Melampus

Solution to the Halting Problem

Introduction

In would seem like insanity to attempt the proof of a problem that has already been proven to be impossible to solve. But an impossibility proof, like any other proof, has assumptions, and if the assumptions are false, then the conclusion does not follow. The impossibility proof for the Halting problem is a proof that within a certain mathematical system, one that is computable, there cannot be a solution to the Halting problem. Hence, the impossibility proof is prima facie a proof that a computer cannot solve the Halting proof. As such, it does not directly relate to the human capacity to solve problems, and only can be made to so relate if one adopts the premise that human beings can only solve at most all those, and only those, problems that computers can solve. This assumption is equivalent to the very thing that strong AI must prove, so to assume it is circular and begs the question. The fact that it has been implicit in all the discussion hitherto, reflects the bias of the age.

The specific assumption made in the above approach is that number theory embraced by computable set theory – that there is no result of number theory that could not be equally proven in computable set theory. It is this assumption that is challenged in this paper.

Number theory is a theory of numbers subject to mathematical induction, which is a principle based on the potential infinite. On the other hand, in set theory there is no direct introduction of the potentially infinite. It is always assumed that number theory can be embedded within set theory, but that assumption can be challenged. Specifically, set theory is not characteristic of number theory, because it has only first-order axiom schema for mathematical induction, whereas number theory is equipped with a second-order axiom of complete induction. Secondly, set theory uses actually, complete infinities with are conceptually not identical to the potential infinite, and introduces these actual infinities through an Axiom of Infinity. Hence, there is prima facie reason to consider that Number Theory is not a sub-theory of (computable) set theory. If it is not, then it may be worth investigating whether the Halting Problem can be solved by means of mathematical induction, in this case, on the number of states of a Turing machine. No one can *a priori* discount such a possibility, and it remains simply to investigate the details.

That such a proof could be discovered is rendered more likely by the following consideration. Given all the 1 state Turing machines, can the Halting Problem be solved for all these? Clearly, yes. Then, consider the 2 state Turing machines – likewise, yes, these can all be solved. Any machine that would be programmed to solve the problem would certainly need more than 2 states, but that would be no bar to constructing such a machine. And so on, through 3, 4, 5, … state machines. Whenever mathematical induction fails, it fails at a specific number of states of machines. Hence, if the Halting Problem is not subject to proof by mathematical induction, at what specific number of states does it fail to be solvable?

Here we see that the reason why the Halting Problem cannot be solved by a computer is that the size of the computer to solve the problem for n state machines must be considerably larger than n. So, any machine that could solve the Halting Problem for all machines of any finite number of states would need infinite states. Since no computer can be infinitely large, no computer can solve the problem.

On the other hand, proof by mathematical induction is in effect a procedure for a given machine of n states to construct another of larger than n states that will solve the Halting problem. Thus, we can *know* that the Halting Problem is soluble for all n state machines without constructing a single machine that solves the problem for all machines. What is illustrated is that human knowledge is

different in kind from computational processes. The question, what do we know? Is not the same as, what do we compute? Proofs are sometimes algorithms, but most of the time, not.

Hash symbol

In this essay the hash symbol, #, denotes a definition or result new to mathematics, introduced in this paper.

The halting problem is not effectively computable

The *halting problem* is defined by Boolos and Jeffrey as follows: -

> **Definition, halting problem**
> We confine our attention to Turing machines which read and write only two symbols *B* and 1.[1] Any such machine *M* can be thought of as computing some total or partial function *f* from positive integers to positive integers as follows: To discover the value (if any) which *f* assigns to the argument *n*, start *M* in its lowest-numbered state, scanning the leftmost of a block of *n* 1s on an otherwise blank tape. If *M* eventually halts in a *standard configuration*, i.e. scanning the leftmost of a block of 1s on an otherwise blank tape, $f(n)$ is the number of 1s in that block. But if *M* never halts, or halts in some nonstandard configuration, $f(n)$. is undefined. (Boolos and Jeffrey [1980] p. 34).

The *halting problem* is the problem of designing an effective procedure for identifying Turing machines which never halt, once started in their lowest-numbered states on blank tapes.

> **Definition, productivity**
> Let *T* be a Turing machine of *n* states using only the symbols 0 and 1. Initially *T* scans only a blank tape. The machine *T* either halts in "standard configuration" - that is scanning the leftmost of an unbroken string of 1s on it otherwise blank tape, or it does not. If it does not it may either not halt at all, or halt scanning some other configuration. The productivity of *T* is defined to be: -

$$p(T)=\begin{cases}\text{the length of the string that } T \text{ scans if it halts in standard configuration}\\ 0 \text{ otherwise}\end{cases}$$

This is a function defined for each Turing machine. From this we may derive another function, $p(n)$ which is defined to be the productivity of the most productive *n*-state Turing machine.

> **Definition, the busy beaver problem**
> Design a Turing machine which computes the function *p*. The machine must read and write only the symbols 0 and 1.

[1] Boolos and Jeffrey [1980] use the symbol *B* where here we use 0.

The stipulation that the machine must read and write the symbols 0 and 1 is justified by the proposition that any function from positive integers to positive integers which is computable at all is computable in monadic notation by a machine which uses only the symbols 0 and 1.[2]

Proposition (Boolos and Jeffrey [1980] p. 30)
If the halting problem is solvable, then the function *p* is computable.
Proof
Suppose we have "a systematic procedure for identifying non-halters ... If we had such a procedure we could apply it to the graphs of all the *n*-state machines while those machines are working (having been started in their lowest-numbered states, on blank tapes). After some finite period of time, each machine will either have halted or have been identified as a non-halter, so that for each *n*, there would be some period of time after which we would know the productivity of every *n*-state machine. For each *n*, we would then be able to compute $p(n)$: the function would be computable in an intuitive sense if there were a systematic procedure for identifying non-halters." (Boolos and Jeffrey [1980] p. 30)

Result, properties of productivity
(Boolos and Jeffrey [1980] p. 35 et seq.)
1. $p(1)=1$

There are 25 different 1 state machines corresponding to 25 flow graphs of one node. By examination of cases we can show that some of these do not halt and of those that do the maximum productivity is 1.
2. $p(47)\geq 100$

It is possible to construct a machine of 47 states with productivity 100. Machines with 47 states do exist with greater productivity but this is not what is asked for in this result.
3. $p(n+1)>p(n)$

We can always add one more state to a given machine of *n* states that adds another 1 and has productivity $p(n)+1$.
4. $p(n+11)\geq 2n$

This is another instance of a concrete *n* state machine that has productivity $p(n+11)=2n$.
Proposition, The busy beaver problem is not effectively computable
There is no machine that can compute $p(n)$.
Proof
Assume that such a machine exists, and let it have *k* states. It would start scanning the leftmost of *n* 1s and finish scanning $p(n)$. Call this a "Busy Beaver Machine" (BB). Then there exists a machine that comprises a machine that writes *n* 1s followed by two copies of the BB machine.

$$\text{Write } n \text{ 1s} \longrightarrow \text{BB} \longrightarrow \text{BB}$$

This machine has $n+2k$ states and its productivity is $p(p(n))$. Then, assuming that BB exists we have: -

$$p(n+2k)\geq p(p(n)).$$

[2] For a proof of this proposition see Boolos and Jeffrey [1980] p. 30. In monadic notation numbers are inputted as strings of 1s, 0 (number) corresponds the string 1, and *n* to a $n+1$ string of 1s.

From the result $p(n+1) > p(n)$ it follows $p(i) > p(j)$ if $i > j$, from which it follows by contraposition that if $j \geq i$ then $p(j) \geq p(i)$. Let $j = n + 2k$ and $i = p(n)$, then this gives with $p(n+2k) \geq p(p(n))$ the conclusion $n + 2k \geq p(n)$. This remains true if *n* is increased by 11: -
$n + 11 + 2k \geq p(n+11)$.
But then $p(n+11) \geq 2n$ entails: -
$n + 11 + 2k \geq 2n$
$11 + 2k \geq n$
which is true for all *n*. But putting $n = 12 + 2k$ we obtain a contradiction. Therefore, BB does not exist.

Implications of a solution to the halting problem

The above theorem has shown that no Turing machine can solve the halting problem. However, it can be proven by complete induction that the halting problem is soluble.

Definitions, determined, complete criterion #
Let T_n represent a Turing machine. Then the statement: T_n *is determined* represents the property of T_n that for any given input configuration of finite information of the Turing tape with starting symbol *S* in state Q_i then it is determined:-

1. Whether T_n halts or does not halt when the machine is started with configuration in any of its internal states $Q_i, 1 \leq i \leq n$.
2. If the machine halts, then the terminal configuration is known.

The information contained in a solution to this problem is called a *complete criterion*.

There is a proviso to this statement. The original problem is in terms of a *standard configuration* – "scanning the leftmost of a block of *n* 1s on an otherwise blank tape". The way to solve the halting problem is to allow for a wider class of input configurations. However, we shall stipulate that this shall be a configuration that can be described by finite information. The finite configuration must indicate an assignment of 1s and 0s to the entire tape, but must do so in ways that can be expressed finitely. For example, the tape could contain the string 0 1 0 1 1 1 1 0 1 with the stipulation that both to the left and right it shall have an infinitely repeated sequence of 0 1. This assigns 1s and 0s to the whole tape, but does so on the basis of finite information.

Complete criterion theorem #
A complete criterion may be written for any Turing machine.

It is the business of this chapter to prove this theorem. Firstly, I illustrate its implications.

Corollary I #
The halting problem is soluble.

Proof, given the theorem
If we are able to solve the halting problem for *any* input configuration of finite information in *starting at any symbol and in any state* then we can solve the problem for machines once started in their lowest-numbered states on blank tapes starting in standard configuration.

Corollary II #
There is a soluble problem that is not Turing computable.

Outline of the proof of Complete criterion theorem

Prior to proving the Complete criterion theorem I first describe a method of annotating the flow diagrams of Turing machines and at the same time show how to classify all 1-state machines. After that I will illustrate the method of solution. Finally, I will prove the theorem by induction on the number of states *n* in a Turing machine T_n. Before embarking on this plan, it would be as well to identify the error in the thinking about this problem that makes one conclude that it is not solvable *on the grounds that it is not Turing computable*. Consider this discussion of the halting problem from Boolos and Jeffrey: -

> The question is, 'Why isn't *p* computable in some intuitive sense?' After all, there are only finitely many different graphs of *n*-state machines if we don't trouble to number any of the nodes except for node 1, the starting node. Then for each *n* we can (in imagination), anyway) set all the *n*-state machines going, starting in state 1 on a blank tape, and await developments. As time passes, one or another of the machines may halt, at which point we can see whether it is scanning the leftmost of an unbroken string of 1s on an otherwise blank tape. If it does halt in that standard positon, we find its productivity by counting the number of 1s in the string, but if it halts in a non-standard position we know that its productivity is 0. But there is a catch: some of the *n*-state machines may never halt, so that no matter how long we wait, it may be that the productivity of one or more of the *n*-state machines cannot be determined in the way we have just sketched. *Those* machines will have productivity 0 because they never halt. (Boolos and Jeffrey [1980] p. 40)

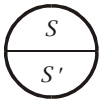
My underlining. The procedure described here is an *empirical procedure*. This is the error. What happens to a Turing machine is *wholly determined* by the states of the machines (its program or set of quadruples) and the configuration of the tape. It is not an empirical question but a combinatorial question. It is also possible to show that the solution to the halting problem constructs a generic set and no Turing machine can construct just such a set.

Classification of 1-state machines

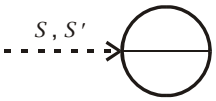
The classification of all 1-state machines is a routine matter. Here I give only so much of the classification as to illumine what follows. About this problem Boolos and Jeffrey [1980] comment: -

> ... there are infinitely many 1-state Turing machines which read and write no symbols other than *B* and 1, but these fall into just 25 distinct classes... (Boolos and Jeffrey [1908] p. 35)

They go on to solve the productivity problem for all 1-state machines. The classification adopted here is for the purpose of solving the halting problem, and does not follow their 25 classes. All 1-state machines may be classified depending on the number of arrows in the state symbol and how those arrows are attached. A 1-state machine starts scanning a determined symbol *S,* which may take one of two values, 0 or 1. Therefore it has two potential inputs and may act differently depending on that symbol and how the input arrows are attached, so we can represent these by dividing the state symbol in two. Let 0 and 1 be called the complements of each other and denoted thus, $0' = 1, 1' = 0$. Let $S \in \{0,1\}$ represent an arbitrary symbol and S' its complement.

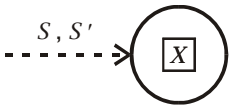


It shall be more convenient to represent this thus: -



A very simple 1-state machine is: -

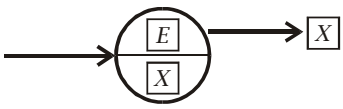
Q_0 One input arrow



No matter how we are lead into this state, it always halts for all configurations. I denote the *upshot* of this machine by $\boxed{X}$,which shall be called its *terminus*.

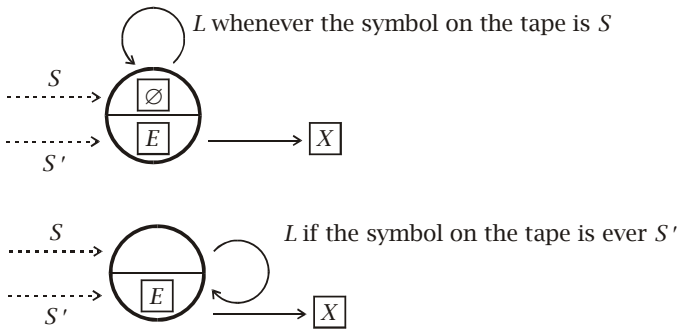
We say that a 1-state machine *has an exit*, denoted $\boxed{E}$, if on scanning a symbol *S* there is an arrow leading out of the state. A state may have infinite arrows leading into it, but as it can only respond to two symbols, we shall say that the maximum number of input arrows in any diagram is two.

$Q_{1,E}$ One input arrow, one exit.

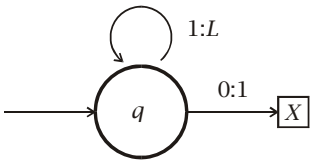


There is only one quadruple for this state and this leads to another state. One outcome has no exit, so is $\boxed{X}$. The other leads to an action and an exit, which is denoted $\boxed{E}$. The two potential inputs are differentiated. That is either 0 leads to $\boxed{E}$ and 1 leads to $\boxed{X}$, or conversely. As a 1-state machine this always halts because the exit arrow leads nowhere, and this is shown by the terminus $\boxed{X}$ at the end of the arrow leading out of the exit. There are other 1-state machines; we consider next the following machine: -

$Q_{2,E,L/R}$ Two arrows, one loop, loop action *L* or *R*. Here illustrated for the loop action *L*.



It is clear that a complete criterion could be determined for such 1-state machines. Here what these machines produce depends on the initial configuration, γ, of the Turing tape. To illustrate how to write a complete criterion for this 1-state machine, consider the specific machine, *q*: -



The machine will halt if it exits at *q*. It needs to be reading a 0 to do this. If a 0 is preceded by a finite sequence of 1s to the right and the machine starts by scanning one of these, then it will also exit. Otherwise, if the sequence of 1s is infinite, then it will not halt. This information can be presented in a table, which comprises the complete criterion for this machine.

Input at *q*
$X:0_q$
$X:0\overline{1}_q$
$\varnothing:\overline{1}_q$

Regarding how to read the information in this table: $X:0_q$ means that if in state *q* the machine is scanning 0 then it will exit at *X*.. $X:0\overline{1}_q$ means that if in state *q* the machine is scanning the last of a *finite* sequence of 1s followed by a 0, then it exits at X. $\varnothing:\overline{1}_q$ means that if in state *q* the machine is scanning an *infinite* sequence of 1s, then the machine will not halt (it loops indefinitely).

Asterisk convention #

The meaning of the first two lines of this complete criterion

$X:0_q$

$X:0\overline{1}_q$

can be combined into one line by an *asterisk convention*. Let 0_q^* denote either 0_q or $0\overline{1}_q$.

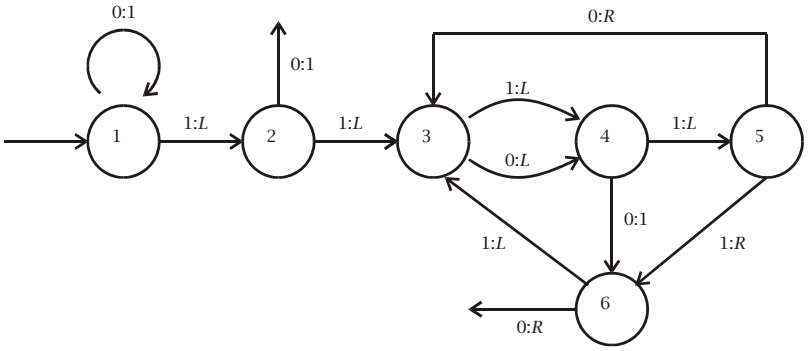
The point of this convention is that whether this machine exits depends solely on the presence of a 0 on the tape, and not on the 1s. So that information can be encoded in a single symbol.

The bar symbol #

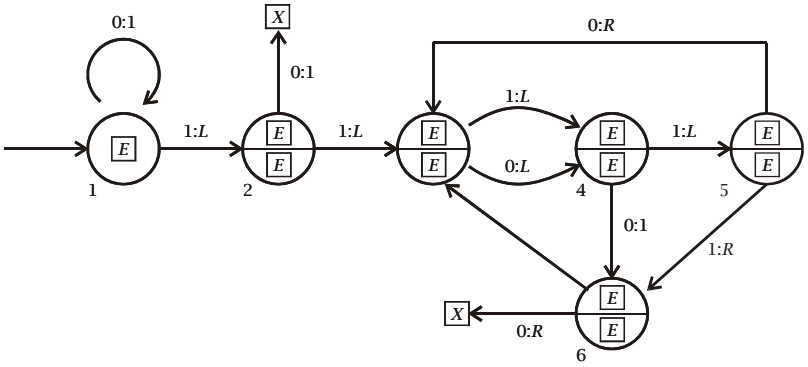
The bar symbol will be used to denote other recurring sequences in general as many state machines may enter into cycles, as shall be explained below. The meaning of the bar symbol depends on context. If it is on its own it represents an infinite sequence and shows what the recurring sequence on the Turing tape must be for the machine not to halt. If it is preceded (or in context succeeded) by another symbol (that must be different, otherwise that symbol is absorbed into the symbol under the bar) it means a finite sequence. (Whether the bar should be preceded or succeeded by another symbol is always clear and determined from context. It depends on whether the next action of the machine is a left or right move.) Although a machine can enter into an infinite loop (non-halting) the information to determine this is actually finite, and the bar represents this finite information and so determines the upshot of the machine.

Illustration of the method of solving the halting problem

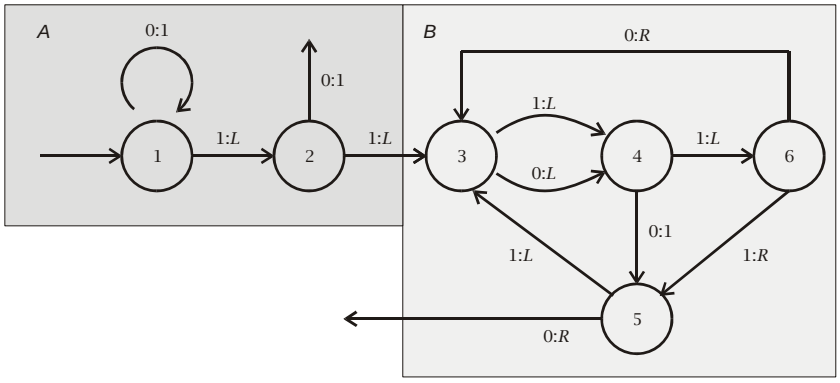
To determine the complete criterion for the following machine: -



The internal organisation of this machine may be annotated as follows: -



In this problem the machine divides into two parts: -



Machine *A* serves to constrain the input at *Q*3, and is clearly determinable in itself. Therefore, we can discard this part and concentration our efforts on machine *B*.

Definition, loop, cycle #

An E *cycle* is a closed path linking exits of individual states. A *loop* is a sub-cycle.

In the above machine (*B*) there are the following loops: -

$Q_3 \to Q_4 \to Q_5 (\to Q_3 \to \ldots)$
$Q_3 \to Q_4 \to Q_6 \to Q_5$
$Q_3 \to Q_4 \to Q_6$

Because of the double link (or "feed") between Q_3 and Q_4 each of these loops is potentially repeated twice.

Definition, maximal period, reduced period #
The *maximal period* of a loop is the number of separate states in the loop. The first and third of these loops has period 3 and the second has period 4. The *reduced period* or just *period* is the number of "moves" on the tape from some starting position required to read when the machine passes once through an entire loop.

For example, in the loop: -

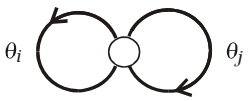
$$Q_3 \xrightarrow{1:L} Q_4 \xrightarrow{0:1} Q_5 \xrightarrow{1:L} Q_3$$

the machine moves to the left twice, so, although the *maximal period* is 3, the *period* is 2. For each of our loops in this case we have: -

loop	max. period	period
$Q_3 \to Q_4 \to Q_5$	3	2
$Q_3 \to Q_4 \to Q_6 \to Q_5$	4	2
$Q_3 \to Q_4 \to Q_6$	3	2

Definition, maximal cycle, indecomposable cycle #
A *maximal cycle* or *cycle* is any path through that takes in all of the independent loops of the machine once. An indecomposable cycle is a cycle that comprises just one loop so that every state in the cycle may be visited only once in any period.

Example
The following is not indecomposable: -



It comprises two loops of periods θ_i, θ_j respectively joined at one state, which may be visited twice in a single period (cycle).

Since each of the loops in our example is a double loop, the *cycle* is 2^6. In a machine of many parts, the practical process of solving the halting problem would be to "cut up" the machine into separate machines and write criteria for each one. A *cycle* is a sequence of states that, on account of the actions of the states (quadruples), the connections between the states and the configuration of the Turing tape at outset may be repeated. Machine *B* in our example consists of closed $\boxed{E}$ cycles except for the one exit at *Q*6. We solve this problem by creating *complete criteria* for the sub-machines[3] defined by: -

[3] The relevance of the following remark can only be appreciated from the reading of later chapters: It is significant that the description of these machines is with meets (conjunctions) rather than joins (disjunctions). It is this that

T_2 : $Q3 \wedge Q4 = (Q3, Q4)$

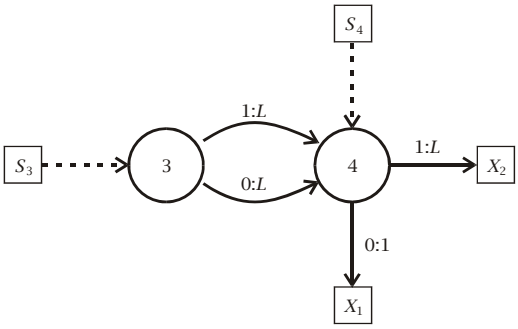
T_3 : $T_2 \wedge Q5 = (Q3, Q4, Q5)$

T_4 : $T_3 \wedge Q6 = (Q3, Q4, Q5, Q6)$

Beginning with the simplest of these machines T_2 .

Solution for T_2

There are possible inputs at Q_3 and Q_4 , which we denote S_3 and S_4 respectively. Hence we have the following diagram: -



There are two possible exits (halting configurations), denoted X_1, X_2 . T_2 has no cycles whatsoever, so it always halts. The aim is to provide a *complete criterion* to determine, for the inputs S_1, S_2 exactly when and how the machine will halt at terminals X_1, X_2 . The problem is represented schematically by the following diagram: -

S_3	S_4
Criterion, T_2	
X_1	X_2

There are two methods that will both completely determine the criterion: -

Definition, entry method #
Method of test input data.

Definition, exit method #
Method of tracing back from test data at the exits.

I also call these the *methods of inputs* and *exits* respectively. Both methods create finite *trees* [See Kay [2007] or Smullyan [1995] for definition). The trees are proofs that a given machine, T_n , is determinate. Here I illustrate the method of test inputs at S_3, S_4 . The tree structures that these generate must be finite because there are only finite states in the machine. In this case there is no cycle, but even if there were, such cycles and/or their permutations, would be finite.

---

indicates that we are proceeding down a poset of finite conditions in the opposite direction to an analytic logic. See Chapter 8 on generic sets – specifically section 8.2.12.

Testing at S_3

0_3		1_3	
0_40	1_40	0_41	1_41
X_1	X_2	X_1	X_2

To explain this diagram. An input of 0 on the tape at S_3 is shown as 0_3. The aim is to show which inputs at S_3 and S_4 lead to which terminals. The diagram shows the result of systematic testing of inputs at S_3. $X_1 : 0_40$ means, for example, and by working backwards, that an input of 00_3 at S_3 terminates at X_1.

Testing at S_4

0_4	1_4
X_1	X_2

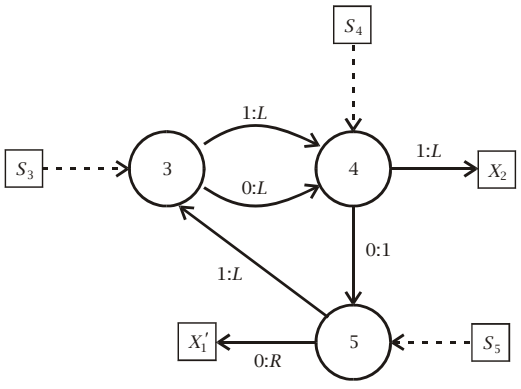
So the machine T_2 is completely determined by the following complete criterion: -

$\boxed{T_2}$

S_3	S_4
$X_1 : 00_3\left(H = \boxed{1}0\right)$	$X_1 : 0_4\left(H = \boxed{1}\right)$
$X_1 : 01_3\left(H = \boxed{1}1\right)$	$X_2 : 1_4\left(H = \boxed{\ }\right)$
$X_2 : 10_3\left(H = \boxed{\ }10\right)$	
$X_2 : 11_3\left(H = \boxed{\ }11\right)$	

H here indicates the *halting configuration* at the given terminus. The box indicates the scanned square of the tape. A blank box indicates that the symbol on the square has not been determined – it could be either 0 or 1. For example, $H = \boxed{\ }11$ arises as a result of an *L* move on the tape and so means that the symbol scanned is determined by whatever was originally on the tape. We progress to determine the complete criterion for $T_3 = (T_2, Q5)$.

Solution for T_3



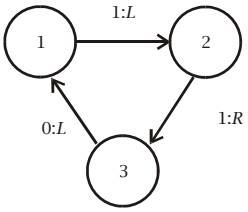
We have converted the original terminus $\boxed{X_1}$ in T_2 to an exit, $\boxed{E}$ to which we have joined the state *Q*5. This has one loop back into the machine T_2, which creates a potential closed cycle; there is a new terminus, labelled X_1'. The symbol $\boxed{\varnothing}$ shall denote an infinitely recurring cycle – that is, a non-halting configuration.

Cycles and loops #

Cycles are concerned with the possibility of a non-halting configuration of the tape cycling in this case through the states $Q_3 \to Q_4 \to Q_5$. The machine will never halt if there is an infinite repetition of some configuration on the tape. The problem is to determine this configuration. The repeat configuration is always finite because there are only finite states in the cycle. Therefore it can be determined. In this example, because of the double feed between 3 and 4, this is in fact two *loops* of one *cycle*. A *loop* is a sub-cycle. There are three states in this cycle, so the maximum period is 6. The *period* of a cycle (or double loop) is the number of digits that can be repeated in a tape configuration that force the machine to indefinitely proceed through the loop without exit. In this example as there is one change of symbol $(0:1)$ this is reduced to 4. Both moves (actions) in the cycle are *L*, hence each loop has a period of 2. Had there been an *L* and an *R* these moves would cancel out so the period of one loop would be 1.

Definition, tape contradiction #

A *tape contradiction* is any assignment of a configuration to the tape that requires the placing both a 0 and a 1 at any point. A *well-defined loop* is a loop that does not presuppose a tape-contradiction. An *illusory loop* is a loop that is not well-defined.



An input of 1 at *Q*1 leads to a contradictory assignment at *Q*3, so this is an illusory loop.

Result, existence of non-halting configurations #

If a machine has a well-defined cycle, then there exists at least one infinite recurring string of symbols with the period of the cycle for which the machine does not halt.

Proof

Starting at a given state match each state to the symbol that causes the machine to exit at that state and travel around the loop. Once you have returned to the original state you have determined a non-halting configuration at that state.

It should be remarked that the determination of a non-halting configuration is always a *finite problem*. The non-halting configuration is a finite sequence of 1s and 0s on the Turing tape that must be repeated indefinitely. This may be encoded as a finite piece of information. It is finite information because the period of the cycle is finite.

Result, decomposition of cycles, indecomposable loops #

Every cycle may be *decomposed* into *indecomposable loops*. The period a cycle is the sum of the period of its indecomposable loops.

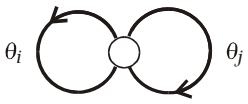
Proof

Proof is by induction on the composition of loops.

Let $\theta(C)$ denote the period of a cycle, *C*.

Let a cycle *C* comprise just one closed E-cycle (loop), L_0, of period $\theta_0 = \theta(L_0)$. Then *C* is indecomposable and $\theta(C) = \theta_0$.

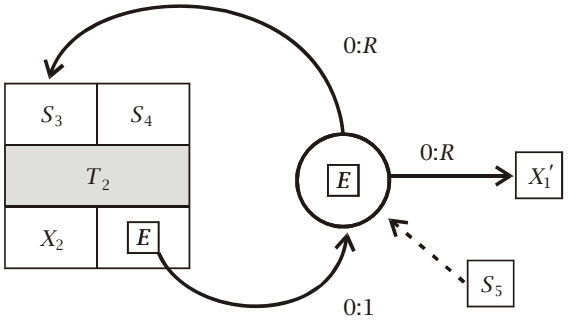
Now let *C* be a cycle that joins loops L_i and L_j at one state, with periods θ_i, θ_j respectively.



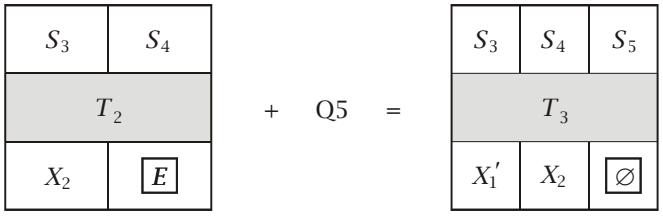
Then the period of *C* is $\theta_i + \theta_j$. By infinite descent the process may be iterated on the loops L_i and L_j until indecomposable loops are obtained.

To continue with the resolution of the problem for T_5. Q_5 has the form: -

but in the diagram that follows this has been simplified. The result of adding *Q5* to the machine T_2 may be represented schematically as follows: -



So the problem has the form: -



In the criterion for T_2 this has the effect of deleting certain lines, that will be replaced by criteria for either X_1' or $\boxed{\varnothing}$. In this case we use the method of exits, which is here the more efficient method. At each stage of the backtracking there should be a test for a tape contradiction. Working backwards from X_1' we obtain: -

Exit at X_1'
0_5

This means that only an input of 0_5 at S_5 exits at X_1'. That is the only way to get out of the loop at this point. When we are using the method of exits we are inverting the machine, and it is a useful technique to draw the inverted machine; that is, the machine obtained by reversing all the instructions.

Exit at X_2				
S_4	1_4			
S_3	10_3		11_3	
S_5	101_5		111_5	
S_4	100_4		110_4	
S_3	1000_3	1001_3	1100_3	1101_3
S_5	10001_5	10011_5	11001_5	11011_5

The above table shows the halting configurations but we need also to determine those configurations that if repeated infinitely lead to the infinite cycle $\boxed{\varnothing}$. This is a finite problem because the period of the repeat cycle is at most four digits. For example, in the case of the input at S_3 there are 16 possible 4 digit inputs. The above table shows that 12 of these are halting configurations. The non-halting configurations can be represented by a bar, thus: $\overline{0000_3}$. In the case of the input at S_5 we have $\overline{00}1_5$, so we need to clarify why the bar sometimes extends over the originally scanned symbol and sometimes not. To determine this we work by using the method of inputs to test the non-halting configurations. In the case of the S_3 input 0000_3 the output is $\square_3 1010$ so we must place the repeat configuration into the box: $\boxed{0000}_3 1010$, or originally at S_3 $\overline{0000}\,\overline{0000}_3$. In the S_5 case the input 001_5 has output $\boxed{1}_5 01$. We must place the repeat configuration into the box, to give $\boxed{001}_5 01$, which corresponds to an original input at S_5 of $\overline{00}\,\overline{00}1_5$. The complete criterion for T_3 is: -

$\boxed{T_3}$

S_3	S_4	S_5
$X_2 : 10_3 \left(H = \square 10\right)$	$X_2 : 1_4 \left(H = \square\right)$	$X_1' : 0_5 \left(H = 0\square\right)$
$X_2 : 11_3 \left(H = \square 11\right)$	$X_2 : 100_4 \left(H = \square 101\right)$	$X_2 : 101_5 \left(H = \square 101\right)$
$X_2 : 1000_3 \left(H = \square 1010\right)$	$X_2 : 110_4 \left(H = \square 111\right)$	$X_2 : 111_5 \left(H = \square 111\right)$
$X_2 : 1100_3 \left(H = \square 1110\right)$	$\varnothing : \overline{000}_4$	$\varnothing : \overline{001}_5$
$X_2 : 1001_3 \left(H = \square 1111\right)$	$\varnothing : \overline{010}_4$	$\varnothing : \overline{011}_5$
$X_2 : 1101_3 \left(H = \square 1111\right)$		
$\varnothing : \overline{0000}_3$		
$\varnothing : \overline{0001}_3$		
$\varnothing : \overline{0100}_3$		
$\varnothing : \overline{0101}_3$		

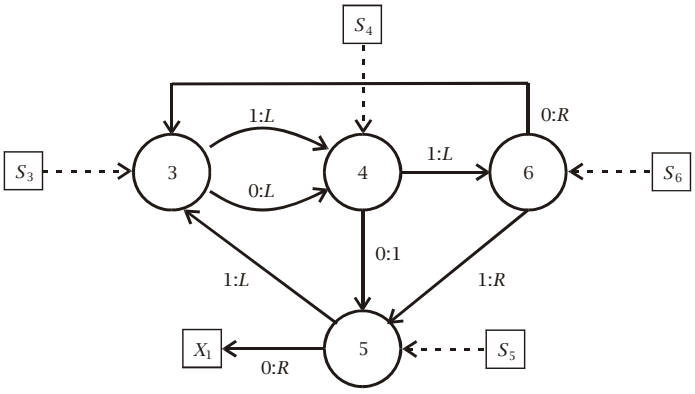
Any of the non-halting combination of choices of non-halting configurations will be non-halting. For example, $\overline{00000001010001013}$ is non-halting; likewise, $\overline{0000}\,0001_3$ is non-halting. The above table provides a complete criterion for T_3 and the problem for T_3 is solved.

Halting configurations #

About the representation of halting configurations. For example, T_3 machine will halt and exit at X_2 if at any stage it's configuration is 1000_3. It can only reach such a configuration after a finite number of moves, so the solution is *completely determined*. Nonetheless, it is useful to have some notation for the representation of halting configurations that are combined with repetitions of configurations that would, if repeated indefinitely, be non-halting. For example, in $1000\,\overline{0000}_3$ the overbar represents *not* an infinite repeat of the string $\overline{0000}_3$, which would be non-halting, but a finite yet indeterminate repetition of that sequence. Because these repeat configurations can be concatenated in any way, the following are also examples of halting configurations:

$1000\ \overline{0000}\ \overline{0001}_3$
$1000\ \overline{\overline{0000}}\ \overline{\overline{0001}}_3$
$10\,\overline{01}\,\overline{00}\,0_4$

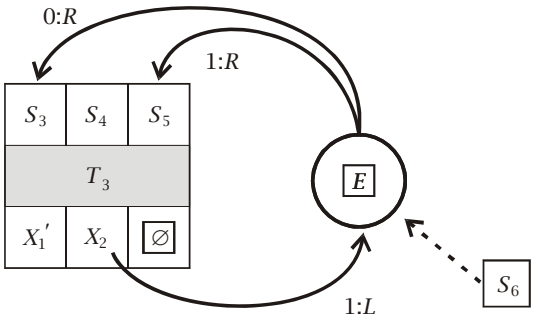
So we must understand the table – it encodes an infinite amount of information that would enable one to determine *for any finite tape configuration whatsoever*, whether it halts or not.

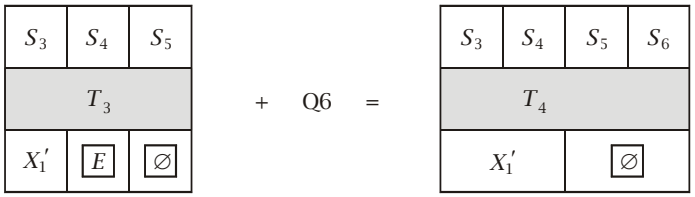


There are five separate loops: -

$Q_3 \to Q_4 \to Q_5$	two loops	period 2
$Q_3 \to Q_4 \to Q_6$	two loop	period 2
$Q_3 \to Q_4 \to Q_6 \to Q_5$	two loops	period 2

The period of the maximal cycle is $6 \times 2 = 12$. This means that by the method of entries (forward testing) we need to consider 2^{12} different strings. So this method is very inefficient in comparison to the backwards method – which identifies all the exit configurations and then leaves all the others as cyclic configurations. The problem has the form: -





$\boxed{T_3}$

S_3	S_4	S_5
~~$X_2:10_3\ (E=\square 10)$~~	~~$X_2:1_4\ (E=\square)$~~	$X_1':0_5\ (E=0\square)$
~~$X_2:11_3\ (E=\square 11)$~~	~~$X_2:100_4\ (E=\square 101)$~~	~~$X_2:101_5\ (E=\square 101)$~~
~~$X_2:1000_3\ (E=\square 1010)$~~	~~$X_2:110_4\ (E=\square 111)$~~	~~$X_2:111_5\ (E=\square 111)$~~
~~$X_2:1100_3\ (E=\square 1110)$~~	$\varnothing:000_4$	$\varnothing:001_5$
~~$X_2:1001_3\ (E=\square 1111)$~~	$\varnothing:010_4$	$\varnothing:011_5$
~~$X_2:1101_3\ (E=\square 1111)$~~		
$\varnothing:0000_3$		
$\varnothing:0001_3$		
$\varnothing:0100_3$		
$\varnothing:0101_3$		

Every input to Q_6 leads back into the system, so the only way to get out of the loop is via X_1'. We can use the forward method in this case to test each exit string. By this method, excepting the input 0 at S_5, every other input takes one back into the loop, so it is not possible to exit this cycle, except via inputs at S_5 and S_6. We find these by the method of exits.

Exit at X_1'		
S_5	0_5	
S_6	$1_6 0$	
S_4	11_4	contradiction

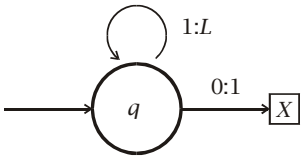
To explain the contradiction: Suppose we exit at *Q*6. Then the tape is $\boxed{0}_6$. The only possible route in is the 1:*R* from *Q*5 because the other route definitely writes a 1 at *Q*6. Then at *Q*5 we have $0\ \boxed{1}_5$ giving also $\boxed{0}_6\ 1$. The only entry to *Q*5 is from *Q*4. So at *Q*4 we have $\boxed{1}_4\ 1$ which is inconsistent with $\boxed{0}_6\ 1$. In conclusion, we have the following complete criterion for T_4: -

T_4

S_3	S_4	S_5	S_6
$\varnothing$	$\varnothing$	$X_1' : 0_5\left(H = 0\square\right)$	$X_1' : 1_6 0\left(H = 10\square\right)$
		$\varnothing : 1_5$	$\varnothing$: otherwise

Second example

We proceed to examine a second example illustrating the method of determining the criterion for machines of many states that involve 1-state machines of the type:- -



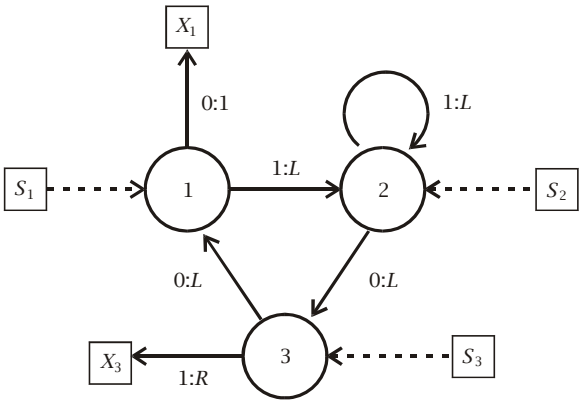
Input at q

$X : 0_q$

$X : 0\,1_q$

$\varnothing : 1_q$

This adds a loop of period 1. We consider the following machine, T_5



This can be solved by the method of inputs or exits; the method of exits is more efficient.

Exit at X_1	
S_1	0_1
S_3	00_3
S_2	000_2^*
S_1	$000_2^* 1_1$

Concerning the introduction of the asterisk symbol * into this table: Observing Q_2 we see that the machine only exits this state *if it is scanning a 0*. If it starts by scanning any finite sequence of 1s then it moves all the way to the left until it reaches a 0. If there is an infinite sequence of 1s on the tape then the machine does not halt. The asterisk * encodes all this information. Fo r example 000_2^* means that *both* 000_2 and $000\overline{1}_2$ are configurations that exit at X_1. Thus, the asterisk shows us that the information encoded in a potentially infinite loop is really *finite information*. It is this *reduction of the infinite to the finite* that enables us to resolve the halting problem. The other test gives: -

Exit at X_3
$S_3 \quad 1_3$
$S_2 \quad 10^*_2$
$S_1 \quad 10^*1_1$
$S_3 \quad 10^*10_3$
$S_2 \quad 10^*100^*_2$
$S_1 \quad 10^*100^*1_1$

This gives the following complete criterion for T_5.

T_5		
S_1	S_2	S_3
$X_1 : 0_1$	$X_1 : 000^*_2$	$X_3 : 1_3$
$X_3 : 10^*1_1$	$X_3 : 10^*_2$	$X_1 : 00_3$
$\varnothing : \overline{1}_1$	$\varnothing : \overline{100^*_2}$	$X_3 : 10^*10_3$
$\varnothing : \overline{00^*1_1}$		$\varnothing : 00^*10_3$

The $\overline{00^*1_1}$ here means that both $\overline{001_1}$ and $\overline{00\overline{1}_1}$ are non-halting loops. Note also that in determining the non-halting configurations the asterisk reduces the number of possible inputs to test. For example, 00111_1 is an instance of $\overline{00^*1_1}$, which is why 111_1 does not appear in the list under S_1.

Because whatever happens to a Turing configuration scanned by a Turing machine is both *finite* and *fully determinate*, it is possible in principle to determine the complete criterion for any Turing machine. The information is finite because the infinitely repeating configurations may be encoded by finite information, here represented by the bar and asterisk in these worked examples. We now proceed to formally demonstrate the truth of this assertion by mathematical induction.

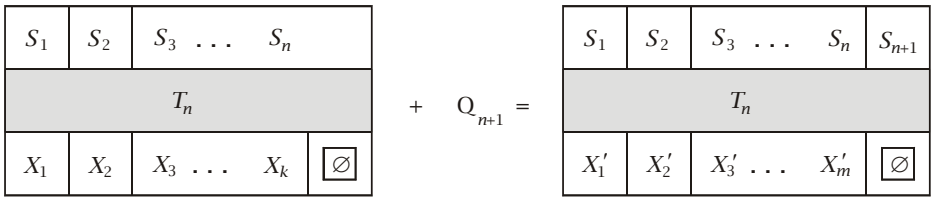
The solution to the halting problem

Complete criterion theorem #
A complete criterion may be written for any Turing machine.

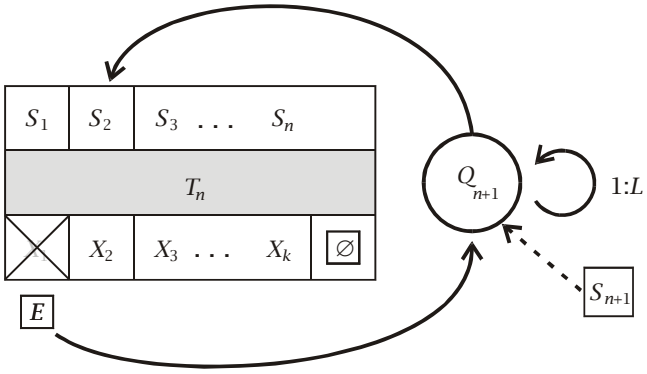
Proof
The proof is by complete induction.

Particular step
For $n = 1$. It is evident from the classification of all 1-state machines that there is a complete criterion for each. It is also solved by Boolos and Jeffrey [1998] on p. 35.

Induction step
The problem takes the form



As there are only a finite number of states in T_n the maximum number of exits is finite. At most, if every interior state of T_n has a double exit, then there are $2n$ exit symbols in T_n, but, assuming each state is connected to at least one other, only $n+1$ of these could be available for attachment to Q_{n+1}. The induction hypothesis is that the criterion for T_n is completely determined. This means that for any configuration with T_n starting in any of the states $Q_1, Q_2, \ldots, Q_n$ (corresponding to inputs $S_1, S_2, \ldots, S_n$) we know exactly which configurations lead to which termini $X_1, X_2, \ldots, X_k$. In attaching T_n to Q_{n+1} we change one or more of the termini in T_n to exits $\boxed{E}$ and direct these towards Q_{n+1}. Let the number of these attachments be κ. We may loop back from Q_{n+1} to one or at most two inputs $S_1, S_2, \ldots, S_n$ in T_n; Q_{n+1} may introduce one or two termini; finally Q_{n+1} may introduce a 1-cycle. An example of a T_{n+1} machine is: -



This lengthens the period of a loop by at most 2 and adds 1-cycle. However, the 1-cycle is completely determined under the asterisk convention – in other words by a single piece of information.

<u>By the method of inputs</u>

It may be that the halting configurations given at those termini attached to Q_{n+1} already fully-determine the effect of Q_{n+1} on the tape configuration and hence lead to a complete determination of T_{n+1} via the one or two inputs back to T_n. If they do not, then the length of the inputs at $S_1, S_2, \ldots, S_n$ must first be increased by every permutation of however many digits is required, including asterisks and bar symbols. The presence of asterisk and bar symbols increases the number of permutations needed, but this problem is finite because the attachment of κ, finite, additional loops lengthens each loop by at most 2. By connection through Q_{n+1} back to T_n this means that we must at most lengthen the inputs at $S_1, S_2, \ldots, S_n$ by a finite number of digits, and listing all necessary inputs is a problem in finite combinatorics. Then by running each of these through T_n and thence through $\boxed{E}$ to Q_{n+1} we determine in each case whether they (1) return to the exits in T_n or (2) halt at the remaining termini in T_n, or (3) enter the cycle, $\varnothing$ in T_n, or (4) halt at any termini in Q_{n+1}, or (5) enter the 1-cycle in Q_{n+1} if there is one, or (6) enter any of the new cycles defined by the route: $T_n \to \boxed{E} \to Q_{n+1} \to S_i$. Hence the complete criterion for T_{n+1} may be determined.

<u>In summary</u>

Let “the problem for ...” be short for “the problem of determining the complete criterion for ...”. Then the proof is as follows: Let the problem for T_n be finite. We obtain T_{n+1} by attaching Q_{n+1} to T_n. The problem for Q_{n+1} is finite. Therefore the problem for T_{n+1} is finite. This is a closure property: the property “the problem for T_k is finite” is closed under the addition of finite information.

By the method of exits

The method of exits works by tracing back information from every terminus through the machine $T_{n+1} = T_n + Q_{n+1}$. In doing so we work around each loop in T_{n+1}, recording any 1-loop in it by an asterisk and each longer loop by a bar symbol. These encode the possibility of a finite repetition of a configuration leading to an exit (halting configuration) as well as identifying the infinitely recurring non-halting configurations. The problem is finite if the period of the maximal cycle in T_{n+1} is finite. But if the period of the maximal cycle in T_n is finite then the addition of Q_{n+1} adds a finite number of loops to the maximal cycle; and the resultant maximal cycle and its period remain finite. Therefore, the problem can be solved by the method of exits for T_{n+1}.

"Paradox" and resolution

Since the procedure here looks like a computable algorithm there is a question as to how it is possible to show that the halting problem can be solved when it is also known that no algorithm can solve it?

Definition, unary notation

In *unary* notation a number *n* is represented by a string of 1s separated from other such strings by 0s. 0 is represented by a single 1; and *n* is represented by an $n+1$ string of 1s.

Definition, Gödel number of a Turing machine

Gödel numbering is a function from formulas to numbers. Its purpose is to encode information into a numerical form. We begin by encoding the symbols and actions of a Turing machine T_n thus: -

$\lceil 0 \rceil = 0 \qquad \lceil 1 \rceil \to 1 \qquad \lceil L \rceil \to 2 \qquad \lceil R \rceil \to 3$

Let the quadruples of T_n be arranged in ascending order: -

$1 \quad 0 \quad \sigma_1' \quad q_1'$
$1 \quad 1 \quad \sigma_1'' \quad q_1''$
$2 \quad 0 \quad \sigma_2' \quad q_2'$
... ...

and then let them be concatenated, thus: -

$1\ 0\ S_1'\ q_1'\ 1\ 1\ \sigma_1''\ q_1''\ 2\ 0\ \sigma_2'\ q_2'$ ...

Denote the *k*th symbol in this list by S_k. Let $2, 3, 5, \ldots, p_i, \ldots$ be an enumeration of all primes. Then the Gödel number of T_n is $\lceil T_k \rceil = 2^1 \cdot 3^0 \cdot 5^{\lceil \sigma_1' \rceil} \cdot \ldots \cdot p_k^{\lceil \sigma_k \rceil}$.

The procedure at first glance

Let $\lceil T_n \rceil$ be the Gödel number of T_n. A recursive procedure would start with $\lceil T_n \rceil$ in unary notation and determine every halting configuration for it.

1. Factorise $\lceil T_{n+1} \rceil$ to recover the quadruples that represent the machine T_{n+1}. The last of these is the state Q_{n-1} adjoined to the machine T_{n-1} which is the collection of the remaining quadruples. Each state Q_i in the machine table corresponds to a potential input configuration, S_i.

2. Determine the exits and loops. Determine the period η of the maximal cycle in T_n including any indeterminate repeats, 0^* or 1^*.

3. Then either

   3.1 Use the method of exits to trace back to the halting and non-halting (loop) configurations.
   3.2 Assuming the criterion for T_{n-1} is available use the method of inputs of period η to determine the halting and non-halting (loop) configurations.

4. Write the criterion for T_n. Denote this $\mathbf{T}_n$. Denote the procedure / putative algorithm by Θ. Then we have $\Theta(\lceil T_n \rceil) = \lceil \mathbf{T}_n \rceil$.

In the discussion that follows this is referred to as "*the* $\lceil T_n \rceil$ *procedure*".

Definition, Turing configuration #

A *Turing configuration* of the Turing tape is a tape comprising only a block of consecutive 1s with blank tape (0s) on either side.

Example

… 0 0 0 1 1 1 1 1 1 1 1 0 0 0 … is in Turing configuration.

Definition, Abacus configuration

An *Abacus configuration* of the Turing tape is one where there is a finite collection of separate blocks of 1s each of which is separated from adjacent blocks by just one 0, the rest of the tape being blank (comprising just 0s).

Example

… 0 0 0 1 1 1 1 1 1 1 1 0 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 0 …

is in Abacus configuration.

The significance of these definitions is explained by the following from Boolos and Jeffrey: "In contrast to a Turing machine, which stores information symbol by symbol on squares of a one-dimensional tape along which it can move a single step at a time, a machine of the seemingly more powerful 'ordinary' sort has access to an unlimited number of *registers* $R_0, R_1, R_2, \ldots$, in each of which can be written numbers of arbitrary size." (Boolos and Jeffrey [1980] p. 54.) The purpose, then, of the contrast between the Turing and the Abacus configurations is to simulate within a Turing machine the manner of which data is inputted in an Abacus machine. The different portions of the Turing tape correspond in Abacus configurations to the registers of the Abacus machine. Note that Boolos and Jeffrey describe the correspondence between registers and portions of the Turing tape in Boolos and

Jeffrey [1980] (p.62). We need to know nothing more about Abacus machines[4], save the following theorem.

Theorem
A total or partial function is Abacus computable iff Turing computable.

Proof
This is proven in Boolos and Jeffrey [1980] – chapters 6, 7 and 8.

I aim to explain why the procedure given above is *not Turing computable* by observations on the nature of the data it processes *in the Turing configuration*. It can be argued that the data should be initially given in Abacus configuration. The above theorem shows that if the procedure is not computable in the Turing configuration then it is not computable in the Abacus configuration, so the objection is anticipated.

In the $\lceil T_n \rceil$ procedure the input information is the Gödel number $\lceil T_n \rceil$. Assuming that the $\lceil T_n \rceil$ procedure is Turing computable, this is presented to the machine in Turing configuration, as a *very large block of 1s.* In order for the process to go further this block is factorised; this converts the Turing configuration into Abacus configuration.

Definition, overhead
Let *f* be any total or partial function. In this context *overhead* shall denote the least number of states in the most efficient Turing machine to compute *f*.

The phrase "large overhead" is intended to convey the (imprecise) notion that the overhead of a machine is a large number in some context which gives subjective meaning to the idea of "large" as opposed to "small". A function with a "large overhead" requires a program with "lots and lots" of states.

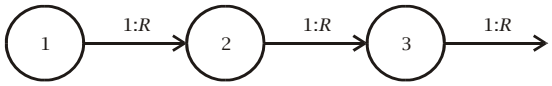
The Busy Beaver and the "factorisation problem"

Now we return to the Busy Beaver problem. Recall that the theorem in question is: There is no machine that can compute $p(n)$, where $p(n)$ is the productivity of an *n* state machine. The "Busy Beaver Machine" (*BB*) is a hypothetical machine that can compute $p(n)$ for arbitrary *n*. It was shown above that it was impossible that such a machine could exist. But let us pretend that *BB* exists and *try* to design it. Then it would compute $p(n)$ by starting off by scanning the leftmost of a string of $n+1$ 1s on an otherwise blank tape. From thence it must construct every Turing machine of *n* states. This is a problem in combinatorics with a large overhead. Let us assume that the overhead is at least as much as the overhead of the $\lceil T_n \rceil$ procedure that was outlined above to solve the problem of the complete criterion of any machine. This $\lceil T_n \rceil$ procedure begins by factorising $\lceil T_n \rceil$. So let us assume that if *BB* exists then its overhead is at least as great as the overhead involved in factorising $\lceil T_n \rceil$. We have $\lceil T_n \rceil = 2^1 \cdot 3^0 \cdot 5^{\lceil \sigma_1 \rceil} \cdot \ldots \cdot p_k^{\lceil \sigma_k \rceil}$ where p_i is the *i*th prime number. This process of factorisation decomposes $\lceil T_n \rceil$ into Abacus configuration, where each fourth register represents the number of an inner state of T_n. So there are at most $\frac{k}{4}$ states in T_n. In order to factorise this number *BB* must be equipped with sub-routines that enable it to divide $\lceil T_n \rceil$ by every prime number $\leq p_k$. Any such sub-routine that divides by p_i is a Turing machine with at least *i* states. That is, its overhead is at least *i*.

[4] Details are in Boolos and Jerfrey [1980].

This is because the division process in a flow diagram involves the iterated movement to the right (or left) *i* times, and then updating some other kind of register.

Machine to move right 3 times



So a machine to factorise $\lceil T_n \rceil$ must have ≥ 2 states in order to divide by 2; then ≥ 3 in order to divide by 3; and so on. So *BB* has at least $2+3+5+\ldots+p_k$ states, where $p_k > k$. Let $\lceil BB \rceil$ be the Gödel number of *BB*. Then *BB* could never factorise $\lceil BB \rceil$. That is *BB* could never solve its own halting problem. To be sure, this conclusion is based on the assumption that the overhead of *BB* is at least as great as the overhead involved in factorising $\lceil T_n \rceil$. But we can also prove this, because we know that *BB* in fact *does not exist.*

Theorem

Let *BB* be a machine that can compute the productivity $p(n)$ for at least some *n*. Then *BB* could never compute its own productivity. I here repeat this argument.

Proof

Let us first recall the proof that the universal *BB* machine does not exist. Assume *BB* exists. Then: -

$$p(n+2k) \geq p(p(n))$$
$$n+2k \geq p(n)$$
$$n+11+2k \geq p(n+11)$$
$$n+11+2k \geq 2n$$
$$11+2k \geq n$$

This is true for all *n*. But putting $n = 12+2k$ we obtain a contradiction. Therefore, *BB* does not exist.

In this context I shall call this the *Boolos argument*. The assumption in this Boolos argument is that the *BB* machine can *solve the halting problem for all machines* – there is no limitation to the number of states *n* for which the BB machine can compute $p(n)$. Let *m* be the number of states in *BB*. On this assumption *BB* can compute $p(m)$, and by substituting *m* for *n* in the above argument, we obtain the contradiction. Hence *BB* cannot compute $p(m)$, which is its own productivity.

Hence a *BB* machine *can exist*, provided that it has an upper limit *N* on the size of the machine for which it can solve the productivity problem. That is to say, the contradiction argument given above proves that no *universal BB* machine can exists, but it does not prove that there could not be a succession of *BB* machines, each of increasing overhead, that could solve the productivity problem for machines with larger and larger numbers of inner states. Then for any given machine of *n* states there exists a BB machine of $\mu(n)$ states, where μ is an increasing function, that can solve the productivity problem for any $n \leq N$. That is, *N* is its limit, and the *BB* machines are indexed: BB_μ. Formally: -

For every *BB* machine, BB_μ, of $\mu(n)$ states, BB_μ computes $p_\mu(n)$ iff $n \leq N = \max(n)$ for some $N \in \mathbb{N}$.

Note that here the productivity function that BB_μ computes has also become indexed: p_μ. Then following through the Boolos argument with $n = N$ we obtain: -

$$p_\mu(N + 2k) \geq p_\mu(p_\mu(N))$$

This line is already invalid, because $p_\mu(N + 2k)$ is not defined.

We see why a *universal BB machine* could not exist. Every machine is in fact *limited in size*. There will be a maximum size of number that such a finite machine BB_μ that uses the $\lceil T_n \rceil$ procedure could factorise, so after this point BB_μ would just simply *not work*. Even if we allow that the inputs are given already in some abacus form, then the size of the computation must place a practical upper limit on the size of the machine that BB_μ could analyse for halting configurations. If BB_μ exists for some machine T_n then there could exist *another BB* machine of states $> \mu$ that can analyse T_μ.

The impossibility of a super beaver machine

Suppose that there is a machine that can build all *BB* machines; let us call this machine "worker", *WK*. Each BB_μ has a maximum input N_μ, and for $n > N_\mu$ the output is undefined, and computes p_μ subject to the rule $p_\mu(N) > N$. Let *WK* build a sequence of machines BB_{μ_0}, BB_{μ_1}, … , BB_{μ_j}, …. This entails that $WK(BB)$ is a "super beaver" machine, *SB*, that can solve every productivity problem. Then *SB* is a *BB* machine and $p_{\mu_{SB}}(N_{SB}) > N_{SB}$. This is a contradiction. So a worker machine that could build all *BB* machines cannot exist. One *might* have a *WK* machine to build a finite sequence of the *BB* machines, but it could never build them all. Any such machine would need an infinite number of states, and hence, does not exist. For suppose we define $SB = \bigcup_{k<\omega} BB_k$. Then, $SB(n) = p(n)$ for all n, and $SB(SB(n)) = SB(n)$. In the Boolos argument the very starting point from which a contradiction is derived is the following: -

Suppose *BB* exists. Then there exists a machine that comprises a machine that writes *n* 1s followed by two copies of the BB machine.

Write n 1s $\longrightarrow$ BB $\longrightarrow$ BB
This machine has $n + 2k$ states and its productivity is $p(p(n))$.

For *SB* such a construction is impossible. We cannot add *SB* to itself, because *SB* has infinite states. To gain further insight into procedures that are non-effective I shall look at the two arguments of Cantor that lay the foundation of infinite cardinal arithmetic.

Definition, equinumerous sets
Let *X, Y* be sets. Then $X \sim Y$ iff there is a one-one mapping from *X* onto *Y*. *X* and *Y* are said to be *equinumerous* or *numerically equivalent*.

Result
$X \sim Y$ is an equivalence relation.

Cardinal number
To each set *X* we assign a cardinal number, denoted $|X|$.

$|X| = |Y|$ **iff** $X \sim Y$

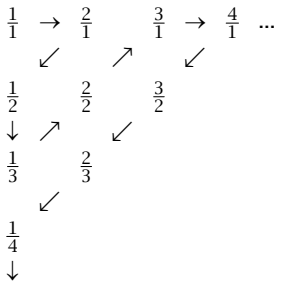
$|X| = 0$ **iff** $X = \varnothing$

Theorem (Cantor)

The set of rational numbers $\mathbb{Q}$ is equinumeous to the set of natural numbers $\mathbb{N}$. That is, $\mathbb{Q} \sim \mathbb{N}$.

Proof

Write the rational numbers as follows



By taking the sequence along the zigzag diagonals as shown every rational number may be placed in correspondence with the set $\mathbb{N}$. This shows $\text{card}\,\mathbb{Q} \le \text{card}\,\mathbb{N}$, but since $\mathbb{N} \subset \mathbb{Q}$ then $\text{card}\,\mathbb{N} \le \text{card}\,\mathbb{Q}$, whence $\text{card}\,\mathbb{Q} = \text{card}\,\mathbb{N}$ and $\mathbb{Q} \sim \mathbb{N}$.

Definition, diagonalisation

The argument used in the above result to establish the equinumerosity of two countably infinite sets is called *diagonalisation.*

Universal machines

Let $M_1, M_2, \ldots, M_n, \ldots$ be a denumerable list of Turing machines. Two machines are said to behave in the same way towards a number *x*, if their outputs for *x* are the same. Two machines are said to be *similar* if they behave in the same way for every number *x*. Let *U* be called a *universal machine*, defined by $U(x,y) = M_x(y)$; i.e. it behaves towards (x,y) as M_x behaves towards y. The universal machine computes the results of all the machines put together.

The universal machine is constructed by diagonalisation from an enumeration of all machines, as in the following table: -

		argument						
index	machine	1	2	...	*y*	....	*h*	...
1	M_1	$M_1(1)$	$M_1(2)$	...	$M_1(y)$	...	$M_1(h)$	...
2	M_2	$M_2(1)$	$M_2(2)$	...	$M_2(y)$	...	$M_2(h)$	...
⋮								
x	M_x	$M_x(1)$	$M_x(2)$	...	$M_x(y)$	...	$M_x(h)$	...
⋮								
h	M_h	$M_h(1)$	$M_h(2)$	...	$M_h(y)$	...	$M_h(h)$	...

Diagonalisation gives rise to a list of all machines and their values: $M_1(1), M_2(1), M_1(2), \ldots$ These just are the values of *U*.

Iteration theorem

There exists a machine *M* such that $M\left[\lceil M_k \rceil, \bar{x}\right] = M_k(\bar{x})$ where $\lceil M_k \rceil$ is the Gödel number of machine M_k.

Theorem

There is no enumeration of all total functions.

Proof

Let $(f_n) = f_1, f_2, \ldots, f_n, \ldots$ be an enumeration of all total functions. Consider the following array of these functions with their values: -

	1		2			m	
f_1	$f_1(1)$		$f_1(2)$	...	...	$f_1(m)$	...
f_2	$f_2(1)$		$f_2(2)$			$f_2(m)$	...
	...			...		...	...
f_n	$f_n(1)$	...	$f_n(2)$	...	...	$f_n(m)$	...

Along the leading diagonal of this array we construct the following function: -

$g(n) = f_n(n) + 1$

Since (f_n) is an enumeration of all total functions, $g = f_k$ for some *k*. Then $g(k) = f_k(k) = f_k(k) + 1$, which is a contradiction. Hence, there can be no enumeration of all total functions.

Definition, anti-diagonalisation

The argument used in the above theorem to establish that two infinite sets are not equinumerous is called *anti-diagonalisation.*

Anti-diagonalisation was used by Cantor to establish that the set of all reals has a cardinality greater than that of the set of all natural numbers. [Chap. 2, Sec. 2.7.6 / 2.2.11] Likewise, here, we see that the set of all total functions has greater cardinality than the set of natural numbers. But let us consider how this is established. Suppose $(f_n) = f_1, f_2, \ldots, f_n$ is now a finite list; then, certainly, we know how to find $g_1(k) = f_k(k) + 1$: $g(n)$ cannot be on the existing list (f_n) but certainly we can add it to the end of the list as f_{n+1}. Then we can create a new function not on the list $g_2(k) = f_k(k) + 1$, and so on *ad infinitum*. Let $G = \lim_{n\to\infty} g_n$. Then *G*, by anti-diagonalisation, cannot be on any list. Nonetheless, the problem is not with the "constructibility" of *G*, which is as well-defined and "constructible" as, for example, any transfinite function, but with the fact that the construction of *G* is always *one-step* ahead of any finite process; I shall say that the *rate at which G is constructed* is greater than the rate at which any finite list is generated. The function $g(n) = f_n(n) + 1$ is *almost effectively computable*. It is "effective" in the wider and *non-computing* sense of the term. I would go so far as to say "highly effective". Certainly, we cannot write down any exact polynomial that corresponds to $G(x) = e^x$ but that does not prevent us from approximating it to any required degree of accuracy using its Taylor series: -

$$g_0(x) = 1 \qquad g_1(x) = 1 + x \qquad g_2(x) = 1 + x + \frac{x^2}{2} \qquad \ldots \qquad g_n(x) = \sum_{k=0}^{n} \frac{1}{k!} x^k \qquad \ldots$$

Let us now compare this with the situation in the above anti-diagonalisation argument, and take, for a concrete example, the function *F* defined by: -

$$f_1(k) = 1 \text{ for all } k$$
$$f_n(k) = \begin{cases} f_{n-1}(k) & \text{if } k \neq n \\ f_{n-1}(n-1) & \text{if } k = n \end{cases}$$
$$F = \lim_{n\to\infty} f_n$$

These inductive rules generate a sequence of approximations to *F* given in this table: -

	n									
	1	2	3	4	5	6	7	8	9	…
f_1	1	1	1	1	1	1	1	1	1	…
f_2	1	2	1	1	1	1	1	1	1	…
f_3	1	2	3	1	1	1	1	1	1	…
f_4	1	2	3	4	1	1	1	1	1	…
f_5	1	2	3	4	5	1	1	1	1	…
…		…		…		…		…		…

We see visually that each approximation is a partial enumeration of the set of natural numbers $\mathbb{N}$ up to some finite value, and an infinite list of 1s thereafter. We are "pushing" the natural numbers into an already existing actually infinite list of 1s. What makes each line of this table into an effectively enumerable function are the dots $(\ldots)$. We are generating the successive approximations to *F* line by line and *potentially* as a list that could be continued indefinitely. The reason why *F* itself is *not enumerable* is because it represents the *actually complete infinite sequence of all natural numbers*, $\mathbb{N}$. We know, of course, that $\mathbb{N}$ is recursively enumerable, in fact, *it is the standard reference set for all enumerable sets*; we define what is enumerable *relative to* $\mathbb{N}$. But when we say that $\mathbb{N}$ is enumerable, we mean *potentially enumerable* as a sequence of approximations with dots $(\ldots)$. Clearly, no computer could actually enumerate all of $\mathbb{N}$ any more than we could. We confront similar issues in the case of the halting problem, only this focuses us even more on the distinction between what we can do and what we know we can do. There is no actually infinite computer program. Every Turing machine whatsoever is finite. That is the reason why the halting problem for every Turing machine is soluble, and very concretely so by the practical method outlined in this paper. Given a finite Turing machine, *T*, we could even, in principle, build another Turing machine, T', of many more states than *T* that could solve the halting problem for *T* but *not for itself*. This is concretely what it means to say that the halting problem is not Turing computable. Thus, we *know* that we can solve the halting problem for all Turing machines, and effectively so, but no Turing machine itself could solve the problem for all Turing machines including itself, for it would have to have an actually infinite number of states, and that is not possible. Always it is assumed that proof by induction is just another effectively computable process, and perhaps we even have some programs that *simulate mathematical induction* in a limited number of cases. Nonetheless, what we learn here is that *proof by induction is a synthetic principle of reasoning*, precisely because it enables us to conclude that *the halting problem for all Turing machines is soluble* whereas no single Turing machine, being in fact a finite entity, could actually employ an effective procedure for this.

When one contemplates the question: is the halting problem for *any* Turing machine soluble, one immediately sees that the answer to this must be "yes!". Why? Because every Turing machine is finite, and whatever happens inside a finite machine is subject to finite explanation. It is no mystery whether or not Turing machines enter into loops of infinite repeating cycles – certainly it has to do not only with the machine but what is on the tape at the beginning of the computation. Therefore, if one has the will and patience, then the problem can be solved. Naturally, the methods outlined here will

be practically very difficult for any Turing machine of some complexity involving many loops; computers certainly could assist, but, even with such assistance from the most powerful computers in the world, the practical solution to a problem involving a machine of many states could require more time than all the ages of the universe will allow. But that is not the ground on which it is maintained that the halting problem is not soluble. It is maintained on theoretic grounds – an impossibility proof that encompasses every finite machine of a potentially infinite collection of such machines. Nonetheless, the halting problem is soluble.

Melampus