

PDF version of the entry
Turing Machines
<http://plato.stanford.edu/archives/spr2016/entries/turing-machine/>
from the SPRING 2016 EDITION of the

STANFORD ENCYCLOPEDIA OF PHILOSOPHY



Edward N. Zalta Uri Nodelman Colin Allen R. Lanier Anderson
Principal Editor Senior Editor Associate Editor Faculty Sponsor

Editorial Board
<http://plato.stanford.edu/board.html>

Library of Congress Catalog Data
ISSN: 1095-5054

Notice: This PDF version was distributed by request to members of the Friends of the SEP Society and by courtesy to SEP content contributors. It is solely for their fair use. Unauthorized distribution is prohibited. To learn how to join the Friends of the SEP Society and obtain authorized PDF versions of SEP entries, please visit <https://leibniz.stanford.edu/friends/>.

Stanford Encyclopedia of Philosophy
Copyright © 2016 by the publisher
The Metaphysics Research Lab
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305

Turing Machines
Copyright © 2016 by the author
David Barker-Plummer

All rights reserved.

Copyright policy: <https://leibniz.stanford.edu/friends/info/copyright/>

Turing Machines

First published Thu Sep 14, 1995; substantive revision Tue Jun 26, 2012

Turing machines, first described by Alan Turing in (Turing 1937a), are simple abstract computational devices intended to help investigate the extent and limitations of what can be computed.

Turing was interested in the question of what it means for a task to be computable, which is one of the foundational questions in the philosophy of computer science. Intuitively a task is computable if it is possible to specify a sequence of instructions which will result in the completion of the task when they are carried out by some machine. Such a set of instructions is called an *effective procedure*, or *algorithm*, for the task. The problem with this intuition is that what counts as an effective procedure may depend on the capabilities of the machine used to carry out the instructions. In principle, devices with different capabilities may be able to complete different instruction sets, and therefore may result in different classes of computable tasks (see the entry on computability and complexity).

Turing proposed a class of devices that came to be known as Turing machines. These devices lead to a formal notion of computation that we will call *Turing-computability*. A task is Turing computable if it can be carried out by some Turing machine.

The proposition that Turing's notion captures exactly the intuitive idea of effective procedure is called the Church-Turing thesis. This proposition is not provable, since it is a claim about the relationship between a formal concept and intuition. The thesis would be refuted by an intuitively acceptable algorithm for a task that is not Turing-computable, and no such counterexample has been found. Other independently defined notions of

computability based on alternative foundations, such as recursive functions and abacus machines have been shown to be equivalent to Turing-computability. These two facts indicate that there is at least something natural about this notion of computability.

Turing machines are not physical objects but mathematical ones. We require neither soldering irons nor silicon chips to build one. The architecture is simply described, and the actions that may be carried out by the machine are simple and unambiguously specified. Turing recognized that it is not necessary to talk about *how* the machine carries out its actions, but merely to take as given the twin ideas that the machine can carry out the specified actions, and that those actions may be uniquely described.

- 1. A Definition of Turing Machine
 - 1.1 The Definition Formalized
- 2. Describing Turing Machines
 - 2.1 Examples
 - 2.2 Instantaneous Descriptions of a Computation
- 3. Varieties of Turing Machines
- 4. What Can Be Computed
- 5. What Cannot Be Computed
 - 5.1 The Busy Beaver
 - 5.2 The Halting Problem
- 6. Alternative Formulations of Computability
 - 6.1 Recursive Functions
 - 6.2 Abacus Machines
- 7. Restricted Turing Machines
- Bibliography
- Academic Tools
- Other Internet Resources
- Related Entries

1. A Definition of Turing Machines

A Turing machine is a kind of *state machine*. At any time the machine is in any one of a finite number of states. Instructions for a Turing machine consist in specified conditions under which the machine will transition between one state and another.

The literature contains a number of different definitions of Turing machine. While differing in the specifics, they are equivalent in the sense that the same tasks turn out to be Turing-computable in every formulation. The definition here is just one of the common definitions, with some variants discussed in section 3 of this article.

A Turing machine has an infinite one-dimensional *tape* divided into cells. Traditionally we think of the tape as being horizontal with the cells arranged in a left-right orientation. The tape has one end, at the left say, and stretches infinitely far to the right. Each cell is able to contain one symbol, either ‘0’ or ‘1’.

The machine has a *read-write head* which is scanning a single cell on the tape. This read-write head can move left and right along the tape to scan successive cells.

The action of a Turing machine is determined completely by (1) the current state of the machine (2) the symbol in the cell currently being scanned by the head and (3) a table of transition rules, which serve as the “program” for the machine.

Each transition rule is a 4-tuple:

$\langle State_{current}, Symbol, State_{next}, Action \rangle$

which can be read as saying “if the machine is in state $State_{current}$ and the cell being scanned contains $Symbol$ then move into state $State_{next}$ taking $Action$ ”. As actions, a Turing machine may either to write a symbol on the tape in the current cell (which we will denote with the symbol in question), or to move the head one cell to the left or right, which we will denote by the symbols « and » respectively.

If the machine reaches a situation in which there is no unique transition rule to be carried out, i.e., there is none or more than one, then the machine halts.

In modern terms, the tape serves as the memory of the machine, while the read-write head is the memory bus through which data is accessed (and updated) by the machine. There are two important things to notice about the setup. The first concerns the definition of the machine itself, namely that the machine's tape is infinite in length. This corresponds to an assumption that the memory of the machine is infinite. The second concerns the definition of Turing-computable, namely that a function will be Turing-computable if there exists a set of instructions that will result in a Turing machine computing the function regardless of the amount of time it takes. One can think of this as assuming the availability of infinite time to complete the computation.

These two assumptions are intended to ensure that the definition of computation that results is not too narrow. This is, it ensures that no computable function will fail to be Turing-computable solely because there is insufficient time or memory to complete the computation. It follows that there may be some Turing-computable functions which may not be carried out by any existing computer, perhaps because no existing machine has sufficient memory to carry out the task. Some Turing-computable functions may not ever be computable in practice, since they may require more memory than can be built using all of the (finite number

of) atoms in the universe. Conversely, a result that shows that a function is not Turing-computable is very strong, since it certainly implies that no computer that we could ever build could carry out the computation. Section 5 shows that some functions are not Turing-computable.

1.1. The Definition Formalized

Talk of “tape” and a “read-write head” is intended to aid the intuition (and reveals something of the time in which Turing was writing) but plays no important role in the definition of Turing machines. In situations where a formal analysis of Turing machines is required, it is appropriate to spell out the definition of the machinery and program in more mathematical terms. Purely formally a machine might be defined to consist of:

- A finite set of states Q with a distinguished start state,
- A finite set of symbols Σ .

A *computation state* describes everything that is necessary to know about the machine at a given moment in its execution. At any given step of the execution s

- Q_s a member of Q is the state that the Turing machine is in,
- σ_s a function from the integers into Σ describes the contents of each of the cells of the tape,
- A natural number h_s is the index of the cell being scanned

The transition function for the machine δ is a function from computation states to computation states, such that if $\delta(S) = T$

- σ_T agrees with σ_S everywhere except on h_S (and perhaps there too).
- If $\sigma_S(h_S) \neq \sigma_T(h_S)$ then $h_T = h_S$ otherwise, $|h_T - h_S| \leq 1$

The transition function determines the new content of the tape by returning a new function σ but this new function is constrained to be very similar to the old. The first constraint above says that the content of the cells of the tape is that same every where except possibly at the cell that was being scanned. Since Turing machines can either change the content of a cell or move the head, then if the cell is modified, then the head must not be moved in the transition, and if it is not modified, then the head is constrained to move at most one cell in either direction. This is the meaning of the second constraint above.

This definition is very similar to that given in the entry on computability and complexity, with the significant difference that in the alternative definition the machine may write a new symbol as well as move during any transition. This change does not alter the set of Turing-computable functions, and simplifies the formal definition by removing the second condition on the transition function in our definition. Both formal definitions permit the alphabet of symbols on the tape to be any finite set, while the original definition insisted on $\Sigma=\{0,1\}$ this change also does not impact the definition of the set of Turing-computable functions.

2. Describing Turing Machines

Every Turing machine has the same machinery. What makes one Turing machine perform one task and another a different task is the table of transition rules that make up the machine's program, and a specified *initial state* for the machine. We will assume throughout that a machine starts in the lowest numbered of its states.

We can describe a Turing machine, therefore, by specifying only the 4-tuples that make up its program. Here are the tuples describing a simple machine.

$\langle s_0, 1, s_0, \gg \rangle$
 $\langle s_0, 0, s_1, 1 \rangle$
 $\langle s_1, 1, s_1, \ll \rangle$
 $\langle s_1, 0, s_2, \gg \rangle$

This machine has three states, numbered s_0 , s_1 and s_2 . The first two instructions describe what happens in state s_0 . There are two possibilities, either the machine is scanning a '1', in which case the head moves to the right and stays in state s_0 . The machine leaves state s_0 and enters s_1 if it is scanning a '0'. It writes a '1' on that transition. The second two instructions describe what happens in state s_1 , namely if it is scanning a '1' the machine moves the head to the left staying in state s_1 . If it is scanning a '0', the head moves to the right and the machine moves into state s_2 . Since there are no instructions for state s_2 , the machine halts if it reaches that state.

When we are interested in examining the behaviour of a Turing machine, it is common and perhaps more perspicuous to represent the machine using a state diagram. Here is the machine represented in this format.

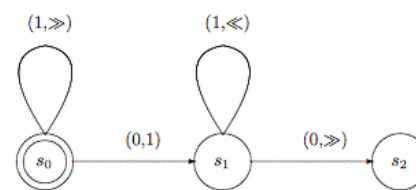


Figure 1: A State Diagram

In this figure, states are represented by the circles, with the unique double circle being the initial state. A transition is represented as an arrow originating from one circle and landing at another (possibly the same) circle. The arrows are labeled by a pair consisting first of the symbol that

must be being scanned for the arrow to be followed, and second the action that is to be taken as the transition is made. The action will either be the symbol to be written, or « or » indicating a move to the left or right.

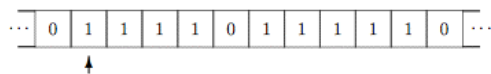
In what follows we will describe Turing machines in the state machine format.

2.1 Examples

In order to speak about a Turing machine that does something useful, we will have to provide an interpretation of the symbols recorded on the tape. For example, if we want to design a machine which will perform some mathematical function, addition say, then we will need to describe how to interpret the ones and zeros appearing on the tape as numbers.

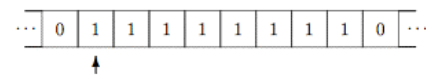
In the examples that follow we will represent the number n as a block of $n+1$ copies of the symbol '1' on the tape. Thus we will represent the number 0 as a single '1' and the number 3 as a block of four '1's.

We will also have to make some assumptions about the configuration of the tape when the machine is started, and when it finishes, in order to interpret the computation. We will assume that if the function to be computed requires n arguments, then the Turing machine will start with its head scanning the leftmost '1' of a sequence of n blocks of '1's. the blocks of '1's representing the arguments must be separated by a single occurrence of the symbol '0'. For example, to compute the sum $3+4$, a Turing machine will start in the following configuration, where the ellipses indicate that the tape has only zeros on the cells that we can't see, and the upward arrow indicates the cell that is currently scanned.



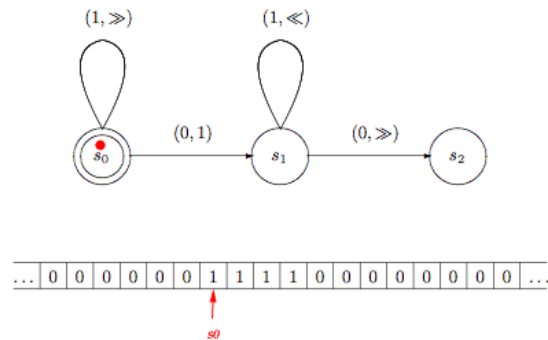
Here the supposed addition machine takes two arguments representing the numbers to be added, starting at the leftmost 1 of the first argument. The arguments are separated by a single 0 as required, and the first block contains 4 '1's, representing the number 3, and the second contains 5 '1's, representing the number 4.

A machine must finish in standard configuration too. There must be a single block of '1's on the tape, and the machine must be scanning the leftmost such '1'. If the machine correctly computes the function then this block must represent the correct answer. So an addition machine started in the configuration above must finish on a tape that looks like this:



Adopting this convention for the terminating configuration of a Turing machine means that we can compose machines by identifying the final state of one machine with the initial state of the next.

Under these conventions, the state diagram in Figure 1 describes a machine which computes the successor (add-one) function. That is when started in standard configuration on a tape representing the number n it will halt in standard configuration representing the number $n+1$. It does this by using state s_0 to scan to the first '0' to the right of the (single) block of '1's. It then replaces that '0' by a '1', and scans left in state s_1 until a '0' is found (this is the first zero to the left of the block of '1's). It then moves back to scan the first '1' and halts in state s_2 .



Above, we see the initial state. Click on the image to see a movie of the execution of the machine. (Click again to stop and reset.)

For another example, consider the machine in Figure 2 which computes the addition function. That is, when started on a standard tape representing the numbers n and m , the machine halts on a tape representing $n+m$.

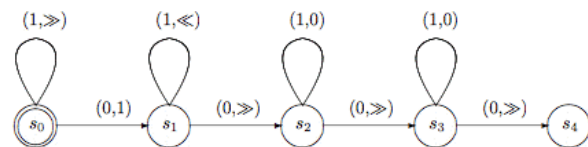


Figure 2: A Machine for Computing $n+m$

Notice that this machine is like the add one machine in that states s_0 through s_2 cause the machine to write a '1' to the right of the first block of '1's, and returns the head to the leftmost '1'. In standard configuration for addition, this joins the two blocks of '1's into a single block, containing $(n+1)+1+(m+1)$ copies of the symbol '1', so that on entering state s_2 the tape represents the number $n+m+2$. In order to correct this, we need to

remove two copies of the symbol '1', which is achieved by states s_2 and s_3 , each of which replaces a '1' by '0' and then moves to the right.

Since the tape initially contains at least two '1's and we write one more, the deletion of two '1's will leave at least one on the tape at the end of the computation, and we will be scanning the leftmost of them.

2.2 Instantaneous Descriptions of a Computation

Recall that we said that an execution state of a Turing machine may be described by the name of the state that the machine is in Q_s , the symbols on the tape, σ_s , and the cell that is currently being scanned h_s . We will represent such a description using a figure like that in Figure 3, in which the arrow represents the currently scanned cell and the name of the current state is written below the arrow.

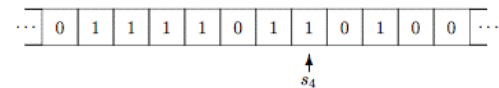


Figure 3: The Instantaneous Description of a Turing Machine Computation

This indicates that our Turing machine is in state s_4 , scanning the indicated cell of the tape. The tape is assumed to contain '0's everywhere that is not visible.

3. Varieties of Turing Machines

We have presented here one of the most common formulations of Turing's basic idea. There are a number of variations to the formulation that turn out to be equivalent to this one, and different authors present Turing machines using any of these. Since they are all provably equivalent to one

another we can consider any of the formulations as being the definition of Turing machine as we find convenient.

Formulation F_1 and formulation F_2 are equivalent if for every machine described in formulation F_1 there is machine a described in F_2 which has the same input-output behavior, and vice versa, i.e., when started on the same tape at the same cell, will terminate with the same tape on the same cell.

Two-way infinite tapes

In our original formulation we specified that the tape had an end, at the left say, and stretched infinitely far to the right. Relaxing this stipulation to allow the tape to stretch infinitely far to right and left results in a new formulation of Turing machines. You might expect that the additional flexibility of having a two-way infinite tape would increase the number of functions that could be computed, but it does not. If there is a machine with a two-way infinite tape for computing some function, there is a machine with a one-way infinite tape that will compute that same function.

Arbitrary numbers of read-write heads

Modifying the definition of a Turing machine so that the machine has several read-write heads does not alter the notion of Turing-computability.

Multiple tapes

Instead of a single infinite tape, we could consider machines possessing many such tapes. The formulation of such a machine would have to allow the tuples to specify which tape is to be scanned, where the new symbol is to be written, and which tape head is to move. Again this formulation is equivalent to the original.

Two-dimensional tapes

Instead of a one-dimensional infinite tape, we could consider a two-dimensional “tape”, which stretches infinitely far up and down as well as left and right. We would add to the formulation that a machine transition can cause the read-write head to move up or down one cell in addition to being able to move left and right. Again this formulation is equivalent to the original.

Arbitrary movement of the head

Modifying the definition of a Turing machine so that the read-write head may move an arbitrary number of cells at any given transition does not alter the notion of Turing-computability.

Arbitrary finite alphabet

In our original formulation we allowed the use of only two symbols on the tape. In fact we do not increase the power of Turing machines by allowing the use of any finite alphabet of symbols.

5-tuple formulation

A common way to describe Turing machines is to allow the machine to both write and move its head in the same transition. This formulation requires the 4-tuples of the original formulation to be replaced by 5-tuples

$$\langle State_0, Symbol, State_{new}, Symbol_{new}, Move \rangle$$

where $Symbol_{new}$ is the symbol written, and $Move$ is one of « and ».

Again, this additional freedom does not result in a new definition of Turing-computable. For every one of the new machines there is one of the old machines with the same properties.

Non-deterministic Turing machines

An apparently more radical reformulation of the notion of Turing machine allows the machine to explore alternative computations in parallel. In the original formulation we said that if the machine specified multiple transitions for a given state/symbol pair, and the machine was in such a state then it would halt. In this reformulation, *all* transitions are taken, and all the resulting computations are continued in parallel. One way to visualize this is that the machine spawns an exact copy of itself and the tape for each alternative available transition, and each machine continues the computation. If any of the machines terminates successfully, then the entire computation terminates and inherits that machine's resulting tape. Notice the word *successfully* in the preceding sentence. In this formulation, some states are designated as *accepting states* and when the machine terminates in one of these states, then the computation is successful, otherwise the computation is unsuccessful and any other machines continue in their search for a successful outcome.

The addition of non-determinism to Turing machines does not alter the definition of Turing-computable.

Turing's original formulation of Turing Machines used the 5-tuple representation of machines. Post introduced the 4-tuple representation, and the use of a two-way infinite tape.

A more complex machine

In addition to performing numerical functions using unary representation for numbers, we can perform tasks such as copying blocks of symbols, erasing blocks of symbols and so on. Here is an example of a Turing machine which when started in standard configuration on a tape containing a single block of '1's, halts on a tape containing two copies of that block of '1's, with the blocks separated by a single '0'. It uses an alphabet consisting of the symbols '0', '1' and 'A'.

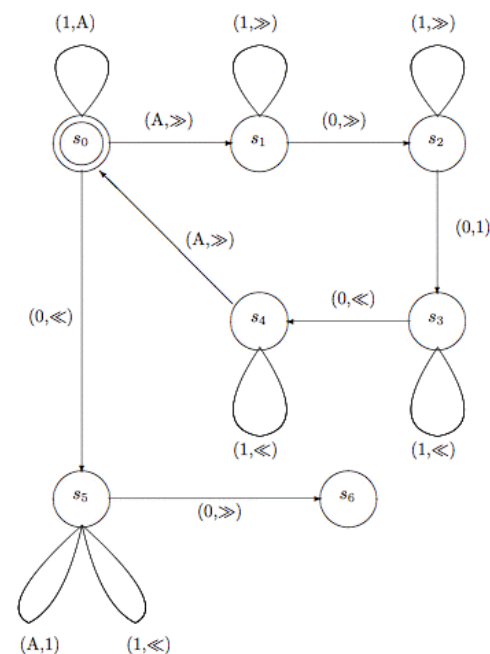


Figure 4: A Machine for Copying a Block of 1s

The action of this machine is to repeatedly change one of the original '1's into an A, and then write a new '1' to the right of all remaining '1' on the tape, after leaving a zero between the original block and the copy. When we run out of the original '1's, we turn the As back into '1's.

The initial state, s_0 , is used to change a '1' into an 'A', and move to the right and into state s_1 . In state s_1 we skip the remainder of the block of '1's until we find a '0' (the block separator) and in s_2 we skip any '1's to the right of that '0' (this is the copy of the block of '1's that we are making). When we reach the end of that block, we find a '0', which we turn into a '1' and head back to the left, and into state s_3 . States s_3 and s_4 skip

leftward over the '1's and separating '0' on the tape until an 'A' is found. When this occurs, we go back into state s_0 , and move rightward.

At this point, we are either scanning the next '1' of the original block, or the original block has all been turned into 'A's, and we are scanning the separator '0'. In the former case, we make another trip through states s_1 – s_4 , but in the latter, we move into state s_5 , moving leftward. In this state we will repeatedly find 'A's, which we replace with '1's, and move to the left. If we find a '0', then all of the 'A's have been turned back into '1's. We will be scanning the '0' to the left of the original cell, and so we move right, and into the final state s_6 .

This copying machine could be used in conjunction with the addition machine of Figure 2 to construct a doubling machine, i.e., a machine which, when started on a tape representing the number n halts on a tape representing $2n$. We could do this by first using the copying machine to produce a tape with two copies of n on the tape, and then using the addition machine to compute $n+n (=2n)$. We would do this by identifying the copying machine's halt state (s_6) with the adding machine's initial state (s_0).

The construction just suggested relies on the fact that the copying machine terminates in standard position, which is required for the adding machine to correctly compute its result. By designing Turing machines which start and end in standard configuration, we can ensure that they may be composed in this manner. In the example, the copying machine has a unique terminating state, but this is not necessary. We might build a Turing machine which indicates the result of its computation by terminating on one of many states, and we can combine that machine with more than one machine, with the identity of the machine which follow dependent on the switching machine. This would enable us to create a machine which

adds one to the input if that input is even, and doubles it if odd, for example (should we want to for some reason).

4. What Can Be Computed

Turing machines are very powerful. For a very large number of computational problems, it is possible to build a Turing machine that will be able to perform that computation. We have seen that it is possible to design Turing machines for arithmetic on the natural numbers, for example.

Computable Numbers

Turing's original paper concerned *computable numbers*. A number is Turing-computable if there exists a Turing machine which starting from a blank tape computes an arbitrarily precise approximation to that number. All of the algebraic numbers (roots of polynomials with algebraic coefficients) and many transcendental mathematical constants, such as e and π are Turing-computable.

Computable Functions

As we have seen, Turing machines can do more than write down numbers. Among other things they can compute numeric functions, such as the machine for addition (presented in Figure 2) multiplication, proper subtraction, exponentiation, factorial and so on.

The *characteristic function* of a predicate is a function which has the value TRUE or FALSE when given appropriate arguments. An example would be the predicate 'IsPrime', whose characteristic function is TRUE when given a prime number, 2, 3, 5 etc and FALSE otherwise, for example when the argument is 4, 9, or 12. By adopting a convention for representing TRUE and FALSE, perhaps that TRUE is represented as a sequence of

two '1's and FALSE as one '1', we can design Turing-machines to compute the characteristic functions of computable predicates. For example, we can design a Turing machine which when started on a tape representing a number terminates with TRUE on the tape if and only if the argument is a prime number. The results of such functions can be combined using the using the boolean functions: AND, NOT, OR, IF-THEN-ELSE, each of which is Turing-computable.

In fact the Turing-computable functions are just the recursive functions, described below.

Universal Turing Machines

The most striking positive result concerning the capabilities of Turing machines is the existence of *Universal Turing Machines* (UTM). When started on a tape containing the encoding of another Turing machine, call it T , followed by the input to T , a UTM produces the same result as T would when started on that input. Essentially a UTM can simulate the behavior of any Turing machine (including itself).

One way to think of a UTM is as a programmable computer. When a UTM is given a program (a description of another machine), it makes itself behave as if it were that machine while processing the input.

Note again, our identification of input-output equivalence with "behaving identically". A machine T working on input t is likely to execute far fewer transitions than a UTM simulating T working on t , but for our purposes this fact is irrelevant.

In order to design such a machine, it is first necessary to define a way of representing a Turing machine on the tape for the UTM to process. To do this we will recall that Turing machines are formally represented as a

collection of 4-tuples. We will first design an encoding for individual tuples, and then for sequences of tuples.

Encoding Turing Machines

Each 4-tuple in the machine specification will be encoded as a sequence of four blocks of '1's, separated by a single '0'

1. The first block of ones will encode the current state number, using the unary number convention above ($n+1$ ones represents the number n).
2. The second block of ones will encode the current symbol, using one '1' to represent the symbol zero, and two to represent the symbol '1' (again because we can't use zero ones to represent '0').
3. The third element of the tuple will represent the new state number in unary number notation.
4. The fourth element represents the action, and there are four possibilities: symbols will be encoded as above, with a block of three '1's representing a move to the left (\ll) and a block of four '1's representing a move to the right (\gg).

Using this convention the tuple $\langle 0, '1', 0, \gg \rangle$ would be represented as in Figure 5.

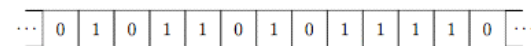


Figure 5: The Encoding of the Tuple $\langle 0, '1', 0, \gg \rangle$

To encode a complete machine, we need to simply write down the tuples on the tape, in any order, but separated from one another by two blank cells so that we can tell where each tupe ends. The add-one machine of Figure 1, would be represented by the somewhat intimidating string shown in Figure 6.

... 010110101111 00 101011011 00 110110110111 00
11010111011110 ...

Figure 6: The Encoding of the Machine in Figure 1

This construction shows that it is possible to encode the tuples of a Turing machine on a Turing machine's tape (in fact we have done this using only the alphabet $\{0,1\}$, while we know from a previous result that we could have used an expanded alphabet which could have resulted in a more perspicuous representation). We want our UTM to be started on a tape which contains the description of a Turing machine in this encoding, followed by the arguments to this described Turing machine. We will adopt the convention that the description of the Turing machine will be separated from the arguments by a block of three '0's, so the UTM can tell where the tuples end and the arguments begin. The add-one machine requires a single argument, so we could start the UTM on a tape as above followed by three '0's and, say, five '1's. The UTM must terminate on a tape which contains a single block of six '1's, having computed exactly what the add-one machine would have done if started on a block of five '1's.

5. What Cannot Be Computed

In the previous section we described a way of encoding Turing machines for input to a Universal Turing Machine. The encoding of a Turing machine is a sequence of '0's and '1' and any such sequence can be interpreted as a natural number. We can think of the encoding of a Turing machine as being a natural number which is the serial number of that machine. Because of the way the encoding works, each Turing machine will have a distinct serial number. Since all of the serial numbers are natural numbers, the number of distinct Turing machines is *countably infinite*.

On the other hand, the number of functions on the natural numbers is uncountable. There are (uncountably) more functions on the natural numbers than there are Turing machines, which shows that there are uncomputable functions, functions whose results cannot be computed by any Turing machine, because there are simply not enough Turing machines to compute the functions.

This proof by counting is somewhat unsatisfactory, since it tells us that there are uncomputable functions, but provides us with no examples. Here we give two examples of uncomputable functions.

5.1 The Busy Beaver

Imagine a Turing machine that is started on a completely blank tape, and eventually halts. If the machine leaves n ones on the tape when it halts, we will say that the *productivity* of this machine is n . We will say that the productivity of any machine that does not halt is 0. Productivity is a function from Turing machine descriptions (natural numbers) to natural numbers. We will write $p(T)=n$ to indicate that the productivity of machine T is n .

Among the Turing machines that have a particular number of states, there is a maximum productivity that a Turing machine with that number of states can have. This too is a function from natural numbers (the number of states) to natural numbers (the maximum productivity of a machine with that number of states). We will write this function as $BB(k)=n$ to indicate the maximum productivity of a k -state Turing machine is n . There may be multiple different k state machines with the maximum productivity n . We call any of these machines a Busy Beaver for k .

There is no Turing machine which will compute the function $BB(k)$, i.e., which when started in standard configuration on a tape with k '1's will halt

in standard configuration on a tape with $BB(k)$ '1's. This example is due to Tibor Radó (Radó 1962).

The proof that there is no such function proceeds by assuming that there is such a machine, i.e. that there is a machine which starts in standard configuration with k '1's on the tape, and halts in standard configuration with $BB(k)$ '1's on the tape. We will call this machine B and assume that it has k states.

There is an n -state machine which writes n '1's on an initially blank tape (exercise for the reader). We can construct a new machine which connects the halting state of this machine to the start state of B and then connecting the halting state of B to the start state of another copy of B . So the first machine writes n '1' and then the first copy of B computes $BB(n)$, but then the second copy of B takes over and computes $BB(BB(n))$. The total number of states in our machine is $n+2k$. Our machine may be a Busy Beaver for $n+2k$, but it is certainly no more productive than such a machine. So (if the Busy Beaver machine exists)

$$BB(n+2k) \geq BB(BB(n)), \text{ for any } n.$$

It is easy to show that the productivity of Turing machines increases as states are added, i.e.,

$$\text{if } i < j, \text{ then } BB(i) < BB(j)$$

(another exercise). Consequently (if the Busy Beaver machine exists)

$$n+2k \geq BB(n), \text{ for any } n.$$

Since this is true for any n , it is true for $n+11$, yielding:

$$n+11+2k \geq BB(n+11), \text{ for any } n.$$

But it is easy to show that $BB(n+11) \geq 2n$ (another exercise, but show that there is an eleven state machine for doubling the number of '1' on the tape, and compose such a machine with the n -state machine for writing n '1's). Combining this fact with the previous inequality we have:

$$n+11+2k \geq BB(n+11) \geq 2n, \text{ for any } n.$$

from which by subtracting n from both sides we have $11+2k \geq n$, for any n , if the Busy Beaver exists, which is a contradiction.

Even though the productivity function is uncomputable, there is considerable interest in the search for Busy Beaver Turing machines (most productive machines with a given number of states). Some candidates can be found by following links in the Other Internet resources section of this article.

5.2 The Halting Problem

It would be very useful to be able to examine the description of a Turing machine and determine whether it halts on a given input. This problem is called the *Halting problem* and is, regrettably, uncomputable. That is, no Turing machine exists which computes the function $h(t,n)$ which is defined to be TRUE if machine t halts on input n and FALSE otherwise.

To see the uncomputability of the halting function, imagine that such a machine H exists, and consider a new machine built by composing the copying machine of Figure 4 with H by joining the halt state of the copier to the start state of H . Such a machine, when started on a tape with n '1's determines whether the machine whose code is n halts when given input n , i.e., it computes $M(n) = h(n,n)$.

Now let's add another little machine to the halt state of H . This machine goes into an infinite sequence of transitions if the tape contains TRUE

when it starts, and halts if the tape contains FALSE (its an exercise for the reader to construct this machine, assume that TRUE is represented by ‘11’, and FALSE by ‘1’).

This composed machine, call it M , halts if the machine with the input code n does not halt on an initial tape containing n (because if machine n does not halt on n , the halting machine will leave TRUE on the tape, and M will then go into its infinite sequence), and vice versa.

To see that this is impossible, consider the code for M itself. What happens when M is started on a tape containing M 's code? Assume that M halts on M , then by the definition of the machine M it does not halt. But equally, if it does not halt on M the definition of M says that it should halt.

This is a contradiction, and the Halting machine cannot exist. The fact that the halting problem is not Turing-computable was first proved by Turing in (Turing 1937b). Of course this result applies to real programs too. There is no computer program which can examine the code for a program and determine whether that program halts.

6. Alternative Formulations of Computability

6.1 Recursive Functions

Recursive function theory is the study of the functions that can be defined using recursive techniques (see the entry on recursive functions). Briefly, the primitive recursive functions are those that can be formed from the basic functions:

the zero function: $z(x) = 0$, for all x

the successor function: $s(x) = x+1$, for all x

the i th projection over j arguments: $p_{i,j}(x_0, \dots, x_j) = x_i$, for all x_i, i, j

by using the operations of composition and primitive recursion:

Composition:

$f(x_1, \dots, x_n) = g(h_1(x_1, \dots, x_n), \dots, h_m(x_1, \dots, x_n))$, for all g, h_1, \dots, h_m

Primitive Recursion:

$f(x, 0) = g(x)$, for any g

$f(x, s(y)) = h(x, y, f(x, y))$, for any h

The recursive functions are formed by the addition of the minimization operator, which takes a function f and returns h defined as follows:

Minimization:

$h(x_1, \dots, x_n) = y$, if $f(x_1, \dots, x_n, y) = 0$ and $\forall t < y (f(x_1, \dots, x_n, t) \text{ is defined and positive})$

$= \text{undefined otherwise.}$

It is known that the Turing computable functions are exactly the recursive functions.

6.2 Abacus Machines

Abacus machines abstract from the more familiar architecture of the modern digital computer (the von Neumann architecture). In its simplest form a computer with such an architecture has a number of addressable registers each of which can hold a single datum, and a processor which can read and write to these registers.

The machine can perform two basic operations, namely: add one to the content of a named register (which we will symbolize as $n+$, where n is the name of the register) and (attempt to) subtract one from a named register, with two possible outcomes: a success branch if the register was

initially non-zero, and a failure branch if the register was initially zero (we will symbolize the operation as n -).

These are called *abacus computers* by Lambek (Lambek 1961), and are known to be equivalent to Turing machines.

The modern digital computer is subject to finiteness constraints that we have abstracted away in the definition of abacus machines, just as we did in the case of Turing machines. Physical computers are limited in the number of memory locations that they have, and in the storage capacity of each of those locations, while abacus machines are not subject to those constraints. Thus some abacus-computable functions will not be computable by any physical machine. (We won't consider whether Turing machines and modern digital computers remain equivalent when both are given external inputs, since that would require us to change the definition of a Turing machine.)

7. Restricted Turing Machines

One way to modify the definition of Turing machines is by removing their ability to write to the tape. The resulting machines are called *finite state machines*. They are provably less powerful than Turing machines, since they cannot use the tape to remember the state of the computation. For example, finite state machines cannot determine whether an input string consists of some As followed by the same number of Bs. The reason is that the machine cannot remember how many As it has seen so far, except by being in a state that represents this fact, and determining whether the number of As and Bs match in all cases would require the machine to have infinitely many states (one to remember that it has seen one A, one to remember that it has seen 2, and so on).


Bibliography


- Barwise, J. and Etchemendy, J., 1993, *Turing's World*, Stanford: CSLI Publications.
- Boolos, G.S. and Jeffrey, R.C., 1974, *Computability and Logic*, Cambridge: Cambridge University Press.
- Davis, M., 1958, *Computability and Unsolvability*, New York: McGraw-Hill; reprinted Dover, 1982.
- Herken, R., (ed.), 1988, *The Universal Turing Machine: A Half-Century Survey*, New York: Oxford University Press.
- Hodges, A., 1983, *Alan Turing, The Enigma*, New York: Simon and Schuster.
- Kleene, S.C., 1936, "General Recursive Functions of Natural Numbers," *Mathematische Annalen*, 112: 727–742.
- Lambek, J., 1961, "How to Program an Infinite Abacus," *Canadian Mathematical Bulletin*, 4: 279–293.
- Lewis, H.R. and Papadimitriou, C.H., 1981, *Elements of the Theory of Computation*, Englewood Cliffs, NJ: Prentice-Hall.
- Lin, S. and Radó, T., 1965, "Computer Studies of Turing Machine Problems," *Journal of the Association for Computing Machinery*, 12: 196–212.
- Petzold, G., 2008, "The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and Turing Machines," Indianapolis, Indiana: Wiley.
- Post, E., 1947, "Recursive Unsolvability of a Problem of Thue," *The Journal of Symbolic Logic*, 12: 1–11.
- Radó, T., 1962, "On Non-computable functions," *Bell System Technical Journal*, 41 (May): 877–884.
- Turing, A.M., 1937a, "On Computable Numbers, With an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, s2-42 (1): 230–265; correction *ibid.*, (1938)


s2-43 (1): 544–546. [Note: This paper was received May 28, 1936 and read to the Society on November 12, 1936, but wasn't actually published until 1937.]


Turing, A.M., 1937b, “Computability and λ -Definability,” *The Journal of Symbolic Logic*, 2: 153–163.

Academic Tools

 How to cite this entry.

 Preview the PDF version of this entry at the Friends of the SEP Society.

 Look up this entry topic at the Indiana Philosophy Ontology Project (InPhO).

 Enhanced bibliography for this entry at PhilPapers, with links to its database.

Other Internet Resources

- “Turing Machines”, *Stanford Encyclopedia of Philosophy* (Summer 2003 Edition), Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/sum2003/entries/turing-machine/>. [This was the original version of the present entry, written by the Editors of the Stanford Encyclopedia of Philosophy.]
- The Alan Turing Home Page
- Bletchley Park, in the U.K., where, during the Second World War, Alan Turing was involved in code breaking activities at Station X.

Busy Beaver

- Michael Somos' page of Busy Beaver references.

The Halting Problem

- Halting problem solvable (funny)

Online Turing Machine Simulators

Turing machines are more powerful than any device that can actually be built, but they can be simulated both in software and hardware.

Software Simulators

There are many Turing machine simulators available. Here are three software simulators that use different technologies to implement simulators using your browser.

- Andrew Hodges' Turing Machine Simulator (for limited number of machines)
- Suzanne Britton's Turing Machine Simulator (A Java Applet)

Here is an application that you can run on the desktop (no endorsement of these programs is implied).

- Visual Turing: freeware simulator for Windows 95/98/NT/2000

Hardware Simulators

- Turing Machine in the Classic Style, Mike Davey's physical Turing machine simulator.
- Lego of Doom, Turing machine simulator using Lego™.
- A purely mechanical Turing Machine, Computer Science Department, École Normal Supérieure de Lyon. (This one has no embedded microprocessor.)

Related Entries

Church, Alonzo | Church-Turing Thesis | computability and complexity |
function: recursive | Turing, Alan

Copyright © 2016 by the author

David Barker-Plummer