Chapter 4

Turing Machines

Alan Turing completed the invention of what became known as Turing Machines in 1936. He was seeking to describe in a precise way what it means to systematically calculate or compute. Before going further let me note that Turing Machines are conceptual devices; while they are easy to describe on paper, and in principle could be built, Turing had no intention of building them or having others build them. (Turing was not simply an “ivory-tower” mathematician; in the late 1940s he was one of the earliest users of one of the few computers then in existence — at Manchester University in England).

Let’s consider what systematic computation meant in 1936. At the time there were machines and tools for helping with calculations: adding machines and slide rules; slide rules continued to be in widespread use until the early 1970s when calculators became relatively cheap. Also, there were card readers and sorters supporting data management (IBM was a leading supplier). There were even computers; but these were people whose job was to compute (a description of a computing room at Los Alamos in 1944 full of human computers can be found in Richard Feynman’s semi-autobiography “Surely You’re Joking Mr. Feynman,” in the chapter “Los Alamos from Below”).

What Turing was trying to formalize were the processes followed when calculating something: the orbit of a planet, the load a bridge could bear, the next move to play in a game of chess, etc., though subsequently he proposed a much broader view of what could be computed (at least in principle).

The key question Turing asked was how do people compute? Essentially people could do three things: they could think in their minds, they could write down something (on a sheet of paper), or they could read something they had written down previously. As we all know, with too many sheets of paper it is hard to keep track of what is where, and consequently it is helpful to number or label the pages. One way of organizing this is to keep the pages bound in a book in numbered order. However, for reasons of conceptual simplicity, Turing preferred to think of the pages as being on a roll, or tape as he called it, with the pages being accessed in consecutive order by rolling or unrolling the roll (this is the way in which long documents were made and read in ancient times).

The actions with the roll or tape, as we shall call it henceforth, were very simple: one could read the current page, one could decide to erase the current page and write something new (pencil, not ink), one could decide to advance to the next page or go back to the previous page. But does this really capture everything? Suppose one wanted to go to page 150 and one was currently on page 28: well, then one needs to advance one page 122 times — a tedious process, for sure, but one that achieves the desired result in the end. The implicit assumption is that one can hold the
count in one’s mind. But what if the numbers are too large to hold in mind? What one would like
to do is to write them on scraps of paper; however, this is cheating. What one has to do is use the
tape pages as the scraps of paper. How this might be done is left as an exercise to the reader (see
Exercise 2 below).

How did Turing abstract the notion of thinking? The crucial hypothesis was that a mind can
hold only a finite number of different thoughts, albeit extremely large, not an infinite number.
Why is this plausible? At the level of neurons (not that this was properly understood in Turing’s
day) simplifying a bit, each neuron is either sending a signal or not; as their number is finite, this
yields only a finite number of possibilities — albeit a ridiculously large number of possibilities. At
the level of elementary particles (electrons, protons, etc.), quantum physics states that there are
only a finite number of states for each particle even if they are not fully knowable, and again this
yields only a finite number of possibilities. So abstractly, thinking can be viewed as follows: the
mind has a (large) finite number of states in which it could currently be, conventionally denoted
by \( q_0, q_1, \ldots, q_{s-1} \). Computation takes the following form: when in state \( q \), on reading the current
page take an action as already described (rewrite the page, move to an adjacent page) and enter a
new state \( q' \) (possibly \( q' = q \)); this action is fully determined by the state \( q \) and the contents of the
current page.

Turing proposed one more conceptual simplification. Just as the states of a mind are finite,
what one can write on a single page is also finite. Accordingly he proposed representing the possible
one-page writings by a finite set of symbols \( a_0, a_2, \ldots, a_{r-1} \), which he called characters; he called
the set \( A = \{a_0, a_1, \ldots, a_{r-1}\} \) the alphabet. Now that “only” a single letter was being written on
each page, he called the pages cells. So an action consists of

1. reading the character written in the current cell;
2. possibly writing a new character in the current cell;
3. possibly moving one position to the left or right on the tape thereby changing which cell is
   the current cell;
4. possibly changing states.

A computational process can then be described by giving the particular rules to be followed for
the calculation at hand. To specify this fully, a few more details are needed. One begins with the
information on which one is calculating, the input, written on a leftmost segment of the tape, and
the mind (now called the finite control) in an initial state (conventionally \( q_0 \)). The computation
finishes with the result, the output, written on a leftmost segment of the tape, or sometimes right
after the input, and the finite control in a final state (conventionally \( q_f \)).

Turing’s thesis was that these machines captured everything that could be done computationally.
As we shall see, anything that can be done on a computer can in principle be done on a Turing
Machine, albeit rather inefficiently (the slowdown is by “only” a polynomial factor). But why study
them if they are horribly inefficient? The reason is that they are conceptually much simpler, and so
they are very useful in understanding what is computationally possible, both in terms of whether
something can be computed at all, and whether something can be computed efficiently (e.g. in
polynomial rather than exponential time).

There is one more detail to mention. As Turing did not want to put any fixed bound on the
length of a computation, although any computation to be useful would have to end eventually, he
imagined that the tape of cells was unending or infinite to the right, and the computation would use as much of the tape as needed.

Now we proceed to make this more formal.

4.1 Turing Machines — A Formal Definition

As illustrated in Figure 4.1, a Turing Machine $M$ comprises a finite control, an unbounded writable tape, and a read/write head. We sometimes use the abbreviation TM for a Turing Machine.

![Figure 4.1: The Components of a Turing Machine](image)

As with all the other machines we have seen, the finite control is simply a labeled, directed graph $G = (Q, E)$. $Q = \{q_0, q_1, \ldots, q_{k-1}\}$ are the possible states (or vertices) of $M$. We will specify the edges $E$ shortly.

Each cell of the tape holds a single character drawn from the alphabet $A = \{a_0, a_1, \ldots, a_{r-1}\}$, which includes the symbols $\mathcal{c}$ and $\mathcal{b}$ (the symbol for blank). It is convenient to let $a_0 = \mathcal{c}$ and $a_1 = \mathcal{b}$.

An edge $(p, q)$ is labeled by the triple $(a, b, m)$ where $p$ is the current state/vertex and $a$ is the character in the cell under the read/write head. The action taken is to write $b$ in the cell under the read/write head (replacing the $a$), move the read/write head one cell in direction $m$ (one of Left or Right, L or R for short), and update the finite control to go to vertex $q$. $M$ is deterministic if it has at most one possible move at each step and otherwise it is non-deterministic. In delta notation, $\delta(p, a) = \{(q_1, b_1, m_1), (q_2, b_2, m_2), \ldots, (q_l, b_l, m_l)\}$, where these are the $l$ possible actions of the Turing Machine when at vertex $p$ with an $a$ in the cell under the read/write head.

To keep the read/write head from trying to move off the left end of the tape, we require that only the character $\mathcal{c}$ can be written in the leftmost cell, and furthermore the only possible move when reading this cell is to move the read/write head to the right while leaving the $\mathcal{c}$ unchanged, i.e. $\delta(p, \mathcal{c}) = \{(\mathcal{c}, q, R)\}$, for some $q$ (see Figure 4.2). Also, to avoid confusing any other cell with the leftmost one, we prohibit the writing of $\mathcal{c}$ in any other cell.

![Figure 4.2: Moving from the left end of the tape](image)

The computation starts with the read/write head over the leftmost cell, and with $\mathcal{c}x\mathcal{b}\mathcal{b}\ldots$ on
the tape, for some \( x \in \Sigma^* \), where \( \Sigma \subseteq A \setminus \{c, b\} \). \( x \) is called the input to \( M \); it is written using an alphabet \( \Sigma \) which is a strict subset of the character set used by TM \( M \), and at the very least excludes characters \( c \) and \( b \).

A deterministic computation ends when \( M \) has no further move to make. There are two modes of computation. The first is to recognize an input, and the second is to compute a function \( f(x) \). As we will argue shortly, the first mode provides the same computational power as the second mode. First, we will explain how each form of computation is carried out. An input \( x \) is recognized if the computation ends with \( M \)'s finite control at a vertex (state) \( q \in F \subseteq Q \), where \( F \) is the set of recognizing vertices. While to compute a function \( f(x) \), \( M \) will write the string \( f(x) \) at the left end of the tape overwriting the input; the computation will end with \( cf(x)\$ \) at the left end of the tape, with the control being at a specified vertex \( q_f \). The purpose of the \$ \) is to mark the end of the output, which may be needed in the case that the tape has been written to the right of where \( f(x) \) is written. We require that \( x, f(x) \in \Sigma^* \), where \( \Sigma \subseteq A \setminus \{c, b, \$\} \).

We can also view function computation as a recognition problem. Suppose the input is a tuple \( (x, y) \) and it is recognized exactly if \( f(x) = y \). Clearly, if we could test the inputs \( (x, 0), (x, 1), (x, 2), \ldots \) in turn, stopping when \( (x, f(x)) \) was reached, we would in effect be computing \( f(x) \). So from the perspective of what can and cannot be computed, there is no meaningful difference between language recognition and function computation.

In order to describe exactly the information held by the Turing Machine at each step of its computation, we introduce the notion of a configuration. This consists of the vertex (or state) of the Turing Machine, the tape contents (up to but not including the all blank portion of the tape), and the position of the read/write head. We will write this as \( (q, uHv) \), where \( q \) is the current vertex, \( uvbb \ldots \) is the current tape contents, and the read/write head is over the leftmost character in \( v \), or if \( v = \lambda \), it is over the \( b \) immediately to the right of \( u \). Some authors use the more compact notation \( uqv \).

A computation consists of a sequence of configurations \( C_0 \vdash C_1 \vdash \ldots \vdash C_m \), where \( C_i \vdash C_{i+1} \) denotes that configuration \( C_{i+1} \) follows configuration \( C_i \) by a single step of the Turing Machine’s computation. We write \( C_i \vdash^* C_j \) if \( C_j \) can be reached from \( C_i \) in 0 or more steps of computation. \( C_0 = (q_0, Hc) \) is the initial configuration on input \( x \), and \( C_m = (q, uHv) \) is a recognizing configuration if \( q \in F \).

The language recognized by \( M \) is defined to be

\[
L(M) = \{ x \mid \text{there is a recognizing configuration } C_m \text{ such that } C_0 \vdash^* C_m \text{ and } C_0 = (q_0, Hc) \}.
\]

Formally, a Turing Machine \( M \) is specified by an 6-tuple \( M = (\Sigma, A, V, \text{start}, F, \delta) \), where \( A \) is the tape alphabet, \( \Sigma \subseteq A \setminus \{c, b\} \) is the input alphabet, \( V \) is the set of vertices (or states) in the finite control, \( F \subseteq V \) is the set of recognizing vertices, and \( \delta \) specifies the edges and their labels (i.e. what is read and the resulting action), and start is the start vertex.

We can always ensure that \( M \) has a single recognizing vertex and no move from the recognizing vertex if desired (see Exercise 4).

Writing Turing Machines in full detail is somewhat painstaking, so we limit ourselves to a few examples to build some intuition.
4.1. Turing Machines — A Formal Definition

Example 4.1.1. TM $M$ takes input $x \in \{0,1\}^+$, where $x$ is viewed as an integer written in binary. The output is the integer $x + 1$ also in binary. To make our task a little simpler, we will write $x$ in left to right order going from the least to the most significant bit. Thus, for example 6 is written as the binary string \(011\) \((0 \times 1 + 1 \times 2 + 1 \times 4)\).

In general, if $x = 1^k0y$, with $y \in \{0,1\}^*$, then $x + 1 = 0^k1y$; while if $x = 1^k$ then $y = 0^k1$.

This suggests a fairly simple implementation: read from left to right over the initial all 1s prefix of $x$, changing the 1s to 0s and change the next character, either a 0 or a 1, to a 1. The finite control is shown below.

Figure 4.3: Turing Machine that adds 1 to its input; the nodes at which the computation ends are shown as recognizing nodes.

4.1.1 Multi Tape Turing Machines

To facilitate the construction of more complex Turing Machines, we make them more versatile.

A multi-tape Turing Machine is very similar to the 1-tape Turing Machine we previously described, except that it can have several tapes — this is a fixed number, specified for each individual machine. The number of tapes does not depend on the input.

Suppose the Turing Machine at hand has $k$ tapes. Each tape will have its own read/write head. Each action will depend on the current vertex (state), and on the contents of the $k$ cells under the read/write heads. The move will rewrite the content of these cells (which might be the previous contents), shift each read/write head left or right or allow it to stay put as desired, and advance to a new vertex (which might be the same vertex). We denote the move of each read/write head by one of L, R, P for a move to the left, right, or staying put, respectively. (With multiple tapes it is no longer possible to simulate staying put by a move to the right followed by a move to the left, for we might want some R/W heads to stay put and others to move right, for example). As before each tape will have a $c$ as its leftmost character, with the rules w.r.t. $c$ unchanged.
Example 4.1.2. Input: $x\#p$
Output: Index in $x$ of the leftmost occurrence of $p$ in $x$, and 0 if there is no occurrence. i.e. if $x = x_1x_2 \ldots x_n$ and if $x_{i+1}x_{i+2} \ldots x_{i+m} = p$ is the first occurrence of $p$ in $x$ then output $i + 1$ in binary on tape 2; if there is no such $i$, output 0.

We use a 4-tape machine which proceeds as follows.

Step 1. Copy $p$ to Tape 3.
Step 2. for $i = 0$ to $n - m$ do
    if $p = x_{i+1}x_{i+2} \ldots x_{i+m}$
    then {write $i + 1$ on Tape 2
        and move to a recognizing vertex; i.e. end the computation}
    else write 0 on Tape 2

Next, we show that having multiple tapes provides no additional recognition power. Specifically, given a $k$-tape Turing Machine $M$, we show there is a 1-tape Turing Machine $	ilde{M}$ that recognizes the same language.

We use $2k$ tracks on $	ilde{M}$'s single tape to simulate $M$'s $k$ tapes. The tracks are organized in pairs, with each pair being used to hold the contents of one of $M$'s tapes and the position of its read/write head. Specifically, if the tape configuration is $(uHv)$, then the first track will store the string $uv$ at its left end, and the second track will store the string $(b)^{|u|}H(b)^{|v|}$ — the parentheses are present for clarity. (See Figure 4.4.) The $i$-th cell on $M$'s single tape stores the contents of the $i$-th cell on each of the $2k$ tracks. Thus, if we represent the tape as a semi-infinite array with $2k$ rows, each of its rows corresponds to a single track, each entry represents a single cell on a single track, while each column corresponds to a single one of $M$'s cells. See Figure 4.5.

$	ilde{M}$ will need a lot of vertices in its finite control to simulate $M$. We will view each of $	ilde{M}$'s vertices as a $2k + 2$ tuple. The second entry will hold $M$'s current state/vertex; each subsequent pair of entries will hold information about a single tape (such as the character under the read/write head, and the direction in which to move the read/write head). The first entry is used to specify $	ilde{M}$'s current action.

To simulate a single one of $M$'s moves or actions, $	ilde{M}$ begins with its read/write head on the cell storing $c$ at the left end of its tape. It then performs its first type of action, a sweep to the right across its tape; whenever it reads an $H$ on a track it records the aligned cell contents on the partner track; to carry out this recording, $	ilde{M}$ updates its state, that is it changes the current vertex so as to in effect record this information. After encountering $k$ $H$ symbols, one for each pair of tracks, $M$'s action is then carried out; this is the second type of action for $	ilde{M}$. It records the results of $M$'s action in $	ilde{M}$'s state — this entails another move to a new current vertex. To perform $M$'s action, $	ilde{M}$ sweeps its read/write head back to the left, and whenever it encounters an $H$ it simulates...
the relevant action by making appropriate changes on the pair of tracks corresponding to the tape which is being updated. Note that if $M$ moves a read/write head to the right, $\tilde{M}$ will need to back up one cell to the right in order to be able to write the $H$ in the correct position. This backing up is a third type of action.

If the non-blank contents on $M$'s longest tape cover $m$ cells, then simulating one step of $M$'s computation will take $\tilde{M}$ up to $2m + 2k$ steps (the $+2k$ term arises due to the possible need to back up one cell for each of $M$'s $k$ tapes, when simulating the action on that tape).

$\tilde{M}$ will recognize its input exactly if it reaches a vertex $v$ which corresponds to one of $M$'s recognizing vertices (i.e., the second entry in $v$'s tuple is one of $M$'s recognizing vertices).

**Comment.** Strictly speaking, $\tilde{M}$ begins with tape contents $czb\ldots$ which it rewrites to $(cH)^k x_1 b^{2k-1} x_2 b^{2k-1} \ldots x_n b^{2k-1}$, where $x = x_1 x_2 \ldots x_n$, and each sequence of $2k$ symbols in the rewritten tape contents denotes the contents of the $2k$ tracks in one of $\tilde{M}$'s cells.

We have shown:

**Lemma 4.1.3.** Let $M$ be a $k$-tape Turing Machine. Then there is a 1-tape Turing Machine $\tilde{M}$ such that $L(\tilde{M}) = L(M)$.

### 4.1.2 The Universal Turing Machine

So far we have needed to create a separate Turing Machine for each computation we wish to perform. But now we will show that a single general purpose Turing Machine, akin to a computer, suffices. This Universal Turing Machine $U$ will take two inputs: A description of a Turing Machine $M$ and an input $x$ to $M$. It will then simulate $M$'s computation on input $x$. We provide a description of how to simulate deterministic Turing Machines. The extension to non-deterministic machines is left as an exercise.

We start by describing a general purpose 3-tape Turing Machine $\tilde{U}$, which we can then simulate using a 1-tape Turing Machine $U$, by applying Lemma 4.1.3. Tape 1 holds $\tilde{U}$'s input. Tape 2 will hold a copy of $M$'s tape. Tape 3 will hold $M$'s current vertex/state. $\tilde{U}$ will then repeatedly simulate the next step of $M$'s computation.

Since the possible $M$ that $\tilde{U}$ needs to be able to simulate can have arbitrarily large finite controls (graphs) and arbitrarily large tape alphabets, $\tilde{U}$ cannot use $M$'s alphabet to carry out the
simulation, nor can it store $M$’s finite control in its own possibly smaller finite control. Instead, $\bar{U}$ will use a single fixed alphabet which it uses to encode (i.e. write) a precise description of $M$’s finite control and tape contents. There are many ways of doing this; we proceed as follows. We can write a Turing Machine $M$ as a tuple $M = (\Sigma, A, V, \text{start}, F, \delta)$ where $\Sigma$ is the input alphabet, $A$ is the tape alphabet, and $\Sigma \subset A - \{c, b\}$, $V$ is its set of vertices or states, start $\in V$ is the start vertex, $F \subseteq V$ is the set of recognizing vertices, and $\delta$ is the transition function.

We will describe how to write out the description of a deterministic TM $M$ in full detail. Let $A = \{a_0, a_1, \ldots, a_{r-1}\}$, and recall that $c = a_0, b = a_1$. We will represent $a_i$ by the $\lceil \log_2 r \rceil$-bit binary string that denotes the integer $i$. $A$ can be written using the following 5-character alphabet: 0, 1, (, ), and “,” (i.e. the comma itself), in the form (binary($a_1$), binary($a_2$), ..., binary($a_{r-1}$)), where binary($a_i$) denotes the log-$r$-bit representation of $a_i$ in binary. $\Sigma$ can be written in the same way. Let $V = \{q_0, q_1, \ldots, q_{s-1}\}$. Then we will represent $q_i$ by the $\lceil \log_2 s \rceil$-bit binary string that denotes the integer $j$. Again, we can use the same 5-character encoding for $V$. $F$ can be written in the same way, while the vertex start just requires the listing of a single vertex in binary. We need to ensure that $F \subseteq V$ and start $\in V$, of course. Each entry in $\delta$ comprises a 5-tuple $(p, q, a, b, m)$, where $p$ denotes the current vertex, $a$ the character in the cell under the read/write head, $q$ the vertex being moved to, $b$ the character being written, and $m$ the move being made (L or R). This tuple can be written as a sequence of five binary strings separated by commas and enclosed in parentheses. $M$ itself can be written as the comma separated encoding of its 6 constituent elements, enclosed in parentheses.

Thus we obtain an encoding of $M$ as a string using 5 characters. In turn, this can be rewritten as a binary string by encoding these 5 characters in binary, using 3 bits for each character. Thus each 1-tape deterministic Turing Machine can be described by a distinct binary string.

It is also straightforward to write an input $x$ to $M$ in binary using the encoding for alphabet $A$: this requires $3 \lceil \log_2 r \rceil$ bits per character in $x$.

The input to $\bar{U}$ is a string $s$ which is supposed to be the encoding of a Turing Machine $M$ followed by the encoding of an input $x$ to $M$. So first stage of $\bar{U}$’s computation will be to check that its input does have this form. This requires many instances of pattern matching. For example, $\bar{U}$ needs to confirm that the encodings for all the characters in $A$ have the same length and that they are all distinct. It needs to confirm that $\Sigma \subset A$ and $a_0, a_1 \in A - \Sigma$. In more detail, to confirm $c = a_0 \in A - \Sigma$, $\bar{U}$ confirms that $a_0$ does not match any of the characters in $\Sigma$. Similar checks need to be made for $V, F$ and start, and for $\delta$. $\bar{U}$ also needs to check that $M$ cannot make any illegal moves: the only time it can write $c$ is when it is reading $c$ and then it must move to the right. We leave these further painstaking but straightforward details to the reader.

Now $\bar{U}$ is ready to carry out the simulation. To this end, $\bar{U}$ will use three tapes. Tape 1 holds its input. Tape 2 is used to maintain an encoding in binary of the characters on $M$’s single tape; to facilitate identifying the start and end of each character encoding, they will be separated by a third character, # say (it is allowed to use a constant number of additional characters beyond 0,1, its $c$ and its $b$). The read/write head for Tape 2 will be used to keep track of the position of $M$’s read/write head. Finally, $\bar{U}$ stores the encoding of $M$’s current vertex on Tape 3. To perform one step of simulation, $\bar{U}$ has to find the 5-tuple in the encoding of $\delta$ that represents the current legal move, if any. Thus, if $a$ is under $M$’s read/write head and $p$ is $M$’s current vertex, $\bar{U}$ has to find the at most one tuple that contains $p$ as its first entry and $a$ as its third entry. This can be done by matching the contents of Tape 3 and the encoded character under the read/write head for the second tape with the encoding of $\delta$ on Tape 1.
4.2. PROGRAMS AND TURING MACHINES ARE EQUIVALENT

If no match is found, then the simulation ends. A transition to \( \tilde{U} \)'s recognizing vertex occurs if \( M \)'s current vertex is in \( F \). Again, to make this determination requires a matching process.

On the other hand, if a match with tuple \((p, q, a, b, m)\) is found then the contents of \( \tilde{U} \)'s Tape 2 and Tape 3 are updated. On Tape 3 \( p \) is overwritten by \( q \), on Tape 2 \( a \) is overwritten by \( b \), and finally on Tape 2, the read/write head is moved one encoded character in the direction \( m \) (here the \# separators make this task easy). There is one more detail to add regarding this move: a move to the right may reach a previously unwritten portion of \( M \)'s tape, in which case \( \tilde{U} \) has to write the encoding of \( \tilde{U} \)'s blank, followed by a \# at the current right end of the written portion of Tape 2. Once this is all done, \( M \) proceeds to the next step of simulation.

**Theorem 4.1.4** (Turing, 1936). There is a 1-tape Universal Turing Machine \( U \) that given an input \( w = \langle M, x \rangle \) recognizes its input \( w \) exactly if \( M \), a 1-tape Turing Machine, recognizes its input \( x \).

**Proof.** The above description showed there is a 3-tape Universal Turing Machine \( \tilde{U} \). But by Lemma 4.1.3, \( \tilde{U} \) can be simulated by a 1-tape Turing Machine \( U \) which recognizes the same language. \( \square \)

The Universal Turing Machine can be thought of as either a general purpose computer or as a compiler. Historically, its significance is that it showed that a general purpose computing machine could carry out all the tasks being done by individual specialized devices. It is worth noting that this has already been realized a century earlier in Babbage's work.

Another important point is that \( U \)'s control has a fixed size. What gives it its computational power is its unbounded memory (in the form of its tape).

4.2 Programs and Turing Machines are equivalent

Next, we wish to argue that the computational power of programs and Turing Machines are the same. To this end, we consider a very simple computing model and programming language. In this language, a program consists of a sequence of incrementally numbered instructions, i.e., instruction 1, instruction 2, etc. Our programs run on a computer with a memory \( M[0, m-1] \) of \( w \)-bit integer values (we discuss the issue of integer size below). For readability we allow variable names such as \( x, y, z \), but each variable corresponds to a specific location in \( M \). There are eight different instructions in all, which involve variables and constants \( c \) which are integer values such as 0, 1, -5, etc.

1. \( x \leftarrow c \); (assign \( x \) a value specified in the program).
2. \( x \leftarrow M[c] \); (assign \( x \) the value stored in \( M[c] \)).
3. \( x \leftarrow M[y] \); (assign \( x \) the value stored in \( M[y] \)).
4. \( M[c] \leftarrow x \); (store \( M[c] \) the value of variable \( x \)).
5. \( M[y] \leftarrow x \); (store \( M[y] \) the value of variable \( x \)).
6. \( x \leftarrow y + z \); (store \( x \) the value \( y + z \)).
7. \( x \leftarrow -y \); (store \( x \) the value \(-y\)).
8. if $x \geq 0$ then go to $l$; (see below).

A program executes starting at its first instruction, and performs the instructions in the program sequentially except when it performs an instruction “if $x \geq 0$ then go to $l$”, and then if $x \geq 0$ it instead executes instruction $l$ next. When it seeks to execute a non-existent instruction subsequent to the last instruction, the computation terminates.

Note that there is no output, and the input is implicit: it is provided via the initial values in the array $M$. We therefore define recognition of an input $v$ by whether the program’s computation terminates on input $v$ or not.

For the program to be able to access a memory location $l \leq m - 1$ it must be able to reference it, and if $l$ is not a constant named in the program then this location can be accessed only via a reference of the form $M[y]$, where $y$ is a variable, which is itself stored in some memory location. This means that $0 \leq y \leq m - 1$, and consequently the memory location storing $y$, and by symmetry each memory location, must be able to store $\lceil \log m \rceil$-bit numbers. Consequently, we will need $w \geq \log m$. Indeed, this is a feature of the RAM model, the standard model used in analysis of algorithms, which requires $w = \Omega(\log m)$, where $m$ is the size of the memory being used.

To enable simulation by a Turing Machine we assume the input to the Turing Machine comprises a pair $(w, x)$, where $w$ indicates the size of integers being stored in each memory location, namely up to $m - 1 = 2^w - 1$, and $x$ indicates the contents of the memory in locations $M[0 : m - 1]$. As we are storing both positive and negative integers, a further bit is required to indicate the sign of each integer.

Also, we need to specify what happens if the computation ever seeks to use more than $w$ bits for an integer: we can have the program increase its counter to a value one larger than the number of lines in the program, which causes the computation to end; i.e. the input is recognized. (It might be more appealing if the input were not recognized in these circumstances; this could be achieved by branching to an infinite loop, but this seems less simple, which explains the choice we made.)

We now describe a Turing Machine simulation of a program $P$ on input $(w, x)$. Tape 1 is used to hold the input $(w, x)$. Tape 2 will be used to store the contents of $M[0, m - 1]$, as a series of pairs of $w + 1$ bit numbers separated by # markers; the first number in each pair is the index of the location, and for simplicity the pairs are stored in increasing order based on the location index; the second number is the value stored in the location. Tape 3 is used to store the number of the next instruction. Three more tapes are used to carry out the instructions. We explain how to simulate a few of the instructions; the others are handled similarly.

$M[y] \leftarrow x$: The value stored in $x$, i.e. in the memory location holding variable $x$, is copied to Tape 4; similarly the value stored in $y$ is copied to Tape 5. Now, $x$, the value on Tape 4, is copied to memory location $M[y]$, the location being provided by the contents of Tape 5. Finally, the instruction counter is incremented by 1.

$x \leftarrow y + z$: The values stored in $y$ and $z$ are copied to Tapes 4 and 5, respectively. They are then added together, with the result being written on Tape 6. This value is then copied to the memory location for variable $x$. Finally, the instruction counter is incremented by 1.

if $x \geq 0$ then go to $l$: The value stored in $x$ is copied to Tape 4. If it non-negative, the instruction counted is reset to $l$ and otherwise it is incremented by 1.

Once again, we can simulate the 6-tape simulating Turing Machine with a 1-tape Turing Machine.

We have shown:
Theorem 4.2.1. Every language \( L \) recognized by a program \( P \) is recognized by a 1-tape Turing Machine.

Generality of the Programming Language  It is straightforward to implement standard programming language instructions using our limited instruction set, such as logic operations, the “if then else” command, while loops, procedure calls, multiplication, etc.

Also it should be clear that we could write a program to simulate a Turing Machine. In particular, we could write a program \( P_U \) that simulates the Universal Turing Machine \( U \), and thus \( P_U \) can be thought of as a universal program. It is a program akin to a compiler; that is it takes two inputs: a Turing Machine \( M \) (or equivalently a program \( P \)) and an input \( x \) for \( M \) and it simulates the execution of \( M(x) \).

Exercises

1. Show that a binary alphabet suffices. That is, given Turing Machine \( M \) with alphabet \( A \), create a new Turing Machine \( M_b \) with a binary alphabet that carries out the same computation as \( M \). Furthermore, \( x \in L(M) \) exactly if \( b(x) \in L(M_b) \), where \( b(x) \) denotes the encoding of \( x \) in binary. Hint. Encode each character of \( A \) using \( \log A \) binary symbols, and make \( M_b \)'s finite control sufficiently larger than that of \( M \) so that it can “read” and “write” a character of \( M \).

2. This exercise shows how to count out a number larger than can be kept in mind.

Design a Turing Machine with a 4-character alphabet \( A = \{0, 1, *, b\} \). Suppose the input is a number \( n \) written in binary at the left end of the tape; at this point the rest of the tape is blank, i.e. every other cell stores a \( b \). Design a Turing Machine to write \( n * \) characters after the input number. You may alter the input number as the computation proceeds. Also assume you are given a procedure (a collection of vertices and actions) that subtracts 1 from a positive number written in consecutive cells at the left end of the tape. Finally, describe the Turing Machine in reasonably high level terms such as: move to the right while reading cells with a ‘*’ until a cell without a ‘*’ is reached.

3. Describe a 3-tape Turing Machine that adds two binary numbers \( x \) and \( y \). They are written at the left ends of Tapes 1 and 2, respectively, in the form \( c x \) and \( c y \). The result, the binary number \( z = x + y \), is to be written on Tape 3, in the form \( c z \). All three numbers are written, in left to right order, from least significant to most significant bit.

4. Let \( M \) be a 1-tape Turing Machine.

   i. Describe a 1-tape Turing Machine \( M' \) which has a single recognizing vertex and such that \( L(M') = L(M) \).

   ii. Describe a 1-tape Turing Machine \( M'' \) which has a single recognizing vertex, no move from the recognizing vertex, and such that \( L(M'') = L(M) \).

5. Describe how to modify the Universal Turing Machine described in this chapter so that it can simulate non-deterministic Turing Machines.
6. Let $N$ be a non-deterministic 1-tape Turing Machine. Describe a deterministic 1-tape Turing Machine $M$ such that $L(M) = L(N)$.

Hint. To recognize an input $x$, $N$ needs to have a recognizing computation. Thus $M$’s task on input $x$ is to determine if $N$ has a recognizing computation on this input. $M$ will need to explore the possible computations in a breadth first fashion, for some non-recognizing computations may have infinite length.

7. Consider the 8-instruction programming language given in the text. Show how to implement the following instructions in this language.

i. **go to** $l$, meaning execute the statement with label $l$ next.

ii. $z = x \land y$, where $x$, $y$ and $z$ are variables that can take the values 0 or 1. You may use the instruction **go to** $l$.

iii. $z = x \lor y$, where $x$, $y$ and $z$ are variables that can take the values 0 or 1. You may use the instruction **go to** $l$.

iv. $z = \neg y$, where $x$ and $y$ are variables that can take the values 0 or 1. You may use the instruction **go to** $l$.

v. if $x \leq 0$ then **go to** $l$.

vi. if $x = 0$ then **go to** $l$.

vii. The “if $x = 0$ then {first sequence of statements} else {second sequence of statements}” construct, where the sequences of statements are given.

viii. The construct “while $x = 0$ do {sequence of statements}, where the sequence of statements is given. You may use the instruction if $x = 0$ then **go to** $l$.

ix. Multiplication: the input comprises the variables $x$ and $y$ storing positive integers and the task is to compute $z = x \times y$.

Hint: Implement the standard long multiplication algorithm.