Computer Science

This chapter contains the report of the Subpanel on Computer Science of the CUPM Panel on a General Mathematical Sciences Program, reprinted with minor changes from Chapter IV of the 1981 CUPM report entitled RECOMMENDATIONS FOR A GENERAL MATHEMATICAL SCIENCES PROGRAM.

A Growing Discipline

Computer Science is a new and rapidly growing scientific discipline. It is distinct from Mathematics and Electrical Engineering. The subject was once closely identified in mathematicians' minds with writing computer programs. In the beginning, however, computer scientists concentrated on the discipline's mathematical theories of numerical analysis, automata, and recursive functions, as well as on programming. In the past decade, theories developed to understand problems in software design (compilers, operation systems, structured programs, etc.) have blossomed. These theories involve the analysis of complex finite structures, and in this sense have a strong mathematical bond with the finite structures common in operations research and diverse areas of applied mathematics.

More importantly, these computer science theories are needed by analysts who design algorithms for complex problems in the mathematical sciences. For this reason, all mathematical sciences students must be given an introduction to the basic concepts of computer science. Further, facility in computer programming is required of all mathematical sciences students so that they can perform practical computations in mathematical sciences courses and in subsequent mathematical sciences careers.

Although only one-third of the country's colleges and universities now have computer science departments, the number of students currently majoring in computer science taught in a computer science department (approximately 50,000 students) is greater than the number of all majors in mathematics, mathematical sciences, and applied mathematics. The computer science recommendations in this chapter are designed for institutions where computer science is taught in a mathematical sciences department or in a mathematics department. When a separate computer science department exists, that department's diversity of computer science offerings will enhance a mathematical sciences major. A mathematical sciences undergraduate program and a computer science undergraduate program should complement one another to the advantage of both departments and their students (for example, see the description of the interaction at Potsdam State in Chapter I, "A General Mathematical Science Program").

Introductory Courses

The foundation for a computer science component in a mathematics department is a one-year introductory sequence. Courses CS1 and CS2, proposed in the Association of Computing Machinery Curriculum 78 (see last section of this chapter), are excellent models for this year sequence. The Subpanel on Computer Science endorses the objectives of these two courses, and recommends that all mathematical sciences majors should be required to take the first course and strongly encouraged to take the second course in this sequence. If the second course is not required, substantial use of computers should be an integral part of other mathematical sciences courses.

The primary emphasis in the first course should be on:

- Problem solving methods and algorithmic design and analysis,
- Implementing problem solutions in a widely used higher-level programming language,
- Techniques of good programming style, and
- Proper documentation.

Lectures should include brief surveys of the history of computing, hardware and architecture, and operating systems.

The second course should include at least one major project. The course should cover topics such as recursive programming, pointers, stacks, queues, linked lists, string processing, searching and sorting techniques. The concepts of data abstraction and algorithmic complexity should be introduced. Proofs of correctness may also be discussed.

Good design and style in programming should be emphasized throughout both courses: the use of identifiers to indicate scope, modularity, appropriate choice of identifiers, good error recovery procedures, checks for integrity of input, and appropriate commentary and

documentation. Of course, efficient algorithms and coding should also be stressed. There is a strong tendency among students to worry only about whether their programs run correctly. Through class lectures and careful grading of programming assignments, the instructor must teach the students the importance of good design, style, and efficiency in programming.

A source of useful commentary about introductory computer science courses is the *SIGCSE (Special Interest Group on Computer Science Education) Bulletin.* The bulletin is published quarterly, and issue #1 each year, which contains papers presented at the SIGCSE annual meeting, is especially valuable.

Most introductory texts have many sample projects. In addition, the following three texts are good general sources of computer projects.

1. Bennett, William R., *Scientific and Engineering Problem-Solving with the Computer,* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
2. Gruenberger, Fred and Jaffray, G., *Problems for Computer Solution,* John Wiley & Sons, New York, 1965.
3. Wetherall, Charles, *Etudes for Programmers,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Mathematicians teaching introductory computer science often emphasize numerical computation in programming assignments. At the introductory level, the computer science issues involved in numerical computation are quite simple. Assignments requiring symbolic manipulation and data organization present more substantive programming problems and, in general, require more thought. The following is a sample assignment that could be given late in the first course:

> Write a program which obtains a five-card poker hand from some source (terminal, input deck, or file), prints the hand in a reasonably well-formatted style, and determines whether or not the hand contains a pair, three of a kind, a straight, a full house, etc.

Intermediate Courses

Intermediate-level computer science courses building on CS1 and CS2 should address basic underlying issues in computer science. In describing computer science in the first two years, the ACM Curriculum 78 report states that the student should be given "a thorough grounding in the implementation of algorithms in programming languages which operate on data structures in the environment of hardware." Thus these courses should develop general topics about algorithms, concepts in programming languages, data structures, and computer hardware.

The intermediate-level courses should be taught by a computer scientist, that is, by an individual who has significant graduate-level training in computer science (see below).

The Subpanel on Computer Science, in concurrence with ACM curriculum groups, strongly rejects the idea of a set of courses that each address a specific programming language, e.g., a sequence of advanced FORTRAN, COBOL, RPG, and APL. The argument for such a sequence is usually based on the employability of students completing it. If indeed this argument is valid, and there is some question about that, it is a short range benefit. Students completing such a sequence will soon find that the lack of underlying concepts will put them at a severe disadvantage. However, it may be acceptable, resources permitting, to have one "vocational" elective course that studies a second higher-level language such as COBOL. Of course, it is also natural to discuss new programming languages in several intermediate (and advanced) computer science courses. However, the new language would not be the focus of the course, but rather a tool used in learning and illustrating fundamental concepts.

The role of numerical and computational mathematics in computer science has diminished in recent years. While the ACM Curriculum 68 treated numerical analysis as part of core computer science, today numerical mathematics is considered by most computer scientists to be simply another mathematical sciences field that has overlap with computer science. Numerical mathematics is very important in a mathematical sciences major, but it is not a part of the computer science component.

Following the CS1 and CS2 courses, the ACM Curriculum 78 specifies six additional courses in core computer science.

CS3 Introduction to Computer Systems
CS4 Introduction to Computer Organization
CS5 Introduction to File Processing
CS6 Operating Systems and Computer Architecture
CS7 Data Structures and Algorithm Analysis
CS8 Organization of Programming Languages

The syllabi of these courses are given at the end of this chapter. Ideally, all six of these courses would be offered. A concentration or a minor in computer science would commonly consist of CS1 and CS2, followed by two of CS3, CS4, and CS5, and two of CS6, CS7, and CS8. For the purposes of a mathematical sciences program, it may be justified to place more emphasis on the software oriented areas. This would imply, if there

was difficulty in offering all six courses, that CS3, CS5, CS7, and CS8 would be most useful. Then CS3, CS5, CS7, and CS8 would be offered once a year, and CS4 and CS6 offered as topics courses every other year.

At many schools, it may not be feasible to offer at least four of these intermediate courses in computer science on a regular basis. Then one can combine parts of these intermediate courses to provide a significant offering in two courses above CS1 and CS2. In this case, only two computer science courses, one elementary and one intermediate, would be offered each semester. One approach would be to combine topics from CS5 and CS7 into one course, and topics from CS3, CS4, and CS6 into the other. This would yield two courses with the following sort of syllabi (for more details about these topics, see the ACM Curriculum 78 syllabi at the end of this chapter):

A1. Algorithms for Data Manipulation
   1. Algorithm design and development illustrated in areas of sorting and research (25%)
   2. Data structure implementation (30%)
   3. Access methods (25%)
   4. Systems design (15%)
   5. Exams (5%)

A2. Computer Structures
   1. Basic logic design (15%)
   2. Number representation and arithmetic (10%)
   3. Assembly systems (35%)
   4. Program segmentation and linkage (15%)
   5. Memory management (10%)
   6. Computer systems structure (10%)
   7. Exams (5%)

This approach focuses on data structures and software issues that relate to operating systems. An alternative approach could concentrate on programming languages and algorithms involved in computer systems performance. This theme could be realized by combining topics in CS3, CS5, and CS8 into one course, and topics in CS4, CS6, and CS7 into the other course. This would yield two courses with the following syllabi:

B1. Language Types and Structures
   1. Assembly systems (25%)
   2. Program segmentation and linkage (15%)
   3. Language definition structure (10%)
   4. Data types and structures (15%)
   5. Control structures and data flow (20%)
   6. Access methods (10%)
   7. Exams (5%)

B2. Algorithms for Computer Systems
   1. Basic logic design (20%)
   2. Algorithm design and analysis (20%)
   3. Procedure activation algorithms (15%)
   4. Memory management (15%)
   5. Process management (15%)
   6. Systems design (10%)
   7. Exams (5%)

It is important to note that an individual wishing to go on from these courses to advanced work in computer science may have to make up, as deficiencies, areas in core computer science that are not represented in these condensed pairs of courses.

Concentrations and Minors

A computer science concentration in a college mathematics department can be defined as an option within a mathematical sciences major or as a "stand-alone" minor. A computer science minor should consist of about six courses, ACM Curriculum 78 courses CS1 and CS2 plus four intermediate courses.

A computer science concentration within a mathematical sciences major has three components:

A. Mathematics: 5-plus courses;
B. Computer Science: 4-6 courses;
C. Applied Mathematics: 3-plus courses.

A. The mathematics component would include the three semester freshman-sophomore "calculus sequence" plus linear algebra. As recommended in Chapter I, "A General Mathematical Sciences Program," any mathematical sciences major should contain upper-level course work of a theoretical nature, typically algebra or advanced calculus. In a major with a computer science concentration, algebra is the natural area. Specifically, the applied algebra course given in Chaper I would be excellent for the computer science concentration. The course's syllabus incorporates most of the topics of the ACM 78 discrete mathematics course (required of computer science majors). A small department could offer applied algebra and standard abstract algebra courses in alternate years. Logic and automata theory are attractive electives in the mathematics component if a mathematics department wishes to focus on more theoretical aspects of computer science.

It should be noted that several computer science educators have questioned the reliance on calculus as the basic mathematics for future computer scientists; ACM Curriculum 78, for instance, requires a (freshman) year of calculus. They advocate a mathematics component based on discrete mathematics with only one semester of calculus (taught, say, in the junior year). See A. Ralston and M. Shaw, "Curriculum 78—Is Computer Science Really that Unmathematical?", *Communications ACM* 23 (1980), pp. 67-70.

B. The computer science component would include ACM Curriculum 78 courses CS1 and CS2 plus two to four intermediate courses, as described in the preceding section. The syllabi of ACM Curriculum 78 core courses are given at the end of this chapter.

C. The applied mathematics component should include a course in numerical analysis and a course in probability and statistics. The third applied mathematics course would be discrete methods, which would cover the combinatorial material in the ACM Curriculum 78 discrete mathematics course in greater depth, including operations-research-related graph modeling (see Chapter I for a full description of this course). The CUPM Mathematical Sciences Program panel recommends that all mathematics departments should offer a discrete methods course. Other good courses for the applied mathematics component are ordinarily differential equations, mathematical modeling, and operations research. The 1971 CUPM *Report on Computational Mathematics* describes courses in computational models, in combinatorial computation, and in differential equations with numerical methods; these courses combine topics from a variety of mathematical sciences and computer science courses and hence are particularly attractive to small departments.

In either the computer science concentration or minor, all six computer science courses are needed for future graduate study in computer science. Incoming graduate students with less preparation are commonly required to make up undergraduate course deficiencies.

Faculty Training

For the foreseeable future, the dominant factor affecting computer science instruction at all institutions, but particularly at smaller colleges and universities, will be the extreme shortage of qualified computer scientists in academe. At smaller colleges and universities it may therefore be effectively impossible to hire a computer scientist to teach core computer science courses. Among the possible solutions to this problem are:

1. Using adjunct faculty to teach computer science courses.
2. Using existing (non-computer science) faculty to teach computer science courses.

The first solution is acceptable for some courses. Although one cannot build a program with adjunct faculty and although staffing courses with adjunct faculty is never as desirable as using full-time faculty (e.g., student advising is a particular problem), this is a feasible way to get computer science courses taught when such faculty exist in the local community. However, since so many smaller colleges are located away from the metropolitan areas where most technical and scientific employers of such adjunct faculty are found, this solution will not be useful to most smaller institutions.

A crucial point that must be emphasized when using existing non-computer science faculty (i.e., mathematicians) to teach computer science courses is that computer science cannot be treated like most other new mathematics course topics which mathematicians will (quickly) learn as they teach it. Mathematicians untrained in computer science are very likely to teach computer science badly, hurting both the students and the mathematics department's reputation. Therefore, if a current mathematics faculty member is to be used to teach computer science, especially beyond the first course, he or she must first acquire some formal education in computer science.

The most plausible approach to such computer science training is through some program of released time. The pertinent questions about the training are: how long? where? and how financed?

Assuming that the mathematician who is to be trained is, at most, familiar with programming in a high-level language, then full-time study for one year is the minimum period needed to acquire the background, knowledge, and experience necessary to teach several of the intermediate-level core computer science courses. Since one year is also the maximum period which would be administratively or financially feasible, this should be viewed as the canonical period for faculty training in computer science. Part-time study over a longer period or a succession of summers can also be considered. However, both because the needs to train faculty in computer science are pressing and because intermittent study is almost always less effective than continuous study, at least one faculty member in a mathematics department should have completed a one-year program of full-time study in computer science.

The most logical place at which to study computer science for the purpose of becoming able to teach it is at a university with undergraduate and graduate (preferably Ph.D.) programs in computer science. Although there are exceptions, the current level of computer science instruction in American colleges and universities is so uneven that only at such institutions can one be reasonably assured of an atmosphere in which there will be the necessary broad understanding of the principles of computer science. Such an atmosphere is particularly important for an academic mathematician preparing to teach the subject.

Another possibility which should be mentioned is for the faculty member to spend one year at one of those

(relatively few) major industrial firms with good in-house training programs in computer science. An additional attraction to this idea is that it might be possible to arrange an exchange in which a member of the firm taught at the college for a year.

Methods of financing such a program of faculty training in computer science are fairly obvious:

1. Through released time at full pay from the mathematician's home institution.
2. Through grants from current, and hopefully new, federal programs; officials of both the MAA and ACM are currently pressing NSF to provide more funds for this purpose.
3. Through grants from private foundations; individual institutions and departments may be more effective than professional associations in obtaining such private funds.
4. Through corporate sponsorship of participation in in-house training programs or academic-corporate exchanges.

Computer Facilities

Facilities to support computing in mathematical sciences instruction can be provided in a variety of ways, ranging from one large centrally administered system to many small personal computing devices. The suitability of a particular means depends not only upon its intended applications, but also upon factors such as cost, ease of use, and local politics. At present, computing services in most colleges and universities are provided by a large centralized facility, the Computing Center. Growing numbers of institutions, however, are beginning to decentralize computing on campus. Three current modes of providing service are discussed below:

- Centralized facilities
- Departmental computers
- Personal computers.

There is a fourth mode that is primarily a form of access to centralized or departmental computers:

- Terminals

The second half of this section discusses the cost and ease of implementation of various applications with different types of computing facilities.

It should be noted that it is possible for an institution to form a consortium with nearby schools to operate a common central computing facility or to buy time (and services) from commercial computing centers. This option allows an institution to have a mix of computing, using large computers for problems requiring great speed or memory size, such as "number crunching," and smaller computers for student programs and other instructional purposes.

CENTRALIZED FACILITIES

Historically, so-called "economies of scale" encouraged the development of increasingly larger computers; and of increasingly larger organizations to administer them. Such computer systems are capable of providing a great variety of services with a low cost for each service. In addition, the organizations which administer these systems can play an important role in developing and supporting instructional uses of computing on campus.

On the other hand, the very size of such facilities and the organizations that administer them create certain problems. First, large systems have a high unit cost, in the range of half a million to several million dollars; replacing or enhancing such a system involves a major administrative decision. Second, instructional users of such systems must often compete with other powerful and better-financed constituencies; either separate facilities are needed to reduce competition among instructional, research, and administrative uses of the computer, or policies are needed to allocate the services provided by a single facility. And third, large organizations can be bureaucratic and inflexible.

DEPARTMENTAL COMPUTERS

For the last ten years minicomputers have provided an alternative to a large centralized facility. Lower unit costs (around $100,000 or less) and the possibility of local control have made it attractive for academic and administrative departments to acquire facilities of their own. Such facilities can be tailored to a department's needs and can provide almost as many services as a large centralized system.

Minicomputers, however, are not necessarily the answer to every department's computing needs. First, there is the question of which services they will provide. Second, there are hidden costs associated with administering any computer facility: personnel are needed to operate and maintain the facility and to provide technical assistance to users. Small departments run the risk of diverting attention from their primary task of teaching mathematics to the subsidiary task of managing such an enterprise. One way to deal with such hidden costs is for departments to contract with a central campus organization to manage their facilities. Third, there are inconveniences for students faced with using, and first learning to use, several different departmental systems. Of course, this difficulty can be overcome by

requiring departments to purchase compatible systems and by interconnecting all systems.

Many academic computing specialists expect interconnected departmental computers to become the dominant means of academic computing in the next decade.

PERSONAL COMPUTERS

The recent development of personal microcomputers provides another alternative for instructional computing. Very low unit costs (one or two thousand dollars) make computing possible for departments otherwise unable to afford or gain authorization for large facilities. Microcomputer facilities suffer from many of the same problems as minicomputer facilities. In addition, microcomputers are limited in the services they provide, are slower than their large competitors, and may not be designed for rugged use by large groups of students. Still they can prove quite adequate for elementary applications. Further, by being less intimidating and more exciting than larger computers, they can play a role in overcoming a student's "computer anxiety."

TERMINALS

Terminals are used for remote, interactive access to large computers. Some have small memories and primitive editing capabilities. Departments often have a greater choice in selecting terminals to connect to computer systems than they do in selecting the systems themselves. Cost, speed, and durability are primary factors influencing the selection of a terminal. By these criteria, video terminals are preferable. The availability of graphical output and local editing features are other factors to consider when choosing terminals. Hard-copy (printing) terminals are more expensive and tend to be slower than video terminals, but they do provide users with a permanent record of their work, and so some printing terminals are necessary (medium or high speed printers can be used in conjunction with video terminals to provide this record). Video terminals may also be used in conjunction with television monitors to provide classroom displays of computer output. For such output to be visible in a large classroom, either many monitors must be provided or the video terminals employed must use larger, and hence fewer, characters in their display.

Applications

The suitability of a particular computing facility depends most upon its intended applications. The rest of this section discusses the most common academic uses of computers and how well different types of computing facilities serve these uses.

INTRODUCTORY PROGRAMMING

Any of the three types of facilities can serve as a vehicle for teaching beginners to program and for introducing computational examples into elementary mathematics courses. Such uses typically involve large numbers of students writing relatively simple programs. Larger facilities tend to provide a greater choice of programming languages, although modern languages such as PASCAL and PL/I are becoming increasingly available even on microcomputers. Larger machines tend to be faster also; even though use of such machines is shared, students will find that they process simple programs much faster than microcomputers. Costs, however, tend to be roughly equal for simple interactive computing on the three types of facilities—around $2.00 per hour. These costs can be reduced significantly by using larger machines in a noninteractive, batch-processing mode. This mode of use, while predominant in the past, is becoming less popular as minicomputers and microcomputers make a more responsive computing environment available and affordable.

ADVANCED PROGRAMMING

Advanced programming is more distinguished from introductory programming in its requirements for more sophisticated languages and for facilities to handle large programs. Microcomputers at present do not meet these requirements; the languages they provide are quite restrictive, and large programs exceed their capacity. Execution times and costs for large programs tend to be lowest on large machines under batch processing, but minicomputers are becoming competitive both in price and speed.

PROGRAM DEVELOPMENT AND MAINTENANCE

Program development is influenced heavily by the computing environment in which it occurs. Convenient interactive editing capabilities accelerate the task of writing and correcting a program; microcomputers, with almost instantaneous response, do a particularly good job of editing. Facilities for file storage enable program development to be spread over several sessions. Large machines provide less expensive storage and much faster retrieval of information; they also facilitate sharing programs among users and provide centralized backup. Microcomputer facilities can distribute the costs of file storage by requiring users to purchase individual floppy disks, but unless a centralized store is provided through a network, sharing information can be difficult.

GRAPHICS

One of the primary attractions of personal microcom-

puters is their ability to generate graphic displays and to enable users to interact with these displays. Larger systems, unless specifically tailored to graphic applications, tend to have primitive graphic facilities at best.

APPLICATION PACKAGES

Application packages available for various machines provide aids for numerical and symbolic computations. Typical areas of application include statistics, linear programming, numerical solution of differential equations, and algebraic formula manipulation. Such packages are more widely available on larger machines. Large computations often require an unacceptably long time on microcomputers (several hours) and may exceed the memory size of small computers.

MISCELLANEOUS APPLICATIONS

Word processing systems facilitate production of course notes, research papers, and term papers. If good word processing facilities are available, they are likely to quickly generate heavy faculty use. Simple word processing software is available for personal computers, but a minicomputer (or powerful $5,000-plus microcomputer) is needed for good mathematically-oriented word processing software, such as the UNIX system. Large computers often have poor word processing capabilities.

Data base systems are of more use in the social sciences than in the mathematical sciences, but can be used to provide real data for analysis in statistics courses. Such systems require a centralized file store on a larger computer.

Real-time data acquisition is of interest in the natural sciences. They can also be used to provide real data for mathematical analysis. Dedicated microcomputers are better suited to laboratory instrumentation than are shared machines.

ACM Curriculum 78

The following computer science course syllabi are reproduced from the ACM Curriculum 78 Report in *Communications of ACM,* March 1979, pp. 147-166. (Copyright 1979, Association for Computing Machinery, Inc.) They provide eight core courses for a computer science major.

CS1. Computer Programming I

OBJECTIVES:

- To introduce problem solving methods and algorithm development;
- To teach a high-level programming language that is widely used; and
- To teach how to design, code, debug, and document programs using techniques of good programming style.

COURSE OUTLINE:

The material on a high-level programming language and on algorithm development can be taught best as an integrated whole. Thus the topics should not be covered sequentially. The emphasis of the course is on the techniques of algorithm development and programming with style. Neither esoteric features of a programming language nor other aspects of computers should be allowed to interfere with that purpose.

TOPICS:

A. *Computer Organization.* An overview identifying components and their functions, machine and assembly languages. (5%)
B. *Programming Language and Programming.* Representation of integers, real, characters, instructions. Data types, constants, variables. Arithmetic expression. Assignment statement. Logical expression. Sequencing, alternation, and iteration. Arrays. Subprograms and parameters. Simple I/O. Programming projects utilizing concepts and emphasizing good programming style. (45%)
C. *Algorithm Development.* Techniques of problem solving. Flowcharting. Stepwise refinement. Simple numerical examples. Algorithms for searching (e.g., linear, binary), sorting (e.g., exchange, insertion), merging of ordered lists. Examples taken from such areas as business applications involving data manipulation, and simulations involving games. (45%)
D. *Examinations.* (5%)

CS2. Computer Programming II

OBJECTIVES:

- To continue the development of discipline in program design, in style and expression, in debugging and testing, especially for larger programs;
- To introduce algorithmic analysis; and
- To introduce basic aspects of string processing, recursion, internal search/sort methods and simple data structures.

PREREQUISITE: CS 1.

COURSE OUTLINE:

The topics in this outline should be introduced as needed in the context of one or more projects involving larger programs. The instructor may choose to begin with the statement of a sizable project, then utilize

structured programming techniques to develop a number of small projects each of which involves string processing, recursion, searching and sorting, or data structures. The emphasis on good programming style, expression, and documentation, begun in CS1, should be continued. In order to do this effectively, it may be necessary to introduce a second language (especially if a language like Fortran is used in CS1). In that case, details of the language should be included in the outline. Analysis of algorithms should be introduced, but at this level such analysis should be given by the instructor to the student.

Consideration should be given to the implementation of programming projects by organizing students into programming teams. This technique is essential in advanced level courses and should be attempted as early as possible in the curriculum. If large class size makes such an approach impractical, every effort should be made to have each student's programs read and critiqued by another student.

TOPICS:

A. *Review.* Principles of good programming style, expression, and documentation. Details of a second language if appropriate. (15%)
B. *Structured Programming Concepts.* Control flow. Invariant relation of a loop. Stepwise refinement of both statements and data structures, or top-down programming. (40%)
C. *Debugging and Testing.* (10%)
D. *String Processing.* Concatenation. Substrings. Matching. (5%)
E. *Internal Searching and Sorting.* Methods such as binary, radix, Shell, quicksort, merge sort. Hash coding. (10%)
F. *Data Structures.* Linear allocation (e.g., stacks, queues, deques) and linked allocation (e.g., simple linked lists). (10%)
G. *Recursion.* (5%)
H. *Examinations.* (5%)

CS3. Introduction to Computer Systems

OBJECTIVES:

- To provide basic concepts of computer systems;
- To introduce computer architecture; and
- To teach an assembly language.

PREREQUISITE: CS 2.

COURSE OUTLINE:

The extent to which each topic is discussed and the ordering of topics depends on the facilities available and the nature and orientation of CS4 described below. Enough assembly language details should be covered and projects assigned so that the student gains experience in programming a specific computer. However, concepts and techniques that apply to a broad range of computers should be emphasized. Programming methods that are developed in CS1 and CS2 should also be utilized in this course.

TOPICS:

A. *Computer Structure and Machine Language.* Memory, control, processing and I/O units. Registers, principal machine instruction types and their formats. Character representation. Program control. Fetch-execute cycle. Timing. I/O Operations. (15%)
B. *Assembly Language.* Mnemonic operations. Symbolic addresses. Assembler concepts and instruction format. Data-word definition. Literals. Location counter. Error flags and messages. Implementation of high-level language constructs. (30%)
C. *Addressing Techniques.* Indexing. Indirect Addressing. Absolute and relative addressing. (5%)
D. *Macros.* Definition. Call. Parameters. Expansion. Nesting. Conditional assembly. (10%)
E. *File I/O.* Basic physical characteristics of I/O and auxiliary storage devices. File control system. I/O specification statements and device handlers. Data handling, including buffering and blocking. (5%)
F. *Program Segmentation and Linkage.* Subroutines. Coroutines. Recursive and re-entrant routines. (20%)
G. *Assembler Construction.* One-pass and two-pass assemblers. Relocation. Relocatable loaders. (5%)
H. *Interpretive Routines.* Simulators. Trace. (5%)
I. *Examinations.* (5%)

CS4. Introduction to Computer Organization

OBJECTIVES:

- To introduce the organization and structuring of the major hardware components of computers;
- To understand the mechanics of information transfer and control within a digital computer system; and
- To provide the fundamentals of logic design.

PREREQUISITE: CS 2.

COURSE OUTLINE:

The three main categories in the outline, namely computer architecture, arithmetic, and basic logic design, should be interwoven throughout the course rather

than taught sequentially. The first two of these areas may be covered, at least in part, in CS3 and the amount of material included in this course will depend on how the topics are divided between the two courses. The logic design part of the outline is specific and essential to this course. The functional, logic design level is emphasized rather than circuit details which are more appropriate in engineering curricula. The functional level provides the student with an understanding of the mechanics of information transfer and control within the computer system. Although much of the course material can and should be presented in a form that is independent of any particular technology, it is recommended that an actual simple minicomputer or microcomputer system be studied. A supplemental laboratory is appropriate for that purpose.

TOPICS:

A. *Basic Logic Design.* Representation of both data and control information by digital (binary) signals. Logic properties of elemental devices for processing (gates) and storing (flipflops) information. Description by truth tables, Boolean functions and timing diagrams. Analysis and synthesis of combinatorial networks of commonly used gate types. Parallel and serial registers. Analysis and synthesis of simple synchronous control mechanisms; data and address buses; addressing and accessing methods; memory segmentation. Practical methods of timing pulse generation. (25%)

B. *Coding.* Commonly used codes (e.g., BCD, ASCII). Parity generation and detection. Encoders, decoders, code converters. (5%)

C. *Number Representation and Arithmetic.* Binary number representation, unsigned addition and subtraction. One's and two's complement, signed magnitude and excess radix number representations and their pros and cons for implementing elementary arithmetic for BCD and excess-3 representations. (10%)

D. *Computer Architecture.* Functions of, and communication between, large-scale components of a computer system. Hardware implementation and sequencing of instruction fetch, address construction, and instruction execution. Data flow and control block diagrams of a simple processor. Concept of microprogram and analogy with software. Properties of simple I/O devices and their controllers, synchronous control, interrupts. Modes of communications with processors. (35%)

E. *Example.* Study of an actual, simple minicomputer or microcomputer system. (20%)

F. *Examinations.* (5%)

CS5. Introduction to File Processing

OBJECTIVES:

- To introduce concepts and techniques of structuring data on bulk storage devices;
- To provide experience in the use of bulk storage devices; and
- To provide the foundation for applications of data structures and file processing techniques.

PREREQUISITE: CS 2.

COURSE OUTLINE:

The emphasis given to topics in this outline will vary depending on the computer facilities available to students. Programming projects should be assigned to give students experience in file processing. Characteristics and utilization of a variety of storage devices should be covered even though some of the devices are not part of the computer system that is used. Algorithmic analysis and programming techniques developed in CS2 should be utilized.

TOPICS:

A. *File Processing Environment.* Definitions of record, file, blocking, compaction, database. Overview of database management system. (5%)

B. *Sequential Access.* Physical characteristics of sequential media (tape, cards, etc.). External sort/merge algorithms. File manipulation techniques for updating, deleting and inserting records in sequential files. (30%)

C. *Data Structures.* Algorithms for manipulating linked lists. Binary, B-trees, B^*-trees, and AVL trees. Algorithms for transversing and balancing trees. Basic concepts of networks (plex structures). (20%)

D. *Random Access.* Physical characteristics of disk, drum, and other bulk storage devices. Algorithms and techniques for implementing inverted lists, multilist, indexed sequential, and hierarchical structures. (35%)

E. *File I/O.* File control systems and utility routines, I/O specification statements for allocating space and cataloging files. (5%)

F. *Examinations.* (5%)

CS6. Operating Systems & Comp. Architecture

OBJECTIVES:

- To develop an understanding of the organization and architecture of computer systems at the

register-transfer and programming levels of system description;
- To introduce the major concept areas of operating systems principles;
- To teach the inter-relationships between the operating system and the architecture of computer systems.

PREREQUISITES: CS3 AND CS4.

COURSE OUTLINE:

This course should emphasize concepts rather than case studies. Subtleties do exist, however, in operating systems that do not readily follow from concepts alone. It is recommended that a laboratory requiring hands-on experience be included with this course.

The laboratory for the course would ideally use a small computer where students could actually implement sections of operating systems and have them fail without serious consequences to other users. This system should have, at a minimum, a CPU, memory, disk or tape, and some terminal device such as a teletype of CRT. The second best choice for the laboratory experience would be a simulated system running on a larger machine.

The course material should be liberally sprinkled with examples of operating system segments implemented on particular computer system architectures. The interdependence of operating systems and architecture should be clearly delineated. Integrating these subjects at an early stage in the curriculum is particularly important because the effects of computer architecture on systems software has long been recognized. Also, modern systems combine the design of operating systems and the architecture.

TOPICS:

A. *Review.* Instruction sets. I/O and interrupt structure. Addressing schemes. Microprogramming. (10%)
B. *Dynamic Procedure Activation.* Procedure activation and deactivation on a stack, including dynamic storage allocation, passing value and reference parameters, establishing new local environments, addressing mechanics for accessing parameters (e.g., displays, relative addressing in the stack). Implementing non-local references. Re-entrant programs. Implementation on register machines. (15%)
C. *System Structure.* Design methodologies such as level, abstract data types, monitors, kernels, nuclei, networks of operating system modules. Proving correctness. (10%)
D. *Evaluation.* Elementary queueing, network models of systems, bottlenecks, program behavior, and statistical analysis. (15%)
E. *Memory Management.* Characteristics of the hierarchy of storage media, virtual memory, paging, segmentation. Policies and mechanisms for efficiency of mapping operations and storage utilization. Memory protection. Multiprogramming. Problems of auxiliary memory. (20%)
F. *Process Management.* Asynchronous processes. Using interrupt hardware to trigger software procedure calls. Process stateword and automatic SWITCH instructions. Semaphores. Ready lists. Implementing a simple scheduler. Examples of process control problems such as deadlock, product/consumers, readers/writers. (20%)
G. *Recovery Procedures.* Techniques of automatic and manual recovery in the event of system failures. (5%)
H. *Examinations.* (5%)

CS7. Data Structures and Algorithm Analysis

OBJECTIVES:

- To apply analysis and design techniques to non-numeric algorithms which act on data structures;
- To utilize algorithmic analysis and design criteria in the selection of methods for data manipulation in the environment of a database management system.

PREREQUSITES: CS5.

COURSE OUTLINE:

The material in this outline could be covered sequentially in a course. It is designed to build on the foundation established in the elementary material, particularly on that material which involves algorithm development and data structures and file processing. The practical approach in the earlier material should be made more rigorous in this course through the use of techniques for the analysis and design of efficient algorithms. The results of this more formal study should then be incorporated into data management system design decisions. This involves differentiating between theoretical or experimental results for individual methods and the results which might actually be achieved in systems which integrate a variety of methods and data structures. Thus, database management systems provide the applications environment for topics discussed in the course.

Projects and assignments should involve implementation of theoretical results. This suggests an alternative way of covering the material in the course; namely,

to treat concepts, algorithms, and analysis in class and deal with their impact on system design in assignments. Of course, some in-class discussions of this impact would occur, but at various times throughout the course rather than concentrated at the end.

TOPICS:

A. *Review.* Basic data structures such as stacks, queues, lists, trees. Algorithms for their implementation. (10%)
B. *Graphs.* Definition, terminology, and property (e.g., connectivity). Algorithms for finding paths and spanning trees. (15%)
C. *Algorithms Design and Analysis.* Basic techniques of design and analysis of efficient algorithms for internal and external sorting/merging/searching. Intuitive notions of complexity (e.g., NP-hard problems). (30%)
D. *Memory Management.* Hashing. Algorithms for dynamic storage allocation (e.g., buddy system, boundary-tag), garbage collection and compaction. (15%)
E. *System Design.* Integration of data structures, sort/merge/search methods (internal and external) and memory media into a simple database management system. Accessing methods. Effects on run time, costs, efficiency. (25%)
F. *Examinations.* (5%)

CS8. Organization of Programming Languages

OBJECTIVES:

- To develop an understanding of the organization of programming languages, especially the run-time behavior of programs;
- To introduce the formal study of programming language specification and analysis;
- To continue the development of problem solution and programming skills introduced in the elementary level material.

PREREQUSITES: CS2; RECOMMENDED: CS3, CS5.

COURSE OUTLINE:

This is an applied course in programming language constructs emphasizing the run-time behavior of programs. It should provide appropriate background for advanced level courses involving formal and theoretical aspects of programming languages and/or the compilation process.

The material in this outline is not intended to be covered sequentially. Instead, programming languages could be specified and analyzed one at a time in terms of their features and limitations based on their run-time environments. Alternatively, desirable specification of programming languages could be discussed and then exemplified by citing their implementations in various languages. In either case, programming exercises in each language should be assigned to emphasize the implementations of language features.

TOPICS:

A. *Language Definition Structure.* Formal language concepts including syntax and basic characteristics of grammars, especially finite state, context-free, and ambiguous. Backus-Naur Form. A language such as Algol as an example. (15%)
B. *Data Types and Structures.* Review of basic data types, including lists and trees. Constructs for specifying and manipulating data types. Language features affecting static and dynamic data storage management. (10%)
C. *Control Structures and Data Flow.* Programming language constructs for specifying program control and data transfer, including DO ... FOR, DO ... WHILE, REPEAT ... UNTIL, BREAK, subroutines, procedures, block structures, and interrupts. Decision tables, recursion. Relationship with good programming style should be emphasized. (15%)
D. *Run-time Consideration.* The effects of run-time environment and binding time on various features of programming languages. (25%)
E. *Interpretative Languages.* Compilation vs. interpretation. String processing with language features such as those available in SNOBOL 4. Vector processing with language features such as those available in SPL. (20%)
F. *Lexical Analysis and Parsing.* An introduction to lexical analysis including scanning, finite state acceptors and symbol tables. An introduction to parsing and compilers including push-down acceptors, top-down and bottom-up parsing. (10%)
G. *Examinations.* (5%)

Subpanel Members

ALAN TUCKER, CHAIR, SUNY-Stony Brook.
GERALD ENGEL, Christopher Newport College.
STEPHEN GARLAND, Dartmouth College.
BERT MENDELSON, Smith College.
ANTHONY RALSTON, SUNY-Buffalo.