2-25-2007

A 2007 Model Curriculum For A Liberal Arts Degree In Computer Science

Liberal Arts Computer Science Consortium

Charles F. Kelemen
Swarthmore College, cfk@swarthmore.edu

Recommended Citation

Liberal Arts Computer Science Consortium and Charles F. Kelemen. (2007). "A 2007 Model Curriculum For A Liberal Arts Degree In Computer Science". *Journal On Educational Resources In Computing.* Volume 7, Issue 2. DOI: 10.1145/1240200.1240202
https://works.swarthmore.edu/fac-comp-sci/20

A 2007 Model Curriculum for a Liberal Arts Degree in Computer Science

Liberal Arts Computer Science Consortium

February 25, 2007

In 1986, guidelines for a computer science major degree program offered in the context of the liberal arts were developed by the Liberal Arts Computer Science Consortium (LACS) [4]. In 1996 the same group offered a revised curriculum reflecting advances in the discipline, the accompanying technology, and teaching pedagogy [6]. In each case, the LACS models represented, at least in part, a response to the recommendations of the ACM/IEEE-CS [1][2]. Continuing change in the discipline, technology, and pedagogy coupled with the appearance of Computing Curriculum 2001 [3] have led to the 2007 Model Curriculum described here.

This report begins by considering just what computer science is and what goals are appropriate for the study of computer science in the landscape of the liberal arts. A curricular model for this setting follows, updating the 1996 revision. As in previous LACS curricula, [4] and [6], the model is practical; that is, students can schedule it, it can be taught with a relatively small size faculty, and it contributes to the foundation of an excellent liberal arts education. Finally, this 2007 Model Curriculum is compared with the recommendations of CC2001 [3].

1 The Discipline of Computer Science

Refining the definition given in 1986 [4], computer science is the study of algorithms and data structures: their creation, analysis, and realization. In particular, computer science is the study of algorithms and data structures with respect to their

1. formal properties,
2. linguistic realizations,
3. hardware realizations, and
4. applications.

The curriculum for a program in *computer science* as described here follows from a specific ordering of emphasis among these four components. The formal properties (1) of algorithms and data structures are emphasized over specific languages (2), machines (3), and applications (4).

This definition has held up well even though the discipline of computer science has evolved substantially. Further refinements might add problem solving to algorithms and data structures, a consideration of social and ethical implications, and the study of what is and what is not possible in the context of algorithmic problem solving. However, even with these refinements, computer science, particularly in a liberal arts setting, emphasizes principles and techniques over operational and syntactic details that change rapidly.

2 The Liberal Arts Landscape

Liberal arts programs in computer science generally emphasize multiple perspectives of problem solving (from computer science and other disciplines), theoretical results and their applications, breadth of study, and skills in communication. In addition to the material content of computer science, the algorithmic approach is a very general and powerful method of organizing, synthesizing, and analyzing information. Three general-purpose capabilities that are among those fundamental to a liberal arts education are the ability to organize and synthesize ideas, the ability to reason in a logical manner and solve problems, and the ability to communicate ideas to others. The design, expression, and analysis of algorithms and data structures utilizes and contributes significantly to the development of all three capabilities.

Rather than consider undergraduate programs as professional degrees, liberal arts programs embrace the premise that graduates working in areas related to their majors will find that their careers develop in unexpected ways – often involving new areas of knowledge and application. Further, many are likely to change professions multiple times over their working lives. Liberal arts programs de-emphasize specific technical details, with the understanding that the half-life of such technical knowledge is now estimated at only about 2.5 years [7]. This environment leads to a balance of courses perhaps quite different from other types of programs. A typical liberal arts curriculum in computer science consists of:

- Computer science courses: about 30%
- Mathematics courses: about 10%
- Other science courses: 5 - 10 %
- Non-science (e.g., humanities, social science) courses: 50 - 55%

Overall, a computer science program in a liberal arts setting typically requires about 40% of a student's undergraduate program (including math). While this balance places limits on the number of computer science courses (e.g., only 8 - 12 required 4-credit courses for the major), core topics can be covered well, and students have considerable contact with non-science courses that promote breadth, communication, teamwork, and multiple perspectives. Computing courses themselves often integrate views of mathematics, science, and engineering (as described in CC1991 [2]). Together, such breadth integrates in a fundamental way several qualities that have emerged as essential for computing professionals: effectiveness of writing and speaking, group interaction, and understanding client perspectives.

3 Computer Science Programs in the Liberal Arts

This perspective on the discipline and the educational environment leads to the following goals for an undergraduate program in computer science taught in a liberal arts setting.

- To enable understanding the capabilities, limitations, and ramifications (technical, ethical, and social) of computing, the state of the art, and current research and development in computer science and related areas;
- To develop an ability to understand and analyze end user needs, master the techniques of creating and applying algorithms and data structures, and analyze their viability, correctness, and efficiency utilizing analytical methods and appropriate theoretical results;
- To become effective at working individually and in teams, building on the work of others, and to be able to communicate technical information with both experts and non-experts;

- To prepare for adapting to changes in hardware and/or software technologies, and new and changing application areas through a firm grasp of fundamental principles and to develop an appreciation of the need for life-long learning;
- To appreciate both the demands and range of opportunities of the computing profession and provide for and encourage creative contribution to the art.

In particular, students in the type of program described here should:

- Understand multiple views of problem solving (e.g., 2 or 3 of imperative, object-oriented, functional);
- Have experience applying theoretical results to solving practical problems;
- Be able to apply critical thinking and problem solving skills across disciplines;
- Have experience with at least one large, team-based project or research project;
- Understand non-scientific perspectives and have sufficient background to be able to communicate effectively with people with those perspectives.
- Recognize the importance of social and ethical issues in computing.

4 Overview of the 2007 Model Curriculum

As in previous versions of the Model Curriculum, our goal has been to design a rigorous program requiring no more than 9 courses in computer science accompanied by 3 courses in mathematics. This 12-course package represents about the right size for a degree program within the liberal arts. The curriculum must reflect current disciplinary content, appropriate liberal arts themes, modern pedagogy, and realistic staffing levels.

The 2007 Model Curriculum springs from three major motivations:

1. Changes in emphasis between Computing Curricula 1991 and Computing Curricula 2001;
2. A renewed commitment to the inclusion of multiple paradigms for problem solving within the curriculum; and
3. The need to update content details from the previous 1996 Revised Model Curriculum [6].

The new curriculum has a structure not unlike that of previous versions – that is, three basic courses in computer science followed by a set of 4 courses capturing what is the essence or core of the discipline, followed by 3 electives made up of advanced courses for depth and applications courses for breadth, accompanied or followed by a culminating project or thesis. As before, the curriculum requires 3 courses in mathematical foundations.

The most obvious change in this 2007 Model Curriculum is the introduction of a new course in software development. We see a need for a more disciplined approach to software development particularly in light of the glaring inadequacies of today's software products. Covering basic algorithms, the syntax and semantics of a programming language, algorithmic efficiency, and basic data structures leaves little time for anything else. This problem was recognized by the CC2001 committee, which stated in its report [3, pp. 29-30]:

> Although the philosophy and structure of introductory courses have varied widely over the years, one aspect of the CS curriculum has remained surprisingly constant: the length of the introductory sequence... We believe the time is right to question this two-course assumption. The number and complexity of topics that entering students must understand have increased substantially, just as the problems we ask them to solve and the tools they must use have become more sophisticated.

To address this issue, the 2007 Model Curriculum includes a software development course that builds on the introductory sequence. This course provides the additional time needed to properly cover such topics as inheritance, software design, design patterns, unit testing and test coverage, design by contract, and APIs (data collections, networking, GUIs, event-handling).

To make room for this new course, the core course entitled *Foundations of Computing* in the 1996 Model Curriculum [6] was eliminated. Two thirds of the material (automata and computability) has been moved to a new foundations of computing course (FC 2) entitled *Theoretical Foundations of Computer Science*, and the material on grammars moved to the Programming Languages course. So the loss of theory is more apparent than real.

The other big change results from the desire to introduce multiple problem-solving paradigms. Different paradigms, such as object-oriented and functional, provide distinctive ways of thinking about and solving problems. Students who experience multiple approaches and appreciate their tradeoffs understand the value of applying alternative problem-solving models when confronting complex, real-world applications. Two approaches that integrate object-oriented and functional programming are presented in detail in Appendices A and B. If desired, functional programming could be replaced by another sufficiently different paradigm, such as logic programming.

In the *objects-first* approach (Appendix A), a traditional CS 1 and CS 2 sequence (courses CS1A and CS2A) is accompanied by FC 1A, *Discrete Structures and Functional Programming*, that includes an introduction to functional programming (as the title suggests) and a traditional set of topics from discrete mathematics. Functional programming applications are specifically selected for their relevance to the discipline of computing to exemplify selected topics in discrete mathematics and enrich the connections between mathematics and computer science.

Recognizing the constraint that in many liberal arts colleges the discrete mathematics course is taught by the Mathematics Department and that mathematics faculty may be unwilling to teach functional programming even with assistance from their computer science colleagues, a second approach is presented in Appendix B. This *functional-first* approach introduces the discipline of computer science and programming via a functional language and some associated mathematical topics in CS 1B followed by a CS 2B course covering an object-oriented language coupled with data structures. Such an introductory sequence has been used at many schools with great success. The functional-first CS 1B/2B would be coupled with a traditional discrete mathematics course (FC 1B) [6].

The number of computer science core courses remains at four. Three are revised versions of courses familiar from previous versions of the Model Curriculum: *Principles of Algorithm Analysis*, *Principles of Programming Languages*, and *Principles of Computer Organization*. The fourth, *Principles of Software Development*, has been added in recognition of the explosion in the size and complexity of object-oriented languages and in object-oriented design and implementation. This course addresses the need for effective and correct software development, not just the object-oriented paradigm. The course explicates advanced concepts in software design, modern software development techniques, and APIs.

To this computer science core is added a second foundations of computing course (FC 2) entitled *Theoretical Foundations of Computer Science.* This course is similar to the 1996 course entitled *Foundations of Computing* [6], covering automata and computability, with the addition of topics from number theory and probability and elementary statistics. The course could be a prerequisite for such advanced electives as compiler construction.

As before, a concentration in computer science entails taking three advanced electives. Completion of the core ideally precedes advanced electives in the discipline. Finally, the program includes a capstone experience that can range from a significant individual or group software development project to research leading to a senior thesis.

Figure 1 summarizes the structure of this 2007 Model Curriculum. In the figure we do not distinguish between the two approaches to the introductory sequences. A solid line in the figure indicates a prerequisite.

Course FC 1 could have CS 1 either as a prerequisite or a co-requisite, which we indicated by using a dotted line. Note that the capstone (research) experience may be integrated into an elective or met by an additional course.

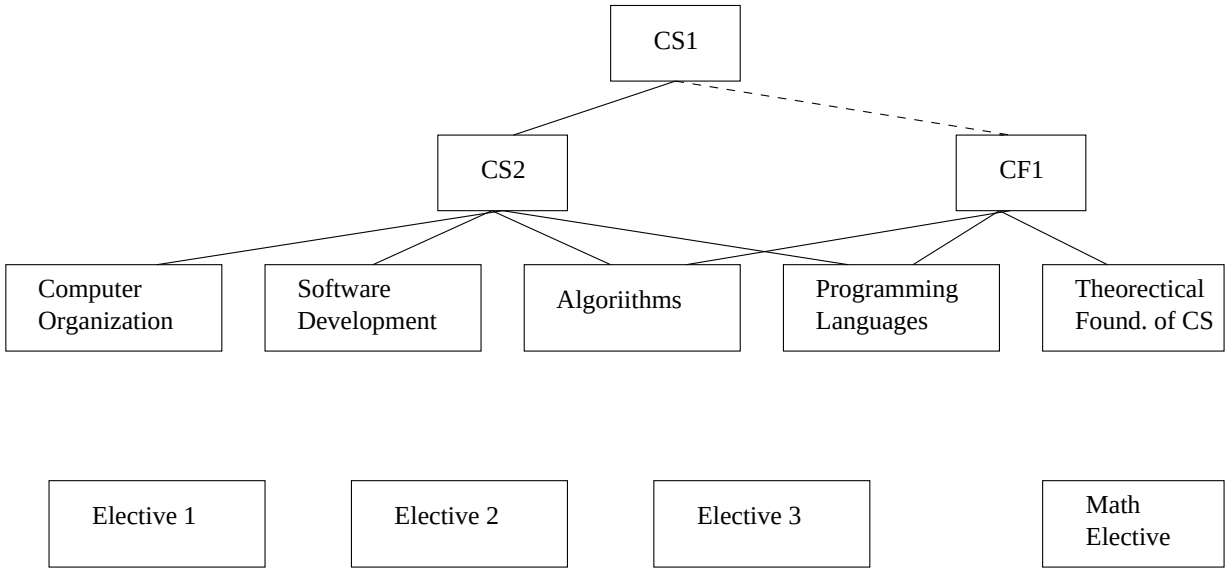

Figure 1: Overview of the Model Curriculum

5 Recurring Themes and Issues

Courses in this 2007 Model Curriculum progress from basic introductory level courses through an intermediate core to advanced electives and undergraduate projects and research. The study of relevant mathematical tools and ideas are integrated with the central issues of computer science, particularly at the introductory and core levels.

Some considerations common to many courses in the curriculum include the following.

Social and Ethical Issues. Many computer science courses, especially the introductory sequence, ask students to confront the social and ethical issues facing the field, as well as the technical issues. Social and ethical concerns should be a theme woven through courses at all levels. The 2007 Model Curriculum recommends that specific social/ethical issues be discussed in connection with related technical issues, to show students that the two are not separable. For example, units on testing or proof should discuss the professional's responsibility to produce correct code; topics such as the Internet or cryptography will be presented as applications of course concepts, and those presentations include discussions of the social impact of the applications on areas such as privacy, accuracy, and security. Tables 1 and 2 later in this report highlight hours spent on social and ethical issues in the introductory sequence and in the core course *Principles of Software Development.*

Schedule. The 2007 Model Curriculum assumes that a course contains about 39 hours of classroom time (this can be achieved in a semester model by meeting for 3 hours per week for 13 weeks, or in a quarter model by meeting for 4 hours per week for 10 weeks). If a four-credit course meets four times a week or three hours of lecture plus a laboratory, then classes might meet for significantly more time in a semester/quarter. If labs are the primary instructional format, then course material is presented through those lab sessions.

Reinforcement. Core topics (e.g., proofs, algorithm analysis, and recursion) should be applied throughout all courses in the curriculum. This ensures that students see that key ideas have applications outside the context in which they are introduced, and allows instructors to reinforce and expand material on an ongoing basis. In general, the course descriptions indicate the amounts of time likely to be spent introducing concepts; continuing reinforcement means that students will in fact spend considerably more time seeing and practicing those concepts. For example, the outlines for FC 1 indicate that logic and proofs are introduced early in a 4-hour unit, but the expectation is that these topics will be applied in class and homework consistently throughout the rest of the semester, as well as in courses such as Algorithms, FC 2, and others. From this perspective, logic and proofs take up 30+ hours of FC 1, plus additional hours elsewhere in the curriculum.

Flexibility. All the course outlines are models to be adapted to local needs. The ordering of topics within each course provides one reasonable ordering for that course, but there are, of course, other pedagogically sound ways to organize the course. One factor in determining a sequence of topics may be the set of motivating examples, labs, and projects instructors choose to use. Another factor would be whether an instructor prefers to cover topics sequentially in depth, or apply a spiral approach in which key concepts are introduced early on and then revisited in more detail later. Even the boundaries between courses are somewhat porous, particularly between CS 1 and CS 2, between CS 2 and Algorithms, and between FC 1 and FC 2.

The times suggested are approximate, and where possible we have left some of a course's nominal 39 hours unused, to allow instructors to supplement material according to local needs and interests.

6 Introductory Courses

The recommendations for introductory courses include a sequence (CS1 and CS2) in programming and data structures, and a course (FC 1) in discrete mathematics topics related to computer science. The details will vary from institution to institution, according to program size, desires or pressures to coordinate majors' introductory courses in either computer science or mathematics with general education offerings, etc. This 2007 Model Curriculum outlines two approaches to the introductory sequence below; others are possible as well.

The need to cover two programming paradigms forces the introductory courses in the 2007 Model Curriculum out of traditional molds. Difficulties caused by this break with tradition can be resolved by distributing treatment of functional programming between the CS and FC sequences. There is a spectrum of possible introductory curricula, anchored at one end by a model in which functional programming is taught solely in the CS courses, and at the other end by a model in which functional programming is taught in the FC 1 course. We describe the first endpoint in Appendix B and the second in A. Many institutions may in fact position themselves between these two extremes.

The similarity in topics covered is shown in Table 1, with only minor differences in coverage between the two approaches. Hence, in the remainder of the paper we mostly do not distinguish between the two approaches, referring to the sequence as CS1, CS2 and FC1. When we need to distinguish between the two, we will refer to the specific courses, i.e., CS1A, CS2A and FC1A in the objects-first approach.

Goals. Regardless of their detailed structure, the introductory courses have the following goals:

- To introduce students to two distinct problem-solving paradigms: functional programming and object-oriented programming
- To introduce students to the mathematical tools and foundations of computer science
- To instill in students an appreciation for the interplay of theory and practice in computer science.

	Objects-First				Functional-First				
Topic	CS1A	CS2A	FC1A	Total	CS1B	CS2B	FC1B	Total	Diff.
History, social, ethical issues	5			5	3	2		5	0
Program execution	1			1	1			1	0
Control constructs	4			4	3	2		5	-1
Variables, types, expressions	4			4	3	2		5	-1
Methods/functions	3			3	2	1		3	0
Higher-order functions			3	3	3			3	0
Anatomy of a class	4			4		3		3	1
Interfaces	2			2		2		2	0
Inheritance	3			3		3		3	0
Alternate paradigm			4	4	1	3		4	0
Tracing, testing, debugging	2	1		3	2	1		3	0
Arrays, lists, strings	6	6		12	7	3		10	2
Informal algorithm analysis	1			1	1			1	0
Exceptions	2			2		2		2	0
Streams, files		2		2	2			2	0
Formal algorithm analysis		3		3			3	3	0
Queues, stacks		3		3		2		2	1
Recursion	2	3		5	4	1		5	0
Trees		6		6	4	2		6	0
Advanced structures		6		6		6		6	0
Graphs		5	4	9		2	7	9	0
Searching, sorting		3		3	3	2		5	-2
Logic and proofs			4	4			4	4	0
Boolean algebra			2	2			2	2	0
Functions			3	3			3	3	0
Sets, relations			4	4			4	4	0
Cardinality, basic probability			7	7			7	7	0
Solving recursive relations			4	4			4	4	0
Matrices			3	3			3	3	0
Total	39	38	38	115	39	39	37	115	0

Table 1: Topic Hours for the Introductory Courses

- To make students think about the ethical and social issues of the software systems they build.

Mathematics. Computer science relies on mathematical ideas in at least three ways:

- Mathematical objects (e.g., sets, relations, logic connectives, trees, graphs) provide the basis for various models in computer science;
- Mathematical reasoning (e.g., assertions, logic, proofs) provides a mechanism to prove the correctness and security of hardware circuits, algorithms, programs, and language constructs; and
- Mathematical tools (e.g., recurrence relations, finite probability, statistics) allow analysis of various algorithms and data structures.

In addition, mathematics is important in many of the subjects that computing supports, such as physics, engineering, economics, and biology.

A common theme in FC 1 is that mathematical and theoretical concepts, objects, and techniques are inherent in computer science. Thus the introduction of new ideas and theory is regularly tied to applications. Furthermore, since courses in discrete structures sometimes have the reputation of being a potpourri of disjoint and irrelevant details, considerable care should be taken to shape a coherent and theme-oriented course, where applications are shown for most topics. The addition of lab assignments should be used to show these connections.

FC 1 should minimally have the same mathematics prerequisites as the traditional first calculus course – that is, proficiency with basic algebra, geometry, trigonometry, logarithms, and exponential functions. Thus, FC 1 assumes that students will have completed a pre-calculus course either in high school or college.

It is important that coverage of induction extend well beyond induction over the integers. In computer science, many (perhaps most) applications of induction involve induction over structures and substructures. Of course, such items can be mapped to the integers (usually by taking size or depth into account), but explicit mention of such mappings typically adds unnecessary complexity to proofs. Students in FC 1 therefore should gain experience with a broad view of induction–with numerous examples involving induction over structures found elsewhere in computer science.

Labs. CS 1 and CS 2 have laboratory sessions to supplement the lectures/discussions, and FC 1 may have laboratories. Furthermore, projects may be scattered throughout these courses, with the nature of the projects highlighting instructor interests and local opportunities for research.

7 Core Courses

The 1986 and 1996 Model Curricula [4, 6] identified a core of four computer science courses for a liberal arts computer science curriculum, building upon the introductory sequence and covering principles of lasting significance. CC 2001 has a similar perspective that core courses of the curriculum implement the goals of *intermediate* courses [3]:

> The intermediate courses in the curriculum are designed to provide a solid foundation that serves as a base for more advanced study of particular topics.

Core courses take the wide ranging themes of CS 1 and CS 2 and FC 1 and begin to focus them on subareas of the discipline. They build upon the basic courses and provide the solid foundation for the more advanced studies that electives represent.

There are five courses in the core: *Principles of Computer Organization*, *Principles of Algorithm Analysis*, *Principles of Software Development*, *Principles of Programming Languages,* and *Theoretical Foundations of Computer Science* (FC 2). Descriptions for each of these courses appear in Appendix C.

As it has evolved as a discipline, an increasing part of the core of computer science has come to depend upon topics in discrete mathematics. Thus, while these curricular recommendations continue to specify three required mathematics-related courses for a liberal arts degree in computer science (just as in the 1996 Model Curriculum [6]), these recommendations show a shift in emphasis to include two semesters of courses that contain significant amounts of discrete mathematics. Specifically, the 2007 Model Curriculum recommends *Theoretical Foundations of Computer Science* (FC 2), building on FC1 and its discrete mathematics, to study areas of theoretical computer science and to present further topics of discrete mathematics as it applies to computer science.

This represents an expansion of the minimum recommendations expressed in CC2001 [3]. FC 1 covers much of knowledge units DS1 through DS6 from CC2001 [3], while highlighting applications that require proof, logic, and counting. FC 2 completes coverage of these topics and provides solid coverage of other foundations for theoretical computer science. While the courses described here represent a revision of the 2-semester sequence described by Pedagogy Focus Group 2 [5], they should not be considered as presenting material at a slower pace than the one-semester course described in CC2001 [3]. Rather, these courses cover substantially more material and that material is covered in greater depth than the one-semester course of CC2001 [3].

FC 2 pays attention to proof techniques throughout, using mathematical theory and rigor to obtain interesting results in computer science. FC 2 covers important foundational material in computer science as well as material that is relevant for advanced courses, such as compiler construction. Thus, FC 2 should be taken along with the computer science core courses.

8 Electives and Research

Electives. While the basic and core courses emphasize the central and enduring themes of the discipline, electives provide the opportunity for the student to tailor the educational program both to prepare for the capstone experience and to address goals for after the baccalaureate degree. To this end, each program will need to offer a sufficient variety of courses to meet students' goals and needs. Some courses may be offered every other year. Selection of elective courses to offer reflects, at least in part, the experience and interests of the faculty. A partial list of possibilities follows:

Artificial Intelligence	Database Systems	Security
Bioinformatics	Distributed Systems	Simulation and Modeling
Compiler Construction	Graphics	Software Engineering
Computational Complexity	Natural Language Processing	Vision
Computational Science	Network Systems	
Computer Architecture	Operating Systems	

The 2007 Model Curriculum recommends a third mathematics-related courses to support a computer science degree. This additional course should be selected by the student in consultation with a faculty advisor to provide additional mathematical background for later work in computer science.

The Capstone (Research) Experience. The capstone experience can take several forms. At one end is the required senior thesis or senior project structured much like a master's thesis or project and given credit equivalent to one or more courses. At the other end is a requirement that the student complete a

project-based course or one of a set of courses designated as having a significant project component. In either case, the goal is to require that the student participate in the identification of a problem, develop a project proposal outlining an approach to the problem's solution, implement the proposed solution, and test or evaluate the result. Since any substantive project should seek to address a real problem, students should devote at least five hours within their development of the project to the potential social impact and ethical implications of this work. The student should document the project in a thesis or report style, as appropriate. The project should represent the culmination of the undergraduate experience in computer science and, as such, should draw upon and extend the basic principles mastered in the student's course work.

9 Comparison of 2007 Model Curriculum with CC2001

Computing Curricula 2001 reflects a change in perspective from Computing Curricula 1991, including a significant decrease in coverage of some subject areas, an expansion of other areas, and the addition of new areas. Although such adjustments may fit some types of schools, the reduction in hours from CC 1991 to CC 2001 in the areas of algorithms and complexity, theory of computation, and programming languages is inconsistent with the liberal arts perspective. Also, while CC 2001 recommends only one course in discrete structures and theory, many liberal arts CS faculty now believe that students require at least two semesters to appropriately master this material. Altogether, the change in emphasis in CC 2001 raises important questions regarding the content covered in a liberal arts curriculum.

Important issues regarding problem-solving paradigms were raised regularly in discussions with members of the Pedagogy Focus Groups and the Task Force for CC 2001. Many group members and Task Force members agreed that the introduction of a new paradigm is valuable, but requires a reasonable time commitment. Multiple paradigms expand students' thinking and perspectives, but students cannot become comfortable with new views of problem solving with the allocation of only 4 or 8 hours in a course. For some on the CC 2001 Task Force, this led to the conclusion that multiple paradigms could not be covered within acceptable time constraints, and multiple paradigms are not part of the CC 2001 core.

Many faculty at liberal arts colleges, however, reach a different conclusion. A liberal arts perspective embraces different views of problem solving, so inclusion of this material at an early stage is an important part of the 2007 Model Curriculum. This means that students should become comfortable with multiple problem-solving paradigms within the first year or two. The next section identifies ways to achieve this objective.

Of course, when a new curriculum includes multiple views of problem solving and also acknowledges modern topics such as are reflected in CC 2001, old packaging of topics cannot work. With liberal arts programs restricted to about 9 computer science courses and 3 mathematics courses, new ideas must replace some previous content. This 2007 Model Curriculum acknowledges these constraints by making appropriate choices for a liberal arts setting.

Table 2 compares the overall 2007 Model Curriculum with CC 2001. In the case of the former, this table shows the knowledge units for the introductory sequence and core courses. The starred (**) knowledge units indicate that the developers of this 2007 Model Curriculum have a different perspective from the CC 2001 Task Force regarding what material is core for a liberal arts computer science program. In particular, this 2007 Curriculum includes significantly more coverage of algorithms and programming languages than CC 2001, both in CC 2001's core and non-core areas.

Beyond the core, the 2007 Model Curriculum expects that at least one elective or a senior project/thesis course will include a research or development project. Topics covered in the electives and project/research activities supplement these totals. In particular, this 2007 Model Curriculum recommends that the project devote 5 hours to the consideration of the social and ethical implications of the project. With this addition (at the star (*) in Table 2), these recommendations provide comparable emphasis in this important area.

	Core Courses				Theor.	Grand	
	Comp. Org.	Algo-rithms	Soft. Dev.	Prog. Lang.	Found. (FC2)	Total w/Intro.	CC2001
DS: Discrete Structures		3			12	49	43
PF: Programming Foundations		3	2	3		39	38
AL: Algs/Complexity-Core **		18	3		6	43	31
AL: Algs/Complexity-Other **		15			11	26	0
AR: Architecture & Org.	39					40	36
HC: Human-Comp. Interaction			4			5	8
GV: Graphics & Visual Comp.						0	3
IS: Intelligent Systems						4	10
IM: Information Management						0	10
OS: Operating Systems			3	6		9	18
NC: Net-Centric Computing			7		3	10	15
PL: Programming Lang.-Core **				23		36	21
PL: Programming Lang.-Other **				7		11	0
SP: Social, Professional Issues			6			11*	16
SE: Software Engineering			14			20	31
CN: Comp. Sci., Num. Methods						0	0
Totals	39	39	39	39	32	303	280

Table 2: 2007 Model Curriculum Comparison with CC 2001

Comparing various subject areas, the liberal arts perspective shows clearly in the 2007 Model Curriculum. The 2007 Model Curriculum has significantly more emphasis than CC 2001 in the areas of algorithms and complexity (with classes P and NP), theory of computation (including automata and computability), and programming languages (including grammars). The 2007 Model Curriculum correspondingly has significantly less emphasis in the areas of intelligent systems, information management, operating systems, and software engineering. Including the project, the 2007 Model Curriculum has about the same level of coverage as CC 2001 in the areas of programming foundations, and social and ethical issues. Both the 2007 Model Curriculum and CC2001 have electives that provide additional hours in areas of particular interest to individual faculty and students.

10 Implementation Details

There are 12 required courses for a computer science major following this model curriculum: 3 introductory courses, 5 core courses, 3 elective courses, and a supporting math course. The following discussion shows that the model curriculum can be completed by students within a normal four-year program and can be implemented with the limited number of faculty members in departments at many liberal arts colleges.

The sample program shown in Table 3 demonstrates how students can complete the required courses within the normal four-year college period. The project is listed in parentheses in each semester of the senior year, but may be taken with any elective.

At liberal arts colleges, however, many students may not begin as computer science majors but switch into the field from other disciplines during their first or second year. Table 4 indicates that it is possible for students to start the program in their second year and still finish in a normal four-year time span.

	Fall	**Spring**
First Year	CS 1	CS 2 FC 1
Second Year	Core 1 FC 2	Core 2 Coordinated Math Course
Third Year	Core 3	Elective 1 Core 4
Fourth Year	Elective 2 (project)	Elective 3 (project)

Table 3: Typical Four-Year Student Schedule

	Fall	**Spring**
First Year		
Second Year	CS 1 FC 1	CS 2 FC 2
Third Year	Core 1 Coordinated Math Course	Core 2 Core 3
Fourth Year	Elective 1 Core 4	Elective 2 Elective 3 (project)

Table 4: Typical Three-Year Student Schedule

Staffing of the proposed model curriculum is possible under modest assumptions. CS 1 and CS 2 are offered every semester. All four core courses are taught once each year. FC 1 may be taught by the mathematics department, but FC 2 is offered by the computer science department once each year. FC 1 and 2 can be taught by computer science faculty or co-taught by mathematics and computer science faculty, depending on local institutional preferences and constraints (hence we name them FC [*F*oundations of Computing], rather than CS or MA). The traditional discrete mathematics course that FC 1 is designed to replace is now taught by mathematics faculty in about half of all colleges and universities and by computer science faculty in the other half. The traditional theory of computation course, which FC 2 is designed to replace, is now taught mainly by computer science faculty, although it can be taken for mathematics major credit at many institutions. Finally, four electives are offered each year, two per semester.

Overall, this minimal, but reasonable, schedule requires the staffing of 13-14 computer science sections each year. Table 5 shows that, teaching 4 courses per year per faculty member, a department of 4 faculty members in a small liberal arts school can offer a rigorous and high-quality undergraduate program in computer science.

11 Conclusions

This 2007 Model Curriculum updates the earlier model curricula [4, 6] of the Liberal Arts Computer Science Consortium. Important characteristics of this 2007 curriculum include:

- 3 introductory, 5 core, and 3 elective computer science courses;
- a new series of foundations of computing courses, including a 2-semester sequence combining about

Annual Sections	Course	Comments
2	CS 1	1 section per semester
2	CS 2	1 section per semester
1	FC 2	1 section per year
4	Core Courses	2 per semester
4	Elective Courses	2 per semester
0-1	Project	Some faculty may receive teaching credit
Total 13-14		

Table 5: A Possible Staffing Schedule

1 1/3 semesters of discrete mathematics with with about 2/3 semesters of other foundations of computer science;

- a solid introduction to problem solving, through the inclusion of multiple paradigms within the introductory sequence (inclusive of the first discrete mathematics course);
- study of core topics in computer organization, algorithm analysis, software development, and programming languages;
- the inclusion of a significant capstone or research experience, possibly as part of an upper-level elective; and
- study of the social and ethical issues surrounding computing as an on-going theme.

In addition to these intellectual strengths, the curriculum fits within the typical constraints of a liberal arts college. The model curriculum has an appropriate size and shape for a liberal arts setting, students can begin a credible major during their second year, and all courses can be taught by approximately four computer science faculty. Altogether, these recommendations build upon the opportunities and strengths available within liberal arts environment to yield an intellectually sound, high quality, rigorous, and modern program that also is practical.

12 References

1. ACM Curriculum Committee on Computer Science. Curriculum 78: Recommendations for the undergraduate program in computer science. *Communications of the ACM* 22, 3 (Mar. 1976), 151-197.
2. ACM/IEEE-CS Joint Curriculum Task Force. Computing Curricula 1991. *IEEE Computer Society Press*, December 17, 1990.
3. The Joint Task Force on Computing Curricula of the IEEE Computer Society and the Association for Computing Machinery, Computing Curricula 2001: Computer Science, December 15, 2001. Available on-line through `http://www.sigcse.org/` .
4. Gibbs, N. and Tucker, A. A model curriculum for a liberal arts degree in computer science. *Communications of the ACM* 29, 3 (Mar. 1986), 202-210.
5. Pedagogy Focus Group 2 on Supporting Courses of the Joint Task Force on Computing Curricular 2001, Draft Report, Version 5.2.

6. Walker, H., and Schneider, G. A revised model curriculum for a liberal arts degree in computer science. *Communications of the ACM* 39, 12 (Dec. 1996), 85-95.

7. Wulf, W. The Urgency of Engineering Education Reform, Transcription of the Keynote Presentation in Proceedings of the West Point Interdisciplinary Math Workshop, October 2000.

Sidebar: Process and Participants

This model curriculum reflects the ongoing discussions of the Liberal Arts Computer Science Consortium (LACS). Initially funded by a grant from the Sloan Foundation, the group produced a model curriculum in 1986 [4] and a revised model curriculum in 1996 [6]. Annual summer meetings have resulted in a range of papers and presentations covering such topics as service courses, formal laboratories, breadth-first experiments in the first two courses, and goals for the first two years of undergraduate computer science.

The following members of LACS have actively contributed to the 2007 Model Curriculum: Joel Adams, Calvin College; Doug Baldwin, State University of New York at Geneseo; Alyce Brady, Kalamazoo College; Amy Briggs, Middlebury College; Kim Bruce, Williams College; Robert Cupper, Allegheny College; Scot Drysdale, Dartmouth College; Max Hailperin, Gustavus Adolphus College; Michael Jipping, Hope College; Charles Kelemen, Swarthmore College; Andrea Lawrence, Spelman College; Takis Metaxas, Wellesley College; Robert Noonan, College of William and Mary; Rhys Price-Jones, Rochester Institute of Technology; David Reed, Creighton University; G. Michael Schneider, Macalaster College; Allen Tucker, Bowdoin College and Henry Walker, Grinnell College

The Consortium met at Grinnell College in August 2002 for an intensive discussion that resulted in an initial version of this Model Curriculum. Another result of the meeting was the creation of small working groups that continued working on the draft curriculum during the 2002/03 academic year. The final outline of the Model Curriculum was developed at the meeting held at Calvin College in August 2003. An additional working group was created to write this document. Again, the groups continued their work during the 2003/04 academic year. Detailed wording was discussed at length during a meeting held at Wellesley College in August 2004, and groups continued the refinement of wording through the 2004/05 academic year. The group agreed on a completed, final document at its meeting at the Rochester Institute of Technology in August 2005. Editing refinements have occupied much of 2006.

The final document also benefited from the presentation of the Model Curriculum at several meetings. We want to thank all those who contributed.

A Objects-First Introduction to Computer Science

In this appendix the ordering of topics within each course provides one reasonable ordering for that course, but there are, of course, other pedagogically sound ways to organize the course.

A.1 CS 1A: Introduction to Algorithmic Problem Solving Using an Object-Oriented Language

This is the relatively traditional introduction to programming. We do believe, however, that since the integration of data and operations on data is central to the object-oriented approach, objects should be introduced early in the course. Furthermore, if the language of instruction is Java, then the use of interfaces as pure types is another key concept that should not be too long delayed.

A.1.1 Course Outline

- **History, ethical and social context of computing** (5 hours)
- **How programs execute** (1 hour)
  - Basic von Neumann architecture
  - Instructions
  - Fetch-decode-execute cycle
  - Basics of compiling, loading, and running programs
- **Basic programming constructs** (8 hours)
  - Variables
  - Types and classes as sets of operations: basic operations on primitives, reading class documentation to discover the operations on classes, using classes from a library
  - Objects: constructing objects, methods as definitions of objects' behaviors, invoking methods, capturing return values in variables
  - Sequential control flow
  - Conditionals
  - Iteration
- **Anatomy of a class** (6 hours)
  - Formal and actual parameters
  - Data available to a method
  - Local variables, parameters, and instance variables
  - Return values; embedded method calls in complex expressions
  - Design and implementation of simple classes
- **Interfaces as type abstractions** (2 hours)
  - Using interfaces as types
  - Dynamic binding of messages to methods (i.e., of method calls to method implementations)

  - Creating interfaces and implementing classes

- **Inheritance: extending classes to provide additional or modified behavior** (3 hours)
- **Software development** (5 hours)
  - Analyzing problem statements, designing for users, software design, testing
  - Basic constructs revisited: a more detailed look at control structures, exceptions, and other constructs
  - Complex Boolean expressions and propositional logic
  - Expressions vs. statements; functions vs. procedures (or accessors vs. modifiers)
  - Variations on conditional and iterative constructs
  - Shared data: class variables
  - Scope
  - Concepts of procedural abstraction
  - Strings
- **Recursion** (2 hours) (alternatively, may be introduced with recursive data structures, either early in CS 1 or in CS 2)
- **Linear and multi-dimensional indexed data structures** (6 hours)
  - Constructing arrays, accessing elements
  - Variations on linear traversals (basic traversal, accumulating results, finding min/max values, linear search)
  - Alternatively, could focus on collections and iterators, or on recursive data structures such as linked lists as the first linear data structure.
- **Informal Algorithm Analysis** (1 hour)
  - Multiple algorithms to solve a problem (e.g., linear and binary search, sorting algorithms)
  - Comparison of appropriateness of algorithms based on
    * Performance
    * Constraints (e.g., binary search requires sorted data in a random-access data structure)
    * Readability and maintainability
- *Total:* 39 hours.

A.2 CS 2A: Data Structures

Programming at the level of CS 1A is a prerequisite for this course.

This course focuses on classic data structures from several perspectives, including implementing basic data structures, utilizing standard library classes as building blocks, and analyzing the appropriateness and performance of various data structures in different contexts. The ordering of topics in the table below assumes that students have been introduced to arrays or vectors before linked lists, and to iteration before recursion, but, the sequence and timing of various topics will vary with the instructor. For example, it is quite possible to cover sets, maps, and hash tables immediately after covering or reviewing linear indexed data structures.

Other topics, such as data structure and algorithm analysis and common sorting and searching algorithms, may be spread out over the course of a semester.

A number of other topics, such as design patterns, graphical user interface development, and such, may appear in a CS 2 course. They are not listed here because the specific set of topics instructors choose to cover will depend on the strengths and interests of the faculty, their choice of motivating examples, and the kinds of projects they wish to assign. Students should, however, have ample opportunity to continue to expand their software development skills in CS 2.

A.2.1 Course Outline

- **Abstraction, e.g., Java interfaces, collection classes, and iterators** (3 hours)
- **Data structures and algorithm analysis** (2 hours)
  - Formal and informal time and space analysis
  - Big-Oh notation
  - Choosing the appropriate data structure or algorithm for a given context
- **Linear and 2-Dimensional Data Structures (much of this may be review, depending on what is covered in CS 1** (6 hours)
  - Linear indexed data structures (arrays, standard library/API classes), variations on linear traversals
  - Linked lists: implementing a linked list and using standard library/API classes
  - Multi-dimensional arrays
  - Insertion, deletion, and traversal on indexed and linked linear data structures
  - Iterator and Visitor design patterns
- **Queues and Stacks** (3 hours)
  - Using queues and stacks appropriately in algorithms.
  - Implementing queues and stacks; the issues behind linked and non-linked implementations.
- **Recursion** (3 hours)
  - Recursive algorithms.
  - Introduction to, or review of, recursive data structures.
- **Trees** (6 hours)
  - Terminology and algorithms, such as traversals and counting the number of nodes or leaves.
  - Binary search trees
  - Other tree-related data structures, such as heaps and priority queues.
- **More advanced linear structures** (6 hours)
  - Dictionaries, Maps
  - Sets

  - Hash Tables

- **Graphs** (5 hours)
- **Common algorithms** (3 hours)
  - Sequential and binary search
  - Sorting algorithms, including n*log (n) and nˆ2 algorithms
- **Unit Testing** (1 hour)
- *Total:* 39 hours.

A.3 FC 1A: Discrete Structures and Functional Programming

Programming (CS 1A or other experience) is a co-requisite for this course. While there are advantages to teaching a functional language to students who are not already thinking imperatively, there may be disadvantages to teaching a functional language to a group of students with disparate backgrounds, some of whom are already comfortable with programming, while others are not.

A.3.1 Themes

- Functional programming
- Functions and their properties
- Proof techniques and their applications
- Counting techniques and finite probability
- Introduction to graphs
- Matrices

A.3.2 Applications

- Tree properties
- Resource allocation graphs and deadlock detection
- Traveling Salesperson Problem
- Properties of powers of adjacency matrices

A.3.3 Course Outline

- **Functional programming** (7 hours)
  - Functional paradigm
  - Higher-order functions
  - Inductively defined data structures

- **Introduction to Logic and Proofs** (these techniques should be used regularly in what follows) (4 hours)
  - Direct proofs
  - Proof by contradiction
  - Mathematical induction, with explicit coverage of structural induction (e.g., on trees and subtrees) for both recursive definitions and proofs
- **Logical operations and DeMorgan's laws** (2 hours)
  - Logical operations
  - DeMorgan's laws for the AND and OR connectives and for universal and existential quantifiers
  - Possible application:
    - Digital circuit design
- **Functions** (3 hours)
  - Properties, composition
  - Possible applications:
    - Symbolic manipulation in a symbolic algebra package
    - Reduction, associativity of composition
- **Sets and Relations** (4 hours)
  - Venn diagrams, complements, Cartesian products, power sets
  - Reflexivity, symmetry, transitivity, equivalence relations
  - Possible applications:
    - Connection with relational databases
    - Common operations (joins, selection)
    - Basics of types and type inference
- **Cardinality, counting, and finite probability** (7 hours)
  - Pigeonhole principle
  - Combinations and permutations
  - Finite probability
  - Expected value
  - Binomial coefficients
  - Possible applications – often using induction proofs
    - Size of power sets
    - Calculating odds of getting certain card hands and of winning lotteries and dice games
- **Recurrence relations** (4 hours)
  - Big "O" notation and "Theta"

  - Basic formulas and solutions
  - The Master Theorem for divide-and-conquer algorithms
  - Possible applications
    * Tree properties (e.g., height, number of leaves)
    * Analysis of divide-and-conquer algorithms

- **Graphs** (4 hours)
  - Directed and undirected, weighted and unweighted
  - Cycles and connectivity, with algorithms to determine these properties
  - Basic algorithms and proof techniques
  - Traversals (depth-first and breadth-first)
  - Possible applications
    * Resource allocation graphs and deadlock detection
    * Internet routing

- **Matrices** (3 hours)
  - Basic properties
  - Possible applications
    * Adjacency matrices and their powers (include careful proofs)

- *Total:* 38 hours

B Functional-First Introduction to Computer Science

In this appendix the ordering of topics within each course provides one reasonable ordering for that course, but there are, of course, other pedagogically sound ways to organize the course.

B.1 CS 1B: Introduction to Computing with a Functional Language

This course introduces functional problem solving together with some imperative problem solving. The course has at least four major goals:

- To provide a general introduction to the fundamental ideas of computer science: algorithms, data structures, and abstraction,
- To present problem-solving from a functional programming perspective,
- To introduce computer programming (algorithm design, documentation, coding, testing, and debugging) in a high-level [functional] programming language, and
- To illustrate important application areas of computing, such as computer simulation, structure of expert systems, and the use of the World Wide Web.

The last section of the course provides a transition to object-oriented problem solving and CS 2 by considering objects as higher-order functions.

While the material on sorting and streams could be omitted to give more time to other areas or to explore other topics, it is helpful to combine the earlier ideas somehow in an interesting application before going on. As a variation, the material in these sections can be done much earlier, with some of the Web applications being done as early as the second week. In this variant, applications related to the World Wide Web serve as a motivation and running example for much of what follows.

B.1.1 Course Outline

- **Getting started** (3 hours)
  - Views of problem solving
  - History and social context of computing
  - Symbols, lists, functions, function definitions
  - Read-execute-print cycle
- **Function composition and the basics of recursion** (4 hours)
  - Conditional evaluation
  - Parameters
  - Recursion with lists
- **Numbers and recursion with numbers** (3 hours)
  - Recursive functions on and returning numbers
  - Reasoning about procedures with pre-conditions and post-conditions
- **Data storage with lists** (4 hours)

  - Lists of lists
  - Pairs
  - Association lists
  - Deep recursion

- **Procedures, evaluation efficiency, and encapsulation** (3 hours)
  - Local binding
  - Tail recursion
  - Procedural abstraction
  - Tracing code
  - Testing
  - Debugging

- **Additional data types and operations** (3 hours)
  - Numbers
  - Characters
  - Strings
  - Functions related to the above

- **Higher-order procedures** (3 hours)
  - Procedures as values
  - Patterns involving procedures

- **Data structures and side effects** (3 hours)
  - Vectors (i.e., the idea of an aggregate data structure with O (1) access, basic accessor operations)
  - Iteration (i.e., the idea of repeating actions in this case, over a data structure)
  - Binary search (linear search is already familiar from the discussion of lists)

- **Basic order analysis** (3 hours)
  - Big-O notation
  - Standard complexity classes
  - Straightforward analysis of familiar algorithms (list algorithms, linear search, binary search)

- **Sorting and algorithmic analysis** (3 hours)
  - At least one quadratic-time sort (e.g., insertion sort) and one $n \log n$ sort (e.g., merge sort)
  - Asymptotic analysis of upper and average complexity bounds
  - Empirical measurements of performance

- **Streams** (3 hours)

  - Input, output, files: interfaces among user-based data, long-term storage, and programs. Application: Web pages, and CGI scripts

- **Objects as higher-order functions** (4 hours)

  - Assignment
  - Encapsulation
  - Method naming
  - Stacks
  - Queues
  - Standard stack/queue operations

- *Total:* 39 hours.

B.2 CS 2B: Object-Oriented Data Structures

Programming at the level of CS 1B is a prerequisite for this course.

This second course in the introductory sequence has four goals:

- To build on previous knowledge to study the design, analysis, and verification of algorithms
- To present problem-solving from an object-oriented programming perspective
- To study standard ADTs, such as stacks, queues, lists, trees, priority queues, heaps, and hash tables, and
- To apply object-oriented principles to solve some interesting application.

In the functional-first/multi-language model, the first part of CS 2B marks a significant transition in ways of approaching problem solving. While students typically find this transition takes effort, the long-term effect is to expand their perspectives. Time introducing a second language here in CS 2B is balanced by the ability to move more quickly later (e.g., in a programming languages course if that would otherwise be where students encounter their second language).

Perspectives and approaches regarding object-oriented design form a unifying theme throughout much of the course.

The ADTs in the latter part of this course should be treated as opportunities to apply and practice the elements of object-oriented programming and algorithm analysis presented in the first part. This will unify the material, add coherence, and reinforce the fundamental concepts of design and analysis.

Variations of this course could highlight such areas as Web-based applications, GUIs, threads, sockets, or file handling.

B.2.1 Course Outline

- **Transition of languages and syntax. Change of perspective from classes and objects as higher-level procedures to classes and objects in a standard object-oriented language** (6 hours)

  - Classes
  - Objects

  - Methods
  - Comparisons of identical programs in both functional and object-oriented languages.

- **Language basics** (3 hours)
  - Conditionals
  - Loops
  - Testing
  - Debugging

- **Additional elements of programming in an object-oriented language** (6 hours)
  - Object-oriented design
  - Libraries
  - Inheritance
  - Polymorphism

- **Examples with arrays and more algorithms** (3 hours)
  - Searching
  - Sorting

- **Analysis of standard recursive algorithms** (3 hours)
  - Recurrence relations (basic formulas and solutions)
  - The Master Theorem for divide-and-conquer algorithms
  - Analysis of divide and conquer algorithms (this analysis is to be used with the ADTs that follow)

- **Exception Handling** (3 hours)
  - Throwing exceptions
  - Catching exceptions
  - Implications for algorithmic development and efficiency

- **Lists in an Object-Oriented Language** (6 hours)
  - Unordered and ordered lists using array and pointer implementations

- **Some Standard ADTs** (6 hours)
  - Priority queues
  - Heaps
  - Heap sort
  - Dictionaries
  - Hash tables

- **Tree and Graph ADTs** (3 hours)

  - Recursive definition
  - Insertion
  - Deletion
  - Search
  - Traversal
  - Efficiencies of the above algorithms

- *Total:* 39 hours.

B.3 FC 1B: Discrete Structures for Computer Science

Programming at the level of CS 1B is a co-requisite for this course.

With several topics from the function-based FC 1A now covered in CS 1B and 2B, FC 1B can focus on mathematical reasoning, objects, counting, probability, and statistics. While this course has been informed by Draft 5.2 of the CC2001 Pedagogy Focus Group 2 on Supporting Courses [5], the resulting course is significantly different.

B.3.1 Themes

- Proof techniques and their applications
- Counting techniques
- Introduction to graphs
- Probability and statistics

B.3.2 Applications

- RSA encryption
- Combinations and permutations
- Elementary probability
- Tree properties
- Resource allocation graphs and deadlock detection

B.3.3 Course outline

- **Introduction to logic and proofs** (these techniques should be used regularly in what follows) (4 hours)
  - Direct proofs
  - Proof by contradiction
  - Mathematical induction, with explicit coverage of structural induction (e.g., on trees and subtrees) for both recursive definitions and proofs
- **Boolean Algebra** (2 hours)

  - The algebra of Boolean values and operations
  - Possible application:
    * Digital circuit design

- **Relations** (2 hours)
  - Reflexivity
  - Symmetry
  - Transitivity
  - Equivalence relations
  - Possible applications:
    * Connection with relational databases
    * Common operations (joins, selection)

- **Sets** (3 hours)
  - Venn diagrams
  - Complements
  - Cartesian products
  - Power sets
  - Possible application:
    * Basics of types and type inference

- **Number theory** (6 hours)
  - Factorability
  - Properties of primes, including Fermat's Little Theorem
  - Modular arithmetic and different bases
  - Possible applications:
    * RSA encryption
    * Hash functions
    * Algorithms for arithmetic operations within a computer (e.g., multiplication)

- **Cardinality and counting** (5 hours)
  - Pigeonhole principle
  - Binomial coefficients
  - Possible applications – often using induction proofs:
    * Size of power sets
    * Combinations
    * Permutations

- **Elementary probability and statistics** (8 hours)

  - Basic finite probability
  - Independence
  - Conditional probability
  - Expectation
  - Descriptive statistics
  - Possible applicators:
    * Average and worst case analysis
    * Calculating odds of getting certain card hands, of winning lotteries, of rolling dice
    * Shape of search trees (without explicit balancing)
    * Analysis of hashing algorithms

- **Introduction to graphs** (7 hours)
  - Graph definitions
  - Directed and undirected, weighted and unweighted
  - Basic algorithms and proof techniques
  - Searching, traversals (depth-first and breadth-first)
  - Possible application:
    * Resource allocation graphs and deadlock detection

- *Total:* 37 hours

C Core Course Descriptions

In this appendix the ordering of topics within each course provides one reasonable ordering for that course, but there are, of course, other pedagogically sound ways to organize the course.

C.1 Principles of Algorithm Analysis

The prerequisites for this course are CS 2 and FC 1.

The course on algorithms is a continuation of the study (begun in CS1 and CS2) of the basic data structures and algorithms. The approach here is more formal, both with respect to correctness and with respect to the time and space resources required for various algorithms and their associated data structures. The main thrust of the course is algorithm design techniques and the interplay between design, analysis, data representation, correctness, and empirical analysis.

The first part of the course outline includes topics all courses should cover. This is followed by a selection of topics that could be used to round out a coherent and interesting course. More topics are listed than can be used in a one semester undergraduate course.

C.1.1 Course outline:

- **Algorithm design techniques** (8 hours)
  - divide and conquer
  - greedy
  - dynamic programming
  - time/space tradeoffs
  - sample algorithms using above techniques: e.g., graph algorithms, pattern matching, and string/text algorithms
- **Proofs of algorithm correctness** (6 hours)
- **Algorithmic analysis** (6 hours)
  - Basic ideas: clear problem statements and recursive algorithms mean inductive proofs of correctness
  - Review big-Oh, big-Theta, introduce big-Omega, little-oh
  - Introduce concepts of worst case, expected case, the relationships of recursive algorithms to recurrence relation to relation solutions, amortized analysis, empirical methods of analysis
- **Algorithmic complexity** (6 hours)
  - P
  - NP
  - NP-complete
  - NP-hard
- **Simple lower bounds** (2 hours)
- **Reduction proofs** (3 hours)

  - lower bound
  - NP-completeness

- Additional Topics that may be covered could include some subset of the following: (8 hours)
  - Graph Algorithms
  - Network Flow and Matching
  - Matrix Operations-Sequential and Parallel
  - Linear Programming
  - Number Theory and Cryptography
  - Data Compression
  - Probabilistic Algorithms
  - Advanced Techniques for String Matching
  - Fast Fourier Transform
  - Geometric Algorithms
  - Algorithms in Computational Biology
  - Indexing and Search Engines
  - More on Parallel Architecture and Algorithms
  - Branch and Bound Algorithms
  - Genetic algorithms
  - Network Learning Algorithms

- *Total:* 39 hours.

There are a number of ways a syllabus for Algorithms could be organized. A bottom-up organization might have students discovering and learning algorithms for various applications and then abstracting from these algorithms to important algorithm design and analysis techniques. A top-down approach might present the design techniques first and then illustrate them with applications to various problems.

C.2 Principles of Computer Organization

The prerequisite for this course is CS 2.

Computer organization presents an overview of computer architecture components and how those components work together. It provides the student with exposure to lower level abstractions such as digital logic, machine language, computer architecture, and data representation. This course focuses the higher level concepts taught in CS1 and FC 1 on a lower level example of computer hardware.

A list of topics is provided below. Based on the length of the semester and the desire to include other topics, these numbers may be modified accordingly.

C.2.1 Course outline:

- **Introductory Material** (7 hours)
  - Overview of the von Neumann architecture: arithmetic-logic unit, control unit, memory, buses, and I/O
  - The internal representation of information: signed and unsigned integers, floating point, and non-numeric data
  - Performance analysis: throughput vs. response time, performance factors (clock rate, instructions per cycle, and instruction count), and benchmarking
- **The Instruction Set Architecture (6 hours)**
  - Instruction set architectures: fetch/decode/execute model of computation, instruction formats and addressing techniques, instruction sets, machine and assembly language
  - Simple assembly language programming or review of compiled high level language code
- **Digital Logic and Microarchitecture Design (13 hours)**
  - Introduction to digital logic: combinatorial and sequential building blocks
  - A simple, non-pipelined processor organization
  - Pipelining and instruction level parallelism
- **Memory Hierarchies (5 hours)**
  - Caching, virtual memory, and paging
- **Input/Output (3 hours)**
  - I/O architectures: interrupts, DMA, bus architectures, secondary storage
- **Multiprocessors, networking, and clusters (5 hours)**
  - Interconnection technologies, cache coherence, and protocols
- *Total:* 39 hours

C.3 Principles of Programming Languages

The prerequisites for this course are CS 2 and FC 1.

The programming languages course has evolved greatly in the last several years. Begun in the 1970s as a survey of different languages, this course now emphasizes principles of language design, programming paradigms, and their underlying theories. It aims to provide students with the tools to understand not only how today's languages are designed, implemented, and used, but also how tomorrow's languages might be designed to meet the new challenges of emerging computer architectures and application domains. The integration of mathematical principles and ideas throughout this course is important for its effective implementation.

C.3.1 Course Outline

- **Overview** (3 hours)
  - language families
  - paradigms
  - virtual machines
  - design principles
- **Formal grammars and language analysis** (6 hours)
  - regular expressions and context free grammars
  - recursive descent parsing
  - table-driven parsing
- **Types** (4 hours)
  - simple and compound types
  - static and dynamic types
  - type safety
  - inheritance
  - subtypes
  - polymorphism
- **Semantics**(2 hours)
  - formal semantics (operational or denotational)
  - relationship with language design and implementation on abstract machines
- **Language translation** (3 hours)
  - simple interpreter of a parse tree
  - structure of a compiler and its relation to language design
- **Control of execution** (3 hours)
  - control structures (including iterators)
  - functions, exception handling, recursion
  - binding and binding time, scope, lifetime
- **Declarations and modules** (3 hours)
  - language support for modularity, information hiding and encapsulation
  - static overloading versus overriding
  - parameter passing
- **Object-oriented languages** (3 hours)

  - classes and methods
  - inheritance
  - design and implementation issues

- **Run-time storage management** (3 hours)
  - memory management
  - garbage collection
  - stack
  - heap

- **Declarative programming paradigms**(3 hours)
  - the functional and/or logic programming paradigms
  - applications in program specification, theorem proving, and other areas of artificial intelligence programming.

- **Distributed and parallel programming paradigms** (6 hours)
  - Threads
  - Remote procedure calls
  - Monitors
  - Locks
  - Uses in client-server applications and other key domains

- *Total: 39 hours*

C.4 Principles of Software Development

The prerequisite for this course is CS 2.

Regardless of which curriculum model is used in CS 1/CS 2 – objects-first or functional-first – there is often too much material to complete in the allotted time frame. The *Principles of Software Development* course builds on the introductory sequence, providing additional time to cover programming topics that should be introduced as early as possible in the computer science curriculum. Without this course, these concepts may be delayed until junior/senior electives or omitted entirely.

While the exact list of topics depends on the topics covered in CS1/CS2, the course could be used to address the following important issues:

C.4.1 Course Outline

- **Advanced concepts in software design and development** (14 hours)
  - object-oriented design issues–e.g., determining classes, state, and behaviors and models of inheritance
  - design patterns
  - design languages, such as UML

  - using APIs

- **Verification** (8 hours)
  - unit testing, systems test
  - test coverage
  - white-box vs. black-box testing
  - design by contract

- **Modern software development techniques** (17 hours)
  - GUIs, event-driven programming, and human-computer interactions (HCI)
  - Threading and multi-threaded code (at the level of an API such as Java threads)
  - The client-server model of networking (at the level of an API such as java.net)
  - Exception handling and fault tolerant computing
  - Streams and stream programming
  - Elementary concepts in security, authentication, and encryption

Many other topics could be included, depending on the interests of the instructor and the characteristics of the computer science program. These additional topics might include material on recursion, graphics, professional and ethical issues, and an introduction to elementary topics in relational databases.

This course should not be viewed as software engineering, usually offered as an advanced elective at the junior/senior level. Principles of Software Development is a required core sophomore/junior level course that treats fundamental concepts in modern software development. Software engineering builds on these concepts to present more advanced material and work on significantly larger problems. A software engineering elective might include such topics as: formal requirements and specifications, rapid prototyping, design methodologies, risk and liability issues, validation and testing strategies, software planning and estimation, project management including budgeting and staffing, software evolution, source code control, and professional practices.

Not only does Principles of Software Development lay the groundwork for a future course in software engineering, it also contains material that prepares students for other electives, for example in operating systems, networking, and cryptography.

C.5 FC 2: Theoretical Foundations of Computer Science:

The course FC 1 is a prerequisite for this course.

This course focuses on finite state machines and computability, and covers a modest amount of number theory and probability and statistics. There is room to add more material from number theory, finite state machines, statistics, or computability.

C.5.1 Themes

- Finite state machines
- Use of careful proof techniques in applications
- Computability
- Basic probability & statistics
- Number theory

C.5.2 Application

- RSA encryption
- Algorithmic efficiency
- Limits of computing – Halting Problem, universal computer/language

C.5.3 Course Outline

- **Number Theory** (4 hours)
  - Factorability
  - Properties of primes, including Fermat's Little Theorem
  - Modular arithmetic
  - Possible applications:
    - RSA encryption
    - Hash functions
- **Finite state machines** (6 hours)
  - Regular expressions
  - DFA
  - DFA and NDFA to accept regular expressions
- **Computability** (14 hours)
  - Introduction of one computing model (e.g., a WHILE or functional language, RAM model, or Turing machines)
  - Introduction of a universal machine/model/interpreter
  - Countable and uncountable sets
  - Diagonalization and proof by contradiction (e.g., uncountability of reals)
  - Undecidability
  - Church's Thesis and other computational models (e.g., from those listed above)
  - Decidable and semi-decidable sets
  - Possible applications:
    - # functions on integers is uncountable, # programs is countable
    - Halting Problem (proof of unsolvability using initially-introduced computing model)
- **Probability & Elementary Statistics** (8 hours)
  - Independence
  - Conditional probability
  - Descriptive statistics
  - Possible applications:
    - Average and worst case analysis
    - Shape of search trees (without explicit balancing)
    - Analysis of hashing algorithms
- *Total:* 32 hours.