Introductory Computer Science Courses: Experiences & Thoughts

Haakon Ringberg

February 7, 2007

1 Overview

The purpose of the following document is discuss how to design the first few courses in an undergraduate computer science (CS) curriculum. The author will report and draw upon on experiences from courses that are taught at select American research universities in order to provide context for the discussion. The document is organized as follows: section 2 discusses possible goals one might have for introductory CS courses; section 3 details the approaches taken by a few courses the author has personal experience with; and the document concludes with the author's opinions in section 4.

2 Goals in Designing the Courses

In order to successfully devise a plan for most endeavors, it is important first to identify what it is one wishes to accomplish in the end. Our present agenda is no exception, as different decisions regarding the content and character of the first few CS courses often stem from divergent opinions regarding the purpose of a computer science education. Perhaps the most basic such debate is whether the fundamental obligation of a CS department is to prepare students to be successful engineers or teach students how to think abstractly about computational problems. These are only two very high-level goals, however, and in this section we will elaborate on these and others.

2.1 Software Engineering

Pro The argument in favor of focusing on training successful engineers is that the majority of computer science graduates enter into industry. A student that enters the job market fully versed in the tools and techniques most prominent in industry will have a clear competitive advantage over those who do not.
Con A common counterargument notes that computer science is such a rapidly changing field that a student who is overly indoctrinated in the momentary particularities of industry will not be able to adapt with the environment.
Within this high-level goal there are various sub-goals:

1. Teach the languages most prominent in industry (e.g. C, C++, C#, Java)
2. Teach the tools most prominent in industry (e.g. IDEs, Oracle DB)
3. Teach general software engineering techniques (e.g. specs, modularization, representation invariants, testing, documentation)

2.2 Algorithms and Abstraction

Pro Designing algorithms is in many ways the most fundamental aspect of computer science. The ability to abstract away the core problem and attack it efficiently is a skill that will serve students well whether they end up industry or academia.

Con A common counterargument is essentially the pro for the engineering approach, i.e. that a student without adequate knowledge of the engineering fundamentals will be at a competitive disadvantage in the job market because one there has to be able to implement the solution in addition to conceive of it.

2.3 Other Goals

There are a myriad of other possible goals for the first required CS courses.

1. There are topics that are important to each of the above two high-level approaches, including standard algorithms and data structures, complexity analysis, and program proof of correctness. The former two, in particular, are absolutely essential for theoreticians and practitioners alike.

2. One can also debate whether it is worthwhile to introduce students to a variety of programming paradigms and/or programming language features. Herein I include functional programming, object-oriented programming, logic programming, eager evaluation, type safety, parametric polymorphism (e.g. C++ templates), higher-order functions, exceptions, pattern matching, etc

3. To what extent should the history of computer science be included in the first few courses?

4. What should be the prominence of computer science's many subfields or related fields: compilers, programming languages, architecture, databases, networking, operating systems, discrete mathematics, artificial intelligence, scientific computing, graphics, computational linguistics, cryptology, security, quantum computing, theory of computing, analysis of algorithms, ... and we haven't even mentioned the web!

5. Should we attempt to market/"sell" computer science as best we can, for example by presenting a high-level though somewhat superficial overview of as many exciting aspects as possible? Such a course would be akin to introductory courses in the traditional sciences (physics, biology, etc).

3 Anecdotal Experiences

3.1 Cornell CS211

CS211 was the second required programming course at Cornell and was taught in Java. It taught recursion, OOP principles, basic algorithms (e.g. searching, sorting) basic data structures (e.g. linked lists, trees, graphs), basic analysis of algorithms, and basic specifications.
As a student, I felt the course was straightforward.
`http://www.cs.cornell.edu/courses/cs211/2001fa/`

3.2 Cornell CS312

The third, and final, required programming course at Cornell was taught in SML. It covered more advanced data structures (e.g. red-black trees, hash maps, splay trees, LRU tries, binary heaps, B-trees), more algorithms (e.g. Dijkstra's, A*, BFS/DFS), substitution and environment models, copying/mark & sweep/reference counting GC, type checking/inference/unification, concurrency, DFAs/NFAs/REs, specifications, abstraction functions, representation invariants, modularization, testing, tail recursion, inductive correctness proofs, analysis of algorithms, locality, Huffman coding, and continuations.
As a student, I felt 312 was the most challenging CS course I had ever taken, but also the one that had taught me the most about CS by a wide margin. I had the same perception as a TA although I came to feel that some topics are probably better saved for later courses (e.g. theory of computation, continuations).
`http://www.cs.cornell.edu/courses/cs312/2004sp/`

3.3 Princeton CS 126

The first required programming course at Princeton is taught in Java. It tries to teach students the basic aspects of programming while presenting a broad range of concepts from CS subfields. In terms of programming, the course introduced basic I/O, arrays, functions, recursion, sorting algorithms (up to quick sort), searching, linked lists, stacks and queues, and symbol tables. The course also taught basic: analysis of algorithms (including O(n log n) for QS), computer architecture (including assembly language), theory of computation, universality, computability, intractability, combinational and sequential circuits, and cryptography.

As a TA, my experience was that few students understood the subject matter taught during the latter half of the course, why the topics are important, and how they are related.

`https://whiteboard.cs.princeton.edu/coursehome.php?course=COS126&semester=fall05`

3.4 Princeton CS 333

CS 333 is called "Advanced Programming Techniques" and is an optional course at Princeton. It covers the high-level strengths and weaknesses of a slew of languages. The students spend the latter half of the semester designing a large, web-based, project with a three-tiered architecture. Topics covered in the class include regular expressions, scripting languages (shell, awk) and their usefulness in testing, yacc & lex, perl, python, HTTP, CGI, PHP, javascript, CSS, XML, Ajax, Java, C++, OOP, interfaces, MySQL, etc. The students are encouraged to write interesting final projects, and the course is successful in bringing home the point that languages are tools that ought to be chosen according to their suitability to the task at hand. The course's success is is no small part due to the Professor's — Brian Kernighan — expertise and experience

`http://www.cs.princeton.edu/courses/archive/spring06/cos333/`

4 Opinions

My potentially unsatisfying response to the software engineering versus algorithm abstraction debate is "yes please, both". I think it is essential that both topics are covered in the first few CS courses. They cannot necessarily be equally emphasized in each course, however, and I believe their relative importance to a course should depend on its purpose. Specifically, it has become increasingly common that U.S. engineering schools require all engineering students to take at least one programming course. For a course that serves such a purpose, and this is often also the first required course for the CS major, I believe the *primary* focus has to be on the basics/fundamentals of programming. I believe students attending this course will come from such varied backgrounds that they will not be able to usefully apply algorithm design without an understanding of how to implement it in some programming language.

While software engineering is admittedly a vague term that is often abused, I believe the constituent concepts mentioned in previous sections ought be included in any undergraduate curriculum. I believe a student that graduates without basic knowledge of modularization, testing, and documentation will be at a competitive disadvantage because these concepts introduce a programmer to the techniques that separate good programmers from those who write unmaintainable hacks. Furthermore, I believe that more advanced techniques such as representation invariants, abstraction functions, and correctness proofs should be included in a curriculum that aims to set a high standard.

However, while I believe that software engineering is important, I do not think a CS curriculum ought to teach only the languages/tools that are most prevalent in industry. There are several reasons for this, including the fact that such particularities change over time. Languages or frameworks can quickly gain inroads in niche areas (e.g. the Ruby on Rails web application framework) and the possibility of switching subfields means that it is useful to be able to readily pick up a new language or programming paradigm. Finally, a student who graduates well-versed in some theory/model/paradigm (e.g. relational databases and SQL) will be able to master a given instance thereof (e.g. MySQL or PL/SQL).

I am not the only one who holds the aforementioned belief. In fact, MIT's first programming course is taught in Scheme, ENS teaches at least some of its required programming courses in OCaml, and Cornell's final required programming course is taught in SML — all of which adhere to the functional programming paradigm. In addition, OCaml and SML have become popular partly because of their safety, which is also increasingly emphasized in industry languages, e.g. much of Microsoft's code is now written in the strongly typed C#.

I also think it is a good idea to have at least one introductory CS course that pushes students hard to excel as programmers. I find that students can often rise to meet such a challenge. CS312 at Cornell is my prime example of this strategy, and to this day it remains my favorite course. Professor Andrew Myers summed up the experience quite well on the last day of class when he said that *"Taking CS312 is liking trying to drink from a fire hydrant."* While I doubt that many students were able to retain all the information they were presented in 312, it is almost without exception the one course that Cornell CS students refer to as having taught them the most about CS.

In terms of the other goals listed in section 2.3, it is probably already clear that I do not think the history of CS or specific CS subfields ought to receive great prominence in introductory CS courses. That is not to say that I oppose their inclusion, but rather they should appear only insofar as they provide context and relevance for given techniques (e.g. string matching and bioinformatics, Gale-Shapley and its usefulness in matching medical students with hospital jobs, etc). In other words, such background can be the spice that a particular professor or application brings to an algorithm. I believe "Algorithm Design" by Professor Jon Kleinberg and Eva Tardos is a book that provides much interesting and insightful background to motivate algorithms.

5 Acknowledgments

The author would like to thank David Menestrina, Jeffrey Vaughan, Emmanuel Schanzer, and Saikat Guha for valuable feedback on this document. Some of their further comments can be found at:
`http://www.cs.princeton.edu/~hlarsen/work/intro_courses.htm`