Prologue: the definition of computation

Jim Hefferon

University of Vermont

hefferon.net

Turing machines

Introduction: What can be done?

Here is a prototypical mathematical job. We get a linear system.

$$\begin{aligned} 2x \qquad &+ z = 3 \\ x - y &- z = 1 \\ 3x - y \qquad &= 4 \end{aligned}$$

Finding a single solution that satisfies this system $(x, y, z) = (3/2, 1/2, 0)$ is okay but we prefer to find them all, as here.

$$\{\, (x, y, z) = (\tfrac{3}{2} - \tfrac{1}{2}z, \tfrac{1}{2} - \tfrac{3}{2}z, z) \;\big|\; z \in \mathbb{R} \,\}$$

And, we are most charmed when we have a procedure that inputs any linear system at all and returns its complete set of solutions.

Introduction: What can be done?

Here is a prototypical mathematical job. We get a linear system.

$$\begin{aligned} 2x \qquad &+ z = 3 \\ x - y &- z = 1 \\ 3x - y \qquad &= 4 \end{aligned}$$

Finding a single solution that satisfies this system $(x, y, z) = (3/2, 1/2, 0)$ is okay but we prefer to find them all, as here.

$$\{\,(x, y, z) = (\tfrac{3}{2} - \tfrac{1}{2}z, \tfrac{1}{2} - \tfrac{3}{2}z, z) \;\big|\; z \in \mathbb{R}\,\}$$

And, we are most charmed when we have a procedure that inputs any linear system at all and returns its complete set of solutions.

In asking what can be done mathematically we are asking what can be done systematically, or uniformily. This course studies what can and cannot be done with a computer.

Some history

In 1928, the famous mathematician D Hilbert and his student W Ackermann proposed the *Entscheidungsproblem*. It asks for a definite procedure that decides, from an input mathematical statement, whether that statement is true or false.

Wanting to know what is true or false requires no explanation. But what is a 'definite procedure'?

In mathematics we take Euclid's development as a paradigm. From the axioms we prove theorems in a sequence of steps. These steps are such that you can verify each one typographically, by pushing symbols around without any unexplainable leaps of intuition. Thus in our mathematical culture a definite procedure is a sequence of fully-specified steps that go from some input to a result. So Hilbert and Ackermann were asking for what we today call an algorithm, which you could cast into code and run on a computer.

Thus, a important question in the air was to define what can be mechanically computed. (Remember that in 1928 there were no computers as we know them today.)

Turing machine

The approach to the definition of mechanical computation that we will follow was given by A Turing. He described a mechanism that does the computing.

As a prototype he imagined a clerk doing by-hand multiplication with a sheet of paper that gradually becomes covered with columns of numbers. With that model he gave a number of conditions that the computer must satisfy.

Turing machine

The approach to the definition of mechanical computation that we will follow was given by A Turing. He described a mechanism that does the computing.

As a prototype he imagined a clerk doing by-hand multiplication with a sheet of paper that gradually becomes covered with columns of numbers. With that model he gave a number of conditions that the computer must satisfy.

- First, it (or he or she) has a memory facility, such as the clerk's paper, where it can put information for later retrieval.

Turing machine

The approach to the definition of mechanical computation that we will follow was given by A Turing. He described a mechanism that does the computing.

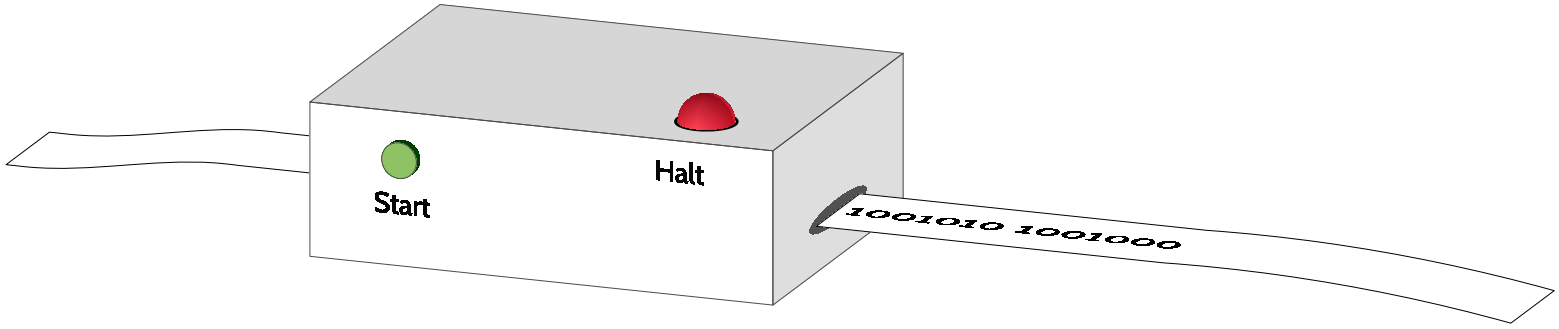
As a prototype he imagined a clerk doing by-hand multiplication with a sheet of paper that gradually becomes covered with columns of numbers. With that model he gave a number of conditions that the computer must satisfy.

- First, it (or he or she) has a memory facility, such as the clerk's paper, where it can put information for later retrieval.
- Second, the computing agent must follow a definite procedure, a precise set of instructions with no room for creative leaps. Part of what makes the procedure definite is that the instructions don't involve random methods, such as counting clicks from radioactive decay to determine which of two possibilities to perform. The other thing making the procedure definite is that the agent is discrete — it does not use continuous methods or analog devices. Thus there is no question about the precision of operations as there might be when reading results off of a slide rule or an instrument dial. In line with this, the agent works in a step-by-step fashion. If needed they could pause between steps, note where they are ("about to carry a 1"), and pick up again later. We say that at each moment the clerk is in one of a finite set of possible states, which we denote q_0, q_1, ...

- Turing's third condition arose because he wanted to investigate what is computable in principle. He therefore imposed no upper bound on the amount of available memory. More precisely, he imposed no finite upper bound — should a calculation threaten to run out of storage space then more is provided. This includes imposing no upper bound on the amount of memory available for inputs or for outputs and no bound on the amount of extra storage, scratch memory, needed in addition to that for inputs and outputs. He similarly put no upper bound on the number of instructions. And, he left unbounded the number of steps that a computation performs before it finishes.

- Turing's third condition arose because he wanted to investigate what is computable in principle. He therefore imposed no upper bound on the amount of available memory. More precisely, he imposed no finite upper bound — should a calculation threaten to run out of storage space then more is provided. This includes imposing no upper bound on the amount of memory available for inputs or for outputs and no bound on the amount of extra storage, scratch memory, needed in addition to that for inputs and outputs. He similarly put no upper bound on the number of instructions. And, he left unbounded the number of steps that a computation performs before it finishes.
- The final question Turing faced is: how smart is the computing agent? For instance, can it multiply? We don't need to include a special facility for multiplication because we can in principle multiply via repeated addition. We don't even need addition because we can repeat the successor operation, the add-one operation. In this way Turing pared the computing agent down until it is quite basic, quite easy to understand, until the operations are so elementary that we cannot easily imagine them further divided, while still keeping that agent powerful enough to do anything that can in principle be done.

Turing pictured a box containing a mechanism and fitted with a tape.



The tape is the memory, sometimes called the ‘store’. The box can read from it and write to it, one character at a time, as well as move a read/write head relative to the tape in either direction. Thus, to multiply, the computing agent can start by reading the two input multiplicands from the tape (the drawing shows 74 and 72 in binary, separated by a blank), can use the tape for scratch work, and can halt with the output written on the tape.

The box is the computing agent, the CPU, sometimes called the ‘control’. The `Start` button sets the computation going. When the computation is finished the `Halt` light comes on.

While executing a calculation, the mechanism steps from state to state. For instance, an agent doing multiplication may determine, because of what state it is in now and because of what it is reading on the tape, that they next need to carry a 1. The agent transitions to a new state, one whose intuitive meaning is that it is where carries take place.

Consequently, machine steps involve four pieces of information. Call the present state q_p and the next state q_n. The symbol that the read/write head is presently pointing to is T_p. Finally, the next tape action is T_n. Possible actions are: moving the tape head left or right without writing, which we denote with $T_n = \texttt{L}$ or $T_n = \texttt{R}$, or writing a symbol to the tape without moving the head, which we denote with that symbol, so that $T_n = \texttt{1}$ means the machine will write a 1 to the tape. As to the set of characters that can go on the tape, we will choose whatever is convenient for the job we are doing. However every tape has blanks in all but finitely many places and so that must be one of the symbols. (We denote blank with B when an empty space could cause confusion.)

The four-tuple $q_p T_p T_n q_n$ is an instruction. For example, the instruction $q_3 \texttt{1B} q_5$ is executed only if the machine is now in state q_3 and is reading a 1 on the tape. If so, the machine writes a blank to the tape, replacing the 1, and passes to state q_5.

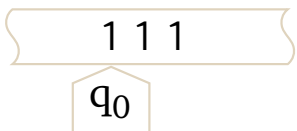
Before the formal definition, we first see a couple of examples.

First example computation: the predecessor function

This Turing machine with the tape symbol set $\Sigma = \{\mathsf{B}, \mathsf{1}\}$ has six instructions.

$$\mathcal{P}_{\text{pred}} = \{q_0\mathsf{B}\mathsf{L}q_1,\ q_0\mathsf{1}\mathsf{R}q_0,\ q_1\mathsf{B}\mathsf{L}q_2,\ q_1\mathsf{1}\mathsf{B}q_1,\ q_2\mathsf{B}\mathsf{R}q_3,\ q_2\mathsf{1}\mathsf{L}q_2\}$$

This shows a stretch of tape along with the machine's state and the position of its read-write head.



We adopt the convention that when we press `Start` the machine is in state q_0. The picture above shows the machine reading 1, so instruction $q_0\mathsf{1}\mathsf{R}q_0$ applies. Thus the first step is that the machine moves its tape head right and stays in state q_0.

Here is the machine again

$$\mathcal{P}_{\text{pred}} = \{q_0\mathsf{B}\mathsf{L}q_1,\ q_0\mathsf{1}\mathsf{R}q_0,\ q_1\mathsf{B}\mathsf{L}q_2,\ q_1\mathsf{1}\mathsf{B}q_1,\ q_2\mathsf{B}\mathsf{R}q_3,\ q_2\mathsf{1}\mathsf{L}q_2\}$$

and here are the later steps of the computation. Briefly, the head slides to the right, blanks out the final 1, and slides back to the start.

Step	*Configuration*
1	1 1 1 q_0
2	1 1 1 q_0
3	1 1 1 q_0
4	1 1 1 q_1
5	1 1 q_1

Step	*Configuration*
6	1 1 q_2
7	1 1 q_2
8	1 1 q_2
9	1 1 q_3

Now because there is no $q_3\mathsf{1}$ instruction, the machine halts.

We can think of this as computing the predecessor function

$$\mathrm{pred}(x) = \begin{cases} x - 1 & \text{– if } x > 0 \\ 0 & \text{– else} \end{cases}$$

because if the machine's initial tape is entirely blank except for n-many consecutive 1's and the head points to the first, then at the end the tape will have $n - 1$-many 1's (except for $n = 0$, where the tape will end with no 1's).

We can think of this as computing the predecessor function

$$\mathrm{pred}(x) = \begin{cases} x - 1 & - \text{ if } x > 0 \\ 0 & - \text{ else} \end{cases}$$

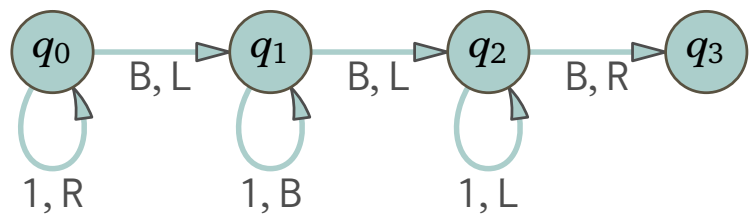
because if the machine's initial tape is entirely blank except for n-many consecutive 1's and the head points to the first, then at the end the tape will have $n - 1$-many 1's (except for $n = 0$, where the tape will end with no 1's).

In place of the set of four-tuples

$$\mathcal{P}_{\mathrm{pred}} = \{\, q_0\mathsf{B}\mathsf{L}q_1,\ q_0\mathsf{1}\mathsf{R}q_0,\ q_1\mathsf{B}\mathsf{L}q_2,\ q_1\mathsf{1}\mathsf{B}q_1,\ q_2\mathsf{B}\mathsf{R}q_3,\ q_2\mathsf{1}\mathsf{L}q_2 \,\}$$

we can also describe this machine with a table or picture it with a graph.

Δ	B	1
q_0	Lq_1	Rq_0
q_1	Lq_2	Bq_1
q_2	Rq_3	Lq_2
q_3	–	–



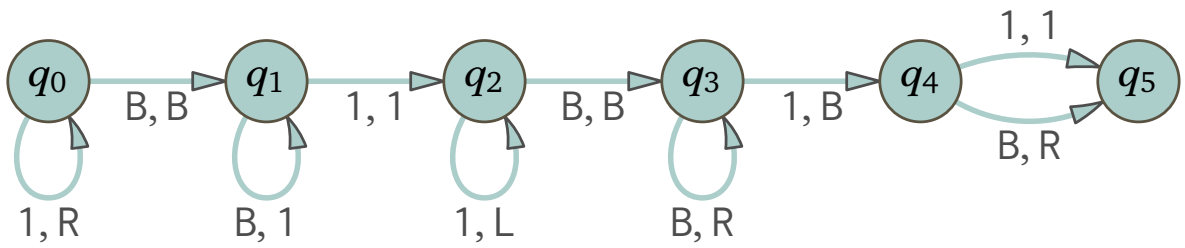
Second example computation: addition of two numbers

We can think of this machine with tape alphabet $\Sigma = \{\mathsf{B}, \mathsf{1}\}$ as adding two natural numbers.

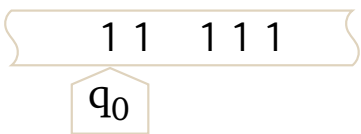
$$\mathcal{P}_{\text{add}} = \{q_0\mathsf{B}\mathsf{B}q_1,\ q_0\mathsf{1}\mathsf{R}q_0,\ q_1\mathsf{B}\mathsf{1}q_1,\ q_1\mathsf{1}\mathsf{1}q_2,\ q_2\mathsf{B}\mathsf{B}q_3,\ q_2\mathsf{1}\mathsf{L}q_2,$$
$$q_3\mathsf{B}\mathsf{R}q_3,\ q_3\mathsf{1}\mathsf{B}q_4,\ q_4\mathsf{B}\mathsf{R}q_5,\ q_4\mathsf{1}\mathsf{1}q_5\}$$

The input numbers are represented by two strings of 1's, separated with a blank. Here is the table and graph for $\mathcal{P}_{\text{add}}$.

Δ	B	1
q_0	Bq_1	Rq_0
q_1	1q_1	1q_2
q_2	Bq_3	Lq_2
q_3	Rq_3	Bq_4
q_4	Rq_5	1q_5
q_5	–	–



This shows the machine ready to compute 2 + 3.



Here are the other steps in the computation (the initialization above is step 0).

Step	*Configuration*
1	1 1 1 1 1 q_0
2	1 1 1 1 1 q_0
3	1 1 1 1 1 q_1
4	1 1 1 1 1 1 q_1
5	1 1 1 1 1 1 q_2
6	1 1 1 1 1 1 q_2

Step	*Configuration*
7	1 1 1 1 1 1 q_2
8	1 1 1 1 1 1 q_2
9	1 1 1 1 1 1 q_3
10	1 1 1 1 1 1 q_3
11	1 1 1 1 1 q_4
12	1 1 1 1 1 q_5

Some Turing machines don't halt

A crucial observation: some Turing machines, for at least some starting configurations, never halt. The machine $\mathcal{P}_{\text{inf loop}} = \{ q_0\mathsf{B}\mathsf{B}q_0,\ q_0\mathsf{1}\mathsf{1}q_0 \}$ never halts, regardless of the input.

Definition of a Turing machine

DEFINITION A Turing machine $\mathcal{P}$ is a finite set of four-tuple instructions $q_p T_p T_n q_n$. In an instruction, the present state q_p and next state q_n are elements of a set of states Q. The input symbol or current symbol T_p is an element of the tape alphabet set Σ, which contains at least two members including one called blank (and does not contain L or R). The action symbol T_n is an element of the action set $\Sigma \cup \{\mathsf{L}, \mathsf{R}\}$.

The set $\mathcal{P}$ must be deterministic: different four-tuples cannot begin with the same $q_p T_p$. Thus, over the set of instructions $q_p T_p T_n q_n \in \mathcal{P}$, the association of present pair $q_p T_p$ with next pair $T_n q_n$ defines a function, the transition function or next-state function $\Delta : Q \times \Sigma \to (\Sigma \cup \{\mathsf{L}, \mathsf{R}\}) \times Q$.

We denote a Turing machine with a $\mathcal{P}$ because although these machines are hardware, the things from everyday experience that they are most like are programs.

The action of a machine

We can give a mathematically precise description of how Turing machines act. Thus for instance we could rigorously prove that the first machine above computes the predecessor of its input.

A configuration of a Turing machine is a four-tuple $\langle q, s, \tau_L, \tau_R \rangle$, where q is a state, a member of Q, s is a character from the tape alphabet Σ, and τ_L and τ_R are strings from Σ^*, including possibly the empty string ε. These signify the current state, the character under the read/write head, and the tape contents to the left and right of the head.

We write $\mathcal{C}(t)$ for the machine's configuration after the t-th transition and say that this is the configuration at step t. We extend that to step 0 by saying that the initial configuration $\mathcal{C}(0)$ is the machine's configuration before we press `Start`.

If two configurations are related by being a step apart then we write $\mathcal{C}(i) \vdash \mathcal{C}(i+1)$. A computation is a sequence $\mathcal{C}(0) \vdash \mathcal{C}(1) \vdash \mathcal{C}(2) \vdash \cdots$. We abbreviate a sequence of $\vdash$'s with $\vdash^*$. If the computation halts then the sequence has a final configuration $\mathcal{C}(h)$ so we could write a halting computation as $\mathcal{C}(0) \vdash^* \mathcal{C}(h)$.

More details are in the text. The example below gives the idea.

EXAMPLE This machine decides whether its input is an odd number.

$$\mathcal{P}_{\text{odds}} = \{\, q_0\mathsf{BB}q_4,\ q_0\mathsf{1B}q_1,\ q_1\mathsf{BR}q_2,\ q_2\mathsf{B1}q_4,\ q_2\mathsf{1B}q_3,\ q_3\mathsf{BR}q_0 \,\}$$

In the computation steps below the machine starts with 111 on the tape and with the read/write head pointing to the leftmost 1. (Recall that in a configuration $\mathcal{C} = \langle q, s, \tau_L, \tau_R \rangle$, the first item is the state, the second is the symbol now being read on the tape, the third is the tape string to the left of the head and the fourth is the string to the right of the head.)

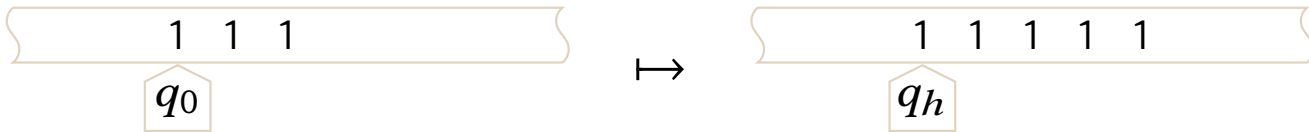
$$\begin{aligned}
\langle q_0, 1, \varepsilon, 11 \rangle &\vdash \langle q_1, \mathsf{B}, \varepsilon, 11 \rangle \\
&\vdash \langle q_2, 1, \varepsilon, 1 \rangle \\
&\vdash \langle q_2, \mathsf{B}, \varepsilon, 1 \rangle \\
&\vdash \langle q_0, 1, \varepsilon, \varepsilon \rangle \\
&\vdash \langle q_1, \mathsf{B}, \varepsilon, \varepsilon \rangle \\
&\vdash \langle q_2, \mathsf{B}, \varepsilon, \varepsilon \rangle \\
&\vdash \langle q_4, 1, \varepsilon, \varepsilon \rangle
\end{aligned}$$

This is just a slightly different presentation than the tape views that we saw earlier. The machine now halts because there is no instruction for $q_p T_p = q_4 1$.

This machine always ends with the head pointing to either a 1 or a B, and nothing else on the tape. Above we got a 1 so we interpret it as saying that the original input, 3, is odd. If instead we got a blank then we would take that to mean that the input is even.

Definition of a computable function

A function is an association of inputs with outputs. For Turing machines the natural association is to link the string on the tape when the machine starts with the one on the tape when it stops. The idea is to compute the value of a string to string function like $\phi(111) = 11111$ as here.



DEFINITION Let $\mathcal{P}$ be a Turing machine with tape alphabet Σ. For input $\sigma \in \Sigma^*$, placing that on an otherwise blank tape and pointing $\mathcal{P}$'s read/write head to σ's left-most symbol is loading that input. If we start $\mathcal{P}$ with σ loaded and it eventually halts then we denote the associated output string as $\phi_{\mathcal{P}}(\sigma)$. If the machine never halts then σ has no associated output. The function computed by the machine $\mathcal{P}$ is the set of associations $\sigma \mapsto \phi_{\mathcal{P}}(\sigma)$.

DEFINITION For $\sigma \in \Sigma^*$, if the value of a Turing machine computation is not defined on σ then we say that the function computed by the machine diverges on that input, written $\phi_{\mathcal{P}}(\sigma)\uparrow$ (or $\phi_{\mathcal{P}}(\sigma) = \perp$). Otherwise we say that it converges, $\phi_{\mathcal{P}}(\sigma)\downarrow$.

Note the difference between the machine $\mathcal{P}$ and the function computed by that machine, $\phi_{\mathcal{P}}$. For example, the machine $\mathcal{P}_{\text{pred}}$ is a set of four-tuples but the predecessor function is a set of input-output pairs, which we might denote $x \mapsto \text{pred}(x)$. Another example of the difference is that machines halt or fail to halt, while functions converge or diverge.

Discussion: representations

We will often consider a function that isn't an association of string input and output, and describe it as computed by a machine. For this we must impose an interpretation on the strings. For instance, with the predecessor machine in Example 1.1 we took the strings to represent natural numbers in unary. The same holds for computations with non-numbers, such as directed graphs, where we also just fix some encoding of the input and output strings. (We could worry that our interpretation might be so involved that, as with a horoscope, the work happens in the interpretation. But we will stick to cases such as the unary representation of numbers where this is not an issue.) Of course, the same thing happens on physical computers, where the machine twiddles bitstrings and then we interpret them as characters in a document or notes in a symphony, or however we like.

When we describe the function computed by a machine, we typically omit the part about interpreting the strings. We say, "this shows that $\phi(3) = 5$" rather than, "this shows that ϕ takes a string representing 3 to a string representing 5." The details of the representation are usually not of interest in this chapter (in the fifth chapter we will count the time or space that they consume).

DEFINITION A computable function, or recursive function, is one computed by some Turing machine (it may be a total function or partial). A computable set, or recursive set, is one whose characteristic function is computable. A Turing machine decides a set if it computes the characteristic function of that set. A relation is computable if it is computable as a set.

EXAMPLE The computable functions form a very big collection. All of these are computable.

$$f_0(x) = 2x \qquad f_1(x) = x^2 \qquad f_2(x) = 2^x$$

So are these.

$$f_3(x) = \begin{cases} x/2 & \text{– if } x \text{ is even} \\ 3x+1 & \text{– if } x \text{ is odd} \end{cases} \qquad f_4(x) = \text{the least prime number greater than x}$$

The computable sets are also a big collection. These sets are computable.

$$S_0 = \{2n \mid n \in \mathbb{N}\} \qquad S_1 = \{x \mid \text{the prime factorization has three distinct primes}\}$$

Church's Thesis

Church's Thesis

A Church asserted the following, which essentially says that Turing's characterization of what can be done by a machine is correct. It explains our spending a whole course thinking about this model of machines and related ones. We will discuss evidence for it below.

> The set of things that can be computed by a discrete and deterministic mechanism is the same as the set of things that can be computed by a Turing machine.

(Some authors call this the Church-Turing Thesis. Here we figure that because Turing has the machine, we can give Church sole possesion of the thesis.)

Evidence

Coverage Everything that people think of as intuitively computable has proven to be computable on a Turing machine.

Evidence

Coverage Everything that people think of as intuitively computable has proven to be computable on a Turing machine.

Convergence There have been many models of computation proposed by researchers and all compute the same set of things, namely the set of things computed by Turing machines.

Evidence

Coverage Everything that people think of as intuitively computable has proven to be computable on a Turing machine.

Convergence There have been many models of computation proposed by researchers and all compute the same set of things, namely the set of things computed by Turing machines.

Consistency The details of the definition, including the number of tapes or tape heads, the number of symbols available, the number of states, all don't change the collection of things computed. (This includes the condition of determinism.)

Evidence

Coverage Everything that people think of as intuitively computable has proven to be computable on a Turing machine.

Convergence There have been many models of computation proposed by researchers and all compute the same set of things, namely the set of things computed by Turing machines.

Consistency The details of the definition, including the number of tapes or tape heads, the number of symbols available, the number of states, all don't change the collection of things computed. (This includes the condition of determinism.)

Clarity Turing's analysis is persuasive of itself.

An empirical question?

As yet, there is no persuasive evidence of physically possible systems that allow you to compute things that a Turing machine cannot compute. (We shall take proposals involving wormholes, black hole event horizons, etc., as too speculative to be as-yet persuasive.)

In particular, we expect that quantum computers, should they become practical, will run faster on some problems. But they are not more capable in principle — the set of things computed by quantum computers is the same as the set of things computed by Turing machines.

Church's Thesis in arguments

Church's Thesis equates 'computable' with 'computable by a Turing machine'. We will leverage this in two ways.

1. It gives our results a larger, philosophical, importance. When we produce a result of the form *No Turing machine can . . .* then we can use Church's Thesis to interpret it as *No mechanism can . . .*

Church's Thesis in arguments

Church's Thesis equates 'computable' with 'computable by a Turing machine'. We will leverage this in two ways.

1. It gives our results a larger, philosophical, importance. When we produce a result of the form *No Turing machine can . . .* then we can use Church's Thesis to interpret it as *No mechanism can . . .*
2. To prove results about Turing machines we often argue intuitively. For instance, consider the problem of whether this function is computable: it inputs a number n and returns 1 when n is a prime, and 0 otherwise. To show that this function can be computed, we will not exhibit a Turing machine set of four-tuple instructions. Instead we will argue that the function is intuitively computable, often by giving some pseudo-code and perhaps some actual code, and then invoke Church's Thesis. Working intuitively is potentially hazardous. However, we have so much experience with computation that our intuition is quite reliable. For us, there is more of a danger of being overwhelmed by the detail of four-tuples than a danger of being too hand-wavy about algorithms.

Primitive recursive functions

Definition by recursion

Grade school students learn addition and multiplication as mildly complicated algorithms (multiplication, for example, involves arranging the digits into a table, doing partial products from right to left, and then adding). In 1861, H Grassmann produced a more elegant definition. Here is the formula for addition, plus: $\mathbb{N}^2 \longrightarrow \mathbb{N}$, which takes as given the successor map, $\mathcal{S}(n) = n + 1$.

$$\text{plus}(x, y) = \begin{cases} x & \text{– if } y = 0 \\ \mathcal{S}(\text{plus}(x, z)) & \text{– if } y = \mathcal{S}(z) \text{ for } z \in \mathbb{N} \end{cases}$$

EXAMPLE This finds the sum of 3 and 2.

$$\text{plus}(3, 2) = \mathcal{S}(\text{plus}(3, 1)) = \mathcal{S}(\mathcal{S}(\text{plus}(3, 0))) = \mathcal{S}(\mathcal{S}(3)) = 5$$

Besides being compact, this has a very interesting feature: 'plus' recurs in its own definition. This is definition by recursion. Whereas the grade school definition of addition is prescriptive in that it gives a procedure, this recursive definition has the advantage of being descriptive because it specifies the meaning, the semantics, of the operation.

Multiplication has the same form.

$$\text{product}(x, y) = \begin{cases} 0 & \text{– if } y = 0 \\ \text{plus}(\text{product}(x, z), x) & \text{– if } y = \mathcal{S}(z) \end{cases}$$

EXAMPLE The expansion of $\text{product}(2, 3)$ reduces to a sum of three 2's.

$$\begin{aligned} \text{product}(2, 3) &= \text{plus}(\text{product}(2, 2), 2) \\ &= \text{plus}(\text{plus}(\text{product}(2, 1), 2), 2) \\ &= \text{plus}(\text{plus}(\text{plus}(\text{product}(2, 0), 2), 2), 2) \\ &= \text{plus}(\text{plus}(\text{plus}(0, 2), 2), 2) \end{aligned}$$

Multiplication has the same form.

$$\text{product}(x, y) = \begin{cases} 0 & - \text{if } y = 0 \\ \text{plus}(\text{product}(x, z), x) & - \text{if } y = \mathcal{S}(z) \end{cases}$$

EXAMPLE The expansion of $\text{product}(2, 3)$ reduces to a sum of three 2's.

$$\begin{aligned} \text{product}(2, 3) &= \text{plus}(\text{product}(2, 2), 2) \\ &= \text{plus}(\text{plus}(\text{product}(2, 1), 2), 2) \\ &= \text{plus}(\text{plus}(\text{plus}(\text{product}(2, 0), 2), 2), 2) \\ &= \text{plus}(\text{plus}(\text{plus}(0, 2), 2), 2) \end{aligned}$$

And exponentiation works the same way.

$$\text{power}(x, y) = \begin{cases} 1 & - \text{if } y = 0 \\ \text{product}(\text{power}(x, z), x) & - \text{if } y = \mathcal{S}(z) \end{cases}$$

EXAMPLE

$$\begin{aligned} \text{power}(3, 2) &= \text{product}(\text{power}(3, 1), 3) \\ &= \text{product}(\text{product}(\text{power}(3, 0), 3), 3) = 9 \end{aligned}$$

Defining by primitive recursion

DEFINITION A function f is defined by the schema of primitive recursion from the functions g and h when this holds.

$$f(x_0, \ldots x_{k-1}, y) = \begin{cases} g(x_0, \ldots x_{k-1}) & - \text{ if } y = 0 \\ h(f(x_0, \ldots x_{k-1}, z), x_0, \ldots x_{k-1}, z) & - \text{ if } y = \mathcal{S}(z) \end{cases}$$

The bookkeeping is that the arity of f, the number of inputs, is one more than the arity of g and one less than the arity of h.

EXAMPLE The function plus is defined by primitive recursion from $g(x_0) = x_0$ and $h(w, x_0, z) = \mathcal{S}(w)$. The function product is defined by primitive recursion from $g(x_0) = 0$ and $h(w, x_0, z) = \text{plus}(w, x_0)$. The function power is defined by primitive recursion from $g(x_0) = 1$ and $h(w, x_0, z) = \text{product}(w, x_0)$.

EXAMPLE The predecessor function is like an inverse to successor. However, since we use the natural numbers we can't give a predecessor of zero, so instead we describe $\text{pred}\colon \mathbb{N} \to \mathbb{N}$ by: $\text{pred}(y)$ equals $y-1$ if $y>0$ and equals 0 if $y=0$. This definition fits the primitive recursive schema.

$$\text{pred}(y) = \begin{cases} 0 & \text{– if } y=0 \\ z & \text{– if } y=\mathcal{S}(z) \end{cases}$$

The arity bookkeeping is that pred has no x_i's so g is a function of zero-many inputs and is therefore constant, $g(\,)=0$, while h has two inputs, $h(a,b)=b$ (the first input gets ignored, that is, a is just a dummy variable).

EXAMPLE Consider proper subtraction, denoted $x \dot{-} y$, described by: if $x \geq y$ then $x \dot{-} y$ equals $x-y$ and otherwise $x \dot{-} y$ equals 0. This definition of that function fits the primitive recursion schema.

$$\text{propersub}(x,y) = \begin{cases} x & \text{– if } y=0 \\ \text{pred}(\text{propersub}(x,z)) & \text{– if } y=\mathcal{S}(z) \end{cases}$$

In the terms of Theorem 3.6, $g(x_0)=x_0$ and $h(w,x_0,z)=\text{pred}(w)$; the bookkeeping works since the arity of g is one less than the arity of f, and, because h has dummy arguments, its arity is one more than the arity of f.

Collection of Primitive Recursive Functions

DEFINITION The set of primitive recursive functions consists of those that can be derived from the initial operations of the zero function $\mathcal{Z}(\vec{x}) = \mathcal{Z}(x_0, \ldots x_{k-1}) = 0$, the successor function $\mathcal{S}(\vec{x}) = x + 1$, and the projection functions $\mathcal{I}_i(\vec{x}) = x_i$ where $i \in \{0, \ldots k-1\}$, by a finite number of applications of the combining operations of function composition and primitive recursion.

Function composition covers not just the simple case of two functions f and g whose composition is defined by $f \circ g\,(\vec{x}) = f(g(\vec{x}))$. It also covers simultaneous substitution, where from $f(x_0, \ldots, x_n)$ and $h_0(y_{0,0}, \ldots, y_{0,m_0})$, . . ., and $h_n(y_{n,0}, \ldots, y_{n,m_n})$, we get $f(\, h_0(y_{0,0}, \ldots, y_{0,m_0}), \ldots h_n(y_{n,0}, \ldots, y_{n,m_n})\,)$, which is a function with $(m_0 + 1) + \cdots + (m_n + 1)$-many inputs.

The collection of primitive recursive functions is very large. It includes addition, proper subtraction, multiplication, integer division, exponentiation, the equality Boolean function, the remainder function, the sum of a bounded number of terms, the product of a bounded number of terms, and just about every function that comes to mind.

General Recursive Functions

Ackermann's function

Every primitive recursive function is intuitively mechanically computable. What about the converse: is every mechanically computable function primitive recursive? Here we will answer 'no'.

Recall that the addition operation is repeated successor, that multiplication is repeated addition, and that exponentiation is repeated multiplication.

$$x + y = \underbrace{\mathcal{S}(\,\mathcal{S}(\,\cdots\mathcal{S}(x)))}_{y \text{ many}} \qquad x \cdot y = \underbrace{x + x + \cdots + x}_{y \text{ many}} \qquad x^y = \underbrace{x \cdot x \cdot \cdots \cdot x}_{y \text{ many}}$$

This is a compelling pattern.

The pattern is especially striking when we express these functions using the schema of primitive recursion. For that, start by defining $\mathcal{H}_0$ to be the successor function, $\mathcal{H}_0 = \mathcal{S}$.

$$\text{plus}(x,y) = \mathcal{H}_1(x,y) = \begin{cases} x & \text{– if } y = 0 \\ \mathcal{H}_0(x, \mathcal{H}_1(x, y-1)) & \text{– otherwise} \end{cases}$$

$$\text{product}(x,y) = \mathcal{H}_2(x,y) = \begin{cases} 0 & \text{– if } y = 0 \\ \mathcal{H}_1(x, \mathcal{H}_2(x, y-1)) & \text{– otherwise} \end{cases}$$

$$\text{power}(x,y) = \mathcal{H}_3(x,y) = \begin{cases} 1 & \text{– if } y = 0 \\ \mathcal{H}_2(x, \mathcal{H}_3(x, y-1)) & \text{– otherwise} \end{cases}$$

The pattern shows in the 'otherwise' lines. Each one satisfies that $\mathcal{H}_n(x,y) = \mathcal{H}_{n-1}(x, \mathcal{H}_n(x, y-1))$.

Because of this pattern we call each $\mathcal{H}_n$ the level n function, so that successor is level 0, addition is the level 1 operation, multiplication is the level 2 operation, and exponentiation is level 3. The definition below takes n as a parameter, writing $\mathcal{H}(n,x,y)$ in place of $\mathcal{H}_n(x,y)$, to get all of the levels into one formula.

DEFINITION This is the hyperoperation $\mathcal{H}\colon \mathbb{N}^3 \to \mathbb{N}$.

$$\mathcal{H}(n,x,y) = \begin{cases} y+1 & \text{– if } n=0 \\ x & \text{– if } n=1 \text{ and } y=0 \\ 0 & \text{– if } n=2 \text{ and } y=0 \\ 1 & \text{– if } n>2 \text{ and } y=0 \\ \mathcal{H}(n-1,x,\mathcal{H}(n,x,y-1)) & \text{– otherwise} \end{cases}$$

The hyperoperation is an Ackermann's function.

LEMMA $\mathcal{H}_0(x,y) = y+1$, $\mathcal{H}_1(x,y) = x+y$, $\mathcal{H}_2(x,y) = x\cdot y$, $\mathcal{H}_3(x,y) = x^y$.

PF. The level 0 statement $\mathcal{H}_0(x,y) = y+1$ is in the definition of $\mathcal{H}$.

We prove the level 1 statement $\mathcal{H}_1(x,y) = x+y$ by induction on y. For the $y=0$ base step, the definition is that $\mathcal{H}(1,x,0) = x$, which equals $x+0 = x+y$. For the inductive step, assume that the statement holds for $y=0,\ldots,y=k$ and consider the $y=k+1$ case. The definition is $\mathcal{H}_1(x,k+1) = \mathcal{H}_0(x,\mathcal{H}_1(x,k))$. Apply the inductive hypothesis to get $\mathcal{H}_0(x,x+k)$. By the prior paragraph this equals $x+k+1 = x+y$.

The other two, $\mathcal{H}_2$ and $\mathcal{H}_3$, are Exercise 4.15. ∎

Our interest in the $\mathcal{H}$ operation is that it is intuitively computable but not primitive recursive.

THEOREM The hyperoperation $\mathcal{H}$ is not primitive recursive.

(This is proved in an Extra section.)

REMARK At first glance it could appear that many of the arguments for Church's Thesis apply to the set of primitive recursive functions. However this theorem, proved by Ackermann, shows that there are functions that are intuitively mechnically computable but which are not primitive recursive. So we need more than the schema of primitive recursion.

μ recursion

DEFINITION Suppose that $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is total, so that for every input tuple there is an output number. Then $f\colon \mathbb{N}^n \to \mathbb{N}$ is defined from g by minimization or μ-recursion, written $f(\vec{x}) = \mu y \left[g(\vec{x}, y) = 0\right]$, if $f(\vec{x})$ is the the least number y such that $g(\vec{x}, y) = 0$.

This is unbounded search: we have in mind that g is mechanically computable and we calculate $g(\vec{x}, 0)$, then $g(\vec{x}, 1)$, etc., waiting until one of them gives the output 0. If that ever happens, so that $g(\vec{x}, n) = 0$ for some least n, then $f(\vec{x}) = n$. If it never happens that the output is 0 then $f(\vec{x})$ is undefined.

μ recursion

DEFINITION Suppose that $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is total, so that for every input tuple there is an output number. Then $f\colon \mathbb{N}^n \to \mathbb{N}$ is defined from g by minimization or μ-recursion, written $f(\vec{x}) = \mu y \left[g(\vec{x}, y) = 0 \right]$, if $f(\vec{x})$ is the the least number y such that $g(\vec{x}, y) = 0$.

This is unbounded search: we have in mind that g is mechanically computable and we calculate $g(\vec{x}, 0)$, then $g(\vec{x}, 1)$, etc., waiting until one of them gives the output 0. If that ever happens, so that $g(\vec{x}, n) = 0$ for some least n, then $f(\vec{x}) = n$. If it never happens that the output is 0 then $f(\vec{x})$ is undefined.

EXAMPLE Euler noticed in 1772 that the polynomial $p(y) = y^2 + y + 41$ initially seems to output only primes.

y	0	1	2	3	4	5	6	7	8	9
$p(y)$	41	43	47	53	61	71	83	97	113	131

We could test whether this ever fails by doing an unbounded search for non-primes. This function tests for the primality of a quadratic polynomial's output.

$$g(x_0, x_1, x_2, y) = g(\vec{x}, y) = \begin{cases} 1 & \text{– if } x_0 y^2 + x_1 y + x_2 \text{ is prime} \\ 0 & \text{– otherwise} \end{cases}$$

Then the search is $f(\vec{x}) = \mu y \left[g(\vec{x}, y) = 0 \right]$. (Note that $f(1, 1, 41) = 40$.)

EXAMPLE Goldbach's conjecture is that every even number can be written as the sum of at most two primes. Here are the first few instances: $2 = 2$, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 5 + 3$, $10 = 7 + 3$. No one knows if it is true. A natural attack is to search for a counterexample. With this auxillary function

$$\text{gb-check}(n) = \begin{cases} i & \text{– the numbers } i \text{ and } n-i \text{ are both prime, and } i \leq n-i \\ 0 & \text{– otherwise} \end{cases}$$

we get a version of the definition's g (here $\vec{x}$ is empty).

$$\text{gb-g}(y) \begin{cases} 1 & \text{– if gb-check}(2y) \neq 0 \\ 0 & \text{– otherwise} \end{cases}$$

The unbounded search is $\text{gb-f}() = \mu y \left[\text{gb-g}(y) = 0 \right]$.

EXAMPLE Goldbach's conjecture is that every even number can be written as the sum of at most two primes. Here are the first few instances: $2 = 2$, $4 = 2 + 2$, $6 = 3 + 3$, $8 = 5 + 3$, $10 = 7 + 3$. No one knows if it is true. A natural attack is to search for a counterexample. With this auxillary function

$$\text{gb-check}(n) = \begin{cases} i & \text{– the numbers } i \text{ and } n-i \text{ are both prime, and } i \leq n-i \\ 0 & \text{– otherwise} \end{cases}$$

we get a version of the definition's g (here $\vec{x}$ is empty).

$$\text{gb-g}(y) \begin{cases} 1 & \text{– if gb-check}(2y) \neq 0 \\ 0 & \text{– otherwise} \end{cases}$$

The unbounded search is $\text{gb-f}() = \mu y \left[\text{gb-g}(y) = 0 \right]$.

EXAMPLE A composite number n such that $b^n \equiv b \pmod{n}$ for all b is a Carmichael number. The first one is 561. This Boolean function is primitive recursive.

$$C(n) = \begin{cases} 1 & \text{– if } n \text{ is a Carmichael number} \\ 0 & \text{– otherwise} \end{cases}$$

Here we define the x-th Carmichael number $F\colon \mathbb{N} \to \mathbb{N}$ by μ-recursion.

$$F(x) = \begin{cases} 561 & \text{– if } x = 0 \\ \mu y \left[y > F(x-1) \text{ and } 1 - C(y) = 0 \right] & \text{– otherwise} \end{cases}$$

DEFINITION A function is general recursive or partial recursive, or μ-recursive, or just recursive, if it can be derived from the initial operations of the zero function $\mathcal{Z}(\vec{x}) = 0$, the successor function $\mathcal{S}(x) = x + 1$, and the projection functions $\mathcal{I}_i(x_0, \ldots, x_i \ldots x_{k-1}) = x_i$ by a finite number of applications of function composition, the schema of primitive recursion, and minimization.

It is beyond our scope, but we could prove that the collection of general recursive functions is the same set as the collection of functions computed by some Turing machine, which we call the 'computable functions'.

(Some authors define the computable functions as the recursive functions, in contrast with our using Turing machines. Such a development has the advantage of going straight to functions without having to do representations. But it has the disadvantage of losing the evident connection with what can be mechanically computed that comes with Turing's approach.)