A Summary of the ACM/IEEE-CS Joint Curriculum Task Force Report

COMPUTING CURRICULA 1991

INTRODUCTION: ACM first published recommendations for undergraduate programs in computer science in 1968 in a report called "Curriculum '68." The report was produced as an activity of the ACM Education Board, which since then has been providing updates to recommendations for computer science programs as well as recommendations for other academic programs in computing.

In 1984 it was recognized by Education Board Chair Robert Aiken that there was a need for a fresh look at undergraduate computer science curricula. The discipline had matured considerably during the previous decade, but there had been enough changes in both subject matter and pedagogy to create a need to update the recommendations that had been published in 1979. One problem was that while many programs had sprung up in response to local demand, the implementors did not have a good feel for the discipline and the programs were too narrow, often lacking depth as well as breadth. It was decided that a good starting point for reconsideration of degree programs in computer science would be to establish a working definition of the discipline.

At about the same time, informal discussions took place between members of the Education Board and members of the IEEE Computer Society regarding the possibility of joint work in the curriculum area. Both societies had published curriculum recommendations independently, but it was recognized that there was a great deal of overlap in interests and considerable commonality between the recommendations of the two groups. Furthermore, joint recommendations from the two societies were likely to have a much greater impact than recommendations from only one of the two.

Consequently, a task force of distinguished computer scientists was formed in 1985 by ACM, in cooperation with the Computer Society, to establish a definition of the discipline and to make recommendations for an introductory course sequence that would provide a foundation for curricula in the discipline. The task force, chaired by Peter Denning, published its report in December 1988. It was decided to complete the task of developing recommendations for the entire undergraduate curriculum by forming another task force jointly with the Computer Society; this was done in February, 1988.

The Joint ACM/IEEE-CS Curriculum Task Force published its report in March, 1991. A summary of the report is provided in the following article, and the entire report can be ordered through the ACM Order Department (Order Number 201910). The report represents the efforts of the task force to present current

thinking on goals and objectives for computing curricula. Recommendations and comments were solicited from the computer science and engineering community through several sessions and extensive discussions at national conferences, through specific requests to experts in various areas, and through the establishment of a group of 120 reviewers.

This report is significant in that it not only represents an update of curriculum recommendations to meet changing needs, but it represents a unified set of recommendations from the two major societies of the discipline for computing curricula in a variety of academic contexts, including programs in liberal arts, sciences, and engineering. The report provides new perspectives on the importance and role of breadth in courses and the curriculum, the importance and role of laboratories in the curriculum, and the importance and role of social, ethical, and professional issues in the curriculum. It also reinforces the importance of theoretical foundations (including mathematics), the development and application of communication skills, and the inclusion of significant design experiences, including working in teams, in a program.

The report is in some sense more difficult to use for assistance in curriculum development than were previous reports. It does not provide a single set of sample courses, or a sample curriculum. The report recognizes that there are many effective ways to organize a curriculum, even for a particular set of goals and objectives: it emphasizes the specification of a minimal set of subject matter that should be included in all programs along with guidelines for organizing the subject matter into courses and incorporating additional material and pedagogy to complete a curriculum. Several examples of curricula for different design objectives are provided. While the task force was sympathetic to the desirability of developing a concise set of curriculum guidelines, it was decided that there is still a need for diversity and well-intentioned experimentation in computing curricula; it is hoped the report will encourage this.

Reports of this type invariably create controversy, and there have indeed been objections to the report expressed by some of the reviewers as well as by others. Hopefully there will be lively debate about the recommendations of the report and about additional recommendations that were not included. Many efforts are under way to implement the report's recommendations, and the results of those efforts will be useful in evaluating the recommendations.

A great deal of thanks goes to the members of the task force who contributed so much time and effort toward the completion of the report and also to the reviewers who provided comments and suggestions. The task force was cochaired by Allen Tucker, representing ACM, and Bruce Barnes, representing the Computer Society of the IEEE. The other members are listed as coauthors of the summary report that follows this introduction. The ACM Education Board is pleased to present this report and to sincerely thank all members of the task force for a lot of hard work and an important contribution to undergraduate education in the discipline of computing.

A. Joe Turner *Chair, ACM Education Board*

This article is a summary of the report *Computing Curricula 1991* [4], which is published as a separate document by the Association for Computing Machinery (ACM) and the Computer Society of the IEEE (IEEE-CS). The purpose of this article is to provide a sufficiently complete snapshot of that report's content so that readers will have a clear sense of its scope, methodology, and general recommendations. While a complete curriculum design document is not presented here, the full report does have that goal. Readers who are interested in using these recommendations as a basis for curriculum design will therefore need to obtain a copy of the full report itself.

In the spring of 1988, the ACM and the IEEE-CS formed the Joint Curriculum Task Force. The charter of the Task Force was to present recommendations for the design and implementation of undergraduate (baccalaureate) curricula in the discipline of computing. Throughout this article, the term *computing* is used to encompass the labels "computer science," "computer science and engineering," "computer engineering," "informatics," and other similar designations for academic programs. Programs in related areas, such as information systems, were not considered by the Task Force.

These recommendations supersede the separate curriculum recommendations [2, 3, 7] of the ACM and the IEEE-CS. Since those recommendations were published, significant changes have been made to the introductory courses [9, 10]. The evolution of accreditation guidelines [1, 5, 11], alternative model curricula [8], and the recent report *Computing as a Discipline* [6] all testify to the rapid and fundamental changes that have taken place. A strong motivation for making such a joint effort at this time thus comes from the fact that the discipline and its pedagogy have

changed significantly in the last several years.

Another motivation for this work comes from the growing recognition that, despite strong and fundamental differences among institutions offering undergraduate programs in computing, these programs share a substantially large curriculum in common. Any curriculum recommendations that attempt to speak for the entire discipline must not only identify this shared subject matter, but also suggest ways in which it can serve as a basis for building undergraduate programs in different kinds of institutions. This work thus serves the interests of a wide constituency: in effect, the faculties of all undergraduate programs that offer concentration programs in computing, whether they occur in colleges of arts and sciences, colleges of engineering, liberal art colleges, or other academic contexts.

Objectives and Overview

The report *Computer Curricula 1991* has the following primary objective: to provide curricular guidance for implementing undergraduate programs in the discipline of computing. The curriculum recommendations are based upon goals for programs and upon a definition of the discipline.

As a second means for attaining this objective, the report's guidelines are based on a comprehensive definition of the discipline, provided in the report *Computing as a Discipline* [6]. This definition was chosen because:

- it is up-to-date and widely available;
- it was developed by a task force of widely respected scholars in a cooperative effort between the ACM and the IEEE-CS; and
- it provides a detailed and comprehensive specification of the subject matter of the discipline.

When used as a basis for curriculum design, this definition provides a conceptual and organizational context from which a common collection of subject matter for all undergraduate programs can be drawn. In the present article, this shared collection of subject matter is called the *common requirements*. That definition also provides the conceptual context from which the advanced and supplementary subject matter can be drawn.

This article is presented in the following major parts:

- A set of curricular and pedagogical considerations that govern the mapping of the common requirements and advanced/supplemental material into a complete undergraduate degree program. These considerations include the roles of laboratories, programming, mathematics and science, professionalism, a new notion called *recurring concepts*, and other educational experiences that combine to make up an entire undergraduate degree program in computing.
- A collection of subject matter modules called *knowledge units* that comprise the common requirements for all undergraduate programs in the field of computing.
- A collection of advanced and supplementary curriculum material that provides depth of study in several of the subject areas. The coverage of this material will vary in accordance with differing overall degree requirements of different types of institutions.

Because these curriculum recommendations are intentionally flexible, they do not prescribe a single set of courses for all undergraduate programs in computing. Instead, they provide guidelines that allow individual departments to design their own programs according to local objectives and constraints.

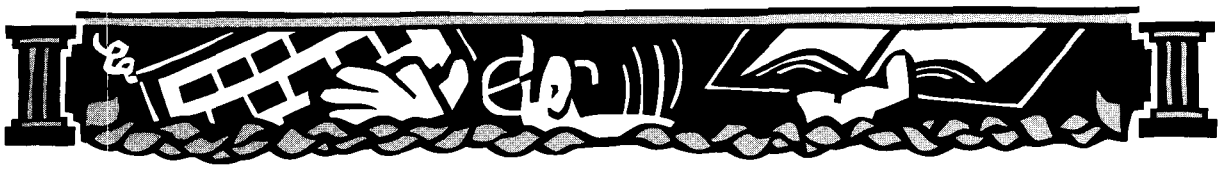
Relationship with Previous Curricular Recommendations

The curricular guidelines in the report *Computing Curricula 1991* are influenced by a number of prior curricular efforts that preceded it, including the ACM guidelines [3], the IEEE-CS guidelines [7], and other alternative guidelines (e.g., [8]). This section summarizes the major similarities and differences between the present report and its predecessors.

These prior reports influenced the report in the following major ways:

- The report *Computing as a Discipline* advocated the integration of laboratory work with classroom lectures, affirmed the importance of design in the curriculum, and proposed a breadth-first approach for the introductory courses in the curriculum.
- The 1983 IEEE-CS report contained in-depth descriptions of important topic areas for computer science and engineering, included laboratory material to support the lecture topics, and used modules and submodules as a basis for organizing subject matter and constructing courses.
- The 1978 ACM report provided detailed course descriptions for undergraduate programs in computer science, and established a prominent role for programming in the curriculum.

While the report *Computing Curricula 1991* also promotes broad discipline coverage for all programs, it offers alternative ways of achieving such breadth. Some alternatives include a breadth-first approach to the introductory courses while others include a more traditional approach. More generally, the present report has significantly broader overall goals than does the *Computing as a Discipline* report. The present report addresses the curricular needs of a broader range of institutions and programs in computing than does the 1983 IEEE-CS report or the 1978 ACM report. Its subject matter is also more current and it casts the role of program-



ming and design in a different light. Overall, it attempts to integrate theory, abstraction, design, and the social context of computing into the curriculum in more compelling ways.

In summary, the report is influenced by its predecessors, yet it is quite different in scope and objectives from any one of them. It reflects the rapid and dramatic evolution of the discipline of computing and its pedagogy that has taken place during the last several years. It is holistic, attempting to reach a wider constituency than any of its predecessors. It is intentionally designed to encourage curriculum innovation and evolution, enabling educators to respond in a timely fashion to future changes in the discipline rather than to simply update earlier models.

Undergraduate Program Goals/Graduate Profiles

Undergraduate programs in computing share common attributes, values, and curricula, and so do their graduates. The following discussion reflects the Task Force's view of these shared attributes and values that serve as a basis for the curriculum recommendations that follow.

Undergraduate programs should prepare graduates to understand the field of computing, both as an academic discipline and as a profession within the context of a larger society. Thus, graduates should be aware of the history of computing, including those major developments and trends—economic, scientific, legal, political, and cultural—that have combined to shape the discipline during its relatively short life.

The first goal for undergraduate programs, therefore, is to provide a coherent and broad-based coverage of the discipline of computing. Graduates should develop a reasonable level of understanding in each of the subject areas and processes that define the discipline, as well as an appreciation for the interrelationships that exist among them.

A second goal for undergraduate programs in computing is to function effectively within the wider intellectual framework that exists within the institutions that house the programs. These institutions vary widely in their respective missions. Some of them emphasize breadth of study over depth, while others emphasize the opposite. Some are rigid in the overall balance between requirements and electives, while others are more flexible.

Third, different undergraduate programs place different levels of emphasis upon the objectives of preparing students for entry into the computing profession, preparing students for graduate study in the discipline of computing, and preparing students for the more general challenges of professional and personal life.

Fourth, undergraduate programs should provide an environment in which students are exposed to the ethical and societal issues that are associated with the computing field. This includes maintaining currency with recent technological and theoretical developments, upholding general professional standards, and developing an awareness of one's own strengths and limitations, as well as those of the discipline itself.

Fifth, undergraduate programs should prepare students to apply their knowledge to specific, constrained problems and produce solutions. This includes the ability to define a problem clearly; to determine its tractability; to study, specify, design, implement, test, modify, and document that problem's solution; and to work within a team environment throughout the entire problem-solving process.

Finally, undergraduate programs should provide sufficient exposure to the rich body of theory that underlies the field of computing, so that students appreciate the intellectual depth and abstract issues that will continue to challenge researchers in the future.

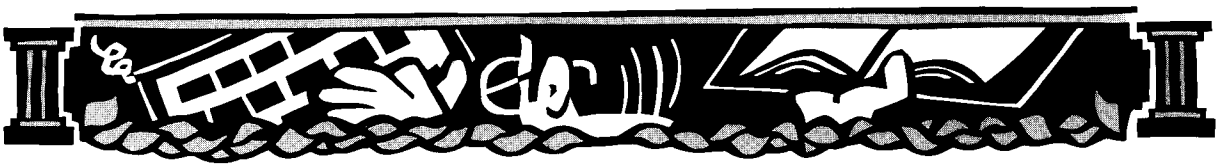
Underlying Principles for Curriculum Design

As noted earlier, the report *Computing as a Discipline* presented a comprehensive and contemporary definition of the discipline of computing. This definition provides a conceptual basis for defining a new teaching paradigm, as well as undergraduate curriculum design guidance for the discipline. The definition includes a specification of the subject matter by identifying nine constituent subject areas and three processes that characterize different working methodologies used in computing research and development. That specification is used in significant ways in this article.

From the subject matter of the nine areas of the discipline, this article identifies a body of fundamental material called the *common requirements,* to be included in every program. The curriculum of each program must also contain substantial emphasis on each of the three processes, which are called *theory, abstraction,* and *design.* In addition, this article identifies a body of subject matter representing the social and professional context of the discipline, also considered to be essential for every program. Finally, a set of concepts that recur throughout the discipline, and that represent important notions and principles that remain constant as the subject matter of the discipline changes, are an important component of every program.

The Nine Subject Areas

Nine subject areas are identified *in Computing as a Discipline* as comprising the subject matter of the discipline. Each of these areas has a significant theoretical base, significant abstractions, and significant design and implementation achievements. While these subject area definitions



cover the entire discipline, they each contain certain fundamental subjects that should be required in all undergraduate programs in computing; this fundamental subject matter is identified later as the common requirements for all programs. On the other hand, certain parts of these subject areas are less central and are therefore not included in the common requirements. These topics are left for the advanced components of the undergraduate or graduate curriculum.

Thus, in some cases, major components of a subject area that appear in its title are not included in the common requirements. For example, the common requirements do not contain subject matter on symbolic computation. However, the subject area title "Numerical and Symbolic Computation" from *Computing as a Discipline* is retained, both in the interest of continuity and because additional subject matter will appear in the advanced components of computing curricula. For instance, symbolic computation appears among the advanced and supplemental topics that are described in an upcoming section, even though it does not appear among the common requirements.

The nine subject areas are:

- Algorithms and Data Structures
- Architecture
- Artificial Intelligence and Robotics
- Database and Information Retrieval
- Human-Computer Communication
- Numerical and Symbolic Computation
- Operating Systems
- Programming Languages
- Software Methodology and Engineering

More detailed discussion of these areas can be found in the report *Computing as a Discipline,* as well as the full report *Computing Curricula 1991* itself.

The Three Processes: Theory, Abstraction, and Design

Because computing is simultaneously a mathematical, scientific, and engineering discipline, different practitioners in each of the nine subject areas employ different working methodologies, or processes, during the course of their research, development, and applications work.

One such process, called *theory,* is akin to that found in mathematics, and is used in the development of coherent mathematical theories. An undergraduate's first encounter with theory in the discipline often occurs in an introductory mathematics course. Further theoretical material emerges in the study of algorithms (complexity theory), architecture (logic), and programming languages (formal grammars and automata). The present curriculum recommendations contain a significant amount of theory that should be mastered by undergraduates in all programs.

The second process, called *abstraction,* is rooted in the experimental sciences. Undergraduate programs in computing introduce students to the process of abstraction in a variety of ways, both in classes and in laboratories. For example, the basic von Neumann model of a computer is a fundamental abstraction whose properties can be analyzed and compared with other competing models. Students are introduced to this model early in their undergraduate coursework. Undergraduate laboratory experiments that emphasize abstraction stress analysis and inquiry into the limits of computation, the properties of new computational models, and the validity of unproven theoretical conjectures.

The third process, called *design,* is rooted in engineering and is used in the development of a system or device to solve a given problem. Undergraduates learn about design both by direct experience and by studying the designs of others. Many laboratory projects are design-oriented, giving students firsthand experience with developing a system or a component of a system to solve a particular problem. These laboratory projects emphasize the synthesis of practical solutions to problems and thus require students to evaluate alternatives, costs, and performance in the context of real-world constraints. Students develop the ability to make these evaluations by seeing and discussing example designs as well as receiving feedback on their own designs.

In all nine subject areas of computing, these three processes of theory, abstraction, and design appear prominently and indispensably. A thorough grounding in each process is thus fundamental to all undergraduate programs in the discipline.

Social and Professional Context

Undergraduates also need to understand the basic cultural, social, legal and ethical issues inherent in the discipline of computing. They should understand where the discipline has been, where it is, and where it is heading. They should also understand their individual roles in this process, as well as appreciate the philosophical questions, technical problems, and aesthetic values that play an important part in the development of the discipline.

Students also need to develop the ability to ask serious questions about the social impact of computing and to evaluate proposed answers to those questions. Future practitioners must be able to anticipate the impact of introducing a given product into a given environment. Will that product enhance or degrade the quality of life? What will the impact be upon individuals, groups, and institutions?

Finally, students need to be aware of the basic legal rights of software and hardware vendors and users, and they also need to

appreciate the ethical values that are the basis for those rights. Future practitioners must understand the responsibility they will bear, and the possible consequences of failure. They must understand their own limitations as well as the limitations of their tools. All practitioners must make a long-term commitment to remaining current in their chosen specialties and in the discipline of computing as a whole.

To provide this level of awareness, undergraduate programs should devote explicit curricular time to the study of social and professional issues. The subject matter recommended for this study appears in the knowledge units under the heading *SP: Social, Ethical, and Professional Issues.*

Recurring Concepts

The discussion thus far has emphasized the division of computing into nine subject areas, three processes, and its social and professional context. However, certain fundamental concepts recur throughout the discipline and play an important role in the design of individual courses and whole curricula. *Computing as a Discipline* refers to some of these concepts as *affinity groups* or *basic concerns throughout the discipline.*[1] The Task Force refers to these fundamental concepts as *recurring concepts* in this article.

Recurring concepts are significant ideas, concerns, principles and processes that help to unify an academic discipline. An appreciation for the pervasiveness of these concepts and an ability to apply them in appropriate contexts is one indicator of a graduate's maturity as a computer scientist or engineer. Clearly, in designing a particular curriculum, these recurring concepts must be communicated in an effective manner; it is important to note that the appropriate use of the recurring concepts is an essential element in the implementation of curricula and courses based upon the specifications given in this article. Additionally, these concepts can be used as underlying themes that help tie together curricular materials into cohesive courses.

Each recurring concept listed in this article

- Occurs throughout the discipline,
- Has a variety of instantiations,
- Has a high degree of technological independence.

Thus, a recurring concept is any concept that pervades the discipline and is independent of any particular technology. A recurring concept is more fundamental than any of its instantiations. A recurring concept has established itself as fundamental and persistent over the history of computing and is likely to remain so for the foreseeable future.

In addition to the three characteristics given above, most recurring concepts

- Have instantiations at the levels of theory, abstraction and design,
- Have instantiations in each of the nine subject areas,
- Occur generally in mathematics, science and engineering.

These additional points make a strong assertion concerning the pervasiveness and persistence of most of the recurring concepts. Not only do they recur throughout the discipline, they do so across the nine subject areas and across the levels of theory, abstraction and design. Furthermore, most are instances of even more general concepts that pervade mathematics, science and engineering.

The following is a list of 12 recurring concepts that are identified as fundamental to computing. Each concept is followed by a brief description.

Binding: the processes of making an abstraction more concrete by associating additional properties with it. Examples include associating (assigning) a process with a processor, associating a type with a variable name, associating a library object program with a symbolic reference to a subprogram, instantiation in logic programming, associating a method with a message in an object-oriented language, creating concrete instances from abstract descriptions.

Complexity of large problems: the effects of the nonlinear increase in complexity as the size of a problem grows. This is an important factor in distinguishing and selecting methods that scale to different data sizes, problem spaces, and program sizes. In large programming projects, it is a factor in determining the organization of an implementation team.

Conceptual and formal models: various ways of formalizing, characterizing, visualizing and thinking about an idea or problem. Examples include formal models in logic, switching theory and the theory of computation, programming language paradigms based upon formal models, conceptual models such as abstract data types and semantic data models, and visual languages used in specifying and designing systems, such as data flow and entity-relationship diagrams.

Consistency and completeness: concrete realizations of the concepts of consistency and completeness in computing, including related concepts such as correctness, robustness, and reliability. Consistency includes the consistency of a set of axioms that serve as a formal specification, the consistency of theory to observed fact, and internal consistency of a language or interface design. Correctness can be viewed as the consistency of component or system behavior to stated specifications. Completeness includes the adequacy of a given set of axioms to capture all desired behaviors, the functional adequacy

[1]Page 9 of [6].

of software and hardware systems, and the ability of a system to behave well under error conditions and unanticipated situations.

Efficiency: measures of cost relative to resources such as space, time, money, and people. Examples include the theoretical assessment of the space and time complexity of an algorithm, the efficiency with which a certain desirable result (such as the completion of a project or the manufacture of a component) can be achieved, and the efficiency of a given implementation relative to alternative implementations.

Evolution: the fact of change and its implications. This involves the impact of change at all levels and the resiliency and adequacy of abstractions, techniques and systems in the face of change. Examples include the ability of formal models to represent aspects of systems that vary with time, and the ability of a design to withstand changing environmental demands and changing requirements, tools and facilities for configuration management.

Levels of abstraction: the nature and use of abstraction in computing; the use of abstraction in managing complexity, structuring systems, hiding details, and capturing recurring patterns; the ability to represent an entity or system by abstractions having different levels of detail and specificity. Examples include levels of hardware description, levels of specificity within an object hierarchy, the notion of generics in programming languages, and the levels of detail provided in a problem solution from specifications through code.

Ordering in space: the concepts of locality and proximity in the discipline of computing. In addition to physical location, as in networks or memory, this includes organizational location (e.g., of processors, processes, type definitions, and associated operations) and conceptual location (e.g., software scoping, coupling, and cohesion).

Ordering in time: the concept of time in the ordering of events. This includes time as a parameter in formal models (e.g., in temporal logic), time as a means of synchronizing processes that are spread out over space, time as an essential element in the execution of algorithms.

Reuse: the ability of a particular technique, concept or system component to be reused in a new context or situation. Examples include portability, the reuse of software libraries and hardware components, technologies that promote reuse of software components, and language abstractions that promote the development of reusable software modules.

Security: the ability of software and hardware systems to respond appropriately to and defend themselves against inappropriate and unanticipated requests; the ability of a computer installation to withstand catastrophic events (e.g., natural disasters and attempts at sabotage). Examples include type-checking and other concepts in programming languages that provide protection against misuse of data objects and functions, data encryption, granting and revoking of privileges by a database management system, features in user interfaces that minimize user errors, physical security measures at computer facilities, and security mechanisms at various levels in a system.

Trade-offs and consequences: the phenomenon of trade-offs in computing and the consequences of such trade-offs; and the technical, economic, cultural and other effects of selecting one design alternative over another. Trade-offs are a fundamental fact of life at all levels and in all subject areas. Examples include space-time trade-offs in the study of algorithms, trade-offs inherent in conflicting design objectives (e.g., ease of use versus completeness, flexibility versus simplicity, low cost versus high reliability and so forth), design trade-offs in hardware, and trade-offs implied in attempts to optimize computing power in the face of a variety of constraints.

In constructing curricula from the overall specifications of the Task Force, curriculum designers must be aware of the fundamental role played by recurring concepts. That is, a recurring concept (or a set of recurring concepts) can help to unify the design of a course, a lecture, or a laboratory exercise. From the instructor's perspective (and also from the student's perspective), a course is rarely satisfying unless there is some "big idea" that seems to hold disparate elements together. We see the use of recurring concepts as one method for unifying the material this way.

At the level of the entire curriculum, the recurring concepts also play a unifying role. They can be used as threads that tie and bind different courses together. For example, in introducing the concept of *consistency* as applied to language design in a programming language course, the instructor might ask students to consider other contexts in which consistency played an important role, such as in a previous software design or computer organization course. By pointing out and discussing the recurring concepts as they arise, instructors can help portray computing as a coherent discipline rather than as a collection of unrelated topics.

From Principles to Curriculum

An undergraduate curriculum in computing should provide each graduate with a reasonable level of instruction in all of the subject areas identified above. This allows each graduate to achieve a *breadth* of

understanding across the entire discipline rather than just in a few of its parts. Breadth is ensured in the present guidelines by way of the common requirements.

The undergraduate curriculum should also provide reasonable *depth* of study in some of the nine subject areas of the common requirements. While the particular way in which subject area depth is achieved will vary among different types of programs, it is essential that depth of study be achieved in one way or another. Depth is ensured in the present guidelines by way of the *advanced/supplemental topics* (see following discussion).

Among the three processes—theory, abstraction, and design—it is fair to assume that some undergraduate programs emphasize more theory than design, while others emphasize more design than theory. However, the process of abstraction will normally be prominent in all undergraduate curricula. Theory, abstraction, and design are included throughout the common requirements, and are reinforced by the integration of laboratory work with subject matter in a principled and thorough way.

To support the development of maturity in the mathematical and scientific aspects of computing, an undergraduate curriculum should also include certain mathematics and science subject matter to complement the subject matter in the discipline itself. Similarly, to support the development of maturity in the scientific and engineering aspects of the discipline, the present guidelines recommend that students regularly engage in laboratory work and other educational experiences.

An overview of a complete undergraduate program in computing can be summarized as shown in Figure 1. This summary provides a general level of guidance for implementing undergraduate programs in the discipline. There, the need to recognize different institutional

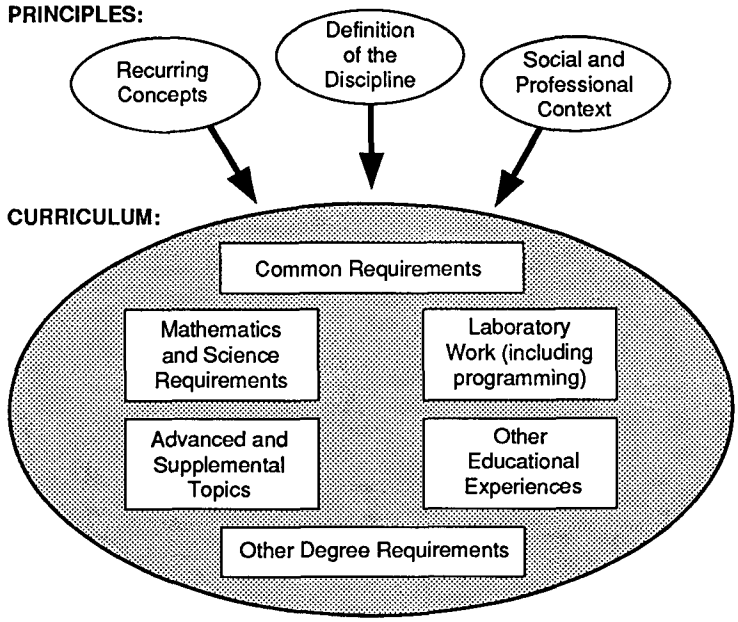


FIGURE 1.
A Complete Curriculum and Its Underlying Principles

contexts and programmatic goals is accommodated by the box in that figure labeled *Other Degree Requirements.*

Assuring Breadth and Depth

To realize the breadth requirements, undergraduate programs should provide a broad selection of topics taken from all nine subject areas of computing. However, because different institutions will want to fulfill these common requirements in different ways, they are not presented as a single prescribed set of courses. Instead, the common requirements' topics are presented as a collection of *knowledge units.* These can be combined in various ways to form different sets of courses, or *implementations,* in different undergraduate settings. Further discussion of constructing courses out of knowledge units appears later, and many examples are given in the full report itself.

To realize the depth requirements, different institutions will offer additional requirements and electives in accordance with their overall educational missions, faculty size, and subject area expertise. The full report describes a number of advanced and supplemental topics that provide depth of study in the nine areas of the discipline of computing.

The Role of Programming

The term *programming* is understood to denote the entire collection of activities that surround the description, development, and effective implementation of algorithmic solutions to well-specified problems. While this definition is not to be construed to mean simply "coding in a particular programming language or for a particular machine architecture," it does necessarily include the mastery of highly stylized skills in a particular programming language or languages. Thus, fluency in a programming language is an essential attribute of the mastery of programming. On the other hand, programming is not to be construed so broadly as to subsume all of the activities involved in software methodology

and engineering. The latter is a much broader notion and includes, for example, the development of specifications and the maintenance of software.

Programming occurs in all nine subject areas in the discipline of computing. It is part of the design process, it is used to implement the models that occur in the abstraction process, and sometimes it even occurs in the process of proving a theoretical result. Thus, the role of programming in the undergraduate curriculum is also multidimensional; students develop programs during the design of software, they exercise and modify programs during other laboratory experiments, and they read programs in the normal course of studying subject matter in textbooks and the published literature. In this latter sense, programming is an extension of the basic skills that students and professionals normally use in day-to-day communication.

Mastery of programming can be accomplished in several ways during an undergraduate's career. First, a substantial portion of the common requirements' laboratory work contains programming as an essential part. For example, the knowledge unit *SE1: Fundamental Problem Solving Concepts* requires laboratory work in which students design, implement, and exercise programs in a modern programming language. Second, undergraduate texts in the various subject areas teach and use conventional programming techniques throughout their presentations of subject matter. Lectures in courses throughout the curriculum also use well-established styles of programming to explain or clarify algorithmic concepts. Third, the computing literature is replete with programs and examples of programming methodology, and undergraduates become familiar with this literature during their studies. Instructors should take care to present examples of good programming style for students to study and emulate.

Thus, undergraduates should develop an early understanding of programming. They should continue to actively engage in the mastery and use of programming paradigms and languages throughout their coursework, homework, and laboratory exercises.

It should be noted here that the common requirements do not assume that students will have any experience with programming before taking their first course in the discipline. However, increasing numbers of students do gain such experience in secondary school. Many feel the amount of attention traditionally paid to the syntax of a programming language in the first course is excessive, and ought to be replaced with a more balanced introduction to the discipline.

For these reasons, *PR: Introduction to a Programming Language* is defined as a separate knowledge unit, but is not a *required* part of the common requirements. That is, most programs will be able to introduce a programming language in conjunction with other knowledge unit material, especially the one named SE1, during the scheduled laboratory periods that accompany the first course. For those programs that do require a more careful treatment of this topic, the optional knowledge unit *PR: Introduction to a Programming Language* can be added to the common requirements to fulfill this need.

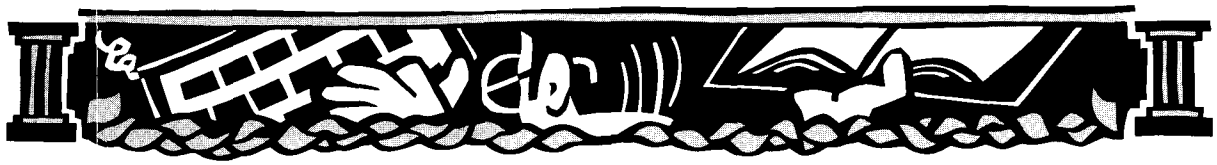
The Role of Laboratories

The report *Computing as a Discipline* makes a clear statement about the purpose and structure of laboratory work in the undergraduate curriculum; this article expands upon that statement by identifying in more detail how these very important laboratory activities can be integrated into an undergraduate curriculum. An undergraduate curriculum in computing is ideally comprised of an integrated program of lectures and laboratory experiences. The learning process occurs as a result of interaction among students, instructors, and the subject matter.

Laboratories demonstrate the application of principles to the design, implementation, and testing of software and hardware systems. Laboratories also emphasize techniques that utilize contemporary tools and lead to good experimental methods, including the use of measuring instruments, diagnostic aids, software and hardware monitors, statistical analysis of results, and oral and written presentation of findings. The laboratories should augment the instruction that takes place in the lectures by having clearly stated objectives that complement the lectures.

The laboratory exercises included in the *Suggested Laboratories* part of each knowledge unit provide a diverse set of learning experiences. Some involve the execution of hardware, software, or simulators to observe some phenomenon, either by data collection or by visualization. Others are designed to increase student expertise in software methodology through the development of alternative design and implementation techniques. Still others are similar to science experiments because they involve hypothesis formation and testing. These types of laboratory experiences combine to increase student problem-solving ability, analytical skill, and professional judgment.

In the presentation of laboratories among the knowledge units, two distinct types of laboratory work are offered. These are called *open* laboratories and *closed* laboratories. An open laboratory is an unsupervised assignment that may involve the use of a computer, software, or hardware for its completion. Students can complete an open laboratory at their own convenience as it does not require direct supervision. Conventional programming assignments are often done in an open laboratory setting.



A closed laboratory is a scheduled, structured, and supervised assignment that involves the use of computing hardware, software, or instrumentation for its completion. Students complete a closed lab by attending a scheduled session, usually 2–3 hours long, at a specific facility. Supervision is provided by the instructor or a qualified assistant who is familiar with the details of the assignment. The specialized equipment, software, and supervision offered by closed laboratories makes them more desirable than open laboratories in certain situations. Closed laboratories are particularly important in situations where the assignment relies on instructor-student interaction or a team effort among students to complete the work. For example, the use of software tools that facilitate the production of large-scale software, CAD programs, or other packages of considerable complexity normally require the initial guidance and advice of an expert.

Whether the lab is open or closed, the facility should be staffed by a knowledgable person during all scheduled laboratory hours. Staffing ensures immediate feedback and guidance during times when students are working independently on laboratory assignments.

The designation *open* or *closed,* along with the laboratories themselves, should be used as suggestions, or statements of preference by the Task Force, rather than prescriptions for all programs to follow. That is, some laboratories that are designated as closed may be done as open laboratories and vice versa. This choice may depend on how the particular lab experience and objectives fit into the curriculum as well as the available facilities in a particular institutional setting.

Whether open or closed, a laboratory assignment should always be carefully planned by the instructor. Descriptions that include clear statements of purpose, methodology, and results should be carefully prepared for the students. Laboratory assignments should be realistically designed, so that an average student can complete the work in the allotted time. It is particularly important to make sure that adequate facilities are available to support the goals of each laboratory assignment.

Completion of a laboratory assignment should be accompanied by a written or oral report by the student. Written reports should reflect a disciplined and mature writing style, in addition to the successful completion of the laboratory work itself. It is expected that there will be homework in addition to laboratory assignments, and this work should also be well-integrated with the subject matter of the lectures.

Other Educational Experiences

Beyond the subject matter of the curriculum, undergraduates in computing should have additional experiences that will help them develop the capacity for critical thinking, problem solving, research methods, and professional development. These experiences can be incorporated into the classroom lectures, laboratories, and extracurricular activities of the undergraduate program. These additional experiences generally fall into three categories:

1. working as part of a team
2. written and oral communication
3. familiarization with the profession

For further discussion of these requirements, readers are encouraged to consult the full report.

The Common Requirements

The common requirements form the basis for a curriculum in computing by providing a platform of knowledge that is considered essential for all students who concentrate in the discipline. These are not the only topics that should be covered, yet they contain the basic body of knowledge that should be part of every curriculum. The common requirements are expressed here as *knowledge units* rather than complete courses, to allow different programs to package the subject matter in different ways. Such variations will occur because different institutions and types of programs will have different pedagogical priorities, educational goals, and general constraints within which they implement the common requirements.

While every knowledge unit is considered to be essential, the depth and breadth of coverage for each topic therein will not be the same. For example, the knowledge unit *AL6: Sorting and Searching* recommends that various sorting and searching algorithms be covered in six lecture hours and associated laboratory work. However, this is only about two weeks' time. Thus, only a few of the multitude of sorting techniques can be covered in any appreciable depth—perhaps insertion sort, heap sort, and quicksort—while various other sorting techniques can be covered in a more surveylike fashion. Instructors will inevitably make tradeoffs between depth and breadth of coverage, given the constraints of the number of lectures in a knowledge unit, in the same way they do now with current textbooks. Furthermore, some knowledge units offer only a broad level of coverage of a topic (see, for instance, the knowledge units designated as PL1, OS1, or AI1). Additional depth in these subjects can be obtained only by exceeding the coverage recommended by the common requirements, either in the coverage of the knowledge unit, or in an advanced course.

Organizing the Common Requirements: The Knowledge Unit

In the following discussion, a *knowledge unit* is understood to designate

a coherent collection of subject matter that is so fundamental within one of the nine subject areas of the discipline listed earlier, or within the area of social, ethical, and professional issues (SP), that it should occur in every undergraduate curriculum. While the subject matter of a knowledge unit is often related to other knowledge units in the common requirements by way of a prerequisite structure, it can nevertheless be introduced within any of several alternative course structures.

For easy cross-referencing among the knowledge units, Table 1 gives the two-letter tags used to identify each of the nine subject areas, the additional area of social and professional context (SP), and the optional introduction to a programming language (PR).

The collected knowledge units are organized by subject area. Each knowledge unit within a subject area is identified by the tag for that subject area. For example, the knowledge units in the area of Algorithms and Data Structures are identified by the tags AL1, AL2, AL3, and so forth.

The following sample knowledge unit illustrates the presentation style for knowledge units:

AL6: Sorting and Searching
Comparison of various algorithms for sorting and searching, with focus on complexity and space versus time trade-offs.

Recurring Concepts: complexity of large problems, consistency and completeness, efficiency, trade-offs and consequences.

Lecture Topics: (six hours minimum)
1. $O(n^2)$ sorting algorithms (e.g., insertion and selection sort); space-time complexity: best, worst cases
2. $O(n \log n)$ sorting algorithms (e.g., quicksort, heapsort, mergesort); space-time complexity: best, worst cases
3. Other sorting algorithms (e.g., Shell sort, bucket sort, radix sort)
4. Comparisons of algorithms
5. Serial search, binary search and binary search tree; space-time complexity: best, worst cases
6. Hashing, collision resolution

Suggested Laboratories: (closed) Corroboration of theoretical complexity of selected sorting and searching algorithms by experimental methods, identifying differences among best, average, and worst case behaviors.

Connections:
Related to: AL5, AL8
Prerequisites: AL4
Requisite for: AI2, OS7, PL9

TABLE 1.

The Subject Areas and Tags

Subject Area	Tag
Algorithms and Data Structures	AL
Architecture	AR
Artificial Intelligence and Robotics	AI
Database and Information Retrieval	DB
Human-Computer Communication	HU
Numerical and Symbolic Computation	NU
Operating Systems	OS
Programming Languages	PL
Introduction to a Programming Language (optional)	PR
Software Methodology and Engineering	SE
Social, Ethical, and Professional Issues	SP

Table 2 offers a complete list of the titles of the knowledge units that comprise the common requirements. While these titles suggest something about the knowledge units' contents, a detailed presentation of each one is given in the full report [4].

The Advanced and Supplemental Curriculum

A complete curriculum will include not only the common requirements, but also certain additional material. This advanced and supplemental material gives each individual student an opportunity to study the subject areas of the discipline in depth. The curriculum should provide depth of study in several of the nine subject areas beyond that provided by the common requirements. Students normally achieve that depth by completing several additional courses in this part of the curriculum. The number of such courses will vary in accordance with institutional norms. The sample implementations in the appendix of the full report illustrate these kinds of variations.

The topics in the following list should be considered as areas where courses may be developed to provide in-depth study in advanced undergraduate and graduate courses. Other topics beyond these are important as well, but will vary with the particular interests and expertise of the faculty in individual programs. However, these topics tend to be so significant to the discipline at this time that several of them ought to appear among the advanced courses offered by any undergraduate program.

Advanced Operating Systems
* Advanced Software Engineering
Analysis of Algorithms
* Artificial Intelligence
Combinatorial and Graph Algorithms
Computational Complexity

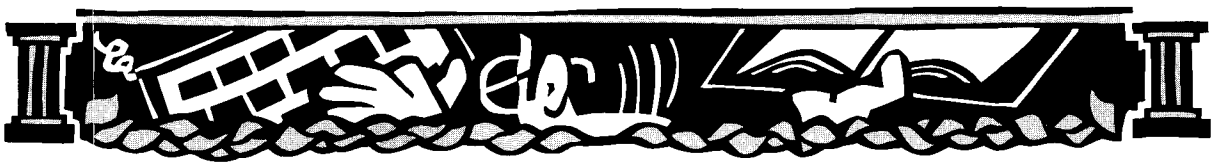


TABLE 2.

Summary of the Common Requirements

AL: Algorithms and Data Structures (approximately 47 lecture hours)
- AL1: Basic Data Structures
- AL2: Abstract Data Types
- AL3: Recursive Algorithms
- AL4: Complexity Analysis
- AL5: Complexity Classes
- AL6: Sorting and Searching
- AL7: Computability and Undecidability
- AL8: Problem-Solving Strategies
- AL9: Parallel and Distributed Algorithms

AR: Architecture (approximately 59 lecture hours)
- AR1: Digital Logic
- AR2: Digital Systems
- AR3: Machine-Level Representation of Data
- AR4: Assembly-Level Machine Organization
- AR5: Memory System Organization and Architecture
- AR6: Interfacing and Communication
- AR7: Alternative Architectures

AI: Artificial Intelligence and Robotics (approximately nine lecture hours)
- AI1: History and Applications of Artificial Intelligence
- AI2: Problems, State Spaces, and Search Strategies

DB: Database and Information Retrieval (approximately nine lecture hours)
- DB1: Overview, Models, and Applications of Database Systems
- DB2: The Relational Data Model

HU: Human-Computer Communication (approximately eight lecture hours)
- HU1: User Interfaces
- HU2: Computer Graphics

NU: Numerical and Symbolic Computation (approximately seven lecture hours)
- NU1: Number Representation, Errors, and Portability
- NU2: Iterative Approximation Methods

OS: Operating Systems (approximately 31 lecture hours)
- OS1: History, Evolution, and Philosophy
- OS2: Tasking and Processes
- OS3: Process Coordination and Synchronization
- OS4: Scheduling and Dispatch
- OS5: Physical and Virtual Memory Organization
- OS6: Device Management
- OS7: File Systems and Naming
- OS8: Security and Protection
- OS9: Communications and Networking
- OS10: Distributed and Real-time Systems

PL: Programming Languages (approximately 46 lecture hours)
- PL1: History and Overview of Programming Languages
- PL2: Virtual Machines
- PL3: Representation of Data Types
- PL4: Sequence Control
- PL5: Data Control, Sharing, and Type Checking
- PL6: Run-time Storage Management
- PL7: Finite State Automata and Regular Expressions
- PL8: Context-Free Grammars and Pushdown Automata
- PL9: Language Translation Systems
- PL10: Programming Language Semantics
- PL11: Programming Paradigms
- PL12: Distributed and Parallel Programming Constructs

SE: Software Methodology and Engineering (approximately 44 lecture hours)
- SE1: Fundamental Problem-solving Concepts
- SE2: The Software Development Process
- SE3: Software Requirements and Specifications
- SE4: Software Design and Implementation
- SE5: Verification and Validation

SP: Social, Ethical, and Professional Issues (approximately 11 lecture hours)
- SP1: Historical and Social Context of Computing
- SP2: Responsibilities of the Computing Professional
- SP3: Risks and Liabilities
- SP4: Intellectual Property

* Computer Networks
* Computer Graphics
  Computer-Human Interface
* Computer Security
* Database and Information Retrieval
  Digital Design Automation
  Fault-Tolerant Computing
  Information Theory
  Modeling and Simulation
  Numerical Computation
* Parallel and Distributed Computing
  Performance Prediction and Analysis
  Principles of Computer Architecture
  Principles of Programming Languages
* Programming Language Translation
  Real-Time Systems
  Robotics and Machine Intelligence
  Semantics and Verification
  Societal Impact of Computing
* Symbolic Computation
* Theory of Computation
* VLSI System Design

Several of the topics in this list are marked with an asterisk (*). This denotes that a more complete description is given in the full report *Computing Curricula 1991*. Those descriptions are intended to give more concrete information that will assist in developing courses in these topic areas.

Mathematics and Science Requirements

An understanding of mathematics and science is important for students who concentrate their studies in computing.

Mathematical maturity, as commonly attained through logically rigorous mathematics courses, is essential to successful mastery of several fundamental topics in computing. Thus, all computing students should take at least one-half year[2] of mathematics courses. These courses should cover at least the following subjects:

Discrete Mathematics: sets, functions, elementary propositional and predicate logic, Boolean algebra, elementary graph theory, matrices, proof techniques (including induction and contradiction), combinatorics, probability, and random numbers

Calculus: differential and integral calculus, including sequences and series and an introduction to differential equations

It should be noted that some of the discrete mathematics topics should be treated early in the curriculum, since they are needed for some of the basic knowledge units.

The half-year mathematics requirement should also include at least one of the following subjects:

Probability: discrete and continuous, including combinatorics and elementary statistics

Linear Algebra: elementary, including matrices, vectors, and linear transformations

Advanced Discrete Mathematics: a second course covering more advanced topics in discrete mathematics

Mathematical Logic: propositional and functional calculi, completeness, validity, proof, and decision problems

Many implementations will have additional mathematics requirements beyond this minimum set. For example, professionally oriented programs will normally require five or six mathematics courses. Students who wish to pursue graduate study in computing are often well-advised to take more mathematics.

Science is important in computing curricula for three reasons. First, as well-educated scientists and engineers, graduates of computing programs should be able to appreciate advances in science because they have an impact on society and on the field of computing. Second, exposure to science encourages students to develop an ability to apply the scientific method in problem solving. Third, many of the applications students will encounter after graduation are found in the sciences.

For these reasons, all computing curricula should include a component that incorporates material from the physical and life sciences. Ideally, courses in this component are those which are designed for science majors themselves.

Programs intended to prepare students for entry into the profession should require a minimum of one-half year of science. This normally includes a year-long course in a laboratory science (preferably physics) and additional work in the natural sciences.

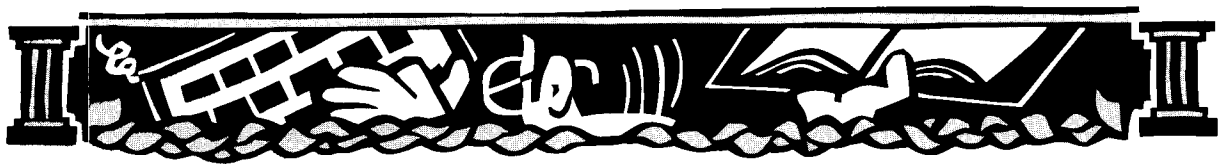
Building a Curriculum

Overall Design Considerations

A curriculum for a particular program depends on many factors, such as the purpose of the program, the strengths of the faculty, the backgrounds and goals of the students, instructional support resources, infrastructure support and, where desired, accreditation criteria. Each curriculum will be site-specific, shaped by those responsible for the program who must consider factors such as institutional goals, opportunities and constraints, local resources, and the prior preparation of students.

Developing a curriculum involves a design process similar to others with which computer scientists and engineers are familiar. A successful implementation will grow from a well-conceived and well-articulated specification of the purpose of the curriculum, the context in which the curriculum will be delivered, and other external opportunities and constraints. As with systems, the success of the implementation of a curriculum depends

[2]We mean the equivalent of one-half academic year of full-time study. Typically, this would be four or five semester-long courses.



on the care with which it is designed.

All curriculum designs, including but not limited to computing curricula, should be guided by certain principles. First, the purpose of a curriculum is to educate students. The curriculum design therefore should focus on providing a way for students to gain the desired knowledge, expertise and experience by the time they graduate. The outcome expected for students should drive the curriculum design.

Second, a curriculum is more than a set of isolated courses. There are unifying ideas and goals that span the whole curriculum. This article has included the discussion of recurring concepts as an indicator of those ideas that should be pervasive in a computing curriculum.

Third, there are many ways for students to learn beyond the standard lecture delivery format. Laboratories, in particular, present many opportunities for innovation. Faculty members should apply creative energy to developing alternative instructional methods.

Fourth, every environment has constraints that must be considered. A curriculum plan must not be overly optimistic nor overly restrictive, but should be realistic, both for students and for faculty. For example, a curriculum plan that requires a student to take four computing courses during the last term of the senior year invites a variety of problems.

Fifth, computing is more than just a collection of facts and algorithms. It is a dynamic, vital discipline that offers many challenges and interesting problems, exciting results, and imaginative applications. The curriculum should try to impart this sense of excitement to students; the material may be difficult, but it need not be dull.

Designing a computing curriculum adds another set of special concerns. The rapid change in the discipline itself demands that a computing curriculum be dynamic, not static. It must be built so that it can evolve along with the subject matter. All curricula should be evaluated periodically to see that they are meeting their goals; computing curricula should be evaluated for content to be sure they are up-to-date. Many departments also face changes that are caused by changing infrastructure, such as their organizations, enrollments, and funding levels.

Another concern in computing curricula comes from the fluid nature of theory in the discipline. As computing is based on artifacts in addition to physical laws, many of its "fundamentals" are subject to constant reinterpretation and reevaluation. Thus, the theory of computing evolves more rapidly than does theory in other sciences.

Computing, however, has one great advantage for curriculum designers. It provides frequent opportunities for accomplishing multiple goals through one instructional experience. Activities can be leveraged for several purposes. For example, a discussion of loops or recursion can simultaneously introduce the concept of searching by the use of appropriately chosen examples.

Designing Courses from Knowledge Units

The knowledge units enumerated earlier and described in the full report specify the scope of topics that all computing students should study. The lecture hours associated with each knowledge unit give an approximate indication of the depth to which these topics should be covered; we intend this to represent the minimum coverage that a typical program should ensure. The prerequisite structure suggests that some sequencing is required in the composition of courses out of knowledge units, but the size of the units allows many organizational options. Additionally, material in a knowledge unit may be split and covered at different times and in different courses, if deemed appropriate.

The knowledge units do not need to be covered completely in what would be identified as a set of "core courses," as long as they are all covered in one required course or another. Furthermore, some parts may be covered either in a standard class setting or in a laboratory setting. Thus, the knowledge units of the common requirements can be combined in various ways to form courses. In this activity, one may use the following guidelines:

- Knowledge units should be combined so that the composite subject matter forms a coherent body of topics for an undergraduate. In some cases, one or more of the recurring concepts can provide the "glue," while in other cases one of the nine subject areas can provide a basis for course organization.
- The combined set of courses that comprise an implementation should have a prerequisite structure that is consistent with the prerequisite structure that exists among their constituent knowledge units.
- The combined set of courses that comprise the implementation should cover all of the knowledge units that make up the common requirements. As specified here, these total about 271 lecture hours, which is equivalent to about seven one-semester courses.[3] This total does not include time spent in scheduled laboratory work, nor does it include the advanced and supplementary coursework that provides depth of study for all majors.

Implementations may, of course, exceed this minimum by assigning

[3]When the optional 12-hour knowledge unit *PR: Introduction to a Programming Language* is incorporated into the curriculum, the total number of lecture hours increases to about 283.

additional depth of coverage or topic selections beyond those that are suggested in the common requirements.

The following are two sample courses, an introductory course entitled "Problem-solving, Programs, and Computers" and another course entitled "Data Structures and Algorithms," that result from combining selected knowledge units under the foregoing guidance. The first of these courses takes knowledge units (KUs) from a number of different subject areas, and has "consistency and completeness" as a recurring concept. The second of these courses has all of its knowledge units taken from a single subject area.

Problem Solving, Programs, and Computers

Topic Summary: This course has three major themes: a rigorous introduction to the process of algorithmic problem solving, an introduction to the organization of the computers upon which the resulting programs run, and an overview of the social and ethical context in which the field of computing exists. Problem-solving rigor is guaranteed by a commitment to the precise specification of problems and a close association between the problem-solving process and that specification. For example, the identification of a loop is closely tied to the discovery of its invariant, which in turn is motivated by the problem specification itself.

Computer organization is introduced by way of a simple von Neumann machine model and assembler, upon which students can develop and exercise simple programs. Elements of the fetch-execute cycle, runtime data representation, and machine language program structure are thereby revealed.

This course contains 40 lecture hours of KU topics, and is taught as a four-credit-hour course. A scheduled weekly laboratory is used to teach programming language syntax and machine organization, as well as to support student programming exercises.

Prerequisites: An introduction to logic, as would usually be found at the beginning of a discrete mathematics course (that can be taken concurrently)

Knowledge units: AL3 (3/3), AL6 (2/6), AR3 (2/3), AR4 (7/15), NU1 (1/3), NU2 (2/4), PL1 (2/2), PL3 (2/2), PL4 (1/4), SE1 (11/16), SE5 (4/8), SP1 (3/3)[4]

Data Structures and Analysis of Algorithms

Topic Summary: This course covers data structures and algorithms in some depth. Topics covered include data structures, a more formal treatment of recursion, an introduction to basic problem-solving strategies, and an introduction to complexity analysis, complexity classes, and the theory of computability and undecidability. Sorting and searching algorithms are presented in the light of the presentation of problem-solving strategies and complexity issues. Finally, parallel and distributed algorithms are introduced briefly.

This course is taught in three lectures and a two-hour laboratory per week. It contains 33 lecture hours devoted to KU topics and their laboratories.

Prerequisites: "Computing II"

Knowledge units: AL1 (9/13), AL3 (2/3), AL4 (4/4), AL5 (4/4), AL6 (3/6), AL7 (3/6), AL8 (6/6), AL9 (2/3)

The full report contains many more examples of course descriptions that are appropriate in a variety of institutional settings.

Integrating the Curriculum into a Course of Study

The computing curriculum will have to be developed to meet institutional requirements and should take advantage of institutional strengths. Ultimately, the computing curriculum will be integrated into a complete four-year course of study.

Some suggestions for steps to start building a curriculum are:

- Identify goals of the program, focusing on student outcomes.
- Identify strengths of the faculty.
- Identify constraints of the local situation.
- Establish a plan and schedule for design, implementation, evaluation, modification, and transition.
- Design and implement the curriculum components.

Related Concerns

The report *Computing Curricula 1991* addresses the relationship between its recommendations with each of the following topics. For further details, readers are encouraged to consult the report itself.

- Faculty and Staff
- Laboratory Resources
- Service Courses and Joint Degree Programs
- Library Support
- Relationship with Accreditation, Placement Tests, and Achievement Tests

Summary

This article summarizes a specification for implementing undergraduate programs in computing, as opposed to a single curriculum. This approach is required because of the need to serve a large and diverse constituency. The ultimate success of this approach will therefore depend strongly upon the creative energies of faculty members at the various institutions that support undergraduate programs in computing.

This specification acknowledges the primary importance of the following elements in undergraduate computing curricula: nine major subject areas; theory, abstraction

[4]In the list of knowledge units that accompany a particular course description, the notation p/q means that p lecture years are used out of a total of q hours that are available from the knowledge unit. Thus, when p = q the entire knowledge unit is used in that course.

and design; recurring concepts; the social and professional context; mathematics and science; language and communication; and integrated laboratory experience. Successful undergraduate programs that follow from this specification will pay close attention to each of these elements in their implementations.

The appendix of the full report *Computing Curricula 1991* contains the following 12 sample curricula that are provided as "proofs of concept" for this specification.

- Implementation A: A Program in Computer Engineering
- Implementation B: A Program in Computer Engineering (Breadth-First)
- Implementation C: A Program in Computer Engineering (Minimal Number of Credit-Hours)
- Implementation D: A Program in Computer Science
- Implementation E: A Program in Computer Science (Breadth-First)
- Implementation F: A Program in Computer Science (Theoretical Emphasis)
- Implementation G: A Program in Computer Science (Software Engineering Emphasis)
- Implementation H: A Liberal Arts Program in Computer Science (Breadth-First)
- Implementation I: A Program in Computer Science and Engineering
- Implementation J: A Liberal Arts Program in Computer Science
- Implementation K: A Liberal Arts Program in Computer Science (Breadth-First)
- Implementation L: A Program in Computer Science (Theoretical Emphasis)

Acknowledgments

The Task Force wishes to thank the following persons who generously served as reviewers for this work in its various stages: Narenda Ahuja, Donald J. Bagert, Jr., James C. Bezdek, Nathaniel Borenstein, Richard J. Botting, Donald Bouldin, Albert W. Briggs, Jr., J. Glenn Brooksheer, Wai-Kai Chen, Lucio Chiaraviglio, Neal S. Coulter, Steve Cunningham, Ruth Davis, Peter J. Denning, Scot Drysdale, Larry A. Dunning, Peter Durato, J. Philip East, Adel S. Elmaghraby, John W. Fendrich, Gary A. Ford, Edwin C. Foudriat, Donald L. Gaitros, David Garnick, Judith L. Gersting, Sakti P. Ghosh, Ratan K. Guha, Stephen T. Hedetniemi, Thomas T. Hewett, Jane Hill, Stuart Hirshfield, James A. Howard, John Impagliazzo, Greg Jones, Charles Kelemen, Willis King, Robert L. Kruse, Yedidyah Langsam, Eugene Lawler, Burt Leavenworth, R. Rainey Little, Dennis Martin, Fred J. Maryanski, Jeffrey J. McConnell, Daniel D. McCracken, Catherine W. McDonald, John McPherson, David G. Meyer, Victor Nelson, Chris Nevison, Robert Noonan, Jeffrey Parker, Margaret Peterson, Paul Purdom, Arthur Riehl, David C. Rine, Rockford J. Ross, Richard Salter, G. Michael Schneider, Greg Scragg, Mary Shaw, Daniel P. Siewiorek, David L. Soldan, Harry W. Tyrer, Annelieses von Mayrhauser, Z.G. Vranesic, Henry Walker, F. Garnett Walters, John Werth, Terry Winograd, Charles W. Winton, Charles T. Wright, Jr., and Grace Chi-Dak N. Yeung.

It should be noted that not all of the reviewers agree with all of the recommendations in this article or the full report. However, all comments were carefully considered by the Task Force throughout this report's development. Additional thanks are due to Kathleen A. Heaphy for proofreading and editorial comments on various drafts of the report itself. ◻

References

1. Accreditation Board for Engineering and Technology, Inc. Criteria for Accrediting Programs in Engineering in the United States. Dec. 1988.
2. ACM Curriculum Committee on Computer Science. Curriculum 68: Recommendations for the undergraduate program in computer science. *Commun. ACM 11,* 3 (Mar. 1968), 151–197.
3. ACM Curriculum Committee on Computer Science. Curriculum 78: Recommendations for the undergraduate program in computer science. *Commun. ACM 22,* 3 (Mar. 1979), 147–166.
4. ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991.* Feb. 1991.
5. Computing Sciences Accreditation Board. Criteria for Accrediting Programs in Computer Science in the United States. Jan. 1987.
6. Denning, P.J., Comer, D.E., Gries, D., Mulder, M.C., Tucker, A.B., Turner, A.J., and Young, P.R. Computing as a discipline. *Commun. ACM 32,* 1 (Jan. 1989), 9–23.
7. Educational Activities Board. The 1983 Model Program in Computer Science and Engineering. No. 932. Dec. 1983.
8. Gibbs, N.E., and Tucker, A.B. Model curriculum for a liberal arts degree in computer science. *Commun. ACM 29,* 3 (Mar. 1986), 202–210.
9. Koffman, E.P., Miller, P.L., and Wardle, C.E. Recommended curriculum for CS1, 1984: A report of the ACM curriculum task force for CS1. *Commun. ACM 27,* 10 (Oct. 1984), 998–1001.
10. Koffman, E.P., Stemple, D., and Wardle, C.E. Recommended curriculum for CS2, 1984: A report of the ACM curriculum task force for CS2. *Commun. ACM 28,* 8 (Aug. 1985), 815–818.
11. Mulder, M.C., and Dalphin, J. Computer science program requirements and accreditation—an interim report of the ACM/IEEE computer society joint task force. *Commun. ACM 27,* 4 (Apr. 1984), 330–335.