Required Readings for CSE/PHI 4/584
Assigned 19 Jan 2007

Contents:

Letters

Omnibus Language Proposal

Most physical scientists, particularly graduate students, need the "dictionary-hunt" knowledge of two or three foreign languages, despite the contrary opinions and high costs cited by Nichols and Everson (Letters, 23 June). I have a suggestion that may seem bizarre at first; it is based on comments made by Fritz Zwicky at a symposium on Modern Methodology at Caltech recently. Briefly, Zwicky feels that languages can best be taught several at a time, as in his native Switzerland. He claims that in this manner, similarities and differences would stand out and be more easily remembered by students. Several of us urged him to prepare a textbook so that his idea could be tried, possibly in a special course for graduate students in the sciences.

No one seems to have given much thought to a course in "scientific languages," say, German, Russian, French, Italian and Spanish. A graduate student usually has had 2 years in one of these so that the comparative aspects of grammar would not be too difficult. As Zwicky points out, scientific terminology tends to be the same in most languages, and the student specializing in physics, for instance, is in any case helped by equations and diagrams. The purpose of such a course would be to give a student confidence in finding and reading articles in foreign journals about his own thesis topic, without spending the time to learn two or three languages thoroughly. The linguists will undoubtedly object to such shallow treatment, but they may be reassured that regular language courses will still be needed for other purposes, and that the five-language course may reduce the bored fringe of disinterested students in regular language classes. The major problem is who can teach such a course? (other than Zwicky)!

THORNTON PAGE
Department of Astronomy, Wesleyan University, Middletown, Connecticut

Methanol: A New Fuel?

"Energy needs versus environmental pollution: a reconciliation?" (16 June, p. 1448) by Leon Green, Jr., proposed a system of energy generation based upon the use of ammonia as a fuel. The general thesis developed is attractive in that it provides for conversion of fossil fuels into a chemical fuel in such a way that waste products can be readily controlled and contained at the point of release. On the other hand, I think that Green's suggestions would have been much more practical if he had given consideration to the production of methanol rather than ammonia.

The chemical process used to convert fuel gas, petroleum fractions, or even coal to methanol is essentially the same as the process used for production of ammonia. In both, the original raw material is converted to a mixture of carbon monoxide and hydrogen which is then further processed to produce the desired final product. The efficiency of conversion is approximately the same in both cases, and a substantial fraction of the carbon originally present in the fossil fuel disappears from the system as carbon dioxide. In the case of ammonia, all of the carbon is separated in this manner; with methanol, about two-thirds is removed.

The cost of erected facilities for the production of ammonia or methanol are roughly comparable. Once very large plants are designed for producing methanol, the relative simplicity possible in handling the product as compared with the requirements for liquifying and pressurizing the ammonia product will probably result in an advantage in the overall investment cost. Methanol can be stored at atmospheric pressure under all normal conditions and can be readily shipped by pipeline, by normal tank car, or tank truck. Because of its very low freezing point and low viscosity, it can be used easily for all conventional fuel requirements.

It is interesting to note that, with some adjustment to the carburetor, methanol can be used as a fuel in ordinary internal combustion engines. It is a completely clean fuel requiring no additives, lead, or other constituents which tend to aggravate atmospheric pollution problems. Of course, it would be essential that the internal combustion engine be adjusted properly to avoid formation of oxygenated hydrocarbon compounds in the exhaust gases.

Of even more interest is the possibility of utilizing methanol directly as a fuel for a direct conversion fuel cell. Substantial work in this direction has been carried out at Institut Français du Pétrole where demonstration cells have already been built and operated for many thousands of hours. Use of methanol in this manner would permit a ready transition from hydrocarbon fuels inside of city areas with a gradual replacement of internal combustion engines by electric motors powered by fuel cells.

Production of methanol could be taken over completely by large energy companies currently refining petroleum and distributing hydrocarbon fuels. The investment required to produce enough methanol to replace all existing fuels would certainly be extremely high, but may not be out of proportion to that required for producing low-sulfur conventional fuels such as is being required by legislation currently being enacted throughout the country.

RONALD G. MINET
Compagnia Tecnica Industrie Petroli S.p.A., Piazzale G. Douhet 31 (EUR), Rome, Italy

Computer Science

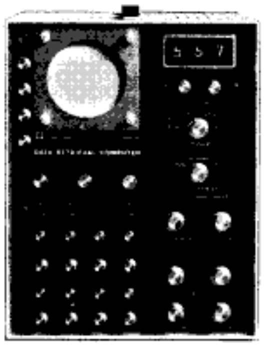
Professors of computer science are often asked: "Is there such a thing as computer science, and if there is, what is it?" The questions have a simple answer:

Wherever there are phenomena, there can be a science to describe and explain those phenomena. Thus, the simplest (and correct) answer to "What is botany?" is, "Botany is the study of plants." And zoology is the study of animals, astronomy the study of stars, and so on. Phenomena breed sciences.

There are computers. Ergo, computer science is the study of computers. The phenomena surrounding computers are

What price signal averaging?

Here's a quick look at the real expense —in data as well as dollars—of signal-averaging devices, including our averager, the Model 7100 Data Retrieval Computer.



Will you pay for less than excellent resolution? You will in any signal averager that has a minimum dwell-time per data point of more than 39 microseconds. Resolution, after all, is a function of the number of data points that can be placed within a region of interest. Our Model 7100 Data Retrieval Computer (DRC) uses *all 400* of its data points for signals occurring within as little as 15.6 milliseconds. The DRC, therefore, gives much better resolution than averagers that use only a fraction of their data points to represent the signal of interest.

Will you pay for less than total versatility? You will in any averager that doesn't have the built-in capability—without add-on options—for interval- and time-histogram analysis, as well as transient-averaging. The DRC will operate in *any* of these three modes, which are selected on a front-panel switch.

Will you pay for less than maximum input sensitivity? You will in an averager that needs a pre-amplifier to accept low-amplitude input signals. The DRC has 20-millivolt input sensitivity. So, most of the time, the DRC requires *no* added pre-amps.

What should you pay for a basic signal averager? That's up to you. But for its price, the DRC offers you more performance, versatility, and convenience than any other comparable signal averager.

The Model 7100 Data Retrieval Computer. Now available at a new, lower price.

For more information, consult your local Nuclear-Chicago sales engineer or write to us.

[illegible]

NUCLEAR-CHICAGO CORPORATION

349 E. Howard Ave., Des Plaines, Ill. 60018 U.S.A.
Donker Curtiusstraat 7, Amsterdam W.

varied, complex, rich. It remains only to answer the objections posed by many skeptics.

Objection 1. Only natural phenomena breed sciences, but computers are artificial, hence are whatever they are made to be, hence obey no invariable laws, hence cannot be described and explained. *Answer.* 1. The objection is patently false, since computers and computer programs are being described and explained daily. 2. The objection would equally rule out of science large portions of organic chemistry (substitute "silicones" for "computers"), physics (substitute "superconductivity" for "computers"), and even zoology (substitute "hybrid corn" for "computers"). The objection would certainly rule out mathematics, but in any event its status as a natural science is idiosyncratic.

Objection 2. The term "computer" is not well defined, and its meaning will change with new developments, hence computer science does not have a well-defined subject matter. *Answer.* The phenomena of all sciences change over time; the process of understanding assures that this will be the case. Astronomy did not originally include the study of interstellar gases; physics did not include radioactivity; psychology did not include the study of animal behavior. Mathematics was once defined as the "science of quantity."

Objection 3. Computer science is the study of algorithms (or programs), not computers. *Answer.* 1. Showing deeper insight than they are sometimes credited with, the founders of the chief professional organization for computer science named it the Association for Computing Machinery. 2. In the definition, "computers" means "living computers"—the hardware, their programs or algorithms, and all that goes with them. Computer science is the study of the phenomena surrounding computers. "Computers plus algorithms," "living computers," or simply "computers" all come to the same thing—the same phenomena.

Objection 4. Computers, like thermometers, are instruments, not phenomena. Instruments lead away to their user sciences; the behaviors of instruments are subsumed as special topics in other sciences (not always the user sciences—electron microscopy belongs to physics, not biology). *Answer.* The computer is such a novel and complex instrument that its behavior is subsumed under no other science; its study does not lead away to user sciences, but to further study of computers. Hence, the computer is not just an instrument but a phenomenon as well, requiring description and explanation.

Objection 5. Computer science is a branch of electronics (or mathematics, psychology, and so forth). *Answer.* To study computers, one may need to study some or all of these. Phenomena define the focus of a science, not its boundaries. Many of the phenomena of computers are also phenomena of some other science. The existence of biochemistry denies neither the existence of biology nor of chemistry. But all of the phenomena of computers are not subsumed under any one existing science.

Objection 6. Computers belong to engineering, not science. *Answer.* They belong to both, like electricity (physics and electrical engineering) or plants (botany and agriculture). Time will tell what professional specialization is desirable between analysis and synthesis, and between the pure study of computers and their application.

Computer scientists will often join hands with colleagues from other disciplines in common endeavor. Mostly, computer scientists will study living computers with the same passion that others have studied plants, stars, glaciers, dyestuffs, and magnetism; and with the same confidence that intelligent, persistent curiosity will yield interesting and perhaps useful knowledge.

ALLEN NEWELL
ALAN J. PERLIS
HERBERT A. SIMON

Graduate School of Industrial Administration, Carnegie Institute of Technology, Pittsburgh, Pennsylvania 15213

"The Big Trouble with Scientific Writing . . ."

When I see articles, as I frequently do these days, exhorting authors to greater simplicity and clarity (*1*), I think of the first little scientific note I wrote, when I was an idealistic graduate student. I wrote it as simply and directly as I could. It began, "The big trouble with diffusion cloud chambers is low radiation resistance," and it went on in the same vein. My co-workers thought it needed a little more work. Secretly I did not agree, so I decided to attempt to make it into a parody of

COMPUTER SCIENCE AND ITS RELATION TO MATHEMATICS

DONALD E. KNUTH

A new discipline called Computer Science has recently arrived on the scene at most of the world's universities. The present article gives a personal view of how this subject interacts with Mathematics, by discussing the similarities and differences between the two fields, and by examining some of the ways in which they help each other. A typical nontrivial problem is worked out in order to illustrate these interactions.

1. What is Computer Science? Since Computer Science is relatively new, I must begin by explaining what it is all about. At least, my wife tells me that she has to explain it whenever anyone asks her what I do, and I suppose most people today have a somewhat different perception of the field than mine. In fact, no two computer scientists will probably give the same definition; this is not surprising, since it is just as hard to find two mathematicians who give the same definition of Mathematics. Fortunately it has been fashionable in recent years to have an "identity crisis," so computer scientists have been right in style.

My favorite way to describe computer science is to say that it is the study of *algorithms*. An algorithm is a precisely-defined sequence of rules telling how to produce specified output information from given input information in a finite number of steps. A particular representation of an algorithm is called a program, just as we use the word "data" to stand for a particular representation of "information" [**14**]. Perhaps the most significant discovery generated by the advent of computers will turn out to be that algorithms, as objects of study, are extraordinarily rich in interesting properties; and furthermore, that an algorithmic point of view is a useful way to organize knowledge in general. G. E. Forsythe has observed that "the question 'What can be automated?' is one of the most inspiring philosophical and practical questions of contemporary civilization" [**8**].

From these remarks we might conclude that Computer Science should have existed long before the advent of computers. In a sense, it did; the subject is deeply rooted in history. For example, I recently found it interesting to study ancient manuscripts, learning to what extent the Babylonians of 3500 years ago were computer scientists [**16**]. But computers are really necessary before we can learn much about the general properties of algorithms; human beings are not precise enough nor fast enough to carry out any but the simplest procedures. Therefore the potential richness of algorithmic studies was not fully realized until general-purpose computing machines became available.

I should point out that computing machines (and algorithms) do not only compute with *numbers*; they can deal with information of any kind, once it is represented

presented inside a computer as if it were a number; but it is really more correct to say that a number is represented inside a computer as a sequence of symbols.

The French word for computer science is *Informatique*; the German is *Informatik*; and in Danish, the word is *Datalogi* [**21**]. All of these terms wisely imply that computer science deals with many things besides the solution to numerical equations. However, these names emphasize the "stuff" that algorithms manipulate (the information or data), instead of the algorithms themselves. The Norwegians at the University of Oslo have chosen a somewhat more appropriate designation for computer science, namely *Databehandling*; its English equivalent, "Data Processing" has unfortunately been used in America only in connection with business applications, while "Information Processing" tends to connote library applications. Several people have suggested the term "Computing Science" as superior to "Computer Science."

Of course, the search for a perfect name is somewhat pointless, since the underlying concepts are much more important than the name. It is perhaps significant, however, that these other names for computer science all de-emphasize the role of computing machines themselves, apparently in order to make the field more "legitimate" and respectable. Many people's opinion of a computing machine is, at best, that it is a necessary evil: a difficult tool to be used if other methods fail. Why should we give so much emphasis to teaching how to use computers, if they are merely valuable tools like (say) electron microscopes?

Computer scientists, knowing that computers are more than this, instinctively underplay the machine aspect when they are defending their new discipline. However, it is not necessary to be so self-conscious about machines; this has been aptly pointed out by Newell, Perlis, and Simon [**22**], who define computer science simply as the study of computers, just as botany is the study of plants, astronomy the study of stars, and so on. The phenomena surrounding computers are immensely varied and complex, requiring description and explanation; and, like electricity, these phenomena belong both to engineering and to science.

When I say that computer science is the study of algorithms, I am singling out only one of the "phenomena surrounding computers," so computer science actually includes more. I have emphasized algorithms because they are really the central core of the subject, the common denominator which underlies and unifies the different branches. It might happen that technology someday settles down, so that in say 25 years computing machines will be changing very little. There are no indications of such a stable technology in the near future, quite the contrary, but I believe that the study of algorithms will remain challenging and important even if the other phenomena of computers might someday be fully explored.

The reader interested in further discussions of the nature of computer science is referred to [**17**] and [**29**], in addition to the references cited above.

about computers which are now being actively studied by computer scientists, and which are hardly mathematical. But if we restrict our attention to the study of algorithms, isn't this merely a branch of mathematics? After all, algorithms were studied primarily by mathematicians, if by anyone, before the days of computer science. Therefore one could argue that this central aspect of computer science is really part of mathematics.

However, I believe that a similar argument can be made for the proposition that mathematics is a part of computer science! Thus, by the definition of set equality, the subjects would be proved equal; or at least, by the Schröder-Bernstein theorem, they would be equipotent.

My own feeling is that neither of these set inclusions is valid. It is alwasy difficult to establish precise boundary lines between disciplines (compare, for example, the subjects of "physical chemistry" and "chemical physics"); but it is possible to distinguish essentially different points of view between mathematics and computer science.

The following true story is perhaps the best way to explain the distinction I have in mind. Some years ago I had just learned a mathematical theorem which implied that any two $n \times n$ matrices A and B of integers have a "greatest common right divisor" D. This means that D is a right divisor of A and of B, i.e., $A = A'D$ and $B = B'D$ for some integer matrices A' and B'; and that every common right divisor of A and B is a right divisor of D. So I wondered how to calculate the greatest common right divisor of two given matrices. A few days later I happened to be attending a conference where I met the mathematician H. B. Mann, and I felt that he would know how to solve this problem. I asked him, and he did indeed know the correct answer; but it was a mathematician's answer, not a computer scientist's answer! He said, "Let $\mathscr{R}$ be the ring of $n \times n$ integer matrices; in this ring, the sum of two principal left ideals is principal, so let D be such that

$$\mathscr{R}A + \mathscr{R}B = \mathscr{R}D.$$

Then D is the greatest common right divisor of A and B." This formula is certainly the simplest possible one, we need only eight symbols to write it down; and it relies on rigorously-proved theorems of mathematical algebra. But from the standpoint of a computer scientist, it is worthless, since it involves constructing the infinite sets $\mathscr{R}A$ and $\mathscr{R}B$, taking their sum, then searching through infinitely many matrices D such that this sum matches the infinite set $\mathscr{R}D$. I could not determine the greatest common divisor of $\binom{1\ 2}{3\ 4}$ and $\binom{4\ 3}{2\ 1}$ by doing such infinite operations. (Incidentally, a computer scientist's answer to this question was later supplied by my student Michael Fredman; see [**15**, p. 380].)

One of my mathematician friends told me he would be willing to recognize computer science as a worthwhile field of study, as soon as it contains 1000 deep well as theorems, say 500 deep theorems and 500 deep algorithms. But even so it is clear that computer science today does not measure up to such a test, if "deep" means that a brilliant person would need many months to discover the theorem or the algorithm. Computer science is still too young for this; I can claim youth as a handicap. We still do not know the best way to describe algorithms, to understand them or to prove them correct, to invent them, or to analyze their behavior, although considerable progress is being made on all these fronts. The potential for "1000 deep results" is there, but only perhaps 50 have been discovered so far.

In order to describe the mutual impact of computer science and mathematics on each other, and their relative roles, I am therefore looking somewhat to the future, to the time when computer science is a bit more mature and sure of itself. Recent trends have made it possible to envision a day when computer science and mathematics will both exist as respected disciplines, serving analogous but different roles in a person's education. To quote George Forsythe again, "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" [**9**].

Like mathematics, computer science will be a subject which is considered basic to a general education. Like mathematics and other sciences, computer science will continue to be vaguely divided into two areas, which might be called "theoretical" and "applied." Like mathematics, computer science will be somewhat different from the other sciences, in that it deals with man-made laws which can be proved, instead of natural laws which are never known with certainty. Thus, the two subjects will be like each other in many ways. The difference is in the subject matter and approach — mathematics dealing more or less with theorems, infinite processes, static relationships, and computer science dealing more or less with algorithms, finitary constructions, dynamic relationships.

Many computer scientists have been doing mathematics, but many more mathematicians have been doing computer science in disguise. I have been impressed by numerous instances of mathematical theories which are really about particular algorithms; these theories are typically formulated in mathematical terms that are much more cumbersome and less natural than the equivalent algorithmic formulation today's computer scientist would use. For example, most of the content of a 35-page paper by Abraham Wald can be presented in about two pages when it is recast into algorithmic terms [**15**, pp. 142–144]; and numerous other examples can be given. But that is a subject for another paper.

3. Educational side-effects. A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them. This knowledge prepares him for much more than writing good computer programs; it is a general-purpose mental tool which will be a definite aid to his understanding of other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not *really* understand something until he can teach it to a *computer*, i.e., express it as an algorithm. "The automatic computer really *forces* that precision of thinking which is alleged to be a product of any study of mathematics" [**7**]. The attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way.

Linguists thought they understood languages, until they tried to explain languages to computers; they soon discovered how much more remains to be learned. Many people have set up computer models of things, and have discovered that they learned more while setting up the model than while actually looking at the output of the eventual program.

For three years I taught a sophomore course in abstract algebra, for mathematics majors at Caltech, and the most difficult topic was always the study of "Jordan canonical form" for matrices. The third year I tried a new approach, by looking at the subject algorithmically, and suddenly it became quite clear. The same thing happened with the discussion of finite groups defined by generators and relations; and in another course, with the reduction theory of binary quadratic forms. By presenting the subject in terms of algorithms, the purpose and meaning of the mathematical theorems became transparent.

Later, while writing a book on computer arithmetic [**15**], I found that virtually every theorem in elementary number theory arises in a natural, motivated way in connection with the problem of making computers do high-speed numerical calculations. Therefore I believe that the traditional courses in elementary number theory might well be changed to adopt this point of view, adding a practical motivation to the already beautiful theory.

These examples and many more have convinced me of the pedagogic value of an algorithmic approach; it aids in the understanding of concepts of all kinds. I believe that a student who is properly trained in computer science is learning something which will implicitly help him cope with many other subjects; and therefore there will soon be good reason for saying that undergraduate computer science majors have received a good general education, just as we now believe this of undergraduate math majors. On the other hand, the present-day undergraduate courses in computer science are not yet fulfilling this goal; at least, I find that many beginning graduafe students with an undergraduate degree in computer science have been more narrowly educated than I would like. Computer scientists are of course working to correct this present deficiency, which I believe is probably due to an overemphasis on computer languages instead of algorithms.

4. Some interactions. Computer science has been affecting mathematics in many ways, and I shall try to list the good ones here. In the first place, of course, computers can be used to compute, and they have frequently been applied in mathematical research when hand computations are too difficult; they generate data which suggests or demolishes conjectures. For example, Gauss said [**10**] that he first thought of the prime number theorem by looking at a table of the primes less than one million. In my own Ph.D. thesis, I was able to resolve a conjecture concerning infinitely many cases by looking closely at computer calculations of the smallest case [**13**]. An example of another kind is Marshall Hall's recent progress in the determination of all simple groups of orders up to one million.

Secondly, there are obvious connections between computer science and mathematics in the areas of numerical analysis [**30**], logic, and number theory; I need not dwell on these here, since they are so widely known. However, I should mention especially the work of D. H. Lehmer, who has combined computing with classical mathematics in several remarkable ways; for example, he has proved that every set of six consecutive integers > 285 contains a multiple of a prime $\geqq 43$.

Another impact of computer science has been an increased emphasis on constructions in all branches of mathematics. Replacing existence proofs by algorithms which construct mathematical objects has often led to improvements in an abstract theory. For example, E. C. Dade and H. Zassenhaus remarked, at the close of a paper written in 1963, "This concept of genus has already proved of importance in the theory of modules over orders. So a mathematical idea introduced solely with a view to computability has turned out to have an intrinsic theoretical value of its own." Furthermore, as mentioned above, the constructive algorithmic approach often has pedagogic value.

Another way in which the algorithmic approach affects mathematical theories is in the construction of one-to-one correspondences. Quite often there have been indirect proofs that certain types of mathematical objects are equinumerous; then a direct construction of a one-to-one correspondence shows that in fact even more is true.

Discrete mathematics, especially combinatorial theory, has been given an added boost by the rise of computer science, in addition to all the other fields in which discrete mathematics is currently being extensively applied.

For references to these influences of computing on mathematics, and for many more examples, the reader is referred to the following sampling of books, each of which contains quite a few relevant papers: [**1**], [**2**], [**4**], [**5**], [**20**], [**24**], [**27**]. Peter Lax's article [**19**] discusses the effect computing has had on mathematical physics.

But actually the most important impact of computer science on mathematics, in my opinion, is somewhat different from all of the above. To me, the most significant thing is that the study of algorithms themselves has opened up a fertile vein of interesting new mathematical problems; it provides a breath of life for many areas of mathematics which had been suffering from a lack of new ideas. Charles

1864: "As soon as an Analytical Engine [i.e., a general-purpose computer] exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise — By what course of calculation can these results be arrived at by the machine in the shortest time?" [**3**]. And again, George Forsythe in 1958: "The use of practically any computing technique itself raises a number of mathematical problems. There is thus a very considerable impact of computation on mathematics itself, and this may be expected to influence mathematical research to an increasing degree" [**26**]. Garrett Birkhoff [**4**, p. 2] has observed that such influences are not a new phenomenon, they were already significant in the early Greek development of mathematics.

I have found that a great many intriguing mathematical problems arise when we try to analyze an algorithm quantitatively, to see how fast it will run on a computer; a typical example of such a problem is worked out below. Another class of problems of great interest concerns the search for best possible algorithms in a given class; see, for example, the recent survey by Reingold [**25**]. And one of the first mathematical theories to be inspired by computer science is the theory of languages, which by now includes many beautiful results; see [**11**] and [**12**]. The excitement of these new theories is the reason I became a computer scientist.

Conversely, mathematics has of course a profound influence on computer science; nearly every branch of mathematical knowledge has been brought to bear somewhere. I recently worked on a problem dealing with discrete objects called "binary trees," which arise frequently in computer representations of things, and the solution to the problem actually involved the complex gamma function times the square of Riemann's zeta function [**6**]. Thus the results of classical mathematics often turn out to be useful in rather amazing places.

The most surprising thing to me, in my own experiences with applications of mathematics to computer science, has been the fact that so much of the mathematics has been of a particular discrete type, examples of which are discussed below. Such mathematics was almost entirely absent from my own training, although I had a reasonably good undergraduate and graduate education in mathematics. Nearly all of my encounters with such techniques during my student days occurred when working problems from this MONTHLY. I have naturally been wondering whether or not the traditional curriculum (the calculus courses, etc.) should be revised in order to include more of these discrete mathematical manipulations, or whether computer science is exceptional in its frequent application of them.

5. A detailed example. In order to clarify some of the vague generalizations and assertions made above, I believe it is best to discuss a typical computer-science problem in some depth. The particular example I have chosen is the one which first led me personally to realize that computer algorithms suggest interesting mathematical problems. This happened in 1962, when I was a graduate student in mathematics;

really ever worn my mathematician's cloak and my computing cap at the same time. A friend of mine remarked that "some good mathematicians at IBM" had been unable to determine how fast a certain well-known computer method works, and I thought it might be an interesting problem to look at.

Here is the problem: Many computer applications involve the retrieval of information by its "name"; for example, we might imagine a Russian-English dictionary, in which we want to look up a Russian word in order to find its English equivalent. A standard computer method, called *hashing*, retrieves information by its name as follows. A rather large number, m, of memory positions within the computer is used to hold the names; let us call these positions $T_1, T_2, \cdots, T_m$. Each of these positions is big enough to contain one name. The number m is always larger than the total number of names present, so at least one of the T_i is empty. The names are distributed among the T_i's in a certain way described below, designed to facilitate retrieval. Another set of memory positions $E_1, E_2, \cdots, E_m$ is used for the information corresponding to the names; thus if T_i is not empty, E_i contains the information corresponding to the name stored in T_i.

The ideal way to retrieve information using such a table would be to take a given name x, and to compute some function $f(x)$, which lies between 1 and m; then the name x could be placed in position $T_{f(x)}$, and the corresponding information in $E_{f(x)}$. Such a function $f(x)$ would make the retrieval problem trivial, if $f(x)$ were easy to compute and if $f(x) \neq f(y)$ for all distinct names $x \neq y$. In practice, however, these latter two requirements are hardly ever satisfied simultaneously; if $f(x)$ is easy to compute, we have $f(x) = f(y)$ for some distinct names. Furthermore, we do not usually know in advance just which names will occur in the table, and the function f must be chosen to work for all names in a very large set U of potential names, where U has many more than m elements. For example, if U contains all sequences of seven letters, there are $26^7 = 8{,}031{,}810{,}176$ potential names; it is inevitable that $f(x) = f(y)$ will occur.

Therefore we try to choose a function $f(x)$, from U into $\{1, 2, \cdots, m\}$, so that $f(x) = f(y)$ will occur with the approximate probability $1/m$, when x and y are distinct names. Such a function f is called a **hash function**. In practice, $f(x)$ is often computed by regarding x as a number and taking its remainder modulo m, plus one; the number m in this case is usually chosen to be prime, since this can be shown to give better results for the sets of names that generally arise in practice. When $f(x) = f(y)$ for distinct x and y, a "collision" is said to occur; collisions are resolved by searching through positions numbered $f(x) + 1$, $f(x) + 2$, etc.

The following algorithm expresses exactly how a hash function $f(x)$ can be used to retrieve the information corresponding to a given name x in U. The algorithm makes use of a variable i which takes on integer values.

STEP 2. If memory position T_i contains the given name x, stop; the derived information is located in memory position E_i.

STEP 3. If memory position T_i is empty, stop; the given name x is not present.

STEP 4. Increase the value of i by one. (Or, if i was equal to m, set i equal to one.) Return to step 2.

We still haven't said how the names get into $T_1, \cdots, T_m$ in the first place; but that is really not difficult. We start with all the T_i empty. Then to insert a new name x, we "look for" x using the above algorithm; it will stop in step 3 because x is not there. Then we set T_i equal to x, and put the corresponding information in E_i. From now on, it will be possible to retrieve this information, whenever the name x is given, since the above algorithm will find position T_i by repeating the actions which took it to that place when x was inserted.

The mathematical problem is to determine how much searching we should expect to make, on the average; how many times must step 2 be repeated before x is found?

This same problem can be stated in other ways, for example in terms of a modified game of "musical chairs." Consider a set of m empty chairs arranged in a circle. A person appears at a random spot just outside the circle and dashes (in a clockwise direction) to the first available chair. This is repeated m times, until all chairs are full. How far, on the average, does the nth person have to run before he finds a seat?

For example, let $m = 10$ and suppose there are ten players: $A, B, C, D, E, F, G, H, I, J$. To get a random sequence, let us assume that the players successively start looking for their seats beginning at chairs numbered according to the first digits of π, namely 3, 1, 4, 1 5, 9, 2, 6, 5, 3. Figure 1 shows the situation after the first six have been seated.

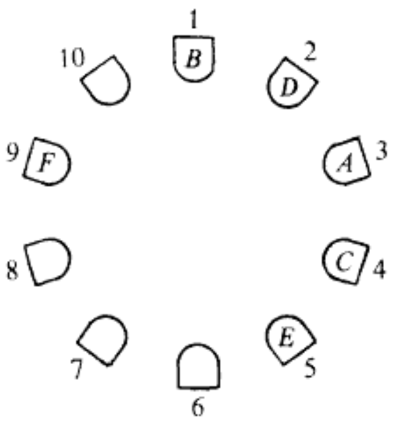


FIG. 1.

(Thus player A takes chair 3, then player B takes chair $1, \cdots,$ player F takes chair 9.) Now player G starts at chair number 2, and eventually he sits down in number 6. Finally, players H, I and J will go into chairs 7, 8, and 10. In this example, the distances travelled by the ten players are respectively 0, 0, 0, 1, 0, 0, 4, 1, 3, 7.

It is not trivial to analyze this problem, because congestion tends to occur; one or more long runs of consecutive occupied chairs will usually be present. In order to see why this is true, we may consider Figure 1 again, supposing that the next player H starts in a random place; then he will land in chair number 6 with probability 0.6, but he will wind up in chair number 7 with probability only 0.1. Long runs tend to get even longer. Therefore we cannot simply assume that the configuration of occupied vs. empty chairs is random at each stage; the piling-up phenomenon must be reckoned with.

Let the starting places of the m players be $a_1 a_2 \cdots a_m$; we shall call this a **hash sequence**. For example, the above hash sequence is 3 1 4 1 5 9 2 6 5 3. Assuming that each of the m^n possible hash sequences is equally likely, our problem is to determine the average distance traveled by the nth player, for each n, in units of "chairs passed." Let us call this distance $d(m,n)$. Obviously $d(m,1) = 0$, since the first player always finds an unoccupied place; furthermore $d(m,2) = 1/m$, since the second player has to go at most one space, and that is necessary only if he starts at the same spot as the first player. It is also easy to see that $d(m,m) = (0+1+\cdots+(m-1))/m = \frac{1}{2}(m-1)$, since all chairs but one will be occupied when the last player starts out. Unfortunately the in-between values of $d(m,n)$ are more complicated.

Let $u_k(m,n)$ be the number of partial hash sequences $a_1 a_2 \cdots a_n$ such that chair k will be unoccupied after the first n players are seated. This is easy to determine, by cyclic symmetry, since chair k is just as likely to be occupied as any other particular chair; in other words, $u_1(m,n) = u_2(m,n) = \cdots = u_m(m,n)$. Let $u(m,n)$ be this common value. Furthermore, $mu(m,n) = u_1(m,n) + u_2(m,n) + \cdots + u_m(m,n) = (m-n)m^n$, since each of the m^n partial hash sequences $a_1 a_2 \cdots a_n$ leaves $m-n$ chairs empty, so it contributes one to exactly $m-n$ of the numbers $u_k(m,n)$. Therefore

$$u_k(m,n) = (m-n)m^{n-1}.$$

Let $v(m,n,k)$ be the number of partial hash sequences $a_1 a_2 \cdots a_n$ such that, after the n players are seated, chairs 1 through k will be occupied, while chairs m and $k+1$ will not. This number is slightly harder to determine, but not really difficult. If we look at the numbers a_i which are $\leqq k+1$ in such a partial hash sequence, and if we cross out the other numbers, the k values which are left form one of the sequences enumerated by $u(k+1,k)$. Furthermore the $n-k$ values crossed out form one of the sequences enumerated by $u(m-1-k, n-k)$, if we subtract $k+1$ from each of them. Conversely, if we take any partial hash sequence $a_1 \cdots a_k$ enumerated by $u(k+1,k)$, and another one $b_1 \cdots b_{n-k}$ enumerated by $u(m-1-k, n-k)$, and if we intermix $a_1 \cdots a_k$ with $(b_1 + k + 1) \cdots (b_{n-k} + k + 1)$ in any of the $\binom{n}{k}$ possible ways, we obtain one of the sequences enumerated by $v(m, n, k)$. Here

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

is the number of ways to choose k positions out of n. For example, let $m = 10$, $n = 6$, $k = 3$; one of the partial hash sequences enumerated by $v(10, 6, 3)$ is 2 7 1 8 2 8. This sequence splits into $a_1a_2a_3 = 2\ 1\ 2$ and $(b_1 + 4)(b_2 + 4)(b_3 + 4) = 7\ 8\ 8$, intermixed in the pattern *ababab*. From each of the $u(4, 3) = 16$ sequences $a_1a_2a_3$ that fill positions 1, 2, 3, together with each of the $u(6, 3) = 108$ sequences $(b_1 + 4)(b_2 + 4)(b_3 + 4)$ that fill three of positions 5, 6, 7, 8, 9, we obtain $\binom{6}{3} = 20$ sequences that fill positions 1, 2, 3, and which leave positions 4 and 10 unoccupied, by intermixing the a's and b's in all possible ways. This correspondence shows that

$$v(m, n, k) = \binom{n}{k} u(k + 1, k)u(m - k - 1, n - k),$$

and our formula for $u(m, n)$ tells us that

$$v(m, n, k) = \binom{n}{k}(k + 1)^{k-1}(m - n - 1)(m - k - 1)^{n-k-1}.$$

This is not a simple formula; but since it is correct, we cannot do any better. If $k = n = m - 1$, the last two factors in the formula give 0/0, which should be interpreted as 1 in this case.

Now we are ready to compute the desired average distance $d(m, n)$. The nth player must move k steps if and only if the preceding partial hash sequence $a_1 \cdots a_{n-1}$ has left chairs a_n through $a_n + k - 1$ occupied and chair $a_n + k$ empty. The number of such partial hash sequences is

$$v(m, n - 1, k) + v(m, n - 1, k + 1) + v(m, n - 1, k + 2) + \cdots,$$

since circular symmetry shows that $v(m, n - 1, k + r)$ is the number of partial hash sequences $a_1 \cdots a_{n-1}$ leaving chairs $a_n + k$ and $a_n - r - 1$ empty while the $k + r$ chairs between them are filled. Therefore the probability $p_k(m, n)$ that the nth player goes exactly k steps is

$$p_k(m, n) = \Bigl(\sum_{r \geqq k} v(m, n - 1, r)\Bigr)/m^{n-1};$$

and the average distance is

$$d(m, n) = \sum_{k \geqq 0} kp_k(m, n) = (m - n)m^{1-n} \sum_{r \geqq k \geqq 0} k \binom{n-1}{r}(r + 1)^{r-1}(m - r - 1)^{n-r-2}$$

$$= \frac{(m - n)m^{1-n}}{2} \sum_{r \geqq 0} r\binom{n-1}{r}(r + 1)^{r}(m - r - 1)^{n-r-2}.$$

At this point, a person with a typical mathematical upbringing will probably stop; the answer is a horrible-looking summation. Yet, if more attention were paid during our mathematical training to finite sums, instead of concentrating so heavily on integrals, we would instinctively recognize that a sum like this can be considerably simplified. When I first looked at this sum, I had never seen one like it before; but I suspected that something could be done to it, since for example, the sum over k of $p_k(m,n)$ must be 1. Later I learned of the extensive literature of such sums. I do not wish to go into the details, but I do want to point out that such sums arise repeatedly in the study of algorithms. By now I have seen literally hundreds of examples in which finite sums involving binomial coefficients and related functions appear in connection with computer science studies; so I have introduced a course called "Concrete Mathematics" at Stanford University, in which this kind of mathematics is taught.

Let $\delta(m,n)$ be the average number of chairs skipped past by the first n players:

$$\delta(m,n) = (d(m,1) + d(m,2) + \cdots + d(m,n))/n.$$

This corresponds to the average amount of time needed for the hashing algorithm to find an item when n items have been stored. The value of $d(m,n)$ derived above can be simplified to obtain the following formulas:

$$d(m,n) = \frac{1}{2}\left(2\frac{n-1}{m} + 3\frac{n-1}{m}\,\frac{n-2}{m} + 4\frac{n-1}{m}\,\frac{n-2}{m}\,\frac{n-3}{m} + \cdots\right),$$

$$\delta(m,n) = \frac{1}{2}\left(\frac{n-1}{m} + \frac{n-1}{m}\,\frac{n-2}{m} + \frac{n-1}{m}\,\frac{n-2}{m}\,\frac{n-3}{m} + \cdots\right).$$

These formulas can be used to see the behavior for large m and n: for example, if $\alpha = n/m$ is the ratio of filled positions to the total number of positions, and if we hold α fixed while m approaches infinity, then $\delta(m,\alpha m)$ increases to the limiting value $\frac{1}{2}\alpha/(1-\alpha)$.

The formula for $\delta(m,n)$ also tells us another surprising thing:

$$\delta(m,n) = \frac{n-1}{2m} + \frac{n-1}{m}\delta(m,n-1).$$

If somebody could discover a simple trick by which this simple relation could be proved directly, it would lead to a much more elegant analysis of the hashing algorithm and it might provide further insights. Unfortunately, I have been unable to think of any direct way to prove this relation.

When $n = m$ (i.e., when all players are seated and all chairs are occupied), the average distance traveled per player is

$$\delta(m,m) = \frac{1}{2}\left(\frac{m-1}{m} + \frac{m-1}{m}\,\frac{m-2}{m} + \frac{m-1}{m}\,\frac{m-2}{m}\,\frac{m-3}{m} + \cdots\right).$$

It is interesting to study this function, which can be shown to have the approximate value

$$\delta(m,m) \approx \sqrt{\frac{\pi m}{8}} - \frac{2}{3}$$

for large m. Thus, the number π, which entered Figure 1 so artificially, is actually present naturally in the problem as well! Such asymptotic calculations, combined with discrete summations as above, are typical of what arises when we study algorithms; classical mathematical analysis and discrete mathematics both play important roles.

6. Extensions. We have now solved the musical chairs problem, so the analysis of hashing is complete. But many more problems are suggested by this one. For example, what happens if each of the hash table positions T_i is able to hold two names instead of one, i.e., if we allow two people per chair in the musical chairs game? Nobody has yet found the exact formulas for this case, although some approximate formulas are known.

We might also ask what happens if each player in the musical chairs game starts *simultaneously* to look for a free chair (still always moving clockwise), starting at independently random points. The answer is that each player will move past $\delta(m,n)$ chairs on the average, where $\delta(m,n)$ is the same as above. This follows from an interesting theorem of W. W. Peterson [**23**], who was the first to study the properties of the hashing problem described above. Peterson proved that the total displacement of the n players, for any partial hash sequence $a_1a_2\cdots a_n$, is independent of the order of the a_i's; thus, 3 1 4 1 5 9 2 leads to the same total displacement as 1 1 2 3 4 5 9 and 2 9 5 1 4 1 3. His theorem shows that the average time $\delta(m,n)$ per player is the same for all arrangements of the a_i, and therefore it is also unchanged when all players start simultaneously.

On the other hand, the average amount of time required until all n players are seated has not been determined, to my knowledge, for the simultaneous case. In fact, I just thought of this problem while writing this paper. New problems flow out of computer science studies at a great rate!

We might also ask what happens if the players can choose to go either clockwise or counterclockwise, whichever is shorter. In the non-simultaneous case, the above analysis can be extended without difficulty to show that each player will then have to go about half as far. (We require everyone to go all the way around the circle to the nearest seat, not taking a short cut through the middle.)

Another variant of the hashing problem arises when we change the cyclic order of probing, in order to counteract the "piling up" phenomenon. This interesting variation is of practical importance, since the congestion due to long stretches of occupied positions slows things down considerably when the memory gets full.

several interesting mathematical aspects, I shall discuss it in detail in the remainder of this article.

A generalized hashing technique which for technical reasons is called **single hashing** is defined by any $m \times m$ matrix Q of integers for which

(i) Each row contains all the numbers from 1 to m in some order;

(ii) The first column contains the numbers from 1 to m in order.

The other columns are unrestricted. For example, one such matrix for $m = 4$, selected more or less at random, is

$$Q_1 = \begin{bmatrix} 1 & 3 & 2 & 4 \\ 2 & 1 & 3 & 4 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{bmatrix} .$$

The idea is to use a hash function $f(x)$ to select a row of Q and then to probe the memory positions in the order dictated by that row. The same algorithm for looking through memory is used as before, except that step 4 becomes

STEP 4′. Advance i to the next value in row $f(x)$ of the matrix, and return to step 2.

Thus, the cyclic hashing scheme described earlier is a special case of single hashing, using a cyclic matrix like

$$Q_2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix} .$$

In the musical chair analogy, the players no longer are required to move clockwise; different players will in general visit the chairs in different sequences. However, if two players start in the same place, they must both follow the same chair-visiting sequence. This latter condition will produce a slight congestion, which is noticeable but not nearly as significant as in the cyclic case.

As before, we can define the measures $d'(m,n)$ and $\delta'(m,n)$, corresponding to the number of times step 4′ is performed. The central problem is to find matrices Q which are **best possible**, in the sense that $\delta'(m,m)$ is minimized. This problem is not really a practical one, since the matrix with smallest $\delta'(m,m)$ might require a great deal of computation per execution of step 4′. Yet it is very interesting to establish absolute limits on how good a single-hashing method could possibly be, as a yardstick by which to measure particular cases.

pleasure of solving is the determination of $d'(m,n)$ for single hashing when the matrix Q is chosen at random, i.e., to find the value of $d'(m,n)$, averaged over all $((m-1)!)^m$ possible matrices Q. The resulting formula is

$$d'_r(m,n) = m - \frac{m-n+1}{m-n+2}\left(1 + \left(m + \sum_{j=1}^{n-1} \frac{1-1/(m+2-j)}{m\prod_{i=1}^{j}(1-1/(m(m+2-i)))}\right) \times \prod_{j=1}^{n-1}\left(1 - \frac{1}{m(m+2-j)}\right)\right).$$

This one I do not know how to simplify at the present time. However, it is possible to study the asymptotic behavior of $d'_r(m,n)$, and to show that

$$\delta'_r(m,m) \approx \ln m + \gamma - 1.5$$

for large m, plus a correction term of order $(\log m)/m$. (Here γ is Euler's constant.) This order of growth is substantially better than the cyclic method, where $\delta(m,m)$ grows like the square root of m; and we know that some single-hashing matrices must have an even lower value for $\delta'(m,m)$ than this average value $\delta'_r(m,m)$. Table 1 shows the exact values of $\delta(m,m)$ and $\delta'_r(m,m)$ for comparatively small values of m; note that cyclic hashing is superior for $m \leqq 11$, but it eventually becomes much worse.

Proofs of the above statements, together with additional facts about hashing, appear in [**18**].

No satisfactory lower bounds for the value of $\delta'(m,m)$ in the best single-hashing scheme are known, although I believe that none will have $\delta'(m,m)$ lower than

$$\left(1+\frac{1}{m}\right)\left(1+\frac{1}{2}+\cdots+\frac{1}{m}\right)-2;$$

this is the value which arises in the musical chairs game if each player follows a random path independently of all the others. J. D. Ullman [**28**] has given a more general conjecture from which this statement would follow. If Ullman's conjecture is true, then a *random* Q comes within $\frac{1}{2}$ of the best possible value, and a large number of matrices will therefore yield values near the optimum. Therefore it is an interesting practical problem to construct a family of matrices for various m, having provably good behavior near the optimum, and also with the property that they are easy to compute in step 4′.

It does not appear to be easy to compute $\delta'(m,m)$ for a given matrix M. The best method I know requires on the order of $m \cdot 2^m$ steps, so I have been able to experiment on this problem only for small values of m. (Incidentally, such experiments represent an application of computer science to solve a mathematical problem suggested by computer science.) Here is a way to compute $\delta'(m,m)$ for a given matrix $Q = (q_{ij})$: If A is any subset of $\{1, 2, \ldots, m\}$, let $\| A \|$ be the number of ele ments in A, and let $p(A)$ be the probability that the first $\|A\|$ players occupy the chairs designated by A. Then it is not difficult to show that

$$p(A) = \frac{1}{m} \sum_{(i,j) \in s(A)} p(A - \{q_{ij}\})$$

when A is nonempty, where $s(A)$ is the set of all pairs (i,j) such that $q_{ik} \in A$ for $1 \leqq k \leqq j$; consequently

$$d'(m,n) = \frac{1}{m} \sum_{\|A\| = n-1} \|s(A)\| \, p(A),$$

$$\delta'(m,m) = \frac{1}{m^2} \sum_{A} \|s(A)\| \, p(A).$$

For example, in the 4×4 matrix Q_1 considered earlier, we have

A	$p(A)$	$\|s(A)\|$	A	$p(A)$	$\|s(A)\|$
$\varnothing$	1	0	$\{4\}$	1/4	1
$\{1\}$	1/4	1	$\{1,4\}$	2/16	2
$\{2\}$	1/4	1	$\{2,4\}$	2/16	2
$\{1,2\}$	3/16	3	$\{1,2,4\}$	9/64	4
$\{3\}$	1/4	1	$\{3,4\}$	4/16	4
$\{1,3\}$	3/16	3	$\{1,3,4\}$	20/64	7
$\{2,3\}$	2/16	2	$\{2,3,4\}$	16/64	6
$\{1,2,3\}$	19/64	7	$\{1,2,3,4\}$	1	16

The first three chairs occupied will most probably be $\{1,3,4\}$; the set of chairs $\{1,2,4\}$ is much less likely. The "score" $\delta'(m,m)$ for this matrix comes to 653/1024, which in this case is worse than the score 624/1024 for cyclic hashing. In fact, cyclic hashing turns out to be the *best* single hashing scheme when $m = 4$.

When $m = 5$, the best single hashing scheme turns out to be obtained from the matrix

$$Q_5 = \begin{bmatrix} 1 & 2 & 4 & 5 & 3 \\ 2 & 3 & 5 & 1 & 4 \\ 3 & 4 & 1 & 2 & 5 \\ 4 & 5 & 2 & 3 & 1 \\ 5 & 1 & 3 & 4 & 2 \end{bmatrix}$$

whose score is 0.7440, compared to 0.7552 for cyclic hashing. Note that Q_5 is very much like cyclic hashing, since cyclic symmetry is present; each row is obtained from the preceding row by adding 1 modulo 5, so that the probing pattern is essentially the same for all rows. We may call this **generalized cyclic hashing**; it is a special case of practical importance, because it requires knowing only one row of Q instead of all m^2 entries.

When $m > 5$, an exhaustive search for the best single hashing scheme would be too difficult to do by machine, unless some new breakthrough is made in the theory. Therefore I have resorted to "heuristic" search procedures. For all $m \leqq 11$, the best single hashing matrices I have been able to find actually have turned out to be generalized cyclic hashing schemes, and I am tempted to conjecture that this will be true in general. It would be extremely nice if this conjecture were true, since it would follow that the potentially expensive generality of a non-cyclic scheme would never be useful. However, the evidence for my guess is comparatively weak;

TABLE 1. Cyclic hashing versus random single hashing

m	$\delta(m, m)$	$\delta_r'(m, m)$
1	0.0000	0.0000
2	0.2500	0.2500
3	0.4444	0.4630
4	0.6094	0.6426
5	0.7552	0.7973
6	0.8874	0.9330
7	1.0091	1.0538
8	1.1225	1.1626
9	1.2292	1.2616
10	1.3301	1.3523
11	1.4262	1.4360
12	1.5180	1.5138
15	1.7729	1.7183
20	2.1468	1.9911
30	2.7747	2.3888
40	3.3046	2.6774
50	3.7716	2.9037
75	4.7662	3.3181
100	5.6050	3.6135

it is simply that (i) the conjecture holds for $m \leqq 5$; (ii) I have seen no counterexamples in experiments for $m \leqq 11$; (iii) the best generalized cyclic hashing schemes for $m \leqq 9$ are "locally optimum" single hashing schemes, in the sense that all possible interchanges of two elements in any row of the matrix lead to a matrix that is no better; (iv) the latter statement is *not* true for the standard (ungeneralized) cyclic hashing scheme, so the fact that it holds for the best ones may be significant.

Even if this conjecture is false, the practical significance of generalized cyclic hashing makes it a suitable object for further study, especially in view of its additional mathematical structure. One immediate consequence of the cyclic property is that $p(A) = p(A + k)$ for all sets A, in the above formulas for computing $d'(m, n)$, where "$A + k$" means the set obtained from A by adding k to each element, modulo m. This observation makes the calculation of scores almost m times faster. Another, not quite so obvious property, is the fact that the generalized cyclic hashing scheme generated by the permutation $q_1 q_2 \cdots q_m$ has the same score as that generated by the "reflected" permutation $q'_1 q'_2 \cdots q'_m$ where $q'_j = m + 1 - q_j$. (It is convenient to say that a generalized cyclic hashing scheme is "generated" by any of its rows.) This equivalence under reflection can be proved by showing that $p(A)$ is equal to $p'(m + 1 - A)$.

I programmed a computer to find the scores for all generalized cyclic hashing schemes when $m = 6$, and the results of this computation suggested that two further simplifications might be valid:

(i) $q_1 q_2 q_3 \cdots q_m$ and $q_2 q_1 q_3 \cdots q_m$ generate equally good generalized cyclic hashing schemes.

(ii) $q_1 \cdots q_{m-2} q_{m-1} q_m$ and $q_1 \cdots q_{m-2} q_m q_{m-1}$ generate equally good generalized cyclic hashing schemes.

In fact, both of these statements are true; here is a typical instance where computing in a particular case has led to new mathematical theorems.

In fact, the above results made me suspect that $q_1 \cdots q_m$ and

$$(m + 1 - q_1) \cdots (m + 1 - q_k) q_{k+1} \cdots q_m$$

will always generate equally good schemes, whenever both of these sequences are permutations. If this statement were true, it would include the three previous results as special cases, for $k = 2$, $m - 2$ and m. Unfortunately, I could not prove it; and I eventually found a counterexample (by hand), namely $q_1 \cdots q_m = 1\ 3\ 8\ 6\ 2\ 7\ 5\ 4$ and $k = 4$. However, this mistaken conjecture did lead to an interesting purely mathematical question, namely to determine how many inequivalent permutations of m objects there are, when $q_1 \cdots q_m$ is postulated to be equivalent to $(\varepsilon q_1 + j) \cdots (\varepsilon q_k + j) q_{k+1} \cdots q_m$, for $\varepsilon = \pm 1$ and $1 \leqq j, k \leqq m$ (whenever these are both permutations, modulo m). We might call these "necklace permutations," by analogy with another well-known combinatorial problem, since they represent the number of different orders in which a person could change the beads of a necklace from all white to all black, ignoring the operation of rotating and/or flipping the necklace over whenever such an operation preserves the current black/white pattern. The total number of different necklace permutations for $m = 1, 2, 3, 4, 5, 6, 7$ is $1, 1, 1, 2, 4, 14, 62$, respectively, and I wonder what can be said for general m.

Returning to the hashing problem, the theorems mentioned above make it possible to study all of the generalized cyclic hashing schemes for $m \leqq 9$, by computer; and the following turn out to be the best:

best permutation	$\delta'_{\min}(m, m)$	$\delta'_{\text{ave}}(m, m)$
1 2 3 4	0.6094	0.6146
1 2 4 5 3	0.7440	0.7514
1 2 5 3 4 6	0.8650	0.8819
1 4 2 3 6 5 7	0.9713	0.9866
1 3 4 8 7 2 6 5	1.0676	1.0919
1 5 2 3 8 4 6 7 9	1.1568	1.1790

The righthand column gives the average $\delta'(m, m)$ over all $m!$ schemes. For $m = 10$ and 11 the best permutations I have found so far are 1 2 8 6 4 9 3 10 7 5 and 1 3 4 8 9 7 11 2 10 6 5, with respective scores of 1.2362 and 1.3103. The *worst* such schemes for $m \leqq 9$ are

worst permutation	$\delta'_{\max}(m,m)$
1 3 2 4	0.6250
1 2 3 4 5	0.7552
1 3 5 2 4 6	0.9132
1 2 3 4 5 6 7	1.0091
1 5 3 7 4 8 2 6	1.1719
1 4 7 2 5 8 3 6 9	1.2638

(This table suggests that the form of the worst cyclic scheme might be obtainable in a simple way from the prime factors of m.)

Finally I have tried to find the worst possible Q matrices, *without* the cyclic constraint. Such matrices can be very bad indeed; the worst I know, for any m, occur when $q_{ij} < q_{i(j+1)}$ for all $j \geqq 2$, e.g.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 3 & 4 & 5 \\ 3 & 1 & 2 & 4 & 5 \\ 4 & 1 & 2 & 3 & 5 \\ 5 & 1 & 2 & 3 & 4 \end{pmatrix}$$

when $m = 5$. Using discrete mathematical techniques like those illustrated above, I have proved that the score for such matrices is

$$\delta'(m, m) = \left(m + 3 + \frac{2}{m}\right)\left(1 + \frac{1}{m}\right)^m - 2.5m - 7 - \frac{2.5}{m},$$

which is approximately $(e - 2.5)m + 3e - 8$ when m is large. We certainly would not want to retrieve information in this way, and perhaps it is the worst possible single hashing scheme.

Thus, the example of hashing illustrates the typical interplay between computer science and mathematics.

I wish to thank Garrett Birkhoff for his comments on the first draft of this paper.

References

1. Amer. Math. Society and Math. Assoc. of America, co-sponsors of conference, The Influence of Computing on Mathematical Research and Education, August 1973.

2. A. O. L. Atkin and B. J. Birch, eds., Computers in Number Theory, Academic Press, New York, 1971.

3. Charles Babbage, Passages from the Life of a Philosopher, (London, 1864). Reprinted in *Charles Babbage and His Calculating Engines*, by Philip and Emily Morrison, Dover, New York 1961; esp. p. 69.

4. Garrett Birkhoff and Marshall Hall, Jr., eds., Computers in Algebra and Number Theory, SIAM-AMS Proceedings, 4 (Amer. Math. Soc., 1971).

5. R. F. Churchhouse and J. -C. Herz, eds., Computers in Mathematical Research, North-Holland, Amsterdam, 1968.

6. N. G. de Bruijn, Donald E. Knuth, and S. O. Rice, The average height of planted plane trees, in *Graph Theory and Computing*, ed. by Ronald C. Read, Academic Press, New York, 1972, 15–22.

7. George E. Forsythe, The role of numerical analysis in an undergraduate program, this MONTHLY, 66 (1959) 651–662.

8. ———, Computer Science and Education, Information Processing 68, 1025–1039.

9. ———, What to do till the computer scientist comes, this MONTHLY, 75 (1968) 454–462.

10. K. F. Gauss, Letter to Enke, *Werke*, vol. 2, 444–447.

11. Seymour Ginsburg, The Mathematical Theory of Context Free Languages, McGraw-Hill, New York; 1966.

12. ———, Sheila Greibach, and John Hopcroft, Studies in abstract families of languages, Amer. Math. Society Memoirs, 87 (1969) 51 pp.

13. Donald E. Knuth, A class of projective planes, Trans. Amer. Math. Soc., 115 (1965) 541–549.

14. ———, Algorithm and program; information and data, Comm. ACM, 9 (1966), 654.

15. ———, Seminumerical Algorithms, Addison-Wesley, Reading, Mass., 1969.

16. ———, Ancient Babylonian algorithms, Comm. ACM, 15 (1972) 671–677.

17. ———, George Forsythe and the development of Computer Science, Comm. ACM, 15 (1972) 721–726.

18. ———, Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

19. Peter D. Lax, The impact of computers on mathematics, Chapter 10 of *Computers and Their Role in the Physical Sciences*, ed. by S. Fernbach and A. Taub, Gordon and Breach, New York, 1970, 219–226.

20. John Leech, ed., Computational Problems in Abstract Algebra, Pergamon, Long Island City, 1970.

21. Peter Naur, 'Datalogy', the science of data and data processes, and its place in education, Information Processing 68, vol. 2, 1383–1387.

22. Allen Newell, Alan J. Perlis, and Herbert A. Simon, Computer Science, Science, 157 (1967) 1373–1374.

23. W. W. Peterson, Addressing for random-access storage, IBM Journal of Res. and Devel., 1 (1957) 130–146.

24. Proc. Symp. Applied Math 15, Experimental Arithmetic, High-Speed Computing, and Mathematics, Amer. Math. Soc., 1963.

25. E. Reingold, Establishing lower bounds on algorithms — A survey, AFIPS Conference Proceedings, 40 (1972) 471–481.

26. Paul C. Rosenbloom and George E. Forsythe, Numerical Analysis and Partial Differential Equations, Surveys in Applied Math 5, Wiley, New York, 1958.

27. Computers and Computing, Slaught Memorial Monograph No. 10, supplement to this MONTHLY, 72 (February 1965) 156 pp.
28. J. D. Ullman, A note on the efficiency of hashing functions, J. ACM, 19 (1972) 569–575.
29. Peter Wegner, Three computer cultures, Advances in Computers, 10 (1970) 7–78.
30. J. H. Wilkinson, Some comments from a numerical analyst, J. ACM, 18 (1971) 137–147.

COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94305.

MAXWELL'S EQUATIONS

THEODORE FRANKEL

1. Introduction. We shall consider Maxwell's equations

$$(1) \cdots \operatorname{div} \mathbf{B} = 0 \qquad (2) \cdots \operatorname{div} \mathbf{D} = \sigma$$

$$(3) \cdots \operatorname{curl} \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \qquad (4) \cdots \operatorname{curl} \mathbf{H} = \mathbf{j} + \frac{\partial \mathbf{D}}{\partial t}$$

in a "non-inductive" medium; i.e., $\mathbf{E} = \mathbf{D}$ is the electric field vector, $\mathbf{B} = \mathbf{H}$ is the magnetic field vector, σ is the charge density, and $\mathbf{j}$ is the current density vector.

These equations are usually taken as axioms in electromagnetic field theory. (1) says that there are no magnetic charges. (2) is Gauss' law, stating that one can compute the total charge inside a closed surface by integrating the normal component of $\mathbf{D}$ or $\mathbf{E}$ over the surface. (3) is Faraday's law; a changing magnetic field produces an electric field. Finally, (4) is Ampere's law curl $\mathbf{H} = \mathbf{j}$ modified by Maxwell's term $\partial\mathbf{D}/\partial t$, stating that currents and changing electric fields produce magnetic fields. Equations (1) and (2) are relatively simple and easily understood while (3) and (4) seem much more sophisticated. It is comforting to know then, that **in a certain sense, Faraday's law** (3) **is a consequence of** (1), **while the Ampere-Maxwell law** (4) **is a consequence of Gauss' law** (2). The precise statement will be found in Section 4. This apparently is a "folk-theorem" of physics; I first ran across the statement of it in an article of J. A. Wheeler ([**3**], p. 84). The precise statement involves only the simplest notions of special relativity and the proof of the statement is an extremely simple application of the formalism of exterior differential forms and could be written down in a few lines. I prefer to preface the proof with a very brief summary of special relativity and of how electromagnetism fits into special relativity, mainly because most (but not all) treatments of this subject motivate their constructions by means of Maxwell's equations; from our view point this would be circular and far less appealing than the approach via the Lorentz force.

2. The Minkowski Space of Special Relativity. Space-time is a 4-dimensional

The 1975 ACM Turing Award was presented jointly to Allen Newell and Herbert A. Simon at the ACM Annual Conference in Minneapolis, October 20. In introducing the recipients, Bernard A. Galler, Chairman of the Turing Award Committee, read the following citation:

"It is a privilege to be able to present the ACM Turing Award to two friends of long standing, Professors Allen Newell and Herbert A. Simon, both of Carnegie-Mellon University.

"In joint scientific efforts extending over twenty years, initially in collaboration with J.C. Shaw at the RAND Corporation, and subsequently with numerous faculty and student colleagues at Carnegie-Mellon University, they have made basic contributions to artificial intelligence, the psychology of human cognition, and list processing.

"In artificial intelligence, they contributed to the establishment of the field as an area of scientific endeavor, to the development of heuristic programming generally, and of heuristic search, means-ends analysis, and methods of induction, in particular; providing demonstrations of the sufficiency of these mechanisms to solve interesting problems.

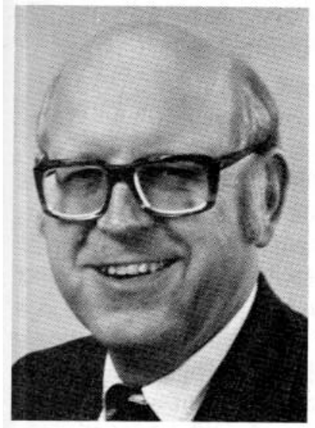
"In psychology, they were principal instigators of the idea that human cognition can be described in terms of a symbol system, and they have developed detailed theories for human problem solving, verbal learning and inductive behavior in a number of task domains, using computer programs embodying these theories to simulate the human behavior.

"They were apparently the inventors of list processing, and have been major contributors to both software technology and the development of the concept of the computer as a system of manipulating symbolic structures and not just as a processor of numerical data.

"It is an honor for Professors Newell and Simon to be given this award, but it is also an honor for ACM to be able to add their names to our list of recipients, since by their presence, they will add to the prestige and importance of the ACM Turing Award."

Computer Science as Empirical Inquiry: Symbols and Search

Allen Newell and Herbert A. Simon



Computer science is the study of the phenomena surrounding computers. The founders of this society understood this very well when they called themselves the Association for Computing Machinery. The machine—not just the hardware, but the programmed, living machine—is the organism we study.

This is the tenth Turing Lecture. The nine persons who preceded us on this platform have presented nine different views of computer science. For our organism, the machine, can be studied at many levels and from many sides. We are deeply honored to appear here today and to present yet another view, the one that has permeated the scientific work for which we have been

Key Words and Phrases: symbols, search, science, computer science, empirical, Turing, artificial intelligence, intelligence, list processing, cognition, heuristics, problem solving.

CR Categories: 1.0, 2.1, 3.3, 3.6, 5.7.

The authors' research over the years has been supported in part by the Advanced Research Projects Agency of the Department of Defense (monitored by the Air Force Office of Scientific Research) and in part by the National Institutes of Mental Health.

Authors' address: Carnegie-Mellon University, Pittsburgh.

cited. We wish to speak of computer science as empirical inquiry.

Our view is only one of many; the previous lectures make that clear. However, even taken together the lectures fail to cover the whole scope of our science. Many fundamental aspects of it have not been represented in these ten awards. And if the time ever arrives, surely not soon, when the compass has been boxed, when computer science has been discussed from every side, it will be time to start the cycle again. For the hare as lecturer will have to make an annual sprint to overtake the cumulation of small, incremental gains that the tortoise of scientific and technical development has achieved in his steady march. Each year will create a new gap and call for a new sprint, for in science there is no final word.

Computer science is an empirical discipline. We would have called it an experimental science, but like astronomy, economics, and geology, some of its unique forms of observation and experience do not fit a narrow stereotype of the experimental method. None the less, they are experiments. Each new machine that is built is an experiment. Actually constructing the machine poses a question to nature; and we listen for the answer by observing the machine in operation and analyzing it by all analytical and measurement means available. Each new program that is built is an expermient. It poses a question to nature, and its behavior offers clues to an answer. Neither machines nor programs are black boxes; they are artifacts that have been designed, both hardware and software, and we can open them up and look inside. We can relate their structure to their behavior and draw many lessons from a single experiment. We don't have to build 100 copies of, say, a theorem prover, to demonstrate statistically that it has not overcome the combinatorial explosion of search in the way hoped for. Inspection of the program in the light of a few runs reveals the flaw and lets us proceed to the next attempt.

We build computers and programs for many reasons. We build them to serve society and as tools for carrying out the economic tasks of society. But as basic scientists we build machines and programs as a way of discovering new phenomena and analyzing phenomena we already know about. Society often becomes confused about this, believing that computers and programs are to be constructed only for the economic use that can be made of them (or as intermediate items in a developmental sequence leading to such use). It needs to understand that the phenomena surrounding computers are deep and obscure, requiring much experimentation to assess their nature. It needs to understand that, as in any science, the gains that accrue from such experimentation and understanding pay off in the permanent acquisition of new techniques; and that it is these techniques that will create the instruments to help society in achieving its goals.

Our purpose here, however, is not to plead for understanding from an outside world. It is to examine one aspect of our science, the development of new basic understanding by empirical inquiry. This is best done by illustrations. We will be pardoned if, presuming upon the occasion, we choose our examples from the area of our own research. As will become apparent, these examples involve the whole development of artificial intelligence, especially in its early years. They rest on much more than our own personal contributions. And even where we have made direct contributions, this has been done in cooperation with others. Our collaborators have included especially Cliff Shaw, with whom we formed a team of three through the exciting period of the late fifties. But we have also worked with a great many colleagues and students at Carnegie-Mellon University.

Time permits taking up just two examples. The first is the development of the notion of a symbolic system. The second is the development of the notion of heuristic search. Both conceptions have deep significance for understanding how information is processed and how intelligence is achieved. However, they do not come close to exhausting the full scope of artificial intelligence, though they seem to us to be useful for exhibiting the nature of fundamental knowledge in this part of computer science.

I. Symbols and Physical Symbol Systems

One of the fundamental contributions to knowledge of computer science has been to explain, at a rather basic level, what symbols are. This explanation is a scientific proposition about Nature. It is empirically derived, with a long and gradual development.

Symbols lie at the root of intelligent action, which is, of course, the primary topic of artificial intelligence. For that matter, it is a primary question for all of computer science. For all information is processed by computers in the service of ends, and we measure the intelligence of a system by its ability to achieve stated ends in the face of variations, difficulties and complexities posed by the task environment. This general investment of computer science in attaining intelligence is obscured when the tasks being accomplished are limited in scope, for then the full variations in the environment can be accurately foreseen. It becomes more obvious as we extend computers to more global, complex and knowledge-intensive tasks—as we attempt to make them our agents, capable of handling on their own the full contingencies of the natural world.

Our understanding of the systems requirements for intelligent action emerges slowly. It is composite, for no single elementary thing accounts for intelligence in all its manifestations. There is no "intelligence principle," just as there is no "vital principle" that conveys by its very nature the essence of life. But the lack of a simple *deus ex machina* does not imply that there are no structural requirements for intelligence. One such requirement is the ability to store and manipulate symbols. To put the scientific question, we may paraphrase the title of a famous paper by Warren McCulloch [1961]: What is a symbol, that intelligence may use it, and intelligence, that it may use a symbol?

Laws of Qualitative Structure

All sciences characterize the essential nature of the systems they study. These characterizations are invariably qualitative in nature, for they set the terms within which more detailed knowledge can be developed. Their essence can often be captured in very short, very general statements. One might judge these general laws, due to their limited specificity, as making relatively little contribution to the sum of a science, were it not for the historical evidence that shows them to be results of the greatest importance.

The Cell Doctrine in Biology. A good example of a law of qualitative structure is the cell doctrine in biology, which states that the basic building block of all living organisms is the cell. Cells come in a large variety of forms, though they all have a nucleus surrounded by protoplasm, the whole encased by a membrane. But this internal structure was not, historically, part of the specification of the cell doctrine; it was subsequent specificity developed by intensive investigation. The cell doctrine can be conveyed almost entirely by the statement we gave above, along with some vague notions about what size a cell can be. The impact of this law on biology, however, has been tremendous, and the lost motion in the field prior to its gradual acceptance was considerable.

Plate Tectonics in Geology. Geology provides an interesting example of a qualitative structure law, interesting because it has gained acceptance in the last decade and so its rise in status is still fresh in memory. The theory of plate tectonics asserts that the surface of the globe is a collection of huge plates—a few dozen in all—which move (at geological speeds) against, over, and under each other into the center of the earth, where they lose their identity. The movements of the plates account for the shapes and relative locations of the continents and oceans, for the areas of volcanic and earthquake activity, for the deep sea ridges, and so on. With a few additional particulars as to speed and size, the essential theory has been specified. It was of course not accepted until it succeeded in explaining a number of details, all of which hung together (e.g. accounting for flora, fauna, and stratification agreements between West Africa and Northeast South America). The plate tectonics theory is highly qualitative. Now that it is accepted, the whole earth seems to offer evidence for it everywhere, for we see the world in its terms.

The Germ Theory of Disease. It is little more than a century since Pasteur enunciated the germ theory of disease, a law of qualitative structure that produced a revolution in medicine. The theory proposes that most diseases are caused by the presence and multiplication in the body of tiny single-celled living organisms, and that contagion consists in the transmission of these organisms from one host to another. A large part of the elaboration of the theory consisted in identifying the organisms associated with specific diseases, describing them, and tracing their life histories. The fact that the law has many exceptions—that many diseases are not produced by germs—does not detract from its importance. The law tells us to look for a particular kind of cause; it does not insist that we will always find it.

The Doctrine of Atomism. The doctrine of atomism offers an interesting contrast to the three laws of qualitative structure we have just described. As it emerged from the work of Dalton and his demonstrations that the chemicals combined in fixed proportions, the law provided a typical example of qualitative structure: the elements are composed of small, uniform particles, differing from one element to another. But because the underlying species of atoms are so simple and limited in their variety, quantitative theories were soon formulated which assimilated all the general structure in the original qualitative hypothesis. With cells, tectonic plates, and germs, the variety of structure is so great that the underlying qualitative principle remains distinct, and its contribution to the total theory clearly discernible.

Conclusion. Laws of qualitative structure are seen everywhere in science. Some of our greatest scientific discoveries are to be found among them. As the examples illustrate, they often set the terms on which a whole science operates.

Physical Symbol Systems

Let us return to the topic of symbols, and define a *physical symbol system*. The adjective "physical" denotes two important features: (1) Such systems clearly obey the laws of physics—they are realizable by engineered systems made of engineered components; (2) although our use of the term "symbol" prefigures our intended interpretation, it is not restricted to human symbol systems.

A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure). Thus, a symbol structure is composed of a number of instances (or tokens) of symbols related in some physical way (such as one token being next to another). At any instant of time the system will contain a collection of these symbol structures. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction and destruction. A physical symbol system is a machine that produces through time an evolving collection of symbol structures. Such a system exists in a world of objects wider than just these symbolic expressions themselves.

Two notions are central to this structure of expressions, symbols, and objects: designation and interpretation.

> *Designation.* An expression designates an object if, given the expression, the system can either affect the object itself or behave in ways dependent on the object.

In either case, access to the object via the expression has been obtained, which is the essence of designation.

> *Interpretation.* The system can interpret an expression if the expression designates a process and if, given the expression, the system can carry out the process.

Interpretation implies a special form of dependent action: given an expression the system can perform the indicated process, which is to say, it can evoke and execute its own processes from expressions that designate them.

A system capable of designation and interpretation, in the sense just indicated, must also meet a number of additional requirements, of completeness and closure. We will have space only to mention these briefly; all of them are important and have far-reaching consequences.

(1) A symbol may be used to designate any expression whatsoever. That is, given a symbol, it is not prescribed a priori what expressions it can designate. This arbitrariness pertains only to symbols; the symbol tokens and their mutual relations determine what object is designated by a complex expression. (2) There exist expressions that designate every process of which the machine is capable. (3) There exist processes for creating any expression and for modifying any expression in arbitrary ways. (4) Expressions are stable; once created they will continue to exist until explicitly modified or deleted. (5) The number of expressions that the system can hold is essentially unbounded.

The type of system we have just defined is not unfamiliar to computer scientists. It bears a strong family resemblance to all general purpose computers. If a symbol manipulation language, such as LISP, is taken as defining a machine, then the kinship becomes truly brotherly. Our intent in laying out such a system is not to propose something new. Just the opposite: it is to show what is now known and hypothesized about systems that satisfy such a characterization.

We can now state a general scientific hypothesis—a law of qualitative structure for symbol systems:

> *The Physical Symbol System Hypothesis.* A physical symbol system has the necessary and sufficient means for general intelligent action.

By "necessary" we mean that any system that exhibits general intelligence will prove upon analysis to be a physical symbol system. By "sufficient" we mean that any physical symbol system of sufficient size can be organized further to exhibit general intelligence. By "general intelligent action" we wish to indicate the same scope of intelligence as we see in humian action: that in any real situation behavior approprate to the ends of the system and adaptive to the demands of the environment can occur, within some limits of speed and complexity.

The Physical Symbol System Hypothesis clearly is a law of qualitative structure. It specifies a general class of systems within which one will find those capable of intelligent action.

This is an empirical hypothesis. We have defined a class of systems; we wish to ask whether that class accounts for a set of phenomena we find in the real world. Intelligent action is everywhere around us in the biological world, mostly in human behavior. It is a form of behavior we can recognize by its effects whether it is performed by humans or not. The hypothesis could indeed be false. Intelligent behavior is not so easy to produce that any system will exhibit it willy-nilly. Indeed, there are people whose analyses lead them to conclude either on philosophical or on scientific grounds that the hypothesis *is* false. Scientifically, one can attack or defend it only by bringing forth empirical evidence about the natural world.

We now need to trace the development of this hypothesis and look at the evidence for it.

Development of the Symbol System Hypothesis

A physical symbol system is an instance of a universal machine. Thus the symbol system hypothesis implies that intelligence will be realized by a universal computer. However, the hypothesis goes far beyond the argument, often made on general grounds of physical determinism, that any computation that is realizable can be realized by a universal machine, provided that it is specified. For it asserts specifically that the intelligent machine is a symbol system, thus making a specific architectural assertion about the nature of intelligent systems. It is important to understand how this additional specificity arose.

Formal Logic. The roots of the hypothesis go back to the program of Frege and of Whitehead and Russell for formalizing logic: capturing the basic conceptual notions of mathematics in logic and putting the notions of proof and deduction on a secure footing. This effort culminated in mathematical logic—our familiar propositional, first-order, and higher-order logics. It developed a characteristic view, often referred to as the "symbol game." Logic, and by incorporation all of mathematics, was a game played with meaningless tokens according to certain purely syntactic rules. All meaning had been purged. One had a mechanical, though permissive (we would now say nondeterministic), system about which various things could be proved. Thus progress was first made by walking away from all that seemed relevant to meaning and human symbols. We could call this the stage of formal symbol manipulation.

This general attitude is well reflected in the development of information theory. It was pointed out time and again that Shannon had defined a system that was useful only for communication and selection, and which had nothing to do with meaning. Regrets were expressed that such a general name as "information theory" had been given to the field, and attempts were made to rechristen it as "the theory of selective information"—to no avail, of course.

Turing Machines and the Digital Computer. The development of the first digital computers and of automata theory, starting with Turing's own work in the '30s, can be treated together. They agree in their view of what is essential. Let us use Turing's own model, for it shows the features well.

A Turing machine consists of two memories: an unbounded tape and a finite state control. The tape holds data, i.e. the famous zeroes and ones. The machine has a very small set of proper operations—read, write, and scan operations—on the tape. The read operation is not a data operation, but provides conditional branching to a control state as a function of the data under the read head. As we all know, this model contains the essentials of all computers, in terms of what they can do, though other computers with different memories and operations might carry out the same computations with different requirements of space and time. In particular, the model of a Turing machine contains within it the notions both of what cannot be computed and of universal machines—computers that can do anything that can be done by any machine.

We should marvel that two of our deepest insights into information processing were achieved in the thirties, before modern computers came into being. It is a tribute to the genius of Alan Turing. It is also a tribute to the development of mathematical logic at the time, and testimony to the depth of computer science's obligation to it. Concurrently with Turing's work appeared the work of the logicians Emil Post and (independently) Alonzo Church. Starting from independent notions of logistic systems (Post productions and recursive functions, respectively) they arrived at analogous results on undecidability and universality—results that were soon shown to imply that all three systems were equivalent. Indeed, the convergence of all these attempts to define the most general class of information processing systems provides some of the force of our conviction that we have captured the essentials of information processing in these models.

In none of these systems is there, on the surface, a concept of the symbol as something that *designates*. The data are regarded as just strings of zeroes and ones—indeed that data be inert is essential to the reduction of computation to physical process. The finite state control system was always viewed as a small controller, and logical games were played to see how small a state system could be used without destroying the universality of the machine. No games, as far as we can tell, were ever played to add new states dynamically to the finite control—to think of the control memory as holding the bulk of the system's knowledge. What was accomplished at this stage was half the principle of interpretation—showing that a machine could be run from a description. Thus, this is the stage of automatic formal symbol manipulation.

The Stored Program Concept. With the development of the second generation of electronic machines in the mid-forties (after the Eniac) came the stored program concept. This was rightfully hailed as a milestone, both conceptually and practically. Programs now can be data, and can be operated on as data. This capability is, of course, already implicit in the model of Turing: the descriptions are on the very same tape as the data. Yet the idea was realized only when machines acquired enough memory to make it practicable to locate actual programs in some internal place. After all, the Eniac had only twenty registers.

The stored program concept embodies the second half of the interpretation principle, the part that says that the system's own data can be interpreted. But it does not yet contain the notion of designation—of the physical relation that underlies meaning.

List Processing. The next step, taken in 1956, was list processing. The contents of the data structures were now symbols, in the sense of our physical symbol system: patterns that designated, that had referents. Lists held addresses which permitted access to other lists—thus the notion of list structures. That this was a new view was demonstrated to us many times in the early days of list processing when colleagues would ask where the data were—that is, which list finally held the collections of bits that were the content of the system. They found it strange that there were no such bits, there were only symbols that designated yet other symbol structures.

List processing is simultaneously three things in the development of computer science. (1) It is the creation of a genuine dynamic memory structure in a machine that had heretofore been perceived as having fixed structure. It added to our ensemble of operations those that built and modified structure in addition to those that replaced and changed content. (2) It was an early demonstration of the basic abstraction that a computer consists of a set of data types and a set of operations proper to these data types, so that a computational system should employ whatever data types are appropriate to the application, independent of the underlying machine. (3) List processing produced a model of designation, thus defining symbol manipulation in the sense in which we use this concept in computer science today.

As often occurs, the practice of the time already anticipated all the elements of list processing: addresses are obviously used to gain access, the drum machines used linked programs (so called one-plus-one addressing), and so on. But the conception of list processing as an abstraction created a new world in which designation and dynamic symbolic structure were the defining characteristics. The embedding of the early list processing systems in languages (the IPLs, LISP) is often decried as having been a barrier to the diffusion of list processing techniques throughout programming practice; but it was the vehicle that held the abstraction together.

LISP. One more step is worth noting: McCarthy's creation of LISP in 1959–60 [McCarthy, 1960]. It completed the act of abstraction, lifting list structures out of their embedding in concrete machines, creating a new formal system with S-expressions, which could be shown to be equivalent to the other universal schemes of computation.

Conclusion. That the concept of the designating symbol and symbol manipulation does not emerge until the mid-fifties does not mean that the earlier steps were either inessential or less important. The total concept is the join of computability, physical realizability (and by multiple technologies), universality, the symbolic representation of processes (i.e. interpretability), and, finally, symbolic structure and designation. Each of the steps provided an essential part of the whole.

The first step in this chain, authored by Turing, is theoretically motivated, but the others all have deep empirical roots. We have been led by the evolution of the computer itself. The stored program principle arose out of the experience with Eniac. List processing arose out of the attempt to construct intelligent programs. It took its cue from the emergence of random access memories, which provided a clear physical realization of a designating symbol in the address. LISP arose out of the evolving experience with list processing.

The Evidence

We come now to the evidence for the hypothesis that physical symbol systems are capable of intelligent action, and that general intelligent action calls for a physical symbol system. The hypothesis is an empirical generalization and not a theorem. We know of no way of demonstrating the connection between symbol systems and intelligence on purely logical grounds. Lacking such a demonstration, we must look at the facts. Our central aim, however, is not to review the evidence in detail, but to use the example before us to illustrate the proposition that computer science is a field of empirical inquiry. Hence, we will only indicate what kinds of evidence there is, and the general nature of the testing process.

The notion of physical symbol system had taken essentially its present form by the middle of the 1950's, and one can date from that time the growth of artificial intelligence as a coherent subfield of computer science. The twenty years of work since then has seen a continuous accumulation of empirical evidence of two main varieties. The first addresses itself to the *sufficiency* of physical symbol systems for producing intelligence, attempting to construct and test specific systems that have such a capability. The second kind of evidence addresses itself to the *necessity* of having a physical symbol system wherever intelligence is exhibited. It starts with Man, the intelligent system best known to us, and attempts to discover whether his cognitive activity can be explained as the working of a physical symbol system. There are other forms of evidence, which we will comment upon briefly later, but these two are the important ones. We will consider them in turn. The first is generally called artificial intelligence, the second, research in cognitive psychology.

Constructing Intelligent Systems. The basic paradigm for the initial testing of the germ theory of disease was: identify a disease; then look for the germ. An analogous paradigm has inspired much of the research in artificial intelligence: identify a task domain calling for intelligence; then construct a program for a digital computer that can handle tasks in that domain. The easy and well-structured tasks were looked at first: puzzles and games, operations research problems of scheduling and allocating resources, simple induction tasks. Scores, if not hundreds, of programs of these kinds have by now been constructed, each capable of some measure of intelligent action in the appropriate domain.

Of course intelligence is not an all-or-none matter, and there has been steady progress toward higher levels of performance in specific domains, as well as toward widening the range of those domains. Early chess programs, for example, were deemed successful if they could play the game legally and with some indication of purpose; a little later, they reached the level of human beginners; within ten or fifteen years, they began to compete with serious amateurs. Progress has been slow (and the total programming effort invested small) but continuous, and the paradigm of construct-and-test proceeds in a regular cycle—the whole research activity mimicking at a macroscopic level the basic generate-and-test cycle of many of the AI programs.

There is a steadily widening area within which intelligent action is attainable. From the original tasks, research has extended to building systems that handle and understand natural language in a variety of ways, systems for interpreting visual scenes, systems for hand-eye coordination, systems that design, systems that write computer programs, systems for speech understanding—the list is, if not endless, at least very long. If there are limits beyond which the hypothesis will not carry us, they have not yet become apparent. Up to the present, the rate of progress has been governed mainly by the rather modest quantity of scientific resources that have been applied and the inevitable requirement of a substantial system-building effort for each new major undertaking.

Much more has been going on, of course, than simply a piling up of examples of intelligent systems adapted to specific task domains. It would be surprising and unappealing if it turned out that the AI programs performing these diverse tasks had nothing in common beyond their being instances of physical symbol systems. Hence, there has been great interest in searching for mechanisms possessed of generality, and for common components among programs performing a variety of tasks. This search carries the theory beyond the initial symbol system hypothesis to a more complete characterization of the particular kinds of symbol systems that are effective in artificial intelligence. In the second section of this paper, we will discuss one example of a hypothesis at this second level of specificity: the heuristic search hypothesis.

The search for generality spawned a series of programs designed to separate out general problem-solving mechanisms from the requirements of particular task domains. The General Problem Solver (GPS) was perhaps the first of these; while among its descendants are such contemporary systems as PLANNER and CONNIVER. The search for common components has led to generalized schemes of representation for goals and plans, methods for constructing discrimination nets, procedures for the control of tree search, pattern-matching mechanisms, and language-parsing systems. Experiments are at present under way to find convenient devices for representing sequences of time and tense, movement, causality and the like. More and more, it becomes possible to assemble large intelligent systems in a modular way from such basic components.

We can gain some perspective on what is going on by turning, again, to the analogy of the germ theory. If the first burst of research stimulated by that theory consisted largely in finding the germ to go with each disease, subsequent effort turned to learning what a germ was—to building on the basic qualitative law a new level of structure. In artificial intelligence, an initial burst of activity aimed at building intelligent programs for a wide variey of almost randomly selected tasks is giving way to more sharply targeted research aimed at understanding the common mechanisms of such systems.

The Modeling of Human Symbolic Behavior. The symbol system hypothesis implies that the symbolic behavior of man arises because he has the characteristics of a physical symbol system. Hence, the results of efforts to model human behavior with symbol systems become an important part of the evidence for the hypothesis, and research in artificial intelligence goes on in close collaboration with research in information processing psychology, as it is usually called.

The search for explanations of man's intelligent behavior in terms of symbol systems has had a large measure of success over the past twenty years; to the point where information processing theory is the leading contemporary point of view in cognitive psychology. Especially in the areas of problem solving, concept attainment, and long-term memory, symbol manipulation models now dominate the scene.

Research in information processing psychology involves two main kinds of empirical activity. The first is the conduct of observations and experiments on human behavior in tasks requiring intelligence. The second, very similar to the parallel activity in artificial intelligence, is the programming of symbol systems to model the observed human behavior. The psychological observations and experiments lead to the formulation of hypotheses about the symbolic processes the subjects are using, and these are an important source of the ideas that go into the construction of the programs. Thus, many of the ideas for the basic mechanisms of GPS were derived from careful analysis of the protocols that human subjects produced while thinking aloud during the performance of a problem-solving task.

The empirical character of computer science is nowhere more evident than in this alliance with psychology. Not only are psychological experiments required to test the veridicality of the simulation models as explanations of the human behavior, but out of the experiments come new ideas for the design and construction of physical symbol systems.

Other Evidence. The principal body of evidence for the symbol system hypothesis that we have not considered is negative evidence: the absence of specific competing hypotheses as to how intelligent activity might be accomplished—whether by man or machine. Most attempts to build such hypotheses have taken place within the field of psychology. Here we have had a continuum of theories from the points of view usually labeled "behaviorism" to those usually labeled "Gestalt theory." Neither of these points of view stands as a real competitor to the symbol system hypothesis, and this for two reasons. First, neither behaviorism nor Gestalt theory has demonstrated, or even shown how to demonstrate, that the explanatory mechanisms it postulates are sufficient to account for intelligent behavior in complex tasks. Second, neither theory has been formulated with anything like the specificity of artificial programs. As a matter of fact, the alternative theories are sufficiently vague so that it is not terribly difficult to give them information processing interpretations, and thereby assimilate them to the symbol system hypothesis.

Conclusion

We have tried to use the example of the Physical Symbol System Hypothesis to illustrate concretely that computer science is a scientific enterprise in the usual meaning of that term: that it develops scientific hypotheses which it then seeks to verify by empirical inquiry. We had a second reason, however, for choosing this particular example to illustrate our point. The Physical Symbol System Hypothesis is itself a substantial scientific hypothesis of the kind that we earlier dubbed "laws of qualitative structure." It represents an important discovery of computer science, which if borne out by the empirical evidence, as in fact appears to be occurring, will have major continuing impact on the field.

We turn now to a second example, the role of search in intelligence. This topic, and the particular hypothesis about it that we shall examine, have also played a central role in computer science, in general, and artificial intelligence, in particular.

II. Heuristic Search

Knowing that physical symbol systems provide the matrix for intelligent action does not tell us how they accomplish this. Our second example of a law of qualitative structure in computer science addresses this latter question, asserting that symbol systems solve problems by using the processes of heuristic search. This generalization, like the previous one, rests on empirical evidence, and has not been derived formally from other premises. However, we shall see in a moment that it does have some logical connection with the symbol system hypothesis, and perhaps we can look forward to formalization of the connection at some time in the future. Until that time arrives, our story must again be one of empirical inquiry. We will describe what is known about heuristic search and review the empirical findings that show how it enables action to be intelligent. We begin by stating this law of qualitative structure, the Heuristic Search Hypothesis.

Heuristic Search Hypothesis. The solutions to problems are represented as symbol structures. A physical symbol system exercises its intelligence in problem solving by search—that is, by generating and progressively modifying symbol structures until it produces a solution structure.

Physical symbol systems must use heuristic search to solve problems because such systems have limited processing resources; in a finite number of steps, and over a finite interval of time, they can execute only a finite number of processes. Of course that is not a very strong limitation, for all universal Turing machines suffer from it. We intend the limitation, however, in a stronger sense: we mean *practically* limited. We can conceive of systems that are not limited in a practical way, but are capable, for example, of searching in parallel the nodes of an exponentially expanding tree at a constant rate for each unit advance in depth. We will not be concerned here with such systems, but with systems whose computing resources are scarce relative to the complexity of the situations with which they are confronted. The restriction will not exclude any real symbol systems, in computer or man, in the context of real tasks. The fact of limited resources allows us, for most purposes, to view a symbol system as though it were a serial, one-process-at-a-time device. If it can accomplish only a small amount of processing in any short time interval, then we might as well regard it as doing things one at a time. Thus "limited resource symbol system" and "serial symbol system" are practically synonymous. The problem of allocating a scarce resource from moment to moment can usually be treated, if the moment is short enough, as a problem of scheduling a serial machine.

Problem Solving

Since ability to solve problems is generally taken as a prime indicator that a system has intelligence, it is natural that much of the history of artificial intelligence is taken up with attempts to build and understand problem-solving systems. Problem solving has been discussed by philosophers and psychologists for two millenia, in discourses dense with the sense of mystery. If you think there is nothing problematic or mysterious about a symbol system solving problems, then you are a child of today, whose views have been formed since mid-century. Plato (and, by his account, Socrates) found difficulty understanding even how problems could be *entertained*, much less how they could be solved. Let me remind you of how he posed the conundrum in the *Meno:*

> Meno: And how will you inquire, Socrates, into that which you know not? What will you put forth as the subject of inquiry? And if you find what you want, how will you ever know that this is what you did not know?

To deal with this puzzle, Plato invented his famous theory of recollection: when you think you are discovering or learning something, you are really just recalling what you already knew in a previous existence. If you find this explanation preposterous, there is a much simpler one available today, based upon our understanding of symbol systems. An approximate statement of it is:

> To state a problem is to designate (1) a *test* for a class of symbol structures (solutions of the problem), and (2) a *generator* of symbol structures (potential solutions). To solve a problem is to generate a structure, using (2), that satisfies the test of (1).

We have a problem if we know what we want to do (the test), and if we don't know immediately how to do it (our generator does not immediately produce a symbol structure satisfying the test). A symbol system can state and solve problems (sometimes) because it can generate and test.

If that is all there is to problem solving, why not simply generate at once an expression that satisfies the test? This is, in fact, what we do when we wish and dream. "If wishes were horses, beggars might ride." But outside the world of dreams, it isn't possible. To know how we would test something, once constructed, does not mean that we know how to construct it—that we have any generator for doing so.

For example, it is well known what it means to "solve" the problem of playing winning chess. A simple test exists for noticing winning positions, the test for checkmate of the enemy King. In the world of dreams one simply generates a strategy that leads to checkmate for all counter strategies of the opponent. Alas, no generator that will do this is known to existing symbol systems (man or machine). Instead, good moves in chess are sought by generating various alternatives, and painstakingly evaluating them with the use of approximate, and often erroneous, measures that are supposed to indicate the likelihood that a particular line of play is on the route to a winning position. Move generators there are; winning move generators there are not.

Before there can be a move generator for a problem, there must be a problem space: a space of symbol structures in which problem situations, including the initial and goal situations, can be represented. Move generators are processes for modifying one situation in the problem space into another. The basic characteristics of physical symbol systems guarantee that they can represent problem spaces and that they possess move generators. How, in any concrete situation they synthesize a problem space and move generators appropriate to that situation is a question that is still very much on the frontier of artificial intelligence research.

The task that a symbol system is faced with, then, when it is presented with a problem and a problem space, is to use its limited processing resources to generate possible solutions, one after another, until it finds one that satisfies the problem-defining test. If the system had some control over the order in which potential solutions were generated, then it would be desirable to arrange this order of generation so that actual solutions would have a high likelihood of appearing early. A symbol system would exhibit intelligence to the extent that it succeeded in doing this. Intelligence for a system with limited processing resources consists in making wise choices of what to do next.

Search in Problem Solving

During the first decade or so of artificial intelligence research, the study of problem solving was almost synonymous with the study of search processes. From our characterization of problems and problem solving, it is easy to see why this was so. In fact, it might be asked whether it could be otherwise. But before we try to answer that question, we must explore further the nature of search processes as it revealed itself during that decade of activity.

Extracting Information from the Problem Space. Consider a set of symbol structures, some small subset of which are solutions to a given problem. Suppose, further, that the solutions are distributed randomly through the entire set. By this we mean that no information exists that would enable any search generator to perform better than a random search. Then no symbol system could exhibit more intelligence (or less intelligence) than any other in solving the problem, although one might experience better luck than another.

A condition, then, for the appearance of intelligence is that the distribution of solutions be not entirely random, that the space of symbol structures exhibit at least some degree of order and pattern. A second condition is that pattern in the space of symbol structures be more or less detectible. A third condition is that the generator of potential solutions be able to behave differentially, depending on what pattern it detected. There must be information in the problem space, and the symbol system must be capable of extracting and using it. Let us look first at a very simple example, where the intelligence is easy to come by.

Consider the problem of solving a simple algebraic equation:

$$AX + B = CX + D$$

The test defines a solution as any expression of the form, $X = E$, such that $AE + B = CE + D$. Now one could use as generator any process that would produce numbers which could then be tested by substituting in the latter equation. We would not call this an intelligent generator.

Alternatively, one could use generators that would make use of the fact that the original equation can be modified—by adding or subtracting equal quantities from both sides, or multiplying or dividing both sides by the same quantity—without changing its solutions. But, of course, we can obtain even more information to guide the generator by comparing the original expression with the form of the solution, and making precisely those changes in the equation that leave its solution unchanged, while at the same time, bringing it into the desired form. Such a generator could notice that there was an unwanted CX on the right-hand side of the original equation, subtract it from both sides and collect terms again. It could then notice that there was an unwanted B on the left-hand side and subtract that. Finally, it could get rid of the unwanted coefficient $(A - C)$ on the left-hand side by dividing.

Thus by this procedure, which now exhibits considerable intelligence, the generator produces successive symbol structures, each obtained by modifying the previous one; and the modifications are aimed at reducing the differences between the form of the input structure and the form of the test expression, while maintaining the other conditions for a solution.

This simple example already illustrates many of the main mechanisms that are used by symbol systems for intelligent problem solving. First, each successive expression is not generated independently, but is produced by modifying one produced previously. Second, the modifications are not haphazard, but depend upon two kinds of information. They depend on information that is constant over this whole class of algebra problems, and that is built into the structure of the generator itself: all modifications of expressions must leave the equation's solution unchanged. They also depend on information that changes at each step: detection of the differences in form that remain between the current expression and the desired expression. In effect, the generator incorporates some of the tests the solution must satisfy, so that expressions that don't meet these tests will never be generated. Using the first kind of information guarantees that only a tiny subset of all possible expressions is actually generated, but without losing the solution expression from this subset. Using the second kind of information arrives at the desired solution by a succession of approximations, employing a simple form of means-ends analysis to give direction to the search.

There is no mystery where the information that guided the search came from. We need not follow Plato in endowing the symbol system with a previous existence in which it already knew the solution. A moderately sophisticated generator-test system did the trick without invoking reincarnation.

Search Trees. The simple algebra problem may seem an unusual, even pathological, example of search. It is certainly not trial-and-error search, for though there were a few trials, there was no error. We are more accustomed to thinking of problem-solving search as generating lushly branching trees of partial solution possibilities which may grow to thousands, or even millions, of branches, before they yield a solution. Thus, if from each expression it produces, the generator creates B new branches, then the tree will grow as B^D, where D is its depth. The tree grown for the algebra problem had the peculiarity that its branchiness, B, equaled unity.

Programs that play chess typically grow broad search trees, amounting in some cases to a million branches or more. (Although this example will serve to illustrate our points about tree search, we should note that the purpose of search in chess is not to generate proposed solutions, but to evaluate (test) them.) One line of research into game-playing programs has been centrally concerned with improving the representation of the chess board, and the processes for making moves on it, so as to speed up search and make it possible to search larger trees. The rationale for this direction, of course, is that the deeper the dynamic search, the more accurate should be the evaluations at the end of it. On the other hand, there is good empirical evidence that the strongest human players, grandmasters, seldom explore trees of more than one hundred branches. This economy is achieved not so much by searching less deeply than do chess-playing programs, but by branching very sparsely and selectively at each node. This is only possible, without causing a deterioration of the evaluations, by having more of the selectivity built into the generator itself, so that it is able to select for generation just those branches that are very likely to yield important relevant information about the position.

The somewhat paradoxical-sounding conclusion to which this discussion leads is that search—successive generation of potentional solution structures—is a fundamental aspect of a symbol system's exercise of intelligence in problem solving but that amount of search is not a measure of the amount of intelligence being exhibited. What makes a problem a problem is not that a large amount of search is required for its solution, but that a large amount *would* be required if a requisite level of intelligence were not applied. When the symbolic system that is endeavoring to solve a problem knows enough about what to do, it simply proceeds directly towards its goal; but whenever its knowledge becomes inadequate, when it enters terra incognita, it is faced with the threat of going through large amounts of search before it finds its way again.

The potential for the exponential explosion of the search tree that is present in every scheme for generating problem solutions warns us against depending on the brute force of computers—even the biggest and fastest computers—as a compensation for the ignorance and unselectivity of their generators. The hope is still periodically ignited in some human breasts that a computer can be found that is fast enough, and that can be programmed cleverly enough, to play good chess by brute-force search. There is nothing known in theory about the game of chess that rules out this possibility. Empirical studies on the management of search in sizable trees with only modest results make this a much less promising direction than it was when chess was first chosen as an appropriate task for artificial intelligence. We must regard this as one of the important empirical findings of research with chess programs.

The Forms of Intelligence. The task of intelligence, then, is to avert the ever-present threat of the exponential explosion of search. How can this be accomplished? The first route, already illustrated by the algebra example, and by chess programs that only generate "plausible" moves for further analysis, is to build selectivity into the generator: to generate only structures that show promise of being solutions or of being along the path toward solutions. The usual consequence of doing this is to decrease the rate of branching, not to prevent it entirely. Ultimate exponential explosion is not avoided—save in exceptionally highly structured situations like the algebra example—but only postponed. Hence, an intelligent system generally needs to supplement the selectivity of its solution generator with other information-using techniques to guide search.

Twenty years of experience with managing tree search in a variety of task environments has produced a small kit of general techniques which is part of the equipment of every researcher in artificial intelligence today. Since these techniques have been described in general works like that of Nilsson [1971], they can be summarized very briefly here.

In serial heuristic search, the basic question always is: what shall be done next? In tree search, that question, in turn, has two components: (1) from what node in the tree shall we search next, and (2) what direction shall we take from that node? Information helpful in answering the first question may be interpreted as measuring the relative distance of different nodes from the goal. Best-first search calls for searching next from the node that appears closest to the goal. Information helpful in answering the second question—in what direction to search—is often obtained, as in the algebra example, by detecting specific differences between the current nodal structure and the goal structure described by the test of a solution, and selecting actions that are relevant to reducing these particular kinds of differences. This is the technique known as means-ends analysis, which plays a central role in the structure of the General Problem Solver.

The importance of empirical studies as a source of general ideas in AI research can be demonstrated clearly by tracing the history, through large numbers of problem solving programs, of these two central ideas: best-first search and means-ends analysis. Rudiments of best-first search were already present, though unnamed, in the Logic Theorist in 1955. The General Problem Solver, embodying means-ends analysis, appeared about 1957—but combined it with modified depth-first search rather than best-first search. Chess programs were generally wedded, for reasons of economy of memory, to depth-first search, supplemented after about 1958 by the powerful alpha beta pruning procedure. Each of these techniques appears to have been reinvented a number of times, and it is hard to find general, task-independent theoretical discussions of problem solving in terms of these concepts until the middle or late 1960's. The amount of formal buttressing they have received from mathematical theory is still miniscule: some theorems about the reduction in search that can be secured from using the alpha-beta heuristic, a couple of theorems (reviewed by Nilsson [1971]) about shortest-path search, and some very recent theorems on best-first search with a probabilistic evaluation function.

"Weak" and "Strong" Methods. The techniques we have been discussing are dedicated to the control of exponential expansion rather than its prevention. For this reason, they have been properly called "weak methods"—methods to be used when the symbol system's knowledge or the amount of structure actually contained in the problem space are inadequate to permit search to be avoided entirely. It is instructive to contrast a highly structured situation, which can be formulated, say, as a linear programming problem, with the less structured situations of combinatorial problems like the traveling salesman problem or scheduling problems. ("Less structured" here refers to the insufficiency or nonexistence of relevant theory about the structure of the problem space.)

In solving linear programming problems, a substantial amount of computation may be required, but the search does not branch. Every step is a step along the way to a solution. In solving combinatorial problems or in proving theorems, tree search can seldom be avoided, and success depends on heuristic search methods of the sort we have been describing.

Not all streams of AI problem-solving research have followed the path we have been outlining. An example of a somewhat different point is provided by the work on theorem-proving systems. Here, ideas imported from mathematics and logic have had a strong influence on the direction of inquiry. For example, the use of heuristics was resisted when properties of completeness could not be proved (a bit ironic, since most interesting mathematical systems are known to be undecidable). Since completeness can seldom be proved for best-first search heuristics, or for many kinds of selective generators, the effect of this requirement was rather inhibiting. When theorem-proving programs were continually incapacitated by the combinatorial explosion of their search trees, thought began to be given to selective heuristics, which in many cases proved to be analogues of heuristics used in general problem-solving programs. The set-of-support heuristic, for example, is a form of working backwards, adapted to the resolution theorem proving environment.

A Summary of the Experience. We have now described the workings of our second law of qualitative structure, which asserts that physical symbol systems solve problems by means of heuristic search. Beyond that, we have examined some subsidiary characteristics of heuristic search, in particular the threat that it always faces of exponential explosion of the search tree, and some of the means it uses to avert that threat. Opinions differ as to how effective heuristic search has been as a problem solving mechanism—the opinions depending on what task domains are considered and what criterion of adequacy is adopted. Success can be guaranteed by setting aspiration levels low—or failure by setting them high. The evidence might be summed up about as follows. Few programs are solving problems at "expert" professional levels. Samuel's checker program and Feigenbaum and Lederberg's DENDRAL are perhaps the best-known exceptions, but one could point also to a number of heuristic search programs for such operations research problem domains as scheduling and integer programming. In a number of domains, programs perform at the level of competent amateurs: chess, some theorem-proving domains, many kinds of games and puzzles. Human levels have not yet been nearly reached by programs that have a complex perceptual "front end": visual scene recognizers, speech understanders, robots that have to maneuver in real space and time. Nevertheless, impressive progress has been made, and a large body of experience assembled about these difficult tasks.

We do not have deep theoretical explanations for the particular pattern of performance that has emerged. On empirical grounds, however, we might draw two conclusions. First, from what has been learned about human expert performance in tasks like chess, it is likely that any system capable of matching that performance will have to have access, in its memories, to very large stores of semantic information. Second, some part of the human superiority in tasks with a large perceptual component can be attributed to the special-purpose built-in parallel processing structure of the human eye and ear.

In any case, the quality of performance must necessarily depend on the characteristics both of the problem domains and of the symbol systems used to tackle them. For most real-life domains in which we are interested, the domain structure has not proved sufficiently simple to yield (so far) theorems about complexity, or to tell us, other than empirically, how large real-world problems are in relation to the abilities of our symbol systems to solve them. That situation may change, but until it does, we must rely upon empirical explorations, using the best problem solvers we know how to build, as a principal source of knowledge about the magnitude and characteristics of problem difficulty. Even in highly structured areas like linear programming, theory has been much more useful in strengthening the heuristics that underlie the most powerful solution algorithms than in providing a deep analysis of complexity.

Intelligence Without Much Search

Our analysis of intelligence equated it with ability to extract and use information about the structure of the problem space, so as to enable a problem solution to be generated as quickly and directly as possible. New directions for improving the problem-solving capabilities of symbol systems can be equated, then, with new ways of extracting and using information. At least three such ways can be identified.

Nonlocal Use of Information. First, it has been noted by several investigators that information gathered in the course of tree search is usually only used *locally*, to help make decisions at the specific node where the information was generated. Information about a chess position, obtained by dynamic analysis of a subtree of continuations, is usually used to evaluate just that position, not to evaluate other positions that may contain many of the same features. Hence, the same facts have to be rediscovered repeatedly at different nodes of the search tree. Simply to take the information out of the context in which it arose and use it generally does not solve the problem, for the information may be valid only in a limited range of contexts. In recent years, a few exploratory efforts have been made to transport information from its context of origin to other appropriate contexts. While it is still too early to evaluate the power of this idea, or even exactly how it is to be achieved, it shows considerable promise. An important line of investigation that Berliner [1975] has been pursuing is to use causal analysis to determine the range over which a particular piece of information is valid. Thus if a weakness in a chess position can be traced back to the move that made it, then the same weakness can be expected in other positions descendant from the same move.

The HEARSAY speech understanding system has taken another approach to making information globally available. That system seeks to recognize speech strings by pursuing a parallel search at a number of different levels: phonemic, lexical, syntactic, and semantic. As each of these searches provides and evaluates hypotheses, it supplies the information it has gained to a common "blackboard" that can be read by all the sources. This shared information can be used, for example, to eliminate hypotheses, or even whole classes of hypotheses, that would otherwise have to be searched by one of the processes. Thus, increasing our ability to use tree-search information nonlocally offers promise for raising the intelligence of problem-solving systems.

Semantic Recognition Systems. A second active possibility for raising intelligence is to supply the symbol system with a rich body of semantic information about the task domain it is dealing with. For example, empirical research on the skill of chess masters shows that a major source of the master's skill is stored information that enables him to recognize a large number of specific features and patterns of features on a chess board, and information that uses this recognition to propose actions appropriate to the features recognized. This general idea has, of course, been incorporated in chess programs almost from the beginning. What is new is the realization of the number of such patterns and associated information that may have to be stored for master-level play: something of the order of 50,000.

The possibility of substituting recognition for search arises because a particular, and especially a rare, pattern can contain an enormous amount of information, provided that it is closely linked to the structure of the problem space. When that structure is "irregular," and not subject to simple mathematical description, then knowledge of a large number of relevant patterns may be the key to intelligent behavior. Whether this is so in any particular task domain is a question more easily settled by empirical investigation than by theory. Our experience with symbol systems richly endowed with semantic information and pattern-recognizing capabilities for accessing it is still extremely limited.

The discussion above refers specifically to semantic information associated with a recognition system. Of course, there is also a whole large area of AI research on semantic information processing and the organization of semantic memories that falls outside the scope of the topics we are discussing in this paper.

Selecting Appropriate Representations. A third line of inquiry is concerned with the possibility that search can be reduced or avoided by selecting an appropriate problem space. A standard example that illustrates this possibility dramatically is the mutilated checkerboard problem. A standard 64 square checkerboard can be covered exactly with 32 tiles, each a 1×2 rectangle covering exactly two squares. Suppose, now, that we cut off squares at two diagonally opposite corners of the checkerboard, leaving a total of 62 squares. Can this mutilated board be covered exactly with 31 tiles? With (literally) heavenly patience, the impossibility of achieving such a covering can be demonstrated by trying all possible arrangements. The alternative, for those with less patience, and more intelligence, is to observe that the two diagonally opposite corners of a checkerboard are of the same color. Hence, the mutilated checkerboard has two less squares of one color than of the other. But each tile covers one square of one color and one square of the other, and any set of tiles must cover the same number of squares of each color. Hence, there is no solution. How can a symbol system discover this simple inductive argument as an alternative to a hopeless attempt to solve the problem by search among all possible coverings? We would award a system that found the solution high marks for intelligence.

Perhaps, however, in posing this problem we are not escaping from search processes. We have simply displaced the search from a space of possible problem solutions to a space of possible representations. In any event, the whole process of moving from one representation to another, and of discovering and evaluating representations, is largely unexplored territory in the domain of problem-solving research. The laws of qualitative structure governing representations remain to be discovered. The search for them is almost sure to receive considerable attention in the coming decade.

Conclusion

That is our account of symbol systems and intelligence. It has been a long road from Plato's *Meno* to the present, but it is perhaps encouraging that most of the progress along that road has been made since the turn of the twentieth century, and a large fraction of it since the midpoint of the century. Thought was still wholly intangible and ineffable until modern formal logic interpreted it as the manipulation of formal tokens. And it seemed still to inhabit mainly the heaven of Platonic ideals, or the equally obscure spaces of the human mind, until computers taught us how symbols could be processed by machines. A.M. Turing, whom we memorialize this morning, made his great contributions at the mid-century crossroads of these developments that led from modern logic to the computer.

Physical Symbol Systems. The study of logic and computers has revealed to us that intelligence resides in physical symbol systems. This is computer sciences's most basic law of qualitative structure.

Symbol systems are collections of patterns and processes, the latter being capable of producing, destroying and modifying the former. The most important properties of patterns is that they can designate objects, processes, or other patterns, and that, when they designate processes, they can be interpreted. Interpretation means carrying out the designated process. The two most significant classes of symbol systems with which we are acquainted are human beings and computers.

Our present understanding of symbol systems grew, as indicated earlier, through a sequence of stages. Formal logic familiarized us with symbols, treated syntactically, as the raw material of thought, and with the idea of manipulating them according to carefully defined formal processes. The Turing machine made the syntactic processing of symbols truly machine-like, and affirmed the potential universality of strictly defined symbol systems. The stored-program concept for computers reaffirmed the interpretability of symbols, already implicit in the Turing machine. List processing brought to the forefront the denotational capacities of symbols, and defined symbol processing in ways that allowed independence from the fixed structure of the underlying physical machine. By 1956 all of these concepts were available, together with hardware for implementing them. The study of the intelligence of symbol systems, the subject of artificial intelligence, could begin.

Heuristic Search. A second law of qualitative structure for AI is that symbol systems solve problems by generating potential solutions and testing them, that is, by searching. Solutions are usually sought by creating symbolic expressions and modifying them sequentially until they satisfy the conditions for a solution. Hence symbol systems solve problems by searching. Since they have finite resources, the search cannot be carried out all at once, but must be sequential. It leaves behind it either a single path from starting point to goal or, if correction and backup are necessary, a whole tree of such paths.

Symbol systems cannot appear intelligent when they are surrounded by pure chaos. They exercise intelligence by extracting information from a problem domain and using that information to guide their search, avoiding wrong turns and circuitous bypaths. The problem domain must contain information, that is, some degree of order and structure, for the method to work. The paradox of the *Meno* is solved by the observation that information may be remembered, but new information may also be extracted from the domain that the symbols designate. In both cases, the ultimate source of the information is the task domain.

The Empirical Base. Artificial intelligence research is concerned with how symbol systems must be organized in order to behave intelligently. Twenty years of work in the area has accumulated a considerable body of knowledge, enough to fill several books (it already has), and most of it in the form of rather concrete experience about the behavior of specific classes of symbol systems in specific task domains. Out of this experience, however, there have also emerged some generalizations, cutting across task domains and systems, about the general characteristics of intelligence and its methods of implementation.

We have tried to state some of these generalizations this morning. They are mostly qualitative rather than mathematical. They have more the flavor of geology or evolutionary biology than the flavor of theoretical physics. They are sufficiently strong to enable us today to design and build moderately intelligent systems for a considerable range of task domains, as well as to gain a rather deep understanding of how human intelligence works in many situations.

What Next? In our account today, we have mentioned open questions as well as settled ones; there are many of both. We see no abatement of the excitement of exploration that has surrounded this field over the past quarter century. Two resource limits will determine the rate of progress over the next such period. One is the amount of computing power that will be available. The second, and probably the more important, is the number of talented young computer scientists who will be attracted to this area of research as the most challenging they can tackle.

A.M. Turing concluded his famous paper on "Computing Machinery and Intelligence" with the words:

> "We can only see a short distance ahead, but we can see plenty there that needs to be done."

Many of the things Turing saw in 1950 that needed to be done have been done, but the agenda is as full as ever. Perhaps we read too much into his simple statement above, but we like to think that in it Turing recognized the fundamental truth that all computer scientists instinctively know. For all physical symbol systems, condemned as we are to serial search of the problem environment, the critical question is always: What to do next?

References

Berliner, H. [1975]. Chess as problem solving: the development of a tactics analyzer. Ph.D. Th., Computer Sci. Dep., Carnegie-Mellon U. (unpublished).

McCarthy, J. [1960]. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM 3,* 4 (April 1960), 184–195.

McCulloch, W.S. [1961]. What is a number, that a man may know it, and a man, that he may know a number. *General Semantics Bulletin* Nos. 26 and 27 (1961), 7–18.

Nilsson, N.J. [1971]. *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, New York.

Turing, A.M. [1950]. Computing machinery and intelligence. *Mind 59* (Oct. 1950), 433–460.

REPORT

COMPUTING AS A DISCIPLINE

The final report of the Task Force on the Core of Computer Science presents a new intellectual framework for the discipline of computing and a new basis for computing curricula. This report has been endorsed and approved for release by the ACM Education Board.

PETER J. DENNING (CHAIRMAN), DOUGLAS E. COMER, DAVID GRIES, MICHAEL C. MULDER, ALLEN TUCKER, A. JOE TURNER, and PAUL R. YOUNG

It is ACM's 42nd year and an old debate continues. Is computer science a science? An engineering discipline? Or merely a technology, an inventor and purveyor of computing commodities? What is the intellectual substance of the discipline? Is it lasting, or will it fade within a generation? Do core curricula in computer science and engineering accurately reflect the field? How can theory and lab work be integrated in a computing curriculum? Do core curricula foster competence in computing?

We project an image of a technology-oriented discipline whose fundamentals are in mathematics and engineering—for example, we represent algorithms as the most basic objects of concern and programming and hardware design as the primary activities. The view that "computer science equals programming" is especially strong in most of our current curricula: the introductory course is programming, the technology is in our core courses, and the science is in our electives. This view blocks progress in reorganizing the curriculum and turns away the best students, who want a greater challenge. It denies a coherent approach to making experimental and theoretical computer science integral and harmonious parts of a curriculum.

Those in the discipline know that computer science encompasses far more than programming—for example, hardware design, system architecture, designing operating system layers, structuring a database for a specific application, and validating models are all part of the discipline, but are not programming. The emphasis on programming arises from our long-standing belief that programming languages are excellent vehicles for gaining access to the rest of the field, a belief that limits our ability to speak about the discipline in terms that reveal its full breadth and richness.

The field has matured enough that it is now possible to describe its intellectual substance in a new and compelling way. This realization arose in discussions among the heads of the Ph.D.-granting departments of computer science and engineering in their meeting in Snowbird, Utah, in July 1984. These and other similar discussions prompted ACM and the IEEE Computer Society to form task forces to create a new approach. In the spring of 1985, ACM President Adele Goldberg and ACM Education Board Chairman Robert Aiken appointed this task force on the core of computer science with the enthusiastic cooperation of the IEEE Computer Society. At the same time, the Computer Society formed a task force on computing laboratories with the enthusiastic cooperation of the ACM.

We hope that the work of the core task force, embodied in this report, will produce benefits beyond the original charter. By identifying a common core of subject matter, we hope to streamline the processes of developing curricula and model programs in the two societies. The report can be the basis for future discussions of computer science and engineering as a profession, stimulate improvements in secondary school courses in computing, and can lead to a greater widespread appreciation of computing as a discipline.

Our goal has been to create a new way of thinking about the field. Hoping to inspire general inquiry into

This article has been condensed from the *Report of the ACM Task Force on the Core of Computer Science.* Copies of the report in its entirety may be ordered, prepaid, from

ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

Please specify order #201880. Prices are $7.00 for ACM members, and $12.00 for nonmembers.

the nature of our discipline, we sought a framework, not a prescription; a guideline, not an instruction. We invite you to adopt this framework and adapt it to your own situation.

We are pleased to present a new intellectual framework for our discipline and a new basis for our curricula.

CHARTER OF THE TASK FORCE

The task force was given three general charges:

1. Present a *description* of computer science that emphasizes fundamental questions and significant accomplishments. The definition should recognize that the field is constantly changing and that what is said is merely a snapshot of an ongoing process of growth.
2. Propose a teaching paradigm for computer science that conforms to traditional scientific standards, emphasizes the development of competence in the field, and harmoniously integrates theory, experimentation, and design.
3. Give a detailed example of an introductory course sequence in computer science based on the curriculum model and the disciplinary description.

We immediately extended our task to encompass both computer science and computer engineering, because we concluded that no fundamental difference exists between the two fields in the core material. The differences are manifested in the way the two disciplines elaborate the core: computer science focuses on analysis and abstraction; computer engineering on abstraction and design. The phrase *discipline of computing* is used here to embrace all of computer science and engineering.

Two important issues are outside the charter of this task force. First, the curriculum recommendations in this report deal only with the introductory course sequence. It does not address the important, larger question of the design of the entire core curriculum, and indeed the suggested introductory course would be meaningless without a new design for the rest of the core. Second, our specification of an introductory course is intended to be an example of an approach to introduce students to the whole discipline in a rigorous and challenging way, an "existence proof" that our definition of computing can be put to work. We leave it to individual departments to apply the framework to develop their own introductory courses that meet local needs.

PARADIGMS FOR THE DISCIPLINE

The three major paradigms, or cultural styles, by which we approach our work provide a context for our definition of the discipline of computing. The first paradigm, *theory*, is rooted in mathematics and consists of four steps followed in the development of a coherent, valid theory:

(1) characterize objects of study (definition);
(2) hypothesize possible relationships among them (theorem);
(3) determine whether the relationships are true (proof);
(4) interpret results.

A mathematician expects to iterate these steps (e.g., when errors or inconsistencies are discovered.

The second paradigm, *abstraction* (modeling), is rooted in the experimental scientific method and consists of four stages that are followed in the investigation of a phenomenon:

(1) form a hypothesis;
(2) construct a model and make a prediction;
(3) design an experiment and collect data;
(4) analyze results.

A scientist expects to iterate these steps (e.g., when a model's predictions disagree with experimental evidence). Even though "modeling" and "experimentation" might be appropriate substitutes, we have chosen the word "abstraction" for this paradigm because this usage is common in the discipline.

The third paradigm, *design*, is rooted in engineering and consists of four steps followed in the construction of a system (or device) to solve a given problem:

(1) state requirements;
(2) state specifications;
(3) design and implement the system;
(4) test the system.

An engineer expects to iterate these steps (e.g., when tests reveal that the latest version of the system does not satisfactorily meet the requirements).

Theory is the bedrock of the mathematical sciences: applied mathematicians share the notion that science advances only on a foundation of sound mathematics. Abstraction (modeling) is the bedrock of the natural sciences: scientists share the notion that scientific progress is achieved primarily by formulating hypotheses and systematically following the modeling process to verify and validate them. Likewise, design is the bedrock of engineering: engineers share the notion that progress is achieved primarily by posing problems and systematically following the design process to construct systems that solve them. Many debates about the relative merits of mathematics, science, and engineering are implicitly based on an assumption that one of the three processes (theory, abstraction, or design) is the most fundamental.

Closer examination, however, reveals that in computing the three processes are so intricately intertwined that it is irrational to say that any one is fundamental. Instances of theory appear at every stage of abstraction and design, instances of modeling at every stage of theory and design, and instances of design at every stage of theory and abstraction.

Despite their inseparability, the three paradigms are distinct from one another because they represent separate areas of competence. Theory is concerned with the ability to describe and prove relationships among objects. Abstraction is concerned with the ability to use those relationships to make predictions that can be compared with the world. Design is concerned with the ability to implement specific instances of those relationships and use them to perform useful actions. Applied mathematicians, computational scientists, and design engineers generally do not have interchangeable skills.

Moreover, in computing we tend to study computational aids that support people engaged in information-transforming processes. On the design side, for example, sophisticated VLSI design and simulation systems enable the efficient and correct design of microcircuitry, and programming environments enable the efficient design of software. On the modeling side, supercomputers evaluate mathematical models and make predictions about the world, and networks help disseminate findings from scientific experiments. On the theory side, computers help prove theorems, check the consistency of specifications, check for counterexamples, and demonstrate test cases.

Computing sits at the crossroads among the central processes of applied mathematics, science, and engineering. The three processes are of equal—and fundamental—importance in the discipline, which is a unique blend of interaction among theory, abstraction, and design. The binding forces are a common interest in experimentation and design as information transformers, a common interest in computational support of the stages of those processes, and a common interest in efficiency.

THE ROLE OF PROGRAMMING

Many activities in computing are not programming—for example, hardware design, system architecture, operating system structure, designing a database application, and validating models—therefore the notion that "computer science equals programming" is misleading. What is the role of programming in the discipline? In the curriculum?

Clearly programming is part of the standard practices of the discipline and every computing major should achieve competence in it. This does not, however, imply that the curriculum should be based on programming or that the introductory courses should be programming courses.

It is also clear that access to the distinctions of any domain is given through language, and that most of the distinctions of computing are embodied in programming notations. Programming languages are useful tools for gaining access to the distinctions of the discipline. We recommend, therefore, that programming be a part of the competence sought by the core curriculum, and that programming languages be treated as useful vehicles for gaining access to important distinctions of computing.

A DESCRIPTION OF COMPUTING

Our description of computing as a discipline consists of four parts: (1) requirements; (2) short definition; (3) division into subareas; and (4) elaboration of subareas. Our presentation consists of four passes, each going to a greater level of detail.

What we say here is merely a snapshot of a changing and dynamic field. We intend this to be a "living definition," that can be revised from time to time to reflect maturity and change in the field. We expect revisions to occur most frequently in the details of the subareas, occasionally in the list of subareas, and rarely in the short definition.

Requirements

There are many possible ways to formulate a definition. We set five requirements for ours:

1. It should be understandable by people outside the field.
2. It should be a rallying point for people inside the field.
3. It should be concrete and specific.
4. It should elucidate the historical roots of the discipline in mathematics, logic, and engineering.
5. It should set forth the fundamental questions and significant accomplishments in each area of the discipline.

In the process of formulating a description, we considered several other previous definitions and concluded that a description meeting these requirements must have several levels of complexity. The other definitions are briefly summarized here.

In 1967, Newell, Perlis, and Simon [5] argued that computer science is the study of computers and the major phenomena that surround them, and that all the common objections to this definition could just as well be used to demonstrate that other sciences are not science. Despite their eloquence, too many people view this as a circular definition that seems flippant to outsiders. It is, however, a good starting point because the definition we present later can be viewed as an enumeration of the major phenomena surrounding computers.

A slightly more elaborate version of this idea was recently used by the Computing Sciences Accreditation Board (CSAB), which said, "Computer science is the body of knowledge concerned with computers and computation. It has theoretical, experimental, and design components and includes (1) theories for understanding computing devices, programs, and systems; (2) experimentation for the development and testing of concepts; (3) design methodology, algorithms, and tools for practical realization; and (4) methods of analysis for verifying that these realizations meet requirements."

A third definition is, "Computer science is the study of knowledge representations and their implementations." This definition suffers from excessive abstraction and few people would agree on the meaning of knowledge representation. A related example that suffers the same fate is, "Computer science is the study of abstraction and the mastering of complexity," a statement that also applies to physics, mathematics, or philosophy.

A final observation comes from Abelson and Sussman, who say, "The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called procedural espistemology—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of 'what is.' Computation provides a framework for dealing precisely with notions of 'how to' [1]."

Short Definition

The discipline of computing is the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, "What can be (efficiently) automated?"

Division into Subareas

We grappled at some length with the question of dividing the discipline into subareas. We began with a preference for a small number of subareas, such as model versus implementation, or algorithm versus machine. However, the various candidates we devised were too abstract, the boundaries between divisions were too fuzzy, and most people would not have identified comfortably with them.

Then we realized that the fundamentals of the discipline are contained in three basic processes—theory, abstraction, and design—that are used by the disciplinary subareas to accomplish their goals. Thus, a description of the discipline's subareas and their relation to these three basic processes would be useful. To qualify as a subarea, a segment of the discipline must satisfy four criteria:

(1) underlying unity of subject matter;
(2) substantial theoretical component;
(3) significant abstractions;
(4) important design and implementation issues.

Moreover, we felt that each subarea should be identified with a research community, or set of related communities, that sustains its own literature.

Theory includes the processes for developing the underlying mathematics of the subarea. These processes are supported by theory from other areas. For example, the subarea of algorithms and data structures contains complexity theory and is supported by graph theory. Abstraction deals with modeling potential implementations. These models suppress detail while retaining essential features; they are amenable to analysis and provide means for calculating predictions of the modeled system's behavior. Design deals with the process of specifying a problem, transforming the problem statement into a design specification, and repeatedly inventing and investigating alternative solutions until a reliable, maintainable, documented, and tested design that meets cost criteria is achieved.

We discerned nine subareas that cover the field:

1. Algorithms and data structures
2. Programming languages
3. Architecture
4. Numerical and symbolic computation
5. Operating systems
6. Software methodology and engineering
7. Database and information retrieval systems
8. Artificial intelligence and robotics
9. Human–computer communication

Elaboration of Subareas

To present the content of the subareas, we found it useful to think of a 9 × 3 matrix, as shown in Figure 1. Each row is associated with a subarea, and theory, abstraction, and design each define a column.

Each square of the matrix will be filled in with specific statements about that subarea component; these statements will describe issues of concern and significant accomplishments.

Certain affinity groups in which there is scientific literature are not shown as subareas because they are basic concerns throughout the discipline. For example, parallelism surfaces in all subareas (there are parallel algorithms, parallel languages, parallel architectures, etc.) and in theory, abstraction, and design. Similar conclusions hold for security, reliability, and performance evaluation.

Computer scientists will tend to associate with the first two columns of the matrix, and computer engineers with the last two. The full description of computing, as specified here, is given in the appendix.

CURRICULUM MODEL

Competence in the Discipline

The goal of education is to develop competence in a domain. Competence, the capability for effective action,

	Theory	Abstraction	Design
1 Algorithms and data structures			
2 Programming languages			
3 Architecture			
4 Numerical and symbolic computation			
5 Operating systems			
6 Software methodology and engineering			
7 Databases and information retrieval			
8 Artificial intelligence and robotics			
9 Human–computer communication			

FIGURE 1. Definition Matrix for the Computing Discipline

is an assessment of individual performance against the standard practices of the field. The criteria for assessment are grounded in the history of the field. The educational process that leads to competence has five steps: (1) motivate the domain; (2) demonstrate what can be accomplished in the domain; (3) expose the distinctions of the domain; (4) ground the distinctions in history; and (5) practice the distinctions [4].

This model has interesting implications for curriculum design. The first question it leads to is, In what areas of computing must majors be competent? There are two broad areas of competence:

1. *Discipline-Oriented Thinking*: The ability to invent new distinctions in the field, leading to new modes of action and new tools that make those distinctions available for others to use.
2. *Tool Use*: The ability to use the tools of the field for effective action in other domains.

We suggest that discipline-oriented thinking is the primary goal of a curriculum for computing majors, and that majors must be familiar enough with the tools to work effectively with people in other disciplines to help design modes of effective action in those disciplines.

The inquiry into competence reveals a number of areas where current core curricula in computing is inadequate. For example, the historical context of the computing field is often deemphasized, leaving many graduates ignorant of computing history and destined to repeat its mistakes. Many computing graduates wind up in business data processing, a domain in which most computing curricula do not seek to develop competence; whether computing departments or business departments should develop that competence is an old controversy. Discipline-oriented thinking must be based on solid mathematical foundations, yet theory is not an integral part of most computing curricula. The standard practices of the computing field include setting up and conducting experiments, contributing to team projects, and interacting with other disciplines to support their interests in effective use of computing, but most curricula neglect laboratory exercises, team projects, or interdisciplinary studies.

The question of what results should be achieved by computing curricula has not been explored thoroughly in past discussions, and we will not attempt a thorough analysis here. We do strongly recommend that this question be among the first considered in the design of new core curricula for computing.

Lifelong Learning

The curriculum should be designed to develop an appreciation for learning which graduates will carry with them throughout their careers. Many courses are designed with a paradigm that presents "answers" in a lecture format, rather than focusing on the process of questioning that underlies all learning. We recommend that the follow-on committee consider other teaching paradigms which involve processes of inquiry, an orientation to using the computing literature, and the development of a commitment to a lifelong process of learning.

INTRODUCTORY SEQUENCE

In this curriculum model, the motivation and demonstration of the domain must precede instruction and practice in the domain. The purpose of the introductory course sequence is precisely this. The principal areas of computing—in which majors must develop competence—must be presented to students with sufficient depth and rigor that they can appreciate the power of the areas and the benefits from achieving competence in them. The remainder of the curriculum must be carefully designed to systematically explore those areas, exposing new concepts and distinctions, and giving students practice in them.

We therefore recommend that the introductory course consist of regular lectures and a closely coordinated weekly laboratory. The lectures should emphasize fundamentals; the laboratories technology and know-how.

This model is traditional in the physical sciences and engineering: lectures emphasize enduring principles and concepts while laboratories emphasize the transient material and skills relating to the current technology. For example, lectures would discuss the design and analysis of algorithms, or the organization of network protocols in functional layers. In the corresponding laboratory sessions, students would write programs for algorithms analyzed in lecture and measure their running times, or instal and test network interfaces and measure their packet throughputs.

Within this recommendation, the first courses in computer science would not only introduce programming, algorithms, and data structures, but introduce material from all the other subdisciplines as well. Mathematics and other theory would be well integrated into the lectures at appropriate points.

We recommend that the introductory course contain a rigorous, challenging survey of the whole discipline. The physics model, exemplified by the Feynman Lectures in Physics, is a paradigm for the introductory course we envisage.

We emphasize that simply redesigning the introductory course sequence following this recommendation without redesigning the entire undergraduate curriculum would be a serious mistake. The experience of physics departments contains many lessons for computing departments in this regard.

Prerequisites

We assume that computing majors have a modest background in programming in some language and some experience with computer-based tools such as word processors, spreadsheets, and databases. Given the widening use of computers in high schools and at home, it might seem that universities could assume that most incoming students have such a background and provide a "remedial" course in programming for the others. We have found, however, that the assumption of adequate high school preparation in programming is quite controversial and there is evidence that adequate preparation is rare. We therefore recommend that computing departments offer an introduction to programming and computer tools that would be a prerequisite (or corequisite) for the introductory courses. We further recommend that departments provide an advanced placement procedure so that students with adequate high school preparation can bypass this course.

Formal prerequisites and corequisites in mathematics are more difficult to state and will depend on local circumstances. However, accrediting boards in computing require considerable mathematics, including discrete mathematics, differential and integral calculus, and probability and statistics. These requirements are often exceeded in the better undergraduate programs. In our description of a beginning computing curriculum, we have spelled out in some detail what mathematics is applicable in each of the nine identified areas of computing. Where possible we have displayed the required mathematical background for each of the teaching modules we describe. This will allow individual departments to synchronize their own mathematical requirements and courses with the material in the modules. In some cases it may be appropriate to introduce appropriate underlying mathematical topics as needed for the development of particular topics in computing. In general, we recommend that students see applications of relevant mathematics as early as possible in their computing studies.

Modular Organization

The introductory sequence should bring out the underlying unity of the field and should flow from topic to topic in a pedagogically natural way. It would therefore be inadequate to organize the course as a sequence of nine sections, one for each of the subareas; such a mapping would appear to be a hodge-podge, with difficult transitions between sections. An ordering of topics that meet these requirements is:

Fundamental algorithm concepts
Computer organization ("von Neumann")
Mathematical programming
Data structures and abstraction
Limits of computability
Operating systems and security
Distributed computing and networks
Models in artificial intelligence
File and database systems
Parallel computation
Human interface

We have grouped the topics into 11 modules. Each module includes challenging material representative of the subject matter without becoming a superficial survey of every aspect or topic. Each module draws material from several squares of the definition matrix as appropriate. As a result, many modules will not correspond one-to-one with rows of the definition matrix. For example, the first module in our example course is entitled Fundamental Algorithm Concepts. It covers the role of formalism and theory, methods in programming, programming concepts, efficiency, and specific algorithms, draws information from the first, second, fourth, and sixth rows of the definition matrix and deals only with sequential algorithms. Later modules, on Distributed Computing and Networks, and on Parallel Computation, extend the material in the first module and draw new material from the third and fifth rows of the definition matrix.

As a general approach, each module contains lectures that cover the required theory and most abstractions. Theory is generally not introduced until it is needed. Each module is closely coupled with laboratory sessions, and the nature of the laboratory assignments is included with the module specifications. Our specification is drawn up for a three-semester course sequence containing 42 lectures and 35 scheduled laboratory sessions per semester. Our specification is not included here, but is in the full report.

We reemphasize that this specification is intended only to be an example of a mapping from the disciplinary description to an introductory course sequence, not a prescription for all introductory courses. Other approaches are exemplified by existing introductory curricula at selected colleges and universities.

LABORATORIES

We have described a curriculum that separates principles from technology while maintaining coherence between the two. We have recommended that lectures deal with principles and laboratories with technology, with the two being closely coordinated.

The laboratories serve three purposes:

1. Laboratories should demonstrate how principles covered in the lectures apply to the design, implementation, and testing of practical software and hardware. They should provide concrete experiences that help students understand abstract concepts. These experiences are essential to sharpen students' intuition about practical computing, and to emphasize the intellectual effort in building correct, efficient computer programs and systems.
2. Laboratories should emphasize processes leading to good computing know-how. They should emphasize programming, not programs; laboratory techniques; understanding of hardware capabilities; correct use of software tools; correct use of documentation; and proper documentation of experiments and projects. Many software tools will be required on host computers to assist in constructing, controlling, and monitoring experiments on attached subsystems; the laboratory should teach proper use of these tools:
3. Laboratories should introduce experimental methods, including use and design of experiments, software and hardware monitors, statistical analysis of results, and proper presentation of findings. Students should learn to distinguish careful experiments from casual observations.

To meet these goals, laboratory work should be carefully planned and supervised. Students should attend labs at specified times, nominally three hours per week. Lab assignments should be planned, and written descriptions of the purposes and methodology of each experiment should be given to the students. The depth of description should be commensurate with students' prior lab experience: more detail is required in early laboratories. Lab assignments should be carried out under the guidance of a lab instructor who ensures that each student follows correct methodology.

The labs associated with the introductory courses will require close supervision and should contain well-planned activities. This implies that more staff will be required per student for these laboratories than for more advanced ones.

The lab problems should be coordinated with material in the lecture parts of the course. Individual lab problems in general will deal with combinations of hardware and software. Some lab assignments emphasize technologies and tools that ease the software development process. Others emphasize analyzing and measuring existing software or comparing known algorithms. Others emphasize program development based on principles learned in class.

Laboratory assignments should be self-contained in the sense that an average student should be able to complete the work in the time allocated. Laboratory assignments should encourage students to discover and learn things for themselves. Students should be required to maintain a proper lab book documenting experiments, observations, and data. Students should also be required to maintain their software and to build libraries that can be used in later lab projects.

We expect that, in labs as in lectures, students will be assigned homework that will require using computers outside the supervised realm of a laboratory. In other words, organized laboratory sessions will supplement, not replace, the usual programming and other written assignments.

In a substantial number of labs dealing with program development, the assignment should be to modify or complete an existing program supplied by the instructor. This forces the student to read well-written programs, provides experience with integration of software, and results in a larger and more satisfying program for the student.

Computing technology constantly changes. It is difficult, therefore, to give a detailed specification of the hardware systems, software systems, instruments, and tools that ought to be in a laboratory. The choice of equipment and staffing in laboratories should be guided by the following principles:

1. Laboratories should be equipped with up-to-date systems and languages. Programming languages have a significant effect on shaping a student's view of computing. Laboratories should deploy systems that encourage good habits in students; it is especially important to avoid outdated systems (hardware and software) in core courses.
2. Hardware and software must be fully maintained. Malfunctioning equipment will frustrate students and interfere with learning. Appropriate staff must be available to maintain the hardware and software used in the lab. The situation is analogous to laboratories in other disciplines.
3. Full functionality is important. (This includes adequate response time on shared systems.) Restricting students to small subsets of a language or system may be useful in initial contacts, but the restrictions should be lifted as the students progress.
4. Good programming tools are needed. Compilers get a lot of attention, but other programming tools are used as often. In UNIX systems, for example, students should use editors like emacs and learn to use tools like the shell, grep, awk, and make. Storage and processing facilities must be sufficient to make such tools available for use in the lab.
5. Adequate support for hardware and instrumentation must be provided. Some projects may require students to connect hardware units together, take measurements of signals, monitor data paths, and the like. A sufficient supply of small parts, connectors, cables, monitoring devices, and test instruments must be available.

The IEEE Computer Society Task Force on Goal Oriented Laboratory Development has studied this subject in depth. Their report includes a discussion of the resources (i.e., staff and facilities) needed for laboratories at all levels of the curriculum.

ACCREDITATION

This work has been conducted with the intent that example courses be consistent with current guidelines of the Computing Sciences Accreditation Board (CSAB). The details of the mapping of this content to CSAB guidelines does not fall within the scope of this committee.

CONCLUSION

This report has been designed to provoke new thinking about computing as a discipline by exhibiting the discipline's content in a way that emphasizes the fundamental concepts, principles, and distinctions. It has also suggested a redesign of the core curriculum according to an education model used in other disciplines: demonstrating the existence of useful distinctions followed by practice that develops competence. The method is illustrated by a rigorous introductory course that puts the concepts and principles into the lectures and technology into closely coordinated laboratories.

A department cannot simply replace its current introductory sequence with the new one; it must redesign the curriculum so that the new introduction is part of a coherent whole. For this reason, we recommend that the ACM establish a follow-on committee to complete the redesign of the core curriculum.

Many practical problems must be dealt with before a new curriculum model can become part of the field.

For example,

1. Faculties will need to redesign their curricula based on a new conceptual formulation.
2. No textbooks or educational materials based on the framework proposed here are currently available.
3. Most departments have inadequate laboratories, facilities, and materials for the educational task suggested here.
4. Teaching assistants and faculty are not familiar with the new view.
5. Good high school preparation in computing is rare.

We recognize that many of our recommendations are challenging and will require substantial work to implement. We are convinced that the improvements in computing education from the proposals here are worth the effort, and invite you to join us in achieving them.

APPENDIX

A DEFINITION OF COMPUTING AS A DISCIPLINE

Computer science and engineering is the systematic study of algorithmic processes—their theory, analysis, design, efficiency, implementation, and application—that describe and transform information. The fundamental question underlying all of computing is, What can be (efficiently) automated [2, 3]. This discipline was born in the early 1940s with the joining together of algorithm theory, mathematical logic, and the invention of the stored-program electronic computer.

The roots of computing extend deeply into mathematics and engineering. Mathematics imparts analysis to the field; engineering imparts design. The discipline embraces its own theory, experimental method, and engineering, in contrast with most physical sciences, which are separate from the engineering disciplines that apply their findings (e.g., chemistry and chemical engineering principles). The science and engineering are inseparable because of the fundamental interplay between the scientific and engineering paradigms within the discipline.

For several thousand years, calculation has been a principal concern of mathematics. Many models of physical phenomena have been used to derive equations whose solutions yield predictions of those phenomena—for example, calculations of orbital trajectories, weather forecasts, and fluid flows. Many general methods for solving such equations have been devised—for example, algorithms for systems of linear equations, differential equations, and integrating functions. For almost the same period, calculations that aid in the design of mechanical systems have been a principal concern of engineering. Examples include algorithms for evaluating stresses in static objects, calculating momenta of moving objects, and measuring distances much larger or smaller than our immediate perception.

One product of the long interaction between engineering and mathematics has been mechanical aids for calculating. Some surveyors' and navigators' instruments date back a thousand years. Pascal and Leibniz built arithmetic calculators in the middle 1600s. In the 1830s, Babbage conceived of an "analytical engine" that could mechanically and without error evaluate logarithms, trigonometric functions, and other general arithmetic functions. His machine, never completed, served as an inspiration for later work. In the 1920s, Bush constructed an electronic analog computer for solving general systems of differential equations. In the same period, electromechanical calculating machines capable of addition, subtraction, multiplication, division, and square root computation became available. The electronic flip-flop provided a natural bridge from these machines to digital versions with no moving parts.

Logic is a branch of mathematics concerned with criteria of validity of inference and formal principles of reasoning. Since the days of Euclid, it has been a tool for rigorous mathematical and scientific argument. In the 19th century a search began for a universal system of logic that would be free of the incompletenesses observed in known deductive systems. In a complete system, it would be possible to determine mechanically whether any given statement is either true or false. In 1931, Gödel published his "incompleteness theorem," showing that there is no such system. In the late 1930s, Turing explored the idea of a universal computer that could simulate any step-by-step procedure of any other computing machine. His findings were similar to Gödel's: some well-defined problems cannot be solved by any mechanical procedure. Logic is important not only because of its deep insight into the limits of automatic calculation, but also because of its insight that strings of symbols, perhaps encoded as numbers, can be interpreted both as data and as programs.

This insight is the key idea that distinguishes the stored program computer from calculating machines. The steps of the algorithm are encoded in a machine representation and stored in the memory for later decoding and execution by the processor. The machine code can be derived mechanically from a higher-level symbolic form, the programming language.

It is the explicit and intricate intertwining of the ancient threads of calculation and logical symbol manipulation, together with the modern threads of electronics and electronic representation of information, that gave birth to the discipline of computing.

We identified nine subareas of computing:

1. Algorithms and data structures
2. Programming languages
3. Architecture
4. Numerical and symbolic computation

5. Operating systems
6. Software methodology and engineering
7. Databases and information retrieval
8. Artificial intelligence and robotics
9. Human-Computer communication

Each has an underlying unity of subject matter, a substantial theoretical component, significant abstractions, and substantial design and implementation issues. Theory deals with the underlying mathematical development of the subarea and includes supporting theory such as graph theory, combinatorics, or formal languages. Abstraction (or modeling) deals with models of potential implementations; the models suppress detail, while retaining essential features, and provide means for predicting future behavior. Design deals with the process of specifying a problem, deriving requirements and specifications, iterating and testing prototypes, and implementing a system. Design includes the experimental method, which in computing comes in several styles: measuring programs and systems, validating hypotheses, and prototyping to extend abstractions to practice.

Although software methodology is essentially concerned with design, it also contains substantial elements of theory and abstraction. For this reason, we have identified it as a subarea. On the other hand, parallel and distributed computation are issues that pervade all the subareas and all their components (theory, abstraction, and design); they have been identified neither as subareas nor as subarea components.

The subsequent numbered sections provide the details of each subarea in three parts—theory, abstraction, and design. The boundaries between theory and abstraction, and between abstraction and design, are necessarily fuzzy; it is a matter of personal taste where some of the items go.

Our intention is to provide a guide to the discipline by showing its main features, not a detailed map. It is important to remember that this guide to the discipline is not a plan for a course or a curriculum; it is merely a framework in which a curriculum can be designed. It is also important to remember that this guide to the discipline is a snapshot of an organism undergoing constant change. It will require reevaluation and revision at regular intervals.

1. ALGORITHMS AND DATA STRUCTURES

This area deals with specific classes of problems and their efficient solutions. Fundamental questions include: For given classes of problems, what are the best algorithms? How much storage and time do they require? What is the tradeoff between space and time? What is the best way to access the data? What is the worst case of the best algorithms? How well do algorithms behave on average? How general are algorithms—i.e., what classes of problems can be dealt with by similar methods?

1.1 Theory

Major elements of theory in the area of algorithms and data structures are:

1. Computability theory, which defines what machines can and cannot do.
2. Computational complexity theory, which tells how to measure the time and space requirements of computable functions and relates a problem's size with the best- or worst-case performance of algorithms that solve that problem, and provides methods for proving lower bounds on any possible solution to a problem.
3. Time and space bounds for algorithms and classes of algorithms.
4. Levels of intractability: for example, classes of problems solvable deterministically in polynomially bounded time (P-problems); those solvable nondeterministically in polynomially bounded time (NP-problems); and those solvable efficiently by parallel machines (NC-problems).
5. Parallel computation, lower bounds, and mappings from dataflow requirements of algorithms into communication paths of machines.
6. Probabilistic algorithms, which give results correct with sufficiently high probabilities much more efficiently (in time and space) than determinate algorithms that guarantee their results. Monte Carlo methods.
7. Cryptography.
8. The supporting areas of graph theory, recursive functions, recurrence relations, combinatorics, calculus, induction, predicate and temporal logic, semantics, probability, and statistics.

1.2 Abstraction

Major elements of abstraction in the area of algorithms and data structures are

1. Efficient, optimal algorithms for important classes of problems and analyses for best, worst, and average performance.
2. Classifications of the effects of control and data structure on time and space requirements for various classes of problems.
3. Important classes of techniques such as divide-and-conquer, Greedy algorithms, dynamic programming, finite state machine interpreters, and stack machine interpreters.
4. Parallel and distributed algorithms; methods of partitioning problems into tasks that can be executed in separate processors.

1.3 Design

Major elements of design and experimentation in the area of algorithms and data structures are:

1. Selection, implementation, and testing of algorithms for important classes of problems such as searching,

sorting, random-number generation, and textual pattern matching.
2. Implementation and testing of general methods applicable across many classes of problems, such as hashing, graphs, and trees.
3. Implementation and testing of distributed algorithms such as network protocols, distributed data updates, semaphores, deadlock detectors, and synchronization methods.
4. Implementation and testing of storage managers such as garbage collection, buddy system, lists, tables, and paging.
5. Extensive experimental testing of heuristic algorithms for combinatorial problems.
6. Cryptographic protocols that permit secure authentication and secret communication.

2. PROGRAMMING LANGUAGES

This area deals with notations for virtual machines that execute algorithms, with notations for algorithms and data, and with efficient translations from high-level languages into machine codes. Fundamental questions include: What are possible organizations of the virtual machine presented by the language (data types, operations, control structures, mechanisms for introducing new types and operations)? How are these abstractions implemented on computers? What notation (syntax) can be used effectively and efficiently to specify what the computer should do?

2.1 Theory

Major elements of theory in the area of programming languages are:

1. Formal languages and automata, including theories of parsing and language translation.
2. Turing machines (base for procedural languages), Post Systems (base for string processing languages), λ-calculus (base for functional languages).
3. Formal semantics: methods for defining mathematical models of computers and the relationships among the models, language syntax, and implementation. Primary methods include denotational, algebraic, operational, and axiomatic semantics.
4. As supporting areas: predicate logic, temporal logic, modern algebra and mathematical induction.

2.2 Abstraction

Major elements of abstraction in the area of programming languages include:

1. Classification of languages based on their syntactic and dynamic semantic models; e.g., static typing, dynamic typing, functional, procedural, object-oriented, logic, specification, message passing, and dataflow.
2. Classification of languages according to intended application area; e.g., business data processing, simulation, list processing, and graphics.
3. Classification of major syntactic and semantic models for program structure; e.g., procedure hierarchies, functional composition, abstract data types, and communicating parallel processes.
4. Abstract implementation models for each major type of language.
5. Methods for parsing, compiling, interpretation, and code optimization.
6. Methods for automatic generation of parsers, scanners, compiler components, and compilers.

2.3 Design

Major elements of design and experimentation in the area of programming languages are:

1. Specific languages that bring together a particular abstract machine (semantics) and syntax to form a coherent implementable whole. Examples: procedural (COBOL, FORTRAN, ALGOL, Pascal, Ada, C), functional (LISP), dataflow (SISAL, VAL), object-oriented (Smalltalk, CLU), logic (Prolog), strings (SNOBOL), and concurrency (CSP, Occam, Concurrent Pascal, Modula 2).
2. Specific implementation methods for particular classes of languages: run-time models, static and dynamic execution methods, typing checking, storage and register allocation, compilers, cross compilers, and interpreters, systems for finding parallelism in programs.
3. Programming environments.
4. Parser and scanner generators (e.g., YACC, LEX), compiler generators.
5. Programs for syntactic and semantic error checking, profiling, debugging, and tracing.
6. Applications of programming-language methods to document-processing functions such as creating tables, graphs, chemical formulas, spreadsheets equations, input and output, and data handling. Other applications such as statistical processing.

3. ARCHITECTURE

This area deals with methods of organizing hardware (and associated software) into efficient, reliable systems. Fundamental questions include: What are good methods of implementing processors, memory, and communication in a machine? How do we design and control large computational systems and convincingly demonstrate that they work as intended despite errors and failures? What types of architectures can efficiently incorporate many processing elements that can work concurrently on a computation? How do we measure performance?

3.1 Theory

Major elements of theory in the area of architecture are:

1. Boolean algebra.
2. Switching theory.

3. Coding theory.
4. Finite state machine theory.
5. The supporting areas of statistics, probability, queueing, reliability theory, discrete mathematics, number theory, and arithmetic in different number systems.

3.2 Abstraction

Major elements of abstraction in the area of architecture are:

1. Finite state machine and Boolean algebraic models of circuits that relate function to behavior.
2. Other general methods of synthesizing systems from basic components.
3. Models of circuits and finite state machines for computing arithmetic functions over finite fields.
4. Models for data path and control structures.
5. Optimizing instruction sets for various models and workloads.
6. Hardware reliability: redundancy, error detection, recovery, and testing.
7. Space, time, and organizational tradeoffs in the design of VLSI devices.
8. Organization of machines for various computational models: sequential, dataflow, list processing, array processing, vector processing, and message-passing.
9. Identification of design levels; e.g., configuration, program, instruction set, register, and gate.

3.3 Design

Major elements of design and experimentation in the area of architecture are:

1. Hardware units for fast computation; e.g., arithmetic function units, cache.
2. The so-called von Neumann machine (the single-instruction sequence stored program computer); RISC and CISC implementations.
3. Efficient methods of storing and recording information, and detecting and correcting errors.
4. Specific approaches to responding to errors: recovery, diagnostics, reconfiguration, and backup procedures.
5. Computer aided design (CAD) systems and logic simulations for the design of VLSI circuits. Production programs for layout, fault diagnosis. Silicon compilers.
6. Implementing machines in various computational models; e.g., dataflow, tree, LISP, hypercube, vector, and multiprocessor.
7. Supercomputers, such as the Cray and Cyber machines.

4. NUMERICAL AND SYMBOLIC COMPUTATION

This area deals with general methods of efficiently and accurately solving equations resulting from mathematical models of systems. Fundamental questions include: How can we accurately approximate continuous or infinite processes by finite discrete processes? How do we cope with the errors arising from these approximations? How rapidly can a given class of equations be solved for a given level of accuracy? How can symbolic manipulations on equations, such as integration, differentiation, and reduction to minimal terms, be carried out? How can the answers to these questions be incorporated into efficient, reliable, high-quality mathematical software packages?

4.1 Theory

Major elements of theory in the area of numerical and symbolic computation are:

1. Number theory.
2. Linear algebra.
3. Numerical analysis.
4. Nonlinear dynamics.
5. The supporting areas of calculus, real analysis, complex analysis, and algebra.

4.2 Abstraction

Major elements of abstraction in the area of numerical and symbolic computation are:

1. Formulations of physical problems as models in continuous (and sometimes discrete) mathematics.
2. Discrete approximations to continuous problems. In this context, backward error analysis, error propagation and stability in the solution of linear and nonlinear systems. Special methods in special cases, such as Fast Fourier Transform and Poisson solvers.
3. The finite element model for a large class of problems specifiable by regular meshes and boundary values. Associated iterative methods and convergence theory: direct, implicit, multigrids, rates of convergence. Parallel solution methods. Automatic grid refinement during numerical integration.
4. Symbolic integration and differentiation.

4.3 Design

Major elements of design and experimentation in the area of numerical and symbolic computation are:

1. High-level problem formulation systems such as CHEM and WEB.
2. Specific libraries and packages for linear algebra, ordinary differential equations, statistics, nonlinear equations, and optimizations; e.g., LINPACK, EISPACK, ELLPACK.
3. Methods of mapping finite element algorithms to specific architectures—e.g., multigrids on hypercubes.
4. Symbolic manipulators, such as MACSYMA and REDUCE, capable of powerful and nonobvious manipulations, notably differentiations, integrations, and reductions of expressions to minimal terms.

5. OPERATING SYSTEMS

This area deals with control mechanisms that allow multiple resources to be efficiently coordinated in the execution of programs. Fundamental questions include: What are the visible objects and permissible operations at each level in the operation of a computer system? For each class of resource (objects visible at some level), what is a minimal set of operations that permit their effective use? How can interfaces be organized so that users deal only with abstract versions of resources and not with physical details of hardware? What are effective control strategies for job scheduling, memory management, communications, access to software resources, communication among concurrent tasks, reliability, and security? What are the principles by which systems can be extended in function by repeated application of a small number of construction rules? How should distributed computations be organized so that many autonomous machines connected by a communication network can participate in a computation, with the details of network protocols, host locations, bandwidths, and resource naming being mostly invisible?

5.1 Theory

Major elements of theory in the area of operating systems are:

1. Concurrency theory: synchronization, determinacy, and deadlocks.
2. Scheduling theory, especially processor scheduling.
3. Program behavior and memory management theory, including optimal policies for storage allocation.
4. Performance modeling and analysis.
5. The supporting areas of bin packing, probability, queueing theory, queueing networks, communication and information theory, temporal logic, and cryptography.

5.2 Abstraction

Major elements of abstraction in the area of operating systems are:

1. Abstraction principles that permit users to operate on idealized versions of resources without concern for physical details (e.g., process rather than processor, virtual memory rather than main-secondary hierarchy, files rather than disks).
2. Binding of objects perceived at the user interface to internal computational structures.
3. Models for important subproblems such as process management, memory management, job scheduling, secondary storage management, and performance analysis.
4. Models for distributed computation; e.g., clients and servers, cooperating sequential processes, message-passing, and remote procedure calls.
5. Models for secure computing; e.g., access controls, authentication, and communication.
6. Networking, including layered protocols, naming, remote resource usage, help services, and local network protocols such as token-passing and shared buses.

5.3 Design

Major elements of design and experimentation in the area of operating systems are:

1. Prototypes of time sharing systems, automatic storage allocators, multilevel schedulers, memory managers, hierarchical file systems and other important system components that have served as bases for commercial systems.
2. Techniques for building operating systems such as UNIX, Multics, Mach, VMS, and MS-DOS.
3. Techniques for building libraries of utilities; e.g., editors, document formatters, compilers, linkers, and device drivers.
4. Files and file systems.
5. Queueing network modeling and simulation packages to evaluate performance of real systems.
6. Network architectures such as ethernet, FDDI, token ring nets, SNA, and DECNET.
7. Protocol techniques embodied in the Department of Defense protocol suite (TCP/IP), virtual circuit protocols, internet, real time conferencing, and X.25.

6. SOFTWARE METHODOLOGY AND ENGINEERING

This area deals with the design of programs and large software systems that meet specifications and are safe, secure, reliable, and dependable. Fundamental questions include: What are the principles behind the development of programs and programming systems? How does one prove that a program or system meets its specifications? How does one develop specifications that do not omit important cases and can be analyzed for safety? How do software systems evolve through different generations? How can software be designed for understandability and modifiability?

6.1 Theory

Major elements of theory in the area of software methodology and tools are:

1. Program verification and proof.
2. Temporal logic.
3. Reliability theory.
4. The supporting areas of predicate calculus, axiomatic semantics, and cognitive psychology.

6.2 Abstraction

Major elements of abstraction in the area of software methodology and tools are:

1. Specification methods, such as predicate transformers, programming calculi, abstract data types, and Floyd–Hoare axiomatic notations.
2. Methodologies such as stepwise refinement, modular

design, modules, separate compilation, information-hiding, dataflow, and layers of abstraction.
3. Methods for automating program development; e.g., text editors, syntax-directed editors, and screen editors.
4. Methodologies for dependable computing; e.g., fault tolerance, security, reliability, recovery, *N*-version programming, multiple-way redundancy, and checkpointing.
5. Software tools and programming environments.
6. Measurement and evaluation of programs and systems.
7. Matching problem domains through software systems to particular machine architectures.
8. Life cycle models of software projects.

6.3 Design
Major elements of design and experimentation in the area of software methodology and tools are:

1. Specification languages (e.g., PSL 2, IMA JO), configuration management systems (e.g., in Ada APSE), and revision control systems (e.g., RCS, SCCS).
2. Syntax directed editors, line editors, screen editors, and word processing systems.
3. Specific methodologies advocated and used in practice for software development; e.g., HDM and those advocated by Dijkstra, Jackson, Mills, or Yourdon.
4. Procedures and practices for testing (e.g., walk-through, hand simulation, checking of interfaces between modules, program path enumerations for test sets, and event tracing), quality assurance, and project management.
5. Software tools for program development and debugging, profiling, text formatting, and database manipulation.
6. Specification of criteria levels and validation procedures for secure computing systems, e.g., Department of Defense.
7. Design of user interfaces.
8. Methods for designing very large systems that are reliable, fault tolerant, and dependable.

7. DATABASE AND INFORMATION RETRIEVAL SYSTEMS
This area deals with the organization of large sets of persistent, shared data for efficient query and update. Fundamental questions include: What modeling concepts should be used to represent data elements and their relationships? How can basic operations such as store, locate, match, and retrieve be combined into effective transactions? How can these transactions interact effectively with the user? How can high-level queries be translated into high-performance programs? What machine architectures lead to efficient retrieval and update? How can data be protected against unauthorized access, disclosure, or destruction? How can large databases be protected from inconsistencies due to simultaneous update? How can protection and performance be achieved when the data are distributed among many machines? How can text be indexed and classified for efficient retrieval?

7.1 Theory
Major elements of theory in the area of databases and information retrieval systems are:

1. Relational algebra and relational calculus.
2. Dependency theory.
3. Concurrency theory, especially serializable transactions, deadlocks, and synchronized updates of multiple copies.
4. Statistical inference.
5. Sorting and searching.
6. Performance analysis
7. As supporting theory: cryptography.

7.2 Abstraction
Major elements of abstraction in the area of databases and information retrieval systems are:

1. Models for representing the logical structure of data and relations among the data elements, including the relational and entity-relationship models.
2. Representations of files for fast retrieval, such as indexes, trees, inversions, and associative stores.
3. Methods for assuring integrity (consistency) of the database under updates, including concurrent updates of multiple copies.
4. Methods for preventing unauthorized disclosure or alteration and for minimizing statistical inference.
5. Languages for posing queries over databases of different kinds (e.g., hypertext, text, spatial, pictures, images, rule-sets). Similarly for information retrieval systems.
6. Models, such as hypertext, which allow documents to contain text at multiple levels and to include video, graphics, and voice.
7. Human factors and interface issues.

7.3 Design
Major elements of design in the area of database and information retrieval systems are:

1. Techniques for designing databases for relational, hierarchical, network, and distributed implementations.
2. Techniques for designing database systems such as INGRES, System R, dBase III, and DB-2.
3. Techniques for designing information retrieval systems such as LEXIS, Osiris, and Medline.
4. Design of secure database systems.
5. Hypertext systems such as NLS, NoteCards, Intermedia, and Xanadu.
6. Techniques to map large databases to magnetic disk stores.
7. Techniques for mapping large, read-only databases onto optical storage media—e.g., CD/ROM and WORMS.

8. ARTIFICIAL INTELLIGENCE AND ROBOTICS

This area deals with the modeling of animal and human (intelligent) behavior. Fundamental questions include: What are basic models of behavior and how do we build machines that simulate them? To what extent is intelligence described by rule evaluation, inference, deduction, and pattern computation? What is the ultimate performance of machines that simulate behavior by these methods? How are sensory data encoded so that similar patterns have similar codes? How are motor codes associated with sensory codes? What are architectures for learning systems, and how do those systems represent their knowledge of the world?

8.1 Theory

Major elements of theory in the area of artificial intelligence and robotics are:

1. Logic; e.g., monotonic, nonmonotonic, and fuzzy.
2. Conceptual dependency.
3. Cognition.
4. Syntactic and semantic models for natural language understanding.
5. Kinematics and dynamics of robot motion and world models used by robots.
6. The supporting areas of structural mechanics, graph theory, formal grammars, linguistics, philosophy, and psychology.

8.2 Abstraction

Major elements of abstraction in the area of artificial intelligence and robotics are:

1. Knowledge representation (e.g., rules, frames, logic) and methods of processing them (e.g., deduction, inference).
2. Models of natural language understanding and natural language representations, including phoneme representations; machine translation.
3. Speech recognition and synthesis, translation of text to speech.
4. Reasoning and learning models; e.g., uncertainty, nonmonotonic logic, Bayesian inference, beliefs.
5. Heuristic search methods, branch and bound, control search.
6. Machine architectures that imitate biological systems, e.g., neural networks, connectionism, sparse distributed memory.
7. Models of human memory, autonomous learning, and other elements of robot systems.

8.3 Design

Major elements of design and experimentation in artificial intelligence and robotics include:

1. Techniques for designing software systems for logic programming, theorem proving, and rule evaluation.
2. Techniques for expert systems in narrow domains (e.g., Mycin, Xcon) and expert system shells that can be programmed for new domains.
3. Implementations of logic programming (e.g, PROLOG).
4. Natural language understanding systems (e.g., Margie, SHRDLU, and preference semantics).
5. Implementations of neural networks and sparse distributed memories.
6. Programs that play checkers, chess, and other games of strategy.
7. Working speech synthesizers, recognizers.
8. Working robotic machines, static and mobile.

9. HUMAN-COMPUTER COMMUNICATION

This area deals with the efficient transfer of information between humans and machines via various human-like sensors and motors, and with information structures that reflect human conceptualizations. Fundamental questions include: What are efficient methods of representing objects and automatically creating pictures for viewing? What are effective methods for receiving input or presenting output? How can the risk of misperception and subsequent human error be minimized? How can graphics and other tools be used to understand physical phenomena through information stored in data sets?

9.1 Theory

Major elements of theory in human-computer communication are:

1. Geometry of two and higher dimensions including analytic, projective, affine, and computational geometries.
2. Color theory.
3. Cognitive psychology.
4. The supporting areas of Fourier analysis, linear algebra, graph theory, automata, physics, and analysis.

9.2 Abstraction

Major elements of abstraction in the area of human-computer communication are:

1. Algorithms for displaying pictures including methods for smoothing, shading, hidden lines, ray tracing, hidden surfaces, transparent surfaces, shadows, lighting, edges, color maps, representations by splines, rendering, texturing, antialiasing, coherence, fractals, animation, representing pictures as hierarchies of objects.
2. Models for computer-aided design (CAD).
3. Computer representations of physical objects.
4. Image processing and enhancement methods.
5. Man-machine communication, including psychological studies of modes of interaction that reduce human error and increase human productivity.

9.3 Design

Major elements of design and experimentation in the area of human–computer communication are:

1. Implementation of graphics algorithms on various graphics devices, including vector and raster displays and a range of hardcopy devices.
2. Design and implementation of experimental graphics algorithms for a growing range of models and phenomena.
3. Proper use of color graphics for displays; accurate reproduction of colors on displays and hardcopy devices.
4. Graphics standards (e.g., GKS, PHIGS, VDI), graphics languages (e.g., PostScript), and special graphics packages (e.g., MOGLI for chemistry).
5. Implementation of various user interface techniques including direct manipulation on bitmapped devices and screen techniques for character devices.
6. Implementation of various standard file interchange formats for information transfer between differing systems and machines.
7. Working CAD systems.
8. Working image enhancement systems (e.g., at JPL for pictures received from space probes).

ACKNOWLEDGMENTS

Many people generously provided written comments in response to drafts of this report. Although it was not possible to accommodate every comment in detail, we did take every comment into account in revising this report. We are grateful to the following people for sending us their comments:

Paul Abrahams	Richard Epstein	J. Paul Myers
J. Mack Adams	Frank Friedman	Bob Noonan
Robert Aiken	C. W. Gear	Alan Perlis
Donald Bagert	Robert Glass	Jesse Poore
Alan Biermann	Nico Habermann	Terrence Pratt
Frank Boesch	Judy Hankins	Jean Rogers
Richard Botting	Charles Kelemen	Jean Sammet
Albert Briggs, Jr.	Ken Kennedy	Mary Shaw
Judy Brown	Elliot Koffman	J. W. Smith
Rick Carlson	Barry Kurtz	Dennis Smolarski
Thomas Cheatham	Doris Lidtke	Ed Upchurch
Neal Coulter	Michael Loui	Garret White
Steve Cunningham	Paul Luker	Gio Wiederhold
Verlynda Dobbs	Susan Merritt	Stuart Zweben
Caroline Eastman	John Motil	

REFERENCES

1. Abelson, H., and Sussman, G. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass., 1985.
2. Arden, B., ed. *See What Can Be Automated?* Report of the NSF Computer Science and Engineering Research Study (COSERS). MIT Press, Cambridge, Mass., 1980.
3. Denning, P. What is computer science? *Am. Sci.* 73 (Jan.–Feb. 1985), 16–19.
4. Flores, F., and Graves, M. Education. (working paper available from Logonet, Inc., 2200 Powell Street, 11th Floor, Emeryville, Calif. 94608.)
5. Newell, A., Perlis, A., and Simon, H. What is computer science? *Sci.* 157 (1967), 1373–1374. (reprinted in *Abacus* 4, 4 (Summer 1987), 32.)