Computing Curricula 2001

The Joint Task Force on Computing Curricula
IEEE Computer Society
Association for Computing Machinery

— DRAFT —
(March 6, 2000)

Composition of the Curriculum 2001 Joint Task Force

Preface

This document represents the first public report of the Computing Curricula 2001 project (CC2001)—a joint undertaking of the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) and the Association for Computing Machinery (ACM) to develop curricular guidelines for undergraduate programs in computing. The final report from the CC2001 Task Force is scheduled for publication in 2001. At this point in the process, one year before the anticipated publication date, we have laid much of the foundation for the final report. At the same time, we have a good deal of work ahead of us in the coming year. Our reason for circulating this early draft is to get feedback from the computing community—practitioners, educators, and students—on our work to date and on our overall directions for the project.

At the early task force meetings, we had decided—as the earlier curriculum committees dating back to Curriculum ’68 had done—to direct our attention only to curricula in computer science and computer engineering. The field of computing, however, has become much broader in recent years and now incorporates many new disciplines that have established their own identity independent of traditional computer science. The feedback that we received from our prospective audience overwhelmingly supported our taking a broader view of the discipline. In this draft, we present an outline of how we intend to expand our scope without becoming so general that the report loses much of its impact.

To date, we have accomplished the following tasks:

- Completed a survey and evaluation of the impact of Computing Curricula 1991
- Assessed the major changes in the discipline over the past decade
- Articulated a set of principles to guide our work
- Developed a proposed organizational structure and strategy for the final report
- Established knowledge area focus groups to define topics for specific areas
- Reviewed the reports of those working groups
- Drafted a list of areas and topics that comprise the body of knowlege for computer science
- Proposed a set of core topics for undergraduates majoring in computer science
- Created pedagogy focus groups to consider broad issues in computing education

The major tasks that remain to be done include

- Update the structure of the report in response to feedback from this draft
- Ask the pedagogy focus groups to update their reports in light of the broader scope
- Obtain final reports for the pedagogy groups and integrate them into the report
- Finalize the definition of the core so that it encompasses more of the breadth in the computing field
- Develop specific models for courses that cover the core areas
- Write the chapters on pedagogical strategies

Chapter 1
Introduction

In the fall of 1998, the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) and the Association for Computing Machinery (ACM) established a joint task force to undertake a major review of curriculum guidelines for undergraduate programs in computing. The charter of the task force was expressed as follows:

> To review the Joint ACM and IEEE/CS Computing Curricula 1991 and develop a revised and enhanced version for the year 2001 that will match the latest developments of computing technologies in the past decade and endure through the next decade.

This report—*Computing Curricula 2001* or *CC2001*for short—builds on several earlier reports including *Curriculum '68* [3], *A Curriculum in Computer Science and Engineering* [19], *Curriculum '78* [4], *The 1983 Model Program in Computer Science and Engineering* [17], and *Computing Curricula 1991* [39]. The relationship of this report to its predecessors—along with the lessons that the CC2001 Task Force derived from them—is discussed in Chapter 2.

Under our charter, the central goal of the CC2001 effort is to revise *Computing Curricula 1991* so that it incorporates the developments of the past decade. Computing has expanded and changed dramatically over that time. One of our first tasks, therefore, was to identify the nature and scope of those changes. The evolution of computing and the effect of that evolution on computing curricula are described in Chapter 3.

From a curricular perspective, one of the most profound changes over the past decade has been a substantial broadening of the discipline. In its early years, computing was often identified with computer science, which draws its foundations primarily from mathematics and electrical engineering. The curriculum reports cited earlier in this chapter reflect this history and focus on curriculum development in those branches of computing that follow most directly from this tradition: computer science and computer engineering. Today, computing has grown to such an extent that these two areas no longer cover what has become a much more diverse discipline. Computing is now a critical part of many academic fields, from business management to molecular biology. In this report, we have tried to understand the structure of that larger discipline and the ways in which the traditional body of knowledge associated with computing fits into that larger world. The question of the increasing breadth of the discipline and its impact on curriculum design is discussed in Chapter 4.

Our consideration of the effectiveness of past reports, the changes over the past decade, and the overall broadening of the discipline have led us to articulate a set of principles that have guided the task force in preparing this report. Those principles are enumerated in Chapter 5 and applied to the specific concern of curriculum design in Chapter 6.

The remaining chapters of the report consist of strategic discussions of various aspects of the computing curriculum organized around general pedagogical themes rather than by specific subdiscipline. These themes, which correspond to the set of six Pedagogy Focus Groups established by the CC2001 Task Force, are as follows:

1. Introductory topics and courses
2. Supporting topics and courses
3. The computing core
4. Professional practices
5. Advanced courses and undergraduate research
6. Computing across curricula

Chapter 2
Lessons from Past Reports

In developing this report, the CC2001 Task Force did not have to start from scratch. We have benefited tremendously from past curriculum studies and are indebted to the authors of those studies for their dedicated efforts. As part of our early work on *Computing Curricula 2001*, we looked carefully at the most recent curriculum studies—particulary *Computing Curricula 1991*—to get a sense of how those studies have influenced computing education. By identifying which aspects of the previous reports have been successful and which have not, we can structure the CC2001 report to maximize its impact. This chapter offers an overview of the earlier reports and the lessons we have taken from them.

2.1 Historical background

Efforts to design model curricula for programs in computer science and computer engineering began in 1960s, shortly after the first departments in these areas were established. In 1968, following on a series of earlier studies [2, 13, 34], the Association for Computing Machinery (ACM) published *Curriculum '68* [3], which offered detailed recommendations for academic programs in computer science, along with a set of course descriptions and extensive bibliographies for each topic area.

Over the next decade, the discipline of computing developed rapidly, to the point that the recommendations in *Curriculum '68* became largely obsolete. During the 1970s, both the ACM and the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE-CS) appointed committees to develop a revised computing curriculum. In 1977, the Education Committee of the IEEE-CS published a report for programs in computer science and engineering [19]. The Computer Society's report was significant in that it took a broader view of the computing discipline, incorporating more engineering into the curriculum and bridging the gap between software- and hardware-oriented programs. Responding to the pressures generated by the rapid development of the computing field, the Computer Society updated its computer science and engineering curriculum in 1983 [17]. The ACM *Curriculum '68* report was superseded by a much more comprehensive *Curriculum '78,* which had a substantial impact on computing education. Among its contributions, *Curriculum '78* proposed a standard syllabus for a set of courses that encompassed the core knowledge of computer science as a discipline.

In the late 1980s, the Computer Society and ACM joined forces to undertake a more ambitious curriculum review, which was published as *Computing Curricula 1991* [39], hereafter referred to as CC1991. The CC1991 report is more comprehensive than its predecessors, but takes a different approach. Unlike *Curriculum '78* and the 1983 IEEE-CS report, each of which focused on identifying a standard syllabus for individual courses, CC1991 divides the body of knowledge associated with computing into individual *knowledge units.* Each knowledge unit in CC1991 corresponds to a topic that must be covered at some point during the undergraduate curriculum, although individual institutions have considerable flexibility to assemble the knowledge units into course structures that fit their particular needs. The appendix of the CC1991 report includes 11 sample implementations that show how the knowledge units can be combined to form courses at a variety of institutions.

2.2 Evaluation of Computing Curricula 1991

The decision to produce a new curriculum report was driven primarily by the enormous changes that have occurred in the computing discipline over the past decade. At the same time, there was also a perception among some computer science educators that CC1991 was not as influential as some of its predecessors. Although CC1991 is certainly more detailed, institutions have sometimes found it harder to adopt than *Curriculum '78* and the IEEE-CS model curriculum in computer science and engineering.

In order to understand both the strengths and the limitations of CC1991, the task force undertook an informal survey of computing educators. We developed a short questionnaire, which we then mailed to the chairs of all computer science departments in the United States and Canada. We also made the questionnaire available more generally through the World Wide Web. A copy of the questionnaire appears in Figure 2-1.

Over 98 percent of the respondents—we received 124 responses through the web and about 30 responses through regular mail—supported the concept of updating the CC1991 report. The survey responses also revealed the following general reactions:

- *Knowledge units are often not as useful as course or curriculum designs.* Although many respondents indicated that they liked the concept of knowledge units as a resource, there was strong sentiment for a greater emphasis on course design along with the knowledge units. Our survey revealed that many institutions continue to work with the curriculum models outlined in *Curriculum '78*, largely because it included specific course designs.
- *There is strong support for a more concrete definition of a minimal core.* CC1991 argues that all undergraduate programs in computer science should incorporate the entire collection of knowledge units in the nine areas that comprise the common requirements. If the area encompassing Introduction to a Programming Language is included, the knowledge units in the common requirements account for 283 hours of classroom time. As our discipline evolves, it is tempting to add new material to the required set, thereby increasing the number of hours mandated by the curriculum. Our survey revealed considerable support for the idea of identifying a smaller set of core topics that would serve as a foundation for more advanced study in a number of computing disciplines, including computer engineering, computer science, software engineering, information technology, and the like. The areas and structure of the more advanced courses could vary markedly depending on the nature of the institution, the academic program, and the needs and interests of individual students.

Figure 2-1. Questionnaire to assess the impact of Computing Curricula 1991

1. Did you use CC1991 in any way in the past?
2. If you are a college or university teacher, do you know if your department ever looked at or used CC1991?
3. If you answered yes to either question, how was it used, and what features of it were helpful?
4. Do you think there is a need to create CC2001? Why?
5. CC1991 had 10 main content areas. Do you think any new areas should be added? Any existing area deleted? Any existing area updated?
6. Do you believe CC2001 should provide guidelines about a minimal core? If so, what would that core include?
7. Do you have any suggestion about the format? CC1991 was designed in terms of knowledge units along with possible model curricula in terms of those knowledge units.
8. Have you any other comments or suggestions for updating CC1991?

- *The report should define more model curricula, with particular emphasis on diverse nature of educational resources, systems, and requirements at different academic departments throughout the country and the world.* Several of the earlier reports, particularly those that preceded CC1991, have focused too heavily on the curricular needs of academic computer science programs in four-year colleges and universities in the United States. Computing today is taught under many different names in a wide variety of institutions throughout the world. Respondents to our survey felt strongly that our revised report should serve a broad community.
- *Curriculum reports should pay greater attention to accreditation criteria for both computer science and computing engineering programs.* Accreditation is an important issue for many survey respondents in the United States. The structure of engineering accreditation, however, is changing markedly with the new criteria proposed by the Accreditation Board for Engineering and Technology (ABET) and the Computing Sciences Accreditation Board (CSAB) [1, 12]. Under the new guidelines, programs will be allowed much greater flexibility than they have enjoyed in the past but must provide a coherent rationale for their curriculum and demonstrate that it meets its stated goals. This report is designed not only to help institutions design their computing curriculum but also to assist them in the preparation of the underlying rationale they need to meet the new accreditation criteria.

Chapter 3
Computing and Change

Today, as we enter a new millennium, computing is an enormously vibrant field. From its inception just half a century ago, computing has become the defining technology of our age. Computers are integral to modern culture and are the primary engine behind much of the world's economic growth. The field, moreover, continues to evolve at an astonishing pace. New technologies are introduced continually, and existing ones become obsolete in the space of a few years.

The rapid evolution of the discipline has a profound effect on computing education, affecting both content and pedagogy. When CC1991 was published, for example, networking was not seen as a major topic area, accounting for only six hours in the common requirements. The lack of emphasis on networking is not particularly surprising. After all, networking was not yet a mass-market phenomenon, and the World Wide Web was little more than an idea in the minds of its creators. Today, a mere ten years later, networking and the web have become the underpinning for much of our economy. They have become critical foundations of the computing domain, and it is impossible to imagine that undergraduate programs would not devote significantly more time to this topic. At the same time, the existence of the web has changed the nature of the educational process itself. Modern networking technology enhances everyone's ability to communicate and gives people throughout the world unprecedented access to information. In most academic programs today—not only in computing but in other fields as well—networking technology has become an essential pedagogical tool.

The charter of the CC2001 Task Force asks us to "review the Joint ACM and IEEE/CS Computing Curricula 1991 and develop a revised and enhanced version for the year 2001 that will match the latest developments of computing technologies." To do so, we felt it was important to spend part of our effort getting a sense of what aspects of computing had changed over the last decade. We believe that these changes fall into two categories—technological and cultural—each of which have a significant effect on computing education. The major changes in each of these categories are described in the individual sections that follow.

3.1 Technological changes

Much of the change that affects computing comes from advances in technology. Many of these advances are part of a ongoing evolutionary process that has continued for many years. Moore's Law—the 1965 prediction by Intel founder Gordon Moore that microprocessor chip density would double every eighteen months—continues to hold true. As a result, we have seen exponential increases in available computing power that have made it possible to solve problems that would have been out of reach just a few short years ago. Other changes in the discipline, such as the rapid growth of networking after the appearance of the World Wide Web, are more dramatic, suggesting that change also occurs in revolutionary steps. Both evolutionary and revolutionary change affects the body of knowledge required for computing and the educational process.

Technological advancement over the past decade has increased the importance of many curricular topics, such as the following:

- The World Wide Web and its applications
- Networking technologies, particularly those based on TCP/IP
- Graphics and multimedia

- Embedded systems
- Relational databases
- Interoperability
- Object-oriented programming
- The use of sophisticated application programmer interfaces (APIs)
- Human-computer interaction
- Software safety
- Security and cryptography
- Application domains

As these topics increase in prominence, it is tempting to include them as undergraduate requirements. Unfortunately, the restrictions of most degree programs make it difficult to add new topics without taking others away. It is often impossible to cover new areas without reducing the amount of time devoted to more traditional topics whose importance has arguably faded with time, such as assembly language programming, formal semantics, and numerical analysis.

3.2 Cultural changes

Computing education is also affected by changes in the cultural and sociological context in which it occurs. The following changes, for example, have all had an influence on the nature of the educational process:

- *Changes in pedagogy enabled by new technologies.* The technological changes that have driven the recent expansion of computing have direct implications on the culture of education. Computer networks, for example, make distance education much more feasible, leading to enormous growth in this area. Those networks also make it much easier to share curricular material among widely distributed institutions. Technology also affects the nature of pedagogy. Demonstration software, computer projection, and individual laboratory stations have made a significant difference in the way computing is taught. The design of computing curricula must take into account those changing technologies.
- *The dramatic growth of computing throughout the world.* Computing has expanded enormously over the last decade. For example, in 1990, few households—even in the United States—were connected to the Internet. A U.S. Department of Commerce study [33] revealed that by 1999 over a third of all Americans had Internet access from some location. Similar growth patterns have occurred in most other countries as well. The explosion in the access to computing brings with it many changes that affect education, including a general increase in the familiarity of students with computing and its applications along with a widening gap between the skill levels of those that have had access and those who have not.
- *The growing economic influence of computing technology.* The dramatic excitement surrounding high-tech industry, as evidenced by the Internet startup fever of the past five years, has significant effects on education and its available resources. The enormous demand for computing expertise and the vision of large fortunes to be made has attracted many more students to the field, including some who have little intrinsic interest in the material. At the same time, the demand from industry has made it harder for most institutions to attract and retain faculty, imposing significant limits on the capacity of those institutions to meet the demand.
- *Greater acceptance of computing as an academic discipline.* In its early years, computing had to struggle for legitimacy in many institutions. It was, after all, a new

discipline without the historical foundations that support most academic fields. To some extent, this problem persisted through the creation of CC1991, which was closely associated with the *Computing as a Discipline* report [14]. Partly as a result of the entry of computing technology into the cultural and economic mainstream, the battle for legitimacy has largely been won. On many campuses, computing has become one of the largest and most active disciplines. There is no longer any need to defend the inclusion of computing education within the academy. The problem today is to find ways to meet the demand.

- *Broadening of the discipline.* As our discipline has grown and gained legitimacy, it has also broadened in scope. In its early years, computing was primarily focused on computer science, which had its roots in mathematics and electrical engineering. Over the years, an increasing number of fields have become part of a much larger, more encompassing discipline of computing. Our CC2001 Task Force believes that understanding how those specialties fit together and how the broadening of the discipline affects computing education must be a critical component of our work.

Chapter 4
The Expanding Scope of Computing

As we observe in the previous chapter, one of the major changes in computing over the past decade is the enormous broadening of the field. In arguing for establishing a broader view of computing as a profession, Peter Denning has enumerated two dozen professional specialties that fall into the domain of information technology, as shown in Figure 4-1. While it is possible to debate this classification scheme, there is no doubt that the discipline of computing has indeed expanded in recent years.

The expansion of the discipline beyond the traditional boundaries of computer science certainly has a significant impact in the broad domain of computing education. At the same time, the problem of developing a coherent curriculum for the computing field as a whole is an extremely difficult undertaking, given the enormous breadth of specialties within the field. In our early meetings, the CC2001 Task Force decided that viewing computing narrowly would give us the best chance of reaching closure in a reasonable time frame. The earlier curriculum studies took this approach, even as new computing specialties began to appear on the scene. Thus, our initial position was that CC2001, like CC1991 before it, would focus on computer science and computer engineering. Although we recognized the importance of software engineering and information systems as disciplines in their own right, we regarded them as being outside our purview. Professional bodies already exist for those disciplines, and it is certainly important that curriculum design in those specialties be undertaken by people with the relevant expertise. Review committees in several of these areas have recently published new curriculum studies, such as the MSIS 2000 curriculum for information systems [21] and the Software Engineering Body of Knowledge (SWEBOK) definition [38].

During our early presentations of the curriculum outline, however, it became clear that our constituency wanted the CC2001 report to take a broader view. These sentiments were expressed strongly at the 1999 Frontiers in Education (FIE) conference in Puerto Rico, where the audience at the CC2001 panel uniformly supported the idea of incorporating the breadth of the discipline into the curriculum design for the following reasons:

- *The new disciplines that now comprise the broad field of computing are at least as important to the academic computing curriculum as traditional computer science.* In the CC1991 report, "the term *computing* is used to encompass the labels *computer science* [and] *computer science and engineering*" but specifically excludes programs in other computing disciplines, such as information systems. It seems presumptuous for computer science and computer engineering to lay claim over the entire computing discipline.

Figure 4-1. The expanding discipline of computing

Artificial intelligence	Human-computer interaction	Network engineering
Bioinformatics	Information science	Performance analysis
Cognitive science	Information systems	Scientific computing
Computational science	Instructional design	Software architecture
Computer science	Knowledge engineering	Software engineering
Database engineering	Learning theory	System administration
Digital library science	Management information systems	System security and privacy
Graphics	Multimedia design	Web service design
		Source: Peter Denning [16]

- *Narrowing the discipline of computing to its traditional components limits the evolution of the discipline through synergies with related fields.* As computing program increase in both breadth and size, it is important to resist the temptation to compensate by creating highly specialized, independent subdisciplines that rarely share ideas. Much of the vitality in computing today comes from the interaction of theory and practice. If applications of computing become increasingly separate from the theoretical underpinnings of computer science, both theory and practice will suffer as a result.
- *Introductory courses that are designed only for potential computer science majors will not serve the best interests of computing education as a whole.* At most institutions today, computer science has a high service load, in the sense that many of its courses are taken by many students who will major in other areas. Computer science today serves as a foundation for a broad range of disciplines, in much the same way that mathematics has done for many years. Academic departments of mathematics, however, are often criticized for concentrating their resources on the pure aspects of the field, even though most of their students, particularly at the introductory level, are interested in more applied topics. In the interest of the broad computing curriculum, computer science should not follow that path. Students, moreover, need to understand the range of options that are available in the computing domain, and it is important for introductory courses not only to prepare students for a range of disciplines but to offer them guidance about the possibilities.

The steering committee of the CC2001 Task Force discussed the question of scope extensively at its meeting of January 2000. The arguments from the respondents at the FIE conference were compelling, but we were nonetheless concerned about expanding our coverage of the computing curriculum to accommodate the much wider vision of the discipline. For one thing, the members of the CC2001 Task Force are not experts in many of the expanded specialty areas and would need to rely on professionals and educators in those domains for curriculum recommendations. A more important concern was whether expanding the curriculum to encompass the broad range of computing disciplines would leave us with any semblance of commonality among the disparate subfields. If the overlap in undergraduate curricula were in fact small, broadening the report might end up reducing its effectiveness for computer science programs without adding much to programs in related areas.

To get a sense of the scale of the overlap among such disciplines as computer science, computer engineering, software engineering, and information systems, the CC2001 Task Force tried to enumerate the set of concept and skills that we would expect undergraduates to know, regardless of discipline. The results of that exercise are show in Figure 4-2. This list is not intended to be comprehensive, but demonstrates clearly that there are many common themes that unite the computing discplines..

We therefore decided—somewhat late in the process—to broaden our focus and develop guidelines for computing curricula that cover a wider range of specialties than the earlier curriculum reports from IEEE-CS and ACM. We do not intend to preempt the work of curriculum committees in related disciplines, but will instead incorporate the excellent work that has already been done in those areas. We will continue our work to define a body of knowledge for computer science. We will, however, also look at how this body of knowledge fits into a larger framework that includes other computing disciplines as well.

Figure 4-2. Skills common to all computing disciplines

By the time of graduation, every undergraduate student of computing should:

- Know what a computer is and understand the functionality of its major components
- Understand the difference between binary and decimal representations and the effect of representation on numeric precision
- Be able to use standard computer-based tools, including e-mail, word processing, and spreadsheets
- Understand the overall mechanics of file systems and directory hierarchies
- Understand the concept of programming language translation and the distinction between interpreters and compilers
- Understand the basic functions of an operating system
- Appreciate the fact that languages and operating systems create a hierarchy of virtual machines
- Understand the principle of abstraction and its applications to computing
- Be able to write simple programs in some language
- Understand fundamental data structures and be able to incorporate them into programs
- Understand the distinction between procedural and object-oriented programming
- Be able to apply basic problem-solving techniques
- Appreciate the concept of an algorithm and the process of algorithmic development
- Recognize the importance of debugging and be able to use testing and debugging strategies
- Have some understanding of algorithmic efficiency and the fundamental limits of computing
- Understand and be able to apply fundamental principles of software engineering
- Recognize the existence and utility of standards in the computing field
- Know what a network is and have a general understanding of how it works
- Understand the structure of the World Wide Web and simple techniques for creating a web page
- Be familiar with the concepts of event-driven and real-time programming
- Understand the basics of the client-server model
- Understand the functionality of databases and information systems
- Be familiar with the fundamental principles of human-computer interaction
- Have sufficient familiarity with discrete mathematics to understand basic logic and the importance of formalism
- Appreciate the range of areas to which computing can be applied
- Have a rough understanding of the distinctions among the various computing disciplines
- Understand something about the economics of computing
- Recognize the ethical, legal, and professional responsibilities associated with work in the computing field

Chapter 5
Principles

Based on our analysis of past curriculum reports and the changes in our discipline outlined in the preceding chapters, the CC2001 Task Force has articulated the following set of principles:

1. *Computing has become an extremely broad discipline that extends well beyond the traditional boundaries of computer science.* Given the number of subdisciplines that have emerged from computer science in recent years, it is no longer reasonable to regard the overall computing curriculum as being essentially identical to computer science. Colleges and universities must be sensitive to the emergence of these new fields and ensure that the foundational courses in computing serve a wide audience.
2. *Despite its growing breadth, computing remains an integrated field of study that draws its foundations from many well-established disciplines.* In all of its subdisciplines, computing draws on basic foundations in mathematics, science, engineering, psychology, management, and many other fields, each of which requires the integration of theory and practice. We endorse the position articulated in the CC1991 report that "mastery of the discipline includes not only an understanding of basic subject matter, but also an understanding of the applicability of the concepts to real-world problems." Particular attention must be paid in the undergraduate curriculum to the importance of laboratory work as it reinforces student mastery of concepts and their application to solving real-life problems in diverse domains.
3. *The rapid evolution of the computing discipline requires an ongoing review of the corresponding curriculum* . Given the pace of change in our discipline, the process of updating the curriculum once a decade has become unworkable. The professional associations in this discipline must establish an ongoing curriculum review process that allows individual components of the curriculum to be updated on a recurring basis.
4. *CC2001 must go beyond knowledge units to offer significant guidance in terms of individual course design.* Although the knowledge-unit structure used in CC1991 can serve as a useful framework, most institutions need more detailed guidance. For such institutions, CC2001 will be effective only to the extent that it defines a small set of alternative models—preferably between two and four—that assemble the knowledge units into reasonable, easily implemented courses. Articulating a set of well-defined models will make it easier for institutions to share pedagogical strategies and tools. It will also provide a framework for publishers who provide the textbooks and other materials for those courses.
5. *CC2001 must identify a relatively small set of core concepts and skills that are required of all computing students.* Historically, the growth of the discipline has led to a parallel expansion in the computer science core. As important new topics emerge, there is a strong temptation to include them as undergraduate requirements. Over the last decade, the discipline has expanded to such an extent that it is no longer possible simply to add new topics without taking others away. Given the constraints of an undergraduate degree, it is difficult to include new topics from software engineering, human-computer interaction, networks, and graphics while retaining the traditional presentation of such classical topics as assembly language programming, compiler construction, and automata theory in the traditional way. It seems likely that the best strategic approach is to *reduce* the size of the required computer science core, allowing greater flexibility to include new topics and adapt the curriculum to changes as they occur. The CC2001 Task Force has agreed that "the core will consist of those

topics for which there is a broad consensus that the topic is essential to undergraduate degrees that include computer science, computer engineering, and other similarly named programs." This definition is meant to encompass the essential requirements common to all undergraduate programs. At the same time, the core does not in itself constitute a complete undergraduate curriculum, but must be supplemented by additional courses that may vary by institution, field of study, or individual student.

6. *CC2001 must provide guidelines for courses beyond the required core.* In addition to specifying the fundamental core of the discipline, CC2001 must provide guidelines for advanced courses that serve as technical electives in more advanced areas.
7. *CC2001 must be international in scope.* The intended audience for CC2001 is not limited to the United States alone, but must instead be useful for computing professional across the world. Curricular requirements abroad are often significantly different from those in the United States.
8. *The development of CC2001 must involve significant industry participation.* Most students who graduate from undergraduate computing programs take jobs in industry, often without seeking more advanced education. To ensure that graduates are properly prepared for the demands they will face in those positions, we believe it is essential to involve practitioners in the design, development, and implementation of new curricula.
9. *CC2001 must include professional practice as an integral component of the undergraduate curriculum.* Because computing is an integrated discipline, it is essential for undergraduate programs to emphasize the practical aspects of the discipline along with the theoretical ones. Today, much of the practical knowledge associated with computing exists in the form of professional practices that exist in industry. To work successfully in those environments, students must be exposed to those practices as part of their education. These practices, moreover, extend beyond computing-specific skills to encompass a wide range of activities including management, ethics and values, written and oral communication, and the ability to work as part of a team.
10. *CC2001 must strive to be useful for its intended audience.* In order to be useful, CC2001 must (1) support programs seeking accreditation from the Computer Science Accreditation Board (CSAB), the Accreditation Board for Engineering and Technology (ABET), and similar organizations outside the United States, (2) be sufficiently general to fulfill the needs of most computing programs with varying emphases and objectives, and (3) be flexible enough to accommodate future advances in the computing discipline in a timely fashion.

Chapter 6
Defining a Curriculum

In order to develop a curriculum, it is essential to develop a detailed understanding of the knowledge encompassed by that discipline. As we note in Chapter 4, the CC2001 Task Force has decided that this accounting must be broad enough to accommodate the range of subdisciplines that come under the general rubric of computing. For areas such as information systems and software engineering, our task force can rely on recent reports issued by curriculum committees in those areas [21, 38]. For computer science and computer engineering, the CC2001 Task Force has the central responsibility for developing that updated body of knowledge. To this end, we have identified a set of knowledge areas and appointed *knowledge area focus groups* to define the body of knowledge for that area. This process is described in more detail in the section entitled "Defining the body of knowledge" below.

Although the definition of the body of knowledge represents a central task of the Computing Curricula 2001 project, it is not sufficient on its own. In some ways, viewing the entire computing curriculum as a body of knowledge misses the forest for the trees. To develop a more complete vision of the curriculum and its implementation, it is important to adopt a more holistic perspective, in which broader issues are allowed to cut across the lines represented by individual knowledge areas. The knowledge areas, after all, reflect the boundaries of established subdisciplines. Using these boundaries as the organizing principle for the curriculum has the conservative structure of reinforcing the existing structure. New curricular ideas and pedagogical strategies often emerge from explorations that transcend those disciplinary boundaries.

To encourage the development of a holistic vision of the curriculum, the CC2001 Task Force established six *pedagogy focus groups,* with the following areas of concern:

1. Introductory topics and courses
2. Supporting topics and courses
3. The computing core
4. Professional practices
5. Advanced study and undergraduate research
6. Computing across curricula

The charter for each of these groups appears later in this chapter, and the final reports from each group will—in later drafts—constitute the next six chapters of the report.

6.1 Defining the body of knowledge

Computing Curricula 1991 organized the undergraduate curriculum by dividing it into nine knowledge areas. Over the last decade, the discipline of computing has grown substantially, to the point that the nine areas identified by Computing Curricula 1991 are no longer sufficient to encompass the knowledge that students are likely to encounter in an undergraduate curriculum. After experimenting with several organizational structures, the CC2001 Task Force has defined an expanded set of 14 knowledge areas, as shown in Figure 6-1:

This revised list of knowledge areas differs from that used in Computing Curricula 1991 in the following ways:

- *Discrete Structures (DS) has been added as a separate area.* In Computing Curricula 1991, discrete mathematics appears only as a prerequisite for topics that require

Figure 6-1. Knowledge areas in Computing Curricula 2001

0. Discrete Structures (DS)
1. Programming Fundamentals (PF)
2. Algorithms and Complexity (AL)
3. Programming Languages (PL)
4. Architecture (AR)
5. Operating Systems (OS)
6. Human-Computer Interaction (HC)
7. Graphics, Visualization, and Multimedia (GR)
8. Intelligent Systems (IS)
9. Information Management (IM)
10. Net-Centric Computing (NC)
11. Software Engineering (SE)
12. Computational Science (CN)
13. Social, Ethical, and Professional Issues (SP)

mathematical maturity. The assumption, presumably, is that such mathematical maturity would come from prior mathematical training or from college-level courses in mathematics. Unfortunately, most mathematics courses in universities—responding to the needs of the physical sciences and classical engineering fields—focus on calculus and other aspects of continuous mathematics rather than on the discrete mathematics required for most computing disciplines. As a result, the necessary discrete mathematics is often taught by faculty in computer science or related departments. The CC2001 Task Force has chosen to emphasize the dependency of computing on discrete mathematics by including it as a separate knowledge area.

- *The need to include instruction in the use of a programming language has been made explicit by the inclusion of a distinct knowledge area on Programming Fundamentals (PF).* Computing Curricula 1991 defined a knowledge area entitled "Introduction to a Programming Language" but identified it as an *optional* component of the curriculum. In part, introductory programming was left out of the common requirements in the hope that an increasing number of students would acquire the necessary skills and experience in secondary school. Unfortunately, this prediction has not been realized. Despite the overwhelming increase in the availability of computing resources to secondary schools, many students arrive at universities with little understanding of programming discipline and the basic principles of software design. For this reason, the CC2001 Task Force has chosen to define a separate Programming Fundamentals area that enumerates the basic programming skills that all students of computing must acquire to prepare themselves for more advanced study.
- *The knowledge area on Social, Ethical, and Professional Issues (SP) has been integrated into the structure of the curriculum in a way that gives it equal weight with the other knowledge areas.* Since the publication of Computing Curricula 1991, there has been a growing consensus that all students of computing must be made aware of the social implications of their work and the ethical responsibilities of being a computing professional. This topic has been identified as a "tenth strand" in the computing curriculum, on an equal footing with the nine subject areas identified by Computing Curricula 1991 [27]. The CC2001 Task Force has therefore included social, ethical, and professional issues as part of the body of knowledge.
- *Graphics, Visualization, and Multimedia (GR) and Net-Centric Computing (NC) have been added as separate knowledge areas.* Many areas of computing have expanded dramatically since the publication of Computing Curricula 1991. As a result, some areas that were formerly topics within a more general area have grown to such an extent that they can no longer fit appropriately into the older structure.

Figure 6-2. Charter for the Knowledge Area Focus Groups

Each focus group assigned to a specific focus area will have a chair and preferably a co-chair. The chair and co-chair of each focus area must be experts in the assigned area. Each focus group may invite up to five additional members. Each focus group will:

1. Review and firm up the scope of the focus area drafted by the joint task force members.
2. Finalize the list of individual topics associated with the focus area.
3. Comment on the three processes—theory, abstraction, and design—as well as the breadth and depth issues documented in Computing Curricula 1991.
4. Decide the required mathematics and physical sciences.
5. Highlight changes compared to Computing Curricula 1991, if applicable.
6. Separate the topics into two levels, corresponding to core topics required of all students in computing and more advanced electives.
7. Suggest model courses and the corresponding lecture/lab hours, with specific course objectives and expected learning outcome, by indicating which topics are included in each course.

After making a preliminary identification of the knowledge areas in early 1999, the CC2001 Task Force appointed a knowledge area focus group to take responsibility for each of the areas. The charge to each knowledge area focus group appears in Figure 6-2. The members of each focus group appear in the acknowledgments in Chapter 14.

The knowledge area focus groups deliberated over the spring of 1999 and submitted preliminary reports to the CC2001 Task Force. The Computing Curricula 1991 steering committee reviewed these reports at its meeting in June 1999. The review process at that meeting had three goals:

1. *To monitor the work of each focus group and make sure that it had fulfilled its charge.* If the steering committee could identify omissions in the report, the focus groups were asked to go back and resupply the missing material.
2. *To review the set of knowledge units identified with each area and assess whether those knowledge units provide adequate coverage of the area.* Once again, if the steering committee found problems in the focus group report, it asked the focus group to provide any necessary updates.
3. *To determine which knowledge units in each area would be part of the required core.* Because each knowledge area focus group is composed of experts in that area, the individuals are likely to have a strong predisposition to be proponents for that area. As a result, the CC2001 Task Force expected each knowledge area group to identify more core topics than could be justified under our minimalist definition of the core. As noted in Chapter 5, the steering committee had agreed that “the core will consist of those topics for which there is a broad consensus that the topic is essential to undergraduate degrees. . . .” If the steering committee itself could not find such a consensus in support of a topic, we eliminated it from the core.

After some additional negotiations with the focus groups, the CC2001 Task Force has identified a set of topics to go with each of the 14 areas. These topics are shown in Figure 6-3.

Figure 6-3. Tentative list of topics in the computer science body of knowledge

DS. Discrete Structures
- DS1. Functions, relations, and sets
- DS2. Basic logic
- DS3. Proof techniques
- DS4. Basics of counting
- DS5. Graphs and trees

PF. Programming Fundamentals
- PF1. Algorithms and problem-solving
- PF2. Fundamental programming constructs
- PF3. Basic data structures
- PF4. Recursion
- PF5. Abstract data types
- PF6. Object-oriented programming
- PF7. Event-driven and concurrent programming
- PF8. Using modern APIs

AL. Algorithms and Complexity
- AL1. Basic algorithmic analysis
- AL2. Algorithmic strategies
- AL3. Fundamental computing algorithms
- AL4. Distributed algorithms
- AL5. Basic computability theory
- AL6. The complexity classes P and NP
- AL7. Automata theory
- AL8. Advanced algorithmic analysis
- AL9. Cryptographic algorithms
- AL10. Geometric algorithms

PL. Programming Languages
- PL1. History and overview of programming languages
- PL2. Virtual machines
- PL3. Introduction to language translation
- PL4. Language translation systems
- PL5. Type systems
- PL6. Models of execution control
- PL7. Declaration, modularity, and storage management
- PL8. Programming language semantics
- PL9. Functional programming paradigms
- PL10. Object-oriented programming paradigms
- PL11. Language-based constructs for parallelism

AR. Architecture
- AR1. Digital logic and digital systems
- AR2. Machine level representation of data
- AR3. Assembly level machine organization
- AR4. Memory system organization
- AR5. I/O and communication
- AR6. CPU implementation

OS. Operating Systems
- OS1. Operating system principles
- OS2. Concurrency
- OS3. Scheduling and dispatch
- OS4. Virtual memory
- OS5. Device management
- OS6. Security and protection
- OS7. File systems and naming
- OS8. Real-time systems

HC. Human-Computer Interaction
- HC1. Principles of HCI
- HC2. Modeling the user
- HC3. Interaction
- HC4. Window management system design
- HC5. Help systems
- HC6. Evaluation techniques
- HC7. Computer-supported collaborative work

GR. Graphics, Visualization, and Multimedia
- GR1. Graphic systems
- GR2. Fundamental techniques in graphics
- GR3. Basic rendering
- GR4. Basic geometric modeling
- GR5. Visualization
- GR6. Virtual reality
- GR7. Computer animation
- GR8. Advanced rendering
- GR9. Advanced geometric modeling
- GR10. Multimedia data technologies
- GR11. Compression and decompression
- GR12. Multimedia applications and content authoring
- GR13. Multimedia servers and filesystems
- GR14. Networked and distributed multimedia systems

IS. Intelligent Systems
- IS1. Fundamental issues in intelligent systems
- IS2. Search and optimization methods
- IS3. Knowledge representation and reasoning
- IS4. Learning
- IS5. Agents
- IS6. Computer vision
- IS7. Natural language processing
- IS8. Pattern recognition
- IS9. Advanced machine learning
- IS10. Robotics
- IS11. Knowledge-based systems
- IS12. Neural networks
- IS13. Genetic algorithms

IM. Information Management
- IM1. Database systems
- IM2. Data modeling and the relational model
- IM3. Database query languages
- IM4. Relational database design
- IM5. Transaction processing
- IM6. Distributed databases
- IM7. Advanced relational database design
- IM8. Physical database design

NC. Net-Centric Computing
- NC1. Introduction to net-centric computing
- NC2. The web as an example of client-server computing
- NC3. Building web applications
- NC4. Communication and networking
- NC5. Distributed object systems
- NC6. Collaboration technology and groupware
- NC7. Distributed operating systems
- NC8. Distributed systems

SE. Software Engineering
- SE1. Software processes and metrics
- SE2. Software requirements and specifications
- SE3. Software design and implementation
- SE4. Verification and validation
- SE5. Software tools and environments
- SE6. Software project methodologies

CN. Computational Science
- CN1. Numerical analysis
- CN2. Scientific visualization
- CN3. Architecture for scientific computing
- CN4. Programming for parallel architectures
- CN5. Applications

SP. Social, Ethical, and Professional Issues
- SP1. History of computing
- SP2. Social context of computing
- SP3. Methods and tools of analysis
- SP4. Professional and ethical responsibilities
- SP5. Risks and liabilities of safety-critical systems
- SP6. Intellectual property
- SP7. Privacy and civil liberties
- SP8. Social implications of the Internet
- SP9. Computer crime
- SP10. Economic issues in computing
- SP11. Philosophical foundations of ethics

6.2 Defining the pedagogical framework

As noted in the introduction to this chapter, defining a body of knowledge for a discipline is not the same as defining a curriculum. While the work of the knowledge area groups in defining the topics that are important within the undergraduate curriculum, it is also important to take a holistic look at the curriculum that transcends the traditional disciplinary boundaries. The CC2001 Task Force established six pedagogy focus groups with the following charges:

1. **Introductory topics and courses**
   - Identify the goals of the introductory curriculum, typically corresponding to the first year of study
   - Report on both the strengths and weaknesses of the traditional programming-first approach at reaching these goals
   - Provide a short list (ideally consisting of between two and four well-specified options) of alternative approaches

2. **Supporting topics and courses**
   - Specify goals of courses that support undergraduate computing curricula
   - Identify a minimal list of supporting courses deemed essential to an undergraduate program, as well as additional supporting courses

3. **The computing core**
   - Specify material that is deemed essential to a foundation in computing
   - Develop the core as a curricular alternative to the traditional approach of organizing programs around artifacts (e.g., courses in compilers, operating systems, databases, and so forth)

4. **Professional practices**
   - Report on effective education in various aspects of professional practices and on how these needs can be integrated into other courses in the curriculum.

5. **Advanced study and undergraduate research**
   - Report on coursework beyond the core
   - Include a specification of how many courses (as a minimum) should be included to produce a reasonable undergraduate experience
   - Report on undergraduate research, including an evaluation of various existing models

6. **Computing across curricula**
   - Articulate those aspects of the computing discipline relevant to all citizens and academic disciplines and propose guidelines for the role computer science can play in helping students achieve that knowledge.

The pedagogy focus groups were formed later in the review process than the counterpart focus groups examining the knowledge areas. The work of the pedagogy focus groups, moreover, is often dependent on the results of the knowledge area focus groups to define the scope of knowledge. The pedagogy focus groups have therefore had a shorter time in which to work.

Despite the limited time, most of the pedagogy focus groups produced draft reports during the second half of 1999. These reports outline the overall direction for each group and serve as a foundation for work over the coming year. These reports, however, were drafted prior to the January 2000 decision by the CC2001 Task Force to expand its scope beyond computer science and computer engineering. The broadening of the definition of the discipline has a substantial effect on the work of many of the pedagogy groups, and the CC2001 Task Force needs to go back to those groups and ask them to review their preliminary reports in light of that change.

In light of the recent change in direction, we have chosen not to include the draft reports from the pedagogy focus groups in this version of the report. They will instead be included in the next release, after the groups have had time to integrate the change in scope.

Chapter 7-12
Reports from the Pedagogy Focus Groups

Note: These chapters will consists of the reports from the six pedagogy focus groups. As discussed at the end of Chapter 6, these groups have not yet had an opportunity to address the expansion in scope to include a broader range of computing disciplines.

Chapter 13
Strategy and Tactics

Note: This chapter will consist of several sections that discuss issues that relate to the design and implementation of computing curricula but are not necessarily a part of it. At the moment, only the section on mathematics and science requirements is included. I expect that the final report will include additional sections on at least the following topics:

- *Service courses*
- *Laboratories and hardware requirements*
- *Faculty and staff*
- *Accreditation issues*
- *Coordination with secondary school curricula*
- *Articulation issues for two-year colleges*

Mathematics and science requirements

In terms of mathematics, the CC2001 Task Force recommends that all students be required to take a one-semester course in each of the following:

- *Discrete mathematics.* All students need exposure to the tools of discrete mathematics. The required concepts are detailed in the description of the Discrete Structures (DS) knowledge area.
- *Probability and statistics.* All students should have some background in basic statistical techniques, focusing primarily on discrete probability with some coverage of mathematical expression and standard statistical measures (normal and Poisson), with an emphasis on the practical application of these techniques to problems that arise in the computing discipline.
- *Additional mathematics.* Students should take at least one additional course to develop mathematical sophistication, which might be in any of a number of areas including calculus, linear algebra, number theory, or symbolic logic. The choice may be dependent upon institutional or departmental requirements or individual student need for advanced courses in computer science.

For science, there is a need for a genuine exposure to the scientific method. We believe that any science requirement should allow substantial flexibility in terms of subject matter, but should include a lab component to provide actual experience with the scientific method.

Chapter 14
Acknowledgments

Many people have contributed to the CC2001 project over the past year. In addition to the members of the Task Force, the following individuals have served on at least one of the focus groups: Ishfaq Ahmad, Donald J. Bagert, Bruce Barnes, Kevin W. Bowyer, Kim Bruce, Amy S. Bruckman, James Caristi, Morris Chang, Yiming Chen, Ashraful Chowdhury, Alan Clements, Thad Crews, George Crocker, Steve Cunningham, Nell Dale, Gordon Davies, Susan Dean, J. Philip East, Ed C. Epp, Sue Fitzgerald, Edward A. Fox, Benjamin Goldberg, Dina Golden, Mark Guzdial, Chris Haynes, Xudong He, Tom Hilburn, Cay Horstmann, Michel Israel, Anil Jain, Barbara Jennings, Ricardo Jimenez-Peris, Ioannis A. Kakadiaris, Amruth Kumar, Gary Leavens, Ernst Leiss, James Lin, Chengwen Liu, Ming T. (Mike) Liu, Philip Machanick, Raghu Machiraju, John Mallozzi, Bill Marion, Bruce R. Maxim, Chris McDonald, Susan A. Mengel, John Mitchell, Mike Murphy, Wakid Najjar, Thomas L. Naps, Patricia Nettnin, Gary Nutt, Jehan-Francois Paris, Marta Patino-Martinez, Yale Patt, Richard E. Pattis, Rhys Price-Jones, Anne-Louise Radimsky, Kay A. Robbins, Sartaj Sahni, Carolyn Schauble, Ben Schneiderman, Henning Schulzrinne, Charles Shipley, Shai Simonson, Carl Smith, Milan Sonka, Lynn Andrea Stein, George Stockman, Bobby Thrash, D. Singh Tomer, Ron Vetter, Henry Walker, Tony Wasserman, Laurie Honour Werth, Curt M. White, Barry Wilkinson, and Terry Winograd.

Please let us know if we have inadvertently omitted any names so that we can correct the list prior to publishing the final version.

Bibliography

1. Accreditation Board for Engineering and Technology, Inc. Accreditation policy and procedure manual. Technical report, November 1999.
2. ACM Curriculum Committee on Computer Science. An undergraduate program in computer science—preliminary recommendations. Communications of the ACM, 8(9):543-552, September 1965.
3. ACM Curriculum Committee on Computer Science. Curriculum '68: Recommendations for the undergraduate program in computer science. Communications of the ACM, 11(3):151-197, March 1968.
4. ACM Curriculum Committee on Computer Science. Curriculum '78: Recommendations for the undergraduate program in computer science. Communications of the ACM, 22(3):147-166, March 1979.
5. ACM Special Interest Group on Computer-Human Interaction. ACM SIGCHI Curricula for Human-Computer Interaction. New York: Association for Computing Machinery, 1992.
6. ACM Two-Year College Education Committee. Guidelines for associate-degree and certificate programs to support computing in a networked environment. New York: The Association for Computing Machinery, September 1999.
7. W. Bennett. A position paper on guidelines for electrical and computer engineering education. IEEE Transactions in Education, E-29(3):175-177, August 1986.
8. John Beidler, Richard Austing, and Lillian Cassel. Compuing programs in small colleges. Communications of the ACM, 28(6):605-611, June 1985.
9. Carnegie Commission on Science, Technology, and Government. Enabling the future: Linking science and technology to societal goals. New York: Carnegie Commission, September 1992.
10. Computing Science and Telecommunications Board. Realizing the information future. Washington DC: National Academy Press, 1994.
11. Computing Science and Telecommunications Board. Being fluent with information technology. Washington DC: National Academy Press, 1999.
12. Computing Sciences Accreditation Board. Criteria for accrediting programs in computer science in the United States. Technical report in preparation for June 2000, Version 1.0, January 2000. `http://www.csab.org/criteria2k_v10.html`.
13. COSINE Committee. Computer science in electrical engineering. Washington, DC: Commission on Engineering Education, September 1967.
14. Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen B. Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. Communications of the ACM, 32(1):9-23, January 1989.
15. Peter J. Denning. Computing the profession. Educom Review, November 1998.
16. Peter J. Denning. Our seed corn is growing in the commons. Information Impacts Magazine, March 1999. `http://www.cisp.org/imp/march_99/denning/03_99denning.htm`.
17. Educational Activities Board. The 1983 model program in computer science and engineering. Technical Report 932, Computer Society of the IEEE, December 1983.
18. Educational Activities Board. Design education in computer science and engineering. Technical Report 971, Computer Society of the IEEE, October 1986.

19. Education Committee of the IEEE Computer Society. A curriculum in computer science and engineering. Publication EHO119-8, Computer Society of the IEEE, January 1977.
20. Norman E. Gibbs and Allen B. Tucker. Model curriculum for a liberal arts degree in computer science. Communications of the ACM, 29(3):202-210, March 1986.
21. John T. Gorgone, Paul Gray, David L. Feinstein, George M. Kasper, Jerry N. Luftman, Edward A. Stohr, Joseph S. Valacich, and Rolf T. Wigand. MSIS 2000: Model curriculum and guidelines for graduate degree programs in information systems. Association for Computing Machinery and Association for Information Systems, January 2000. **`http://cis.bentley.edu/ISA/pages/documents/msis2000jan00.pdf`**.
22. Charles F. Kelemen (editor), Owen Astrachan, Doug Baldwin, Kim Bruce, Peter Henderson, Dale Skrien, Allen Tucker, and Charles Ban Loan. Computer Science Report to the CUPM Curriculum Foundations Workshop in Physics and Computer Science. Report from a workshop at Bowdoin College, October 28-31, 1999.
23. Elliot P. Koffman, Philip L. Miller, and Caroline E. Wardle. Recommended curriculum for CS1: 1984 a report of the ACM curriculum task force for CS1. Communications of the ACM, 27(10):998-1001, October 1984.
24. Elliot P. Koffman, David Stemple, and Caroline E. Wardle. Recommended curriculum for CS2, 1984: A report of the ACM curriculum task force for CS2. Communications of the ACM, 28(8):815-818, August 1985.
25. Edward A. Lee and David G. Messerschmitt. Engineering and education for the future. IEEE Computer, 77-85, January 1998.
26. Doris K. Lidtke, Gordon E. Stokes, Jimmie Haines, and Michael C. Mulder. ISCC ’99: An information systems-centric curriculum ’99, July 1999. **`http://www.iscc.unomaha.edu`**.
27. C. Dianne Martin, Chuck Huff, Donald Gotterbarn, Keith Miller. Implementing a tenth strand in the CS curriculum. Communications of the ACM, 39(12):75-84, December 1996.
28. Michael C. Mulder. Model curricula for four-year computer science and engineering programs: Bridging the tar pit. Computer, 8(12):28-33, December 1975.
29. Michael C. Mulder and John Dalphin. Computer science program requirements and accreditation—an interim report of the ACM/IEEE Computer Society joint task force. Communications of the ACM, 27(4):330-335, April 1984.
30. Peter G. Neumann. Computer related risks. New York: ACM Press, 1995.
31. Jay F. Nunamaker, Jr., J. Daniel Couger, Gordon B. Davis. Information systems curriculum recommendations for the 80s: Undergraduate and graduate programs. Communications of the ACM, 25(11):781-805, November 1982.
32. National Science Foundation Advisory Committee. Shaping the future: New expectations for undergraduate education in science, mathematics, engineering, and technology. Washington DC: National Science Foundation, 1996.
33. National Telecommunications and Information Administration. Falling through the Net: Defining the digital divide. Washington, DC: Department of Commerce, November 1999.
34. President’s Science Advisory Commission. Computers in higher education. Washington DC: The White House, February 1967.

35. Mary Shaw. The Carnegie-Mellon curriculum for undergraduate computer science. New York: Springer-Verlag, 1985.
36. Mary Shaw and James E Tomayko. Models for undergraduate courses in software engineering. Pittsburgh: Software Engineering Institute, Carnegie Mellon University, January 1991.
37. Mary Shaw. We can teach software better. Computing Research News 4(4):2-12, September 1992.
38. Software Engineering Coordinating Committee. Guide to the Software Engineering Body of Knowledge (SWEBOK). Stone Man Version 0.5. Joint project of the IEEE Computer Society and the Association for Computing Machinery. October 1999. **`http://www.swebok.org/stoneman/version05/`**.
39. Allen B. Tucker, Bruce H. Barnes, Robert M. Aiken, Keith Barker, Kim B. Bruce, J. Thomas Cain, Susan E. Conry, Gerald L. Engel, Richard G. Epstein, Doris K. Lidtke, Michael C. Mulder, Jean B. Rogers, Eugene H. Spafford, and A. Joe Turner. Computing Curricula '91. Association for Computing Machinery and the Computer Society of the Institute of Electrical and Electronics Engineers, 1991.
40. U.S. Congress, Office of Technology Assessment. Educating scientists and engineers: Grade school to grad school. OTA-SET-377. Washington, DC: U.S. Government Printing Office, June 1988.
41. Henry M. Walker and G. Michael Schneider. A revised model curriculum for a liberal arts degree in computer science. Communications of the ACM, 39(12):85-95, December 1996.
42. Lofti A. Zadeh. Computer science as a discipline. Journal of Engineering Education, 58(8):913-916, April 1968.

Appendix A
CS Body of Knowledge

The topics shown in Table A-1 represent the body of knowledge for programs in computer science, as developed by the knowledge area focus groups. For each area, topics that are considered essential for all undergraduate programs in computer science are underlined. Each of these core topics is also associated with an estimate of the minimum amount of time that must be devoted to that material.

In future versions of this report, this body of knowledge will be supplemented with others that cover other disciplines within the computing field.

Figure A-1. Computer science body of knowledge with core topics underlined

DS. Discrete Structures (37 core hours)
- DS1. Functions, relations, and sets (6)
- DS2. Basic logic (10)
- DS3. Proof techniques (12)
- DS4. Basics of counting (5)
- DS5. Graphs and trees (4)

PF. Programming Fundamentals (65 core hours)
- PF1. Algorithms and problem-solving (8)
- PF2. Fundamental programming constructs (10)
- PF3. Basic data structures (12)
- PF4. Recursion (6)
- PF5. Abstract data types (9)
- PF6. Object-oriented programming (10)
- PF7. Event-driven and concurrent programming (4)
- PF8. Using modern APIs (6)

AL. Algorithms and Complexity (31 core hours)
- AL1. Basic algorithmic analysis (4)
- AL2. Algorithmic strategies (6)
- AL3. Fundamental computing algorithms (12)
- AL4. Distributed algorithms (3)
- AL5. Basic computability theory (6)
- AL6. The complexity classes P and NP
- AL7. Automata theory
- AL8. Advanced algorithmic analysis
- AL9. Cryptographic algorithms
- AL10. Geometric algorithms

PL. Programming Languages (5 core hours)
- PL1. History and overview of programming languages (2)
- PL2. Virtual machines (1)
- PL3. Introduction to language translation (2)
- PL4. Language translation systems
- PL5. Type systems
- PL6. Models of execution control
- PL7. Declaration, modularity, and storage management
- PL8. Programming language semantics
- PL9. Functional programming paradigms
- PL10. Object-oriented programming paradigms
- PL11. Language-based constructs for parallelism

AR. Architecture (33 core hours)
- AR1. Digital logic and digital systems (3)
- AR2. Machine level representation of data (3)
- AR3. Assembly level machine organization (9)
- AR4. Memory system organization (5)
- AR5. I/O and communication (3)
- AR6. CPU implementation (10)

OS. Operating Systems (22 core hours)
- OS1. Operating system principles (2)
- OS2. Concurrency (6)
- OS3. Scheduling and dispatch (3)
- OS4. Virtual memory (3)
- OS5. Device management (2)
- OS6. Security and protection (3)
- OS7. File systems and naming (3)
- OS8. Real-time systems

HC. Human-Computer Interaction (3 core hours)
- HC1. Principles of HCI (3)
- HC2. Modeling the user
- HC3. Interaction
- HC4. Window management system design
- HC5. Help systems
- HC6. Evaluation techniques
- HC7. Computer-supported collaborative work

GR. Graphics (no core hours)
- GR1. Graphic systems
- GR2. Fundamental techniques in graphics
- GR3. Basic rendering
- GR4. Basic geometric modeling
- GR5. Visualization
- GR6. Virtual reality
- GR7. Computer animation
- GR8. Advanced rendering
- GR9. Advanced geometric modeling
- GR10. Multimedia data technologies
- GR11. Compression and decompression
- GR12. Multimedia applications and content authoring
- GR13. Multimedia servers and filesystems
- GR14. Networked and distributed multimedia systems

IS. Intelligent Systems (10 core hours)
- IS1. Fundamental issues in intelligent systems (2)
- IS2. Search and optimization methods (4)
- IS3. Knowledge representation and reasoning (4)
- IS4. Learning
- IS5. Agents
- IS6. Computer vision
- IS7. Natural language processing
- IS8. Pattern recognition
- IS9. Advanced machine learning
- IS10. Robotics
- IS11. Knowledge-based systems
- IS12. Neural networks
- IS13. Genetic algorithms

IM. Information Management (10 core hours)
- IM1. Database systems (2)
- IM2. Data modeling and the relational model (8)
- IM3. Database query languages
- IM4. Relational database design
- IM5. Transaction processing
- IM6. Distributed databases
- IM7. Advanced relational database design
- IM8. Physical database design

NC. Net-Centric Computing (15 core hours)
- NC1. Introduction to net-centric computing (9)
- NC2. The web as an example of client-server computing (6)
- NC3. Building web applications
- NC4. Communication and networking
- NC5. Distributed object systems
- NC6. Collaboration technology and groupware
- NC7. Distributed operating systems
- NC8. Distributed systems

SE. Software Engineering (30 core hours)
- SE1. Software processes and metrics (6)
- SE2. Software requirements and specifications (6)
- SE3. Software design and implementation (6)
- SE4. Verification and validation (6)
- SE5. Software tools and environments (3)
- SE6. Software project methodologies (3)

CN. Computational Science (no core hours)
- CN1. Numerical analysis
- CN2. Scientific visualization
- CN3. Architecture for scientific computing
- CN4. Programming for parallel architectures
- CN5. Applications

SP. Social and Professional Issues (16 core hours)
- SP1. History of computing (1)
- SP2. Social context of computing (2)
- SP3. Methods and tools of analysis (2)
- SP4. Professional and ethical responsibilities (2)
- SP5. Risks and liabilities of safety-critical systems (2)
- SP6. Intellectual property (3)
- SP7. Privacy and civil liberties (2)
- SP8. Social implications of the Internet (2)
- SP9. Computer crime
- SP10. Economic issues in computing
- SP11. Philosophical foundations of ethics

DS. Discrete Structures (37 core hours)

DS1. Functions, relations, and sets (core—6 hours)

Functions (surjections, injections, inverses, composition)
Relations (reflexivity, symmetry, transitivity, equivalence relations)
Sets (Venn diagrams, complements, Cartesian products, power sets)
Pigeonhole principle
Cardinality and countability

DS2. Basic logic (core—10 hours)

Propositional logic
Logical connectives
Truth tables
Validity
Implication, converse, inverse, negation, contradiction
Predicate logic
Limitations of predicate logic
Universal and existential quantification
Modus ponens and modus tallens

DS3. Proof techniques (core—12 hours)

The structure of formal proofs
Direct proofs
Proof by counterexample
Proof by contraposition
Proof by contradiction
Mathematical induction
Strong induction
Recursive mathematical definitions
Well orderings

DS4. Basics of counting (core—5 hours)

Counting arguments
Permutations and combinations
Solving recurrence relations

DS5. Graphs and trees (core—4 hours)

Trees
Undirected graphs
Directed graphs
Spanning trees

PF. Programming Fundamentals (65 core hours)

PF1. Algorithms and problem-solving (core—8 hours)

Problem-solving strategies
Debugging strategies
Structured decomposition
The concept and properties of algorithms
The role of algorithms in the problem-solving process
Introduction to algorithmic complexity
Empirical measurements of performance

PF2. Fundamental programming constructs (core—10 hours)

Basic syntax and semantics of a higher-level language
Variables, types, and assignment
Conditional and iterative control structures
Functions and parameter passing
Simple I/O
Exception handling

PF3. Basic data structures (core—12 hours)

Primitive types
Arrays
Records
Strings and string processing
Data representation in memory
Pointers (or the notion of a reference in an object-oriented language)
Linked structures
Static, stack, and heap allocation
Runtime storage management
Strategies for choosing the right data structure

PF4. Recursion (core—6 hours)

The concept of recursion
Recursive mathematical functions
Simple recursive procedures (Towers of Hanoi, generating permutations)
Divide-and-conquer strategies
Recursive backtracking
Implementation of recursion

PF5. Abstract data types (core—9 hours)

The importance of data abstraction
Abstract programming interfaces
Abstract data types
Encapsulation and levels of visibility
Information hiding
Iteration protocols
Specific ADT structures (stacks, queues, symbol tables, trees, graphs)

PF6. Object-oriented programming (core—10 hours)

Object-oriented design
Classes, subclasses, and inheritance

Class hierarchies
Polymorphism
Fundamental design patterns

PF7. Event-driven and concurrent programming (core—4 hours)

Event-handling methods
Event propagation
Managing concurrency in event handling

PF8. Using modern APIs (core—6 hours)

API programming
Class browsers and related tools
Programming by example
Debugging in the API environment

AL. Algorithms and Complexity (31 core hours)

AL1. Basic algorithmic analysis (core—4 hours)

Asymptotic analysis of upper and average complexity bounds
Identifying differences among best, average, and worst case behaviors
Big “O” and little “o” notation
Standard complexity classes
Time and space tradeoffs in algorithms
Using recurrence relations to analyze recursive algorithms

AL2. Algorithmic strategies (core—6 hours)

Brute-force algorithms
Greedy algorithms
Divide-and-conquer
Backtracking
Branch-and-bound
Heuristics
Pattern matching and string/text algorithms
Numerical approximation algorithms

AL3. Fundamental computing algorithms (core—12 hours)

Simple numerical algorithms
Sequential and binary search algorithms
Quadratic sorting algorithms (selection, insertion)
O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)
Hash tables, including collision-avoidance strategies
Binary search trees
Representations of graphs (adjacency list, adjacency matrix)
Depth- and breadth-first traversals
Shortest-path algorithms (Dijkstra’s and Floyd’s algorithms)
Transitive closure (Floyd’s algorithm)
Minimum spanning tree (Prim’s and Kruskal’s algorithms)
Topological sort

AL4. Distributed algorithms (core—3 hours)

Consensus and election
Termination detection
Fault tolerance
Stabilization

AL5. Basic computability theory (core—6 hours)

Finite-state machines
Context-free grammars
Uncomputable functions
The halting problem
Implications of uncomputability

AL6. The complexity classes P and NP

Tractable and intractable problems
Definition of the classes P and NP
NP-completeness (Cook’s theorem)

Standard NP-complete problems
Reduction techniques

AL7. Automata theory

Deterministic finite automata (DFAs)
Nondeterministic finite automata (NFAs)
Equivalence of DFAs and NFAs
Regular expressions
The pumping lemma for regular expressions
Push-down automata (PDAs)
Relationship of PDAs and context-free grammars
Properties of context-free grammars
Turing machines
Nondeterministic Turing machines
Sets and languages
Chomsky hierarchy

AL8. Advanced algorithmic analysis

Amortized analysis
Online and offline algorithms
Randomized algorithms
Dynamic programming
Combinatorial optimization

AL9. Cryptographic algorithms

Historical overview of cryptography
Private-key cryptography and the key-exchange problem
Public-key cryptography
Digital signatures
Security protocols
Applications (zero-knowledge proofs, authentication, and so on)

AL10. Geometric algorithms

Line segments: properties, intersections
Convex hull finding algorithms

PL. Programming Languages (5 core hours)

PL1. History and overview of programming languages (core—2 hours)

Early languages (FORTRAN, ALGOL, COBOL, LISP, BASIC)
The evolution of procedural languages
Object-oriented paradigm and languages
Mostly-functional, algorithmic, higher-order languages with eager evaluation
Purely-functional, algorithmic paradigm
Declarative (non-algorithmic) languages
Parallel programming paradigms
Scripting paradigm

PL2. Virtual machines (core—1 hour)

What is a virtual machine?
Hierarchy of virtual machines presented to the user

PL3. Introduction to language translation (core—2 hours)

Comparison of pure interpreters vs. compilers
Language translation phases (lexical analysis, parsing, code generation, optimization)

PL4. Language translation systems

Application of regular expressions in lexical scanners
Parsing (concrete and abstract syntax, abstract syntax trees)
Code generation by tree walking
Optimization techniques
Application of CFGs in table-driven and recursive descent parsing

PL5. Type systems

Data type as set of values with set of operations
Data types (elementary, product, coproduct, algebraic, recursive, arrow, parameterized)
Type checking models
Semantic models of user-defined types (type abbreviations, ADTs, type equality)
Parametric polymorphism
Subtype polymorphism
Type-checking algorithms

PL6. Models of execution control

Order of evaluation of sub-expressions
Exceptions and exception handling
Parallel composition (S1||S2)
Functions with delayed evaluation (closures, lazy evaluation)

PL7. Declaration, modularity, and storage management

Declaration models (binding, visibility, scope, and lifetime)
Parameterization mechanisms
Type parameterization
Mechanisms for sharing and restricting visibility of declarations
Garbage collection

PL8. Programming language semantics

Informal semantics
Overview of formal semantics
Denotational Semantics.
Axiomatic Semantics

PL9. Functional programming paradigms

Overview and motivation
Recursion over lists, natural numbers, trees, and other recursively-defined data
Pragmatics (debugging by divide and conquer; persistency of data structures)
Amortized efficiency for functional data structures
Closures, and uses of functions as data (infinite sets, streams)

PL10. Object-oriented programming paradigms

Mechanisms for defining classes and instances
Object creation and initialization
Inheritance and dynamic dispatch
Sketch of run-time representation of objects and method tables
Distinction between subtyping and inheritance
Advanced OO type problems

PL11. Language-based constructs for parallelism

Communication primitives for tasking models with explicit communication
Communication primitives for tasking models with shared memory
Programming primitives for data-parallel models
Comparison of language features for parallel and distributed programming
Optimistic concurrency control vs. locking and transactions
Coordination languages (Linda)
Asynchronous remote procedure calls (pipes)
Other approaches (functional, nondeterministic)

AR. Architecture (33 core hours)

AR1. Digital logic and digital systems (core—3 hours)

Simple building blocks (logic gates, flip-flops, counters, registers)
Logic expressions
Simple adders, structure of a simple arithmetic-logic unit (ALU)

AR2. Machine level representation of data (core—3 hours)

Numeric data representation and number bases
Fixed- and floating-point systems
Signed and twos-complement representations
Representation of nonnumeric data

AR3. Assembly level machine organization (core—9 hours)

Basic organization
Control unit; instruction fetch, decode, and execution
Instruction sets and types (data manipulation, control, I/O)
Assembly/machine language programming
Instruction formats
Addressing modes
I/O and interrupts

AR4. Memory system organization (core—5 hours)

Storage systems and technology
Memory hierarchy
Main memory organization and operations
Latency, cycle time, bandwidth, and interleaving
Cache memories (address mapping, replacement and store policy)

AR5. I/O and communication (core—3 hours)

Input/output control methods, interrupts
Synchronization, open loop, handshaking
External storage, physical organization, and drives
Bus systems, control, direct-memory access (DMA)

AR6. CPU implementation (core—10 hours)

Hardwired realization of CPU
Microprogrammed realization; formats and coding
Varieties of instruction formats
Architectural support for operating systems and compilers
Instruction pipelining
Introduction to instruction-level parallelism (ILP)

OS. Operating Systems (22 core hours)

OS1. Operating system principles (core—2 hours)

Structuring methods and the layered model
Applications needs and the evolution of hardware/software techniques
Device organization
Interrupts: methods and implementations
Concept of user/system state and protection

OS2. Concurrency (core—6 hours)

States and state diagrams
Structures (ready list, process control blocks, and so forth)
Dispatching and context switching
The role of interrupts
Concurrent execution
The “mutual exclusion” problem
Deadlock: causes, conditions, prevention
Models and mechanisms (semaphores, monitors, rendezvous)
Producer-consumer problems

OS3. Scheduling and dispatch (core—3 hours)

Preemptive and nonpreemptive scheduling
Schedulers and policies
Processes and threads
Deadlines and real-time issues

OS4. Virtual memory (core—3 hours)

Review of physical memory and memory management hardware
Overlays, swapping, and partitions
Paging and segmentation
Memory mapped files
Placement and replacement policies
Working sets and thrashing

OS5. Device management (core—2 hours)

Characteristics of a serial or parallel device
Buffering strategies
Free lists and device layout
Servers and interrupts
Recovery from failures

OS6. Security and protection (core—3 hours)

Overview of system security
Security methods and devices
Protection, access, and authentication
Models of protection
Memory protection
Encryption
Recovery management

OS7. File systems and naming (core—3 hours)

File layout
Directories: contents and structure
Naming, searching, access, backups
Fundamental file concepts (organization, blocking, buffering)
Sequential files
Nonsequential files

OS8. Real-time systems

Process and task scheduling
Memory and disk management
Failures, risks, and recovery
Special concerns in real-time systems

HC. Human-Computer Interaction (3 core hours)

HC1. Principles of HCI (core—3 hours)

Conceptual Models
Mapping
Affordances
Constraints
Seven Stages of Action
Schneiderman’s 8 Golden Rules
Information Visualization

HC2. Modeling the user

Model Human Processor
Keystroke Level Model
Fitt’s law

HC3. Interaction

Input devices (Keyboard, Pointing, Voice)
Output devices (Displays, Color, Sound)
Interaction Styles (direct manipulation, menu selection, form-fill-in, command languages)

HC4. Window management system design

Windows
Icons
Menus
Dialogue Boxes
Concepts (grids, simplicity, consistency, white space)

HC5. Help systems

Context Sensitive Help
Tutorials
Reference Material

HC6. Evaluation techniques

Cognitive Walkthrough
Heuristic Evaluation
Expert Reviews
Controlled Experiments (subjects, dependant & independent variables, statistics)

HC7. Computer-supported collaborative work

Synchronous / Asynchronous tools
Audio / Video
Shared Workspaces

GR. Graphics, Visualization, and Multimedia (nothing in core)

GR1. Graphic systems

Raster and vector graphics system
Video display devices
Physical and logical input devices
Issues facing the developer of graphical systems
Hierarchy of graphics software
User interface

GR2. Fundamental techniques in graphics

Halftoning
Font generation: outline vs. bitmap
Representation of polyhedral objects
Scan conversion of 2D primitive, forward differencing
Tessellation of curved surfaces
Homogeneous coordinates
Affine transformations (scaling, rotation, translation)
Viewing transformation
Clipping
Hidden surface removal methods
Z-buffer and frame buffer, color channels (a channel for opacity)

GR3. Basic rendering

Color models (RGB, HVS, CYM)
Light source properties; material properties; ambient, diffuse, and specular reflections
Phong reflection model
Rendering of a polygonal surface, flat shading, Gouraud shading, and Phong shading
Texture mapping, bump texture, environment map
Ray tracing
Image synthesis, sampling techniques, and anti-aliasing

GR4. Basic geometric modeling

Parametric polynomial curves and surfaces
Implicit curves and surfaces
Bézier curves and surfaces, control points, de Casteljau algorithm
B-spline curves and surfaces, local editing, knots, control points
NURBS curves and surfaces
Constructive Solid Geometry (CSG) for solid modeling
Boundary Representation of solids (B-Rep)

GR5. Visualization

Basic viewing and interrogation functions for visualization
Visualization of vector fields, tensors, and flow data
Visualization of scalar field or height field: iso-surface by the marching cube method
Direct volume data rendering: ray-casting, transfer functions, segmentation, hardware

GR6. Virtual reality

Stereoscopic display
Force feedback simulation, haptic devices
Viewer tracking

Collision detection
Visibility computation
Time-critical rendering, multiple levels of details (LOD)
Image-base VR system
Distributed VR, collaboration over computer network
Interactive modeling
User interface issues
Applications in medicine, simulation, and training

GR7. Computer animation

Color animation
Physical based animation
Animation of articulated structures: forward and inverse kinematics
Scripting system
Key-frame animation, inbetweening, quaternions for orientation representation
Motion capture
Behavioral and procedural animation, particle system
Metamorphosis
Free-form deformation

GR8. Advanced rendering

Shadow computation
Radiosity for global illumination computation, form factors
A two-pass approach to global illumination
Monte Carlo methods for global illumination
Image-based rendering, panorama viewing, plenoptic function modeling and sampling
Rendering of complex natural phenomenon
Non-photorealistic rendering

GR9. Advanced geometric modeling

Implicit surfaces, soft object
Algebraic curves and surfaces
Subdivision surfaces
Multi-resolution analysis of polygonal meshes, wavelets
Deformable models: snakes and balloons (active contours)
Procedural modeling, fractals
3D model acquisition from range data
3D data fitting
Geometric operations, intersection, blending, faring, offsetting, sweeping, etc.

GR10. Multimedia data technologies

Analog and digital representations, human perception
Sound and audio, image and graphics, animation and video
Standard file formats for audio, graphics and image data
Standards for videoconferencing, computer telephony and motion pictures
TV broadcasting standards
Display and input devices
Digital cameras and scanners
Buses, I/O channels
Tape, disk and RAID
CD and DVD ROM standards

GR11. Compression and decompression

Information theory
Lossless compression techniques
Digital audio compression
DCT, wavelet and fractal compressions
Scalable and progressive encoding
H.26x and Mpeg video compression standards

GR12. Multimedia applications and content authoring

Multimedia applications and requirements
Design issues for content authoring
Human computer interface basics
Authoring tools and production systems
Web authoring and programming
Interactive multimedia titles

GR13. Multimedia servers and filesystems

Multimedia server requirements
RAID storage systems
Storage hierarchy
Data placement on disks
Buffer management
QOS support and admission control
Processor scheduling
Disk scheduling
Multimedia information management systems
Provision of user interactivity
Content-based information retrieval

GR14. Networked and distributed multimedia systems

Characteristics of multimedia communications
Layers, protocols and services
Local area networks (LAN) and wide area networks (WAN)
ATM and ISDN for multimedia communications
MBONE multicast and applications
Admission control, QOS negotiation and traffic policing
Distributed multimedia systems, client-serve concepts
Server configuration and network connection
Streaming servers and network scheduling
Networked multimedia synchronization

IS. Intelligent Systems (10 core hours)

IS1. Fundamental issues in intelligent systems (core—2 hours)

Definitions of intelligent systems
Optimality vs. speed tradeoff

IS2. Search and optimization methods (core—4 hours)

Problem spaces
Brute-force search (DFS, BFS, uniform cost search)
Heuristic search (best-first, A*, IDA*)
Local search (hill-climbing, simulated annealing, genetic search)
Game-playing methods (minimax search, alpha-beta pruning)
Constraint satisfaction (backtracking and heuristic repair)

IS3. Knowledge representation and reasoning (core—4 hours)

Representation of space and time
Representations of events and actions
Probabilistic reasoning
Bayes theorem
Predicate calculus and resolution
Logic programming and theorem proving
AI planning systems

IS4. Learning

Unsupervised vs. supervised learning
Inductive vs. deductive
Classification vs. clustering vs. prediction
Decision tree learning and neural network learning as examples

IS5. Agents

Action selection and planning
Collaboration between people and agents
Communication between people and agents
Expert assistants
Agent architectures
Interacting with stochastic environments
Reinforcement learning
Multi-agent systems
Game theory and auctions

IS6. Computer vision

Image acquisition, processing, and display
Edge detection
Camera models
Calibration of camera models from images
Color constancy
Texture
Segmentation
Object recognition
Motion
Tracking

IS7. Natural language processing

Deterministic and stochastic grammars
Parsing algorithms
Corpus-based methods
Information retrieval
Language translation

IS8. Pattern recognition

Statistical pattern recognition
Syntactic pattern recognition
Bayesian decision theory
Linear discriminant functions
Feature extraction for representation
Feature extraction for classification
Supervised learning
Unsupervised learning and clustering

IS9. Advanced machine learning

Learning belief networks
Decision-tree learning
Reinforcement learning algorithms
Neural net learning
Genetic algorithms and evolutionary programming
Inductive logic programming
PAC learning and beyond

IS10. Robotics

Navigation and control
Optimization and learning
Perception
Path planning
Direct and inverse kinematics
Robot programming
Robot simulation environments

IS11. Knowledge-based systems

Design and development of knowledge-based systems
Knowledge representation mechanisms
Reasoning with uncertainty (nonmonotonic logics, certainty factors, fuzzy logic)
Knowledge acquisition techniques
Knowledge engineering
Tools for knowledge-based system development

IS12. Neural networks

Single-layer networks
Supervised learning
Multi-layer perceptrons and back-propagation
Competitive learning networks
Examples of multi-layer networks
Other network architectures and their applications

IS13. Genetic algorithms

Brief history of evolutionary computation
Theoretical foundations of genetic algorithms
Implementing a genetic algorithm
Applications of genetic algorithms
Genetic programming

IM. Information Management (10 core hours)

IM1. Database systems (core—2 hours)

History and motivation for database systems
Components of database systems
DBMS functions
Database architecture and data independence
Recent developments and applications (hypertext, hypermedia, multimedia)

IM2. Data modeling and the relational model (core—8 hours)

Data modeling
Entity-Relationship model
Object-Oriented model
Relational data model
Mapping conceptual schema to a relational schema
Entity and referential integrity
Relational algebra and relational calculus

IM3. Database query languages

Overview of database languages
SQL (data definition, query formulation, update sublanguage, constraints, integrity)
QBE and 4th generation environments
Introduction to Object Query Language
Embedding non-procedural queries in a procedural language

IM4. Relational database design

Database design
Functional dependency
Normal forms (1NF, 2NF, 3NF, BCNF)

IM5. Transaction processing

Transactions
Failure and Recovery
Concurrency Control

IM6. Distributed databases

Distributed data storage
Distributed query processing
Distributed transaction model
Concurrency control

IM7. Advanced relational database design

Multivalued dependency (4NF)
Join dependency (PJNF, 5NF)
Representation theory

IM8. Physical database design

Storage and file structure
Indexed files
Hashed files

B-trees
Files with dense index
Files with variable length records

NC. Net-Centric Computing (15 core hours)

NC1. Introduction to net-centric computing (core—9 hours)

Background and history of the Internet
The architecture of the Internet
The five-layer reference model (physical, data link, network, transport, application)
Host name resolution and the Domain Name Service
Public-key cryptography and digital certificates
Distributed computing
Networked multimedia systems

NC2. The web as an example of client-server computing (core—6 hours)

Introduction to client-server programming
Designing clients and servers
Introduction to the technologies of the web (URLs, HTML, HTTP, applets, etc.)

NC3. Building web applications

JavaScript and other client-side programming within a web browser
CGI and other server-side programming with web-based application servers

NC4. Communication and networking

Protocol suites
Streams and datagrams
Client-server communication and group communication
Remote procedure calls
Internetworking and routing

NC5. Distributed object systems

Serializing objects
Persistent objects
Remote procedure calls
Distributed object frameworks
Java's distributed object framework (JavaBeans, Java RMI, and JINI)
COM and DCOM as distributed object framework
CORBA and IDL
Lightweight distributed objects with XML
Security issues in distributed object systems

NC6. Collaboration technology and groupware

Audio/video interpersonal applications
Shared workspace for computer-supported collaborative work
Audio/video distribution
Audio/videoconferencing
Multimedia document transfer
Multimedia server-based applications
Virtual reality
Emerging topics in CSCW

NC7. Distributed operating systems

Distributed processes and threads
Distributed file systems

Name services
Time, synchronization and coordination
Replication
Concurrency control
Shared and distributed transactions
Distributed shared memory

NC8. Distributed systems

Characterization of distributed systems
Partition and allocation of distributed tasks
Load balancing of distributed systems
Modeling and analysis of distributed systems
Distributed languages
Fault tolerance and recovery
Security issues of distributed systems

SE. Software Engineering (30 core hours)

SE1. Software processes and metrics (core—6 hours)

Software life-cycle and process models
Software metrics (product and process metrics)
Introduction to software standards and documentation
Software quality assurance
Configuration management and control
Project planning and risk management
Software estimation
Software maintenance and re-engineering

SE2. Software requirements and specifications (core—6 hours)

Requirements elicitation, review and acceptance
Requirements modeling, analysis and specification
Rapid prototyping
Formal and executable specifications

SE3. Software design and implementation (core—6 hours)

Using the requirements specification document
Introduction to software architecture
Object-oriented decomposition and design
Functional decomposition and structured design
Design experimentation and prototyping
Detailed design and implementation
Design reviews

SE4. Verification and validation (core—6 hours)

Aspects of quality and their importance (reliability, correctness, etc.)
Quality assurance techniques and the software development process
Testing methods (acceptance, integration, unit; black box vs. white box testing)
Coverage measures (line, condition, branch)
Testing specialized classes of applications (real-time, embedded, database, web, GUI)
Concepts of formal verification (assertions, invariants, pre- and post-conditions)
Software testing management (test plan development, test case design, tracking, release engineering)

SE5. Software tools and environments (core—3 hours)

History of development tools and environments; the significance of Unix
Programming environments (integrated editors, compilers, interpreters, debuggers)
Modeling tools (structured, object-oriented, database, and GUI modeling)
Testing tools (test coverage, defect tracking, test management)
Product management (version control, configuration management)
Tool integration mechanisms (control integration, COM and CORBA, repositories)

SE6. Software project methodologies (core—3 hours)

Team management
Project planning
Requirements engineering
Design specification and review
Product engineering

Software quality assurance
Software configuration management
Documentation

CN. Computational Science (nothing in core)

CN1. Numerical analysis

Floating-point arithmetic
Error, stability, convergence
Taylor’s series
Iterative solutions for finding roots (Newton’s Method)
Curve fitting; function approximation
Numerical differentiation and integration (Simpson’s Rule)
Explicit and implicit methods
Differential equations (Euler’s Method)
Linear algebra
Finite differences

CN2. Scientific visualization

Concepts
Tools
Examples

CN3. Architecture for scientific computing

Vector architecture and pipelining
MIMD machines
Distributed systems and the network-of-workstations (NOW) approach
Networks
Timing, measurement, terminology (MFLOPS, and so forth)
Benchmarks and elementary performance measurement

CN4. Programming for parallel architectures

Review of parallel programming techniques
Parallel algorithms for scientific computation
Effects of array element and loop ordering
Example languages

CN5. Applications

Simulation
Molecular dynamics
Fluid dynamics
Celestial mechanics
Optimization (linear programming, integer programming, dynamic programming)
Structural analysis
Geology
Computerized tomography
Military and defense applications

SP. Social, Ethical, and Professional Issues (16 core hours)

SP1. History of computing (core—1 hour)

History of computer hardware
History of computer software
History of networking
Pioneers of computing

SP2. Social context of computing (core—2 hours)

Introduction to the social implications of computing
Social implications of networked communication
Overview of intellectual property issues in computing

SP3. Methods and tools of analysis (core—2 hours)

Making and evaluating ethical arguments
Identifying and evaluating ethical choices
Understanding the social context of design
Identifying assumptions and values

SP4. Professional and ethical responsibilities (core—2 hours)

The nature of professionalism
The role of the professional in public policy
Maintaining awareness of consequences
Ethical dissent
Codes of ethics

SP5. Risks and liabilities of safety-critical systems (core—2 hours)

Historical examples of software risks (such as the Therac-25 case)
Implications of software complexity
Risk assessment and management

SP6. Intellectual property (core—3 hours)

Foundations of intellectual property
Copyrights, patents, and trade secrets
Software piracy
Software patents and the look-and-feel debate
International issues concerning intellectual property

SP7. Privacy and civil liberties (core—2 hours)

Historical basis for privacy protection
Privacy implications of massive database systems
The "Code of Fair Information Practices"
Technological strategies for privacy protection
Freedom of expression in cyberspace
Restrictions on expression
International and intercultural implications

SP8. Social implications of the Internet (core—2 hours)

History and growth of the Internet
Control of the Internet

The nature of online communication
Online communities
Access to the Internet
International implications

SP9. Computer crime

History and examples of computer crime
Hacking and its effects
Viruses, worms, and Trojan horses
Crime prevention strategies

SP10. Economic issues in computing

Monopolies and their economic implications
Labor shortages in computing
Pricing strategies in the computing domain
Inequalities of access based on economic class

SP11. Philosophical foundations of ethics

Philosophical frameworks (consequentialist and deontological theories)
Problems of ethical relativism
Scientific ethics in historical perspective
Differences in scientific and philosophical approaches