Turing Machines and Computability

Charles E. Baker

July 6, 2010

1 Introduction

Turing machines were invented by Alan Turing in 1936, as an attempt to axiomatize what it meant for a function to be "computable."[1] It was a machine designed to be able to calculate any calculable function. Yet this paper also revealed startling information about the incomputability of some very basic functions.[2] In this paper, we attempt to demonstrate the power and the limits of computability through this construct, the (deterministic) Turing machine. To set the scope of our paper, some definitions are in order.

First, a Turing machine runs simple algorithms. Following S. B. Cooper, we say that an *algorithm* is "a finite set of rules, expressed in everyday language, for performing some general task. What is special about an algorithm is that its rules can be applied in potentially unlimited instances."[3] Of course, using an algorithm to calculate on arbitrary input will eventually be too costly in time and space to carry out.[4]. Yet the point is that the Turing machines should be able to run algorithms in arbitrary time, at least those algorithms that are amenable to computation.

Also, we insist that our algorithms can specifically be used to compute functions. This is particularly troublesome, however, as natural algorithms may not halt in finite time, which means that they cannot compute function values directly.[5] Also, even if an algorithm stops, it may not stop in a nice way: most

[1] [3], p. 1, 5, 42
[2] [3], p. 1
[3] [3], p. 3
[4] [1], p. 23-4
[5] The classic example is the "hailstone" map

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

Repetitions of this map may be computed by a machine, but affirming or refuting the conjecture that its iterates are eventually attracted to the 3-cycle (1, 4, 2) is difficult, or enter some other cycle, is difficult ([5]). Therefore, if we define the "first-one function" $h(n)$ to be the first k such that the kth iteration of the map f, starting with n, gives 1, this function may not be defined for certain values.

modern programming languages intentionally possess "break" commands to stop computations, so that algorithms do not complete and the output, if it exists, may not be comprehensible.

To deal with both of these difficulties, the natural solution is not to force an algorithm to define a function, but a partial function: a *partial function* from X to Y, where X and Y are sets, is a function from some *subset* of X to Y. Note that by this definition, even the function from the empty domain to Y is a partial function.

Finally, we must define what a "good" or "computable" function is; therefore, we follow [1]'s description of the notion of *effectively computable*. "A function f from positive integers to positive integers is called *effectively computable* if a list of instructions can be given that in principle make it possible to determine the value $f(n)$ for any argument n . . . [t]he instructions must be completely definite and explicit . . . the instructions should require no external sources of information."[6]. In particular, by this definition we limit ourselves to *deterministic* algorithms: there can be no "oracle" that tells you what to do at a step, unless you construct it yourself, nor can there be any random choice.[7]

Therefore, in this paper, we not only define Turing machines, but show the limitations in their computation of functions, and suggest how the class of functions they compute compares to the "effectively computable" notion given above.

2 Deterministic Turing Machine Setup

A deterministic Turing machine is simply a preprogrammed read-write machine with a finite number of instructions. The Turing machine can be thought of as sitting on a bi-infinite tape extending left-to-right, divided into squares. Each square contains either the blank symbol B or the filled symbol 1.[8] The Turing machine starts in one of a finite number of states: for convenience, we give a machine N *active states*, denoted $q_1, q_2, \ldots, q_N$ ($N \in \mathbb{Z}^+ \cup \{0\}$), and a *halting state* q_{N+1}. By convention we always start the machine in state q_1. When the machine reaches state q_{N+1}, it simply stops. This situation is viewed in Diagram 1.

The Turing machine (call it T for Turing) works step-by-step. At each step, the machine does the following. It recalls what state q_i it is in, and reads the symbol of the square it is sitting on, either B or 1. Given that input, it responds with *one* of the following actions:

1. The machine, T, writes a B over whatever is written on the tape. This action is denoted B.

[6] [1], p. 23

[7] While nondeterministic Turing machines have been studied (see for instance [3], Chapters 10-11), we will not be studying them here.

[8] This notation is partially inspired by [2], and will generalize well to a greater number of symbols.

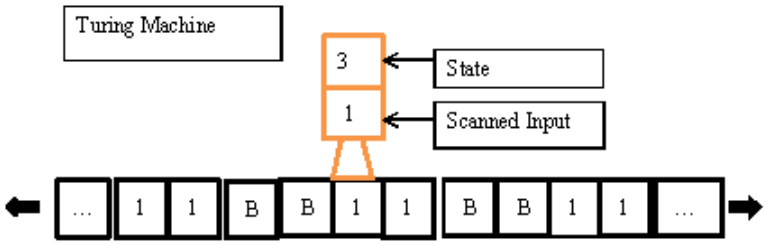

Figure 1: Turing Machine.

2. T writes a 1 over whatever is written on the tape. Denoted 1.

3. T moves one square left on the tape. Denoted L.

4. T moves one square right on the tape. Denoted R.

Finally, it *also* moves to another state q_j (which can be the same state q_i, as happens in recursive procedures). Then this step ends and the next step begins.

Therefore, an instruction to the machine can be encoded by the quadruple

$$q_i, S, A, q_j,$$

which means "start in state q_i, and read in the input. To respond to input symbol S, complete action A and enter the new state q_j."

We create a Turing machine by giving it a set of instructions. These instructions should be *complete* in the sense that all active states must have their instructions: for each active state q_i, we must have a quadruple to deal with q_iB and q_i1. Further, the instructions should be *consistent* insofar as the machine should never recieve contradictory assumptions or have to decide what to do. For example, we cannot give a single Turing machine the instructions q_1, B, R, q_2 and q_1, B, L, q_3, for this would either require the machine to move left and right simultaneously, or to choose one of the instructions over the other.

Definition 2.1 ([3], Definition 2.4.1 (1)) *A set X of quadruples is* ***consistent*** *if*

$$q_iSAq_j, q_iSA'q_k \in X \Longrightarrow A = A' \quad \text{and} \quad q_j = q_k.$$

This definition is precisely the way to avoid a machine that goes left and right simultaneously. This is all that is really required for a Turing machine. In fact, [3] defines a Turing machine to be "a finite consistent set of quadruples."[9] Our

[9] [3], p. 36.

definition is slightly more restrictive, because we add the halting state to control halting. Yet given a consistent set of quadruples, one can create a Turing machine in our form by "filling in the blanks" for any state with a partial set of instructions: we take any combination of state and input that has no instruction and tell it to respect what is written [B to B, 1 to 1] and to go to the halting state.[10]

Therefore, we have a set of quadruples that defines the Turing machine. Further, this allows us to write the Turing machine as a sequence of instructions: we write the quadruples in lexicographic order on the state number and the order number: i.e., we have

$$q_1BA_{1,B}q_{j_{1,B}}, q_11A_{1,1}q_{j_{1,1}}, q_2BA_{2,B}, q_{j_{2,B}}, q_21A_{2,1}q_{j_{2,1}}, \ldots, q_NBA_{N,B}q_{j_{N,B}}, q_N1A_{N,1}q_{j_{N,1}},$$

where $A_{n,S}$ is one of the actions $B, 1, L, R$ and $q_{j_{n,S}}$ is just some other state. Since we have agreed on this ordering, we can actually simplify by removing the somewhat redundant initial 2 terms of each quadruple, and get the sequence

$$A_{1,B}, q_{j_{1,B}}, A_{1,1}, q_{j_1,1}, \ldots, A_{N,B}, q_{j_{N,B}}, A_{N,1}, q_{j_{N,1}},$$

Thus, we can encode any N-state Turing machine as a $4N$-length sequence of terms in the finite alphabet $\{B, 1, L, R, q_1, \ldots, q_N\}$. Since the set of finite sequences in a finite alphabet is finite, and legitimate tuples are only a subset of these, the set of N-length Turing machines is finite.[11] Therefore, we have the first interesting fact about Turing machines.

Theorem 2.2 *The set of Turing machines is enumerable.*

Proof. The countable union of countable sets is countable, and the set of Turing machines is the union of the sets of N-state Turing machines. There are countably many such sets (indexed by $\mathbb{N}$, in fact) and each one is finite.

■

3 Examples

We now give some basic examples of Turing machines. Before beginning, we note that to this point, we have not assumed anything about the initial state of the tape. It could theoretically start with all entries B, or all entries 1, or whatever combination we wish. It is clear that a Turing machine's behavior depends on the initial state of the tape, and where the machine is on the tape; therefore, our

[10] This convention follows [1], p. 35-6

[11] Following [1], p. 36, we note that the odd-numbered positions only take the action symbols and the even-numbered positions only take the state symbols. [1], p. 36, also gives an explicit enumeration.

examples make assumptions on the tape as well as on the machine. In particular, we will generally assume that all but finitely many squares are blank (B) at any given time.[12]

To display these assumptions and the state of the machine, it is useful to make a diagrammatic convention. We write the tape's state by writing the sequence of 1's and B's on the tape, where it is understood that unspecified squares are blank (B). We write that the machine is in state q_i and at a certain position if we subscript the symbol in that position by the state number.[13] For example,

$$1\ 1\ B_3\ 1\ 1$$

denotes that the Turing machine is in state 3, scanning a blank separating 2 two-length blocks of 1's, on an otherwise blank tape.

Example 1. The simplest Turing machine is simply the one with no active states, which immediately halts on the state q_1, which is both its initial and its halting state. We denote this $T_\emptyset$.

Example 2. Turing machines do not necessarily halt: the simplest way to cause this possibility is to never call the halting state. One such machine is the following machine with one active state ($N = 1$), so the halting state is q_2:[14]

$$q_1\ B\ 1\ q_1, q_1\ 1\ 1\ q_1.$$

This machine simply replaces a blank by a 1, preserves a 1 if it reads a 1, and then stays in state q_1. It never goes to the halting state q_2, so it just stays in one place and runs forever.

For the next examples, we will be representing the integer n by a string of n 1's, flanked by B symbols to terminate it. This is not precisely our final convention, but it is easier to see some calculations this way.

Example 3: Parity[15] There is a Turing machine which, starting on the leftmost of a block of n ones on an otherwise blank tape, halts on the leftmost of a block of 1 or 2 1's on an otherwise blank tape, depending as to whether n is odd or even. A flowchart diagram gives more clarity than a list of quadruples here. Here we write arrows to represent going from state q_i to q_j, replace q_i by simply i, and write $S : A$ to denote the symbol-action relation. See Figure 2.

The machine starts by simply repetitively erasing a 1 and moving right to the next square. This could be done in two states if all we wanted was to do was erase the 1's, but by including four states, we encode the parity of the function as to which pair of states the function is in. Therefore, when the machine finally

[12] This assumption will make sense in light of our conventions in the next section.
[13] [1], p. 27
[14] Example based on [1], p. 36-7
[15] Example from [1], p. 29.

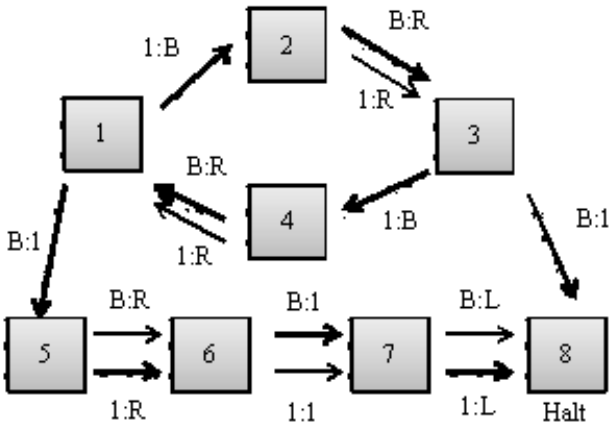

Figure 2: Parity-checker Turing machine.

has removed everything and it reads a B, it knows what the parity is and writes in the appropriate number of ones, then goes to the leftmost 1 and terminates. For example, applied to $n = 4$, the diagrams of the successive states are as follows:

$1_1\ 1\ 1\ 1\ B\ B$, $B_2\ 1\ 1\ 1\ B\ B$, $B\ 1_3\ 1\ 1\ B\ B$, $B\ B_4\ 1\ 1\ B\ B$,
$B\ B\ 1_1\ 1\ B\ B$, $B\ B\ B_2\ 1\ B\ B$, $B\ B\ B\ 1_3\ B\ B$, $B\ B\ B\ B_4\ B\ B$,
$B\ B\ B\ B\ B_1\ B$, (breakout),
$B\ B\ B\ B\ 1_5\ B$, $B\ B\ B\ B\ 1\ B_6$, $B\ B\ B\ B\ 1\ 1_7$, $B\ B\ B\ B\ 1_8\ 1$,
halts

Note that the 1 instructions for states 2 and 4 are never carried out, since these states are never called except if the square has been overwritten to a B first. Our definition is strict enough that it must include unnecessary instructions. Also, it is indeed true that our new string of 1's starts farther to the right then where we originally started, but we do not particularly care: our tape is bi-infinite, so there is no natural "starting point."

Example 4: Addition[16] There is a Turing machine which, if it starts on the leftmost square of a block of n 1's, then a blank, then a blank of m 1's, produces a block of $n + m$ ones on an otherwise blank slate, leaving the machine on the leftmost 1.

The procedure is simple: the machine simply erases the first 1, then moves to the blank, fills it in, then the n and m blocks have been combined. The Turing machine moves back to the left and halts. It must overshoot when coming back, however, since the machine does not know where the end of the block of 1's is

[16]Example from [1], p. 29.

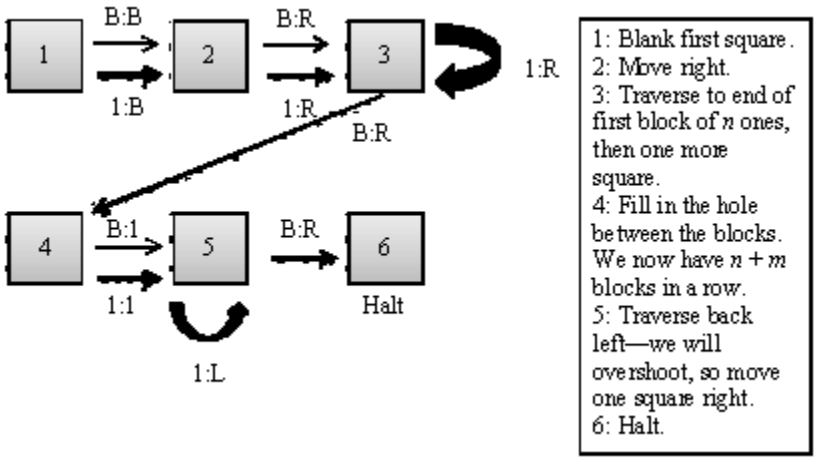

Figure 3: Addition Turing machine.

until it overshoots it. A flowchart is given in Figure 3.

Example 5: Parity of Sum: A Turing machine exists such that, given two blocks of n and m ones on an otherwise blank tape, with the Turing machine on the leftmost 1, the Turing Machine ends up on the leftmost slot of a block of 2 or 1 1's, depending on whether the sum $n + m$ is odd or even.

This is an illustration of the usefulness of the halting-state convention: we can easily concatenate two Turing machines T with N active states and U with M active states as follows: we create the machine TU by starting with the procedure for T, but replacing T's halting state by the initial state of U so that if T halts, we immediately start U from wherever the Turing machine is now.[17] Therefore, to create the above machine, we simply start with the addition machine, and since we chose to move the addition machine back to the beginning of the block, it is in the proper starting position to give the correct answer when we execute the parity machine.

Example 6: Multiplication: There is a Turing machine which, given n 1's, then a blank, then m 1's, on an otherwise blank tape, and starting at the leftmost 1, produces a string of $n \times m$ 1's, halting on the leftmost 1. While a full writeout of the multiplication machine is beyond the scope of this text,[18] we would like to suggest how the machine works in terms of the subroutines it must run.

First of all, note that there is no way for a finite Turing machine to store even in its state code the value of n or m, given that these can be arbitrarily large. Therefore, we rely on the tape to store this information for us.[19] Fortunately, since

[17] Concatenation is left-to-right here.

[18] See [1], p. 30-1, for a full machine flowchart.

[19] Note that this illustrates a general fact of the Turing machine: that it has no external

multiplication is repeated addition, we just need to use n to count the number of m's we add together. The first, n-length block will act as a counter.

To start the machine, we erase a 1 in the first slot, and move one to the right. If we read a B, we only had $n = 1$, so we just had $1 \times m = m$, and we simply move until we meet the m block, then terminate. Otherwise, we move to the beginning of the second block, which can be done in one state.[20] Then we begin a "leapfrog" procedure to move the stack of m 1's over by its own length, thereby implicitly *adding* n. We then traverse back to the leftmost of the remaining $n - 1$ ones in the first block. We repeat: therefore, each time we erase a 1 in the first stack of ones, we move the second stack of m ones over by m, thus really *adding* m *each time*. When we finally erase the last 1, (which we recognize, for moving one to the right we find a blank), we move to where the original block of m started; then we fill in all the 1's as long as we read B, thus making visible the repeated addition we did. Then, when we read in a 1, we have finally reached our existing m ones and filled everything in, so the machine walks to the left of the block of 1's and terminates.

The point is that this procedure shows one way of how recursion works in a Turing machine, recursion being an integral part of computability theory. Now, just as repeated addition is multiplication, repeated multiplication is exponentiation, and should be the next in this sequence[21], but directly writing this machine is horrible. In practice, we simply use theoretical results from equivalent notions of computability to solve these problems. To do so, however, we must make some conventions on how these machines are used to compute partial functions.

4 Using Turing Machines to Calculate Partial Functions

To calculate functions, we must make some sort of method to encode input and output on the tape, and where the machine starts and ends. We mostly follow [1] in our choice of conventions.

First, take T a Turing machine, and take k a positive integer. We set $\mathbb{N}$ to be the natural numbers, *including* 0, and we will use T to calculate a partial function from $\mathbb{N}^k$ (k-tuples in $\mathbb{N}$) to $\mathbb{N}$. First of all, to include zero, we encode the integer n in general as a sequence of $n + 1$ 1's, flanked on either side by B to terminate it.[22]

memory , but rather uses the tape itself and its internal state for memory ([2], p. 5).

[20] The first state moves right if it reads a one or a zero, but calls the same state again if it reads a one, to repeat. If it moves to the next state, we just read a zero, but we moved right anyways, so the Turing machine should now be sitting on the first 1 of the second block.

[21] [1], p. 33, 51; [3], p. 17

[22] This is different from the implicit convention above, but it is possible to "endcap" our procedures to ensure that we can switch between the two forms.

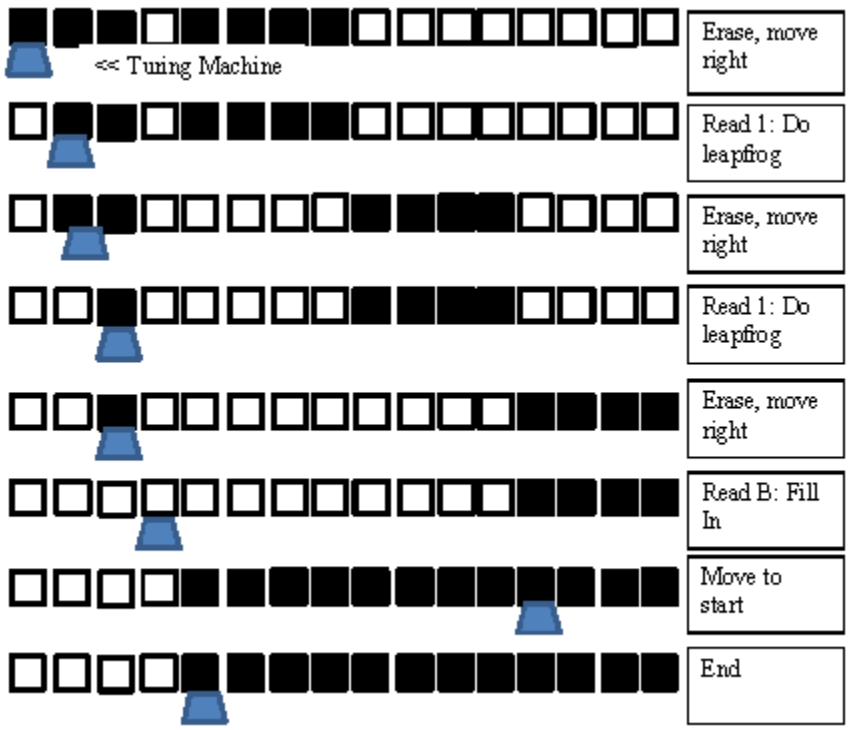

Figure 4: Illustration of 3×4.

4.1 Standard Initial Configuration

1. To encode the input vector $(n_1, \ldots, n_k)$, we take the initial state of the input tape to be
$$\underbrace{1\,1\ldots1}_{n_1+1 \text{ ones}}\ B\ \underbrace{11\ldots1}_{n_2+1 \text{ ones}}\ B\ \ldots\ \underbrace{11\ldots1}_{n_k+1 \text{ ones}},$$
and all unspecified squares have symbol B.[23]

2. The machine itself starts at the leftmost 1 in the n_1 input vector.

Since the starting state is always q_1, the standard initial configuration for input $(n_1, \ldots, n_k)$ is written in our diagram notation as

$$\ldots B\ B\ 1_1\ 1\ 1\ldots1\ B\ 1\ 1\ \ldots1\ B\ldots\ B\ 1\ 1\ldots1\ B\ B\ \ldots$$

.

This is fairly standard. Perhaps more surprising is the final state convention.

4.2 Standard Final Configuration

1. To output the integer n, the tape must be a string of $n + 1$ ones on an otherwise blank tape. The starting position of this string, however, does not have to be the same starting position as our original input string.

2. The machine T must be in state $N + 1$ (its halting state).

3. T must be sitting on the leftmost square of the segment of 1's.

In out diagram representation, the standard final configuration is

$$\ldots B\ B\ \underbrace{1_{N+1}\ 1\ldots1}_{n+1 \text{ ones}}\ B\ B\ \ldots.$$

If the machine halts and *any* of the above conditions are not satisfied, the machine is not in standard final configuration. For example, the following examples[24] are not in standard final configuration:

$B_{N+1}\ 1\ 1\ 1\ 1\ 1$	(The machine is not in the right position.)
$B\ 1\ 1_{N+1}\ 1\ 1$	(The machine is not in the right position.)
$B\ 1_{N+1}\ 1\ 1\ B\ B\ 1\ 1$	(The tape output is not of the correct form.)

[23] Since the tape is bi-infinite, we are deliberately vague about assigning a formal starting point.

[24] from [1], p. 32

4.3 The function output

To compute a (partial) function of k integer inputs, therefore, we start the Turing machine in standard initial configuration with some initial input vector $(n_1, \ldots, n_k)$, then let it run until it halts, if it halts, and we see whether or not it is in standard final configuration. In symbols, for k a positive integer and T a Turing machine, we define the partial function $f_{T,k}$ of k inputs calculated by T as

$$f_{T,k}(n_1, \ldots, n_k) = \begin{cases} \text{undefined} & \text{if } T \text{ never halts for the given input, or if } T \text{ halts in a nonstandard configuration.} \\ n & \text{if } T \text{ halts in standard final configuration for some nonnegative integer } n. \end{cases}$$

The fact that Turing functions naturally compute partial functions as opposed to total functions is quite apparent here: this is the more natural method to give ourselves a way to handle a nonhalting or irregularly halting Turing machine.

Therefore, we may now define Turing computability.

Definition 4.1 ([1], p. 33) *A partial function $g : \mathbb{N}^k$ to $\mathbb{N}$ is **Turing computable** if there is a Turing machine T such that $g = f_{T,k}$, $f_{T,k}$ defined as above.*

The remainder of this exposition dedicates itself to understanding the class of Turing-computable functions.

5 Incomputability

It is apparent from our previous results and some standard procedures that not all functions are Turing computable. Restricting ourselves to one-argument input for the moment, we recognize by our earlier observation that the set of Turing machines is countable the corollary that the set of one-argument functions they compute is countable. Yet the set of functions $f : \mathbb{N} \to \mathbb{N}$ is uncountable: indeed, even deciding on the function's range is choosing an element of the power set of $\mathbb{N}$, $\mathcal{P}(\mathbb{N})$. More specifically, we can get a diagonalization argument:[25] fix an enumeration $T_1, T_2, \ldots$ of the Turing machines. Then let

$$d(n) = \begin{cases} 2 & \text{if } f_{T_n,1}(n) \text{ exists and equals 1} \\ 1 & \text{otherwise} \end{cases}$$

If d were Turing computable, then it would be computable by the Turing machine T_m for some m: i.e. $d = f_{T_m,1}$. Therefore, we have

$$\begin{aligned} d(m) = f_{T_m,1}(m) &= \\ &= \begin{cases} 2 & \text{if } f_{T_m,1}(m) \text{ exists and equals 1} \\ 1 & \text{otherwise} \end{cases} \\ &= \begin{cases} 2 & \text{if } d(m) \text{ exists and equals 1} \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

[25] [1], p. 37

This is a contradiction, so d is not Turing-computable.

On the one hand, the addition of self-reference makes it possible for this contradiction to occur, so the existence of a contradiction is plausible. On the other hand, it is possible to argue that this function "should" be computable. Consider the following method to compute $d(n)$:[26]

1. Begin by taking an integer n. Take it for granted that we can computably determine which Turing machine T_n this integer encodes, if we use the explicit map of [1], p. 36. Then we can consider the one-argument function $f_{T_n,1}$ that T_n generates.
2. We know that the input is n, so by our encoding conventions, we can start attempting to define $f_{T_n,1}(n)$.
3. Our machine either terminates or it doesn't.
   (a) If the machine never terminates for input n, we get $d(n) = 1$.
   (b) If the machine terminates, we see if it is in standard final configuration for the output 1. If so, we output $d(n) = 2$; otherwise (if the function does not terminate in standard configuration, or if it outputs a different integer), we output $d(n) = 1$.

What's wrong with this argument? Mainly, the fact that we used the fact that the machine either terminates or it doesn't: this is an obvious fact, *but not necessarily a computable one.* Indeed, a prerequisite for this function to be computable is the ability to compute whether or not the machine halts, and this is not obviously computable.[27] We can directly show that this process is not Turing computable.

To be precise, define $h(m, n)$ to be the *halting function* that determines whether or not the one-argument function derived from machine T_m halts with input n:

$$h(m,n) = \begin{cases} 1 & \text{if } f_{T_m,1} \text{ halts for input } n \\ 2 & \text{otherwise} \end{cases}$$

This halting-problem, however, can be encoded in a way that forces self-reference.

Theorem 5.1 ([1], Theorem 4.2) *The halting function h is not Turing computable.*

Proof. Following [1], we use two Turing machines. The first is a copying machine C that sends (n) to (n, n), in our standard encoding, returning the machine to its appropriate position:

$$B\ 1_1\ 1\ 1\ B \longrightarrow B\ 1_{N+1}\ 1\ 1\ B\ 1\ 1\ 1\ B.$$

[26] Example from [1], p. 37-8

[27] We can certainly determine whether a machine halts after k steps, but it may be trickier to determine this for all k.

The second is a "dithering machine" D that halts for input $n \neq 1$, but moves back and forth indefinitely for input $n = 1$. We leave the construction of these machines to the reader.

The point is to make heavy use of the concatenation properties of Turing machines, as facilitiated by our convention of the halting state. Suppose that there exists a Turing machine H such that $f_{H,2}(m, n) = h(m, n)$; i.e., H's action on two inputs computes the halting function. The we concatenate and run the programs $T = CHD$:[28] given input n in standard form, C will convert it to (n, n), so that H will compute $h(n, n)$ in standard final configuration. Then D will read H's output: if H outputs a 1, i.e., if the one-argument function corresponding to machine T_n terminates for input n, then we get D to dither indefinitely. Yet if D receives input 2, that is, if the nth Turing machine does not halt for input n, then D will stop.

Yet this is clearly the beginning of a self-referential contradiction: for if CHD is some finite-state deterministic Turing machine, it corresponds to some integer m. Then consider $T_m = CHD$ acting on input m. Either it terminates or it does not terminate.

1. If CHD terminates for input m, then $CH(m) = H(m, m) = 1$, because $T_m = CHD$ terminates with input m. Therefore, $CHD(m)$ will, after the C and H routines, come to routine D with input 1, so the dithering machine will never halt, so $CHD(m)$ does not terminate. So by assuming that it does terminate, it does not terminate.

2. If CHD does not terminate for input m, then $CH(m) = H(m, m) = 2$, so $D(2)$ terminates, so $CHD(m) = D(2)$ terminates. So by assuming that it does not terminate, then it does terminate.

Contradiction.

■

There are other, more practical incomputable problems. For example, a Turing incomputable problem is the "finite word problem," to determine for an *arbitrary* finitely presented group G with a given presentation, whether or not a given finite word reduces to the identity.[29]

Given these negative results, the reader may be wondering if Turing computability is the best that can be done. Surely there is some other method of computing partial functions that gives a better outcome? For deterministic routines, however, the answer, thus far, is "no."

[28] Concatenation is left-to-right here.

[29] [4].

6 Computability and Church's Thesis.

It is now appropriate to consider some history.[30]. Around the 1930's, several attempts to axiomatize the notion of "effectively computable" arose: in addition to Turing's papers of 1936, the following other methods were significant.

Recursive functions. These were used by Gödel in his logical work.[31]. We create a class of functions, the "primitive recursive" functions, whose domain for each f is some $\mathbb{N}^k$, k depending on f, and whose codomain is $\mathbb{N}$. These start with the one-argument zero-function, the successor function $n \to n+1$, and the ith-coordinate-vector projection functions on $\mathbb{N}^k$ for each k. We close under the operations of concatenation and recursion. Adding the closure under minimalization (a technical process related to finding the "first zero" of a function with respect to its last input) gives the full class of recursive functions.[32] Indeed, it is easy to see that addition and number-theoretic constructs such as division and remainder functions are recursive, as well as functions as wild as the Ackermann function.[33]

Unlimited Register Machines. This was another sort of computer model, developed in 1950 by Shepherdson and Sturgis.[34]. It involves an infinite line of "registers," each holding an nonnegative integer, where the machine operations are "increment the amount in a register" and "*if* a register is not empty, then decrement, *else* do a different action."[35]. This is more clearly related to the modern notion of computer memory. It is easy to show that this function can handle every element in the "stacking" sequence of complexity that is addition, then multiplication (repeated addition), then exponentiation (repeated multiplication), then. . . .[36].

The λ-calculus. This model was developed by Alonzo Church and Steven Kleene from 1934, and now enjoys some use in computer programming.[37]

These are all distinct notions of computability, each seemingly different, and possibly more powerful: certainly, the register machines, which effectively replace our tape with registers capable of holding any nonnegative integer, not just a finite set of symbols, seem more powerful. What is interesting is that these notions are all equivalent, insofar as the family of computable functions is the same for each.

Theorem 6.1 ([1], Theorems 5.6, 5.8, 8.2; [3], Theorem 11.2.18) *A function is Turing computable if and only if it is recursive, if and only if it is URM-computable, if and only if it is lambda-computable.*

[30] This section mostly due to [3], p. 1-8
[31] [3], p. 7
[32] [3], p. 18
[33] [3], p. 12-6.
[34] [3], p. 8; [1] refers to these machines as "abacus machines" (p. 46)
[35] [1], p. 47
[36] [1], p. 49-51
[37] [3], p. 5, 180

This is not only a powerful theoretical result, but the relationship with recursion allows us to show that some of our conventions about Turing machines did not significantly affect the universe of computable functions. For example, having a singly infinite or bi-infinite tape makes no difference, nor does it change anything to change the tape symbols, as long as there are only finitely many of them.[38]. Further, if we can prove a function is computable in one method, then it is computable in all of them; hence, we have several points of view from which to find computable functions. Since all of these methods generate the same class of computable functions, we have the following conjectures about computability:

The Church-Markov-Turing thesis. "A partial function $\phi : \mathbf{N}^n \to \mathbb{N}$ is computable (in any accepted informal sense) if and only if is computable by some binary Turing machine."[39].

Now, since this is defined in terms of informal notions rather than formal ones, we cannot prove this thesis,[40]. Yet may are willing to assume the thesis, for there are many points in its favor: all current formalizations give the same class of functions, no clear counterexample exists, and Turing's exposition "was so persuasive . . . that it made any non-equivalent model of computability impossible to envisage."[41] To the best of our understanding, the Turing machines represent the boundaries of computability, so that the incomputability results from the previous section have real weight. Therefore, they do indeed represent (for now) the power and the limitations of computability.

References

[1] G. Boolos, J. P. Burgess, and R. C. Jeffrey, *Computability and Logic*, 4th ed., Cambridge U. Press, New York, 2006.

[2] D. S. Bridges, *Computability: A Mathematical Sketchbook*, Graduate Texts in Mathematics, Springer-Verlag, New York, 1994.

[3] S. B. Cooper, *Computability Theory*, Chapman & Hall/CRC Mathematics, Boca Raton, 2003.

[4] J. D. Hampkins, et al, "What are the Most Attractive Turing Undecidable Problems in Mathematics?" *MathOverflow*. Accessed at mathOverflow.net, 24 Jun 2010 version. http://mathoverflow.net/questions/11540/what-are-the-most-attractive-turing-undecidable-problems-in-mathematics.

[38] [1], p. 94
[39] [2], p. 32
[40] [2], p. 32
[41] [2], p. 32; [3], p. 42

[5] E. W. Weisstein, "Collatz Problem," *Wolfram MathWorld*. Accessed at www.mathworld.wolfram.com, 2 July 2010 version.
http://mathworld.wolfram.com/CollatzProblem.html