Philosophy and Computing

An introduction

Luciano Floridi

London and New York

Chapter 2

The digital workshop

From the laboratory to the house

At the end of 1993 there were about 300 million users of personal computers (PCs) and in 1997 their number was estimated at almost 2 billion. The mass diffusion of the PC since the middle of the 1970s has spread the need for computer literacy, supermarkets list computers as ordinary commodities, and nowadays everyone is supposed to be able to choose and work with a computer, without having any special competence. A PC is just slightly more difficult to use than a car, or domestic white or brown goods, not least because the latter too have become computerised. Of course, each device has a number of features that one should be informed about if one aims to distinguish between the different models, choose more wisely according to one's own needs and employ them to their best. Nevertheless, it is also true that the products on sale are all subject to the same marketing strategies, so the price/performance ratio is hardly ever a source of big surprises. Consequently, if one buys a PC of a decent brand in the nearest shop, with a dash of common sense and an eye to one's wallet rather than to the latest advertisements, one is unlikely to make a big mistake. All this is to reassure the computer-terrified. On the other hand, if you wish to buy a better machine, obtain more for less, get the most from your PC and aim at high performance you need some essential information about its basic features. Moreover, and more importantly, if you are interested in the philosophical issues arising from the information revolution you cannot avoid having some elementary knowledge of the technology in question. The task of the present chapter is to provide this minimal background. It is far from being a technical introduction to computing, which would require a whole volume by itself (see for example Boolos and Jeffrey 1989; Brookshear 1989 and 1997; Cohen 1997; Dewdney 1996; Hopcroft and Ullman 1979; Lewis and Papadimitriou 1998; Minsky 1967; Moret 1998; Savage 1998; Sipser 1997; Taylor 1997). It is only a guide to what one needs to know to be able to take advantage of the technology in question and understand more critically some of the points I shall make in the following chapters.

What is a computer?

Assuming that the computers we shall mostly be concerned with have a classic structure (for some non-classic machines see Chapter 5), answers to this question can be of two kinds: epistemological or commercial. We shall begin with the former and end with the latter, though with a proviso. In the history of science and technology it is often the case that, when people eventually achieve significant results, this is more because they have been chipping away from different angles at what is later discovered to be the same problem, rather than because they have systematically hacked through it inch by inch, so to speak. The history of computing (Goldstine 1972; Metropolis *et al.* 1980; Shurkin 1984; Williams 1997) is a typical example and in the following pages I do not mean to review all the steps, some of which were apparently lost for long periods of time, but only to sketch the major conceptual breakthroughs that can help us to understand what computers are and can do for us.

Before the computer: two semantic distinctions

Human beings rarely wish to think, and whenever thinking requires either repetitive or difficult mental processes we positively dislike the practice, so efforts to devise "intelligent" mechanisms that may help, when not replace altogether, the minds of their owners as well as their muscles, are as old as human history.

For a long time, the only widespread tool for performing calculations was the abacus, usually a frame with balls strung on wires. It was no small advance. A good abacist is capable of making arithmetical calculations at great speed, as Kiyoshu Matzukai, a Japanese abacist, showed in 1946, when he won a two-day contest against an electronic calculator.

It was only after the Renaissance that the number of projects and the variety of calculating machines actually devised began to grow steadily, in connection with the new computational demands brought about by geographical and astronomic discoveries, the scientific revolution, the mathematisation of physics and the emergence of ever more complex forms of conflict (the need for precise ballistic calculations has been behind the history of computing ever since the invention of gun powder), of economic processes (e.g. financial mathematics) and of social organisation (e.g. statistics). The more demanding the intellectual tasks, the more pressing the search became for machines that could take care of them instead of reluctant and unreliable human beings. Some devices, such as Leonardo's, were never built; others, such as Napier's multiplication machine ("Napier's bones"), the slide rule, in which multiplication and subtraction are computed by relying on physical distances, Pascal's adding machine (1642), or Leibniz's multiplication machine (1671), enjoyed some success. Specific mechanical devices were still in widespread use for specialised calculations in several fields as late as the

1960s, and only in the 1970s did their economic value and computational power become uncompetitive compared to that of digital computers.

Mechanical calculators have long been out of fashion, so why mention them at all? Because, from a conceptual point of view, and using our present terminology, calculating machines helped to disseminate two fundamental ideas. The first is the distinction between the following three processes.

1 *data input*, constituting the raw material elaborated by the machine;
2 *the process of elaboration of the data input*, mechanically carried out by the device, following pre-determined patterns or sets of rules. Computing machines are mechanical not in the sense of (a) not being electrical, since there are mechanical machines operated by electrical engines, but in the positive sense of (b) including operational stages requiring no human operator or a human operator with no knowledge of the meaning of the operations performed, and, more often, in the negative sense of (c) not being digital (see below); and
3 *information output*, constituting the result of the process.

The other distinction concerns the semantic system adopted and hence the further distinction between analogue and digital computers.

An analogue computer performs calculations through the interaction of continuously varying physical phenomena, such as the shadow cast by the gnomon on the dial of a sundial, the approximately regular flow of sand in an hourglass or of water in a water clock, and the mathematically constant swing of a pendulum, exploited since the seventeenth century to make the first really accurate clocks and then, in terms of spring-controlled balance wheels, to construct watches. Clearly, it is not the use of a specific substance or reliance on a specific physical phenomenon that makes a computer analogue, but the fact that operations are directly determined by the measurement of continuous physical transformations of whatever solid, liquid or gaseous matter is employed. There are analogue computers that use continuously varying voltages and a Turing machine, as we shall see in a moment, is a digital computer but may not be electrical.

Given their physical nature, analogue computers operate in real time (i.e. time corresponding to time in the real world) and therefore can be used to monitor and control events as they happen, in a 1:1 relation between the time of the event and the time of computation (think of the hourglass). However, because of their nature, analogue computers cannot be general-purpose machines but can only perform as necessarily specialised devices. As far as analogue signals can be subject to substantial interference and physical distortion (think of the scratches on a vinyl record, or the background noise in a radio programme), they are easily prone to malfunctioning, though, conversely, they have a high degree of resilience (fault tolerance), being subject to graceful degradation (you can play a vinyl record almost forever) and able to deal successfully with noisy or damaged input data.

Now, old calculating devices, such as the abacus and the slide rule, are the precursors of analogue computers, with which they share a fundamentally geometrical and topological semantics: they deal with the continuum of physical variations, and perform calculations by relying on mechanical rotations, circular movements, reallocations, distances, contiguities, ratios, etc. For this reason, they may all be considered machines based on the geometrical management of a Euclidean space of information.

In contrast, our computers handle only digital signals in series of distinct steps, and they are the outcome of an algebraic treatment of information. Digital signals are binary (i.e. encoded by means of combinations of only two symbols called *bits*, *b*inary dig*its*) strings of 0/1 comparable to the dots and dashes in the Morse code. For example, in binary notation the number three is written 11. Since the value of any position in a binary number increases by the power of 2 (doubles) with each move from right to left (i.e. . . . 16, 8, 4, 2, 1; note that it could have been 1, 2, 4, 8, 16, and so on, but the binary system pays due homage to the Arabic language) 11 means $[(1 \times 2) + (1 \times 1)]$, which adds up to 3 in the decimal system. Likewise, if you calculate the binary version of 6, equivalent to $[(1 \times 4) + (1 \times 2) + (0 \times 1)]$ you will see that it can only be 110. A *bit* is the smallest semantic unity, nothing more than the presence or absence of a signal, a 1 or a 0. A *nibble* is a series of four bits, and a series of 8 bits forms a *byte* (*by* eight), a semantic molecule through which it becomes possible to generate a table of 256 (2^8) alphanumeric characters (a *word* is used to represent multiple bytes). Each character of data can then be stored as a pattern of 8 bits. The most widely used of such codes is the ASCII (American Standard Code for Information Interchange), which relies on only 7 bits out of 8 and therefore consists of a table of 128 characters. Thus in ASCII representation, each digit of a number is coded into its own byte. Here is how a computer spells "GOD" in binary: 010001110100111101000100, that is

G	off = 0	on = 1	off = 0	off = 0	off = 0	on = 1	on = 1	on = 1
O	off = 0	on = 1	off = 0	off = 0	on = 1	on = 1	on = 1	on = 1
D	off = 0	on = 1	off = 0	off = 0	off = 0	on = 1	off = 0	off = 0

Quantities of bytes are calculated according to the binary system:

- 1 Kilobyte (Kbytes or Kb) = 2^{10} = 1,024 bytes
- 1 Megabyte (Mbytes or Mb) = 2^{20} = 1,048,576 bytes
- 1 Gigabyte (Gbytes or Gb) = 2^{30} = 1,073,741,824 bytes
- 1 Terabyte (Tbytes or Tb) = 2^{40} = 1,099,511,627,776 bytes

As will become clearer in a moment, such economical semantics has at least three extraordinary advantages.

To begin with, bits can equally well be represented logically (true/false), mathematically (1/0) and physically (transistor = on/off, switch = open/closed, electric circuit = high/low voltage, disc or tape = magnetised/unmagnetised, CD = presence/absence of pits, etc.), and hence provide the common ground where mathematical logic, the logic of circuits and the physics of information can converge. This means that it is possible to construct machines that are able to recognise bits physically and behave logically on the basis of such recognition. This is a crucial fact. The only glimpse of intelligence everyone is ready to attribute to a computer uncontroversially concerns the capacity of its devices and circuits to discriminate between binary differences. If a computer can be said to perceive anything at all, it is the difference between a high and a low voltage according to which its circuits are then programmed to behave. Finally, digital signals can have only two states, thus they can be either present or absent, and such *discrete variation* means that a computer will hardly ever get confused about what needs to be processed, unlike an analogue machine, which can often perform unsatisfactorily. Above all, a digital machine can recognise if a signal is incomplete and hence recover, through mathematical calculations, data that may have become lost if there is something literally odd about the quantity of bits it is handling.

A computer can handle operations involving terabytes of information only because it is capable of processing series of bits (billions of 0s and 1s specially ordered into patterns to represent whatever needs to be symbolised either as retrievable data or as executable instructions) at extraordinary speed, something a mechanical device could never achieve. Thus a digital computer, as we know it today, could not be constructed as long as three other fundamental elements were missing:

1 a clear distinction between the fixed hardware of the machine and its replaceable instructions, and hence a theory of algorithms (the programmed instructions implementing the algorithm are known as software, and the machine that executes the instructions is known as hardware);
2 the harmonisation of data and instructions by means of the same binary notation;
3 the transformation of the string of binary data and list of instructions into a flow of electric pulses that could travel easily and safely at very high speed (the speed of light is 30 cm/nanosecond, but electricity travelling through copper wire has a limit of 9 cm/nanosecond).

By itself, the binary codification of data and the following logic of bits was already known to Leibniz (see Leibniz 1968). Nevertheless, even at the end of the nineteenth century, it would have been extremely counterintuitive to pursue the project of the automation of computations and data management by multiplying the number of elements to be processed. Suppose you wish to construct a simple device, with gears and cogs, which could automatically

calculate additions and subtractions of whole numbers. The last thing you would begin with would be a digital codification of numbers from 0 to 9 and of the corresponding instructions, a process that would enormously increase the quantity of data to be taken into consideration by the machine. Yet this is precisely what happens in a computer, which does not recognise the capital letter A as such, for example, and instead of processing only one datum must process the 7 positions of the 0s and 1s in the unique binary string 1000001 standing for "A" in the ASCII table. Although both Leibniz and, later on, Ferdinand de Saussure (Saussure 1983), had made it quite clear that meaningfulness is achievable by the simple presence or absence of one signal, for a long time (roughly until Alan Turing's theoretical achievements) the most promising programme of research for the automatic elaboration of information was thought to be a mechanical *ars combinatoria*. It was hoped that this would enable macroscopic blocks of simplified information such as letters, numbers, words or even whole concepts to be usefully combined following predetermined routes. The speed of the mechanical process was left to the mechanical engine, but the correct management of the necessary instructions remained dependent on human understanding. We have already encountered Swift's ironic description of such a programme of research.

Babbage's analytical engine

It is on the conceptual basis provided by mechanical analogue computers that Charles Babbage, later helped by his associate Augusta Ada Byron (Countess of Lovelace), the daughter of Byron, designed the Difference Engine and then the much more ambitious Analytical Engine (1835), the first true precursor of the modern digital computer (Hyman 1982; Morrison and Morrison 1961). Again, our interest in the device is conceptual rather than technical. The actual engine was never constructed, was fully mechanical, and relied on the ten digits to carry on its calculations, but its blueprint contained a number of brilliant new ideas, some of which we still find applied in the construction of our computers. Four deserve to be mentioned here:

1 The computing process was based on the rule of finite differences for solving complex equations by repeated addition without dividing or multiplying.
2 Instructions for the engine were programmed using conditional algorithms, i.e. branching instructions with loop-back clauses (the pattern is "if such and such is the case then go back to the previous step number n and re-start from there, if such and such is not the case then go to the next step number m") made possible by the punched card system.
3 Both data and instructions came from outside, as input encoded on punched cards.
4 The engine itself implemented only a basic logic and consisted of an

input device, a memory called the Store, a central processing unit called the Mill, and a printer used as an output device.

Since the analytic engine was capable of performing different operations by following different programs, it could qualify as a general-purpose machine, and in 1991 the National Museum of Science and Technology in London created a working example, using Babbage's plans and parts available to him in his time, that turned out to be perfectly efficient.

Turing machines

For centuries, human ingenuity addressed the problem of devising a conceptual and discrete language that would make it possible to assemble and disassemble ever larger semantic molecules according to a compositional logic. Today, we know that this was the wrong approach. Data had to be fragmented into digital atoms, yet the very idea that the quantity of elements to be processed had to be multiplied to become truly manageable was almost inconceivable, not least because nobody then knew how such huge amounts of data could be processed at a reasonable speed. The road leading to semantic atomism was blocked and the analytical engine was probably as close as one could get to constructing a computer without modifying the very physics and logic implemented by the machine. This fundamental step was first taken, if only conceptually, by Alan Turing (Carpenter and Doran 1986; Herken 1988; Hodges 1992; Millican and Clark 1996; on the conceptual and logical issues concerning computability see Boolos and Jeffrey 1989; Minsky 1967; Montague 1962).

Alan Turing's contributions to computer science are so outstanding that two of his seminal papers, "On Computable Numbers, with an Application to the Entscheidungsproblem" (Turing 1936) and "Computing Machinery and Intelligence" (Turing 1950), have provided the foundations for the development of the theory of computability, recursion functions and artificial intelligence. In Chapter 5, we shall analyse Turing's work on the latter topic in detail. In what follows, I shall merely sketch what is now known as a Turing machine and some of the conceptual problems it raises in computation theory. In both cases, the scope of the discussion will be limited by the principal aim of introducing and understanding particular technologies. (Barwise and Etchemendy 1993 is a standard text and software application for the study of Turing machines.)

A simple Turing machine (TM) is not a real device, nor a blueprint intended to be implemented as hardware, but an abstract model of a hypothetical computing system that Turing devised as a mental experiment in order to answer in the negative a famous mathematical question (Herken 1988). In 1928, David Hilbert had posed three questions:

1 Is mathematics complete (can every mathematical statement be either proved or disproved)?

2 Is mathematics consistent (is it true that contradictory sta as "1 = 2" cannot be proved by apparently correct methods
3 Is mathematics decidable (is it possible to find a compleical method whereby, given any expression *s* in the logical system *S*, we can determine whether or not *s* is provable

The last question came to be known as the *Entscheidungsp*31, Kurt Gödel proved that every formal system sufficiently powress arithmetic is either incomplete or inconsistent, and that, if anm is consistent, its consistency cannot be proved within itself (. In 1936, Turing offered a solution to the *Entscheidungsproblem*. hat, given the rigorous representation of a mechanical process bTM, there are *decision problems* (problems that admit Yes or No ε are demonstrably unsolvable by TM.

To understand what a Turing machine is it may helpf it graphically, as a flowchart (a stylised diagram showiious instructions constituting the algorithm and their relat one another), a matrix, or just a program. For our present p can describe a TM as a (possibly fully mechanical) elementarder/ player consisting of:

1 a control unit that can be in only one of two internalıally symbolised by 0/1 (s ∈ {0,1}), operating
2 a read/write head that can move to the right or to the lɛL}), to scan
3 an unlimited tape, divided into symmetric squares, eachnost one symbol α or β (where both α and β ∈ {0,1} andinite number of squares bearing a 1). The tape holds the fir the machine (the string of 0s and 1s), stores all partial re the execution of the instructions followed by the control unring of 0s and 1s generated by the head), and provides the the output of the final result of the computation.

The computational transitions of TM are then regulatertial function: *f*: (α, s) → (β, m, s') (a function *f*: S → T is an openaps strings of symbols over some finite alphabet S to other stbols over some possibly different finite alphabet T; a partial funonly for a proper subset of S) and the machine can be fullyoy a sequence of ordered quintuples: for example, <0, α, β, R, 1d as the instruction "in state 0, if the tape square contains an α, thove one cell right and go into state 1". Note that we have alreal the finite alphabet of TM by limiting it to only two symbols and also limited the number of tapes that TM can use to only one:r of types of operations that TM can perform is very limited. le of activity TM may:

- read a symbol at a time from the current square of the tape (the active square);
- write a symbol on the active square;
- change the internal state of the control unit into a (possibly) different state;
- move the head one space to the right or to the left (whether it is the tape or the head that moves is irrelevant here);
- halt (i.e. carry out no further operations).

TM begins its computation by being in a specified internal state, it scans a square, reads its symbol, writes a 0 or 1, moves to an adjacent square, and then assumes a new state by following instructions such as "if the internal state = 0 and the read symbol on the active square = 1 then write 1, move left, and go into internal state = 1". The logical sequence of TM operations is fully determined at all times by TM's internal state (the first kind of input), the symbol on the active square (the second kind of input) and the elementary instructions provided by the quintuples. The machine can be only in a finite number of states ("functional states"), each of which is defined by the quintuples. All this means that a standard TM qualifies as *at least* a deterministic finite state machine (FSM, also known as finite automaton or transducer. Note that I say "at least" because a TM can do anything a simple FSM can do but not vice versa) in that it consists of:

- a set of states, including the initial one;
- a set of input events;
- a set of output events;
- a state transition function that takes the current state and an input event and returns as values the new set of output events and the next state.

TM is deterministic because each new state is uniquely determined by a single input event. At any particular moment in time, TM is always in a fully describable state and something that I shall later analyse in terms of algorithmic design ensures a very high level of control over the computational process. Any particular TM provided with a specific list of instructions could be described in diagrammatic form by a flow chart, and this helps to explain why TM is better understood as a program or software, and therefore as a whole algorithm, than as a mechanical device. After all, the mechanical nature of the tape recorder is irrelevant, and any similar device would do.

Despite the apparent simplicity of a TM, it is possible to specify lists of instructions that allow specific TMs to compute an extraordinary number of functions (more precisely, if a function is computable by a TM this means that its computation can be transformed into a series of quintuples that constitute the TM in question). How extended is this class of functions? To answer this question we need to distinguish between two fundamental results achieved by Turing, which are usually known as Turing's theorem (TT) and

the Church–Turing thesis (CTT), and a number of other corollaries and hypotheses, including Church's thesis (for a more detailed discussion see Copeland 1997. Some of the following remarks are partly based on an excellent article by B. J. Copeland in the *Stanford Encyclopaedia of Philosophy*, available on the Internet, see webliography).

The theorem proved by Turing was that there is a Universal Turing machine (UTM) that can emulate the behaviour of any special-purpose TM. There are different ways of formulating this result, but the one which is most useful in this context, in order to distinguish TT from other hypotheses, refers to the class of functions that are computable by a machine. Turing's theorem says that there is a UTM that computes (C) any function f that is computable by a TM (TMC):

$$(\mathrm{TT})\ \ \forall f \exists\, \mathrm{x}\,(\mathrm{TMC}(f) \rightarrow (\mathrm{UTM}(\mathrm{x}) \wedge \mathrm{C}(\mathrm{x}, f)))$$

TT means that, given any TM, there is a UTM whose tape contains the complete description of TM's data and instructions and can mechanically reproduce it or, more briefly, that can be programmed to imitate TM. The smallest UTM, devised by Minsky in 1967, can use as little as only 4 symbols and 28 instructions. TT is a crucial result in computation theory: to say that a UTM is a TM that can encompass any other TM is like saying that, given m specific flow charts, drawn in a standard and regimented symbolism, which describe the execution of as many specific tasks, there is a universal flow chart n, written with the same symbols, that can reproduce any of them and thus perform the same tasks. This "super flow chart", UTM, is a general-purpose programmable computing device that provides the logical foundation for the construction of the PC on our desk. Its universality is granted by the distinction between the elementary operations, performed by the hardware, and the instructions, which are specified by a given program and are contained in the software. Unlike the abacus, an analogue calculator or a special-purpose TM, the same UTM can perform an unlimited number of different tasks, i.e. it can become as many TMs as we wish. Change the software and the machine will carry out a different job. In a way that will become clearer in a moment, the variety of its functions is limited only by the ingenuity of the programmer. The importance of such a crucial feature in the field of computation and information theory can be grasped by imagining what it would be like to have a universal electric engine in the field of energy production, an engine that could work as a drill, a vacuum cleaner, a mixer, a motor bike, and so forth, depending on the program that managed it. Note that sometimes UTMs may generically and very misleadingly (see below) be called Super Turing machines.

Turing's fundamental theorem brings us to a second important result, a corollary of his theorem:

(U) a UTM can compute anything a computer can compute.

This corollary may be the source of some serious misunderstandings. If by U one means roughly that

(U.a) a UTM can be physically implemented on many different types of hardware

or, similarly, that

(U.b) every conventional computer is logically (not physically) equivalent to a UTM

then U is uncontroversial: all computer instruction sets, high-level languages and computer architectures, including multi-processor parallel computers, can be shown to be functionally UTM-equivalent. Since they belong to the same class of machines, in principle any problem that one can solve can also be solved by any other, given sufficient time and space resources (e.g. tape or electronic memory), while anything that is in principle beyond the capacities of a UTM will not be computable by other traditional computers. All conventional computers are UTM-compatible, as it were. However, on the basis of a more careful analysis of the concept of computability, the corollary U is at best incorrectly formulated, and at worst completely mistaken. To understand why we need to introduce the Church–Turing thesis.

There are many contexts in which Turing presents this thesis (it is originally formulated in Turing 1936; for a survey of the problems concerning the thesis see Galton 1996). In 1948, for example, Turing wrote that "[TMs] can do anything that could be described as 'rule of thumb' or 'purely mechanical', so that 'calculable by means of a [TM]' is the correct accurate rendering of such phrases" (Turing 1948:7). A similar suggestion was also put forward by Alonzo Church (Church 1936) and nowadays this is known as the Church–Turing thesis: if a function f is effectively computable (EC) then f is computable by an appropriate TM, hence by a UTM (UTMC: henceforth I shall allow myself to speak of TMC or UTMC indifferently, whenever the context does not generate any ambiguity), or more formally

$$(\text{CTT})\ \forall f(\text{EC}(f) \rightarrow \text{TMC}(f))$$

Broadly speaking, CTT suggests that the intuitive but informal notion of "effectively computable function" can be replaced by the more precise notion of "TM-computable function". CTT implies that we shall never be able to provide a formalism F that both captures the former notion *and* is more powerful than a Turing machine, where "more powerful" means that all TM-computable functions are F-computable but not vice versa. What does it mean for a function f to be effectively computable? That is, what are the characteristics of the concept we are trying to clarify? Following Turing's approach, we say that f is EC if and only if there is a method m that qualifies as a procedure of computation (P) that effectively computes (C) f:

$$\text{(a)}\ \forall f(\text{EC}(f) \leftrightarrow \exists \text{m}\ (\text{P(m)} \wedge \text{C}$$

A method m qualifies as a procedure that effectively if and only if m satisfies all the following four conditions:

(1) m is finite in length and time.

m is set out in terms of a finite list of discrete, exact repeatable instructions, which, after a given time (after a given reps), begins to produce the desired output. To understand the finim in length and time recall that in a TM the set of instructions il by a finite series of quintuples (more precisely, we say that a Tular set of quintuples), while in an ordinary computer the set ns is represented by a stored program, whose application is perfigh a fetch–execute cycle (obtaining and executing an instructionence of (1) is the halting problem that we shall analyse at the encon.

(2) m is fully explicit and non-ambiguous.

Each instruction in m is expressed by means of a fi of discrete symbols belonging to a language L and is completelyy interpretable by any system capable of reading L.

(3) m is faultless and infallible.

m contains no error and, when carried out, always ame desired output in a finite number of steps.

(4) m can be carried out by an idiot savant.

m can (in practice or in principle) be carried out by aand patient human being, without any insight, ingenuity or the instrument, by using only a potentially unlimited quantity of stl time (it is better to specify "potentially unlimited" rather thain order to clarify the fact that any computational procedure tha requires an actually infinite amount of space and time to begi its output never ends and is not effectively computable; see belon (4) makes explicit that m requires no intelligence. A consequencat whatever a UTM can compute is also computable *in principle* being. I shall return to this point in Chapter 5. At the moment, it isnotice that, to become acceptable, the converse of CTT requires ss, hidden by the "in principle" clause, for the human being in quehave to live an arbitrarily long time, be infinitely patient and pse the same kind of stationery resources used by UTM. I suppo: to imagine such a Sisyphus in Hell than in a computer room, ost intuitive sense, the one endorsed by Turing himself (see Chapesis is easily acceptable as true by definition.

More briefly, we can now write that:

$$\text{(b)}\ \forall \text{m}\ (((\text{P(m)} \wedge \text{C(m,}f\text{)}) \leftrightarrow (\{1,2$$

When a TM satisfies {1,2,3,4} we can say that it a particular algorithm, if a UTM implements {1,2,3,4} then it is*able system*

and it is not by chance that the set of conditions {1,2,3,4} resembles very closely the set of conditions describing a good algorithm for a classical Von Neumann machine (see below). The main difference lies in the fact that condition (4) is going to be replaced by a condition indicating the *deterministic* and *sequential* nature of an algorithm for VNM. Since the criteria are less stringent, any good algorithm satisfies {1,2,3,4}, and the three expressions "programmable system", "system that satisfies the algorithmic criterion" and "system that satisfies conditions {1,2,3,4}" can be used interchangeably, as roughly synonymous.

CTT holds that if a computational problem cannot be solved by a TM then it cannot be solved by an algorithmic system. Typical cases of computational procedures satisfying the algorithmic criterion are provided by truth tables (Galton 1990; Schagrin *et al.* 1985) and tableaux (Hodges 1977; Jeffrey 1994; Smullyan 1968) in propositional logic, and the elementary operations in arithmetic, such as the multiplication of two integers. It takes only a few moments to establish that $a = 149 \times b = 193 = c = 28757$, although, since in this example both a and b are prime numbers (integers greater than 1 divisible only by 1 and themselves), it is interesting to anticipate here the fact that there is no efficient algorithm to compute the reverse equation, i.e. to discover the values of a and b given c, and that the computation involved in the prime factorisation of 28757 could take us more than an hour using present methods. This is a question concerning the complexity of algorithms that we shall discuss in more detail in Chapter 5. Here, it is worth remarking that the clause "in principle", to be found in condition (4) above, is important because, together with the unbounded resources available to the idiot savant, it means that huge elementary calculations, such as $7^{98876} \times 3^{8737}$, do not force us to consider the multiplication of integers a procedure that fails the test, no matter how "lengthy" the computation involved is.

Clearly, conditions {1,2,3,4} are sufficiently precise to provide us with a criterion of discrimination, but they are not rigorous and formal enough to permit a logical proof. This seems to be precisely the point of CTT, which is perhaps best understood as an attempt to provide a more satisfactory interpretation of the intuitive concept of effective computation, in terms of TM-computability. From this explanatory perspective, wondering whether it is possible to falsify CTT means asking whether it is possible to show that CTT does not fully succeed in capturing our concept of effective computation in its entirety. To show that CTT is no longer satisfactory we would have to prove that there is a class of functions that qualifies as effectively computable but is demonstrably not computable by TM, that is

$$\text{(NOT-CTT)}\ \exists f(\text{EC}(f) \wedge \neg \text{TMC}(f))$$

The difficulty in proving NOT-CTT lies in the fact that, while it is relatively easy to discover functions that are not TM-computable but can be calculated by other mathematical models of virtual machines – all non-recursive functions would qualify (see below) – it is open to discussion whether these functions can also count as functions that are effectively computable in the rather precise, though neither sufficiently rigorous nor fully formal sense, adopted in (a) and (b) and specified by the algorithmic criterion. The problem of proving whether NOT-CTT is the case can be reformulated in the following terms: does the existence of Super Turing machines (STMs) falsify CTT? STMs are a class of theoretical models that can obtain the values of functions that are demonstrably not TM-computable. These include the ARNN (analogue recurrent neural network) model of Siegelmann and Sontag (1994; Siegelmann 1995, 1998).

ARNNs consist of a structure of n interconnected, parallel processing elements. Each element receives certain signals as inputs and transforms them through a scalar – real-valued, not binary – function. The real-valued function represents the graded response of each element to the sum of excitatory and inhibitory inputs. The activation of the function generates a signal as output, which is in turn sent to the next element involved in a given transformation. The initial signals originate from outside the network, and act as inputs to the whole system. Feedback loops transform the network into a dynamical system. The final output signals are used to encode the end result of the transformation and communicate it to the environment. Recurrent ANNs are mathematical models of graphs not subject to any constraints. We shall discuss the general class of artificial neural networks at greater length in Chapter 5, where we see some of their structural properties, their application-areas and the kind of philosophical mistakes that can be made when interpreting their operations. Note that neural networks may represent dynamic systems, but the latter can also be discrete models. The question concerning the computational significance of such models is perfectly reasonable, and trying to answer it will help us to understand better the meaning of CTT, and the power of UTM (recall that we began this section by asking how large the class of functions that are UTM-computable is).

Let us begin by presenting a second hypothesis – sometimes simply mistaken for CTT and sometimes understood as a "strong" version of it – which is plainly falsified by the existence of STM. Following the literature on the topic, I shall label it M:

$$\text{(M)}\ \forall f(\text{M/C}(f) \rightarrow \text{TMC}(f))$$

M says that if f is a *mechanically calculable* (M/C) function – f can be computed by a machine working on finite data in accordance with a finite set of instructions – then f is TM-computable (for a recent restatement of M see for example Bynum and Moor 1998: 2). M is irrecoverably false. It may never be possible to implement and control actual STMs – depending on the model, STMs may require an actually infinite number of processing elements or, if this number is finite, an infinite degree of precision in the computational capacity of each processing element – but this is irrelevant here. A TM is also

a virtual machine, and the demonstration of the existence of a class of Super Turing (virtual) machines is sufficient to prove, at the mathematical level, that not every function that can *in principle* be *calculated* by any machine is also *computable* by a TM, that is

$$(\text{STM})\ \exists f(\text{M/C}(f) \wedge \neg\, \text{TMC}(f))$$

Since we can prove STM, this falsifies M:

$$(\text{NOT-M})\ \text{STM} \rightarrow \neg\, \text{M}$$

The first consequence of what has been said so far is that, while CTT does not support any substantial philosophical conclusion about the possibility of strong AI, NOT-M undermines any interpretation of the feasibility of strong AI based on M. The brain may well be working as a computational engine running mechanically computable functions without necessarily being a UTM ("brain functions" may be TM-uncomputable), in which case it would not be programmable nor "reproducible" by a UTM-equivalent system. But more on the strong AI programme in Chapter 5. A second consequence of NOT-M is that the concept of calculability and the weaker concept of input–output transformation must be distinguished from that of effective computability in such a way as to make room for the possibility that the behaviour of a system of state-transitions may be determined (i.e. obtained or describable in terms of calculations or transformations) without necessarily being, for that reason alone, effectively computable and therefore fully predictable (this applies to microsystems such as ANNs as well as to macrosystems such as weather conditions).

We may now wonder whether NOT-M implies that CTT is also falsified, that is, whether we should also infer that

$$(\not\text{CTT})\ \text{STM} \rightarrow (\text{NOT-CTT})$$

Some computer scientists and philosophers seem to hold that $\not\text{CTT}$ is the case (Bringsjord 1998; note that, although Siegelmann 1995 is typically interpreted as a defence of $\not\text{CTT}$ and indeed it has been used as an important source in favour of $\not\text{CTT}$, she has later clarified that her actual position is not in contrast with the one presented here. See also Siegelmann 1998.). They interpret the existence of STMs as ultimate evidence that CTT is no longer tenable and needs to be revised. They may in fact be referring to M, in which case they are demonstrably right, that is $\not\text{CTT}$ = NOT-M. However, if we refer more accurately to NOT-CTT, $\not\text{CTT}$ is incorrect, for STMs do not satisfy the first half of the conjunction. They are theoretical machines that can compute classes of TM-uncomputable functions, but they do not qualify as machines in the sense specified by the algorithmic criterion, that is STMs implement *computational processes* but not rule-based, *computational procedures* that *effectively compute f* in the sense specified by {1,2,3,4}. In Bringsjord 1998, for example, we find that the argument in favour of $\not\text{CTT}$ requires the class of

real numbers to be effectively decidable, something that Bringsjord himself acknowledges to be doubtful, and that is definitely untenable if "effectively" is correctly specified by {1,2,3,4}. In STMs we gain more computational power at the expense of a complete *decoupling*[5] between programming and computation (we have calculation as an input-output transformation phenomenon determined by the structure of the hardware, without having computational programmability as a rule-based procedure), while in UTM-compatible systems we gain complete *coupling* between the programmable algorithmic procedure and the computational process of which it is a specification (in terms of computer programs, the process takes place when the algorithm begins its fetch-execute cycle) at the expense of computational power.

A likely criticism of the previous analysis is that it may end up making the defensibility of CTT depend on a mere stipulation, or definitional criterion. There is some truth in this objection, and by spelling it out we reach the second important result inferable from the existence of STMs (the first is NOT-M).

A purely definitional position with respect to CTT may hold that all computable functions are TM-computable and vice versa:

$$(\text{CTT}^{\text{def}})\ \forall f(\text{C}(f) \leftrightarrow \text{TMC}(f))$$

Obviously, if $(\text{CTT}^{\text{def}})$ is the case under discussion, since M follows from it, defenders of $\not\text{CTT}$ may really be referring to $(\text{CTT}^{\text{def}})$ when they seem to move objections against CTT. In which case, it is possible to show that they are arguably right in evaluating the significance of STMs for CTT. For the existence of STMs proves that $(\text{CTT}^{\text{def}})$ is either incorrect and hence untenable, or that it is tenable but then only as a matter of terminological convention, i.e. it should actually be rewritten thus:

$$(\text{Def.})\ (\forall f(\text{C}(f) =_{\text{def.}} \text{TMC}(f))$$

My suggestion is that the possibility of STMs is sufficient to let us abandon (Def.). This is the second interesting contribution to our understanding of the validity of CTT, made by defenders of the computational significance of STMs. If we adopt (Def.), it becomes thoroughly unclear precisely what kinds of operation STMs perform when they obtain the values of TM-uncomputable functions. The acceptance of (Def.) would force us to conclude that STMs are not computing and, although this remains a viable option, it is certainly a most counterintuitive one, which also has the major flaw of transforming the whole problem of the verification/falsification of CTT into a mere question of vocabulary or axiomatic choice. As a result, it is more useful to acknowledge that STMs should be described as *computing* the values of *f*. We have seen, however, that they do not *effectively compute f*, in the sense specified by the algorithmic criterion, although they *calculate* its values. Given the differences between TMs and STMs, the class of calculable

functions is therefore a superclass of the class of effectively computable functions. This is the strictly set-theoretic sense in which Super Turing machines are *super*: whatever can be computed by a TM can be calculated by a STM but not vice versa.

Up to Turing power, all computations are describable by suitable algorithms that, in the end, can be shown to be equivalent to a series of instructions executable by a Turing machine. This is the Church–Turing thesis. From Turing power up, computations are no longer describable by algorithms, and the process of calculation is detached from the computational procedure implementing and hence controllable via instructions. This is the significance of STMs. Since there are discrete dynamical systems (including parallel systems, see below) that can have super-Turing capacities, the distinction between *effective computability* and *calculability* cannot be reduced to the analogue/digital or continuous/discrete systems distinction; the key concept is rather that of general-purpose, rule-based transformation.

It turns out that Turing's model of algorithmic computation does not provide a complete picture of all the types of computational processes that are possible. Some types of artificial neural networks are computing models that offer an approach to computational phenomena that is complementary and potentially superior to the one provided by conventional algorithmic systems. In terms of computational power, digital computers are only a particular class of computers, though they may be the only physically implementable, general-purpose devices. However, even if STMs enlarge our understanding of what can be computed, it should be clear that this has no direct bearing on the validity of CTT. The fact that there are STMs demonstrates that M is false and shows that CTT^{def} is either provably incorrect or trivially true but unhelpful, but STMs do not prove that CTT is in need of revision in any significant sense, because the latter concerns the meaning of *effective computation*, not the extent of what can be generally calculated by a system. CTT remains a "working hypothesis", still falsifiable if it is possible to prove that there is a class of functions that are effectively computable in the sense of {1,2,3,4} but are not TM-computable.

So far, any attempt to give an exact analysis of the intuitive notion of an effectively computable function – the list includes Post Systems, Markov Algorithms, λ-calculus, Gödel–Herbrand–Kleene Equational Calculus, Horn Clause Logic, Unlimited Register machines, ADA programs on unlimited memory machines – has been proved either to have the same computational power as a Universal Turing machine (the classes of functions computable by these systems are all TM-computable and vice versa) or to fail to satisfy the required algorithmic criterion, so the Church–Turing thesis remains a very plausible interpretation of the concept of effective computation. This holds true for classical parallel processing computers (PPC) and non-classical quantum computers (QC) as well (but see Deutsch 1985 for a modified version of CTT). As I hope will become clearer in Chapter 5, either PPCs and QCs are implementable machines that perform *effective* computations, in which case they can only compute recursive functions that are in principle TM-computable and CTT is not under discussion, or PPCs and QCs are used to model *virtual* machines that can calculate TM-uncomputable functions, but then these idealised parallel or quantum STMs could not be said to compute their functions effectively, so CTT would still hold.

From the point of view of what is technologically achievable, not just mathematically possible, PPCs and QCs are better understood as "super" Turing machines only in the *generic* sense of being machines that are exponentially more efficient than ordinary TMs, so rather than "super" they should be described as *richer* under time constraints (i.e. they can do more in less time). The "richness" of PPCs and QCs lets us improve our conception of the tractability of algorithms in the theory of complexity, but does not influence our understanding of the decidability and computability of problems in the theory of computation. Since they are not in principle more powerful than the classical model, *richer* computers do not pose any challenge to CTT.

At this point, we can view a UTM as the ancestor of our personal computers, no matter what processors the latter are using and what software they can run. The computational power of a UTM does not derive from its hardware (in theory a TM and a UTM can share the same elementary hardware) but depends on

- the use of algorithms;
- the intelligence and skills of whoever writes them;
- the introduction of a binary language to codify both data and instructions, no longer as actual numbers but as symbols;
- the potentially unlimited amount of space provided by the tape to encode the whole list of instructions, the input, the partial steps of computation and the final output; and finally
- the potentially unlimited amount of time the machine may take to complete the huge amounts of very simple instructions provided by the algorithms in order to achieve its task.

Despite their impressive capabilities, UTMs are really able to perform only the simplest of tasks, based on recursive functions. A recursive function is, broadly speaking, any function f that is defined in terms of the repeated application of a number of simpler functions to their own values, by specifying a recursion formula and a base clause. More specifically, the class of recursive functions includes all functions generated from the four operations of addition, multiplication, selection of one element from an ordered n-tuple (an ordered n-tuple is an ordered set of n elements) and determination of whether $a < b$ by the following two rules: if F and $G_1, \ldots, G_n$ are recursive, then so is $F(G_1, \ldots, G_n)$; and if H is a recursive function such that for each a there is an x with $(Ha, x) = 0$, then the least such x is recursively obtainable.

We can now state the last general result of this section. According to Church's thesis, every function that is effectively computable is also recursive (R) and vice versa:

$$(\mathrm{CT})\ \forall f(\mathrm{EC}(f) \leftrightarrow \mathrm{R}(f))$$

CT should not be confused with, but can be proved to be logically equivalent to, CTT since it can be proved that the class of recursive functions and the class of TM-computable functions are identical. Like CTT then, CT is not a theorem but a reasonable conjecture that is supported by a number of facts and nowadays results widely accepted as correct. If we assume the validity of CT, then we can describe a function f from set A to set B as recursive if and only if there is an algorithm that effectively computes $f(x)$, for $x \in A$. This is the shortest answer we can give to our original question concerning the extension of the class of functions computable by a UTM. We still have to clarify what it means for a problem to be provably uncomputable by a Turing machine.

Recall clause (1) above: m is finite in length and time. A computer is a discrete-and-finite states machine that can deal only with a finite number of digits, these being data or instructions. Now this indicates that it is both a fallible and a limited machine. Its basic arithmetic (see below) can never treat real numbers that require an infinite list of digits to be represented, but only finite approximations, and this may then give rise to rounding errors, which may occur when real numbers are represented as finite numbers (thus the same program running on two different computers may not always produce the same answers because of different conventions for handling rounding errors implemented by hardware manufacturers; this unsatisfactory situation is bound to be eliminated thanks to the adoption of common standards of approximation, not because approximations will no longer be necessary), discretisation errors, which may occur when a continuous problem is translated into a discrete problem, or convergence errors, which may occur when iterative methods are used to approximate a solution. Of course, if the task in question is endless, such as the generation of the infinite expansion of a computable real number, then there is a sense in which the algorithm cannot terminate, but it would still be a correct algorithm. This is a different case from that represented by problems that cannot be solved by any TM because there is no way of predicting in advance whether or when the machine will ever stop. In the former case, we know that the machine will never stop. Likewise, given sufficient resources and a record of the complete functional state of the execution of the algorithm at each step, it is possible to establish that, if the current state is ever identical to some previous state, the algorithm is in a loop. Problems are provably TM-unsolvable when it is possible to demonstrate that, in principle, it is impossible to determine in advance whether TM will ever stop or not. The best-known member of this class is the halting problem (HP). Here is a simple proof by contradiction of its

undecidability (a direct demonstration of the existence of undecidable problems can be achieved by using Gödel numbers; see Brookshear 1997: 411–15 for a very clear illustration):

1 Let us assume that HP can be solved.
2 If (1) then, for any algorithm N, there is an algorithm P such that P solves HP for N.
3 Let us code N so that it takes another algorithm Q as input, i.e. $Q \Rightarrow N$.
4 Make a copy of Q and code it so that $Q \Rightarrow Q$.
5 Let P evaluate whether $Q \Rightarrow Q$ halts (i.e. whether Q will halt with Q as an input) and let the algorithm N be coded depending on the output of P.
6 If the output of P indicates that $Q \Rightarrow Q$ will halt, then let N be coded so that, when it is executed, it goes into an endless loop.
7 If the output of P indicates that $Q \Rightarrow Q$ will not halt, then let N be coded so that, when it is executed, it halts. In other words, let N be coded in such a way that it does exactly the opposite of what the output of P indicates that $Q \Rightarrow Q$ will do. It is now easy to generate a self-referential loop and then a contradiction by assuming $N = Q$. For when we use algorithm N as input to algorithm N:
8 if the output of P indicates that $N \Rightarrow N$ will halt, then because of (7) when N is executed it will enter into an endless loop and it will not halt.
9 If the output of P indicates that $N \Rightarrow N$ will not halt, then, because of (8) when N is executed it will halt. So, according to (8), if $N \Rightarrow N$ halts then it does not halt, but if it does not halt then according to (9) it does halt, but then it does not halt and so forth. This is a contradiction: N does and does not halt at the same time. Therefore:
10 there is at least one algorithm N such that P cannot solve HP for it, but (2) is true, so (1) must be false: HP cannot be solved.

Sometimes, a simple way to show that a computational problem is undecidable is to prove that its solution would be equivalent to a solution of HP.

After Turing, we have a more precise idea of the concepts of "mechanical procedure", "effective computation" and "algorithm". This was a major step, soon followed by a wealth of mathematical and computational results. Nevertheless, a UTM leaves unsolved a number of practical problems, above all the unlimited resources (space and time) it may require to complete even very simple computations. To become a reliable and useful device, a UTM needs to be provided with a more economical logic, which may take care of the most elementary operations by implementing them in the hardware (some hardware can be regarded as a rigid but highly optimised form of software), and a more efficient architecture (HSA, hardware system architecture, that is, the structure of the system, its components and their interconnections). In summary, one may say that Boole and Shannon provided the former, and Von Neumann, with many others including Turing himself, the latter (Von

Neumann's work was the epilogue to prior innovative ideas, hence, when I refer to a Von Neumann machine I shall do so only for the sake of simplicity).

Boolean logic

Boolean algebra is named after George Boole, who first formulated it in his *An Investigation into the Laws of Thought, on which are founded the Mathematical Theories of Logic and Probabilities* (1854). Boole's work went largely ignored (Peirce was one of the few exceptions) until Claude Shannon, the father of the mathematical theory of information, rediscovered it when working on telephone circuits, thus laying the groundwork for the internal logic of our computers. In 1930, when writing his doctoral thesis, and then in 1938 in a classic paper entitled "A Symbolic Analysis of Relay and Switching Circuits", Shannon showed that, once propositional logic – with its two alethic values T (true) and F (false), their truth tables and proof system – is translated into a Boolean algebra (Boolean logic), it can be implemented electronically by means of high and low voltage impulses passing through switching circuits capable of discriminating between on and off states (Shannon 1993).

In modern notation, a Boolean algebra is any 6-tuple $\{B, \oplus, \otimes, \neg, 0, 1\}$ that satisfies the following conditions:

1 B is a set of elements.
2 $\oplus$ and $\otimes$ are two binary operations on B (a binary operation on a set B is a function from $B \times B$ to $\oplus$, for example the truth tables for "OR" and "AND") that are
   (a) *commutative*, a binary operation, *, on B is said to be commutative if and only if
   $\forall x\, \forall y\, (((x \in B) \wedge (y \in B)) \rightarrow (x * y = y * x))$
   (b) *associative*, a binary operation, *, on B is said to be associative if and only if
   $\forall x\, \forall y\, \forall z\, ((((x \in B) \wedge (y \in B)) \wedge (z \in B)) \rightarrow (x * (y * z) = (x * y) * z))$
   (c) *idempotent*, a binary operation, *, on B is said to be idempotent if and only if
   $\forall x\, ((x \in B) \rightarrow (x * x = x))$.
3 Each binary operation is *distributive* over the other; a binary operation $\otimes$ is said to be distributive over a binary operation $\oplus$ on a set B if and only if $\forall x\, \forall y\, \forall z\, ((((x \in B) \wedge (y \in B)) \wedge (z \in B)) \rightarrow (w \otimes (y \oplus z) = (w \otimes y) \oplus (w \otimes z)))$.
4 The constant 0 is the identity for $\oplus$ and the constant 1 is the identity for $\otimes$. An identity for a binary operation, *, on B is an element e in B for which
   $\forall x\, ((x \in B) \rightarrow (x * e = x = e * x))$.

5 The *complement operation* $\neg$ is a unary operation satisfying the condition
   $\forall x\, (x \in B \rightarrow ((x \oplus \neg x = 1) \wedge (x \otimes \neg x = 0)))$.

Since propositional logic, interpreted as a 6-tuple $\{\{F, T\}, \vee, \wedge, \neg, T, F\}$, can be shown to satisfy such conditions it qualifies as a Boolean algebra, and this holds true in set theory as well, where B is the set of subsets of a given set, the operations of intersection ($\cap$) and union ($\cup$) replace $\wedge$ and $\vee$ respectively, and the set complement plays the role of Boolean algebra complement. The question then becomes how we implement a Boolean algebra electronically. We need electronic switches arranged into logical gates, which can be assembled as components of larger functional units. Once this is achieved, it becomes a matter of technological progress to construct increasingly efficient logic gates that are smaller and faster. We have seen that the basic Boolean operators are AND, OR and NOT. The three operators can be constructed as as many gates that assess their inputs, consisting of high or low voltages, and produce a single output, still consisting of a high or low voltage, as their "logical" conclusions:

1 The AND gate yields an output = T/1/On/High (depending on the language we wish to use: mathematical logic, Boolean logic, switch logic or electric voltage) only if both its inputs are high voltages, otherwise the output = F/0/Off/Low.
2 The OR gate yields an output = T/1/On/High whenever it is not the case that both its inputs are low voltages, otherwise the output = F/0/Off/Low.
3 The NOT gate yields an output = T/1/On/High if its input = F/0/Off/Low, and vice versa.

We can then combine such elementary gates in more complex gates to obtain:

4 the XOR gate = exclusive OR, which yields an output = T/1/On/High only if its inputs are different, otherwise the output = F/0/Off/Low;
5 the NAND gate = NOT + AND, which yields an output = T/1/On/High whenever it is not the case that both its inputs are high voltages, otherwise the output = F/0/Off/Low; and
6 the NOR gate = NOT + OR, which yields an output = T/1/On/High only if both its inputs are low voltages, otherwise the output = F/0/Off/Low.

And since combinations of several NAND gates, or OR gates, are sufficient to yield any possible output obtainable from the combination of the set {AND, OR, NOT} (in propositional calculus all 16 two-place truth-functions can be expressed by means of either NAND or NOR, the only two "universal" functions), the logic circuits of a UTM can be constructed by using only one kind of logic gate, taking advantage of the fact that to design circuits built with negative logic gates takes fewer transistors.

Once classic (i.e. two-value) propositional logic, Boolean algebra, the

algebra of set theory and switching algebra become interchangeable, electronic technology can be used to build high-speed logical switching circuits such as adders, multipliers, etc. from simple logical units, and a real UTM can be physically implemented by assembling gates together. For example, by adjusting the sequence of the combinations of gates, we obtain adders that make it possible for a UTM to execute its list of instructions for addition of two binary digits rapidly and reliably. Depending on the technology we use, physical switchers may be huge or very small, slow or incredibly fast, more or less efficient, robust and power-consuming. Logically speaking there is no real difference, although it may be convenient to classify computers by generation depending on their type of hardware.

Generations

We are used to the speed with which computers evolve. A new model has an average life of four to six months, after which a newer and better model supersedes it. This is a life-span that seems to be determined more by financial and marketing factors than by technological improvements, which sometimes occur even faster. As a result, the history of computing has moved so fast through so many technological stages that in fifty years it has accumulated a full archaeological past and four time scales, based on the three macro revolutions, represented by the appearance of the first computers, PCs, and the Internet – the third age of ICT; the evolution of programming languages (see below), the evolution of interfaces (the latter time scale was suggested by John Walker, founder of Autodesk); and finally the improvements in the structure of hardware, which we shall see in this section.

The first generation of machines, developed in the 1940s and 1950s, had gates constructed with valves and wire circuits. In 1940, John Atanasoff and Clifford Berry used 300 vacuum tubes as switches to construct their device, and so did John Mauchly and J. P. Eckert when they developed the ENIAC at the University of Pennsylvania Moore School. A vacuum tube works as a switch by heating a filament inside the tube. When the filament is very hot, it releases electrons into the tube whose flow can be controlled by the tube itself, so that the absence or presence of a small but detectable current can stand for the required 0 and 1. Given the great inefficiency of vacuum tubes, they were soon replaced by more manageable, reliable and less energy-consuming transistors. Invented in 1947 by William Shockley, John Bardeen, and Walter Brattain of Bell Labs, a standard transistor is a semiconductor amplifying device with three terminals attached to as many electrodes. Current flows between two electrodes and is modified depending on the electrical variations affecting the third electrode, thus implementing the standard 0/1 logic. The semiconductor constituting the transistor is usually a crystalline material such as silicon (but germanium or gallium arsenide, though more expensive,

can also be employed to increase the quality of the performance), that can be modified to allow current to flow under specific circumstances.

Second-generation computers, emerging in the late 1950s and early 1960s, incorporated the new technology of smaller, faster and more versatile transistors and printed circuits. Once devised, however, transistors posed a problem of complexity of structure that was solved in 1958 by Jack St Clair Kilby of Texas Instruments: he first suggested how to assemble together a large collection of interconnected transistors by manufacturing an integrated circuit (chip).

The third generation, from the late 1960s onwards, used integrated circuits, which resulted in the further reduction of size (hence increase in speed), and in reduced failure rate, production costs and hence sale prices. The chip became a microprocessor in 1971, when Intel constructed its 4004, the first commercial member of a successful family, which could process 4 bits at a time. The commercial PC as we know it today was born. There followed the 8080, the 8086, then the 8088 (8 and 16 bit, 10Mhz) and the 80186 (8 bit). In 1982, the 80286 (16 bit and 16Mhz) was launched, in 1985 there appeared the 80386 (32 bit), then in 1990 the 80486, and in 1993 the 80586, known as the Pentium. The difference between the 8080 and the Pentium in terms of velocity of execution was 1/300, and after 1993 such an evolution led to talk of fourth-generation computers, based on microprocessors, sophisticated programming languages and ultra-large scale integration.

The scale-integration parameter refers to the fabrication technology that allows the integration of transistors on a single chip:

- in the 1960s, MSI (moderate-scale integration) allowed the integration of $10^2/10^3$ transistors per chip;
- in the 1970s, with LSI (large-scale integration) it became possible to integrate $10^4/10^5$ transistors;
- in the 1980s, VLSI (very LSI) allowed the integration of $10^5/10^7$ transistors;
- in the 1990s, ULSI (ultra LSI) made possible the integration of $10^7/10^9$ transistors.

For a general perspective on the corresponding evolution of computing speed one may compare the following figures. To multiply together two 10-digit numbers,

- a first generation computer (vacuum tubes and wire circuits) took approx. 1/2000 second;
- a second generation computer (transistors and printed circuits) took approx. 1/100,000 second;
- a third generation computer (integrated circuit) took 1/2,400,000 second;
- a fourth generation computer (LSI and higher) took approx. 1/2,000,000,000 second.

During the mid-1980s there was much talk about a fifth generation of computers, with a ten-year project, sponsored by the Japanese Ministry of International Trade and Industry, and initiated in 1981, aimed at the construction of intelligent systems. It was an interesting failure, whose consequences will be discussed in Chapter 5. The first half of the 1990s was dominated by the partly unexpected revolution caused by the Internet, not by artificial intelligence machines.

Von Neumann machine

In a famous paper, entitled "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" (1946; see Neumann 1961–3), John von Neumann, also relying on a wealth of work done in the area, suggested the essential architecture of a UTM that became universally accepted as standard in the following decades. A Von Neumann machine (VNM) is any UTM that satisfies the following criteria:

- It is a stored-program computer.
  Instructions and data (including partial data generated by the computer during the process) are all stored in a single memory unit rather than separately.
- It is a random-access machine.
  "Random-access" is a conventional, but rather misleading, term referring to the ability of a VNM to access an arbitrarily chosen (not really random) memory location at each step. The memory of the machine can be accessed "randomly", that is, directly, by using data addresses (think of them as telephone numbers for a group of individuals), rather than sequentially, that is by scanning the whole memory space every time, as happens with the ideal tape of a UTM (using the same analogy, one may compare this to the home addresses of the same individuals). The address system allows the re-usability of subroutine processes – roughly chunks of instructions, specifically dedicated to the execution of basic and common tasks – that can be "called into action" within the main program, whenever needed, as a single line of instructions (something like "now go to this address, execute the subroutine you will find there and come back to the present address when you are ready").
- It is a sequential machine.
  The central processing unit (CPU) of a VNM carries out instructions sequentially, i.e. one after the other in time, and never performs two operations simultaneously.
- It is a single path machine.
  There is a single path between the memory unit and the CPU, although this "von Neumann bottleneck" may be just logical.

For all these reasons, it is sometimes suggested that a good definition of a

VNM would be "control-flow machine". Given the previous model, the construction of a VNM requires three basic hardware components:

1 a main memory unit working as storage medium, partly comparable to a UTM's unlimited tape. It is worth recalling that, in the 1970s, personal computers commonly stored programs on an everyday audio-cassette recorder;
2 a CPU, comparable to a UTM's control unit and head;
3 an input/output (I/O) system, again comparable to a UTM's tape.

The I/O system provides data and instructions to the memory unit from the outside. The memory unit stores programs and initial, intermediate and final data. The control unit (CU) of the CPU interprets the instructions provided by the program and causes them to be executed one after the other, performing a specified sequence of read and write operations on the memory itself. The arithmetic and logic unit (ALU) of the CPU, where we find the logic gates introduced above, performs the arithmetic and logic operations. When the process is complete, the output system provides the result.

To achieve its goals a VNM repeatedly performs the following series of actions:

1 Start.
2 Check instructions.
  (a) If there is any unexecuted instruction to be executed then go to (3).
  (b) Else (i.e. if there are no unexecuted instructions then) go to (9).
3 Retrieve the first unexecuted instruction to be executed from memory.
4 Decode the retrieved instruction.
5 Retrieve from memory all data required by the instruction decoded in (4).
6 Execute the instruction decoded in (4) by processing the data retrieved in (5).
7 Store results obtained in (6) in memory.
8 Go to (2).
9 Stop processing.
10 Transmit the output.

In 1951, UNIVAC 1, the first American computer available on the market, already had a Von Neumann architecture and nowadays VNMs are the most common example of UTMs, although we shall see in the next chapter that there are computers (the web computers) which are neither UTM nor VNM and, in Chapter 5, that there are also several types of computers that are still instances of UTMs but are not VNMs.

The physical problems faced by a UTM are partially solved in a VNM by implementing the most basic logical operations in the hardware (logic gates working as Boolean operators) and improving the execution time of the instructions. The latter strategy is simple but very efficient. Suppose our UTM takes one hour to execute all its instructions because it must read/write

a tape 1 km long. To shorten the process one can speed up its movements in several ways (better materials, better algorithms, better structure, better organisation of tasks, etc.), but the most elementary solution is to make the spaces occupied by each square of the tape half their size. The string of 1s and 0s remains the same and so does the logic implemented, but the machine can now move from one to the other in approximately half the time. Decrease their size more than a thousand times and exactly the same machine will need less than a second to perform the same task. Consider now that the circuits of a personal computer are much more than a thousand times smaller than those of a machine like UNIVAC and you will start to understand why, though they are both VNM, the former is orders of magnitude more powerful than the latter. By means of microscopic circuits and chips built through photochemical processes, a PC takes only a fraction of the time required by UNIVAC to achieve the same results, so it can do much more in much less time. This is why microprocessors have become constantly smaller since their first appearance.

At the beginning of 1998, nanotechnology produced ordinary chips with up to 5.5 million transistors measuring 0.25 micrometre each (1 micrometre = 1 millionth of a metre; in the past it was known as a micron; a micron = 1 thousandth of a millimetre; the width of a human hair is approximately 1 millimetre), small enough to pass very easily through the narrowest eye of a needle; 0.15 micron transistors and chips hosting more than 10 million transistors were physically realisable, if very expensive; and transistors as small as 0.1 micron (sailing on one of these through the eye of the same needle we would not be able to see the latter's circumference), with chips hosting up to 50 million of them, were expected to be realisable soon. Theoretically, it was possible to store hundreds of gigabytes of data in an area no larger than the head of a pin. All this made the construction of SOCs (whole computational systems realised on a single chip) perfectly possible in 1998. Unfortunately, there are obvious limits to such "micro-strategy". The ultimate speed of a chip is fixed by the rate at which electronic on/off switches can be flicked. This, in turn, depends on the time it takes for individual electrons, the ultimate "data-carriers", to move across the device. The execution time can be decreased by increasing the speed of the operations, and the latter can be increased dramatically by reducing the space required by the data and their movements, but one cannot increase the data storage density, the superconductivity and the microminiaturisation of circuits indefinitely, and this is not just because of diminishing returns, which are in any case relevant for actual production, but also mathematically, because of subatomic physical properties and quantum noise. The travelling speed of a signal cannot be increased indefinitely. Thus, the more demanding a computational problem is, the more evident the original physical limits of a UTM turn out to be. We must now leave this question on one side, but in Chapter 5 we shall see that there are problems whose computational complexity is so great as to overcome all possible resources physically available to any Von Neumann machine and to require new solutions.

Programming languages and software

As physical realisations of UTM, the computational power of computers depends entirely on the design of their instructions and the physical speed at which they can execute them. The amount of time–space resources available and the quality of the algorithms are therefore crucial (Harel 1992). Software is the soul and blood of our machine but, to change our metaphor, "she loves me, she loves me not . . ." is roughly all a computer is really capable of doing. One may say that computers have very little intelligence of their own, for they can only recognise high and low voltages, or the presence or absence of love, if you prefer, and manage their flux. A binary code is all they can reliably understand (likewise, clocks can record particular movements, but they do not understand time). A computer's operations are invoked by internal instructions that are unintentionally (blindly) triggered by external inputs. This is the sense in which a computer can do only what it is explicitly programmed to do, unlike an organism like a dog, for example, which is naturally structured to respond to the environment in a way that is interactively open. Computer instructions are therefore vital. Take the software away and a computer is just a useless lump of plastic, silicon and metal. Programming is the practical art of writing instructions for a specific instance of a Universal Turing machine. The better the programming, the better the machine will perform, and it should not be surprising that a machine using good software may be better at doing something than its programmer. After all, our cars do run faster than our engineers. But more on this in Chapter 5.

We have seen that to accomplish a task, a computer must perform a detailed sequence of operations that are indicated by an algorithm in terms of a finite list of instructions, expressed by a finite number of symbols. Now, a correct algorithm for a classical computer needs to satisfy a number of properties that I have already partly outlined when discussing the Church–Turing thesis. An algorithm must be

1 *explicit* and *non-ambiguous*; it is completely and uniquely interpretable by any computing device capable of reading the symbols of its language;
2 *faultless* and *infallible*; it contains no error and, when carried out, it obtains always the same output (this is also known as *reliability*);
3 *finite*; it consists of a finite list of (repeatable) instructions that after a given time (begin to) produce the desired output;
4 *deterministic* and *sequential*; it prescribes one operation to be performed at a time, which operation is to be performed at a given time t_1, and which step is to be taken at the following time t_2, once the operation at time t_1 has been performed.

To write a good algorithm for a computing system we need an appropriate language, but when we analyse the "language" of a computer, it is important to distinguish between several levels of representation. At the most basic level we find the *physical system*, made up of a structure of integrated circuits, wires, etc., where there is not yet any explicit representation but rather physical phenomena. Physical phenomena acquire a logical meaning only at the higher level of the *logical system*, where we find OR-gates for example, or the interpretation of a high/low voltage as 1/0. The logical system is a first abstraction made possible by the physical system. A third level is then constituted by the *abstract system*, where the logical patterns, occurring at the second level, acquire a more complex meaning: for example patterns of 0/1 can now represent alphanumeric characters, while commands may stand for whole sets of logical instructions. We then reach the higher-level *conceptual system*, represented by software applications and the programming languages used to write them. Clearly, software is written using programming languages that in the end must result in a series of bits interpretable by the gates of the CPU, the so-called machine code at the level of the logical system, which in turn must correspond to physical patterns interpretable by the physical system at the physical level.

What now seems a long time ago, programmers were as close as possible to the physical level and used to write instructions in *machine language*, that is actual strings of 0s and 1s. When the 0s and 1s were physical vacuum-tube ON–OFF switches, the latter had to be set manually and programming a simple task such as sorting an array of names could take a team of programmers days of hard work. Later on, more intuitive, symbolic representations of machine languages were devised, known as *assembly languages*; they assigned short, mnemonic codes, usually of three letters each, to each machine language instruction, which therefore became easier to handle. Programming started to distance itself from the physical implementation of the software. Assembly languages are converted to machine code by an assembler that translates the source code, i.e. the whole list of mnemonic codes and symbolic operands, into objective (i.e. machine) code. There followed easier and much more intuitive high-level languages, such as FORTRAN (Formula Translating System, a *de facto* standard in scientific computing), ADA (a language especially suitable for parallel computing, named after Augusta Ada Byron), ALGOL, COBOL, BASIC (Beginner's All-purpose Symbolic Instruction Code), which make programming far more intuitive. Two of these programming languages that have been very popular for AI applications are LISP (LISt Processor) and PROLOG (PROrogramming in LOGic). LISP was developed in the 1950s and it was specifically designed for processing heterogeneous lists of symbols. PROLOG was developed in the 1970s and is based on first order logic. OOL (object-oriented languages, see Chapter 4), such as Smalltalk, Objective C, C++ and OO extensions to LISP (CLOS: Common LISP Object System) and PROLOG (L&O: Logic & Objects), are also used for AI programming.

In high-level languages, English words, such as OPEN or PRINT, are used as commands standing for whole sequences of hundreds of machine language instructions. They are translated into machine code by the computer itself thanks to *compilers* or *interpreters*. Compilers are programs that convert other programs, written in a programming language and known as the *source programs*, into programs written in machine language (executable machine code) known as the *object programs*, or into programs written in assembly language, which are then converted into object programs by a separate assembler. The translation is performed before the program is run. Interpreters are programs that directly execute other programs written in a programming language. The conversion of each part of the program is performed as it is being executed.

The development of programming language design has essentially been determined by four factors:

- *Naturalisation*: Programming languages have become increasingly user-friendly by moving away from the machine language and acquiring more intuitive symbolic interfaces.
- *Problem-orientation*: New programming languages have tended to make the structure of resulting programs fit more closely the structure of the problems that they deal with rather than the machine language.
- *Portability*: New programming languages have considered the ease with which the resulting program can be "ported", i.e. made to run on a new platform and/or compiled with a new compiler reliably and with a minimum effort.
- *Maintainability*: New programming languages have considered the ease with which the resulting program can be changed through time to make corrections or add enhancements.

The direction is evident: languages and applications have been steadily moving away from the physical system towards the conceptual system.

If we look now at the resulting software, it is common to distinguish three main categories: operating systems (OS), utilities and applications. Commercially, the distinction is becoming increasingly blurred, but it is still useful to make it. The OS constitutes the environment within which we can manage the computer and run other programs. In the past, the best-known OS was MS-DOS (Microsoft Disk OS) and today it is MS-Windows in its various versions (note that before Win95, MS-Windows was not an OS but a GUI, since it required MS-DOS in the background to run), a multi-tasking software (i.e. it allows you to run more than one program at a time). However, Unix, a multi-user general-purpose OS available in many versions, some of which are free, is also widely used, especially since it is very popular for scientific computing and Internet service providers. Without an OS, a PC is practically useless. The best OS have plug and play facilities that allow the user to connect the PC to other peripherals without too many difficulties

(though sometimes it is more a case of "plug and pray"). Some versions of MS-Windows offer the possibility to run 16-bit (backward compatibility) as well 32-bit applications. Utilities are special software designed to make the management of the computerised system easier and more efficient. Applications are complete, self-contained programs designed to perform specific tasks, such as *word-processing*, an expression coined by IBM in 1964 to describe the function of a brand of electronic typewriter, and cannot be used unless they are installed via the OS.

Types of commercial computers

So far we have described a computer as an electronic, general-purpose, programmable, digital data-processing device that can store, retrieve, process and output data according to programmed instructions. Let us now turn to what a computer is from a commercial perspective.

Computers available on the market are often classified into five groups according to their size, power and intended use:

(1) *Palmtops or notepads*: These are very small devices, similar to pocket calculators. They can have word-processing functions and work as personal organisers. Their processing power and memory are very limited but increasing, and the best models can dialogue with a PC and use the Internet.

(2) *PCs (personal computers,* also known as *microcomputers* or *home computers)*: These can be separated into laptops (also known as notebooks), when they are portable, or desktops and towers, depending on the structure of the case, the main part of the machine. We shall analyse the structure of a PC in a moment.

(3) *Workstations*: These are a more powerful variety of PC, usually with larger screens for more complex graphic applications, and provided with fast connections for communication. They are multi-user (can support many users simultaneously) and multi-tasking, and can be very expensive. As a general rule, the most powerful workstations have the same overall computing power (100 megaflops, 1 megaflop = 1 million floating-point operations per second, a common unit of measurement of performance of computers) as the previous generation supercomputers.

(4) *Minicomputers*: This is a rather old-fashioned term, mainly used to refer to computers built between about 1960 and 1985, that were smaller and less powerful than a mainframe but more powerful than the ordinary PC. They were explicitly designed for multiple users and to be connected to many terminals, but they were replaced by workstations.

(5) *Mainframes*: The heavy-weight players of the family, they can process millions of instructions per second and are the fastest and most powerful stations available. These are used for any high-volume data processing, such as the high performance computing (HPC) necessary, for example, for world-wide weather forecasting, the management of huge company databases,

military centres, scientific laboratories or the administration of very large networks. Supercomputers belong to the top class of mainframes, can process billions of instructions per second and are approximately 50,000 times faster than personal computers.

The dividing lines between the five groups are getting less and less clear the more powerful and the smaller their hardware becomes. In Chapter 5, we shall encounter other types of computers, but for our present purposes we can now safely limit our attention to the standard PC to be found in a shop or a university lab.

The personal computer

The PC is a typical Von Neumann machine and represents the most common class of computers available today, dominating the Soho (small office and home computing) and educational markets. It is normally a single-user machine, of moderate price, capable of increasingly good performance (multi-tasking, the possibility of running more than one program at a time, is now standard) and can easily be connected to other computers to form a network and thus share resources (data, software, peripherals).

The typical hardware configuration of a PC includes eight types of components. In the same system box, the case, we find, among other things, the three usual components of all VNMs: the CPU, the hard disk and at least one drive for the operations of input and output (I/O) of data.

(1) The CPU (central processing unit)

This is the microprocessor that makes the whole machine work, consisting of five principal components: the ALU, the registers, the internal clock, the program counter (PC) and the CU (control unit). These can be found on the motherboard, also known as system board, a fibreglass and copper platform bearing all the principal components of the machine and possibly a number of slots for expansion cards, small circuit boards necessary to perform specialised tasks such as sound reproduction (sound card), video-graphics or communication (ethernet cards). The ALU (arithmetic and logic unit) is the component of the CPU responsible for the logic and arithmetic operations. The registers provide a limited storage for immediate data and results during the computer's operation, the CU is responsible for performing the machine cycle (fetch, decode, execute, store), and the program counter is the register that contains the address of the next instruction to be executed sequentially by the CPU. The CPU communicates with internal and external devices by means of a set of conductors, called buses, which carry data between the various types of memory (internal buses) and to and from the ports (external buses) to which are attached peripherals, such as the keyboard, the mouse, the monitor or a printer. The width of the bus (data path) represents the largest data item that the bus can carry, and is expressed in bits (remember 8 bits = 1

byte). We have already encountered the Intel 4004, a processor that could work with only 4 bits of data at a time. Today, current microprocessors have 32-bit and 64-bit buses (they can process 4 or 8 bytes simultaneously), both internally and externally, although internal buses tend to be wider (usually twice the width) since their size affects the speed of all operations. The clock is an electronic device, normally a stable oscillator, that generates a repetitive series of signals whose constant frequency, expressed in hertz (MHz, 1 MHz = 1 million cycles of electrical pulses per second) is used to synchronise the activities of the system. *Ceteris paribus* (we are comparing the same type of CPU), the higher the MHz of the clock, the faster the computer performs its most basic operations such as adding two numbers. To get some idea of the speed of a clock, consider that in 1981 the original IBM PC had a clock rate of 4.77 MHz, and that during the first half of 1998 systems with a 266 MHz microprocessor were entry-level, while a Pentium II processor ran at 400 MHz. Instructions and data are stored in the memory of the system, but contrary to what happens in the simplified model of a VNM, the CPU of a PC is supported by at least four types of memory:

(a) *The ROM (read only memory):* This is the permanent (non-volatile and non-modifiable) memory, embedded in the microprocessor. It can only be read and is used for storage of the lowest level instructions (firmware, a list of about a hundred primitive machine language operations wired into the chip) required by the system to begin to work (bootstrap)

(b) *The RAM (random access memory)*, also known as SRAM (static RAM) or DRAM (dynamic RAM) depending on specific features: This is a volatile, modifiable, internal and electronic memory, built from semiconductor integrated circuits. To perform its tasks, the CPU needs to retrieve data and instructions from a memory unit, but to speed up the process the CPU does not access external memories such as the hard disk directly. Instead, data are transferred to faster internal memory before they can be used. The RAM is the logical space directly accessed by the CPU, where relevant instructions and data are temporarily stored for as long as they are needed, but are lost when the computer is switched off. Data in a RAM can be accessed very quickly by the CPU because the order of access to different locations does not affect the speed of access. Without a large RAM, the PC cannot run most of the latest versions of software, so new computers are provided with increasingly large memories, and if a PC with 2 Mb of RAM was standard at the beginning of the 1990s, at the beginning of 1998 entry-level machines had 32 or 64 Mb of RAM.

(c) *Different types of external, magnetic or optical, memory units*: These are represented by hard disks, floppy disks, magnetic tapes or CD-ROMs (see below), which work as high-capacity, non-volatile but much slower storage mediums. In this case too, storage capacity has been constantly increasing.

(d) *Cache memory*: This is a limited memory, built from very fast chips (they can be accessed in less than $1/1^{12}$ seconds). The cache memory, rather than the main memory, is used to hold the most recently and often accessed data, and hence speed up their subsequent access.

(2) The hard disk (HD)

This is the non-volatile magnetic memory of the machine that is kept when the computer is switched off. In 1995 the standard PC could have 500 Mb of memory; at the beginning of 1998, a 2 Gb disk was ordinary.

(3) At least one drive for the operations of input and output (I/O) of data

This is usually the floppy disk drive (FDD), but for the operations of input CD-ROM players are becoming the standard device (see below).

(4) One peripheral for the input operations, the CD drive

A CD-ROM (a compact disk read only memory; a WORM is a "write once read many times" CD) is identical to an audio compact disk, an optical medium that takes advantage not of magnetic fields but of laser light phenomena. Digital data are stored not electronically but as microscopic pits, whose presence or absence can be read by a laser beam. CDs are therefore pressed, are much more durable and economical than magnetic disks (HD), but their lower speed of access and the fact that, because of their technology, they cannot be easily transformed into reusable (erase/write many times) memories, make them suitable chiefly for permanent data storage. More technically, CDs have a very high degree of *coercivity* (strictly speaking, this a measure of how difficult it is to rewrite the encoded information on a *magnetic* support). A CD can hold over 550 megabytes, that is about 80 million words, approximately equivalent to 1,000 books of 200 pages each. During the first half of 1998 the velocity of a CD drive ranged between 24 and 32-speed.

(5) More and more often, the standard configuration of a PC may include a modem, an essential piece of hardware for any Internet service via telephone.

A modem is an electronic device, managed by a software package, that allows a computer to transmit data over a telephone line to and from other computers. There are two basic methods whereby bytes can be transferred: either one bit at a time (serial communication) or all eight bits at the same time (parallel communication). A printer, for example, is a parallel device, but a modem is a serial device that converts (MOdulates) digital data (series of bits) into audible tones (analogue signals) and sends them as serial frequencies over dial-up phone lines to another modem, which then re-converts (DEModulates) them back into digital data again. The speed at which a modem transfers data is expressed in BPS (i.e. multiple of 8 bits per second, such as 9,600, 14,400 or 57,600) and is also often referred to as the baud rate. Built-in error correction and data compression algorithms allow modems to communicate more reliably and two or three times faster than the rated speed.

Outside the system box there are two peripherals for the data input:

(6) the keyboard,
(7) the mouse,

and a single peripheral for the data output,

(8) the VDU (video display unit), that is, a monitor.

The computer revolution began precisely when it became possible, in the 1970s, to interact with a computer in a user-friendly environment constituted by visual aids, keyboard and mouse. Standard keyboards are known as QWERTY from their top-left six letters, and represent a very powerful interface between the user and the machine. We work with letters, numbers and symbols that the computer immediately translates into strings of bits. The other essential input device is the mouse, a point-click-drag little box made popular by the graphical user interface (GUI) invented at Xerox PARC and popularised by Apple for the management of the icons, buttons and windows of WIMP (Windows, Icons, Menus and Pointers) applications. Virtually all commercial software requires the use of a mouse and many applications are WYSIWYG ("what you see (on the screen) is what you get (out of the printer)"). The monitors – often 14″ or 15″ screens, but 17″, 19″ or 21″ screens are easily available at higher prices – have resolution expressed in number of pixels (PICtures ELement, the smallest resolvable rectangular area of an image) or DPI (dot per inch) that can be displayed, and are commonly capable of displaying about 800×600 pixels in any of 256 colours (8-bit colour), although for "photorealistic" images one needs over 65,000 colours (16-bit colour) and, preferably, many millions (24-bit colour). When discussing UTMs and Von Neumann machines, I have never mentioned the presence of a VDU and this is because, strictly speaking, the monitor is not an essential part of the computer. In theory, one can use a PC perfectly well even if the screen is not working at all since the latter is there only to make the user's life easier for the I/O operations. This is why it is not necessary to buy a new screen every time one buys a new computer.

Peripheral devices, such as printers or modems, are plugged into the computer through sockets, called ports, which allow them to communicate with the computer by giving them access to the computer's main data lines. Input and output devices, e.g. the modem, commonly plugs into a serial port, also known as Comm port. Printers and other devices that require faster data communication often plug into parallel ports. SCSI (small computer system interface) ports can handle multiple devices concurrently and they are extremely fast and versatile. They can be used for more "demanding" devices such as scanners.

Clearly, among the main variables to be taken into account to evaluate a PC we find

- the speed of the CPU. This is determined by the clock speed, the number of bits it can process internally and the data path. MIPS (million instructions per second) are often used to measure the absolute speed of a CPU, but the indication of how fast a CPU can run may be misleading when used to compare different CPUs;
- the size of the various memories;
- the quality of the monitor (at least size and resolution);
- the speed of the CD drive;
- the presence and speed of the modem.

At this point, it is worth recalling a well-known fact. The world of personal computers is unfortunately divided into two: on the one hand there is a niche of Apple Macintosh (Mac) users (between 1994 and 1998 Apple's share of the global market decreased from 9.4 per cent to 2.6 per cent) and, on the other hand, the vast majority of PC users. There are more than a thousand manufacturers of PCs and, as one may guess, such competition favourably affects the price of the products, if not necessarily the quality. It would be tedious to list the arguments in favour of one or the other type of machine, though, and I would advise the novice to familiarise him- or herself with both types of machines anyway, since their basic software is now the same and differences in use are becoming negligible; what may be a little more interesting is to understand why, given the fact that both a Mac and a PC are VNMs, they are not compatible.

A computer can be described at three different levels of analysis: computational power, types of algorithms used by the system and implementation architecture. The same architecture can have different hardware (HSA) implementations. If such implementations share the same or similar information structure architecture (ISA, this is the programmer-visible portion of a processor's architecture, such as the machine language instructions and registers) of their components, then they all belong to the same family capable of operating with the same software (at least in principle), even if they have a different internal organisation (as in the case of cloned chips). Thus, we speak of IBM-compatible, DOS-compatible or Windows-compatible computers to mean that the machines in question can all work with the same software. On the other hand, since each type of CPU recognises a certain set of instructions, if two microprocessors implement different ISA then their software will not be compatible. This is why we cannot use software written for a Macintosh with an IBM-compatible and vice versa, although both are VNMs. It is like saying that a whale and a dog are both mammals but cannot eat the same food.