Turing vs. Super-Turing: a Defence of the Church-Turing Thesis

The following section comes from the second chapter of L. Floridi, *Philosophy and Computing* (London: Routledge, forthcoming). It is an introductory text to ITC (Information Technology and Communication) and conceptual issues in computer science written for students in philosophy with an elementary training in mathematical logic but no particular competence in computer science.

[…]

2.3. 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 analytic 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.

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" and "Computing Machinery and Intelligence", have provided the foundations for the development of the theory of computability, recursion functions and artificial intelligence. In chapter five, 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.

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. 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 statements such as "1 = 2" cannot be proved by apparently correct methods)?
3. Is mathematics decidable (is it possible to find a completely mechanical method whereby, given any expression s in the logico-mathematical system S, we can determine whether or not s is provable in S?)

The last question came to be known as the *Entscheidungsproblem.* In 1931, Kurt Gödel proved that every formal system sufficiently powerful to express arithmetic is either incomplete or inconsistent, and that, if an axiom system is consistent, its consistency cannot be proved within itself. In 1936, Turing offered a solution to the *Entscheidungsproblem.* He showed that, given the rigorous representation of a mechanical process by means of TM, there are *decision problems* (problems that admit Yes/No answers) that are demonstrably unsolvable by TM.

To understand what a Turing machine is it may help to think of it graphically, as a

flowchart—a stylised diagram showing the various instructions constituting the algorithm and their relationship to one another—a matrix, or just a program. For our present purposes, we can describe a TM as a (possibly fully mechanical) elementary tape recorder/player consisting of

1. a control unit that can be in only one of two internal states s, usually symbolised by 0/1 ($s \in \{0,1\}$), operating
2. a read/write head that can move to the right or to the left ($m \in \{R,L\}$), to scan
3. an unlimited tape, divided into symmetric squares, each bearing at most one symbol α or β (where both α and $\beta \in \{0,1\}$ and there is a finite number of squares bearing a 1). The tape holds the finite input for the machine (the string of 0s and 1s), stores all partial results during the execution of the instructions followed by the control unit (the new string of 0s and 1s generated by the head), and provides the medium for the output of the final result of the computation.

The computational transitions of TM are then regulated by the partial function: $f: (\alpha, s) \rightarrow (\beta, m, s')$ (a function $f: S \rightarrow T$ is an operation that maps strings of symbols over some finite alphabet S to other strings of symbols over some possibly different finite alphabet T, a partial function holds only for a proper subset of S) and the machine can be fully described by a sequence of ordered quintuples: for example, $<0, \alpha, \beta, R, 1>$ can be read as the instruction "in state 0, if the tape square contains an α, then write β, move one cell right and go into state 1". Note that we have already simplified the finite alphabet of TM by limiting it to only two symbols and that we have also limited the number of tapes that TM can use to only one. The number of types of operations that TM can perform is very limited. In each cycle 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 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. 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.

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 any function that is computable by a TM:

TT) " ¦ $x (TMC(¦) ® (UTM(x) Ù C(x, ¦)))

TT means that, given any TM, there is a UTM whose tape contains the description of TM's data and instructions and can mechanically reproduce it or, more briefly, that can be programmed to imitate TM. **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 specified by a given program, contained in the software. Unlike the abacus, an analog 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 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 his thesis. 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, see webliography). A similar suggestion was also put forward by Alonzo Church 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 (TMC) 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

CTT) "¦ (EC(¦) ® TMC(¦))

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:

a) "¦ (EC(¦) « $m (P(m) Ù C(m,¦))

A method **m** qualifies as a procedure that effectively computes f iff **m** satisfies all the following four conditions:

1. **m** is finite in length and time

m is set out in terms of a finite number of discrete, exact and possibly repeatable instructions, which, after a given time (after a given number of steps), begin to produce the desired output. To understand the finite nature of **m** in length and time recall that in a TM the set of instructions is constituted by a finite series of quintuples (more precisely, we say that a TM is a particular set of quintuples), while in an ordinary computer the set of instructions is represented by a stored program, whose application is performed through a fetch-execute cycle (obtaining and executing an instruction). A consequence of (1) is the halting problem that we shall analyse at the end of this section.

2. **m** is fully explicit and non-ambiguous

each instruction in **m** is expressed by means of a finite number of discrete symbols belonging to a language L and is completely and uniquely interpretable by any system capable of reading L.

3. **m** is faultless and infallible

m contains no error and, when carried out, always obtains the same 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 a meticulous and patient human being, without any insight, ingenuity or the help of any instrument, by using only a potentially unlimited quantity of stationery and time (it is better to specify "potentially unlimited" rather than "infinite" in order to clarify the fact that any computational procedure that necessarily requires an actually infinite amount of space and time never ends and is not effectively

computable, see below). A consequence of (4) is that whatever a UTM can compute is also computable *in principle* by a human being. I shall return to this point in chapter five. At the moment, suffice to notice that, to become acceptable, the converse of CTT requires some provisos, hidden by the "in principle" clause, for the human being in question would have to be immortal, infinitely patient and precise, and use the same kind of stationery resources used by UTM. I suppose it is easier to imagine such a Sisyphus in Hell than in a computer room, but in its most intuitive sense, the one endorsed by Turing himself (see chapter five), the thesis is easily acceptable as true by definition.

More briefly, we can now write that:

a) " m (((P(m) Ù C(m,¦)) « ({1,2,3,4}(m)))

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

Typical cases of computational procedures satisfying the algorithmic criterion are provided by truth tables and tableaux 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 × 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 five. 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 qualify as effectively computable but are demonstrably not computable by TM, that is

NOT-CTT) $¦ (EC(¦) Ù ¬ TMC(¦))

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)/(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 (analog recurrent neural network) model of Siegelmann and Sontag or the dynamic systems of Koiran, Garzon, Cosnard and Moore. ARNNs consist of a structure of n interconnected, parallel processing elements. Each element receives certain signals as inputs and computes 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 computation. 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 computation 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 five. The dynamic systems of Koiran et al. are mathematical structures representing models of systems whose state changes with time, and which may therefore exhibit chaotic behaviour. 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:

M) $\forall f$ **(C/M (**f**)** $\rightarrow$ **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 conditions—then f is TM-computable. **M** is irrecoverably false. It may never be possible to implement and control actual STMs—depending on the model, STMs require either 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 a any machine is also *computable* by a TM, that is

STM) $\exists f$ **(M/C (**f**)** $\wedge$ $\neg$**TMC(**f**))**

Since we can prove **STM**, this falsifies **M**:

NOT-M) STM $\rightarrow$ $\neg$ **M**

An interesting 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 (what is sometimes called GOFAI, see chapter five), **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 program in chapter five. At the moment, we may ask whether **NOT-M** implies that **CTT** is also falsified, that is, whether we should also infer that

~~CTT~~) STM ® (NOT-CTT)

Some computer scientists, most notably Hava T. Siegelmann, seem to hold that **~~CTT~~** is the case. 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 **~~CTT~~ = NOT-M**. However, if we refer more accurately to **NOT-CTT**, **~~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 *computational procedures* that *effectively compute* f in the sense specified by {1,2,3,4}. In STMs we gain more computational power at the expense of a complete *decoupling*[1] between programming and computation (we have calculation as a phenomenon without having computational programmability as a 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 program, 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 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** holds that all computable functions are TM-computable and vice versa:

CTT^{def}) " ¦ (C(¦) « TMC(¦))

Obviously, if **(CTT^{def})** is the case, then **M** follows, so defenders of **~~CTT~~** may really be referring to **(CTT^{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 **(CTT^{def})** is either false and hence untenable, or that it is tenable but then only as a matter of terminological convention, i.e. it should actually be re-written thus:

Def.) (" ¦ (C(¦) =$_{def.}$ TMC(¦))

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 kind of operations 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

---

[1] Coupling is the technical word whereby we refer to the strength of interrelations between the components of a system (e.g. the modules of a program, or the processing elements of an artificial neural network). These interrelations concern the number of references from one component to another, the complexity of the interface between the components, the amount of data passed or shared between components and the amount of control exercised by one component over another. The tighter the coupling, the higher the interdependency, the looser the coupling the lower the interdependency. Completely decoupled components—systems with the a null degree of interdependency—have no common data and no control flow interaction.

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 controllable via instructions. This is the significance of STMs. Since there are discrete dynamical systems (including parallel systems, see below) that can have superturing capacities, the distinction between *effective computability* and *calculability* cannot be reduced to the analog/digital or continuous/discrete systems distinction. 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. Artificial neural networks and dynamic systems 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 so far they have been the only physically implemented general-purpose abstract 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**$^{\mathbf{def}}$ is either provably false or trivially true but useless, but STMs do not show 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 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 of 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 reasonable way of looking at the concept of effective computation. This holds true for classical Parallel Processing Computers (PPC) and non-classical Quantum Computers (QC) as well. As I hope it will become clearer in chapter five, 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:

CT) " ¦ (EC(¦) « R(¦))

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 iff 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. 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 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. Algorithmic 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 decision problems is achievable via diagonalisation, but goes beyond the scope of this section):

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 **W** as input, i.e. **W** ⇒ **N**
4. make a copy of **W** and code it so that **W** ⇒ **W**
5. let **P** evaluate whether **W** ⇒ **W** halts (i.e. whether **W** will halt with **W** as an input) and let the algorithm **N** be coded depending on the output of **P**
6. if the output of **P** indicates that **W** ⇒ **W** 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 **W** ⇒ **W** 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 **W** ⇒ **W** will do. It is now easy to generate a self-referential loop and then a contradiction by assuming **N** = **W**. For when we use algorithm **N** as input to algorithm **N**

8. if the output of **P** indicates that **N** ⇒ **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** ⇒ **N** will not halt, then, because of (8) when **N** is executed it will halt.

So, according to (8), if **N** ⇒ **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 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, the simplest way to show that a computational problem cannot be solved 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, that may take care of the most elementary operations by implementing them in the hardware, 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 others, the latter.