THE SKILLS HIERARCHY AND CURRICULUM

Philip Machanick
Department of Computer Science, University of the Witwatersrand
2050 Wits, South Africa
philip@cs.wits.ac.za

Abstract

A commonly accepted hierarchy of cognitive skills puts analysis and synthesis near the top, with straightforward knowledge and comprehension at the bottom. A typical Computer Science curriculum, though, usually starts with programming, an activity that requires both analysis and synthesis. The student, without the right conceptual skills, attempts synthesis without analysis, resulting in poor programming skills. This paper presents a view of Computer Science curriculum, drawing on earlier work on abstraction-first learning [Machanick 1998], to propose a curriculum which starts from lower-order cognitive skills, while working up to higher-order skills in later years.

1. Introduction

It is commonly accepted that the hierarchy of cognitive skills—as defined in Bloom's Taxonomy—has analysis and synthesis near the top, and straightforward knowledge and comprehension at the bottom [Bloom 1956]. If you look at a typical Computer Science curriculum, though, where does it start? Usually with programming, an activity that requires both analysis and synthesis. What typically happens is that the student, without the right conceptual skills, attempts synthesis without analysis, resulting in poor programming skills—we end up training hackers. This paper presents a proposal to restructure the Computer Science curriculum, drawing on earlier work on abstraction-first learning [Machanick 1998], to propose a curriculum which starts from lower-order cognitive skills, while working up to higher-order skills in later years.

Since the last major revision of the ACM/IEEE curriculum appeared in 1991 [ACM 1991], perhaps it's time there was a major review of our underlying assumptions.

The proposed ordering is presented first in terms of an abstract breakdown of Computer Science, to avoid assuming that current subjects or topics fit the new framework. A possible curriculum based on the abstract breakdown is then presented.

The abstract breakdown is based on two principles: low-level-cognition-first (LLCF), and abstraction-first (AF) learning. The LLCF strategy is based on the following hierarchy of skills (with a further level of breakdown): know, comprehend and apply, and finally, analyze and synthesize. The highest-level skill in Bloom's Taxonomy, evaluate, is reserved for graduate-level courses and research. The AF approach is based on the following ordering: use an abstract virtual machine, understand its components, construct with existing components, build new components, design new abstractions.

How can these orderings apply to setting up a Computer Science curriculum? The AF approach leads to defining layers of a virtual machine, with emphasis on the outer layers early, and more depth of the inner layers later. The LLCF strategy leads to deciding which cognitive skills to exercise and evaluate at each stage of the curriculum.

The remainder of this paper discusses the proposed approaches in more detail. First, Bloom's Taxonomy is revisited, then the AF approach is described, then the LLCF strategy. An combined strategy leading to a curriculum proposal is put together next, and finally a conclusion presents a way forward.

2. Bloom's Taxonomy

2.1 introduction

Bloom's Taxonomy has long been recognized as a valid approach for dividing skills into those at a beginner's cognitive level, through to higher-level abilities. One of the strengths of this classification of skills is that a taxonomy, unlike a straightforward classification, is rooted in anobjectively-determined framework, whereas a classification in its more general sense may be based on arbitrary criteria [Bloom 1956]. Bloom's Taxonomy is founded in extensive research and surveys of educators. Accordingly, it is a useful framework for judging the appropriateness of a given kind of task for a given skill level.

Much of the material in Bloom's report relates to artistic or creative work but relatively minor adaptation makes the ideas suitable for Computer Science. For illustrative purposes, examples from Computer Science are added occasionally to this summary of Bloom's report; the major discussion of application to Computer Science education follows in remaining sections of the paper.

The taxonomy orders skills as follows, from lowest cognitive skills to highest:

- knowledge—factual knowledge
- comprehension
- application
- analysis
- synthesis
- evaluation

While other breakdowns are possible (and the authors of the original report acknowledge this), it is useful to use this broad breakdown as a basis for examining any curriculum in terms of the order it presents material and the demands it makes of students at each stage of their studies. Any deviation from the order suggested by Bloom's Taxonomy can of course be justified, but it is a useful start to compare skills against such an accepted taxonomy, to reveal any major problems in the way a curriculum is structured. This paper is rooted in concerns that Computer Science curricula are designed with too much emphasis on which topics to teach when, and too little on which cognitive skills to exercise and evaluate when.

Let us briefly consider the kinds of knowledge covered by each level of the taxonomy.

2.1 knowledge

Knowledge covers a range of areas from simple isolated facts, terminology, to specific facts. Educational goals in these areas essentially cover simple recall, ability to recover knowledge from standard sources,

knowledge or properties of entities in a given knowledge domain, being able to define concepts, being able to establish limits on the meaning of words in a given (e.g. technical) context and being able to converse intelligently about a given subject. After knowledge of specifics, there's knowledge of ways of dealing with specifics: knowledge of conventions, knowledge of trends, knowledge of classifications and categories, knowledge of criteria, and knowledge of methodology. The final level of the knowledge part of the hierarchy is knowledge of universals and abstractions in a field: principles and generalizations, and theories and structures.

Note that none of these points restrict the complexity or sophistication of the concepts being dealt with, just the depth to which they need to be understood. In general, the educational objective at this level is that the student should know concepts, without necessarily having a deep understanding.

2.2 comprehension

Comprehension is the next level. Here, the learner is expected to start to make sense of concepts, and be able to deal with them in a way that shows understanding. Comprehension has to be tested in different ways to factual knowledge: students have to demonstrate the ability to interpret their knowledge, and to make predictions which extend their existing knowledge.

Bloom's report proposes testing skills in this area by translation, interpretation and extrapolation, very different kinds of examination techniques than would be used in a straightforward knowledge assessment.

2.3 application

Application is distinguished from comprehension in that comprehension can be demonstrated by showing that a student *could* do something; application by showing that the student *will* actually do it.

Bloom's report illustrates the difference between comprehension and application with an example where the application-based test of skills starts with a problem for which the student must restructure the problem domain to match a familiar example, classify the problem, select an abstraction (theory, principle, etc.) and use the abstraction to solve the problem. By contrast, in a test of comprehension, the student would be given the abstraction and be told to solve the problem.

By way of example, in Computer Science, application would be represented by giving a student a problem for which a familiar algorithm would probably work, in which the problem was to find a suitable algorithm, and show that it had the required properties (complexity, correctness, etc.). Comprehension on the other hand would be tested by a problem in which the student was given the algorithm and asked to show that it had the required properties.

2.4 analysis

Analysis is yet a more advanced form of application of knowledge, requiring skills in organizing and structuring components of a solution, and ensuring that the overall solution works. Analysis is broken down into elements of a solution, and combination of those elements.

Analysis of elements requires recognition of unstated assumptions, understanding the difference between facts on the one hand and hypotheses or opinions on the other, and understanding the relationship between a conclusion and the steps to arrive at a conclusion. Combination of elements requires understanding relationships between components (what is and isn't relevant, causal relationships and to identify logical

fallacies), and understanding how elements are organized (recognition of form and pattern, recognize viewpoints in others' work).

For Computer Science, analysis would include design skills—module structure of a program, constructing algorithms out of other algorithms, design of data structures for complex programs, and performing analysis of all of these elements of the programming task.

2.5 synthesis

Synthesis involves constructing complete solutions out of components. While there are aspects of this skill in lower levels of the hierarchy, synthesis requires more complete understanding of the overall process, and the ability to arrive at a more complete solution.

The aspects of synthesis most relevant to Computer Science include producing a plan to meet requirements of a task (including proposing how to test an hypothesis and integrating results of research into a solution plan, ability to produce a complete design from a given specification and the ability to use theory to define a new process), ability to derive a new set of abstract relations (formulate hypotheses, ability to convert specific instances to a conceptual structure, and ability to make generalizations).

2.6 evaluation

Evaluation in the sense used in Bloom's report is the ability to make judgments, and is intended to reflect a higher level of cognitive skill than for example choosing between alternative methods, as might have to be done in application, analysis or synthesis.

Assessment of students' judgment may be based on various criteria: the ability to assess accuracy of reported facts, ability to apply given criteria to judge a piece of work, the ability to find logical fallacies in a given piece of work, the ability to compare major theories and pieces of work, and the ability to assess judgments and values involved in choices others have made.

3. Abstraction-First (AF) Ordering

The Abstraction-First (AF) approach is another classification (possibly taxonomy) of skills, which is specific to Computer Science, though it may have other applications.

The general idea of the abstraction-first approach is to draw on ideas developed to simplify programming to simplify the task of teaching programming. Another part of the motivation for the course is the observation in industry that practicing reuse does not happen automatically; some form of re-education is necessary, if programmers have been schooled in more traditional coding styles [Auer 1995; Berg *et al*. 1995; Fayad and Tsai 1995; Frakes and Fox 1995]. Designing from scratch is meant to be harder than using a library of reusable classes, so why not teach using reusable classes before introducing programming from scratch? In particular, why teach students to program in a style that does not emphasis good programming principles, then try to make them unlearn their bad habits? Examples of bad habits that are hard to break include failure to design before coding, failure to break code down using procedural abstraction, failure to use data abstraction to hide inessential design decisions and failure to document interfaces. All of these practices are actually taught in many introductory programming courses, just as many introductory programming courses in the early days of structured programming started by teaching unstructured coding practices (monolithic main programs, even sometimes **goto** before loops) which students were later expected to unlearn. In both cases, instructors are tempted to think of the things they learnt first as "easy" and therefore should be first in the curriculum.

The AF approach is based on starting with the highest level of a virtual machine, and moves downward. The order of topics in a data abstraction and algorithms course, for example, is:

- user-level abstractions—non-computer examples, user interfaces
- understanding abstractions created by others—working through examples built using class libraries and frameworks
- reusing abstractions created by others—using class libraries and frameworks to construct simple programs
- building new abstractions—creating new classes
- building new general abstractions—creating new container classes

Interleaved with this is algorithm and data structure analysis, which is first introduced in terms of given classes, and is extended to cover analysis of new classes and data structures.

The abstraction-first approach has been applied with some success in a C++-based data abstraction and algorithms course at second-year level; the novel ordering appeared to work, in that students' results did not suffer relative to an earlier more conventional course, despite the fact that much more ground was covered [Machanick 1998].

It is proposed here that the AF approach be extended to other areas of the Computer Science curriculum. One way of doing this is for Computer Science to be seen as the study of layered virtual machines, resulting from which it becomes appropriate to start with the highest-level virtual machine, and work inwards to the lowest-level virtual machine.

4. Low-Level-Cognition-First (LLCF) Ordering

The Low-Level-Cognition-First (LLCF) ordering is a novel strategy, based on adapting Bloom's Taxonomy to Computer Science.

The general idea is to treat subjects early on in the curriculum in a way which only requires lower-level cognitive skills, gradually working up to the higher-level skills. The intent is to propose an alternative to conventional curricula which start with programming, typically move on to systems subjects in the second year and end with programming languages, software engineering and possibly more theoretical areas in third year. There may be some variation around this ordering, but starting early with programming in some form is almost universal.

The most important idea in the LLCF strategy is that the year in which content is placed is less important than the kind of skills that are tested. Some areas—such as inventing a new theory, designing large systems, or researching a new approach to instruction-set design—inherently use higher-level cognitive skills than others. However, many subjects can be handled at different levels. For example, programming can be explained in a factual way: this is what the task is, what a program does, what language elements are. On the other hand, a complete understanding of programming from scratch (given a problem with no hint as to solution method) requires analysis and synthesis.

Table 1: Combining the Orderings

	virtual machine layers			
skills hierarchy	*applications*	*tools*	*systems*	*hardware*
know	run, use for problems (e.g. spreadsheet, DB), understand I/O specs, measure performance *vs*. known complexity	purpose of compilers, libraries, interfaces, formal definition of interface's "contract"	launch programs, organize work, use internet, use OS tools e.g. resource usage	know parts, higher-level organization, size/performance/cost issues
comprehend, apply	combine components' "contracts": I/O spec, pre- and post-conditions, complexity analysis	automata and formal languages, relational database and other models, normalization	concurrency, queuing, scheduling, resource management, networks, distributed systems	computer organization: logic circuits, quantitative design (benchmarks, simulations)
analyze, synthesize	extend libraries, design new software architectures	projects	projects	architecture: board-, chip-level, instruction-set, memory, I/O, etc.

A specific proposal for the use of the LLCF strategy is to deal with straight knowledge in introductory courses, comprehension and application in intermediate courses, analysis and synthesis in advanced courses, and to save evaluation for graduate-level courses and research. There can be some debate as to where the dividing lines should be placed (for example, if comprehension should be introduced later in the first year), but the fact that Bloom's Taxonomy is based on an objectively-determined hierarchy makes a strong case for using it as a basis for ordering the depth at which subjects are taught.

In the LLCF model, then, understanding what a given program does, understanding what the building blocks to create programs are, etc., are suitable introductory skills—but designing a program from scratch is not. At most, learning to convert an algorithm to code could be considered an introductory skill, but even that is doubtful, unless the algorithm requires little translation, as translation is a "comprehension"-level skill in Bloom's Taxonomy.

5. Combined Strategy and Computer Science Curriculum

If the LLCF strategy is combined with the AF strategy, it suggests that ordering of topics should proceed from higher-level to lower-level virtual machines, while also proceeding from lower-level cognitive skills to higher-level skills. One possible way of achieving this ordering is to work from a high level to a low level virtual machine across a year, while moving to a new cognitive skill level each year.

Table 1 illustrates how the two orderings can be combined, based on spreading virtual machine layers across a year and skill levels down the years.

Table 2 reformulates Table 1 as a potential 3-year curriculum. Note how programming is moved later, yet half of the third year is devoted to project work.

Table 2: A Sample Curriculum

	topics			
year	*applications*	*tools*	*systems*	*hardware*
1	use and measurement of applications	introduction to programming tools	basic NW and operating system	basic computer organization
2	basic program construction	formal languages and databases	advanced NW and operating system	computer organization
3	advanced programming and software architecture	compiler and DB projects	NW and operating system projects	advanced architecture

A slightly more detailed breakdown of each topic, illustrating how the skill levels are developed down the years, follows.

Year 1

- *use and measurement of applications*—know user interfaces and virtual machine concept; know major components of a computer; know functionality of a range of programs; know how user requirements are specified; know the purpose and use of user-level documentation; given a complexity result, measure a program's speed in a lab, and verify that a graph of run time matches the predicted speed

- *introduction to programming tools*—know the difference between source code, compiled code and libraries; know the steps in the program construction process; know the purpose of programming environments including editors, compilers and linkers; know the need for error checking and debugging; know the how correctness of software is specified and verified

- *basic NW and operating system*—know the purpose of the operating system and networks; know the aspects of the virtual machine which the network and operating system provide; know virtual machines in more detail than before; know major operating system and networking concepts: processes, scheduling, resource management, memory hierarchy, protection, routing, the internet and internet-based tools

- *basic computer organization*—know the major components making up a computer; know the difference between machine code/assembly language and high-level languages; know the major microarchitecture components (registers, pipeline, buses, cache, main memory); know the role of I/O devices; know why the lower levels of the virtual machine are usually hidden

Year 2

- *basic program construction*—given a library, understand its interfaces and use its components to add to a given program; given an algorithm, construct a procedure or function to implement it; add a new function or procedure into an existing program and show that it works as specified; given a complexity result for an algorithm, prove it is correct; given two alternative algorithms, compare the known complexity results, and determine both theoretically and experimentally which is the better; given the specification of a data structure, use the data structure (implemented in a class library) in a program

- *formal languages and databases*—understand turing machines, finite automata and pushdown automata; prove simple results for each model; construct simple programs in turing machines, and simple recognizers in the other two (using supplied tools); understand the value of relations as a data representation; given unnormalized relations, perform normalization; understand the application of these concepts to databases; understand the general value and applicability of these formal approaches

- *advanced NW and operating system*—implement simple examples of concurrency, given algorithms and interfaces to system calls; understand algorithms to implement queuing; understand issues in implementation of concurrency primitives and how they are used (locks, semaphores, critical regions, race conditions); understand resource management policies and algorithms including paging and scheduling; understand layered network models; understand major policy issues in networks including routing, congestion control, switches, routers and media

- *computer organization*—understand how logic circuits are building blocks of the major components; understand how to simplify logic and combine elements to create circuits; understand how design affects performance and how performance is measured; understand how assembly-language programs are written; relate assembly language on one had to high-level languages and on the other to the hardware; understand how I/O devices interface to the hardware

Year 3

- *advanced programming and software architecture*—understand the architecture of an existing library or application and extend it; design a new relatively simple architecture (library or application): mainly project-oriented

- *compiler and DB projects*—use principles of formal language to implement small projects, to understand how compiler construction tools work (not only for parts of a compiler); do a database project; report on the value of the formal methods in these areas

- *NW and operating system projects*—implement software using operating system calls, including use of pipes or other higher-level models of inter-application communication, multithreaded applications, use of low-level networking protocols such as UDP to implement higher-level protocols; simulation

- *advanced architecture*—design principles including instruction set architecture, performance impact of variations in design, overall system design, hardware-software interactions; code generation and assembly language

Notice how the first year is mainly about *knowing*, the second year is mainly about *understanding*, while the final year focuses on *constructing* and *designing.* This is in keeping with a conventional engineering curriculum, which builds background first, and only later introduces design and construction [Baber 1997].

To complete the curriculum, mathematical background is discussed briefly. In the first year, little is required. Algorithm analysis is given, and results need not be proved. However, some familiarity with the notion of complexity and the growth rates of functions should result from the applications topic. During the first year, it would be opportune to build mathematical background in discrete mathematics, especially proof by induction. It would also be useful to introduce some statistics as a prerequisite for the advanced networks and operating systems course. An LLCF strategy of course could also be applied here, but let us leave that for mathematics educators to debate. This discrete mathematics background would be essential

for the second year when basic program construction and formal languages are taught. The more classical mathematical background of calculus could provide some useful background to the final year, where application-specific projects may need calculus (e.g. if simulations are based on solution of differential equations). It would therefore be useful if the second-year mathematics curriculum built on the first year topics, and added in calculus.

6. Conclusion

The proposed curriculum in some ways seems a rather radical change from current practice. However, it does offer a number of quite practical advantages. For one, the major equipment-related component of the course is pushed to the final year, when classes are smaller and students more capable of looking after themselves. For another, the notion that Computer Science is just about learning to program is dispelled. The difficulty of dealing with classes where some students already know how to program and others have never seen a computer before is also addressed: students who've already learnt to program have something new to learn from day one while real novices have a relatively gentle introduction. Another useful gain is that programming tools are dealt with purely as examples early in the course, so there is much less pressure to adopt the latest technology and change the curriculum frequently, at least in the more introductory years.

All of those gains though are a bonus. The real gain is in moving to a model where the content is a better fit to the cognitive skill level that can be expected of students during each year of study. What's more, we move away from teaching students concepts they cannot fully comprehend, resulting in bad habits which have to be unlearnt.

The most common argument against this approach (in discussion with colleagues) is that students expect to be able to start programming early on. In this model, there is nothing to stop students who already know how to program from extending given examples if they want to. However, the curriculum does not require that kind of programming skill early on, as all hands-on work in the first year uses pre-written software. Perhaps in any case it is time to tell students who have a hacker mentality that this is a mindset that does not go with a university-level education in Computer Science. After all, it is not as if we are short of students. In any case, students ultimately go for the curriculum that is most appreciated in the job market. If this model has the intended result—producing a more disciplined style of graduate who understands the design process, the value of reusable libraries, virtual machines, etc.—such graduates will come to be in great demand in industry over time, once industry sees how much better their work is.

References

[ACM 1991] A Summary of the ACM/IEEE-CS Joint Curriculum Task Force Report: Computing Curricula 1991, *Comm. ACM*, vol. 34 no. 6 June 1991, pp 68–84.

[Auer 1995] Ken Auer. Smalltalk Training: As Innovative as the Environment, *Comm. ACM*, vol. 38 no. 10 October 1995, pp 115–117.

[Baber 1997] Robert L. Baber. CS Education and an Engineering Approach to Software Development, *Proceedings of the 27th Southern African Computer Lecturer's Association Conference*, Wilderness, South Africa, June 1997 pp. 22-24.

[Berg *et al*. 1995] William Berg, Marshall Cline and Mike Girou. Lessons Learned from the operating system/400 OO Project, *Comm. ACM*, vol. 38 no. 10 October 1995, pp 54–64.

[Bloom 1956] Benjamin S Bloom (ed.). *Taxonomy of Educational Objectives: Book 1 Cognitive Domain*, Longman, London, 1956.

[Fayad and Tsai 1995] Mohamed E Fayad and Wei-Tek Tsai. Object-Oriented Experiences, *Comm. ACM*, vol. 38 no. 10 October 1995, pp 51–53.

[Frakes and Fox 1995] William B Frakes and Christopher J Fox. Sixteen Questions About Software Reuse, *Comm. ACM*, vol. 38 no. 6 June 1995, pp 75–87,112.

[Machanick 1998] Philip Machanick. The Abstraction-First Approach to Data Abstraction and Algorithms, *Computers & Education*, 1998, in press.