Computer science's Holy Grail

The art of computing gave birth to its own science, which, since it is abstract and mathematical, is a mystery to most people. This is a pity because computer science explores the limits of tomorrow's computers. The next three pages examine its practitioners' current obsession: "P=NP?"

SUPPOSE you have 10,000 numbers and want to find out quickly whether any group of them adds up to 17. This sounds a straightforward enough job for a computer. Alas, this is not the case. Take the problem to a computer programmer and he will shake his head sadly and say that he does not know any practical way to do it. This is strange, because computers do all sorts of complicated things in a trice. And the 10,000-number problem is, after all, so simple that it can be stated in one short sentence. You have stumbled on a member of a class of problems known as NP.

In fact you have hit a problem in this class that is in some ways the most difficult of all (computer scientists call it an NP-complete problem). NP has had computer scientists tied up in knots for the last 15 years. Nobody has found a way of making these problems easy, but nobody has shown that there is no way to do so.

It is more than idle curiosity that drives theoretical computer scientists to search for an answer one way or the other. It would be useful to have fast solutions to some of the problems in NP. The travelling-salesman problem, a mathematicians' old chestnut, is an example. It seeks the cheapest route for a salesman who must visit several cities on a sales trip. No fast way to solve it is known, but nobody has shown that there is no way. NP-problems are in limbo: are they different from the class of problems with fast solutions (called P) or are they one and the same? This question, usually put as "P=NP?" in shorthand, has become the Holy Grail of theoretical computer science.

Remember that our number problem was said not to have a practical solution by computer. What exactly does it mean for a problem to have a practical computer solution? Suppose you work for a telephone company and need to produce the local telephone book. At some point you have to sort through the list of everybody who has a telephone line. Since, for some cities, this can involve millions of names, you need to make sure it can be done quickly. And you not only have to consider how well the computer sorts this year's names, you also have to worry about next year. You need a program that does not take too much additional time as the number of names increases. The best measure of the efficiency of such a program is an indication of how the computation time needed to perform the task rises with the number of names. The relation between the running time and the size of the input (here, the number of names) is called the time complexity of a program.

Computer theorists say a problem has a practical computational solution if there is a program with polynomial time complexity that solves it (or, alternatively, that it can be solved "in polytime"). This means that the time needed to solve it depends directly on the size of the input, or on the size of the input multiplied by itself, or on the size of the input multiplied by itself twice, or thrice, or four times, and so on. Such problems are said to be in the class P (for polynomial time).

Like all theorists, computer theorists tend to be a little unrealistic at times: actually, despite this definition, not all P-problems have genuinely practical solutions. Most programs that run in a time that is any larger than the size of the input cubed (ie, multiplied by itself twice) are probably going to be impractical. This is because the greater the power of the polynomial (the more times you multiply the size of the input by itself) the longer the program will take as the input size increases.

Towards exponential blow-up

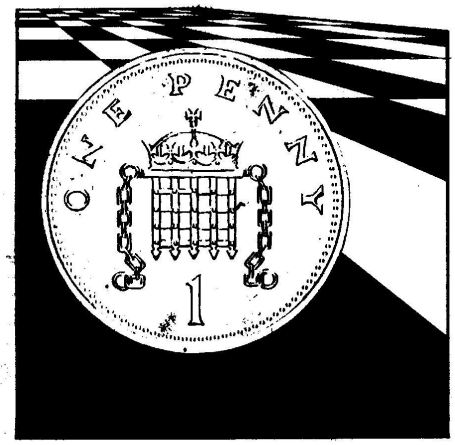
NP is the class of problems with solutions that can be checked in polynomial time. For example, in the "subset-sum problem" considered at the beginning of this article: if you want to convince somebody that there is a group of numbers whose sum is 17, all you need do is provide a group that does add up to 17. A computer takes practically no time at all to add up a given group of numbers and check whether or not the result is 17. Note that this implies nothing about how hard it is to find a solution, only that if somebody thinks they have a solution, a computer can easily check it. The trouble with problems such as subset-sum is that however hard computer scientists try, they can come up with little better than a program that inspects every possible group of numbers from the 10,000 provided and checks to see if the sum is 17. With a few tricks, it is possible to get the number of combinations to be inspected down to just over one thousand billion billion billion. Given that the fastest computers operate at a rate of mere millions of instructions per second, solving the problem is a lost cause.

This obstacle is known as exponential blow-up. All the known programs for problems such as subset-sum and the travelling salesmen suffer from the fact that when you add just one more element to the input (such as one more city in the case of the travelling salesman) the amount of computation time required is multiplied by some number. Such a program is said to have exponential time complexity. This quickly makes the computation time extremely large. Imagine a chessboard with a penny on the first square, two on the second square, and so on, with the number of pennies doubling on each square. On the last square, there will be enough pennies to buy around ten billion tons of gold.

When computer scientists defined the

class P in 1964, NP was not even a dot on the horizon. But they were turning up problem after problem that exhibited the troublesome features of subset-sum: easy to check a solution if you have one, difficult to find the solution in the first place. Scheduling the operation of different bits of machinery at a factory to get the most efficient production is another such problem, for which no polynomial-time program has yet been found. Current programs use rough and ready rules of thumb to get an answer that is good, but probably not the best.

During the 1960s, scientists noticed that some NP problems could be reduced to other NP problems, which turns out to be a helpful start. For instance, a travelling-salesman problem can be converted into an instance of the subset-sum problem with the help of a conversion program that runs in polynomial



time. At the moment, this does not help much because subset-sum problems are just as difficult to solve as travelling-salesman problems. But if you could find a polynomial time solution to subset-sum problems, you could automatically get a polynomial solution to travelling salesman problems by tacking the program to solve the subset-sum problem on to the conversion program. This is because the running time for the two-part program would be the sum of the times for its constituent programs, and adding two polynomials gives you another polynomial. Suddenly, solving many problems became as easy—or as difficult—as solving one of them.

The main breakthough came in 1971 when Dr Stephen Cook, a computer scientist at the University of Toronto, proved a remarkable theorem. He showed that all NP problems could be reduced to a single NP problem in logic called satisfiability (or SAT). If SAT has a fast solution, every NP problem has a fast solution. SAT is therefore said to be an NP-complete problem. It became, in one sense, the most difficult problem in NP. In 1982, Dr Cook got a Turing award—computer science's equivalent of the Nobel prize.

Hard on Dr Cook's heels came Dr Richard Karp from the University of California at Berkeley. Dr Karp reduced SAT to a raft of other NP-problems. At first, this sounds an odd thing to do because Dr Cook had already shown that everything in NP can be reduced to SAT. But the fact that SAT itself can be reduced to subset-sum, and to a handful of other NP-problems, as Dr Karp showed, means that SAT cannot be harder to solve than subset-sum. Dr Cook's reduction of everything in NP to SAT showed, in effect, that nothing in NP was harder than SAT, so—taking Dr Cook's and Dr Karp's results together—it follows that nothing in NP is harder than subset-sum. Subset-sum, like SAT, is NP-complete. Dr Karp, who gave the question "P=NP?" its present form in a paper published in 1972, won the Turing award in 1985.

Dr Leonid Levin, a Russian mathematician now at Boston University (there are quite a few emigré Russian mathematicians working in computer science in America) developed the concept of NP-completeness independently, if a little later than Dr Cook and Dr Karp. The concept of completeness is crucial to a problem such as "P=NP?". Either you show that P=NP is true or you show that it is false. To show that it is true, you must show that every NP-problem is in P. But there are infinitely many NP-problems. On the other hand, showing that P=NP is false would mean producing an NP-problem that cannot under any circumstances be solved by a polynomial time program. Which problem from NP do you select? You may choose one only to find that it does belong in P, which still leaves you in the dark about all the other (infinitely many) problems in NP.

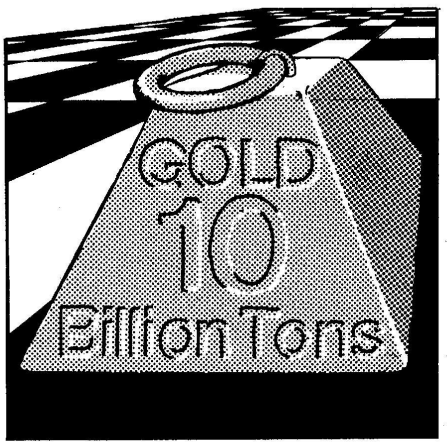
The concept of NP-complete problems helps you here, because it tells you which problems in NP to look at. The NP-complete problems are the hardest ones in NP. If any NP-complete problem can be shown to be in P, then all of NP is in P. Likewise, if you are working on the assumption that NP is different from P, your best bet is to show that some NP-complete problem is not in P, because if any problem in NP is not in P it will be the hardest one. Cook's theorem allowed computer scientists to confine their attention to the complete problems, and ignore the rest.

Can oracles help?

Even so, and despite their best efforts, computer scientists have got nowhere with the problem in the 15 years since Dr Karp first brought it to their attention. Perhaps surprisingly, there are three possible answers to "P=NP?": yes, no, and indeterminable. Although each has its champions, most computer theorists believe that the answer is no—largely because people have been trying to find polynomial time programs for NP-problems for a long time, and have failed miserably.

Not only have computer scientists failed to prove that P is not equal to NP, they have managed to show (worse luck) that none of the traditional methods for distinguishing classes of problems can work in the case of NP. This emerged from work on some strange computers called oracle machines.

Imagine an ordinary computer that is attached to a black box. In the black box lives an elf, who is an expert on a certain problem—call it A—but doesn't know about anything else. If a programmer asks the elf true-or-false questions about A, the elf answers instantaneously. Now it is possible to define two classes of problems, P^A and NP^A in the same way that P and NP were defined: P^A is all problems that can be solved in polytime by the computer with the aid of the elf, while NP^A is all problems that have solutions that can be checked in polytime. The computer now has the extra power of this



elf, or oracle. With this extra power the computer can solve problems in far less time than before. Suppose, for example, that the elf knows all about subset-sum. Then the oracle computer can compute any NP-problem in polynomial time, since it need only convert the NP-problem to subset-sum (which takes polynomial time) and ask the oracle the answer (which it gives instantaneously).

Such oracle computers are unrealistic, so what does imagining them prove? Imagining one oracle did not achieve much, but imagining two let computer scientists discover something new. It is possible to work out the details of two different oracles, called A and B, such that $P^A=NP^A$ is true, but $P^B=NP^B$ is false. This is a depressing result for computer scientists. Computer theorists have some tried and tested methods of showing that two classes of problems are different. But all these methods work independently of the presence of an oracle. This means that if the methods were to show that P=NP is false, then $P^A=NP^A$ would be false for every oracle A. But there is a particular oracle B for which $P^B=NP^B$ is true. Thus none of the traditional methods can ever show that P=NP is false.

Another attempt at the problem uses random oracles. In a random oracle the

black box contains a little elf with a coin. When asked any true-or-false question the elf simply flips his coin, answering true if he gets heads, and false if he gets tails—unless he is asked a question he has already answered, in which case he gives the same answer as before. For complicated reasons, considering random oracles turns out to be a way of considering all possible oracles at once. Computer scientists found that, for almost all oracle machines, P and NP were not the same. What they wanted to show was that if something was overwhelmingly likely for a random oracle machine, it must be true for machines without oracles. Unfortunately this turned out to be wrong.

Back to the wiring

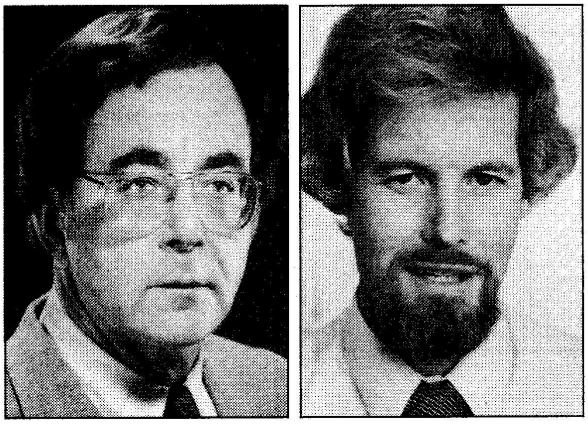
At the moment, most of the work on the "P=NP?" problem concentrates on circuits, which is ironic. Computer science developed by abstracting computation away from its material basis in electronics and other hardware in order to consider it mathematically. Now computer scientists are turning back to circuits to answer the questions raised by those mathematical abstractions.

The idea is to consider all the digital circuits that can solve a certain problem. The problem is encoded using 0s and 1s and fed into the inputs of the circuit, which yields the answer (1 or 0). A circuit is made up of simple components called gates. The link between circuits and the "P=NP?" question lies in the number of gates required by a circuit to solve a particular problem. If the circuit for a given problem needs more than a polynomial number of gates, that problem cannot be in P. So computer scientists try to show that, for example, SAT cannot be solved by a circuit with only a polynomial number of gates. If it cannot, P=NP must be false.

One of the leading computer scientists now working on circuits is Dr Michael Sipser at the Massachusetts Institute of Technology (MIT). Dr Sipser and his colleagues concentrate on much easier problems than NP-complete ones. They have spent a lot of time on the circuit for a problem called parity, which works out whether there is an odd or an even number of 1s in a string of 0s and 1s. The problem of parity is a straightforward one that is definitely in P, but studying satisfiability without looking at simpler problems first would be just too difficult. One technique is to handicap the circuit in some way. For instance, computer scientists might restrict the type of gate used. If they can sort out the simpler restricted cases, they may be able to apply the principles they learn there to the general case.

Plenty of work on circuits has already been done by scientists in the Soviet Union, as a group of of graduate students at the University of California at Berkeley accidentally discovered last year. The students had come up with what they and most others thought was a novel result about the minimum number of gates needed to solve the parity problem. To their chagrin, they learned from a paper in an obscure Soviet journal that it had already been done several years earlier. Dr Alexander Razbarov from the Steklov Institute in Moscow seems to be the leading researcher. Dr Sipser collars the occasional Russian graduate student at MIT to translate for him when the latest paper from Dr Razbarov arrives. Circuit analysis of this sort is not of interest only to theorists. The makers of semiconductors would like to know just how few gates they can get away with using on their chips.

If it were proved that P=NP is false, computer scientists would know for sure that there are no fast ways of solving problems such as the travelling salesman. Some people would be happy to hear it. For a long time, so-called "unbreakable" codes were designed by making up codes, giving them to mathematicians, and letting the mathematicians chew on them for a while. If they could not break them after concentrated effort, then the code was deemed to be usable. The problem with this approach is that a code might turn out to be crackable after just a teeny bit more effort—say one day after it has been passed as uncrackable. If P and NP are not the same, then there are some problems whose solutions are easy to check but difficult to obtain. Much of modern cryptography—at least the part of it that is publicly known—works on this assumption. Some cryptographers would be quite happy to learn that P=NP is false.



Karp appreciated Cook's discovery

There are still some researchers who believe that P=NP. Many of them are not taken very seriously because they produce endless numbers of flawed papers that purport to show that P=NP. One problem with such papers is their length. Since all the obvious ways of making a fast program for NP-complete problems have been tried, any new attempt is going to be fairly devious. On the other hand, bad proofs purporting to show that NP is different from P are not unknown, either.

Dr David Johnson at AT&T's Bell Laboratories in New Jersey has a modest (and almost serious) proposal to stem the tide of bad papers. He proposes that anybody who wants their proof published in a reputable journal should post a $1,000 bond. If the proof turned out to be rubbish, they would forfeit the bond. As an added incentive, forfeited money would go into a pot that would be given to the first verified proof.

If "P=NP?" turned out to have no answer, everybody would lose their money. Before the second world war, a Viennese mathematician, Kurt Gödel, and others proved that some questions in mathematics can never be answered. It is possible that "P=NP?" is one of them. Possible, but inherently unlikely, according to most mathematicians. After all, either there is a program that does subset-sum in polynomial time or there is not. Computer scientists tend to invoke Gödel late at night when they are tired and frustrated.

The smart money is on NP as a separate class, but nobody expects to have an easy time proving it. Dr Cook and Dr Johnson think that the whole field needs an overhaul before the status of NP can be established one way or the other. This is no cause for despair. Computer science is still a young field compared with physics and mathematics. There are ancient unsolved problems in mathematics, such as Fermat's Last Theorem, which get chipped down piece by piece over the years. P=NP has many more implications than Fermat's Last Theorem, an unsolved chestnut about polynomials that has taxed mathematicians for over 300 years. Pure mathematicians are being drawn into the field and computer science problems are being solved by using branches of mathematics, such as geometry, which seemed at first unrelated to computer science.

Everyone concedes that it is difficult to prove even the simplest results in computer science. Even proofs that are easy to understand seem extraordinarily hard to think up, and they may prove unexpected things. This summer Dr Neil Immerman of Yale University settled a question, known as "NL=co-NL?", which is even older though less significant than "P=NP?". A relatively straightforward two-page proof showed that Dr Immerman's answer to the question was yes—the opposite of what most computer scientists expected. Many computer scientists were amazed at how easy the proof was. As one graduate student in computer science put it: "a bunch of complexity theorists are all kicking themselves", which sums up the present atmosphere in a curiously tricky field. ■