Think globally, act locally.
— Attributed to Patrick Geddes (c.1915), among many others.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.
— Alan Turing, "Computing Machinery and Intelligence" (1950)

Never worry about theory
as long as the machinery does what it's supposed to do.
— Robert Anson Heinlein, *Waldo & Magic, Inc.* (1950)

It is a sobering thought that when Mozart was my age,
he had been dead for two years.
— Tom Lehrer, introduction to "Alma", *That Was the Year That Was* (1965)

6 Turing Machines

In 1936, a few months before his 24th birthday, Alan Turing launched computer science as a modern intellectual discipline. In a single remarkable paper, Turing provided the following results:

- A simple formal model of mechanical computation now known as *Turing machines*.
- A description of a single *universal* machine that can be used to compute *any* function computable by *any* other Turing machine.
- A proof that no Turing machine can solve the *halting problem*—Given the formal description of an arbitrary Turing machine M, does M halt or run forever?
- A proof that no Turing machine can determine whether an arbitrary given proposition is provable from the axioms of first-order logic. This is Hilbert and Ackermann's famous *Entscheidungsproblem* ("decision problem").
- Compelling arguments[1] that his machines can execute *arbitrary* "calculation by finite means".

Although Turing did not know it at the time, he was not the first to prove that the *Entscheidungsproblem* had no algorithmic solution. The first such proof is implicit in the work of Kurt Gödel; in lectures on his incompleteness theorems[2] at Princeton in 1934, Gödel described a model of ***general recursive functions***, which he largely credited to Jacques Herbrand.[3] Gödel's incompleteness theorem can be viewed as a proof that some propositions cannot be proved using general recursive functions. However, Gödel viewed his definition as a "heuristic principle" rather than an accurate model of effective computation, or even a complete definition of recursion.

The first *published* proof was written by Alonzo Church and published just a new months before Turing's paper, using another different model of computation, now called the ***untyped λ-calculus***. Turing and Church developed their results independently; indeed, Turing rushed

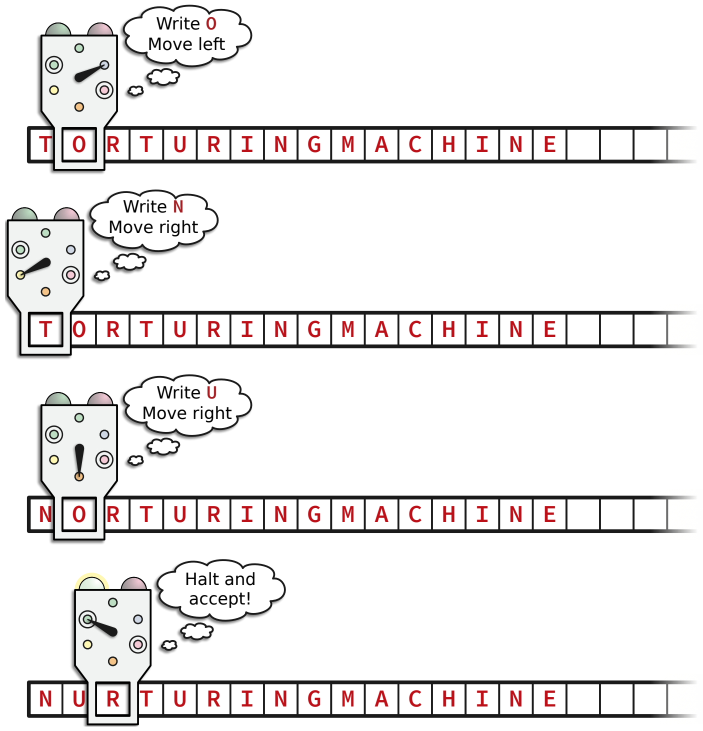
[1]As Turing put it, "All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically." The claim that anything that can be computed can be computing using Turing machines is now known as the ***Church-Turing thesis***.

[2]which he published at the ripe old age of 25

[3]Herbrand was a brilliant French mathematician who was killed in a mountain-climbing accident at the age of 23.

the submission of his own paper immediately after receiving a copy of Church's paper, pausing only long enough to prove that any function computable via λ-calculus can also be computed by a Turing machine and vice versa.[4] Church was the referee for Turing's paper; between the paper's submission and its acceptance, Turing was admitted to Princeton as a PhD student, where Church became his advisor. He finished his PhD two years later.

Informally, Turing described an abstract machine with a finite number of ***internal states*** that has access to "memory" in the form of a ***tape***. The tape consists of a semi-infinite sequence of ***cells***, each containing a single symbol from some arbitrary finite alphabet. The Turing machine can access the tape only through its ***head***, which is positioned over a single cell. Initially, the tape contains an arbitrary finite ***input string*** followed by an infinite sequence of ***blanks***, and the head is positioned over the first cell on the tape. In a single iteration, the machine reads the symbol in that cell, possibly write a new symbol into that cell, possibly changes its internal state, possibly moves the head to a neighboring cell, and possibly halts. The precise behavior of the machine at each iteration is entirely determined by its internal state and the symbol that it reads. When the machine halts, it indicates whether it has ***accepted*** or ***rejected*** the original input string.


Write O
Move left
T O R T U R I N G M A C H I N E
Write N
Move right
T O R T U R I N G M A C H I N E
Write U
Move right
N O R T U R I N G M A C H I N E
Halt and accept!
N U R T U R I N G M A C H I N E

A few iterations of a six-state Turing machine.

6.1 Why Bother?

Students used to thinking of computation in terms of higher-level operations like random memory accesses, function calls, and recursion may wonder why we should even consider a model as simple and constrained as Turing machines. Admittedly, Turing machines are a terrible model for thinking about ***fast*** computation; simple operations that take constant time in the standard

[4]At roughly the same time as Turing's paper, Church and his more recent PhD graduate Steven Kleene independently proved that all general recursive functions are definable in the λ-calculus and vice versa. At the time, Kleene was 27, and Church was 33.

random-access model can require *arbitrarily* many steps on a Turing machine. Worse, seemingly minor variations in the precise definition of "Turing machine" can have significant impact on problem complexity. As a simple example (which will make more sense later), we can reverse a string of n bits in $O(n)$ time using a two-tape Turing machine, but the same task provably requires $\Omega(n^2)$ time on a single-tape machine.

But here we are not interested in finding *fast* algorithms, or indeed in finding algorithms at all, but rather in proving that some problems cannot be solved by *any* computational means. Such a bold claim requires a formal definition of "computation" that is simple enough to support formal argument, but still powerful enough to describe arbitrary algorithms. Turing machines are ideal for this purpose. In particular, Turing machines are powerful enough to *simulate other Turing machines*, while still simple enough to let us build up this self-simulation from scratch, unlike more complex but efficient models like the standard random-access machine

(Arguably, self-simulation is even simpler in Church's λ-calculus, or in Schönfinkel and Curry's combinator calculus, which is one of many reasons those models are more common in the design and analysis of programming languages than Turing machines. Those models are much more abstract; in particular, they are harder to show equivalent to standard iterative models of computation.)

6.2 Formal Definitions

Formally, a Turing machine consists of the following components. (Hang on; it's a long list.)

- An arbitrary finite set Γ with at least two elements, called the ***tape alphabet***.
- An arbitrary symbol $\square \in \Gamma$, called the ***blank symbol*** or just the ***blank***.
- An arbitrary nonempty subset $\Sigma \subseteq (\Gamma \setminus \{\square\})$, called the ***input alphabet***.
- Another arbitrary finite set Q whose elements are called ***states***.
- Three distinct special states $\mathsf{start}, \mathsf{accept}, \mathsf{reject} \in Q$.
- A ***transition*** function $\delta\colon (Q \setminus \{\mathsf{accept}, \mathsf{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

A ***configuration*** or ***global state*** of a Turing machine is represented by a triple $(q, x, i) \in Q \times \Gamma^* \times \mathbb{N}$, indicating that the machine's internal state is q, the tape contains the string x followed by an infinite sequence of blanks, and the head is located at position i. Trailing blanks in the tape string are ignored; the triples (q, x, i) and $(q, x\square, i)$ describe exactly the same configuration.

The transition function δ describes the evolution of the machine. For example, $\delta(q, a) = (p, b, -1)$ means that when the machine reads symbol a in state q, it changes its internal state to p, writes symbol b onto the tape at its current location (replacing a), and then decreases its position by 1 (or more intuitively, moves one step to the left). If the position of the head becomes negative, no further transitions are possible, and the machine ***crashes***.

We write $\boldsymbol{(p, x, i) \Rightarrow_M (q, y, j)}$ to indicate that Turing machine M transitions from the first configuration to the second in one step. (The symbol $\Rightarrow$ is often pronounced "yields"; I will omit the subscript M if the machine is clear from context.) For example, $\delta(p, a) = (q, b, \pm 1)$ means that

$$(p, xay, i) \;\Rightarrow\; (q, xby, i \pm 1)$$

for any non-negative integer i, any string x of length i, and any string y. The evolution of any Turing machine is ***deterministic***; each configuration C yields a *unique* configuration C'. We write $\boldsymbol{C \Rightarrow^* C'}$ to indicate that there is a (possibly empty) sequence of transitions from configuration C to configuration C'. (The symbol $\Rightarrow^*$ can be pronounced "eventually yields".)

The initial configuration is $(w, \mathsf{start}, 0)$ for some arbitrary (and possibly empty) ***input string*** $w \in \Sigma^*$. If M eventually reaches the accept state—more formally, if $(w, \mathsf{start}, 0) \Rightarrow^* (x, \mathsf{accept}, i)$ for some string $x \in \Gamma^*$ and some integer i—we say that M ***accepts*** the original input string w. Similarly, if M eventually reaches the reject state, we say that M ***rejects*** w. We must emphasize that "rejects" and "does not accept" are *not* synonyms; if M crashes or runs forever, then M neither accepts nor rejects w.

We distinguish between two different senses in which a Turing machine can "accept" a language. Let M be a Turing machine with input alphabet Σ, and let $L \subseteq \Sigma^*$ be an arbitrary language over Σ.

- M ***recognizes*** or ***accepts*** L if and only if M accepts every string in L but nothing else. A language is ***recognizable*** (or *semi-computable* or *recursively enumerable*) if it is recognized by some Turing machine.
- M ***decides*** L if and only if M accepts every string in L and rejects every string in $\Sigma^* \setminus L$. Equivalently, M decides L if and only if M recognizes L and halts (without crashing) on all inputs. A language is ***decidable*** (or *computable* or *recursive*) if it is decided by some Turing machine.

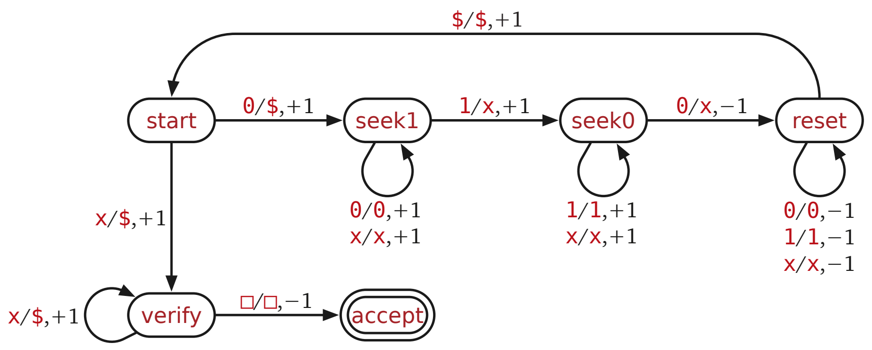
Trivially, every decidable language is recognizable, but (as we will see later), not every recognizable language is decidable.

6.3 A First Example

Consider the language $L = \{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$. This language is neither regular nor context-free, but it can be decided by the following six-state Turing machine. The alphabets and states of the machine are defined as follows:

$$\Gamma = \{\mathtt{0}, \mathtt{1}, \mathtt{\$}, \mathtt{x}, \square\}$$
$$\Sigma = \{\mathtt{0}, \mathtt{1}\}$$
$$Q = \{\mathsf{start}, \mathsf{seek1}, \mathsf{seek0}, \mathsf{reset}, \mathsf{verify}, \mathsf{accept}, \mathsf{reject}\}$$

The transition function is described in the table on the next page; all unspecified transitions lead to the reject state. The figure below shows a graphical representation of the same machine, which resembles a drawing of a DFA, but with output symbols and actions specified on each edge. For example, we indicate the transition $\delta(p, \mathtt{0}) = (q, \mathtt{1}, +1)$ by writing $\mathtt{0}/\mathtt{1}, +1$ next to the arrow from state p to state q.



A graphical representation of the example Turing machine

$\delta(\;p\;, a) = (\;q\;, b, \Delta)$	explanation
$\delta(\text{start}, \texttt{0}) = (\text{seek1}, \texttt{\$}, +1)$	mark first $\texttt{0}$ and scan right
$\delta(\text{start}, \texttt{x}) = (\text{verify}, \texttt{\$}, +1)$	looks like we're done, but let's make sure
$\delta(\text{seek1}, \texttt{0}) = (\text{seek1}, \texttt{0}, +1)$	scan rightward for $\texttt{1}$
$\delta(\text{seek1}, \texttt{x}) = (\text{seek1}, \texttt{x}, +1)$	
$\delta(\text{seek1}, \texttt{1}) = (\text{seek0}, \texttt{x}, +1)$	mark $\texttt{1}$ and continue right
$\delta(\text{seek0}, \texttt{1}) = (\text{seek0}, \texttt{1}, +1)$	scan rightward for $\texttt{0}$
$\delta(\text{seek0}, \texttt{x}) = (\text{seek0}, \texttt{x}, +1)$	
$\delta(\text{seek0}, \texttt{0}) = (\text{reset}, \texttt{x}, +1)$	mark $\texttt{0}$ and scan left
$\delta(\text{reset}, \texttt{0}) = (\text{reset}, \texttt{0}, -1)$	scan leftward for $\texttt{\$}$
$\delta(\text{reset}, \texttt{1}) = (\text{reset}, \texttt{1}, -1)$	
$\delta(\text{reset}, \texttt{x}) = (\text{reset}, \texttt{x}, -1)$	
$\delta(\text{reset}, \texttt{\$}) = (\text{start}, \texttt{\$}, +1)$	step right and start over
$\delta(\text{verify}, \texttt{x}) = (\text{verify}, \texttt{\$}, +1)$	scan right for any unmarked symbol
$\delta(\text{verify}, \square) = (\text{accept}, \square, -1)$	success!

The transition function for a Turing machine that decides the language $\{\texttt{0}^n\texttt{1}^n\texttt{0}^n \mid n \geq 0\}$.

Finally, we trace the execution of this machine on two input strings: $\texttt{001100} \in L$ and $\texttt{00100} \notin L$. In each configuration, we indicate the position of the head using a small triangle instead of listing the position explicitly. Notice that we automatically add blanks to the tape string as necessary. Proving that this machine actually decides L—and in particular, that it never crashes or infinite-loops—is a straightforward but tedious exercise in induction.

$$
\begin{aligned}
&(\text{start}, \underset{\blacktriangle}{\texttt{0}}\texttt{01100}) \Rightarrow (\text{seek1}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{1100}) \Rightarrow (\text{seek1}, \texttt{\$0}\underset{\blacktriangle}{\texttt{1}}\texttt{100}) \Rightarrow (\text{seek0}, \texttt{\$0x}\underset{\blacktriangle}{\texttt{1}}\texttt{00}) \Rightarrow (\text{seek0}, \texttt{\$0x1}\underset{\blacktriangle}{\texttt{0}}\texttt{0})\\
&\Rightarrow (\text{reset}, \texttt{\$0x}\underset{\blacktriangle}{\texttt{1}}\texttt{x0}) \Rightarrow (\text{reset}, \texttt{\$0}\underset{\blacktriangle}{\texttt{x}}\texttt{1x0}) \Rightarrow (\text{reset}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{x1x0}) \Rightarrow (\text{reset}, \underset{\blacktriangle}{\texttt{\$}}\texttt{0x1x0})\\
&\Rightarrow (\text{start}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{x1x0})\\
&\Rightarrow (\text{seek1}, \texttt{\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{1x0}) \Rightarrow (\text{seek1}, \texttt{\$\$x}\underset{\blacktriangle}{\texttt{1}}\texttt{x0}) \Rightarrow (\text{seek0}, \texttt{\$\$xx}\underset{\blacktriangle}{\texttt{x}}\texttt{0}) \Rightarrow (\text{seek0}, \texttt{\$\$xxx}\underset{\blacktriangle}{\texttt{0}})\\
&\Rightarrow (\text{reset}, \texttt{\$\$xx}\underset{\blacktriangle}{\texttt{x}}\texttt{x}) \Rightarrow (\text{reset}, \texttt{\$\$x}\underset{\blacktriangle}{\texttt{x}}\texttt{xx}) \Rightarrow (\text{reset}, \texttt{\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{xxx}) \Rightarrow (\text{reset}, \texttt{\$}\underset{\blacktriangle}{\texttt{\$}}\texttt{xxxx})\\
&\Rightarrow (\text{start}, \texttt{\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{xxx})\\
&\Rightarrow (\text{verify}, \texttt{\$\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{xx}) \Rightarrow (\text{verify}, \texttt{\$\$\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{x}) \Rightarrow (\text{verify}, \texttt{\$\$\$\$\$}\underset{\blacktriangle}{\texttt{x}}) \Rightarrow (\text{verify}, \texttt{\$\$\$\$\$\$}\underset{\blacktriangle}{\square})\\
&\Rightarrow (\text{accept}, \texttt{\$\$\$\$\$}\underset{\blacktriangle}{\texttt{\$}}) \Rightarrow \textbf{accept!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string $\texttt{001100} \in L$

$$
\begin{aligned}
&(\text{start}, \underset{\blacktriangle}{\texttt{0}}\texttt{0100}) \Rightarrow (\text{seek1}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{100}) \Rightarrow (\text{seek1}, \texttt{\$0}\underset{\blacktriangle}{\texttt{1}}\texttt{00}) \Rightarrow (\text{seek0}, \texttt{\$0x}\underset{\blacktriangle}{\texttt{0}}\texttt{0})\\
&\Rightarrow (\text{reset}, \texttt{\$0}\underset{\blacktriangle}{\texttt{x}}\texttt{x0}) \Rightarrow (\text{reset}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{xx0}) \Rightarrow (\text{reset}, \underset{\blacktriangle}{\texttt{\$}}\texttt{0xx0})\\
&\Rightarrow (\text{start}, \texttt{\$}\underset{\blacktriangle}{\texttt{0}}\texttt{xx0})\\
&\Rightarrow (\text{seek1}, \texttt{\$\$}\underset{\blacktriangle}{\texttt{x}}\texttt{x0}) \Rightarrow (\text{seek1}, \texttt{\$\$x}\underset{\blacktriangle}{\texttt{x}}\texttt{0}) \Rightarrow (\text{seek1}, \texttt{\$\$xx}\underset{\blacktriangle}{\texttt{0}}) \Rightarrow \textbf{reject!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string $\texttt{00100} \notin L$

6.4 Variations

There are actually several formal models that all fall under the name "Turing machine", each with small variations on the definition we've given. Although we do need to be explicit about *which* variant we want to use for any particular problem, the differences between the variants are relatively unimportant. For any machine defined in one model, there is an equivalent machine in each of the other models; in particular, all of these variants recognize the same languages and decide the same languages. For example:

- **Halting conditions.** Some models allow multiple accept and reject states, which (depending on the precise model) trigger acceptance or rejection either when the machine enters the state, or when the machine has no valid transitions out of such a state. Others include only explicit accept states, and either equate crashing with rejection or do not define a rejection mechanism at all. Still other models include halting as one of the possible *actions* of the machine, in addition to moving left or moving right; in these models, the machine accepts/rejects its input if and only if it halts in an accepting/non-accepting state.

- **Actions.** Some Turing machine models allow transitions that do not move the head, or that move the head by more than one cell in a single step. Others insist that a single step of the machine *either* writes a new symbol onto the tape *or* moves the head one step. Finally, as mentioned above, some models include halting as one of the available actions.

- **Transition function.** Some models of Turing machines, including Turing's original definition, allow the transition function to be undefined on some state-symbol pairs. In this formulation, the transition function is given by a set $\delta \subset Q \times \Gamma \times Q \times \Gamma \times \{+1, -1\}$, such that for each state q and symbol a, there is at most one transition $(q, a, \cdot, \cdot, \cdot) \in \delta$. If the machine enters a configuration from which there is no transition, it halts and (depending on the precise model) either crashes or rejects. Others define the transition function as $\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{-1, +1\})$, allowing the machine to *either* write a symbol to the tape *or* move the head in each step.

- **Beginning of the tape.** Some models forbid the head to move past the beginning of the tape, either by starting the tape with a special symbol that cannot be overwritten and that forces a rightward transition, or by declaring that a leftward transition at position 0 leaves the head in position 0, or even by pure fiat—declaring any machine that performs a leftward move at position 0 to be invalid.

To prove that any two of these variant "species" of Turing machine are equivalent, we must show how to transform a machine of one species into a machine of the other species that accepts and rejects the same strings. For example, let $M = (\Gamma, \square, \Sigma, Q, s, \text{accept}, \text{reject}, \delta)$ be a Turing machine with explicit accept and reject states. We can define an equivalent Turing machine M' that halts only when it moves left from position 0, and accepts only by halting while in an accepting state, as follows. We define the set of accepting states for M' as $A = \{\text{accept}\}$ and define a new transition function

$$\delta'(q, a) := \begin{cases} (\text{accept}, a, -1) & \text{if } q = \text{accept} \\ (\text{reject}, a, -1) & \text{if } q = \text{reject} \\ \delta(q, a) & \text{otherwise} \end{cases}$$

Similarly, suppose someone gives us a Turing machine $M = (\Gamma, \square, \Sigma, Q, s, \text{accept}, \text{reject}, \delta)$ whose transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ allows the machine to transition without moving its head. Then we can construct an equivalent Turing machine $M' = (\Gamma, \square, \Sigma, Q', s, \texttt{accept}, \texttt{reject}, \delta')$ that moves its head at every transition by defining $Q' := Q \times \{0,1\}$ and

$$\begin{aligned}\delta'((p,0),a) &:= \begin{cases}((q,1),b,+1) & \text{if } \delta(p,a) = (q,b,0),\\ ((q,0),b,\Delta) & \text{if } \delta(p,a) = (q,b,\Delta) \text{ and } \Delta \neq 0,\end{cases}\\ \delta'((p,1),a) &:= ((p,0),a,-1).\end{aligned}$$

6.5 Computing Functions

Turing machines can also be used to compute functions from strings to strings, instead of just accepting or rejecting strings. Since we don't care about acceptance or rejection, we replace the explicit accept and reject states with a single halt state, and we define the ***output*** of the Turing machine to be the contents of the tape when the machine halts, after removing the infinite sequence of trailing blanks. More formally, for any Turing machine M, any string $w \in \Sigma^*$, and any string $x \in \Gamma^*$ that does not end with a blank, we write $\boldsymbol{M(w) = x}$ if and only if $(w, s, 0) \Rightarrow_M^* (x, \texttt{halt}, i)$ for some integer i. If M does not halt on input w, then we write $\boldsymbol{M(w)\nearrow}$, which can be read either "M diverges on w" or "$M(w)$ is undefined." We say that M ***computes*** the function $f : \Sigma^* \to \Sigma^*$ if and only if $M(w) = f(w)$ for every string w.

6.5.1 Shifting

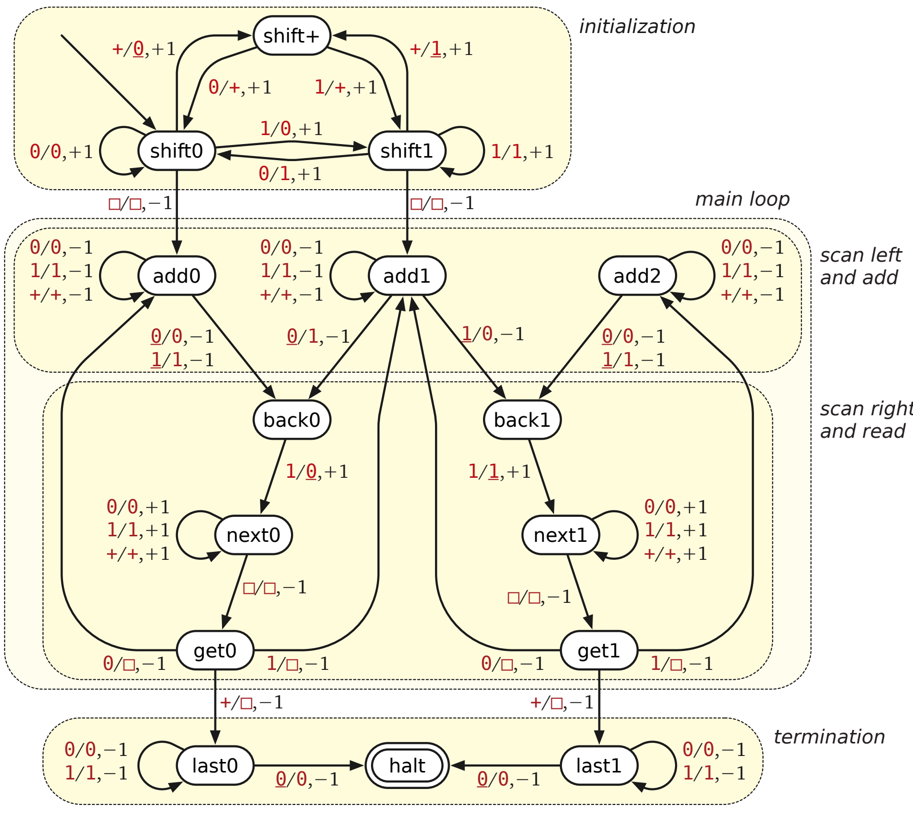
One basic operation that is used in many Turing machine constructions is ***shifting*** the input string a constant number of steps to the right or to the left. For example, given any input string $w \in \{\texttt{0}, \texttt{1}\}^*$, we can compute the string $\texttt{0}w$ using a Turing machine with tape alphabet $\Gamma = \{\texttt{0}, \texttt{1}, \square\}$, state set $Q = \{0, 1, \texttt{halt}\}$, start state 0, and the following transition function:

$\delta(p,\ a)$	$=$	$(\ q\ ,$	$b,$	$\Delta\ )$
$\delta(0,\ \texttt{0})$	$=$	$(\ 0\ ,$	$\texttt{0},$	$+1)$
$\delta(0,\ \texttt{1})$	$=$	$(\ 1\ ,$	$\texttt{0},$	$+1)$
$\delta(0,\ \square)$	$=$	$(\texttt{halt},$	$\texttt{0},$	$+1)$
$\delta(1,\ \texttt{0})$	$=$	$(\ 0\ ,$	$\texttt{1},$	$+1)$
$\delta(1,\ \texttt{1})$	$=$	$(\ 1\ ,$	$\texttt{1},$	$+1)$
$\delta(1,\ \square)$	$=$	$(\texttt{halt},$	$\texttt{1},$	$+1)$

By increasing the number of states, we can build a Turing machine that shifts the input string any fixed number of steps in either direction. For example, a machine that shifts its input to the left by five steps might read the string from right to left, storing the five most recently read symbols in its internal state. A typical transition for such a machine would be $\delta(12345, 0) = (01234, \texttt{5}, -1)$.

6.5.2 Binary Addition

With a more complex Turing machine, we can implement binary addition. The input is a string of the form $w\texttt{+}x$, where $w, x \in \{\texttt{0}, \texttt{1}\}^n$, representing two numbers in binary; the output is the binary representation of $w + x$. To simplify our presentation, we assume that $|w| = |x| > 0$; however, this restrictions can be removed with the addition of a few more states. The following figure shows the entire Turing machine at a glance. The machine uses the tape alphabet $\Gamma = \{\square, \texttt{0}, \texttt{1}, \texttt{+}, \underline{\texttt{0}}, \underline{\texttt{1}}\}$; the start state is shift0. All missing transitions go to a fail state, indicating that the input was badly formed.



A Turing machine that adds two binary numbers of the same length.

Execution of this Turing machine proceeds in several phases, each with its own subset of states, as indicated in the figure. The initialization phase scans the entire input, shifting it to the right to make room for the output string, marking the rightmost bit of w, and reading and erasing the last bit of x.

$\delta(\;p\;, a)$	=	$(\;q\;, b, \Delta)$
$\delta(\text{shift0}, 0)$	=	$(\text{shift0}, 0, +1)$
$\delta(\text{shift0}, 1)$	=	$(\text{shift1}, 0, +1)$
$\delta(\text{shift0}, +)$	=	$(\text{shift+}, \underline{0}, +1)$
$\delta(\text{shift0}, \square)$	=	$(\text{add0}, \square, -1)$
$\delta(\text{shift1}, 0)$	=	$(\text{shift0}, 1, +1)$
$\delta(\text{shift1}, 1)$	=	$(\text{shift1}, 1, +1)$
$\delta(\text{shift1}, +)$	=	$(\text{shift+}, \underline{1}, +1)$
$\delta(\text{shift1}, \square)$	=	$(\text{add1}, \square, -1)$
$\delta(\text{shift+}, 0)$	=	$(\text{shift0}, +, +1)$
$\delta(\text{shift+}, 1)$	=	$(\text{shift1}, +, +1)$

The first part of the main loop scans left to the marked bit of w, adds the bit of x that was just erased plus the carry bit from the previous iteration, and records the carry bit for the next iteration in the machines internal state.

$\delta(p, a) = (q, b, \Delta)$	$\delta(p, a) = (q, b, \Delta)$	$\delta(p, a) = (q, b, \Delta)$
δ(add0, 0) = (add0, 0, −1)	δ(add1, 0) = (add1, 0, −1)	δ(add2, 0) = (add2, 0, −1)
δ(add0, 1) = (add0, 0, −1)	δ(add1, 1) = (add1, 0, −1)	δ(add2, 1) = (add2, 0, −1)
δ(add0, +) = (add0, 0, −1)	δ(add1, +) = (add1, 0, −1)	δ(add2, +) = (add2, 0, −1)
δ(add0, $\underline{0}$) = (back0, 0, −1)	δ(add1, $\underline{0}$) = (back0, 1, −1)	δ(add2, $\underline{0}$) = (back1, 0, −1)
δ(add0, $\underline{1}$) = (back0, 1, −1)	δ(add1, $\underline{1}$) = (back1, 0, −1)	δ(add2, $\underline{1}$) = (back1, 1, −1)

The second part of the main loop marks the previous bit of w, scans right to the end of x, and then reads and erases the last bit of x, all while maintaining the carry bit.

$\delta(p, a) = (q, b, \Delta)$	$\delta(p, a) = (q, b, \Delta)$
δ(back0, 0) = (next0, $\underline{0}$, +1)	δ(back1, 0) = (next1, $\underline{0}$, +1)
δ(back0, 1) = (next0, $\underline{1}$, +1)	δ(back1, 1) = (next1, $\underline{1}$, +1)
δ(next0, 0) = (next0, 0, +1)	δ(next1, 0) = (next1, 0, +1)
δ(next0, 1) = (next0, 0, +1)	δ(next1, 1) = (next1, 0, +1)
δ(next0, +) = (next0, 0, +1)	δ(next1, +) = (next1, 0, +1)
δ(next0, □) = (get0, □, −1)	δ(next1, □) = (get1, □, −1)
δ(get0, 0) = (add0, □, −1)	δ(get1, 0) = (add1, □, −1)
δ(get0, 1) = (add1, □, −1)	δ(get1, 1) = (add2, □, −1)
δ(get0, +) = (last0, □, −1)	δ(get1, +) = (last1, □, −1)

Finally, after erasing the + in the last iteration of the main loop, the termination phase adds the last carry bit to the leftmost output bit and halts.

$\delta(p, a) = (q, b, \Delta)$
δ(last0, 0) = (last0, 0, −1)
δ(last0, 1) = (last0, 0, −1)
δ(last0, $\underline{0}$) = (halt, 0,)
δ(last1, 0) = (last1, 0, −1)
δ(last1, 1) = (last1, 0, −1)
δ(last1, $\underline{0}$) = (halt, 1,)

6.6 Variations on Tracks, Heads, and Tapes

Multiple Tracks

It is sometimes convenient to endow the Turing machine tape with multiple ***tracks***, each with its own tape alphabet, and allow the machine to read from and write to the same position on all tracks simultaneously. For example, to define a Turing machine with three tracks, we need three tape alphabets Γ_1, Γ_2, and Γ_3, each with its own blank symbol, where (say) Γ_1 contains the input alphabet Σ as a subset; we also need a transition function of the form

$$\delta : Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \to Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \times \{-1, +1\}$$

Describing a configuration of this machine requires a quintuple (q, x_1, x_2, x_3, i), indicating that each track i contains the string x_i followed by an infinite sequence of blanks. The initial configuration is $(\text{start}, w, \varepsilon, \varepsilon, 0)$, with the input string written on the first track, and the other two tracks completely blank.

But any such machine is equivalent (if not *identical*) to a single-track Turing machine with the (still finite!) tape alphabet $\Gamma := \Gamma_1 \times \Gamma_2 \times \Gamma_3$. Instead of thinking of the tape as three infinite sequences of symbols, we think of it as a single infinite sequence of "records", each containing three symbols. Moreover, there's nothing special about the number 3 in this construction; a Turing machine with *any* constant number of tracks is equivalent to a single-track machine.

Doubly-Infinite Tape

It is also sometimes convenient to allow the tape to be infinite in both directions, for example, to avoid boundary conditions. There are several ways to simulate a doubly-infinite tape on a machine with only a semi-infinite tape. Perhaps the simplest method is to use a semi-infinite tape with two tracks, one containing the cells with positive index and the other containing the cells with negative index in reverse order, with a special marker symbol at position zero to indicate the transition.

0	+1	+2	+3	+4	···
►	−1	−2	−3	−4	···

Another method is to shuffle the positive-index and negative-index cells onto a single track, and add additional states to allow the Turing machine to move two steps in a single transition. Again, we need a special symbol at the left end of the tape to indicate the transition:

►	0	−1	+1	−2	+2	−3	+3	···

A third method maintains two sentinel symbols ► and ◄ that surround all other non-blank symbols on the tape. Whenever the machine reads the right sentinel ◄, we write a blank, move right, write ◄, move left, and then proceed as if we had just read a blank. On the other hand, when the machine reads the left sentinel ►, we shift the entire contents of the tape (up to and including the right sentinel) one step to the right, then move back to the left sentinel, move right, write a blank, and finally proceed as if we had just read a blank. Since the Turing machine does not actually have access to the position of the head *as an integer*, shifting the head and the tape contents one step right has no effect on its future evolution.

►	−3	−2	−1	0	+1	+2	+3	+4	+5	◄	···

Using either of the first two methods, we can simulate t steps of an arbitrary Turing machine with a doubly-infinite tape using only $O(t)$ steps on a standard Turing machine. The third method, unfortunately, requires $\Theta(t^2)$ steps in the worst case.

Insertion and Deletion

We can also allow Turing machines to insert and delete cells on the tape, in addition to simply overwriting existing symbols. We've already seen how to insert a new cell: Leave a special mark on the tape (perhaps in a second track), shift everything to the right of this mark one cell to the right, scan left to the mark, erase the mark, and finally write the correct character into the new cell. Deletion is similar: Mark the cell to be deleted, shift everything to the right of the mark one step to the left, scan left to the mark, and erase the mark. We may also need to maintain a mark in some cell to the right every non-blank symbol, indicating that all cells further to the right are blank, so that we know when to stop shifting left or right.

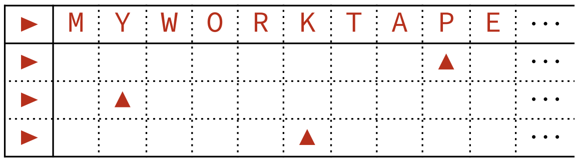
Multiple Heads

Another convenient extension is to allow machines simultaneous access to more than one position on the tape. For example, to define a Turing machine with *three* heads, we need a transition function of the form

$$\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{-1, +1\}^3.$$

Describing a configuration of such a machine requires a quintuple (q, x, i, j, k), indicating that the machine is in state q, the tape contains string x, and the three heads are at positions i, j, k. The transition function tells us, given q and the three symbols $x[i], x[j], x[k]$, which three symbols to write on the tape and which direction to move each of the heads.

We can simulate this behavior with a single head by adding additional tracks to the tape that record the positions of each head. To simulate a machine M with three heads, we use a tape with four tracks: track 0 is the actual work tape; each of the remaining tracks has a single non-blank symbol recording the position of one of the heads. We also insert a special marker symbols at the left end of the tape.



We can simulate any single transition of M, starting with our single head at the left end of the tape, as follows. Throughout the simulation, we maintain the internal state of M as one of the components of our current state. First, for each i, we read the symbol under the ith head of M as follows:

> Scan to the right to find the mark on track i, read the corresponding symbol from track 0 into our internal state, and then return to the left end of the tape.

At this point, our internal state records M's current internal state and the three symbols under M's heads. After one more transition (using M's transition function), our internal state records M's *next* state, the symbol to be written by each head, and the direction to move each head. Then, for each i, we write with and move the ith head of M as follows:

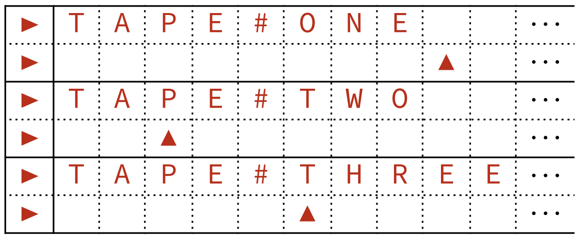
> Scan to the right to find the mark on track i, write the correct symbol onto on track 0, move the mark on track i one step left or right, and then return to the left end of the tape.

Again, there is nothing special about the number 3 here; we can simulate machines with *any* fixed number of heads.

Careful analysis of this technique implies that for any integer k, we can simulate t steps of an arbitrary Turing machine with k independent heads in $\Theta(t^2)$ time on a standard Turing machine with only one head. Unfortunately, this quadratic blowup is unavoidable. It is relatively easy to recognize the language of *marked palindromes* $\{w \bullet w^R \mid w \in \{0, 1\}^*\}$ in $O(n)$ time using a Turing machine with two heads, but recognizing this language provably requires $\Omega(n^2)$ time on a standard machine with only one head. On the other hand, with much more sophisticated techniques, it is possible to simulate t steps of a Turing machine with k head, for any fixed integer k, using only $O(t \log t)$ steps on a Turing machine with just *two* heads.

Multiple Tapes

We can also allow machines with multiple independent tapes, each with its own head. To simulate such a machine with a single tape, we simply maintain each tape as an independent track with its own head. Equivalently, we can simulate a machine with k tapes using a single tape with $2k$ tracks, half storing the contents of the k tapes and half storing the positions of the k heads.



Just as for multiple tracks, for any constant k, we can simulate t steps of an arbitrary Turing machine with k independent tapes in $\Theta(t^2)$ steps on a standard Turing machine with one tape, and this quadratic blowup is unavoidable. Moreover, it is possible to simulate t steps on a k-tape Turing machine using only $O(t \log t)$ steps on a *two*-tape Turing machine using more sophisticated techniques. (This faster simulation is easier to obtain for multiple independent tapes than for multiple heads on the same tape.)

By combining these tricks, we can simulate a Turing machine with any fixed number of tapes, each of which may be infinite in one or both directions, each with any fixed number of heads and any fixed number of tracks, with at most a quadratic blowup in the running time.

6.7 Simulating a Real Computer

6.7.1 Subroutines and Recursion

★★★

Use a second tape/track as a "call stack". Add save and restore actions. In the simplest formulation, subroutines do not have local memory. To call a subroutine, save the current state onto the call stack and jump to the first state of the subroutine. To return, restore (and remove) the return state from the call stack. We can simulate t steps of any recursive Turing machine with $O(t)$ steps on a multitape standard Turing machine, or in $O(t^2)$ steps on a standard Turing machine.

More complex versions of this simulation can adapt to

6.7.2 Random-Access Memory

★★★

Keep [address•data] pairs on a separate "memory" tape. Write address to an "address" tape; read data from or write data to a "data" tape. Add new or changed [address•data] pairs at the end of the memory tape. (Semantics of reading from an address that has never been written to?)

Suppose all memory accesses require at most ℓ address and data bits. Then we can simulate the kth memory access in $O(k\ell)$ steps on a multitape Turing machine or in $O(k^2\ell^2)$ steps on a single-tape machine. Thus, simulating t memory accesses in a random-access machine with ℓ-bit words requires $O(t^2\ell)$ time on a multitape Turing machine, or $O(t^3\ell^2)$ time on a single-tape machine.

6.8 Universal Turing Machines

With all these tools in hand, we can now describe the pinnacle of Turing machine constructions: the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w\rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w\rangle$ if and only if M accepts w.
- U rejects $\langle M, w\rangle$ if and only if M rejects w.

The next few pages, I will sketch a universal Turing machine U that uses the input alphabet $\{\texttt{0}, \texttt{1}, \texttt{[}, \texttt{]}, \bullet, |\}$ and a somewhat larger tape alphabet (via marks on additional tracks). However, I will *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|)\rceil$ bits. Thus, if $\Gamma = \{\mathit{0}, \mathit{1}, \$, x, \square\}$, we might use the following encoding:

$$\langle \mathit{0}\rangle = \texttt{001}, \qquad \langle \mathit{1}\rangle = \texttt{010}, \qquad \langle \$\rangle = \texttt{011}, \qquad \langle x\rangle = \texttt{100}, \qquad \langle \square\rangle = \texttt{000}.$$

The input string w is encoded by its sequence of symbol encodings, with separators $\bullet$ between every pair of symbols and with brackets [and] around the whole string. For example, with this encoding, the input string *001100* would be encoded on the input tape as

$$\langle \mathit{001100}\rangle = \texttt{[001}\bullet\texttt{001}\bullet\texttt{010}\bullet\texttt{010}\bullet\texttt{001}\bullet\texttt{001]}$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q\rangle$ of $\lceil \lg|Q|\rceil$ bits. Without loss of generality, we encode the start state with all 1s and the reject state with all 0s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start}\rangle = \texttt{111} \qquad \langle \mathit{seek1}\rangle = \texttt{010} \qquad \langle \mathit{seek0}\rangle = \texttt{011} \qquad \langle \mathit{reset}\rangle = \texttt{100}$$
$$\langle \mathit{verify}\rangle = \texttt{101} \qquad \langle \mathit{accept}\rangle = \texttt{110} \qquad \langle \mathit{reject}\rangle = \texttt{000}$$

We encode the machine M itself as the string $\langle M\rangle = \texttt{[}\langle \mathit{reject}\rangle\bullet\langle\square\rangle\texttt{]}\langle\delta\rangle$, where $\langle\delta\rangle$ is the concatenation of substrings $\texttt{[}\langle p\rangle\bullet\langle a\rangle|\langle q\rangle\bullet\langle b\rangle\bullet\langle\Delta\rangle\texttt{]}$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq \mathrm{reject}$. We encode the actions $\Delta = \pm 1$ by defining $\langle -1\rangle := \texttt{0}$ and $\langle +1\rangle := \texttt{1}$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \$) = (\mathit{start}, \$, +1)$ would be encoded as

$$\texttt{[100}\bullet\texttt{011|001}\bullet\texttt{011}\bullet\texttt{1]}.$$

Our first example Turing machine for recognizing $\{\mathit{0}^n\mathit{1}^n\mathit{0}^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
         [010•001|010•001•1][010•100|010•100•1]
         [010•010|011•100•1][011•010|011•010•1]
         [011•100|011•100•1][011•001|100•100•1]
         [100•001|100•001•0][100•010|100•010•0]
         [100•100|100•100•0][100•011|001•011•1]
         [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $\texttt{[}\langle q\rangle\bullet\langle a\rangle\texttt{]}$ indicating that (1) the cell contains symbol a; (2) if $q \ne \textit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \textit{reject}$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets `[` and `]`.

For example, with the encodings described above, the initial configuration $(\textit{start}, \underset{\blacktriangle}{\texttt{0}}\texttt{01100}, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$\texttt{[}\underbrace{\texttt{[111}\bullet\texttt{001]}}_{\textbf{start}\ \texttt{0}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\textit{reject}\ \texttt{0}}\underbrace{\texttt{[000}\bullet\texttt{010]}}_{\textit{reject}\ \texttt{1}}\underbrace{\texttt{[000}\bullet\texttt{010]}}_{\textit{reject}\ \texttt{1}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\textit{reject}\ \texttt{0}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\textit{reject}\ \texttt{0}}\texttt{]}$$

Similarly, the intermediate configuration $(\textit{reset}, \texttt{\$0x}\underset{\blacktriangle}{\texttt{1}}\texttt{x0}, 3)$ would be encoded as follows:

$$\texttt{[}\underbrace{\texttt{[000}\bullet\texttt{011]}}_{\textit{reject}\ \texttt{\$}}\underbrace{\texttt{[000}\bullet\texttt{011]}}_{\textit{reject}\ \texttt{0}}\underbrace{\texttt{[000}\bullet\texttt{100]}}_{\textit{reject}\ \texttt{x}}\underbrace{\texttt{[010}\bullet\texttt{010]}}_{\textbf{reset}\ \texttt{1}}\underbrace{\texttt{[000}\bullet\texttt{100]}}_{\textit{reject}\ \texttt{x}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\textit{reject}\ \texttt{0}}\texttt{]}$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings `[[` and `]]` each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle \textit{start}, x, 0\rangle$ of the initial configuration of M with input x. First, we write $\texttt{[[}\langle\textit{start}\rangle\bullet$. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write $\texttt{[[}\langle\textit{start}\rangle\bullet$.
- Instead of copying each separator $\bullet$, write $\texttt{][}\langle\textit{reject}\rangle\bullet$
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle\textit{start}\rangle$ and $\langle\textit{reject}\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle\textit{start}\rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell $\texttt{[}\langle p\rangle\bullet\langle a\rangle\texttt{]}$ such that $p \ne \textit{reject}$. Then we scan through the encoded transition function $\langle\delta\rangle$ to find the unique encoded tuple $\texttt{[}\langle p\rangle\bullet\langle a\rangle\texttt{|}\langle q\rangle\bullet\langle b\rangle\bullet\langle\Delta\rangle\texttt{]}$ whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle\bullet\langle b\rangle$ of the tuple to the work tape. Now if $q = \textit{accept}$, then U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \ne \textit{reject}$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with $\texttt{[}\langle\textit{reject}\rangle\bullet\langle\square\rangle\texttt{]]}$ (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks $\texttt{[}\langle\textit{reject}\rangle\bullet\langle\square\rangle\texttt{]}$ instead of *actual* blanks $\square$.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string $\mathtt{001100}$. The execution of M on input w eventually reaches the configuration $(\mathit{seek1}, \$\$x\underset{\blacktriangle}{1}x0, 3)$. At the start of the corresponding step in U's simulation, U is in the following configuration:

$$\underset{\blacktriangle}{\texttt{[}}\texttt{[000}\bullet\texttt{011][000}\bullet\texttt{011][000}\bullet\texttt{100][010}\bullet\texttt{010][000}\bullet\texttt{100][000}\bullet\texttt{001]]}$$

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix $\texttt{[}\langle \mathit{reject}\rangle\bullet$ of the machine encoding $\langle M\rangle$ on the input tape. U finds a match in the fourth encoded cell.

$$\texttt{[[000}\bullet\texttt{011][000}\bullet\texttt{011][000}\bullet\texttt{100][010}\underset{\blacktriangle}{\bullet}\texttt{010][000}\bullet\texttt{100][000}\bullet\texttt{001]]}$$

Next, U scans the machine encoding $\langle M\rangle$ for the substring $\texttt{[010}\bullet\texttt{010}$ matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition $\texttt{[010}\bullet\texttt{010|011}\bullet\texttt{100}\bullet\texttt{1]}$. U copies the state-symbol pair $\texttt{011}\bullet\texttt{100}$ from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

$$\texttt{[[000}\bullet\texttt{011][000}\bullet\texttt{011][000}\bullet\texttt{100][}\underline{\texttt{011}\bullet\texttt{100}}\underset{\blacktriangle}{\texttt{]}}\texttt{[000}\bullet\texttt{100][000}\bullet\texttt{001]]}$$

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

$$\texttt{[[000}\bullet\texttt{011][000}\bullet\texttt{011][000}\bullet\texttt{100][}\underline{\texttt{000}}\bullet\texttt{100][}\underline{\texttt{011}}\underset{\blacktriangle}{\bullet}\texttt{100][000}\bullet\texttt{001]]}$$

Finally, U scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration $(\mathit{seek0}, \$\$xx\underset{\blacktriangle}{x}0, 4)$, as required.

$$\underset{\blacktriangle}{\texttt{[}}\texttt{[000}\bullet\texttt{011][000}\bullet\texttt{011][000}\bullet\texttt{100][000}\bullet\texttt{100][011}\bullet\texttt{100][000}\bullet\texttt{001]]}$$

Exercises

In the following problems, a *standard* Turing machine has a single semi-infinite tape, one read-write head, and the input alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. For problems that ask you to *construct* a standard Turing machine, you may assume without loss of generality that the initial tape contains a special symbol ▶ just to the left of the input string, indicating the left end of the tape; the read-write head starts just to the right of this symbol. For problems that ask you to *simulate* a standard Turing machine, you may assume without loss of generality that the tape alphabet is $\{\mathtt{0}, \mathtt{1}, \square\}$.

Turing Machine Programming

1. Describe standard Turing machines that decide each of the following languages:

   (a) Palindromes over the alphabet $\{\mathtt{0}, \mathtt{1}\}$

   (b) $\{ww \mid w \in \{\mathtt{0}, \mathtt{1}\}^*\}$

(c) $\{0^a 1^b 0^{ab} \mid a, b \in \mathbb{N}\}$

2. Let $\langle n \rangle_2$ denote the binary representation of the non-negative integer n. For example, $\langle 17 \rangle_2 = 10001$ and $\langle 42 \rangle_2 = 101010$. Describe standard Turing machines that compute the following functions from $\{0,1\}^*$ to $\{0,1\}^*$:

   (a) $w \mapsto www$
   (b) $1^n 0 1^m \mapsto 1^{mn}$
   (c) $1^n \mapsto 1^{2^n}$
   (d) $1^n \mapsto \langle n \rangle_2$
   (e) $0^* \langle n \rangle_2 \mapsto 1^n$
   (f) $\langle n \rangle_2 \mapsto \langle n^2 \rangle_2$

3. Describe standard Turing machines that write each of the following infinite streams of bits onto their tape. Specifically, for each integer n, there must be a finite time after which the first n symbols on the tape always match the first n symbols in the target stream.

   (a) An infinite stream of 1s
   (b) 010110111011110111110111111 0…, where the nth block of 1s has length n.
   (c) The stream of bits whose nth bit is 1 if and only if n is prime.
   (d) The ***Thue-Morse sequence*** $T_0 \bullet T_1 \bullet T_2 \bullet T_3 \cdots$, where

$$T_n := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ T_{n-1} \bullet \overline{T_{n-1}} & \text{otherwise} \end{cases}$$

   where $\overline{w}$ indicates the binary string obtained from w by flipping every bit. Equivalently, the nth bit of the Thue Morse sequence if 0 if the binary representation of n has an even number of 1s and 1 otherwise.

   0110100110010110100101100110100110010110011010010110…

   (e) The ***Fibonacci*** sequence $F_0 \bullet F_1 \bullet F_2 \bullet F_3 \cdots$, where

$$F_n := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-2} \bullet F_{n-1} & \text{otherwise} \end{cases}$$

   01011010110110101101011011011010110110101101011011010101…

Simulation by "Weaker" Machines

4. A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing

the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

Prove that any standard Turing machine can be simulated by a two-stack machine. That is, given any standard Turing machine M, describe a two-stack machine M' that accepts and rejects exactly the same input strings as M.

5. A ***k*-register machine** is a finite-state automaton with k non-negative integer registers. Formally, a k-register machine consists of a finite set Q of states (which include start, accept, and reject) and a transition function

$$\delta : Q \times \{0,1\}^k \to Q \times \{\text{halve}, \text{nop}, \text{double}, \text{double+1}\}^k$$

that takes the internal state and the *signs* of the registers as input, and produces a new internal state and *changes* to the registers as output. The instructions halve, nop, double, and double+1 change any register with value n to $n/2$, n, $2n$, and $2n+1$, respectively.

For example, if $\delta(p,0,1,0,1) = (q, \text{halve}, \text{nop}, \text{halve}, \text{double+1})$, then from the configuration $(p,0,2,3,1)$, the machine would transition to $(q,1,2,1,3)$.

Prove that any standard Turing machine (with suitably encoded input and output) can be simulated by a two-register machine. The input to the register machine is encoded in reversed binary in one of the registers, so the parity of the register value is the first input bit; the other register is initially zero.

6. A ***k*-counter machine** (also known as a ***Minksy machine***) is a finite-state automaton with k non-negative integer registers. Formally, a k-counter machine consists of a finite set Q of states (which include start, accept, and reject) and a transition function

$$\delta : Q \times \{0,+\}^k \to Q \times \{\text{inc}, \text{nop}, \text{dec}\}^k$$

that takes the internal state and the *signs* of the registers as input, and produces a new internal state and changes to the registers as output. The instructions inc, nop, and dec change any register with value n to $n+1$, n, and $n-1$, respectively. The transition function must forbid decrementing a register whose value is already zero.

For example, if $\delta(p,0,+,+,+) = (q, \text{inc}, \text{dec}, \text{nop}, \text{dec})$, then from the configuration $(p,0,2,3,1)$, the machine would transition to $(q,1,1,3,0)$.

(a) Prove that any standard Turing machine (with suitably encoded input and output) can be simulated by a three-counter machine. *[Hint: Simulate a two-**register** machine, using the third counter for scratch work.]*

(b) Prove that any three-counter machine (with suitably encoded input and output) can be simulated by a two-counter machine. *[Hint: Store all three counters in a single integer of the form $2^a 3^b 5^c$, and use the other counter for scratch work.]*

⋆(c) Prove that a three-counter machine can compute *a suitable encoding of* any computable function. Specifically, for any computable function $f : \mathbb{N} \to \mathbb{N}$, prove there is a three-counter machine M that transforms any input $(n,0,0)$ into $(f(n),0,0)$. *[Hint: First* *transform* $(n,0,0)$ *to* $(2^n,0,0)$ *using all three counters; then run a two- (or three-)counter TM simulation to obtain* $(2^{f(n)},0,0)$*; and finally transform* $(2^{f(n)},0,0)$ *to* $(f(n),0,0)$ *using all three counters.]*

★(d) Prove that not two-counter machine can transform $(n,0)$ to $(2^n,0)$. This impossiblity result was independently proved by Bārzdiņš in 1963, Yao in 1971, and Schroeppel in 1972. [5]

7. A ***hole-punch*** Turing machine is a standard Turing machine with two restrictions. First, the tape alphabet has only two symbols □ and ■, and thus the input alphabet is the singleton set {■}. Second, the machine can never write a blank (□) over a non-blank (■); intuitively, the machine can punch new holes (■s) into the tape, but it cannot erase holes.

   Prove that any standard Turing machine (with a unary input alphabet) can be simulated by a hole-punch Turing machine.

8. A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

9. ⋆(a) Prove that any standard Turing machine can be simulated by a standard Turing machine with only three states. *[Hint: Keep an encoding of the state of the simulat**ed** machine on the tape of the simulat**ing** machine.]*

   ★(b) Prove that any standard Turing machine can be simulated by a standard Turing machine with only *two* states.

Simulating "Stronger" Machines

10. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta\colon Q\times\Gamma\to Q\times\Gamma\times\{\uparrow,\leftarrow,\downarrow,\rightarrow\}$, where the arrows indicate which direction the head should move.

    (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

[5]Ja. M. Barzdin′ [Jānis Bārzdiņš]. Ob odnom klasse mašin T′ûringa (mašiny Minskogo) [On a class of Turing machines (Minsky machines)]. *Algebra i Logika* 1(6):42–51, 1963. In Russian. Sorry.
Oscar H. Ibarra, Nicholas Q. Trân. A note on simple programs with two variables. *Theoretical Computer Science* 112(2): 391–397, 1993.
Rich Schroeppel. A two counter machine cannot calculate 2^N. Artificial Intelligence Memo 257, MIT AI Lab, May 1972. [Schroeppel claims that the same result was independently proved by Frances Yao in 1971.

(b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

- Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
- Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
- Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
- Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

11. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

    Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

12. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

    Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem 4!)

13. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

    Show that any tape-stack Turing machine can be simulated by a standard Turing machine.