Combining the strengths of a liberal arts environment with those of computer science education.

A Revised Model Curriculum for a Liberal Arts Degree in Computer Science

HENRY M. WALKER
AND
G. MICHAEL SCHNEIDER

THIS ARTICLE PRESENTS RECOMMENDATIONS FOR A high-quality undergraduate computer science major within a liberal arts setting. These recommendations build upon the traditional strengths of a liberal arts education while ensuring reasonable depth in the fundamental areas of computer science. This article updates the 1986 *Communications* article "Model Curriculum for a Liberal Arts Degree in Computer Science" by Gibbs and Tucker [8] (see the sidebar "The Revision Process"). In updating [8], we draw upon important new educational and technological developments in the discipline as well as the recommendations of Computing Curricula 1991 of the ACM/IEEE-CS Joint Curriculum Task Force [2].

The Liberal Arts Environment

A LIBERAL ARTS EDUCATION STRIVES TO MAINTAIN BALANCE between a breadth of study in a variety of fields and a deeper understanding of a major subject area. Exposure to a number of disciplines encourages students to understand and practice problem-solving skills from different perspectives. At the same time, a liberal arts education involves the investigation of a major at a reasonable level of detail. The recommendations presented in this article demonstrate that a carefully chosen sequence of courses can provide a rigorous program of study in computer science. While a liberal arts major may not involve a large number of computer science courses, the core areas of the discipline can be carefully and fully covered. This emphasis on the core of the discipline means that a liberal arts program will emphasize fundamental principles, with applications following from and building upon these ideas in a hierarchical fashion.

Further, since many liberal arts programs are located at small undergraduate colleges, an important challenge to this revised curriculum is to create a high-quality computer science program in the presence of potentially limited faculty resources.

Existing Curricular Recommendations

This article draws upon two sets of curricular recommendations: Computing Curricula 1991 [2], published by a joint ACM/IEEE task force in March 1991, and the original 1986 "Model Curriculum for a Liberal Arts Degree in Computer Science" [8] by Gibbs and Tucker.

Computing Curricula 1991 was a fundamental rethinking of the recommendations described in ACM Curriculum 1978 [1]. It identified many important contemporary issues, and using as its basis the 1989 article "Computing as a Discipline" [7], it organized the core of computer science into 56 curricular modules, called knowledge units, structured within 11 subject areas. The description of a knowledge unit included a listing of recurring themes, lecture topics, together with a minimum number of lecture hours for presenting this material, ideas for laboratories, and connections between this knowledge unit and others. The main subject areas and the amount of time spent on each area are shown in the left-most two columns of Table 1. This material represents the core of computer science that, in the opinion of the committee, must be part of all undergraduate majors in computer science.

While Computing Curricula 1991 does an excellent job listing topics and defining the core of computer science, the knowledge unit approach requires faculty to analyze their current course offerings with regard to these 11 areas and 56 modules. In principle, sample programs contained in the Appendix of Computing Curricula 1991 may help this process, and the knowledge unit approach does permit great experimentation and flexibility. In practice, however, the review and redesign of an entire curriculum is a tremendous job, and it seems that few colleges and universities have been willing to undertake this task.

Table 1. Comparison of the hours spent in each subject area in Computing Curricula 1991 and the Revised Model Curriculum

	Curricula 1991	Approx. Hours Recommended in this Proposal						
			Introductory Courses			Core Courses (Alg.) (Found.) (Lang.)		
Subject Area	*Total*	*Total*	*CS1*	*CS2*	*CS3*	*CO1*	*CO2*	*CO3*
Algorithms & Data Structures	47	76	13	20		19	24	
Architecture	59	38	1	1	30	2		4
AI and Robotics	9	3				3		
Data Base & Info Retrieval	9	2	1		1			
Human-Computer Comm.	8	3	1	2				
Intro. to a Prog. Language	12	15	10	5				
Numerical & Symbolic Lang.	7	7	1	1		5		
Operating Systems	31	25	1	1	9	5		9
Programming Languages	46	48	3	2			16	27
Software Methodology & Eng.	44	13	5	4		4		
Social and Ethical Issues	11	10	4	4		2		
Total Number of Hours	283	240						

The need to organize topics into a hierarchical course structure suitable for a liberal arts setting was addressed in the work of Gibbs and Tucker [8], which refined Curriculum 78 [1] by taking into account newly published recommendations for CS1 and CS2 [9, 10], and basing curricular decisions on fundamental liberal arts principles and realities. For several years, it served as virtually the only comprehensive and timely model for a four-year computer science curriculum, and consequently the 1986 Model Curriculum had a considerable impact upon computer science offerings on many campuses. Today, literally dozens of schools have a curricular structure based on those original 1986 recommendations.

To complete this review of existing curricular recommendations, we briefly mention the current accreditation standards of the Computing Sciences Accreditation Board (CSAB) [4]. In general, CSAB

requirements are based upon the notion of a concentrated focus within the undergraduate curriculum. For example, CSAB guidelines mandate that an undergraduate program must devote a minimum of 2 and 1/3 years of study (about 58%) to computer science and required mathematics and laboratory science courses. In contrast, a liberal arts education emphasizes breadth as well as reasonable depth in a major and encourages a student to complete a second major or minor. Thus, a liberal arts computer science major typically requires no more than 40%–45% of a student's undergraduate program, including supporting mathematics and science courses. Since the principles motivating CSAB requirements and a liberal arts education are quite different, CSAB requirements will not be considered relevant to the recommendations presented in this article.

Proposed Revisions to the 1986 Model Curriculum

EVEN TODAY, 10 YEARS AFTER ITS RELEASE, MANY ELEMENTS of the 1986 Model Curriculum [8] are still highly appropriate: 1) the division of courses into levels (introductory, core, and electives) to provide a hierarchical framework for the overall curriculum; 2) the size of the major program, which represents an appropriate balance between a computer science major and the requirements of other liberal arts courses; 3) the introduction of mathematics courses at an early stage, which allows adequate depth and sophistication in succeeding courses; and 4) the core program, which still identifies the fundamental areas of the discipline: computer systems, algorithms, computability, and programming languages.

However, in the 10 years since the appearance of that original 1986 report, the discipline has changed greatly. A number of important curricular improvements and enhancements are possible based upon these advances in the discipline and new teaching techniques. For example, many sources [2, 7, 11] advocate the incorporation of formal laboratories into all first-year courses. Similarly, both "Computing as a Discipline" [7] and Computing Curricula 1991 [2] observe that theory is best considered as a recurring theme that touches many courses, rather than viewed as a single course with little effect on the remainder of the curriculum. The growing concern over the use and misuse of computers argues for a greater treatment of social and ethical issues and an early discussion of professional standards and conduct. Such experiences, insights, and developments have led to the following major changes to the 1986 Model Curriculum:

1. Recognition of the importance of different language and problem-solving paradigms and their early inclusion into the curriculum;
2. A significant increase in the coverage of parallelism and distributed systems in the intermediate and core courses;
3. Addition of modern topics drawn from the areas of operating systems and architecture into introductory and core courses;
4. The addition of required formal laboratories into the first two computer science courses;
5. The integration of theory as a recurring theme throughout the core;

The Revision Process

This article reflects the ongoing discussions of an active working group, the Liberal Arts Computer Science Consortium. Initially funded by a grant from the Sloan Foundation, the group's first major product was the 1986 article, "A Model Curriculum for a Liberal Arts Degree in Computer Science," by Gibbs and Tucker [8]. Our annual summer meetings have led to a range of papers and presentations covering such topics as service courses, formal laboratories, experiments involving a breadth-first emphasis in the first courses, and goals for the first two years of undergraduate computer science.

The following members of the Liberal Arts Computer Science Consortium have been active in discussing this Revised Model Curriculum, contributing ideas, and making many suggestions:

Nancy Baxter, Dickinson College; James Bradley, Calvin College; Kim Bruce, Williams College; Robert Cupper, Allegheny College; Scot Drysdale, Dartmouth College; Stuart Hirshfield, Hamilton College; Michael Jipping, Hope College; Charles Kelemen, Swarthmore College; Christopher Nevison, Colgate University; Robert Noonan, College of William and Mary; Richard Salter, Oberlin College; G. Michael Schneider, Macalester College; Greg Scragg, SUNY at Geneseo; Allen Tucker, Bowdoin College; Henry M. Walker, Grinnell College; Tom Whaley, Washington and Lee University

After circulating drafts of this proposal in the spring of 1994, the Consortium met at Swarthmore College in June, 1994 for intensive discussions and an extensive reworking of the topics and structure of the proposed curriculum. Subsequent drafts were circulated widely in the fall of 1994, and helpful responses were received from faculty at several other liberal arts colleges. We want to thank all of those involved in helping to prepare this article.

6. An increase in the number of mathematics courses required for completion of the program;
7. The explicit inclusion of social and ethical issues into introductory and intermediate-level courses, and;
8. The addition of a required senior project.

The next section of the report will describe in detail the courses in this revised curriculum. However, the notion of a "course" may vary from one institution to another. For example, courses may be designated as 3 credits or 4 credits, or all courses may be treated as equal. Our recommendations are based on a system in which a student typically takes four courses per semester. Graduation then is based upon the completion of eight full semesters or 32 courses. In this context, one course is considered to be about one-fourth of a student's load over a semester or about 1/32nd of the graduation requirement.

Four Main Curriculum Levels

ONE OF THE MOST important characteristics of any curriculum is hierarchy—a sequence of courses that build on each other. Each layer of the hierarchy expands and enriches ideas presented in earlier courses, increasing student understanding of the material.

The 1986 Model Curriculum was based on the idea of a three-layer curriculum, using layers called Introductory, Core, and Electives. This revision expands and refines these layers in several ways. First, a new level of computer science courses is inserted between the introductory and core levels so that elements of computer architecture, organization, networks, and systems are introduced relatively early and can be assumed in later core courses. Second, mathematics is introduced as early as possible, and additional mathematics courses are included so suitable concepts and techniques can be presented in both the core and elective courses. Third, a research or development project is included as a required component of at least one elective within each student's major. The overall structure of the Revised Model Curriculum is shown in Table 2.

The following sections describe the structure of each layer of this four-level curriculum.

Introductory and Intermediate Levels (Levels 1 and 2)

The revised introductory course sequence (level 1) is composed of two courses, referred to as CS 1 and MA 1. They introduce fundamental concepts of computer science and discrete mathematics and give students a broad overview of the discipline. Subsequent courses, referred to as CS 2 and CS 3 (level 2), build upon this introduction to provide a deeper understanding of computer science and lay the foundations for further study.

The presentation of this basic material could follow two distinctly different paths—the traditional model or the breadth-first model. In the traditional model, the four courses CS 1, CS 2, CS 3, and MA 1 are viewed as separate and distinct entities, each of which addresses a single coherent topic, (e.g., programming, data structures, computer organization, and discrete mathematics). This is the model used in the 1986 curriculum and the majority of schools today. In the breadth-first model, the topics are more intertwined, and the four courses would not be seen as distinct classes but as an integrated introduction to the discipline extending over three or four semesters [12].

Table 2. The Revised Model Curriculum—levels and courses

Required:
9 Computer science – 1 introductory, 2 intermediate, 3 core, 3 electives
3 Required mathematics courses
12 Required courses total

Organization:

Level 1. Introductory Level
- CS 1: Computer Science I (with formal laboratory)
- MA 1: Discrete Mathematics

Level 2. Intermediate Level
- CS 2: Computer Science II (with formal laboratory)
- CS 3: Computer Organization and Architecture

Integral Mathematics Component
- Two courses selected by the student and advisor to support core courses and electives

Level 3. Core Level
- CO 1: Sequential and Parallel Algorithms
- CO 2: Foundations of Computing
- CO 3: Programming Languages and Systems

Level 4. Computer Science Electives and Project
- Three computer science elective courses building upon the core
- An independent research or development project with library, written, and oral components

CS 1: Computer Science I/CS 2 Computer Science II

Regardless of which of these two approaches is taken, the first two computer science courses should cover the following topics:

- Algorithms and algorithmic problem solving
- Two fundamental problem-solving paradigms chosen from: procedural, object-oriented or functional Examples:

  - –a procedural language (C, Pascal) in CS 1 and an object-oriented model (C++, Object Pascal, Delphi, Smalltalk, Turing) in CS 2;
  - –a functional language (Scheme, CLOS, ML, Miranda) in CS 1 and either a procedural or object-oriented language for CS 2;
  - –a hybrid language (C++, CLOS) to integrate the coverage of two models (object-oriented/procedural, functional/object oriented) over two semesters
- Control structures, recursion and iteration
- Classical computer science algorithms: (e.g., sorting, searching, pattern matching)
- Time and space analysis, including O- and/or Ø notation
- Concepts of modularity, decomposition, procedural and data abstraction, program libraries
- The formal specification of abstract data types
- Major data structures: arrays, records, stacks, queues, lists, simple trees and examples of their uses in computer science
- Software development, including program specifications, design, coding, debugging, testing, verification, and documentation
- A brief introduction to the internal representation of information
- Social issues: intellectual property, liability, privacy, ethical behavior, access

In the traditional approach, this range of topics would be introduced using the courses called Computer Science I (CS 1) and Computer Science II (CS 2) described in [8–10]. In this plan, CS 1 is offered at the introductory level, while CS 2 has CS 1 as a prerequisite and therefore is at a somewhat higher, intermediate level.

While there is a good deal of similarity between the 1986 Model Curriculum and this introductory sequence, these new proposals also reflect important new directions in computer science education that have occurred since 1986. A fundamental question is which language models to use—the 1986 curriculum made the following statement regarding language issues:

To do justice to the concept of procedural and data abstraction, it would be helpful for students to study a language that directly supports this concept, e.g., Modula-2, CLU, or Ada. Thus, it may be necessary [in CS 2] to introduce a different language than the one learned in CS 1 [8].

Even in 1986, the designers of the Model Curriculum realized that it may be necessary to consider a set of languages, rather than one, to meet the needs of the introductory sequence. However, rather than viewing this as a desirable option, it now seems mandatory that the courses CS 1 and CS 2 introduce at least two of the fundamental problem-solving paradigms currently in use in computer science—procedural, object-oriented, and functional. By introducing at least two different world views and models of thinking early in the curriculum, students will be less likely to develop "tunnel vision," the belief that there is only a single model of computation, a single way of doing things. They will be exposed to alternative paradigms, each with its own unique strengths and weaknesses.

The second difference between the current proposals and the 1986 Model Curriculum is the explicit requirement of formal laboratories for CS 1 and CS 2. In the last few years, it has become widely accepted that computer science is both a theoretical and an empirical discipline based on formal lecture material as well as experimental laboratory work. Therefore, there must be a formal laboratory component associated with both CS 1 and CS 2. (Laboratories for other courses in the computer science curriculum are certainly possible and highly beneficial, but due to staffing limitations, they are not explicitly required.) These regularly scheduled laboratory sessions should be staffed by faculty or teaching assistants, meet weekly for at least 1 hour (longer, if possible), have a prearranged sequence of activities to be carried out by the student, and require the completion of written exercises. They should not be limited to skills acquisition labs (e.g., learning to use a debugger or editor), but should be based on the principles of the scientific method, including experimentation, observation, data collection, data analysis, and hypothesis confirmation or rejection. For a discussion of the format and structure of introductory computer science laboratories along with their benefits to the student, see [2, 11].

In addition to the introductory and intermediate computer science courses, students must also develop their reasoning abilities and mathematical backgrounds.

The third difference is the inclusion of some theoretical topics in both CS 1 and CS 2, such as the time/space complexity of algorithms and the formal specification of abstract data types. As mentioned earlier, it is important that formal ideas drawn from mathematics and logic not be compartmentalized into a one-course "theory ghetto." Wherever possible, these concepts should be introduced, even if only briefly,

and made a part of the introductory and intermediate levels of the curriculum. Students should appreciate early on that mathematical ideas such as proof, counting, and formal models are not simply academic exercises but an essential part of the discipline.

Finally, we have included in the introductory sequence an explicit discussion of ethical and social issues. For many students, CS 1 and CS 2 are their only computer science courses. Topics such as ownership of intellectual property, constitutional right of privacy, ethical behavior, and professional standards can influence the behavior of students during class and beyond and can help identify some of the social consequences of computing.

MA 1: Discrete Mathematics

Computer science draws upon mathematics in many ways, and computer science students need to gain a mastery of specific topics as well as a general level of mathematical insight and maturity. Unfortunately, such mastery and maturity require time to develop. Further, some mathematical analysis is appropriate in computer science courses from the very beginning (e.g., CS 1). Thus, as a practical matter, some mathematical material should be included as a part of introductory computer science courses, regardless of what mathematics courses students can take. For example, an introduction to conditional statements in CS 1 could include a discussion of Boolean expressions and Boolean manipulations, such as DeMorgan's Laws.

While the inclusion of mathematical material into early computer science courses can be an effective mechanism for developing mathematical maturity, a computer science program should also ensure that students cover the fundamental concepts of discrete mathematics as early as possible, so that these ideas and techniques can be used in subsequent courses. In many colleges and universities, this material will be covered in a separate discrete mathematics course, which should include the following topics:

- Sets
- Functions
- Relations, including partial order
- Methods of propositional logic
- Introduction to predicate logic
- Counting
- Recurrence relations
- Asymptotic analysis
- Proof, including induction
- Introduction to probability
- Graphs

CS 3: Computer Organization and Architecture

While students in CS 2 work with high-level abstractions such as objects, functions, and abstract data types, there is also the need to expose students at an early stage to lower-level abstractions such as digital logic, machine language, computer architecture, data representations, and elements of distributed systems. CS 3 provides this perspective as it covers the following topics:

- Introduction to digital logic and/or microcode
- Conventional Von Neumann architecture
- The internal representation of information
- Instruction formats and addressing, instruction sets, machine and assembly language programming
- The fetch/execute model of computation
- Alternative architectures: RISC/CISC, SIMD/MIMD parallel
- Memory architecture and algorithms: cache, virtual memory, paging, segmentation
- I/O architecture: interrupts, memory-mapped I/O
- Overview of distributed systems, multiprocessors, networks, parallel systems

Overall, CS 3 provides low-level hardware and system models that complement the high-level abstractions discussed in CS 2.

Integral Mathematics Component

In addition to the introductory and intermediate computer science courses described in the previous section, students must also develop their reasoning abilities and mathematical backgrounds. Thus, the 1986 Model Curriculum recommended a two-course mathematics requirement—a one-semester Discrete Mathematics course and a second course chosen from a set that included a second discrete mathematics course, linear algebra, and calculus. While this recommendation increased the students' mathematical foundation, it did not allow students the opportunity to fully develop the reasoning skills associated with mathematical maturity.

Thus, these revised recommendations specify *three* required mathematics courses, two beyond Discrete Mathematics. This change will begin to provide appropriate background for the core courses and/or electives and help to develop reasoning skills and mathematical maturity. Since different areas of computer science draw upon different mathematical topics, and since students have widely varying interests, it is unrealistic to believe a single requirement will fill the needs of every student. Instead, the Revised Model Curriculum suggests specific courses be selected by students in consultation with an advisor and with the intent of providing appropriate background for later work. For example, a linear algebra course might be appropriate for students planning to take a course in graphics. Other likely mathematics course options might include Calculus I and II, Discrete Mathematics

II, Probability and Statistics, and Mathematical Modeling.

The Core Courses (Level 3)

The introductory and intermediate courses just described address wide-ranging themes that affect all of computer science—themes such as algorithms, problem solving, data types, proofs, abstractions, and the Von Neumann architecture. The purpose of these initial courses is to give the student a broad overview of the central concepts of computer science, mathematics, and logic.

The third level of the curriculum, called the core, focuses on individual subareas of the discipline and demonstrates to the student that computer science, like any scientific discipline, is divided into separate and distinct subareas. The core is composed of the following three courses, which can be taken in any order: CO 1. Sequential and Parallel Algorithms; CO 2. Foundations of Computing; CO 3. Programming Languages and Systems. The following paragraphs describe each of these redesigned core courses.

CO 1: Sequential and Parallel Algorithms

This course examines the design and efficiency of algorithms from both sequential and parallel perspectives. It also investigates problem-solving strategies and the relative difficulty of various classes of problems and problem-solving techniques. Possible topics include the following:

- Algorithmic paradigms, including divide and conquer, greedy methods, dynamic programming
- Advanced data structures not covered in CS 2, such as heaps, hashing, or height-balanced trees
- Graph algorithms—minimum spanning trees, shortest path, connected components
- Proofs of algorithm correctness
- Complexity analysis of both parallel and recursive algorithms
- Parallel algorithms—parallel sorting and searching, reduction, parallel prefix
- Algorithms for numerical problem solving, error analysis, stability
- Algorithms associated with process synchronization
- Algorithms concerning deadlock: its avoidance, detection, and resolution

As time permits, additional algorithms from a range of problem domains can be presented to illustrate the use of various problem-solving approaches. For example, an instructor may wish to introduce and discuss algorithms from areas such as computational geometry or scientific visualization. It is worth emphasizing the inclusion of algorithms drawn from the areas of computer architecture and systems (e.g., deadlock and process synchronization). It is important for students to see connections between courses and the relationships between courses. By using algorithms drawn from the area of systems, this will demonstrate the important relationship between a study of algorithms (CO 1) and computer organization (CS 3).

CO 2: Foundations of Computing

Mathematical concepts recur throughout the Revised Model Curriculum, and introductory and intermediate courses discuss topics such as the complexity analysis of algorithms, Boolean logic, the use of recurrence relations to analyze recursive algorithms, and the formal description of abstract data types. This core course builds upon those ideas, demonstrating the logical and mathematical foundations of computer science and providing a context for this theory by using the ideas in some important applications, as shown in the following list:

- *Models of computation:* finite and pushdown automata, nondeterminism, recursive functions, regular expressions
- *Grammars and parsing:* language syntax, context-free grammars, BNF, parsing; short overview of language processors, compilers, interpreters
- *Solvable and unsolvable problems:* Turing machines; Church's thesis and universal Turing machines; the halting problem, unsolvability
- *P and NP complexity classes:* the classes P and NP; NP-complete problems and intractable problems; approximate, nonoptimal solutions to NP problems

Theoretical models and their practical applications should be thoroughly intertwined throughout this course. For example, after discussing various types of automata, one could discuss the concepts of a simple lexical parser. A discussion of context-free grammars and pushdown automata could lead to an explanation of table-driven parser generators. With these applications, there may be time only for the inclusion of selected proofs related to Turing machines and Church's thesis.

CO 3: Programming Languages and Systems

The third core course explores computer languages and system environments from multiple perspectives, including connections with problem-solving paradigms, language design, implementation, and capabilities for parallelism. The course demonstrates the application of formal methods to programming languages through discussions of formal semantics and verification. As several language models have already been covered in CS 1 and CS 2, significantly less time needs to be spent in this new course on the teaching of

language paradigms than in previous language courses; correspondingly, much more time should be available to address issues of design and implementation. Course topics may be divided into the following major groups:

- *Language paradigms:* procedural, functional, object-oriented, logic
- *Language design and implementation issues:* language representations, data structures, control structures; binding, the run-time environment; language support for modularity, information hiding, exception handling, encapsulation; type issues—static and dynamic types, polymorphism, type inference
- *Language issues related to parallelism:* language mechanisms for parallel architectures; processes, process management, interprocess communications; message passing, shared memory
- *Proving properties of programs:* formal semantics (axiomatic, operational, and/or denotational); program proofs

To summarize, the Revised Model Curriculum utilizes six computer science courses plus discrete mathematics to introduce students to the breadth of computer science and to cover the core areas of the discipline. It also specifies two additional mathematics courses chosen to support advanced course work.

Relationship of the Introductory and Core Sequence to Curricula 1991

THE COURSES RECOMMENDED FOR THE FIRST THREE levels cover many of the areas suggested in Computing Curricula 1991 [2], although some differences in emphasis may be noted. Overall, the core of Computing Curricula 1991 includes 283 hours of lecture, with some additional, but unspecified, time devoted to laboratories. This proposal recommends six computer science courses at the introductory, intermediate, or core level. Since formal course work normally will include lectures, discussions, and laboratories, estimates of how much time will be devoted to lecture during each course must be very approximate. As a conservative estimate, one might suppose that a typical course meets three times a week for a 14-week semester, and each of these recommended courses would total 42 hours. Allowing a few hours for tests or reviews, each course would contain approximately 40 hours of lecture, and six courses would involve a total of 240 hours. Thus, in terms of size, the required courses in this proposal involve about 43 fewer hours of lecture than Computing Curricula 1991, a difference of about 15%.

Beyond this difference in required contact hours, there are also significant differences in content. Table 1 shows the approximate number of hours spent in each of the 11 topic areas specified in Computing Curricula 1991 and compares this total against the recommended number of hours in that report. The assignment of lecture hours to topics is based on the course descriptions given in this article. Specific numbers in the table should be treated as tentative and approximate, as any such tabulation depends on many subjective choices regarding course design. The numbers do allow, however, a rough comparison.

While specific numbers in this table should be treated as approximate, these values suggest a significant difference in emphases between the two programs. Specifically, these liberal arts recommendations clearly emphasize the topics of algorithms and data structures with 76 hours spent on these topics compared to 47 in Computing Curricula 1991. Further, even with fewer overall hours of lecture required in this proposal, the courses required here roughly meet the goals prescribed by Computing Curricula 1991 in programming languages, operating systems, numerical and symbolic computation, and social issues. This reflects the liberal arts emphasis on fundamental concepts and the pervasive role of theory throughout the curriculum. On the other hand, there is significantly less material in such applied areas as architecture, software engineering, robotics, and database systems, although such topics could be covered in more depth in electives.

Electives and the Senior Project (Level 4)

The fourth level of the Revised Model Curriculum is the computer science electives. The purpose of these advanced electives is threefold:

1. Introduce students to specific areas of research and development within computer science;
2. Show how ideas presented in the introductory, intermediate, and core sequences can be used to solve important real-world problems in computer science; and
3. Involve seniors in a research or development project.

POINT NUMBER 2 IS PARTICULARLY IMPORTANT AND IS the reason why the first three levels of the curriculum should be completed (or nearly completed) prior to beginning the electives. Courses like graphics, networks, and artificial intelligence are typically viewed by students as the "fun" courses, and students usually want to enroll in these courses as soon as possible, frequently well before they have completed the core. This should be discouraged as the elective courses build upon the important concepts studied in the first six computer science courses.

For example, an elective in the area of computer networks would likely draw upon hardware topics from

CS 3, process synchronization algorithms introduced in CO 1, and graph algorithms (for routing) studied in MA 1. Similarly, a compiler course would utilize the concepts of regular expressions and finite state machines presented in CO 2 as well as the language design and implementation concepts discussed in both CS 3, CO 2, and CO 3. Thus, the student must resist the urge to hurry into the elective level without first building the base of ideas and knowledge on which these elective courses will rest.

The specific electives included in a program will, of course, depend on many factors, including faculty interests and the size of the student population. An alphabetized list of electives that might appear in the catalogs of typical liberal arts computer science programs follows:

EL 1 Advanced Algorithms
EL 2 Artificial Intelligence
EL 3 Compiler Design
EL 4 Computer Architecture
EL 5 Database Principles
EL 6 Formal Semantics
EL 7 Graphics
EL 8 Natural Language Understanding
EL 9 Networks
EL 10 Numerical Methods
EL 11 Operating Systems
EL 12 Parallel Processing
EL 13 Simulation
EL 14 Social and Ethical Issues of Computing
EL 15 Software Engineering
EL 16 Theory of Computation
EL 17 Topics in Computer Science

Since one of the hallmarks of a liberal arts curriculum is interdisciplinary study, it is also possible, and even beneficial, to have courses offered in other departments that are of interest to computer science students and that meet part of the elective requirement. Examples of such courses might include:

- Physics: Hardware design, logic design
- Linguistics: Computational linguistics
- Business: Management information systems, systems analysis
- Education: computer-aided instruction, computers in education

WHILE STUDENTS SHOULD TAKE ENOUGH COURSES TO gain knowledge in advanced areas of computer science, major requirements must fit within the bounds prescribed by a liberal arts education. Thus, these proposed recommendations specify that a computer science major must complete a minimum of three electives chosen from the preceding list or from other approved courses available at their institution. However, as schedules permit, majors are strongly encouraged to take an additional course or two at this level if it can be accommodated within the major program and the rules of the college.

Regardless of which electives are offered, the curriculum should be arranged so that each student will work on a significant project during his or her senior year. This project should allow seniors to participate either in a one-on-one scholarly endeavor with a faculty member or in a small, possibly interdisciplinary, team involving both students and faculty. This experience could last either one semester or possibly a full year, depending on the number of graduates, faculty, and resources available. While this work could be done independently with a faculty member, it is expected that for the majority of students this experience will be included as part of an elective course. While the nature of the project will vary from school to school, at a minimum it should involve these three important components:

1. A review of one or more scholarly papers from the primary literature of computer science;
2. The writing of a significant scientific paper or substantial technical document to give the student experience in writing for a scientific audience; and
3. An oral presentation to students and/or faculty.

Additional details about the senior project will depend on both the interests and capabilities of the student and the time available to both students and faculty. Therefore, there must be flexibility and options on how this requirement can be fulfilled in order to meet the range of abilities and interests of the graduates. Some possibilities include:

- An actual research project involving a one-on-one research experience between the student and a faculty member lasting either 1 or 2 semesters. At many schools, this experience is called an Honors Project. This experience also may be integrated into an elective course at the senior level.
- A group software-engineering project involving a team of students specifying, designing, implementing and documenting a substantial software project over a one- or two-semester time frame.
- A special "senior research course" required of all graduating seniors. For example, this could be a course looking at important emerging areas of research within the discipline.
- A review of the literature within a specific area and the writing of a scholarly paper summarizing one's findings as part of an elective course.

Table 3. Typical schedules for students completing a computer science major

	Schedule for a Student Starting Computer Science in the First Year		Schedule for a Student Starting Computer Science in the Second Year	
	Fall	*Spring*	*Fall*	*Spring*
First Year	CS 1 MA 1	CS 2		
Second Year	CS 3 Math 2	Core 1 Math 3	CS 1 MA 1	CS 2 Math 2
Third Year	Core 2	Core 3 Elective 1	CS 3 Math 3	Core 1 Core 2
Fourth Year	Elective 2 (project)	Elective 3 (project)	Core 3 Elective 1	Elective 2 Elective 3 (project)

All these models could be reasonable implementations of a senior project experience. The key concern is whether or not they include the three required characteristics previously mentioned: exposure to the primary research literature of computer science, the writing of a scholarly technical paper, and an oral presentation of findings and results to one's peers.

This required senior project experience represents an important and unique change from the original 1986 proposal. Completing a major project can be extremely helpful for undergraduates in computer science and other fields. Expanded research opportunities for undergraduates are vital, so that they, like graduate students, can experience the excitement (and frustrations) of scholarly inquiry and the scientific method. Students should be exposed to the literature of computer science, and they should practice both oral and written technical communication. A senior project can provide these important learning experiences. Liberal arts colleges, with their smaller number of graduating seniors and their emphasis on oral and written communications, are in an excellent position to implement this important curricular initiative [5, 6].

In total, this revised model curriculum requires a total of 9 computer science courses (1 introductory, 2 intermediate, 3 core, 3 electives), and 3 mathematics courses, a total of 12 required courses. Assuming that students take 32 courses during their college studies, then the recommended 12 course major represents 38% of the total course load—certainly within the limits of most liberal arts colleges. If one or two additional laboratory science courses are added as part of the general distribution requirement, the total undergraduate computer science degree program will contain 13 or 14 computer science, mathematics or science courses out of the possible 32.

Possible Course Structures and Schedules

There are 12 required courses for a computer science major following this revised model curriculum. The following discussion shows that these recommendations can be completed by students within a normal four-year program and can be implemented with the limited number of faculty available at many liberal arts colleges.

The sample programs shown in Table 3 demonstrate how students can complete the required courses within the normal four-year college period (in fact, two possible schedules are given). The first chart shows a schedule for the student who begins work in computer science immediately upon entering college. At liberal arts colleges, however, many students do not begin as computer science majors but switch into the field from other areas during their first or second year. The second chart indicates that scheduling is flexible enough to permit students to start the program in their second year and still complete it within a four-year time span. The project is listed in parentheses in each semester of the senior year, but it may be taken with any elective. Further, if an elective were taken in the third year, then one elective could be taken each semester of the fourth year, and the senior project could extend for a full year.

Table 4. A possible staffing schedule

Annual Number of Sections	Course	Comments
2	CS 1	(1 section each semester)
2	CS 2	(1 section each semester)
1	CS 3	(1 section each year)
3	Core courses	(Each core course offered annually)
4	Elective courses	(Four electives per year)
0–1	Project	(In addition to teaching electives, some faculty members may receive teaching credit for guiding student research)
Total 12–13		

Staffing of the proposed four-

level curriculum is possible under only modest assumptions. For example, assuming that the Mathematics department teaches both Discrete Mathematics and the two subsequent mathematics courses, then a modest but fully reasonable schedule might offer CS 1 and CS 2 every semester, CS 3 and all three core courses once each year, and four electives per year, two each semester. Given these assumptions, Table 4 shows the teaching load needed to staff the program.

Overall, this minimal, but reasonable, schedule requires the staffing of only 12 to 13 computer science sections each year. Details concerning how these courses are scheduled among faculty will depend upon such institutional policies as teaching loads, faculty size, and the number of sections of service courses for non-majors. Certainly additional sections would enhance the curriculum by offering greater scheduling flexibility, more elective courses, and greater research opportunities. However, Table 4 shows that, with a relatively modest number of sections, a small liberal arts computer science faculty can offer a rigorous and high-quality undergraduate program in computer science.

Conclusion

Computer science is a young discipline, and its curriculum is growing, changing, and evolving in response to new developments and new ideas. The four-level curriculum described here (a revision of [8]), consisting of 9 computer science and 3 mathematics courses, allows a liberal arts college to maintain the strengths of the liberal arts environment while at the same time providing sufficient depth and rigor within the discipline. In addition to an emphasis on the fundamental principles of computer science, the curriculum includes expanded mathematical study, exposure to the empirical aspects of computer science via formal laboratories, and—most importantly—gives all students a chance to experience the excitement of scientific research and inquiry. Taken together, these characteristics make this a strong and rigorous program of study that can be implemented with the resources and faculty available today at most liberal arts colleges.

References

1. ACM Curriculum Committee on Computer Science. Curriculum 78: Recommendations for the undergraduate program in computer science. *Commun. ACM 22*, 3 (Mar. 1979), 151–197.
2. ACM/IEEE-CS Joint Curriculum Task Force. Computing Curricula 1991. *ACM/IEEE-CS Joint Curriculum Task Force Report.* ACM Press and IEEE Computer Society Press, 1991.
3. Clancy, M.J. and Linn, M.C. *The case for case studies of programming problems.* Presented at the annual meeting of the American Educational Research Association (San Francisco, Calif., Mar. 1989).
4. Computing Sciences Accreditation Board. Criteria for accrediting programs in computer science in the United States. *1993 Annual Report of the Computing Sciences Accreditation Board*, Stamford, CT, 26–41.
5. Cupper, R. *The required senior thesis and a topics and research methods junior seminar course.* Presented at the 1991 Liberal Arts Computer Science Consortium Meeting (Grinnell College, June 1991).
6. Cupper, R. *Undergraduate research in computer science at Allegheny College.* Presented at the 1993 Liberal Arts Computer Science Consortium Meeting (Washington and Lee University, June 1993).
7. Denning, P.J., Comer, D.E., Gries, D., Mulder, M.C., Tucker, A.B., Turner, A.J., and Young, P.R. Computing as a discipline. *Commun. ACM 32*, 1 (Jan. 1989), 9–23.
8. Gibbs, N. and Tucker, A. Model curriculum for a liberal arts degree in computer science. *Commun. ACM 29*, 3 (Mar. 1986), 202–210.
9. Koffman, E.P., Stemple, D. and Wardle, C.E. Recommended curriculum for CS1: 1984, A report of the ACM curriculum task force for CS1. *Commun. ACM 27*, 10 (Oct. 1984), 998–1001.
10. Koffman, E.P., Stemple, D. and Wardle, C.E. Recommended curriculum for CS2: 1984, A report of the ACM curriculum task force for CS2. *Commun. ACM 28*, 8 (Aug. 1985), 815–818.
11. Parker, J., Cupper, R., Kelemen, C., Molnar, D., Scragg, G. Laboratories in the computer science curriculum. *J. Comput. Sci. Education* (1990), 205–221.
12. Tucker, A. and Garnick, D. A breadth-first introductory curriculum in computing. *Comput. Sci. Education 2*, 3 (Fall 1991).

HENRY M. WALKER (walker@math.grin.edu) is a professor of Mathematics and Computer Science at Grinnell College in Grinnell, Iowa.

G. MICHAEL SCHNEIDER (schneider@macalester.edu) is a professor of Mathematics and Computer Science at Macalester College in St. Paul, Minnesota.