1 Definition

Finite automata do not suffice as a fully general-purpose model of computation, since as we have seen, there are simple functions that are intuitively computable but which cannot be computed by any DFA. The thing DFAs lack is memory—all a DFA can remember is what state it’s currently in (and not how it got there). To obtain a general-purpose model, which is capable of expressing any computation that could be programmed in your favorite programming language, we augment the DFA model with unrestricted access to an unlimited amount of memory. This model is called Turing machines after its inventor, and the precise definition is as follows.

Syntax. A Turing machine (TM) consists of the following components:

- A finite set of states $S$, including a start state and a halt state.
- An infinite tape of cells.
- A finite set of symbols $A$ called the alphabet (which must include a special blank symbol $\cdot$).
  In each step of the computation, each cell of the tape will hold one symbol from the alphabet.
- A tape head (for reading and writing), which is positioned at a cell of the tape in each step.
- A transition function $F: A \times (S \setminus \{q_{\text{halt}}\}) \rightarrow A \times S \times \{+1, -1\}$.

Semantics. Initially:

- The state is the start state.
- The tape contains the input string (with infinitely many blanks to the left and to the right).
- The tape head is at the leftmost cell of the input.

In each step of the computation, if the symbol under the tape head is $a \in A$ (the TM is “reading $a$”), the current state is $s \in S$, and the transition function evaluates to $F(a, s) = (a', s', b)$, then the TM does the following:

- Write $a'$ under the tape head (overwriting the $a$).
- Go to state $s'$.
- Add $b$ to the tape head position (move right one cell if $b = +1$, move left one cell if $b = -1$).

If and when the computation reaches the halt state, the output is considered to be the contents of the tape at that time. Alternatively, if we only care about solving decision problems, then we could have two halt states, one of which accepts (outputs 1) and the other of which rejects (outputs 0).

State diagrams. A Turing machine can be drawn as a state diagram much like a DFA, but with more complicated transition labels. The transition $F(a, s) = (a', s', b)$, which represents reading $a$, writing $a'$, going from state $s$ to state $s'$, and moving the head in direction $b$, is pictorially represented as:

$$
\begin{array}{c}
S \\
\rightarrow a \rightarrow a', b \\
S'
\end{array}
$$
2 Examples

Example 1. To illustrate the definition, consider the following TM. (We will see shortly that it computes a natural function.)

Here $S = \{q_1, q_2, q_{halt}\}$, with $q_1$ being the start state and $q_{halt}$ being the halt state. The alphabet is $A = \{0, 1, \_\}$, and the input must be written with just 0’s and 1’s since the blanks indicate where the input starts and ends on the tape. The transition function is:

<table>
<thead>
<tr>
<th>$a$</th>
<th>$s$</th>
<th>$F(a, s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q_1$</td>
<td>$(0, q_1, +1)$</td>
</tr>
<tr>
<td>1</td>
<td>$q_1$</td>
<td>$(1, q_1, +1)$</td>
</tr>
<tr>
<td>_</td>
<td>$q_1$</td>
<td>$(_, q_2, -1)$</td>
</tr>
<tr>
<td>0</td>
<td>$q_2$</td>
<td>$(1, q_{halt}, -1)$</td>
</tr>
<tr>
<td>1</td>
<td>$q_2$</td>
<td>$(0, q_2, -1)$</td>
</tr>
<tr>
<td>_</td>
<td>$q_2$</td>
<td>$(1, q_{halt}, -1)$</td>
</tr>
</tbody>
</table>

For example, the first row of the table means that if the TM is reading 0 while in state $q_1$, then it writes 0 (so the tape contents don’t change in this step), stays in state $q_1$, and moves the head one cell to the right. The fifth row means that if the TM is reading 1 while in state $q_2$, then it overwrites the 1 with a 0, stays in state $q_2$, and moves the head one cell to the left. For the transitions that enter $q_{halt}$, the direction the tape head moves is irrelevant, and we arbitrarily chose $-1$ in this example.

We can see how the TM computes, for example, on input 1011 by writing the sequence of configurations it goes through. A configuration consists of the current state, tape contents, and tape head location (which we illustrate with an arrow under the corresponding cell).

<table>
<thead>
<tr>
<th>step</th>
<th>state</th>
<th>tape</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>$q_1$</td>
<td>\ldots _ _ 1 0 1 1 _ \ldots</td>
</tr>
<tr>
<td>1</td>
<td>$q_1$</td>
<td>\ldots _ _ 1 0 1 1 _ \ldots</td>
</tr>
<tr>
<td>2</td>
<td>$q_1$</td>
<td>\ldots _ _ 1 0 1 1 _ \ldots</td>
</tr>
<tr>
<td>3</td>
<td>$q_1$</td>
<td>\ldots _ _ 1 0 1 1 _ \ldots</td>
</tr>
<tr>
<td>4</td>
<td>$q_1$</td>
<td>\ldots _ _ 1 0 1 1 _ \ldots</td>
</tr>
<tr>
<td>5</td>
<td>$q_2$</td>
<td>\ldots _ _ 1 0 1 1 _ \ldots</td>
</tr>
<tr>
<td>6</td>
<td>$q_2$</td>
<td>\ldots _ _ 1 0 1 0 _ \ldots</td>
</tr>
<tr>
<td>7</td>
<td>$q_2$</td>
<td>\ldots _ _ 1 0 0 0 _ \ldots</td>
</tr>
<tr>
<td>terminal</td>
<td>$q_{halt}$</td>
<td>\ldots _ _ 1 1 0 0 _ \ldots</td>
</tr>
</tbody>
</table>
We can see that in general, the TM reads through the input left-to-right, staying in $q_1$ and not changing the tape contents, until it finds the blank just past the input, at which time it transitions to $q_2$ and moves the head back onto the rightmost input symbol. Then it moves left, flipping 1’s to 0’s until it finds a 0 or a blank, which it then flips to 1 and halts. It will find a blank while moving left if the input had only 1’s to begin with. Thus, for example, input 0 gets transformed to output 1, and 11 gets transformed to 100, and 01010111 gets transformed to 01011000. It should now be clear what the TM is accomplishing: it interprets the input as a binary integer, and increments it! (The above example execution incremented eleven to twelve.)

**Example 2.** Let’s design a TM that takes a bit string and decides whether it’s a palindrome (i.e., looks the same forward and backward). The main idea is to check that the first and last bits match (rejecting if not), then check that the second and second-to-last bits match, and so on, by shuttling the tape head back and forth between the ends of the input and using the states to “remember” the current bit that needs to be matched. When the first and last bits are matched, the TM will also erase them (overwrite them with blanks) so that for the next round it can find the second and second-to-last bits (since when it hits a blank, the TM knows it has gone one step too far). Then we erase those two matched bits, and so on, until no bits remain, at which time the TM accepts. We don’t care what’s left on the tape when the TM halts, because the output is just whether it reaches the accept state or the reject state.

Here, $q_1$ means we are reading the leftmost bit (of what remains of the input). If it’s 0, we remember this by going to $q_2$, and if it’s 1 we remember this by going to $q_4$. In either case, we erase this bit and read past the rest of the bits until we hit a blank and back up, entering $q_3$ or $q_5$ respectively. Now we are reading the rightmost bit, so we check whether it matches the bit we’re remembering: if we read a 0 in $q_3$ or read a 1 in $q_5$, then the bits match, so we erase the last bit and use $q_6$ to “reset” the head to the left end (of what remains of the input) by reading through any bits until we hit a blank; going to $q_1$ then starts the next round. If we read the wrong bit in
q_3 or q_5, then we know for sure the input was not a palindrome, so we enter the reject state. If we read a blank in q_1, then the input was an even-length palindrome: in the last round there were two remaining matching bits, the first of which was erased on the way to q_2 or q_4, and the second of which was erased on the way to q_6. If we read a blank in q_3 or q_5, then the input was an odd-length palindrome: in the last round there was a single remaining bit (the middle of the original input), which was erased on the way to q_2 or q_4.

This TM takes \( \Theta(n^2) \) steps on inputs of length \( n \). This goes against the intuition that palindromes should be decidable in \( O(n) \) time: \( O(1) \) time to read the first and last bits and make sure they match, \( O(1) \) time to read the second and second-to-last bits and make sure they match, and so on. However, this is just an artifact of the sequential-access nature of the model: since a TM’s head can only move one cell at a time, it must waste a lot of time moving between distant parts of the input. For example, it takes linear time just to go from the first bit to the last bit, even though this ought to be a constant-time operation in a model with random access to the memory. Nevertheless, this doesn’t matter for the purpose of computability, since (as we discuss later) anything computable in a reasonable random-access model can also be computed (albeit somewhat more slowly) by a sequential-access TM.

**Example 3.** TMs are very tedious to describe explicitly, for all but the simplest computations. Thus, it is often acceptable to give a higher-level description of how a TM operates. For example, let’s design a TM that decides whether an input string from \( \{0, 1, \#\}^* \) is of the form \( x\#y\#z \) where \( x = y = z \in \{0, 1\}^* \). Deciding whether two bit strings are equal is similar to deciding whether a bit string is a palindrome (i.e., whether its left half equals the reverse of its right half), which we saw how to do in the previous example. We could check whether \( x = y \) by shuttling the tape head back and forth, checking that the first bits match and erasing them, then checking that the second bits match and erasing them, and so on. Then we’d like to check whether \( y = z \); however, we just foolishly erased \( x \) and \( y \) and no longer have access to them. Instead, we’d like to check whether \( x = y \) without destroying \( y \). The point of the erasing was just so we could always find where we left off in each string, so we can employ an alternative trick to accomplish this without erasing \( y \): just replace each bit of \( y \) with a new alphabet symbol which is a “marked” version of that bit—the marks allow us to find where we left off (the next bit to be checked comes after the last marked bit), but we can also recall what \( y \) is by ignoring the marks. More precisely, we create symbols \( \overline{0} \) (“marked 0” and “marked 1”) so the tape alphabet is now \( \{0, 1, \#, \overline{0}, \overline{1}\} \), and we can clear the marks (overwrite any \( \overline{0} \) with 0 and any \( \overline{1} \) with 1) before comparing \( y \) and \( z \). This is the intuition for the design of the TM. Drawing a full state diagram would be kind of pedantic, so we summarize with a high-level description:

1. Scan from left to right, making sure the input has exactly two \( \# \)'s (hence is of the form \( x\#y\#z \) for some \( x, y, z \in \{0, 1\}^* \)), rejecting if not, then return the tape head to the left end.
2. Check that \( x = y \) by repeating the following:
   2.a. If the current symbol is a bit:
      2.a.i. Mark the current bit and remember its value.
      2.a.ii. Scan right past the first \( \# \) and all marked bits following it.
      2.a.iii. If the current symbol is \( \# \), reject (since \( |x| > |y| \)).
      2.a.iv. Else if the current bit doesn’t match the remembered one, reject.
      2.a.v. Else mark the current bit and forget the remembered one.
      2.a.vi. Scan left past the \( \# \) to find the leftmost unmarked symbol.
2.b. Else the current symbol is #:
   2.b.i. Scan right past all marked bits.
   2.b.ii. If the current symbol is a bit, reject (since $|x| < |y|$).
   2.b.iii. Else scan left, clearing all the marks and putting the head on the first bit of $y$.
   2.b.iv. Break out of the loop.
3. Perform a loop similar to step 2, to check that $y = z$.
4. Accept.

3 Variants

The Church–Turing Thesis is the claim that any computation that can be carried out by a computing device (whether it’s digital, mechanical, human, or any other variety) can also be carried out by a Turing machine. This is not a theorem, because it merely states that the abstract model of Turing machines provides the “right” definition of algorithms or programs. This hypothesis forms the philosophical underpinning of the field of computer science, and is established by consensus of the community. Evidence in support of the Church–Turing Thesis comes in the form of observations that any reasonable model of computation that people have devised can be straightforwardly simulated by TMs. For starters, we can take a look at variants and generalizations of TMs, and see that they are no more powerful than the original definition.

For example, if we generalize the definition of TMs by allowing “stay put” moves ($b = 0$) in addition to right ($b = +1$) and left ($b = -1$) moves, this doesn’t add any power to the model since a stay put move can be simulated by a right move (that performs the desired write) followed by a left move (that doesn’t change the tape symbol).

As a somewhat less trivial example, a TM with $k > 1$ tapes (and one head per tape, which can move independently of each other) can be simulated by a single-tape TM: We can interpret the cells on the single tape as partitioned into groups of $k$ cells, the $i$th of which corresponds to a cell on the $i$th tape of the TM we’re simulating. We can use $k$ “marked symbols” (introduced in Example 3) to indicate the locations of the $k$ tape heads. Thus, a single iteration of the simulation consists of sweeping across the tape to find the $k$ marks, remembering the corresponding symbols, then sweeping back again to update the symbols and mark locations according to the $k$-tape TM’s transition function (assuming all the $k$ writes and head movements can depend on all the $k$ values that are read).

Finally, any doubts about whether TMs are a fully general-purpose model should be removed by considering that TMs can simulate any program written in a standard assembly language: We may partition the tape into a small area containing the contents of all the registers, and an unbounded area that correspond to main memory. The assembly instructions are hardwired into the transition function. ALU-type operations can be performed directly on the registers (see Example 1 for the implementation of a particularly simple arithmetic instruction). To perform (random-access) loads and stores, the desired memory location can be found by alternately advancing a “mark” and decrementing a copy of the address until it hits 0. Conditional branches can be performed by transitioning to a particular state after reading some register.