WHAT TO DO TILL THE COMPUTER SCIENTIST COMES

BY

G. E. FORSYTHE

Reprinted from the AMERICAN MATHEMATICAL MONTHLY
Vol. 75, No. 5, May, 1968

WHAT TO DO TILL THE COMPUTER SCIENTIST COMES*

GEORGE E. FORSYTHE, Computer Science Department, Stanford University

Computer science departments. What is computer science anyway? This is a favorite topic in computer science department meetings. Just as with definitions of mathematics, there is less than total agreement and—moreover—you must know a good deal about the subject before any definition makes sense. Perhaps the tersest answer is given by Newell, Perlis, and Simon [8]: just as zoology is the study of animals, so computer science is the study of computers. They explain that it includes the hardware, the software, and the useful algorithms computers perform. I believe they would also include the study of computers that *might* be built, given sufficient demand and sufficient development in the technology. In an earlier paper [4], the author defines computer science as the art and science of representing and processing information. Some persons [10] extend the subject to include a study of the structure of information in nature (e.g., the genetic code).

Computer scientists work in three distinguishable areas: (1) design of hardware components and especially total systems; (2) design of basic languages and software broadly useful in applications, including monitors, compilers, time-sharing systems, etc.; (3) methodology of problem solving with computers. The accent here is on the principles of problem solving—those techniques that are common to solving broad classes of problems, as opposed to the preparation of individual programs to solve single problems. Because computers are used for such a diversity of problems (see below), the methods differ widely. Being new, the subject is not well understood, and considerable energy now goes into experimental solution of individual problems, in order to acquire experience from which principles are later distilled. But in the long run the solution of problems in field X on a computer should belong to field X, and computer science should concentrate on finding and explaining the principles of problem solving.

One example of methodological research in computer science is the design and operation of "interactive systems," in which a man and a computer are appropriately coupled by keyboards and console displays (perhaps within a time-sharing system) for the solution of scientific problems.

Because of our emphasis on methodology, Professor William Miller likens the algorithmic and heuristic aspects of problem solving in computer science to the methodology of problem solving in mathematics so ably discussed by Professor Pólya in several books [9]. In computer science there is great stress on the dynamic action of computation, rather than the static presentation of logical structure. It tends to attract men of action, rather than contemplative men. Our students want to *do* something from the first day.

Computer science is at once abstract and pragmatic. The focus on actual

* Expanded version of a presentation to a panel session before the Mathematical Association of America, Toronto, 30 August 1967. The author is grateful to Professors T. E. Hull, William Miller, and Allen Newell for various ideas used in the paper.

computers introduces the pragmatic component: our central questions are economic ones like the relations among speed, accuracy, and cost of a proposed computation, and the hardware and software organization required. The (often) better understood questions of existence and theoretical computability—however fundamental—remain in the background. On the other hand, the medium of computer science—information—is an abstract one. The meaning of symbols and numbers may change from application to application, either in mathematics or in computer science. Like mathematics, one goal of computer science is to create a basic structure in terms of inherently defined concepts that is independent of any particular application.

Computer science has hardly started on the creation of such a basic structure, and in our present developmental stage computer scientists are largely concerned with exploring what computers can and cannot economically do. Let me emphasize the *variety* of fields in which computing has become an important tool. One of these is applied mathematics, as Professor Lax emphasizes, but this is merely one. Others include experimental physics, business data processing, economic planning, library work, the design of almost anything (including computers), education, inventory management, police operations, medicine, air traffic control, national population inventories, space science, musical performance, content analyses of documents, and many others. I must emphasize that the amount of computing done for applied mathematics is an almost invisible fraction of the total amount of computing today.

There is frequent discussion of whether computer science is part of mathematics—i.e., applied mathematics or "mathematical science." In a purely intellectual sense such jurisdictional questions are sterile and a waste of time. On the other hand, they have great importance within the framework of institutionalized science—e.g., the organization of universities and of the granting arms of foundations and the Federal Government.

I am told that the preponderant opinion among administrators in Washington is that computer science is part of applied mathematics. I believe the majority of university computer scientists would say it is not; cf. [8]. I would have to ask you how mathematicians feel about the matter. COSRIMS (Committee on the Support of Research in the Mathematical Sciences, appointed by the National Academy of Science—National Research Council) has taken the position that computer science is a mathematical science, but many of the discussions emphasize differences between mathematics and computer science.

In spite of the infancy of our subject, there are approximately 40 computer science departments in the United States and Canada today. There is no longer any doubt that computer science will have a separate university organization for several coming decades. I believe that the creation of these separate departments is a correct university response to the computer revolution, for I do not think computers would be well studied in an environment dominated by either mathematicians or engineers. However, finding suitable faculty members is very difficult today.

What are these computer science departments doing? Answer: Roughly the same things that mathematics departments are doing: education, research, and service. We teach computer science to three types of students: to our majors at the B.S., M.S., and Ph.D. levels, to technical students who need computing as a tool, and to any students who wish to become acquainted with computing as an important ingredient of our civilization. We do research in our several specialties: e.g., numerical analysis, programming languages and systems, heuristic methods of problem solving, graphical data representation and processing, time-sharing systems, logical design, business data processing, etc. We perform an unusually large amount of community service in helping our colleagues with their computing problems, both individually and by advising or managing the university computation center.

At Stanford University our graduate students are distributed among roughly three major areas of computer science: numerical mathematics (about 10 percent), programming languages and systems (about 50 percent), and artificial intelligence (about 40 percent). I have to emphasize that my own research field —numerical mathematics—is drawing only about 10 percent of our students. This is because the other two areas have problems that seem more exciting, important, and solvable at this particular stage of computer science. Moreover, they require less prior education, permitting the student to start original research at a younger stage. Thus in the past fifteen years many numerical analysts have progressed from being queer people in mathematics departments to being queer people in computer science departments!

Computer science is rich in designs of programming systems and languages, full of techniques for meeting this and that difficulty, and heavily beset with colleagues who request help. We are poor in theorems and general theories; our deep intellectual questions are shared with logic, economics, applied physics, and mathematics. On the other hand, the totality of techniques and ideas built into many of our moderate-sized computing systems (say an Algol compiler or a large eigenvalue routine) is quite impressive, for a computer is extremely good at dealing with very complex situations.

Most of known computer science must be considered as *design technique*, not theory. This doesn't bother us, as we all know that a period of developing technique necessarily precedes periods of consolidating theory, whether the subject be physics, mathematics, biology or computer science. As long as computers continue changing drastically every three or four years, there is scarcely a chance to sit down and contemplate the creation of a theory. In this respect our subject is reminiscent of early engineering, and also of mathematical analysis in the time after Newton. I wish to emphasize my belief that this is a passing stage of computer science.

The most valuable acquisitions in a scientific or technical education are the general-purpose mental tools which remain serviceable for a lifetime. I rate natural language and mathematics as the most important of these tools, and computer science as a third. The mathematics you teach reaches its effective

application largely through digital computing, and hence you and your students need to know some computer science. The learning of mathematics and computer science together has pedagogical advantages, for the basic concepts of each reinforce the learning of the other (e.g., the concepts of *function* in mathematics and *procedure* in Algol 60).

I have emphasized certain differences between computer science and mathematics, particularly because I feel this audience may not be aware of them. However, in another sense computer science and mathematics are remarkably similar. The computer industry is overwhelmed by the pains of growing so large so fast. In 1967 there are over 40,000 computers in the United States. Many thousands of programmers are constantly at work, producing software and descriptions thereof. These people work under extreme pressure of time, and many have had little supervised practice in the twin arts of programming for computers and expounding for human beings. Many compromises are made in the hurried effort to make reasonably available to users programs that work reasonably well (if not perfectly).

Seen from this hurly-burly of production, we academic mathematicians and computer scientists look much alike. We both insist on high standards of rigor and exposition (in mathematicians' language), or performance and documentation (in computer science terminology), and place a higher premium on quality than on promptness. As the computer era matures, we may find ourselves more and more thrown together in defense of this intellectual attitude. For the typical industrial programmer has little sympathy for it. He knows that the computer is often powerful enough to overcome the slipshod way it is understood and used. As an academic type, I can hardly admit it, but I have seen enough computing to believe it. Despite some grave deficiencies in users' understanding of the operation of hardware and software, the fact is that most large programs yield results that are satisfactory to the user—results that satisfy him as well or better than the analyses he used to get from mathematicians!

We academic types must surely defend our premise that critical analyses and proofs are worthwhile in this age of wholesale number-crunching.

What can you do now? And now follow my answers to the question of the title.

First, you can get a little acquainted with computing. This involves two steps:

Step A: Learn to program some automatic digital computer in some language—e.g., Fortran Algol, PL/1—and actually use the computer enough to find out some of the fascination and frustrations of the computerman's world.

Step B: Read some books from the list at the end of this paper. Since computer science is not yet very deep and mathematicians are very smart people, this should not be onerous.

Second, you can study how computing intersects mathematics. Applied mathematics is no longer the same subject, now that you have a magnificent

experimental tool at hand. Moreover, there are several undergraduate courses that owe their large enrollments largely to their wide applications in technology and science: e.g., linear algebra, and ordinary differential equations. I think both of these courses should be substantially influenced by computers.

In a linear algebra course, along with concepts like rank, determinant, eigenvalues, linear systems, and so on, ought to go some constructive computational methods suitable for automatic computers. There is plenty of literature now, and I think some of it should be worked into courses in linear algebra. If not, then an instructor should loudly confess that he is ignoring these topics, and furnish some reading lists for his students.

The same goes for ordinary differential equations. Here the situation is slightly different, in that textbooks in this field usually do say something about numerical methods. The trouble is that it usually dates from before the days of computers. It should be expunged and replaced with at least an equivalent amount of orientation in today's useful numerical methods for computers. See [7] for Professor Hull's suggestions.

I think also that the calculus courses should be influenced by an awareness of computing, but I do not expect this to be a very large fraction of the courses. See [6] for some ideas.

The alternative to weaving computational material into various mathematics courses is to teach computational mathematics in separate courses, in either the department of mathematics or the computer science department. This alternative is the accepted method at present, but many have felt it should be only a temporary expedient. If computational mathematics is taught in the computer science department, what effective mechanism can there be to reunite the theoretical and the computational aspects of mathematics?

There is a good deal of interest nowadays in *computer-aided instruction.* I don't expect this to have a very large application to university mathematics teaching. However, I should like to call your attention to the usefulness of a computer-controlled cathode-ray-tube display and "light pen" in giving vivid graphical representations of sophisticated concepts. In one of these, developed by Professor William McKeeman and Mr. William Rousseau at Stanford University, the scope shows both the complex z plane and the plane of $f(z)$, for any simple elementary function f typed at the console. When the light pen traces any curve in the z-plane, a dot of light traces the curve $f(z)$. Many of the elementary theorems of analytic function theory receive an impressive illustration in this way. Professor Marvin Minsky has used similar displays in dealing with nonlinear ordinary differential equations.

At a more fundamental level, the emergence of computer science has added one more applier of mathematics. Along with operations research, economics, and other more recently mathematized subjects, computer science is relatively more interested in *discrete mathematics* (e.g., combinatorics, logic, graph and flow theory, automata theory, probability, number theory, etc.; see [1]), than

in *continuum* mathematics (e.g., calculus, differential equations, complex variables, etc.). Hence the mathematics department (in my view) should devote much thought to organizing its curriculum suitably from the standpoint of consumers of discrete mathematics. I feel that currently common curricula are inherited from the days when continuum mathematics was more in demand (from physics, mechanical engineering, etc.).

Third, you can help the computer scientist find his way to your campus, and make him feel welcome. Above all, please don't judge him as a mathematician, for he isn't one and isn't supposed to be one—his values are different. The difference in values between mathematics and numerical analysis is the subject of a provocative paper [5].

When the computer scientist does arrive on campus, be prepared for a rather large impact. He is tied to a rampant field of rapidly growing interest to students and scholars everywhere. He will need many colleagues and new buildings. He may take some of the heat off mathematics faculties by providing a partial substitute for mathematics as a research tool. This vast energy may have some undesirable side effects on your sense of importance and even your budget.

Fourth, if you are really enthusiastic, I recommend tackling some research problems of a mathematical nature that would help computer science (and your own publication list). There are serious and important mathematical questions at almost every turn, and most computer scientists aren't very good at mathematics. I will leave to Professor Lax the important area of experimental mathematics. One area of computer science with a probable payoff is the automation of algebra and analysis. So far, most actual computing consists of automated *arithmetic*. A Fortran program, for example, asks a computer to carry out addition, subtraction, multiplication and division of (simulated) real or complex numbers, in a sequence which is dynamically determined by the course of the computation. There is little else. It is clear that computers are capable of automated *algebra*, and there have been experimental systems for this since about 1961. They are still primitive. Some of the roadblocks to further development occur at surprising places. One is the question of *simplification* (e.g., of rational polynomial expressions in n variables). What do we mean by simplification? How shall we do it? See Brown [2] for one indication of the depth of the problem.

Proposed by Dr. R. W. Hamming, but still largely in the future, is the partial automation of *analysis*. Faced with an initial-value problem for an ordinary differential equation, for example, a computer should be able to put the problem into some sort of normal form (using automated algebra, of course). Then the computer should inspect the normal form to see whether it is a recognized standard equation. If it is, then a solution formula should be obtained from a table, and then transformed (by automated algebra) back into the variables originally presented. Of course, the user may want a table of values. The computer then must decide whether to use the solution formula (if one exists), or to

compute a numerical solution. In the latter case, a numerical integration formula must be automatically selected (or devised), and then used (by automatic arithmetic) to produce a table of answers and error bounds (more automated analysis). There are many unsolved problems in this program, and mathematicians are uniquely qualified to define the problems and start their solution.

Most computation to date has been *serial* in nature, with only one computation or decision being made at a time within the central processor. Soon to arrive will be *parallel* computers, in which from two to perhaps several hundred operations can be formed simultaneously. The general pattern of serial computation has been well understood since the work of Babbage, Aiken, von Neumann, and others. There are good research problems in analyzing parallel computation and identifying the important features. See [3] for a recent contribution.

There are good research problems in the theoretical aspects of the design of algorithms. Initiated by Post, Turing, and others, there is an important theory that tells us that some functions are computable on a "Turing machine," and some are not. (Turing machines differ in theoretical capability from existing computers only in having infinite storage capacity.) This theory has been extended to state that some problems can be solved on a Turing machine with a suitable algorithm, but for some problems no such algorithm can exist.

It is essential to know that a problem is solvable, but this is only the beginning. What is needed next is information about how much computer storage is required for the program and data, and how long the algorithm will run. In other words, we need theoretical information on the complexity of solvability. There are some results by Kolmogorov and others on the complexity of a computable function, but much more research is needed.

Other research problems lie in areas further removed from mathematics. One such area is computer graphics—the uses of computers for dealing directly with information in the form of structures. (Examples: representing graphs of mathematical trees, design of networks, recognition of three-dimensional block structures from photographs, automatic reading of bubble chamber pictures.) In this area there are problems of representing information, both visually and inside a computer store, and of processing the information. Most algorithms are being created by persons with only a modest knowledge of mathematics, and it seems likely that an interested mathematician could both help solve some computing problems and find worth-while mathematical problems.

In summary, here are my four answers to the question of the title:

(1) Learn a little about computer science.
(2) Consider how mathematics curricula should be affected by computer science.
(3) Help the computer scientist find his way, but expect a big blast after he gets there.
(4) Think of computer science as a possible source of mathematical research problems.

Some books to read

Here are some suggested book readings in computer science:

F. L. Alt (editor), *Advances in Computers*, annual serial volume, of which the eighth was issued in 1967, Academic Press. (These contain interesting survey articles on a wide variety of topics in computer science.)

Anonymous, *Information*, Freeman, 1966. (Originated as the September 1966 issue of the *Scientific American*.)

Jeremy Bernstein, *The Analytical Engine: Computers, Past, Present, and Future*, Random House, 1964. (A good book to start with; it originally appeared in the *New Yorker*.)

Edward A. Feigenbaum and Julian Feldman (editors), *Computers and Thought*, McGraw-Hill, 1963. (These articles are devoted to the topic of "artificial intelligence": to what extent can computers accomplish tasks heretofore performed by human minds?)

L. Fox (editor), *Advances in Programming and Non-Numerical Computation*, Pergamon, 1966. (Series of articles explaining programming and nonnumerical computation to the uninitiated mathematician. The main nonnumerical applications dealt with here are theorem-proving, game-playing, and information retrieval.)

T. E. Hull, *Introduction to Computing*, Prentice-Hall, 1966. (A first course in Fortran and its use in computing, both arithmetic and symbolic, by a mathematician and numerical analyst. It has a good annotated bibliography that can serve to expand the present list.)

Kenneth E. Iverson, *A Programming Language*, Wiley, 1962. (The author has created a notation useful for describing the logical design of automatic computers and for programming computers. In other works the author makes it clear that he would like his notation to replace mathematical notation, which he finds full of inconsistencies.)

Marvin Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, 1967. (An advanced undergraduate textbook on automata, computability, and so on. Actual automatic computers are never far out of the author's mind.)

B. Randell and L. J. Russell, *Algol 60 Implementation*, Academic Press, 1960. (This book describes a program that translates a program written in Algol 60 into the machine-language program of an actual computer. Such programs are called "compilers," and are by far the most frequent programs run by computers.)

Saul Rosen (editor), *Programming Systems and Languages*, McGraw-Hill, 1967. (One of the most sophisticated of the emerging parts of computer science is the theory of programming languages. It extends from abstract theories of written linguistics over to the psychological questions of what languages human beings can most effectively use.)

Peter Wegner (editor), *Introduction to Systems Programming*, Academic Press, 1964. (By a *system* the author means any program that controls the course of programs through a computer, programs that translate from one language to another, etc. Such systems are the "intelligence" that turns a bare pile of electronic componentry into an effective "living" computing machine.)

J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, 1965. (This is devoted to computing the eigenvalues and eigenvectors of a finite square matrix, by a man who has personally tested and analyzed most known methods. You will be surprised at how little space is wasted in the 662 pages.)

References

1. Edwin F. Beckenbach (editor), Applied Combinatorial Analysis, Wiley, New York, 1964.

2. W. S. Brown, Rational exponential expressions and a conjecture concerning π and e, manuscript, Bell Telephone Laboratories, 1967.

3. A. B. Carroll and R. T. Wetherald, Application of parallel processing to numerical weather prediction, J. Assoc. Comput. Mach., 14 (1967) 591–614.

4. George E. Forsythe, A university's educational program in computer science, Comm. Assoc. Comput. Mach., 10 (1967) 3–11.

5. R. W. Hamming, Numerical analysis vs. mathematics, *Science*, 148 (23 April 1965) 473–475.

6. R. W. Hamming, Calculus and the Computer Revolution, CUPM, P.O. Box 1024, Berkeley, California 94701, 1966.

7. T. E. Hull, The Numerical Integration of Ordinary Differential Equations, CUPM, P.O. Box 1024, Berkeley, California 94701, 1966.

8. Allen Newell, Alan J. Perlis, and Herbert A. Simon, What is computer science? *Science*, 157 (22 Sept. 1967) 1373–4.

9. George Pólya, How To Solve It, 2nd ed. Anchor Book A93, Doubleday, New York. (Several other books.)

10. University of Chicago, *Graduate Programs in the Divisions, Announcements 1966–67*, pp. 175–177, describing their Committee on Information Sciences.