The role of theory in computing

The main themes of modern computer science theory emerged about 25 years ago when the first high level languages (FORTRAN, ALGOL, COBOL) were developed. The foundations for much of this theory has a much longer pedigree: in the study of mathematical algorithms (Euclid, Gauss, Turing) and algebra and logic (Boole, Frege, Hilbert, Church).

In broad terms, computer science theory is concerned with the study of abstract programs or *algorithms*, encompassing issues such as: what notations we use to represent algorithms, how we design correct algorithms, how we measure and analyse the complexity of algorithms, how we classify and relate algorithms. The remarkable achievement of the British mathematician Turing (1936) was to propose a formal notion of "algorithm" which provides a satisfactory mathematical foundation for this theory.

During the 1960s, the emphasis in theoretical computer science was upon conventional representations of algorithms (such as are represented today in BASIC programs) closely linked to a machine architecture which was invented by von Neumann in the 1940s (and is still used in almost all modern commercial computers). The inspiration for much theoretical work during the period 1960-70 came from the problems of compiling (translating computer programs from a high-level language into machine code), and this has provided the basis for many important practical programming tools (such as program generators, or compiler-compilers). Other influences which became important later in the 1960s were the development of operating systems (programs to give many users simultaneous access to a single machine) and early researches in Artificial Intelligence. Over the last decade, computer science theory has broadened very significantly in response to more sophisticated applications of computers, and to the challenges of concurrent programming in particular.

To give a flavour of why theory is crucially important in computer science, here are some questions and brief answers intended to illustrate some of the achievements and limitations of our present understanding.

Complexity and feasibility

Can we write a program which will statically analyse programs to decide whether they terminate on a given input?

No - Turing's fundamental work (1936) show this to be outside the scope of any conceivable algorithm. We can't even use static analysis to decide whether a program which accepts as input an arbitrary polynomial equation with integer coefficients and searches for a solution will terminate. Gauss showed in 1801 how to solve the latter problem if the input polynomial is quadratic, but the problem is still apparently infeasible ie takes too much computational time in relatively simple cases. The problem of factorisation: "solve xy=N" is one example of a problem to which Gauss's algorithm applies, but is too inefficient. Factorisation is topical because it has been used as the basis of a new method of encryption, such that

factorising a very large number = breaking the code.

Important progress on this problem has been made very recently by Hendrik Lenstra.

Can we write practical programs which will - automatically generate timetables? - find an optimal tour for a travelling salesman? - reduce Boolean expressions to a standard form?

Probably not! - each of these problems has been shown to be closely related to a particular kind of "decision problem" of the form

given an instance of an object, does it have a specified property?

for which no efficient solution has been found. For instance: given a set of cities is there a tour which visits them each once and once only of less than some specified length?

Our scepticism about the existence of an efficient solution to such a decision problem rests on the fact that it lies in a large class of decision problems - the NP-complete problems which have all been shown essentially equivalent, but none of which is known to be efficiently soluble. This result has practical implications, in that seeking an efficient algorithm to solve such a decision problem is the kind of task best avoided in a commercial environment - however great the pay-off for success might be!

How fast can we multiply integers and matrices / sort records /factorise integers / test for primality ?

Theory has produced some fast algorithms for integer multiplication based on Fourier transform techniques which are of direct practical interest. For matrix multiplication - contrary to intuition - the obvious algorithms aren't necessarily the best. In effect, it is possible to devise esoteric matrix multiplication algorithms which would outperform conventional matrix multiplication algorithms on very large matrices. At present, this result is of little practical relevance, since "very large" is "larger than is practically interesting"! This may well change in time however.

Sorting records is a well-researched and understood problem, for which a variety of efficient methods are known, and where we can be confident that radical improvements can no longer be made in a conventional framework. An interesting feature of one of the most practically important methods of sorting a large number of records (quicksort) is that it exploits random selection, and can be shown to be behave very badly in the worst-case (which "never" arises), and exceptionally well on average.

Testing numbers for primality is an interesting example of a problem for which simple algorithmic solutions of the form: "the given number is shown to be probably prime with a specified probability" have been obtained. Such ideas raise significant philosophical issues!

Data representation and semantics

How do we interpret all the syntactically correct programs in PASCAL (or your favourite programming language)? (A program is syntactically correct if it's structure accords with the definition of a valid PASCAL program.) For instance, when is a syntactically correct program "meaningless"?

This topic - the theory of interpretation of programs - is one of the deepest areas of research, otherwise known as ***semantics.*** *The problem has generated very sophisticated mathematical models within which valid programs can be given formal interpretations. The original motivation for such formal models was the need to tell the compiler writer what his/her compiler should do with programs which "look OK", but whose intended behaviour is obscure.*

An important development in this area has been the introduction of programming languages quite unlike PASCAL for which much more satisfactory mathematical models have been devised, and which in principle can be unambiguously interpreted. Making practical systems based on such languages is an important focus for modern research. Functional programming (as in pure LISP) and logic programming (as in pure PROLOG) are examples. Many controversial issues remain eg can such languages be implemented as efficiently, would non Von Neumann computers be more appropriate for their implementation?

What is an appropriate mathematical model for the representation of "knowledge"? How do we best organise data for reasoning?

The representation of data in a computer system was traditionally based on files of records. As greater awareness of the advantages of centralisation of data within a data base became apparent in the 1970s, the need for more flexible and mathematically sophisticated ways of representing data became apparent. The models which were developed in the early 70s by theorists (relational data bases) are now becoming commercially established. Meanwhile modern applications, especially in connection with CAD and AI, make still greater demands upon data base organisation, and many different new approaches are being explored.

Design, specification and verification of programs

What mathematical principles are appropriate for the design of complex software / hardware systems? How should we specify and communicate formally about a complex software / hardware system?

The need for ways of specifying programs in an abstract fashion first became critical as larger programs came to be written, and communication between many programmmers within a team became essential. The development of moderately large and tolerably reliable software systems proved to be very unexpectedly difficult, and remains problematical despite the advent of new programming tools and techniques. Automatic verification, transformation and translation are expected to play a bigger role in the future, but there remain many challenging research issues both technical and administrative if effective use is to be made of such techniques.

As computer technology advances, the considerations which have traditionally applied to software design become as relevant for hardware systems, and the need for a techniques which make appropriate use of enormous resources feasible has become very clear. These techniques will have to be based upon richer theoretical foundations than we have at present discovered.

Is it realistic to test complex software / hardware systems, and to what extent can we reason about their correctness?

One of the lessons of modern computer science is that pragmatic "intuitive" approaches to system design are hopelessly inadequate. For relatively trivial problems in telecommunications, for instance, such as describing appropriate protocols for the reliable transfer of data between two processes over a noisy channel, it has proved impossible to rely upon informal methods of specifying an algorithm such as are typically used by the most programmers. Even international standards for protocols have proved to contain bugs. Testing has shown the presence of such bugs, but exhaustive testing of such protocols is impossible, and there is no way in which testing can validate the protocol in the absence of better theoretical insights.

The concept of systematically developing and simultaneously justifying a correct specification is central in current research into system design, but there are as yet no satisfactory ways to do this in such areas as software for telecommunications.

Horizons for computing

What will be the form of programming languages and computer systems in the future? How can we exploit the enormous computing resources which are becoming available effectively?

Functional languages, logic languages, object-oriented languages ... it seems reasonable that the significance of languages will be determined more and more by the mathematical principles which they support. Theoreticians will continue to be interested in finding the mathematical techniques needed to ensure that languages admit satisfactory interpretation and implementation. It's difficult to see how the potential complexity of future computer systems can be exploited without richer mathematical theories of computation.

Special purpose hardware of many different types has been studied in the last decade. These include refinements of the traditional Von Neumann model which incorporate features allowing parallel computation (eg DAP and Cray architectures), and more radical designs motivated by functional and logic programming ideas. Because of the scope of modern techniques for fabrication, the distinction between hardware and software is becoming less important, and the central relevance of theoretical principles for software design for future developments is becoming apparent.

How do we develop - perhaps even can we develop? - programs which "learn" and "show intelligence"?

Despite appearances, we still don't understand very much at all about "learning" and "intelligence". The techniques used by modern chess programs have been developed to a stage where fundamental problems must be solved before further progress can be made. Areas such as vision, automatic translation and theorem proving raise exceedingly difficult issues, many of which centre around the significance of "knowledge" in human reasoning and interpretation (cf above).

Logic is the mathematical discipline which is central to the study of Artificial Intelligence, and some progress has been made towards the automation of reasoning. A crucial question is how far the problems of unsolvability and infeasibility (cf above) will necessarily obstruct progress: they certainly exclude the possibility of comprehensive solutions to some basic problems.

Finally, it is worth remarking that (arguably!) the apparent woolliness of some basic AI concepts shouldn't deter us from further investigation. 100 years ago it would probably have seemed outrageous to suggest that the concept of an algorithm could be given a formal interpretation as precise as that proposed by Turing in 1936, and the search for an appropriate theory of intelligence may well generate other surprises of this nature.