Background

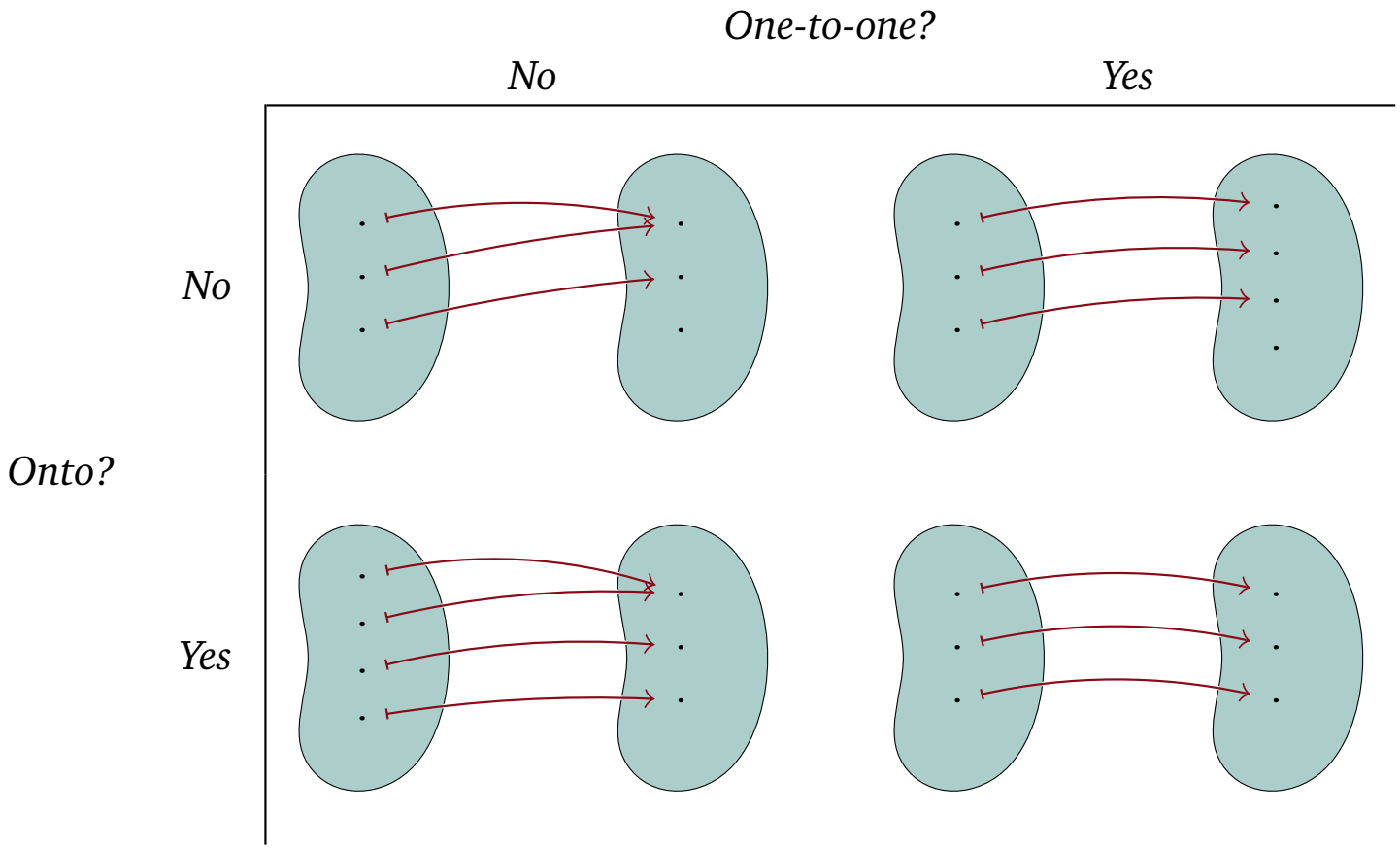
Jim Hefferon

University of Vermont

hefferon.net

Cardinality

Finite sets have the same size iff there is a correspondence

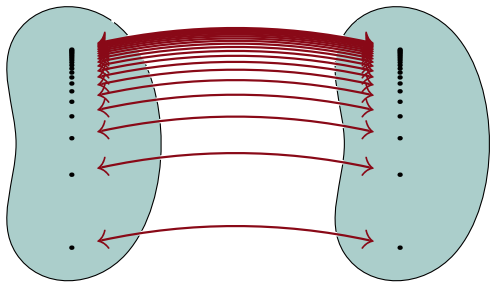


LEMMA For a function with a finite domain, the number of elements in its domain is greater than or equal to the number of elements in its range. If the function is one-to-one then its domain has the same number of elements as its range, while if it is not one-to-one then its domain has more elements than its range. Consequently, two finite sets have the same number of elements if and only if they correspond, that is, if and only if there is a function from one to the other that is a correspondence.

Cardinality definition

LEMMA The relation between two sets of 'there is a correspondence from one to the other' is an equivalence relation.

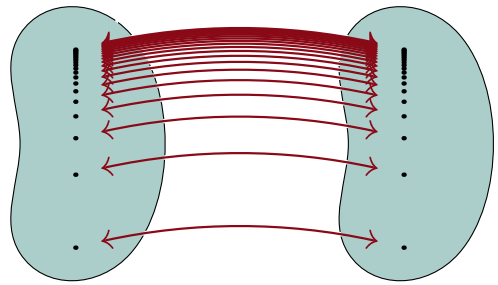
DEFINITION Two sets have the same cardinality or are equinumerous, denoted $|S_0| = |S_1|$, if there is a correspondence between them.



Cardinality definition

LEMMA The relation between two sets of 'there is a correspondence from one to the other' is an equivalence relation.

DEFINITION Two sets have the same cardinality or are equinumerous, denoted $|S_0| = |S_1|$, if there is a correspondence between them.

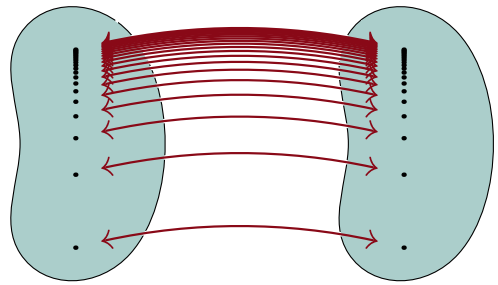


EXAMPLE These have the same cardinality: (1) $|A| = |B|$ where $A = \{x \in \mathbb{R} \mid 0 \leq x < 1\}$ and $B = \{x \in \mathbb{R} \mid 1 \leq x < 5\}$ (2) $|C| = |D|$ where $C = \{x \in \mathbb{N} \mid 0 \leq x < 10\}$ and $D = \{x \in \mathbb{N} \mid 1 \leq x < 11\}$. Further, (3) if $I = \{x \in \mathbb{R} \mid 0 < x < 1\}$ then $|I| = |\mathbb{R}|$ and (4) if $S = \{x \in \mathbb{N} \mid x \text{ is a perfect square}\}$ then $|S| = |\mathbb{N}|$.

Cardinality definition

LEMMA The relation between two sets of 'there is a correspondence from one to the other' is an equivalence relation.

DEFINITION Two sets have the same cardinality or are equinumerous, denoted $|S_0| = |S_1|$, if there is a correspondence between them.



EXAMPLE These have the same cardinality: (1) $|A| = |B|$ where $A = \{x \in \mathbb{R} \mid 0 \leq x < 1\}$ and $B = \{x \in \mathbb{R} \mid 1 \leq x < 5\}$ (2) $|C| = |D|$ where $C = \{x \in \mathbb{N} \mid 0 \leq x < 10\}$ and $D = \{x \in \mathbb{N} \mid 1 \leq x < 11\}$. Further, (3) if $I = \{x \in \mathbb{R} \mid 0 < x < 1\}$ then $|I| = |\mathbb{R}|$ and (4) if $S = \{x \in \mathbb{N} \mid x \text{ is a perfect square}\}$ then $|S| = |\mathbb{N}|$.

DEFINITION A set is finite if it has the same cardinality as $\{0, 1, \ldots n\}$ for some $n \in \mathbb{N}$, or if it is empty. Otherwise it is infinite.

DEFINITION A set with the same cardinality as the natural numbers is countably infinite. A set that is either finite or countably infinite is countable. If a set is the range of a function whose domain is the natural numbers then we say the function enumerates, or is an enumeration of, that set.

EXAMPLE The set $\{ n^2 \mid n \in \mathbb{N} \}$ is countably infinite. It is enumerated by the function $f : \mathbb{N} \to \mathbb{N}$ given by $f(x) = x^2$. Note that this enumeration is effective.

EXAMPLE The set $\mathbb{N} - \{0, 1, 2\} = \{3, 4, 5, 6, 7, \ldots\}$ is countably infinite. The function $f(x) = x + 3$ closes the gap.

n	0	1	2	3	4	5	6	...
$f(n)$	3	4	5	6	7	8	9	...

This function is clearly one-to-one and onto, as well as computable.

EXAMPLE The set of prime numbers P is countable. There is a function $p : \mathbb{N} \to P$ where $p(n)$ is the n-th prime, so that $p(0) = 2$, $p(1) = 3$, etc.

EXAMPLE The set of integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ is countable. The natural correspondence alternates between positive and negative numbers.

$n \in \mathbb{N}$	0	1	2	3	4	5	6	...
$f(n) \in \mathbb{Z}$	0	+1	−1	+2	−2	+3	−3	...

Cross product

EXAMPLE If $N_2 = \{0, 1\} \times \mathbb{N}$ then $|N_2| = |\mathbb{N}|$.

$$\begin{array}{cc} \vdots & \vdots \\ \langle 0, 3\rangle & \langle 1, 3\rangle \\ \langle 0, 2\rangle & \langle 1, 2\rangle \\ \langle 0, 1\rangle & \langle 1, 1\rangle \\ \langle 0, 0\rangle & \langle 1, 0\rangle \end{array}$$

The picture above shows N_2 as two $\mathbb{N}$'s. Nonetheless we can enumerate it.

Cross product

EXAMPLE If $N_2 = \{0, 1\} \times \mathbb{N}$ then $|N_2| = |\mathbb{N}|$.

$$\begin{array}{cc} \vdots & \vdots \\ \langle 0,3\rangle & \langle 1,3\rangle \\ \langle 0,2\rangle & \langle 1,2\rangle \\ \langle 0,1\rangle & \langle 1,1\rangle \\ \langle 0,0\rangle & \langle 1,0\rangle \end{array}$$

The picture above shows N_2 as two $\mathbb{N}$'s. Nonetheless we can enumerate it.

$n \in \mathbb{N}$	0	1	2	3	4	5	...
$\langle i, j\rangle \in S$	$\langle 0,0\rangle$	$\langle 1,0\rangle$	$\langle 0,1\rangle$	$\langle 1,1\rangle$	$\langle 0,2\rangle$	$\langle 1,2\rangle$	...

The map from the table's top row to the bottom is a pairing function. Its inverse, from bottom to top, is an unpairing function.

EXAMPLE If $N_3 = \{0, 1, 2\} \times \mathbb{N}$ then $|N_3| = |\mathbb{N}|$. For any finite k, if $N_k = \{0, 1, \ldots k-1\} \times \mathbb{N}$ then $|N_k| = |\mathbb{N}|$.

LEMMA The cross product of two finite sets is finite, and therefore countable. The cross product of a finite set and a countably infinite set, or of a countably infinite set and a finite set, is countably infinite.

Cantor's correspondence

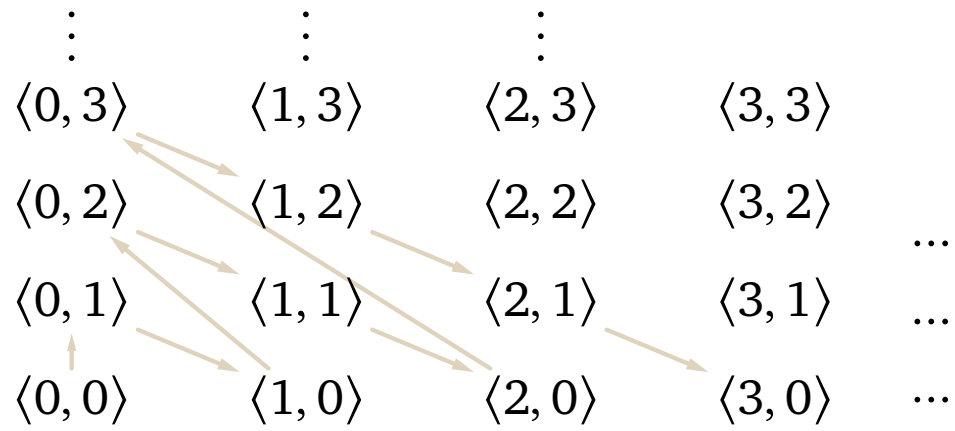
Cantor's correspondence

Next is to enumerate an array that is unbounded in two dimensions.

$$
\begin{array}{cccc}
\vdots & \vdots & \vdots & \\
\langle 0,3\rangle & \langle 1,3\rangle & \langle 2,3\rangle & \langle 3,3\rangle \\
\langle 0,2\rangle & \langle 1,2\rangle & \langle 2,2\rangle & \langle 3,2\rangle \\
\langle 0,1\rangle & \langle 1,1\rangle & \langle 2,1\rangle & \langle 3,1\rangle \\
\langle 0,0\rangle & \langle 1,0\rangle & \langle 2,0\rangle & \langle 3,0\rangle
\end{array}
\begin{array}{c}
\\
\\
\cdots \\
\cdots \\
\cdots
\end{array}
$$

Cantor's correspondence

Next is to enumerate an array that is unbounded in two dimensions.

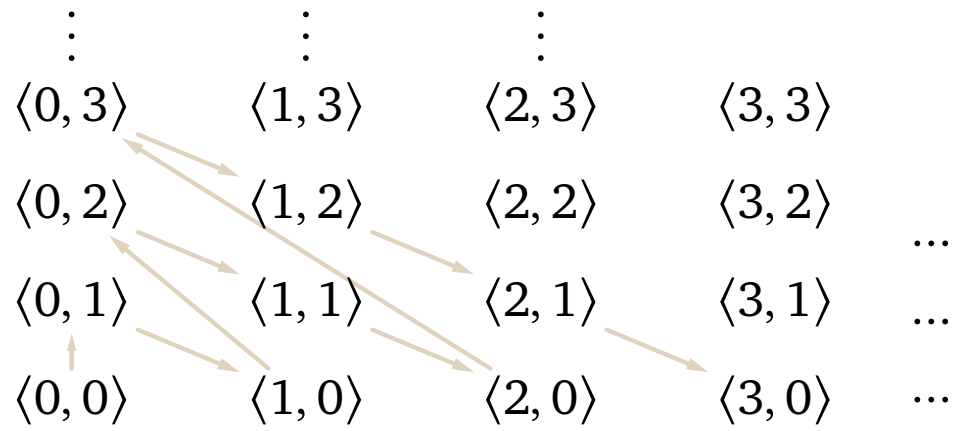


$n \in \mathbb{N}$	0	1	2	3	4	5	6	...
$\langle x, y\rangle \in \mathbb{N}^2$	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 0,2\rangle$	$\langle 1,1\rangle$	$\langle 2,0\rangle$	$\langle 0,3\rangle$	...

DEFINITION Cantor's correspondence cantor: $\mathbb{N}^2 \to \mathbb{N}$ or unpairing function, is the above correspondence. Its inverse $\text{cantor}^{-1}: \mathbb{N} \to \mathbb{N}^2$ is Cantor's pairing function. (A notation for $\text{cantor}(x, y)$ common elsewhere is $\langle x, y\rangle$.)

Cantor's correspondence

Next is to enumerate an array that is unbounded in two dimensions.



$n \in \mathbb{N}$	0	1	2	3	4	5	6	...
$\langle x, y\rangle \in \mathbb{N}^2$	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 0,2\rangle$	$\langle 1,1\rangle$	$\langle 2,0\rangle$	$\langle 0,3\rangle$	...

DEFINITION Cantor's correspondence cantor: $\mathbb{N}^2 \to \mathbb{N}$ or unpairing function, is the above correspondence. Its inverse cantor^{-1}: $\mathbb{N} \to \mathbb{N}^2$ is Cantor's pairing function. (A notation for cantor(x, y) common elsewhere is $\langle x, y\rangle$.)

LEMMA The cross product $\mathbb{N} \times \mathbb{N}$ is countable, for instance under Cantor's correspondence cantor: $\mathbb{N}^2 \to \mathbb{N}$. Further, the sets $\mathbb{N}^3 = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, and $\mathbb{N}^4$, ... are all countable.

COROLLARY The cross product of finitely many countable sets is countable.

Counting Turing machines

Each Turing machine instruction is a four-tuple, a member of $Q \times \Sigma \times (\Sigma \cup \{\mathsf{L}, \mathsf{R}\}) \times Q$, where Q is the set of states and Σ is the tape alphabet. The results of this section imply that we can enumerate the set of instructions, so that there is an instruction that corresponds to 0, one corresponding to 1, etc. This enumeration is effective — there is a program that takes in a natural number and outputs the corresponding instruction, as well as a program that takes in an instruction and outputs the corresponding number.

With that, we can effectively number the Turing machines. The exact numbering that we use doesn't matter much as long as it is has the properties in the definition below.

But for illustration here is one way: starting with a Turing machine $\mathcal{P}$, use the prior paragraph to convert each of its instructions to a number, giving a set $\{i_0, i_1, \ldots i_n\}$. Then define the number e associated with $\mathcal{P}$ to be the one that when written in binary has 1 in bits $i_0, \ldots i_n$, that is, $e = 2^{i_0} + 2^{i_1} + \cdots + 2^{i_n}$. This association is effective. Its inverse is also effective: given $e \in \mathbb{N}$, represent it as $e = 2^{j_0} + \cdots + 2^{j_k}$ and the set of instructions corresponding to the numbers $j_0, \ldots j_k$ is the desired Turing machine. (Except that we must first check that the instruction set is deterministic, that no two of the instructions begin with the same $q_p T_p$. We can check this effectively and if it is not true then let the machine associated with e be empty, $\mathcal{P} = \{\}$.)

Definition A numbering is a function that assigns to each Turing machine a natural number. A numbering is acceptable if it is effective: (1) there is a program that takes as input the set of instructions and gives as output the associated number, (2) the set of numbers for which there is an associated machine is computable, and (3) there is an effective inverse that takes as input a natural number and gives as output the associated machine.

We fix some acceptable numbering that we will use for the rest of the course.

DEFINITION A numbering is a function that assigns to each Turing machine a natural number. A numbering is acceptable if it is effective: (1) there is a program that takes as input the set of instructions and gives as output the associated number, (2) the set of numbers for which there is an associated machine is computable, and (3) there is an effective inverse that takes as input a natural number and gives as output the associated machine.

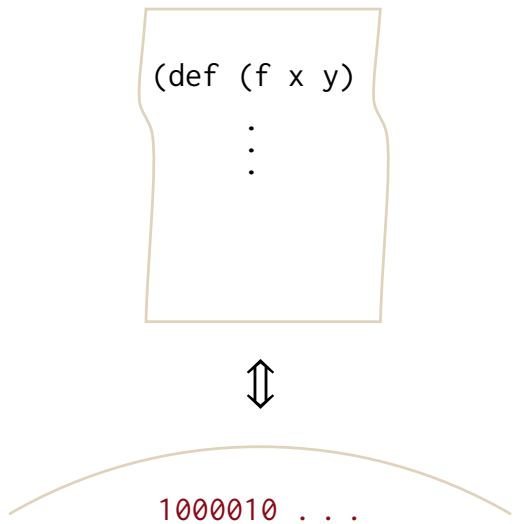
We fix some acceptable numbering that we will use for the rest of the course.

LEMMA (PADDING LEMMA) Every computable function has infinitely many indices: if f is computable then there are infinitely many distinct $e_i \in \mathbb{N}$ with $f = \phi_{e_0} = \phi_{e_1} = \cdots$. We can effectively produce a list of such indices.

PF. Let $f = \phi_e$. Let q_j be the highest-numbered state in $\mathcal{P}_e$. For each $k \in \mathbb{N}^+$ consider the Turing machine obtained from $\mathcal{P}_e$ by adding the instruction $q_{j+k}\mathsf{BB}q_{j+k}$, This gives an effective sequence of Turing machines $\mathcal{P}_{e_1}, \mathcal{P}_{e_2}, \ldots$ with distinct indices, all having the same behavior, $\phi_{e_k} = \phi_e = f$. ∎

A way to informally think about numbering

Turing machines are like programs. Imagine that you write a program $\mathcal{P}$ and save it to disc. It lives on the hard drive as a bitstring. Think of that bitstring as a number, e, written in binary. In this analogy, the program and the number correspond. In both directions the association is effective: from the source code in your editor the system derives its bit string representation, and from the bit string representation on the disc the system can recover the program's source.



This is only an analogy and it isn't perfect — one problem is that leading `0`'s in the bit string cause ambiguity — but it can helpin thinking about numbering. A Turing machine's index number e is a name, a way to refer to that machine. That reference is effective, meaning that there is a program that goes from the Turing machine source to the index and a program that goes from the index to the source.

Diagonalization

There are sets that cannot be counted

THEOREM There is no onto map $f : \mathbb{N} \to \mathbb{R}$. Hence, the set of reals is not countable.

Before the proof, an example.

n	*Decimal expansion of* $f(n)$
0	42 . 3 1 2 7 7 0 4 ...
1	2 . 0 1 0 0 0 0 0 ...
2	1 . 4 1 4 1 5 9 2 ...
3	−20 . 9 1 9 5 9 1 9 ...
4	0 . 1 0 1 0 0 1 0 ...
5	−0 . 6 2 5 5 4 1 8 ...
⋮	⋮

We will produce a number $z \in \mathbb{R}$ that does not equal any of the $f(n)$'s.

There are sets that cannot be counted

THEOREM There is no onto map $f : \mathbb{N} \to \mathbb{R}$. Hence, the set of reals is not countable.

Before the proof, an example.

n	*Decimal expansion of* $f(n)$
0	$42.3127704\ldots$
1	$2.0100000\ldots$
2	$1.4141592\ldots$
3	$-20.9195919\ldots$
4	$0.1010010\ldots$
5	$-0.6255418\ldots$
$\vdots$	$\vdots$

We will produce a number $z \in \mathbb{R}$ that does not equal any of the $f(n)$'s.

Ignore what is to the left of the decimal point. To its right go down the diagonal, taking the digits 3, 1, 4, 5, 0, 1 . . . Construct the desired z by making its first decimal place something other than 3, making its second decimal place something other than 1, etc. Specifically, if the diagonal digit is a 1 then z gets a 2 in that decimal place and otherwise z gets a 1 there. Thus, in this example $z = 0.121112\ldots$

THEOREM There is no onto map $f : \mathbb{N} \to \mathbb{R}$. Hence, the set of reals is not countable.

PF. We will show that no map $f : \mathbb{N} \to \mathbb{R}$ is onto.

Denote the i-th decimal digit of $f(n)$ as $f(n)[i]$ (if $f(n)$ is a number with two decimal representations then use the one ending in 0's). Let g be the map on the decimal digits $\{0, \ldots, 9\}$ given by: $g(j) = 2$ if j is 1 and $g(j) = 1$ otherwise.

Now let z be the real number that has 0 to the left of its decimal point, and whose i-th decimal digit is $g(f(i)[i])$. Then for all i, $z \neq f(i)$ because $z[i] \neq f(i)[i]$. So f is not onto. ∎

Uncountable sets

DEFINITION A set that is infinite but not countable is uncountable.

EXAMPLE There are many uncountable sets. We can adjust the argument from the prior slide to show that there is no onto function from $\mathbb{N}$ to S where $S = \{ x \in \mathbb{R} \mid x > 5 \}$, or $S = [\sqrt{3} \,..\, 10) = \{ x \in \mathbb{R} \mid \sqrt{3} \leq x < 10 \}$.

Hierarchy of infinities

DEFINITION The set S has cardinality less than or equal to that of the set T, denoted $|S| \leq |T|$, if there is a one-to-one function from S to T.

EXAMPLE For finite sets it just means that the number of elements in S is less than or equal to the number of elements in T. An example is $S = \{10, 11, 12\}$ and $T = \{0, 1, 2, 3\}$. The function $f : S \to T$ given by $10 \mapsto 0$, and $11 \mapsto 1$, and $12 \mapsto 2$ is one-to-one and so $|S| \leq |T|$.

EXAMPLE Let $S = \{0, 2, 4, \ldots\}$ be the even numbers. The inclusion map $f : S \to \mathbb{N}$ given by $s \mapsto s$ is one-to-one. So $|S| \leq |\mathbb{N}|$.

EXAMPLE Similarly, $|\mathbb{N}| \leq |\mathbb{R}|$ via the inclusion map $f(x) = x$. Since $\mathbb{R}$ is uncountable, this less-than relation between cardinalities is strict.

EXAMPLE Let $S = [0\,..\,2)$ and $T = [10\,..\,16)$ be intervals of real numbers. The function $f : \mathbb{R} \to \mathbb{R}$ given by $f(x) = 3x + 10$ is one-to-one, so $|S| \leq |T|$.

(Also one-to-one is the function $g : \mathbb{R} \to \mathbb{R}$ given by $g(y) = (y - 10)/6$ and so we also have $|T| \leq |S|$. A result that we cover in the exercises says that if there is a one-to-one function both ways then the two sets correspond, so $|T| = |S|$.)

Cantor's Theorem

For the next result, recall that a set's characteristic function $\mathbb{1}_S$ is the Boolean function determining membership: $\mathbb{1}_S(s) = T$ if $s \in S$ and $\mathbb{1}_S(s) = F$ if $s \notin S$. (We sometimes instead use the bits `1` for T and `0` for F.)

THEOREM (CANTOR'S THEOREM) A set's cardinality is strictly less than that of its power set.

Cantor's Theorem

For the next result, recall that a set's characteristic function $\mathbb{1}_S$ is the Boolean function determining membership: $\mathbb{1}_S(s) = T$ if $s \in S$ and $\mathbb{1}_S(s) = F$ if $s \notin S$. (We sometimes instead use the bits 1 for T and 0 for F.)

THEOREM (CANTOR'S THEOREM) A set's cardinality is strictly less than that of its power set.

Before the proof, an example. The harder half is showing that no map from S to $\mathcal{P}(S)$ is onto. For example, consider the set $S = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and this function $f : S \to \mathcal{P}(S)$.

$$\mathsf{a} \overset{f}{\longmapsto} \{\mathsf{b}, \mathsf{c}\} \qquad \mathsf{b} \overset{f}{\longmapsto} \{\mathsf{b}\} \qquad \mathsf{c} \overset{f}{\longmapsto} \{\mathsf{a}, \mathsf{b}, \mathsf{c}\} \tag{$*$}$$

Below, the first row lists the values of the characteristic function $\mathbb{1}_{f(\mathsf{a})}$ on the inputs a, b, and c. The second row lists the values for $\mathbb{1}_{f(\mathsf{b})}$. And, the third row lists $\mathbb{1}_{f(\mathsf{c})}$.

$s \in S$	$f(s)$	$\mathbb{1}_{f(s)}(\mathsf{a})$	$\mathbb{1}_{f(s)}(\mathsf{b})$	$\mathbb{1}_{f(s)}(\mathsf{c})$
a	$\{\mathsf{b}, \mathsf{c}\}$	F	T	T
b	$\{\mathsf{b}\}$	F	T	F
c	$\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$	T	T	T

We show that f is not onto by producing a member of $\mathcal{P}(S)$ that is not any of the three sets in $(*)$. For that, take the table's diagonal FTT and flip the values to get TFF. That describes the characteristic function of the set $R = \{\mathsf{a}\}$.

Pf. First, $|S| \leq |\mathcal{P}(S)|$ because the inclusion map $\iota : S \to \mathcal{P}(S)$ given by $\iota(s) = \{s\}$ is one-to-one. For the second half we will show that there is no corespondence, that no map from a set to its power set is onto. Fix $f : S \to \mathcal{P}(S)$ and consider this element of $\mathcal{P}(S)$.

$$R = \{ s \;\big|\; s \notin f(s) \}$$

We will demonstrate that no member of the domain maps to R and thus f is not onto. Suppose that there exists $\hat{s} \in S$ such that $f(\hat{s}) = R$. Consider whether $\hat{s}$ is an element of R. We have that $\hat{s} \in R$ if and only if $\hat{s} \in \{ s \;\big|\; s \notin f(s) \}$. By the definition of membership, that holds if and only if $\hat{s} \notin f(\hat{s})$, which holds if and only if $\hat{s} \notin R$. The contradiction means that no such $\hat{s}$ exists. ∎

Diagonalization depends on changing the entries. Before we leave this argument note where the changing happens, where the bits get flipped, namely in the definition of R.

Example The collection containing all subsets of $\mathbb{N}$

$$\mathcal{C} = \{ \varnothing, \; \ldots \{1, 2, 6\}, \; \ldots \{0, 1, 4, 9, \ldots\}, \; \ldots \}$$

has a larger cardinality than does the set $\mathbb{N}$.

The functions not computed by any machine

COROLLARY The cardinality of the set $\mathbb{N}$ is strictly less than the cardinality of the set of functions $f : \mathbb{N} \to \mathbb{N}$.

PF. Let the set of functions be F. There is a one-to-one map from $\mathcal{P}(\mathbb{N})$ to F, namely the one that associates each subset $S \subseteq \mathbb{N}$ with its characteristic function, $\mathbb{1}_S : \mathbb{N} \to \mathbb{N}$. Therefore $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| \leq |F|$. ∎

The functions not computed by any machine

COROLLARY The cardinality of the set $\mathbb{N}$ is strictly less than the cardinality of the set of functions $f : \mathbb{N} \to \mathbb{N}$.

PF. Let the set of functions be F. There is a one-to-one map from $\mathcal{P}(\mathbb{N})$ to F, namely the one that associates each subset $S \subseteq \mathbb{N}$ with its characteristic function, $\mathbb{1}_S : \mathbb{N} \to \mathbb{N}$. Therefore $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| \leq |F|$. ∎

Turing machines compute functions from $\mathbb{N}$ to $\mathbb{N}$. There are uncountably many such functions, but only countably many Turing machines. Consequently, there are function that are not computed by any Turing machine.

This is like the child's game of Musical Chairs. In the game there are more children than chairs and so some child is left without a chair. Here, there are more functions from $\mathbb{N}$ to $\mathbb{N}$ than there are Turing machines and similarly there must be a function without a machine to compute it.

The functions not computed by any machine

COROLLARY The cardinality of the set $\mathbb{N}$ is strictly less than the cardinality of the set of functions $f : \mathbb{N} \to \mathbb{N}$.

PF. Let the set of functions be F. There is a one-to-one map from $\mathcal{P}(\mathbb{N})$ to F, namely the one that associates each subset $S \subseteq \mathbb{N}$ with its characteristic function, $\mathbb{1}_S : \mathbb{N} \to \mathbb{N}$. Therefore $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})| \leq |F|$. ∎

Turing machines compute functions from $\mathbb{N}$ to $\mathbb{N}$. There are uncountably many such functions, but only countably many Turing machines. Consequently, there are function that are not computed by any Turing machine.

This is like the child's game of Musical Chairs. In the game there are more children than chairs and so some child is left without a chair. Here, there are more functions from $\mathbb{N}$ to $\mathbb{N}$ than there are Turing machines and similarly there must be a function without a machine to compute it.

In the light of Church's Thesis we interpret this to say that there are jobs that no computer can do.

We started the course by asking "What can be done?" We now have part of the answer: not everything.

Universality

Universal Turing machine

THEOREM (TURING, 1936) There is a Turing machine that when given the inputs e and x will have the same output behavior as does $\mathcal{P}_e$ on input x.

This machine will fail to halt if $\mathcal{P}_e$ fails to halt on x and otherwise will halt and yield the matching output.

Universal Turing machine

THEOREM (TURING, 1936) There is a Turing machine that when given the inputs e and x will have the same output behavior as does $\mathcal{P}_e$ on input x.

This machine will fail to halt if $\mathcal{P}_e$ fails to halt on x and otherwise will halt and yield the matching output.

This machine is in a sense an all-powerful Turing machine, $\mathcal{UP}$. It is a Universal Turing Machine in that this one device can be made to have any desired computable behavior. So we don't need infinitely many different Turing machines, we just need this one.

- A universal machine $\mathcal{UP}$ is like an interpreter, or like an operating system.
- This converts from doing jobs in hardware to doing them in software.
- The universal machine does not get as input a Turing machine, it gets the index of a Turing machine, which is computationally equivalent to a representation of that machine.
- Yes, we could feed $\mathcal{UP}$ its own index.

Parametrization

Partial evaluation is the freezing of some of a machine's inputs. An example is to start with this two-input function.

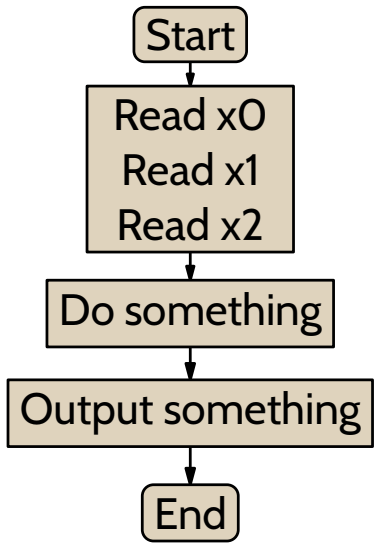
```
(define (power base exponent) ; Def is a bit silly because
  (expt base exponent))       ; expt does the same and is built in
```

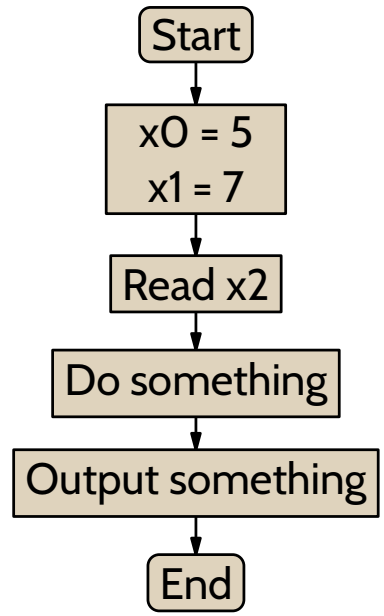
Get a family of functions by parametrizing the exponent.

```
(define (identity_fcn base)
  (power base 1))

(define (square_fcn base)
  (power base 2))

(define (cube_fcn base)
  (power base 3))
```

Freezing variables

On the left is a three input program. By Church's Thesis there is a Turing machine with this behavior; let the index of that machine be e.





On the right we have frozen the first two inputs. The *s-m-n* Theorem, which we are about to see, says that we can do this uniformly, that there is a program which takes in e, 5, and 7 and outputs the index of the Turing machine on the right.

Further, the *s-m-n* Theorem says that a single program works in all freeze-two-inputs-and-leave-one-as-is situations. We can write $s_{2,1}$ for the computable function that does this. We usually just call it s.

In short, where e is the index of the Turing machine outlined on the left, the index of the machine on the right is $s(e, 5, 7)$.

The *s-m-n* Theorem

Universality says that there is a computable function $U : \mathbb{N}^2 \to \mathbb{N}$ such that $U(e, x) = \phi_e(x)$. There, the letter e travels from the function's argument to an index. In the next result we go from the argument to being part of the index.

THEOREM (*S-M-N* THEOREM, OR PARAMETER THEOREM) For every $m, n \in \mathbb{N}$ there is a computable total function $s_{m,n} : \mathbb{N}^{1+m} \to \mathbb{N}$ such that for an $m+n$-ary function $\phi_e(x_0, \ldots x_{m-1}, x_m, \ldots x_{m+n-1})$, freezing the initial m variables at $a_0, \ldots a_{m-1} \in \mathbb{N}$ gives the n-ary computable function $\phi_{s(e,a_0,\ldots a_{m-1})}(x_m, \ldots x_{m+n-1})$.

The *s-m-n* Theorem

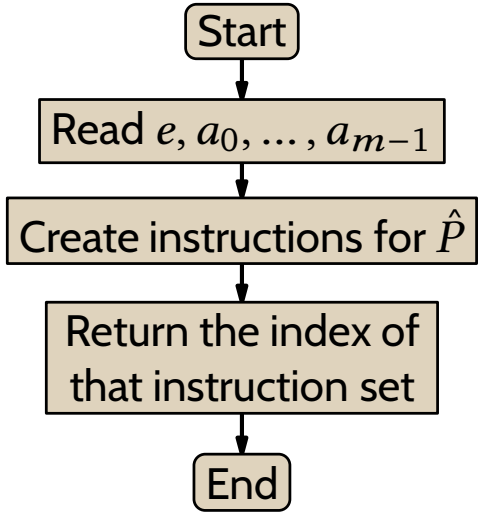
Universality says that there is a computable function $U : \mathbb{N}^2 \to \mathbb{N}$ such that $U(e, x) = \phi_e(x)$. There, the letter e travels from the function's argument to an index. In the next result we go from the argument to being part of the index.

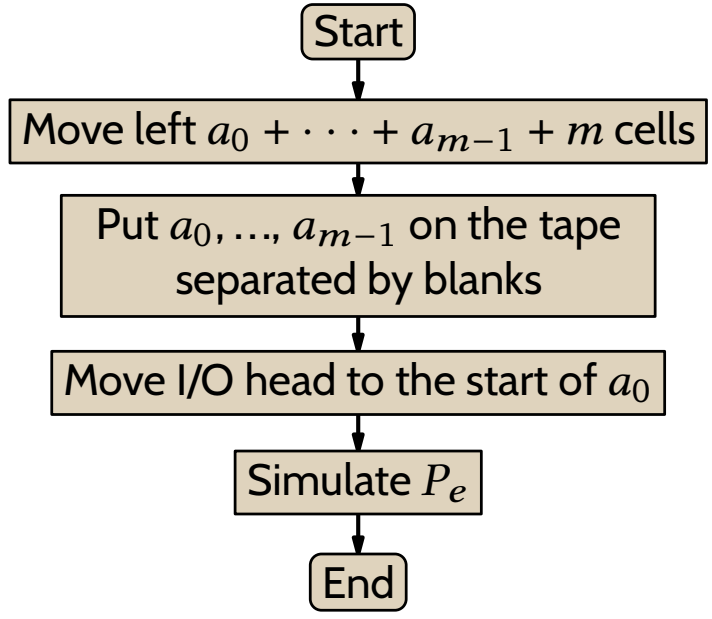
THEOREM (*S-M-N* THEOREM, OR PARAMETER THEOREM) For every $m, n \in \mathbb{N}$ there is a computable total function $s_{m,n} : \mathbb{N}^{1+m} \to \mathbb{N}$ such that for an $m+n$-ary function $\phi_e(x_0, \ldots x_{m-1}, x_m, \ldots x_{m+n-1})$, freezing the initial m variables at $a_0, \ldots a_{m-1} \in \mathbb{N}$ gives the n-ary computable function $\phi_{s(e,a_0,\ldots a_{m-1})}(x_m, \ldots x_{m+n-1})$.

PF. We will produce the function s to satisfy three requirements: it must be effective, it must input an index e and an m-tuple $a_0, \ldots a_{m-1}$, and it must output the index of a machine $\hat{\mathcal{P}}$ that, when given the input $x_m, \ldots x_{m+n-1}$, will return the value $\phi_e(a_0, \ldots a_{m-1}, x_m, \ldots x_{m+n-1})$, or fail to halt if that function diverges.

The idea is that the machine that computes s will construct the instructions for $\hat{\mathcal{P}}$. We can get from the instruction set to the index using Cantor's encoding, so with that we will be done.

Below on the left is the flowchart for the machine that computes the function s. In its third box it creates the set of four-tuple instructions shown on the right, which make the machine $\hat{\mathcal{P}}$. The machine on the left needs $a_0, \ldots a_{m-1}$ for the right side's second, third, and fourth boxes, and it needs e for the fifth box.

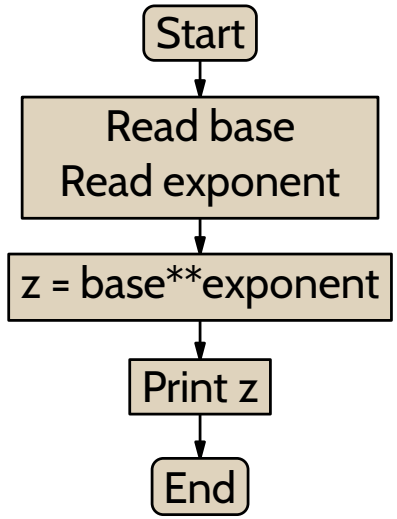




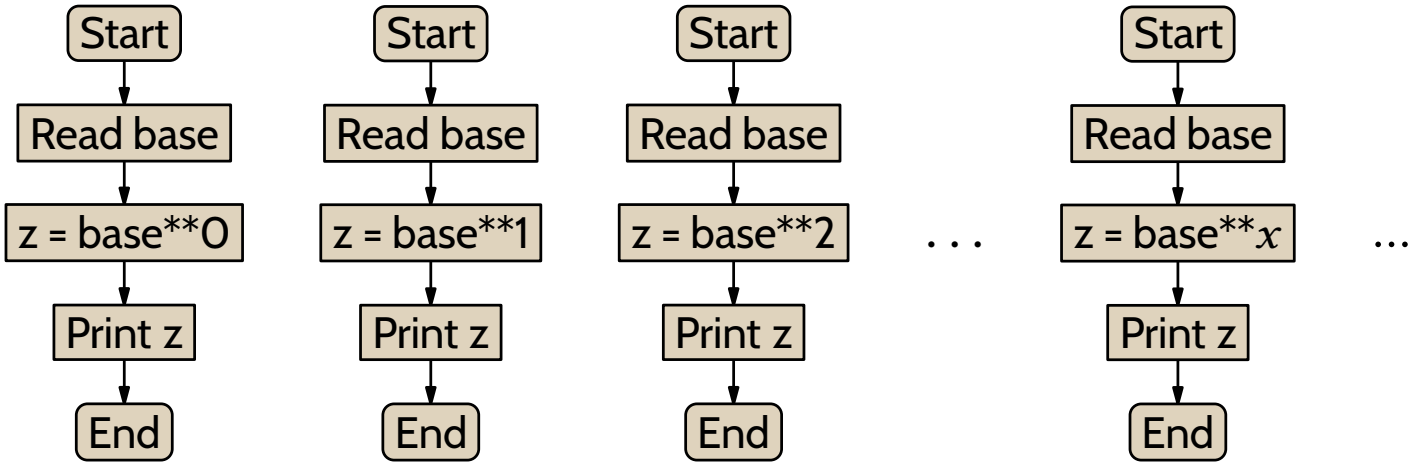
The Turing machine $\hat{\mathcal{P}}$ does not first read its inputs $x_m, \ldots x_{m+n-1}$. Instead, it first moves left and writes $a_0, \ldots a_{m-1}$ on the tape, in unary and separated by blanks, and with a blank between a_{m-1} and x_m. (Recall that the a_i are parameters, not variables. They are fixed. They are, so to speak, hard-coded into $\hat{\mathcal{P}}$, which is how it knows what to write.) Then using universality, $\hat{P}$ simulates Turing machine P_e and lets it run on the entire list of inputs now on the tape, $a_0, \ldots a_{m-1}, x_m, \ldots x_{m+n-1}$. ∎

The *s-m-n* Theorem gives a uniform family of functions

Recall again the power routine.



Suppose that sketch describes the Turing machine with index e. Looking at the different machines indexed by $s(e, x)$ gives a family of routines parametrized by the exponent.



Unsolvability

The Halting problem: motivation

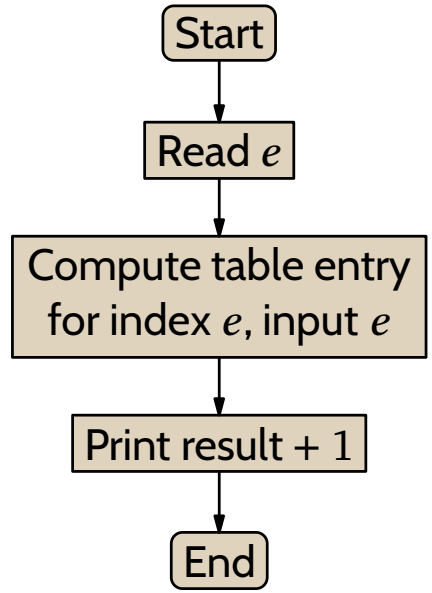
We have shown that there are some functions that are not mechanically computable. But to do this we used Cantor's Theorem that says the function exists but doesn't construct it. In this subject we prefer to construct things. So we will now go through the proof of Cantor's Theorem and effectivize it.

The Halting problem: motivation

We have shown that there are some functions that are not mechanically computable. But to do this we used Cantor's Theorem that says the function exists but doesn't construct it. In this subject we prefer to construct things. So we will now go through the proof of Cantor's Theorem and effectivize it.

On the left is the table from the discussion leading to Cantor's Theorem.

Function	*Input* 0	1	2	3	4	5	6	...
ϕ_0	3	1	2	7	7	0	4	...
ϕ_1	0	5	0	0	0	0	0	...
ϕ_2	1	4	1	5	9	2	6	...
ϕ_3	9	1	9	1	9	1	9	...
ϕ_4	1	0	1	0	0	1	0	...
ϕ_5	6	2	5	5	4	1	8	...
⋮			⋮					



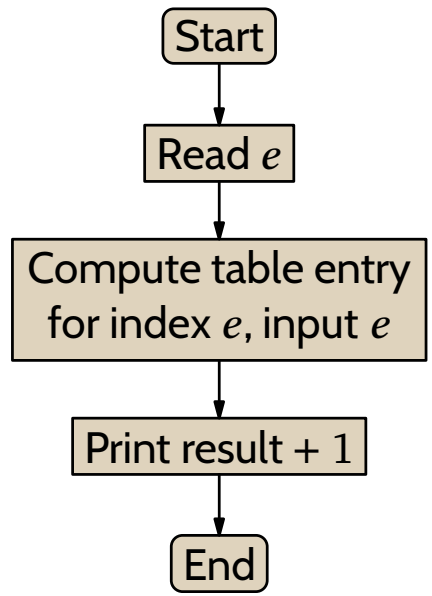
Consider the diagonalization flowchart. All the boxes except the middle one are trivial. For the middle, use universality to simulate Turing machine e running input e.

The Halting problem: motivation

We have shown that there are some functions that are not mechanically computable. But to do this we used Cantor's Theorem that says the function exists but doesn't construct it. In this subject we prefer to construct things. So we will now go through the proof of Cantor's Theorem and effectivize it.

On the left is the table from the discussion leading to Cantor's Theorem.

Function	*Input* 0	1	2	3	4	5	6	...
ϕ_0	3	1	2	7	7	0	4	...
ϕ_1	0	5	0	0	0	0	0	...
ϕ_2	1	4	1	5	9	2	6	...
ϕ_3	9	1	9	1	9	1	9	...
ϕ_4	1	0	1	0	0	1	0	...
ϕ_5	6	2	5	5	4	1	8	...
⋮			⋮					



Consider the diagonalization flowchart. All the boxes except the middle one are trivial. For the middle, use universality to simulate Turing machine e running input e.

We seem to have produced a program whose output isn't on the list of all program outputs. That's obviously impossible. Where is the flaw in the reasoning?

The Halting problem: definition

PROBLEM (Halting PROBLEM) Given $e \in \mathbb{N}$, determine whether $\phi_e(e)\downarrow$, that is, whether Turing machine $\mathcal{P}_e$ halts on input e.

DEFINITION The Halting problem set is $K = \{ e \in \mathbb{N} \mid \phi_e(e)\downarrow \}$.

The Halting problem: definition

PROBLEM (Halting PROBLEM) Given $e \in \mathbb{N}$, determine whether $\phi_e(e)\downarrow$, that is, whether Turing machine $\mathcal{P}_e$ halts on input e.

DEFINITION The Halting problem set is $K = \{ e \in \mathbb{N} \mid \phi_e(e)\downarrow \}$.

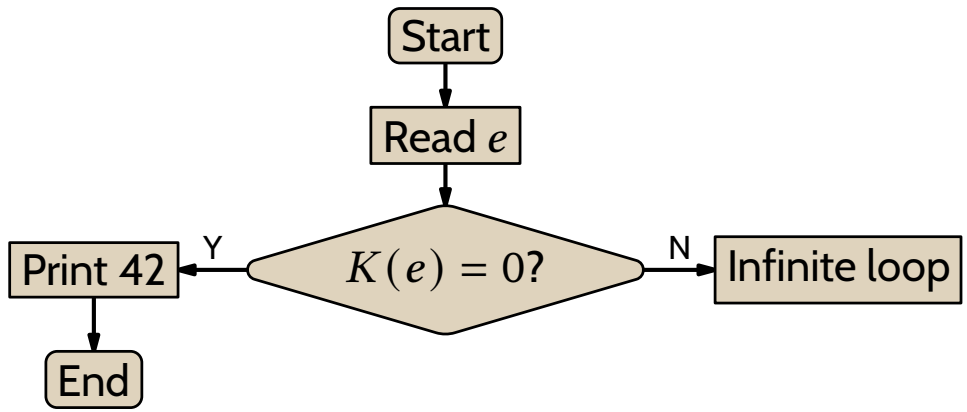
THEOREM The Halting problem is unsolvable by any Turing machine.

PF. Assume otherwise, that there exists a Turing machine with this behavior.

$$\mathbb{1}_K(e) = K(e) = \texttt{halt\_decider}(e) = \begin{cases} 1 & - \text{if } \phi_e(e)\downarrow \\ 0 & - \text{if } \phi_e(e)\uparrow \end{cases}$$

Then the function below is also mechanically computable. The flowchart illustrates how f is constructed; it uses the above function in its decision box. (In the top case the particular output value 42 doesn't matter, all that matters is that f converges.)

$$f(e) = \begin{cases} 42 & - \text{if } \phi_e(e)\uparrow \\ \uparrow & - \text{if } \phi_e(e)\downarrow \end{cases}$$



Since this is mechanically computable, it has a Turing machine index. Let that index be e_0, so that $f(x) = \phi_{e_0}(x)$ for all inputs x.

Now consider $f(e_0) = \phi_{e_0}(e_0)$ (that is, feed the machine $\mathcal{P}_{e_0}$ its own index). If it diverges then the first clause in the definition of f means that $f(e_0)\downarrow$, which contradicts the assumption of divergence. If it converges then f's second clause means that $f(e_0)\uparrow$, also a contradiction. Since assuming that `halt_decider` is mechanically computable leads to a contradiction, that function is not mechanically computable. ∎

Showing other problems are unsolvable by relating them to the Halting problem

EXAMPLE Consider this problem: given $i \in \mathbb{N}$, determine whether Turing machine i halts on input 1. We are asking whether this function is computable.

$$\texttt{halts\_on\_one\_decider}(i) = \begin{cases} 1 & \text{– if } \phi_i(1)\downarrow \\ 0 & \text{– otherwise} \end{cases}$$

We will show that if it were mechanically computable then we could solve the Halting problem.

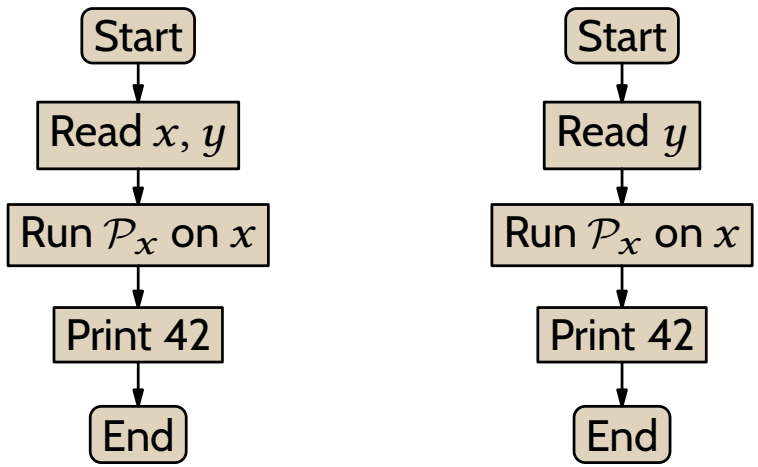
Showing other problems are unsolvable by relating them to the Halting problem

EXAMPLE Consider this problem: given $i \in \mathbb{N}$, determine whether Turing machine i halts on input 1. We are asking whether this function is computable.

$$\texttt{halts\_on\_one\_decider}(i) = \begin{cases} 1 & \text{– if } \phi_i(1)\downarrow \\ 0 & \text{– otherwise} \end{cases}$$

We will show that if it were mechanically computable then we could solve the Halting problem.

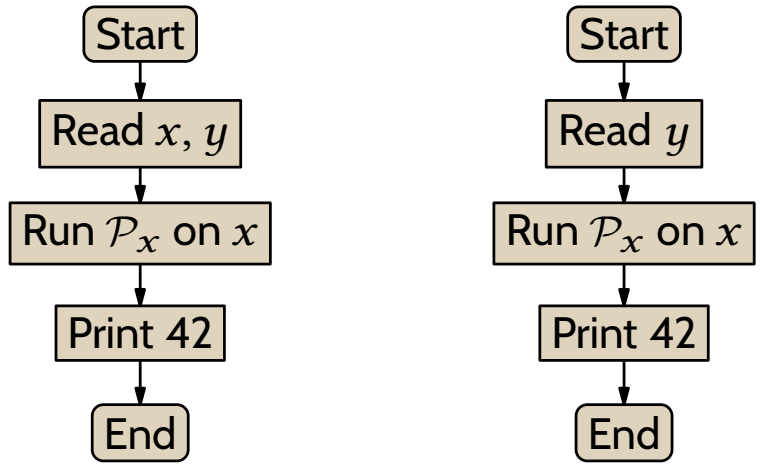
Consider the machines sketched here.



On the right, x is hard-coded. That machine halts on any input if and only if it gets through the middle box (the output 42 is not important; we just like to have both input and output). In particular, it halts on input $y = 1$ if and only if $\phi_x(x)\downarrow$.

So we can turn the 'does $\phi_x(x)\downarrow$?' questions into questions that `halts_on_one_decider` can answer. The proof involves recognizing that we can switch uniformly to questions of the second kind.

For that proof, here are the flowcharts again. On the left is a sketch of a single machine. We can write it as a program, using universality for the third box, so by Church's Thesis there is a Turing machine with this behavior. Let that machine's index be e_0.



Then the *s-m-n* Theorem gives what's on the right, the family of machines, $\mathcal{P}_{s(e_0,x)}$. We have already observed that $\phi_x(x)\downarrow$ if and only if `halts_on_one_decider`$(s(e_0, x)) = 1$. So if we could mechanically check whether a machine halts on 1 then we could mechanically solve the Halting problem. We can't solve the Halting problem, so `halts_on_one_decider` is not a mechanically computable function.

Similar examples

Each of these is unsolvable. For each, show that by using the Halting problem.

1. The problem of determining whether a given Turing machine ever outputs a 19.

$$\texttt{outputs\_nineteen\_decider}(i) = \begin{cases} 1 & \text{– if there is } n \text{ such that } \phi_i(n) = 19 \\ 0 & \text{– otherwise} \end{cases}$$

2. The problem of determining whether a given Turing machine gives as output the square of its input.

$$\texttt{square\_decider}(i) = \begin{cases} 1 & \text{– if } \phi_i(n) = n^2 \text{ for all n} \\ 0 & \text{– otherwise} \end{cases}$$

Discussion: the Halting problem is unsolvable

- Inherent in the nature of mechanical computation, necessary to avoid contradiction, is that some computations fail to halt.

Discussion: the Halting problem is unsolvable

- Inherent in the nature of mechanical computation, necessary to avoid contradiction, is that some computations fail to halt.
- "Unsolvable" means unsolvable by a Turing machine. Below is a perfectly good function that solves the Halting problem but it isn't mechanically computable — no Turing machine has this input/output behavior.

$$\texttt{halt\_decider}(e) = \begin{cases} 1 & \text{– if } \phi_e(e)\downarrow \\ 0 & \text{– if } \phi_e(e)\uparrow \end{cases}$$

Discussion: the Halting problem is unsolvable

- Inherent in the nature of mechanical computation, necessary to avoid contradiction, is that some computations fail to halt.
- "Unsolvable" means unsolvable by a Turing machine. Below is a perfectly good function that solves the Halting problem but it isn't mechanically computable — no Turing machine has this input/output behavior.

$$\texttt{halt\_decider}(e) = \begin{cases} 1 & \text{– if } \phi_e(e)\downarrow \\ 0 & \text{– if } \phi_e(e)\uparrow \end{cases}$$

- Unsolvability of the Halting problem does not mean that for no program can we tell if that program halts. The program on the left below halts, for all inputs. Nor does it mean that for no program can we tell if the program does not halt. The program on the right does not halt, for all inputs.

```
(define (print-zero)
  (read)
  (display 0))
```

```
(define (unbounded-growth)
  (define (grow x)
    (grow (+ 1 x)))

  (read)  ; wait for user input
  (grow 0))
```

Instead, the unsolvability of the Halting Problem says that there is no single program that, for all input e, correctly computes in a finite time whether $\mathcal{P}_e$ halts on e.

Discussion continued

- Unsolvability of the Halting Problem says that there is no single program that, for all e, correctly computes in a finite time whether $\mathcal{P}_e$ halts on input e. The "finite time" qualifier is there because we could think to use a universal Turing machine to simulate $\mathcal{P}_e$ on e, but if that machine failed to halt then we would not find out in a finite time.
  The "single program" qualifier is a little subtler. For any index e, either $\mathcal{P}_e$ halts on e or it does not. That is, for any e one of these two programs gives the right answer.

```
(define (print-no)
  (display 0))
```

```
(define (print-yes)
  (display 1))
```

  Thus, the unsolvability of the Halting Problem is about the non-existence of a single program that works across all indices. It is about uniformity, or rather, the impossibility of uniformity.

Rice's Theorem

Rice's Theorem

DEFINITION Two computable functions have the same behavior, $\phi_e \simeq \phi_{\hat{e}}$, if they converge on the same inputs $x \in \mathbb{N}$ and when they do converge, they have the same outputs.

DEFINITION A set $\mathcal{I}$ of natural numbers is an index set when for all $e, \hat{e} \in \mathbb{N}$, if $e \in \mathcal{I}$ and $\phi_e \simeq \phi_{\hat{e}}$ then also $\hat{e} \in \mathcal{I}$.

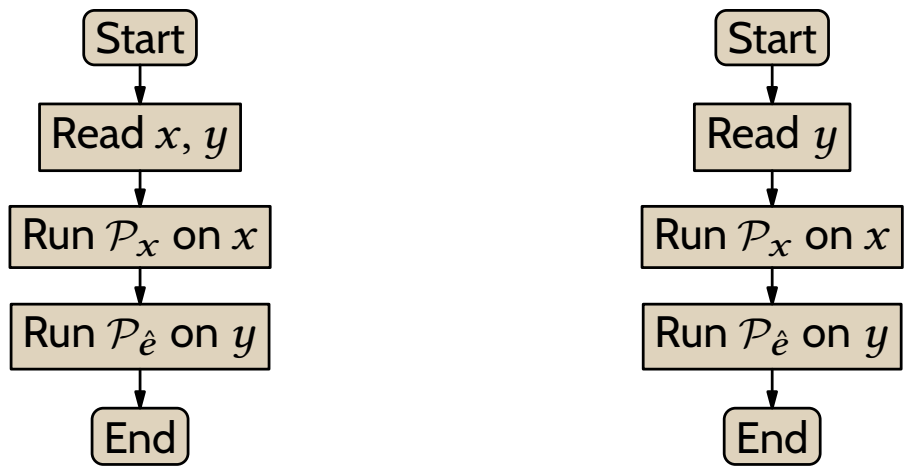
EXAMPLE These are index sets: $\{e \in \mathbb{N} \mid \phi_e(x) = 9\}$, and $\{e \in \mathbb{N} \mid \phi_e(x) = 2x \text{ or } \phi_e(x) = x + 6\}$, and (3) $\{e \in \mathbb{N} \mid \phi_e(1)\downarrow\}$.

THEOREM (RICE'S THEOREM) Every index set that is not trivial, that is not empty and not all of $\mathbb{N}$, is not computable.

Pf. Let $\mathcal{I}$ be an index set. Choose an $e \in \mathbb{N}$ so that $\phi_e(y)\uparrow$ for all y. Then either $e \in \mathcal{I}$ or $e \notin \mathcal{I}$. We shall show that in the second case $\mathcal{I}$ is not computable. The first case is similar and is Exercise 6.33.

So assume $e \notin \mathcal{I}$. Since $\mathcal{I}$ is not empty there is an index $\hat{e} \in \mathcal{I}$. Because $\mathcal{I}$ is an index set, $\phi_e \neq \phi_{\hat{e}}$. Thus there is an input y such that $\phi_{\hat{e}}(y)\downarrow$.

Consider the flowchart on the left below. Note that $\hat{e}$ is not an input, it is hard-coded into the source. By Church's Thesis there is a Turing machine with that behavior, let it be $\mathcal{P}_{e_0}$. Apply the *s-m-n* theorem to parametrize x, resulting in the uniformly computable family of functions $\phi_{s(e_0,x)}$ whose computation is outlined on the right.



We've constructed the machine on the right so that if $\phi_x(x)\uparrow$ then $\phi_{s(e_0,x)} \simeq \phi_e$ and thus $s(e_0, x) \notin \mathcal{I}$. Further, if $\phi_x(x)\downarrow$ then $\phi_{s(e_0,x)} \simeq \phi_{\hat{e}}$ and thus $s(e_0, x) \in \mathcal{I}$. It follows that if $\mathcal{I}$ were mechanically computable, so that we could effectively check whether $s(e_0, x) \in \mathcal{I}$, then we could solve the Halting problem. ∎

Similar examples

Show each of these is unsolvable using Rice's Theorem.

1. The problem of determining whether a given Turing machine halts on input 1.

$$\mathcal{I}_0 = \{ i \mid \mathcal{P}_i \text{ halts on input } 1 \}$$

2. The problem of determining whether a given Turing machine ever outputs a 19.

$$\mathcal{I}_1 = \{ i \mid \text{there is an } x \text{ so that } \mathcal{P}_i \text{ halts on } x \text{ and outputs } 19 \}$$

3. The problem of determining whether a given Turing machine gives as output the square of its input.

Computably enumerable sets

Recall also that a set of natural numbers is computable, or decidable, if its characteristic function is computable.

DEFINITION A set of natural numbers is computably enumerable if it is effectively listable, that is, if it is the range of a total computable function, or is the empty set. Alternate terms are recursively enumerable (or c.e., or r.e.), or semicomputable or semidecidable.

The 'enumerable' comes from the image of a computable function generating the set elements as a stream of numbers.

$$\phi_e(0),\ \phi_e(1),\ \phi_e(2),\ \ldots$$

EXAMPLE Some computably enumerable sets are $\{1, 3, 5\}$, and $\{2n \mid n \in \mathbb{N}\}$, and $\{y \mid \text{there is an } x \text{ so that } y = \phi_{19}(x)\}$, and $\{x \mid \phi_{10}(x)\!\downarrow\}$.

The contrast between computable and computably enumerable is that a set S is computable if there is a Turing machine that decides membership, that inputs a number x and decides either 'yes' or 'no' whether $x \in S$. But with computably enumerable, given some x we can set up a machine to monitor the number stream and if x appear then this machine decides 'yes'. However this machine need not ever discover 'no'. Restated, a set is computably enumerable if there is a Turing machine that recognizes members, while a set is computable if there is a Turing machine that recognizes both members and nonmembers.

LEMMA The following are equivalent for a set of natural numbers. (1) It is computably enumerable, that is, either it is empty or it is the range of a total computable function. (2) It is the domain of a partial computable function. (3) It is the range of a partial computable function.

DEFINITION $W_e = \{x \mid \phi_e(x)\downarrow\}$

LEMMA The following are equivalent for a set of natural numbers. (1) It is computably enumerable, that is, either it is empty or it is the range of a total computable function. (2) It is the domain of a partial computable function. (3) It is the range of a partial computable function.

DEFINITION $W_e = \{x \mid \phi_e(x)\downarrow\}$

LEMMA If a set is computable then it is computably enumerable. A set is computable if and only if both it and its complement are computably enumerable.

COROLLARY The Halting problem set K is computably enumerable. Its complement K^{c} is not.

LEMMA The following are equivalent for a set of natural numbers. (1) It is computably enumerable, that is, either it is empty or it is the range of a total computable function. (2) It is the domain of a partial computable function. (3) It is the range of a partial computable function.

DEFINITION $W_e = \{x \mid \phi_e(x)\downarrow\}$

LEMMA If a set is computable then it is computably enumerable. A set is computable if and only if both it and its complement are computably enumerable.

COROLLARY The Halting problem set K is computably enumerable. Its complement K^{c} is not.

Part of our interest in these sets is philosophical. By Church's Thesis we can think that, in a sense, the collection of computable sets consists of the only sets that we will ever fully know. With that thinking, sets that are semidecidable but not decidable are at the limits of our knowledge, where we can determine membership but not nonmembership.

Fixed point theorem

Sequences of computable functions

So far we have studied individual computable functions ϕ. We next consider sequences of them.

$$\phi_{i_0},\ \phi_{i_1},\ \phi_{i_2},\ \phi_{i_3},\ \ldots$$

There, the indices seem to be any old sequence of numbers. But in this subject it is most natural for this sequence to be effectively computed, for the sequence to be the output of some computable function, $i_0 = \phi_e(0), i_1 = \phi_e(1), \ldots$ That gives us this sequence, for some e.

$$\phi_{\phi_e(0)},\ \phi_{\phi_e(1)},\ \phi_{\phi_e(2)},\ \phi_{\phi_e(3)},\ \ldots$$

To compute $\phi_{\phi_e(i)}(x)$, first compute $\phi_e(i) = w$ and then use that value w to compute $\phi_w(x)$. Thus, if $\phi_e(i)$ diverges then $\phi_{\phi_e(i)}(x)$ also diverges.

EXAMPLE The squaring function $\phi_{e_0}(x) = x^2$ gives this sequence of computable functions

$$\phi_0,\ \phi_1,\ \phi_4,\ \phi_9,\ \ldots$$

and the doubler function $\phi_{e_1}(x) = 2x$ gives this sequence.

$$\phi_0,\ \phi_2,\ \phi_4,\ \phi_6,\ \ldots$$

There are countably many computable functions ϕ_e so this table that lists every computable sequence $\langle \phi_{\phi_e(n)} \mid n \in \mathbb{N} \rangle$ of computable functions.

		Sequence term				
		$n = 0$	$n = 1$	$n = 2$	$n = 3$	$\dots$
Sequence	$e = 0$	$\phi_{\phi_0(0)}$	$\phi_{\phi_0(1)}$	$\phi_{\phi_0(2)}$	$\phi_{\phi_0(3)}$	$\cdots$
	$e = 1$	$\phi_{\phi_1(0)}$	$\phi_{\phi_1(1)}$	$\phi_{\phi_1(2)}$	$\phi_{\phi_1(3)}$	$\cdots$
	$e = 2$	$\phi_{\phi_2(0)}$	$\phi_{\phi_2(1)}$	$\phi_{\phi_2(2)}$	$\phi_{\phi_2(3)}$	$\cdots$
	$e = 3$	$\phi_{\phi_3(0)}$	$\phi_{\phi_3(1)}$	$\phi_{\phi_3(2)}$	$\phi_{\phi_3(3)}$	
	$\vdots$	$\vdots$	$\vdots$			

What happens when we diagonalize?

When diagonalization fails

Recall our first example of diagonalization, the proof that the set of real numbers is not countable. We assumed that there is an $f : \mathbb{N} \to \mathbb{R}$ and considered its inputs and outputs, as illustrated in this table.

n	$f(n)$'s decimal expansion							
0	42	. 3	1	2	7	7	0	4 ...
1	2	. 0	1	0	0	0	0	0 ...
2	1	. 4	1	4	1	5	9	2 ...
3	−20	. 9	1	9	5	9	1	9 ...
⋮		⋮						

Let row n's decimal representation be $d_n = \hat{d}.d_{n,0}\,d_{n,1}\,d_{n,2}\,...$ Go down the diagonal to the right of the decimal point to get the sequence of digits $\langle d_{0,0}, d_{1,1}, d_{2,2}, ...\rangle$, which above is $\langle 3, 1, 4, 5, ...\rangle$. Using that sequence, construct a number $z = 0.z_0\,z_1\,z_2\,...$ by making its n-th decimal place be something other than $d_{n,n}$. In our earlier example we took a digit transformation t given by $t(d_{n,n}) = 2$ if $d_{n,n} = 1$, and $t(d_{n,n}) = 1$ otherwise, so that the table above yields $z = 0.1211\,...$ Then the diagonalization argument culminates in verifying that z is not any of the rows.

When diagonalization fails

Recall our first example of diagonalization, the proof that the set of real numbers is not countable. We assumed that there is an $f : \mathbb{N} \to \mathbb{R}$ and considered its inputs and outputs, as illustrated in this table.

n	$f(n)$'s decimal expansion							
0	42	. 3	1	2	7	7	0	4 ...
1	2	. 0	1	0	0	0	0	0 ...
2	1	. 4	1	4	1	5	9	2 ...
3	−20	. 9	1	9	5	9	1	9 ...
⋮		⋮						

Let row n's decimal representation be $d_n = \hat{d}.d_{n,0}\,d_{n,1}\,d_{n,2}\,...$ Go down the diagonal to the right of the decimal point to get the sequence of digits $\langle d_{0,0}, d_{1,1}, d_{2,2}, ...\rangle$, which above is $\langle 3, 1, 4, 5, ...\rangle$. Using that sequence, construct a number $z = 0.z_0\,z_1\,z_2\,...$ by making its n-th decimal place be something other than $d_{n,n}$. In our earlier example we took a digit transformation t given by $t(d_{n,n}) = 2$ if $d_{n,n} = 1$, and $t(d_{n,n}) = 1$ otherwise, so that the table above yields $z = 0.1211\,...$ Then the diagonalization argument culminates in verifying that z is not any of the rows.

What if the transformation is such that the diagonal is a row, $z = f(n_0)$? Then find where the diagonal crosses that row. That array member is unchanged by the transformation, $d_{n_0,n_0} = t(d_{n_0,n_0})$. Conclusion: if diagonalization fails then the transformation has a fixed point.

We can apply that insight to the table of all computable sequences of computable functions.

		Sequence term				
		$n = 0$	$n = 1$	$n = 2$	$n = 3$	$\ldots$
Sequence	$e = 0$	$\phi_{\phi_0(0)}$	$\phi_{\phi_0(1)}$	$\phi_{\phi_0(2)}$	$\phi_{\phi_0(3)}$	$\cdots$
	$e = 1$	$\phi_{\phi_1(0)}$	$\phi_{\phi_1(1)}$	$\phi_{\phi_1(2)}$	$\phi_{\phi_1(3)}$	$\cdots$
	$e = 2$	$\phi_{\phi_2(0)}$	$\phi_{\phi_2(1)}$	$\phi_{\phi_2(2)}$	$\phi_{\phi_2(3)}$	$\cdots$
	$e = 3$	$\phi_{\phi_3(0)}$	$\phi_{\phi_3(1)}$	$\phi_{\phi_3(2)}$	$\phi_{\phi_3(3)}$	
	$\vdots$	$\vdots$	$\vdots$			

The natural transformation is this, where f is a computable function.

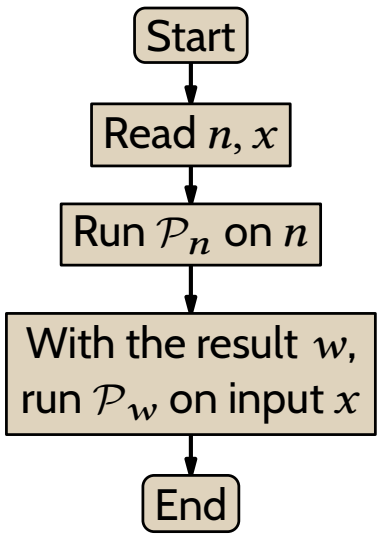
$$\phi_x \overset{t_f}{\longmapsto} \phi_{f(x)}$$

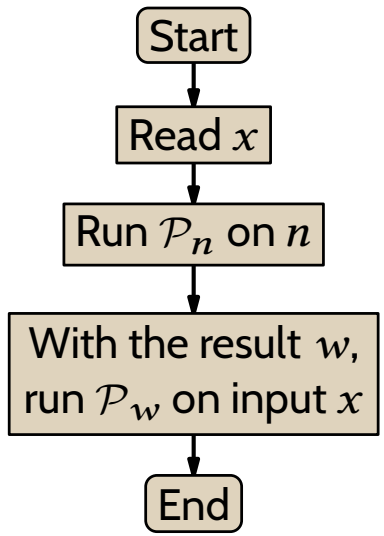
We next argue that under this transformation, diagonalization fails. Thus, the transformation t_f has a fixed point.

THEOREM For any total computable function f there is a number k such that $\phi_k = \phi_{f(k)}$.

THEOREM For any total computable function f there is a number k such that $\phi_k = \phi_{f(k)}$.

PF. The array diagonal is $\phi_{\phi_0(0)}$, $\phi_{\phi_1(1)}$, $\phi_{\phi_2(2)}$ ... The flowchart on the left below is a sketch of a function $f(n,x) = \phi_{\phi_n(n)}(x)$. Church's Thesis says that some Turing machine computes this function; let that machine have index e_0. Apply the *s-m-n* theorem to parametrize n, giving the right chart, which describes the family of machines. The n-th member of that family, $\phi_{s(e_0,n)}$, computes the n-th function on the diagonal of the above array.



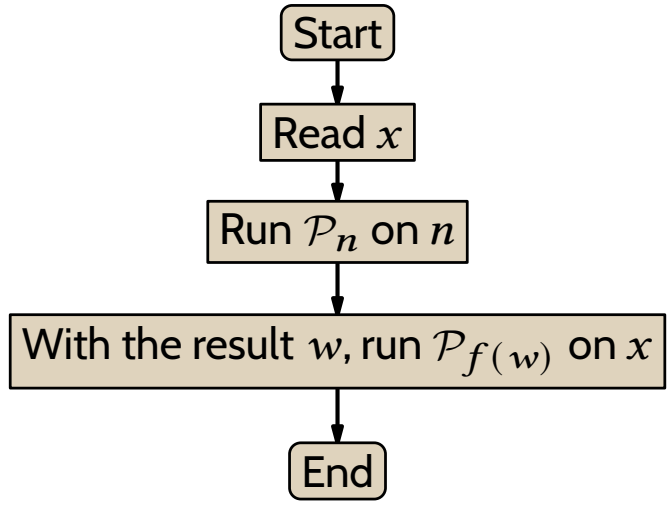


$$d_n(x) = \begin{cases} \phi_{\phi_n(n)}(x) & \text{– if } \phi_n(n)\downarrow \\ \uparrow & \text{– otherwise} \end{cases}$$

The index e_0 is fixed, so $s(e_0, n)$ is a function of one variable. Let $g(n) = s(e_0, n)$, so that the diagonal functions are $\phi_{g(n)}$. This function g is computable and total.

Under t_f those functions are transformed to $\phi_{fg(0)}$, $\phi_{fg(1)}$, $\phi_{fg(2)}$, ... The composition $f \circ g$ is computable and total since f is specified as total.

$$t_f(d_n)\ (x) = \begin{cases} \phi_{f\phi_n(n)}(x) & \text{– if } \phi_n(n)\downarrow \\ \uparrow & \text{– otherwise} \end{cases}$$



As the flowchart underlines, $\phi_{fg(0)}$, $\phi_{fg(1)}$, $\phi_{fg(2)}$, ... is a computable sequence of computable functions. Hence it is one of the table's rows. Let it be row v, making $\phi_{fg(m)} = \phi_{\phi_v(m)}$ for all m. Consider where the diagonal sequence $\phi_{g(n)}$ intersects that row: $\phi_{g(v)} = \phi_{\phi_v(v)} = \phi_{fg(v)}$. The fixed point for f is $k = g(v)$. ∎

COROLLARY There is an index e so that $\phi_e = \phi_{e+1}$.

PF. The function $f(x) = x + 1$ is computable and total. So there is an $e \in \mathbb{N}$ such that $\phi_e = \phi_{f(e)}$. ∎

This does not say that $\mathcal{P}_e = \mathcal{P}_{e+1}$. It says that the two Turing machines have the same input/output behavior, that $\phi_e = \phi_{e+1}$.

COROLLARY There is an index e so that $\phi_e = \phi_{e+1}$.

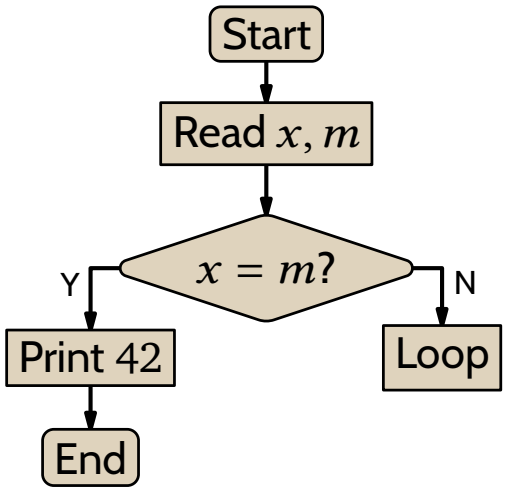
PF. The function $f(x) = x + 1$ is computable and total. So there is an $e \in \mathbb{N}$ such that $\phi_e = \phi_{f(e)}$. ∎

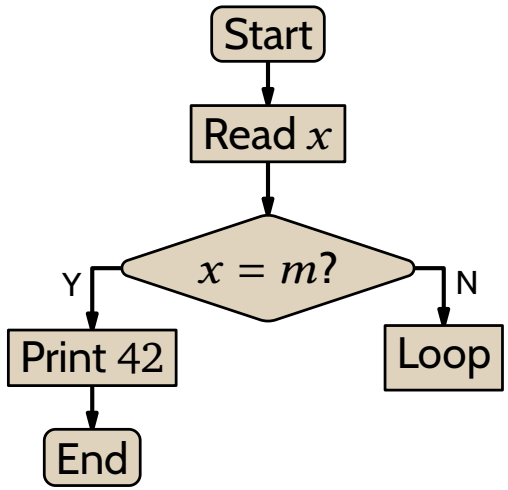
This does not say that $\mathcal{P}_e = \mathcal{P}_{e+1}$. It says that the two Turing machines have the same input/output behavior, that $\phi_e = \phi_{e+1}$.

The Fixed Point theorem is very strange. Our numbering of Turing machines does not seem to have anything to do with the behavior of the machines. Nonetheless, under the mild assumption that our numbering is acceptable there must be adjacent machines with the same behavior.

COROLLARY There is an index e such that $\mathcal{P}_e$ halts only on e.

PF. Consider the program described by the flowchart on the left. By Church's Thesis it can be done with a Turing machine, $\mathcal{P}_{e_0}$. Parametrize to get the program on the right, $\mathcal{P}_{s(e_0,m)}$.



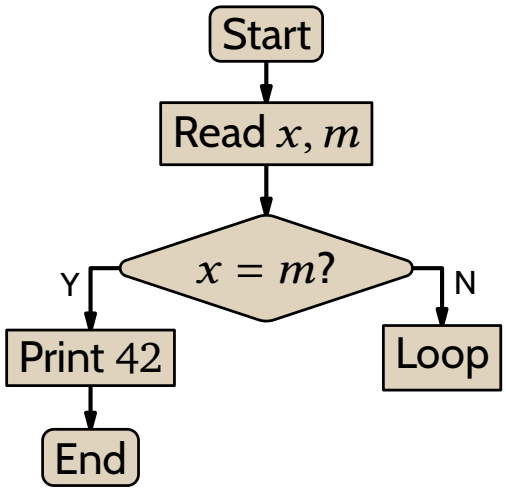


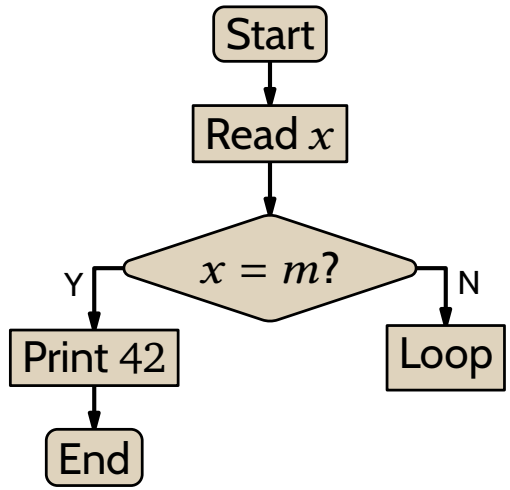
$$\phi_{s(e_o,m)}(x) = \begin{cases} 42 & \text{– if } x = m \\ \uparrow & \text{– otherwise} \end{cases}$$

Since e_0 is fixed (it is the index of the machine on the left), $s(e_0, x)$ is a total computable function of one variable, $f(m) = s(e_0, m)$, where the associated Turing machine halts only on input m. That function has a fixed point, $\phi_{f(e)} = \phi_e$, and the associated Turing machine $\mathcal{P}_e$ halts only on e. ∎

COROLLARY There is an index e such that $\mathcal{P}_e$ halts only on e.

PF. Consider the program described by the flowchart on the left. By Church's Thesis it can be done with a Turing machine, $\mathcal{P}_{e_0}$. Parametrize to get the program on the right, $\mathcal{P}_{s(e_0,m)}$.





$$\phi_{s(e_o,m)}(x) = \begin{cases} 42 & \text{– if } x = m \\ \uparrow & \text{– otherwise} \end{cases}$$

Since e_0 is fixed (it is the index of the machine on the left), $s(e_0, x)$ is a total computable function of one variable, $f(m) = s(e_0, m)$, where the associated Turing machine halts only on input m. That function has a fixed point, $\phi_{f(e)} = \phi_e$, and the associated Turing machine $\mathcal{P}_e$ halts only on e. ∎

Again, this interaction between numbering and behavior is very strange. It is also quite interesting. Think of the index number as the computable function's name, so that this is an interaction between its name and how it acts. In particular, the machine $\mathcal{P}_e$ of this example halts only on its name, e.