the *formal*. For instance, consider the "convex hull" problem. This relatively simple geometric problem is difficult to express formally but appeals to the intuitive sense. Consider this standard description from Skiena's *The Algorithm Design Manual* [5]:

> *Input Description:* A set *S* of *n* points in *d*-dimensional space.
> *Problem description:* Find the smallest convex polygon containing all the points of *S*.

Clearly, this description would not be appealing to the typical CS I student, who may or may not be able to produce a concrete example of a convex hull given a set of points graphed on a plane. In the intuitive sense, however, the convex hull problem becomes clear, at least in two or three dimensions: Given a set of nails protruding from a board, what shape will be formed when a rubber band is stretched around the entire figure? As an example, consider the diagram of a convex hull, represented by a black line around a set of green points in Figure 1.

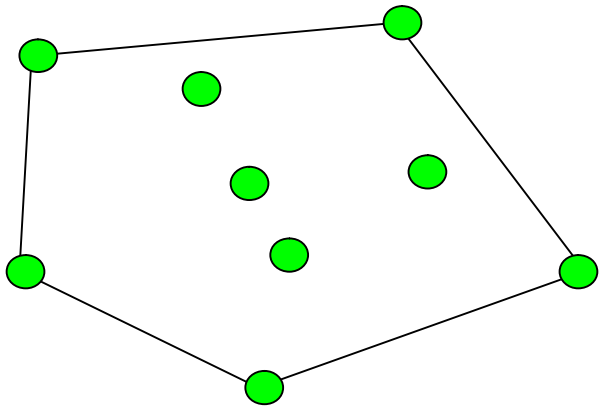

Figure 8: A convex hull

This description may be less formal or rigorous than that given by Skiena, but serves to communicate a concept that may be difficult to understand otherwise. The sign of a strong algorithms student is one who can communicate problems and their solutions fluently in either mode, formal or intuitive, and can translate between one and the other.[1]

This example also illuminates another key concept in teaching algorithms: the modes of *communication* of

[1] For those who are interested, one simple, albeit not optimally efficient, solution of the convex hull problem is the Gift Wrapping algorithm, which is described on any number of websites.

problems and their solutions. Problems may be best expressed using a mathematical formula, a verbal description, a diagram, a tactile demonstration, or any different mode. Knowing which to choose helps a student become a better communicator within the computing environment, which is typified in the professional world by collaboration between developers who often are separated by considerable physical distances.

Yet another skill gained from the study of algorithms is *modeling*. Students may be presented with a situation in need of automation or computerized assistance and asked how they would contribute. By separating key information and isolating a particular problem to solve, the student may be able effectively to reduce a seemingly complex set of relationships to a statement no more complicated than that given above for the convex hull problem. The ability to identify and suggest generalized solutions to particular algorithmic problems can help students find underlying simplicity in difficult problems or, as often happens in computer science, reduce a new problem to one that already has been solved. In fact, sometimes the process of modeling itself can bring about interesting observations by serendipity. For example, what started as a lousy model for rabbit reproduction became one of the most studied problems in mathematics: the Fibonacci sequence [6].

Once a problem has been expressed and modeled, the *scientific method* becomes an invaluable tool for the systematic examination of possible algorithmic solutions. For almost any problem, there exists any number of viable solutions; for this reason, the most intuitive or obvious solution may not necessarily be the best. Consider three possible methods for sorting numbers:

- The *Random Juggle*: repeatedly shuffle the list of numbers, each time checking if they are in order – O(?)
- The *Bubble Sort* or *Shell Sort*: systematically compare pairs of numbers selected from the list and swap them to be in order relative to each other – $O(n^2)$
- The *Heap Sort*: split the list of numbers into "piles" of one number each; repeatedly merge these piles, each time maintaining order, until all the numbers have merged into one list – $O(n \log n)$

Although each of these methods is indeed valid, they are listed above in order of efficiency. A student wishing to verify this fact could initiate a virtual experiment, implementing each and measuring its speed in terms of number swaps or time taken to sort a particular list. The results of such an experiment also can be checked using theoretical methods, such as the asymptotic runtime of each algorithm (listed above). This efficient approach to algorithm design is representative the scientific method, which is valuable for any number of fields.

There is any number of other skills that can be gleaned from the study of algorithms as part of a computer science course. More importantly, however, note the following observation:

None of the skills described above includes the words "computer" or "program."

Students who master the design and implementation of algorithms have gained an important skill regardless of their particular career or educational choices. Just as former students of geometry may or not remember the proof of the Pythagorean Theorem, an intuition for its implications can lead to better navigation of maps or roads. Students who learn an algorithmic approach to problem-solving are able to take difficult challenges, list the underlying problems they present, and systematically solve each methodically. These skills will be valuable in the workplace as long as the computer industry can stay in business.

4. Moving On

For the advanced student who finds himself or herself intrigued by the study of algorithms during or after taking the introductory computer science course, there is a wide variety of resources available to extend such an interest. At the college level, a more formalized approach to algorithms is often taught as a second-year course for computer science majors. Although this presents itself as a natural option for college students, high school students need not put off their studies until after graduation. Ideally, an introduction to algorithms could be taught as a post-APCS course for underclassmen who have already taken and passed the AP Computer Science exam. Such course can be taught comprehensively with a relatively low level of mathematical sophistication, making it approachable after only high school level math. This upper-level course may become a viable option for high schools in which students are taking college-level introductory courses earlier and earlier.

When there is not enough interest or teaching support for an entire algorithms course, motivated students still have several possible options to pursue. For instance, the USA Computing Olympiad (www.usaco.org) offers training pages for learning specific algorithms and programming techniques as well as monthly contests during the school year that range from basic to highly advanced levels. Top scorers from the advanced contests are invited an annual "Training Camp," where a team is selected to represent the United States in the International Olympiad in Informatics (IOI). Other opportunities for team or individual algorithm and computer science work include the American Computer Science League (www.acsl.org), the Internet Problem Solving Contest (ipsc.ksp.sk), and the Continental Math League (www.continentalmathleague.hostrack.com). Each of these programs offers opportunities for individual achievement or teamwork in designing algorithms or programs to solve a given problem, usually in limited time. This type of study will most likely be more worthwhile for students interested

in pursuing college majors in computer science than learning new languages or studying for a computer certification exam likely to expire or become obsolete by the time the student enters the workplace.

Even if a student's study of algorithms stops at the introductory computer science level, he or she will have gained a fundamental and highly practical set of problem-solving skills applicable to any number of fields. Engineers, lawyers, artists, and doctors certainly employ systematic approaches to problem-solving on a daily basis. The ability to identify algorithmic problems within a complex environment and formulate solutions will prove beneficial within the context of computer programming or any other situation. Given appropriate curriculum adjustments, this skill naturally could be taught within an introductory computer science classroom, leading to both a greater understanding of CS itself and valuable connections with other fields of study.

References

[1] Baldwin, D. Using Scientific Experiments in Early Computer Science Laboratories. In *Proceedings of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education (SIGCSE 1992)* (Kansas City, Missouri, USA). ACM Press, New York, NY, 1992, 102-106.

[2] Carter, L. "Why Students with an Apparent Aptitude for Computer Science Don't Choose to Major in Computer Science." In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)* (Houston, Texas, USA). ACM Press, New York, NY, 2006, 27-31.

[3] Moorman, P. and E. Johnson. "Still a Stranger Here: Attitudes Among Secondary School Students Towards Computer Science." In *Proceedings of the Eighth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2003)* (Thessaloniki, Greece). ACM Press, New York, NY, 2003, 193-197.

[4] O'Lander, R. "Factors Effecting High School Student's Choice of Computer Science as a Major." In *Proceedings of the Symposium on Computers and the Quality of Life (CQL 1996)* (Philadelphia, Pennsylvania, USA). ACM Press, New York, NY, 1996, 25-31.

[5] Skiena, S. *The Algorithm Design Manual*. New York: Springer-Verlag, 1998.

[6] Winston, P. and B. Horn. *Lisp*. Reading: Addison-Wesley Publishing, 1989.