Nifty Assignments

What is Computer Science?

Nick Parlante

> Computer Science is not about computers, any more than astronomy is about telescopes
> – *Edsger Dijkstra*

I've been thinking about the meaning of the January "What is CS" thread on the SIGCSE email list. Like many of you, I got behind reading it, and in the back of my mind, I was afraid as I caught up that it might degenerate into that special hell demonstrated on so many newsgroup and Slashdot threads — a "discussion" gradually dominated by the most polar, loud, and least self-disciplined contributors.

I need not have worried. The discussion was entirely reasonable. I would argue that the many positions shared the idea of a sort of spectrum of Computer Science with "science" on the one end and "engineering" on the other. In the comparisons between science and engineering, I was curious to see if the above Dijkstra quote would pop up, and ultimately it did, although it came out in a tone more of reflection than combat. My sense is that the Dijkstra quote has not aged well.

Science – Algorithmic

First, we have the traditional mathematical, algorithmic research within Computer Science. In this category, I think of purely algorithmic ideas, such as Huffman coding, or solving Rubik's Cube, or the page-reference insight that underlies Google's page ranking heuristic. We might group this sort of work into an "algorithmic" category of Computer Science. At this abstract end of things, the ideas have a pleasing, mathematical timelessness. Each distills some fundamental insight that will remain true across time and systems. You know that a piece of work is at this abstract, algorithmic end of things when its major components can be worked out using only chalk and a blackboard.

I think we all admire the elegant, mathematical purity of the algorithmic side. By the same token, a person who strongly identifies with the algorithmic side might look down on the engineering side, cluttered as it is with the often messy details and compromises of actual implementation.

Every Nifty Assignment has a core issue that will consume most of the student's attention. For some Nifty Assignments, the core issue is purely algorithmic — the students will spend most of their time wrestling with an algorithmic problem, and once they have a solution algorithmic in mind, the specific language or syntax or whatever they use to express their solution is not so important. For example, Nifty Assignments with an algorithmic core include Rich Pattis' DNA project, Joe Zachary's Random Writer, or the Tetris playing AI in my own Tetris project.

Engineering – Systems

On the other hand, the engineering side is made of the techniques and challenges of building up complex systems. We look at a complex problem and think through strategies and designs we can assemble to solve the whole thing. I'll place this under the moniker "engineering" although perhaps I should call it "design" or "systems engineering."

One challenge on this side involves the marriage of the core algorithmic idea with the available technology – how can it best be done? Obviously we must take account of the efficiency of the solution on the computer. More subtly, we must think about the solution's efficiency in its use of programmer time, and the characteristics of the solution beyond the problem at hand, such as its capacity to change and generalize in the future. These engineering tradeoffs are governed by the worldly details of the currently available systems, and the modularity needed so that many people can co-ordinate their efforts.

In contrast to an algorithmic challenge, the overarching challenge on the engineering side is *complexity*. Essentially, this reduces to the abstraction and divide-and-conquer themes that run through CS from the first day. Given a large problem, how can it best be carved up into individually solvable pieces? The design gets bonus points for integrating testing, taking best advantage of available library code, and generalizing appropriately for future uses.

Among Nifty Assignments, often a good part of the challenge operates at this complexity-taming design level. Take for example, Julie Zelenski's Quilt project, or my own Darwin's World or Name Surfer. I would say that the algorithms inside these projects are fairly straightforward. The main challenge is the complex-system problem of modularizing and organizing the solution out of its component parts.

I can see where the algorithm people might want to look down their noses at the engineering people for reasons of mathematical purity. However, there are just as many reasons for the engineering people to hold their heads up high.

Arise, Young Engineer

First, I would argue that the engineering problems have just as much intellectual heft as the algorithmic side. There is nothing simple or obvious about a nice solution to a complex system. Granted, the engineering side tends more

towards tradeoffs and judgment calls versus the more pure esthetic of a nice algorithm.

Most importantly, look at the most interesting software projects in the world today, such as Eclipse, Mozilla, JIT optimizers, or anti-spam frameworks. Each of these contains a few, key algorithmic problems – the mathematics behind the encryption in Mozilla, or the escape-analysis in the JIT engine. Those are classical algorithm problems, and as such, they sound interesting and appealing to all of us. However, look at the projects as a whole. Each has a few islands of pure algorithm integrated into a large mesh of system complexity. The Eclipse project is a fantastic example. What is the main challenge in building something like Eclipse? What knowledge does someone need to contribute to that project? I would argue that many of its features – displaying, editing, organizing, compiling, debugging – are individually pretty modest by modern algorithmic standards. The challenge of building Eclipse is coming up with an elegant design that allows all those components (and their authors!) to work together as part of a complex system. Certainly, many problems we and our students will want to solve in the future will require exactly Eclipse style complexity-taming design.

Can't We All Just Get Along?

The obvious conclusion, and the tone on the SIGCSE list, is that CS students will need to be fluent in both algorithms and engineering. Algorithmic ideas are the basic building-blocks within our complex systems. Our students will also certainly need the engineering design skills for Eclipse-like complex systems. Perhaps there was a day when there were a few algorithmic leaders figured out the key issues, leaving the implementation as a finishing exercise. That is not what interesting CS projects look like today. It is very difficult to find a problem that does not show significant aspects from both sides.

I suspect that the advances in modern tools, languages and techniques have ballooned the size and number of components in a typical project. Just look at what we can assign for CS1 projects now versus 20 years ago. This inflation in project size makes complexity-taming design increasingly important. In this respect, I think that the Dijkstra quote has not aged well versus what I see as the challenges of modern Computer Science.

Another argument is that the term "Computer Science" should be reserved for the algorithmic side, using some form of "engineering" to refer to the systems design side. I agree that there are two distinct themes at work, however I think it's too late to change what Computer Science means – it clearly refers to the unification of both ideas. I think this no accident. The term Computer Science refers to both because the community recognizes that both skills are vital. I need only think of the most interesting modern projects like Mozilla, or Eclipse or gcc, or Nifty Assignments like Tetris or Name Surfer. In all those cases, a person strong on only one side or the other would be lost.