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 in a precise way. We used to say that a sequence of symbols, such as a name, is re-

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.

2. Is Computer Science Part of Mathematics? Certainly there are phenomena

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 $\left(\begin{smallmatrix}1 & 2\\ 3 & 4\end{smallmatrix}\right)$ and $\left(\begin{smallmatrix}4 & 3\\ 2 & 1\end{smallmatrix}\right)$ 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 theorems. This criterion should obviously be changed to include algorithms as

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 graduate 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 Babbage, one of the "fathers" of computing machines, predicted this already in

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; computer programming was a hobby of mine, and a part time job, but I had never

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 1. Set the value of i equal to $f(x)$.

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.

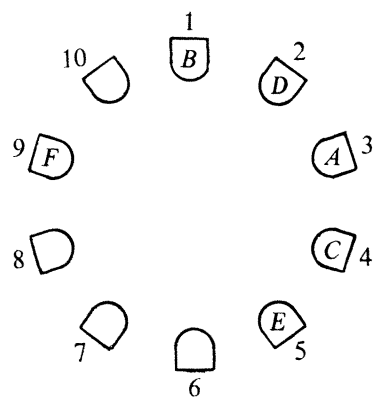

FIG. 1.

A "musical chairs" game which corresponds to an important computer method.

(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

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

and the average distance is

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

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

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 $a_1 a_2 \cdots a_n$, 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. Since the analysis of this practical problem is largely unresolved, and since it has

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

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

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

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

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.

One of the most difficult problems in algorithmic analysis that I have had the

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 $\{1, 2, \cdots, m\}$, 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

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

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