Computer Science != Programming

ENGW3302: Project 1 Draft 1

Citation: IEEE

Paul Langton

September 11, 2019

1 Introduction

Programming computers is easy. People laugh and descend into self-doubt when I say that, but an hour of Internet searching and copy-pasting can accomplish a huge array of simple tasks while requiring no more knowledge of computers than how to use a mouse and keyboard. I have been programming computers since my freshman year of high school, but my education in Computer Science (CS) did not begin until college. CS is another beast entirely. The goal is no longer to program a computer; it is to understand the capabilities of computers and develop new methods of automation.

My college education in CS can be effectively viewed through the lens of the *problem* and the expression of the corresponding solution. The questions my instructors sought to answer were: How do we express a problem and its solution? Can we solve every problem with computers? Finally, if we can't, why are we still trying? In this essay I will expand on ~~my hubris~~ a few relevant events in my education, connect them to these key questions about the nature of problems, and reflect on the teaching methods employed.

2 Abstraction

The first concept I ever struggled with in CS was that of recursion, or the invocation of a function A inside A itself. It is an intrinsically confusing concept because there are few parallels in the real world. In general, objects in our world cannot contain themselves. The concept finally clicked in a lecture in Olin Shivers' Fundies I class. He told us we could think of recursion as a simple, abstract, two-step process over a set of data: act on the first thing, and then call your function on the rest of the data. This was in stark contrast to any of my previous instruction, which taught recursion as a recursive condition, an end condition, and an increment. This approach describes the structure of recursion in a block of code but betrays its purpose, which is to act on data.

This approach draws its roots from Matthias Felleisen, who advocated for the use of functional programming in teaching undergraduate CS for precisely the demonstrated reason: it is easy to teach basic CS operations (like doing something to each item in a list) using functional paradigms when compared with the imperative style, as is common in introductory CS courses [1]. Using this simple functional programming-based abstraction, Olin was able to effectively convey the intuitively complex concept of recursion. This moment marked the first step in my understanding of *problems*: there is more than one way to express a problem, and certain abstractions are better suited for certain problems.

3 Computation

Armed with the above knowledge and drunk with my newfound powers of abstraction, my third year brought the next major development in my problem lens. It was time for me to realize that, no matter how clever an abstraction

was applied, certain problems could never be solved by computers. Rewind several years to 10th grade me, staring at sudoku puzzles, trying to find a clever automated way to solve them, without success. In a few short years I would be sitting in my computing theory class in a lecture on NP-Completeness only to realize that solving sudoku is NP-Complete in the size of the board. The "NP-complete" problems compose a set of tasks which are solvable, but there exists no "clever" solution. The only way to guarantee a correct answer is to try all possible solutions, and thus the NP-complete problems have no *computationally efficient* solution.

This fact is underpinned by results from Cook and Karp who established the NP-Complete class of problems, and showed via reduction that every problem in NP is equivalent to Boolean Satisfiability [2] [3]. Boolean satisfiability basically asks if you can find an assignment of either "true" or "false" to a set of variables that show up in a boolean equation (e.g. $A \wedge B \vee C$). Reduction is a proof technique which entails assuming you have a solution to one hard problem P, and using it to solve some other known hard problem Q. This shows that P must be at least as hard as Q. This exact reduction technique was popularized in teaching undergraduate CS theory by Michael Sipser [4]. Until this point I had been using intuition to judge relative difficulty, so while this result was unsatisfying to some degree, the reduction technique gave me a powerful tool to reason about the difficulty of a problem.

4 Application

Despite the previous setback, I remained confident in my abilities. I knew how to figure out whether problems were solvable and I could solve those problems with abstractions. Then I took Advanced Algorithms with Professor Nguyen and I realized a) neither of those things are true, and b) I don't know enough math to be remotely useful. His lecture style was unfailingly traditional: he went up to the chalkboard, wrote math for 1.5 hours, and took the occasional question. The real value of the course came from the instructor's precise understanding of every minute detail of the material. It is hard to identify a single aha! moment from this course. Every lecture was a thrilling learning experience and a firehose of new information. I learned as I went, picking up more useful nuggets of math than in all my previous math classes combined. We went in depth on real world problems like Locality-Sensitive Hashing (LSH) [5], stochastic gradient descent [Cauchy], and many others. $1 + x \leq e^x$ can be used to show that LSH will not overload any machine with high probability. Making weighted random decisions instead of deterministic ones will make certain algorithms perform more than twice as well. Stochastic gradient descent is how machine learning algorithms "learn", by slowly sliding down the "inaccuracy hill". The theory behind the content covered in class was vast. It was during this course that I came to realize how many interesting problems existed between the extremes of "doable by a college freshman" and "computationally impossible." It also showed me how distinctly powerful and *applicable* these seemingly abstract problems could be. It was that realization in conjunction with this course that led me to pursue my PhD.

5 Instruction

It is worth noting that the course instruction throughout each of these experiences was a combination of lecture and project-based learning. The lecture has taken a lot of flak in modern times [6]. I think people conflate problems caused by manufactured slide decks, droning instructors, and 200+ lecture populations with problems caused by the lecture style. Without a doubt my best learning was done in classes with no more than 60 students, an engaged professor , chalkboard writing instead of slides, *and in lecture style*. I found this environment to be especially effective when discussing more theory-heavy techniques like gradient descent. My single data point far from exonerates the lecture, but hey you came here to read about my learning journey, not collect statistics on teaching style.

Project-based learning, while not as prominent, was another significant part of my learning. This style is accredited to John Dewey and underlines the importance of learning through doing [7]. It was implemented at least twice per semester, mainly in homeworks. The Networks course, for example, had us implement the Raft protocol [8], which was both challenging and rewarding. The project-based style instilled great respect in me for the implementors and maintainers of high-performance production systems. While these methods were interesting, I think the majority of my intellectual gain came from genuine interest in the material. Anecdotally, the challenge, the quality of the content, and the familiarity of professor with their material matters more to my learning process than the supporting instructional philosophy.

6 Acknowledgemention

I would like to thank Olin Shivers and Huy Lê Nguy˜ên for giving me material with which to write this paper. I would also like to thank the Democratic Party for doing the absolute bare minimum by beginning impeachment proceedings against the crying orange baby we elected as temporary dictator-for-life-via-executive-order.

References

[1] M. Felleisen, *How to Design Programs*, 1st ed. MIT Press, Cambridge, MA, 2001.

[2] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: ACM, 1971, pp. 151–158. [Online]. Available: http://doi.acm.org/10.1145/800157.805047

[3] R. M. Karp, *Reducibility among Combinatorial Problems*. Boston, MA: Springer US, 1972, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2$_9$

[4] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, Stamford, CT, 1996.

[5] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala, "Locality-preserving hashing in multidimensional spaces," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 618–625. [Online]. Available: http://doi.acm.org/10.1145/258533.258656

[6] S. Freeman, S. L. Eddy, M. McDonough, M. K. Smith, N. Okoroafor, H. Jordt, and M. P. Wenderoth, "Active learning increases student performance in science, engineering, and mathematics," *Proceedings of the National Academy of Sciences*, vol. 111, no. 23, pp. 8410–8415, 2014. [Online]. Available: https://www.pnas.org/content/111/23/8410

[7] J. Dewey, "My pedagogic creed," in *The School Journal*, ser. Volume LIV, Number 3 (January 16, 1897), 1897, pp. 77–80.

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666