Lecture 12

The Turing machine model of computation

For most of the remainder of the course we will study the *Turing machine* model of computation, named after Alan Turing (1912–1954) who proposed the model in 1936. Turing machines are intended to provide a simple mathematical abstraction of general computations. As we study this model, it may help you to keep this thought in the back of your mind.

The idea that Turing machine computations are representative of a fully general computational model is called the *Church–Turing thesis*. Here is one statement of this thesis (although it is the idea rather than the exact choice of words that is important):

Church–Turing thesis: Any function that can be computed by a mechanical process can be computed by a Turing machine.

This is not a mathematical statement that can be proved or disproved—if you wanted to try to prove a statement along these lines, the first thing you would most likely do is look for a mathematical definition of what it means for a function to be "computed by a mechanical process," and this is precisely what the Turing machine model was intended to provide.

While people have actually built machines that behave like Turing machines, this is mostly a recreational activity—the Turing machine model was never intended to serve as a practical device for performing computations. Rather, it was intended to provide a rigorous mathematical foundation for reasoning about computation, and it has served this purpose very well since its invention.

It is worth mentioning that there are other mathematical models that play a similar role to Turing machines, as a way of formalizing the notion of computation. For instance, Alonzo Church proposed λ-calculus a short time before Turing proposed what we now call the Turing machine, and a sketch of a proof of the

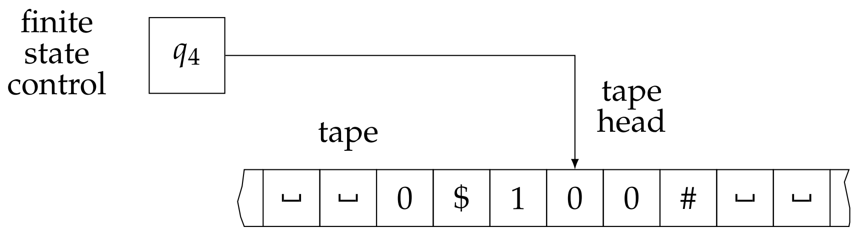


Figure 12.1: An illustration of the three components of a Turing machine: the finite state control, the tape head, and the tape. The tape is infinite in both directions (although it appears that this Turing machine's tape was torn at both ends to fit it into the figure).

equivalence of the two models appears in Turing's 1936 paper. Others, including Kurt Gödel, Stephen Kleene, and Emil Post, also contributed important ideas in the development of these notions. It is the case, however, that the Turing machine model provides a particularly clean and appealing way of formalizing computation, which is a reasons why this model is often the one that is preferred.

12.1 Turing machine components and definitions

We will begin with an informal description of the Turing machine model before stating the formal definition. There are three components of a Turing machine:

1. The *finite state control*. This component is in one of a finite number of states at each instant, and is connected to the tape head component.
2. The *tape head*. This component scans one of the tape squares of the tape at each instant, and is connected to the finite state control. It can read and write symbols from/onto the tape, and it can move left and right along the tape.
3. The *tape*. This component consists of an infinite number of tape squares, each of which can store one of a finite number of tape symbols at each instant. The tape is infinite both to the left and to the right.

Figure 12.1 illustrates these three components and the way they are connected.

The idea is that the action of a Turing machine at each instant is determined by the state of the finite state control together with just the one symbol that is stored by the tape square that the tape head is currently reading. Thus, the action is determined by a finite number of possible alternatives: one action for each state/symbol pair. Depending on the state and the symbol being scanned, the action that the machine is to perform may involve changing the state of the finite state control, changing the symbol on the tape square being scanned, and/or moving the tape head to the left or right. Once this action is performed, the machine will again have some state for its finite state control and will be reading some symbol on its tape, and the process continues. One may consider both deterministic and nondeterministic variants of the Turing machine model, but our main focus will be on the deterministic version of this model.

We also give Turing machines an opportunity to stop the computational process and produce an output. The possibility of producing a string as an output will be discussed later, but for now we'll concentrate just on *accepting* or *rejecting*, as we did for DFAs (for instance). To do this, we require that there are two special states of the finite state control: an *accept* state q_{acc} and a *reject* state q_{rej}. If the machine enters one of these two states, the computation immediately stops and *accepts* or *rejects* accordingly. We do not allow the possibility that q_{acc} and q_{rej} are the same state—these must be distinct states.

When a Turing machine begins a computation, an input string is written on its tape, and every other tape square is initialized to a special *blank* symbol (which may not be included in the input alphabet). Naturally, we need an actual symbol to represent the blank symbol in these notes, and we will use the symbol $\sqcup$ for this purpose. More generally, we will allow the possible symbols written on the tape to include other non-input symbols in addition to the blank symbol, as it is sometimes convenient to allow this possibility.

Formal definition of DTMs

With the informal description of Turing machines from above in mind, we will now proceed to the formal definition.

Definition 12.1. A *deterministic Turing machine* (or DTM, for short) is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \tag{12.1}$$

where Q is a finite and nonempty set of *states*; Σ is an alphabet, called the *input alphabet*, which may not include the blank symbol $\sqcup$; Γ is an alphabet, called the *tape alphabet*, which must satisfy $\Sigma \cup \{\sqcup\} \subseteq \Gamma$; δ is a transition function having the form

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}; \tag{12.2}$$

$q_0 \in Q$ is the *initial state*; and $q_{\text{acc}}, q_{\text{rej}} \in Q$ are the *accept* and *reject* states, which satisfy $q_{\text{acc}} \neq q_{\text{rej}}$.

The interpretation of the transition function is as follows. Suppose the DTM is currently in a state $q \in Q$, the symbol stored in the tape square being scanned by

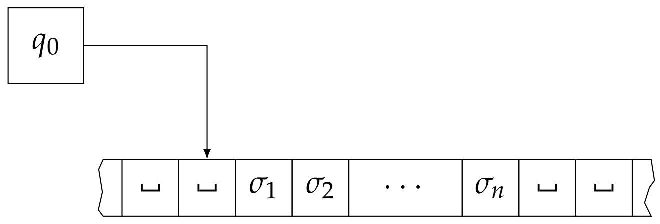


Figure 12.2: The initial configuration of a DTM when run on input $w = \sigma_1\sigma_2 \cdots \sigma_n$.

the tape head is σ, and it holds that $\delta(q, \sigma) = (r, \tau, D)$ for $D \in \{\leftarrow, \rightarrow\}$. The action performed by the DTM is then to (i) change state to r, (ii) overwrite the contents of the tape square being scanned by the tape head with τ, and (iii) move the tape head in direction D (either left or right). In the case that the state is q_{acc} or q_{rej}, the transition function does not specify an action, because we assume that the DTM stops once it reaches one of these two states.

12.2 Turing machine computations

If we have a DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, and we wish to consider its operation on some input string $w \in \Sigma^*$, we assume that it is started with its components initialized as illustrated in Figure 12.2. That is, the input string is written on the tape, one symbol per square, with each other tape square containing the blank symbol, and with the tape head scanning the tape square immediately to the left of the first input symbol. (In the case that the input string is ε, all of the tape squares start out storing blanks.)

Once the initial arrangement of the DTM is set up, the DTM begins taking *steps*, as determined by the transition function δ in the manner suggested above. So long as the DTM does not enter one of the two states q_{acc} or q_{rej}, the computation continues. If the DTM eventually enters the state q_{acc}, it *accepts* the input string, and if it eventually enters the state q_{rej}, it *rejects* the input string. Notice that there are three possible alternatives for a DTM M on a given input string w:

1. M accepts w.
2. M rejects w.
3. M runs forever on input w.

In some cases one can design a particular DTM so that the third alternative does not occur, but in general this could be what happens (assuming that nothing external to the DTM interrupts the computation).

Representing configurations of DTMs

In order to speak more precisely about Turing machines and state a formal definition concerning their behavior, we will require a bit more terminology. When we speak of a *configuration* of a DTM, we are speaking of a description of all of the Turing machine's components at some instant:

1. the *state* of the finite state control,
2. the *contents* of the entire tape, and
3. the *tape head position* on the tape.

Rather than drawing pictures depicting the different parts of Turing machines, such as was done in Figure 12.2, we use the following compact notation to represent configurations. If we have a DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, and we wish to refer to a configuration of this DTM, we express it in the form

$$u(q, a)v \tag{12.3}$$

for some state $q \in Q$, a tape symbol $a \in \Gamma$, and (possibly empty) strings of tape symbols u and v such that

$$u \in \Gamma^* \backslash \{\sqcup\}\Gamma^* \quad \text{and} \quad v \in \Gamma^* \backslash \Gamma^*\{\sqcup\} \tag{12.4}$$

(i.e., u and v are strings of tape symbols such that u does not start with a blank and v does not end with a blank). What the expression (12.3) means is that the string uav is written on the tape in consecutive squares, with all other tape squares containing the blank symbol; the state of M is q; and the tape head of M is positioned over the symbol a that occurs between u and v.

For example, the configuration of the DTM in Figure 12.1 is expressed as

$$0\$1(q_4, 0)0\# \tag{12.5}$$

while the configuration of the DTM in Figure 12.2 is

$$(q_0, \sqcup)w \tag{12.6}$$

(for $w = \sigma_1 \cdots \sigma_n$).

When working with descriptions of configurations, it is convenient to define a few functions as follows. We define $\alpha : \Gamma^* \rightarrow \Gamma^* \backslash \{\sqcup\}\Gamma^*$ and $\beta : \Gamma^* \rightarrow \Gamma^* \backslash \Gamma^*\{\sqcup\}$ recursively as

$$\begin{aligned} \alpha(w) &= w && (\text{for } w \in \Gamma^* \backslash \{\sqcup\}\Gamma^*) \\ \alpha(\sqcup w) &= \alpha(w) && (\text{for } w \in \Gamma^*) \end{aligned} \tag{12.7}$$

and

$$\begin{aligned}\beta(w) &= w && (\text{for } w \in \Gamma^*\backslash\Gamma^*\{\sqcup\})\\ \beta(w\sqcup) &= \beta(w) && (\text{for } w \in \Gamma^*),\end{aligned} \tag{12.8}$$

and we define

$$\gamma : \Gamma^*(Q \times \Gamma)\Gamma^* \rightarrow \big(\Gamma^*\backslash\{\sqcup\}\Gamma^*\big)(Q \times \Gamma)\big(\Gamma^*\backslash\Gamma^*\{\sqcup\}\big) \tag{12.9}$$

as

$$\gamma(u(q,a)v) = \alpha(u)(q,a)\beta(v) \tag{12.10}$$

for all $u, v \in \Gamma^*$, $q \in Q$, and $a \in \Gamma$. This may look more complicated than it really is—the function γ just throws away all blank symbols on the left-most end of u and the right-most end of v, so that a proper expression of a configuration remains.

A yields relation for DTMs

Now we will define a *yields* relation, in a similar way to what we did for context-free grammars. This will in turn allow us to formally define acceptance and rejection for DTMs.

Definition 12.2. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a DTM. We define a *yields* relation $\vdash_M$ on pairs of expressions representing configurations as follows:

1. For every choice of $p \in Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p,a) = (q, b, \rightarrow), \tag{12.11}$$

the yields relation includes these pairs for all $u \in \Gamma^*\backslash\{\sqcup\}\Gamma^*$, $v \in \Gamma^*\backslash\Gamma^*\{\sqcup\}$, and $c \in \Gamma$:

$$\begin{aligned} u(p,a)cv &\vdash_M \gamma\big(ub(q,c)v\big)\\ u(p,a) &\vdash_M \gamma\big(ub(q,\sqcup)\big)\end{aligned} \tag{12.12}$$

2. For every choice of $p \in Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p,a) = (q, b, \leftarrow), \tag{12.13}$$

the yields relation includes these pairs for all $u \in \Gamma^*\backslash\{\sqcup\}\Gamma^*$, $v \in \Gamma^*\backslash\Gamma^*\{\sqcup\}$, and $c \in \Gamma$:

$$\begin{aligned} uc(p,a)v &\vdash_M \gamma\big(u(q,c)bv\big)\\ (p,a)v &\vdash_M \gamma\big((q,\sqcup)bv\big)\end{aligned} \tag{12.14}$$

In addition, we let $\vdash_M^*$ denote the reflexive, transitive closure of $\vdash_M$. That is, we have

$$u(p,a)v \vdash_M^* y(q,b)z \tag{12.15}$$

if and only if there exists an integer $m \geq 1$, strings $w_1, \ldots, w_m, x_1, \ldots, x_m \in \Gamma^*$, symbols $c_1, \ldots, c_m \in \Gamma$, and states $r_1, \ldots, r_m \in Q$ such that $u(p,a)v = w_1(r_1, c_1)x_1$, $y(q,b)v = w_m(r_m, c_m)x_m$, and

$$w_k(r_k, c_k)x_k \vdash_M w_{k+1}(r_{k+1}, c_{k+1})x_{k+1} \tag{12.16}$$

for all $k \in \{1, \ldots, m-1\}$.

That definition looks a bit technical, but it is actually very simple—whenever we have

$$u(p,a)v \vdash_M y(q,b)z \tag{12.17}$$

it means that by running M for one step we move from the configuration represented by $u(p,a)v$ to the configuration represented by $y(q,b)z$; and whenever we have

$$u(p,a)v \vdash_M^* y(q,b)z \tag{12.18}$$

it means that by running M for some number of steps, possibly zero steps, we will move from the configuration represented by $u(p,a)v$ to the configuration represented by $y(q,b)z$.

Acceptance and rejection for DTMs

Finally, we can write down a definition for acceptance and rejection by a DTM, using the relation $\vdash_M^*$ we just defined.

Definition 12.3. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a DTM and let $w \in \Sigma^*$ be a string. If there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_{\text{init}}, \sqcup)w \vdash_M^* u(q_{\text{acc}}, a)v, \tag{12.19}$$

then M *accepts* w. If there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_{\text{init}}, \sqcup)w \vdash_M^* u(q_{\text{rej}}, a)v, \tag{12.20}$$

then M *rejects* w. If neither of these conditions hold, then M *runs forever* on input w.

In words, if we start out in the initial configuration of a DTM M on an input w and start it running, we *accept* if we eventually hit the accept state, we *reject* if we eventually hit the reject state, and we *run forever* if neither of these two possibilities holds.

Similar to what we have done for DFAs, NFAs, CFGs, and PDAs, we write $\mathrm{L}(M)$ to denote the language of all strings that are accepted by a DTM M. In the case of DTMs, the language $\mathrm{L}(M)$ doesn't really tell the whole story—a string $w \notin \mathrm{L}(M)$ might either be rejected or it may cause M to run forever—but the notation is useful nevertheless.

Decidable and Turing-recognizable languages

It has been a lecture filled with definitions, but we still need to discuss two more. All of the definitions are important, but these are particularly important because they introduce concepts that we will be discussing for the next several lectures.

Definition 12.4. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language.

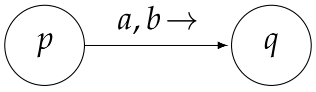
1. The language A is *Turing recognizable* if there exists a DTM M for which it holds that $A = \mathrm{L}(M)$.
2. The language A is *decidable* if there exists a DTM M that satisfies two conditions: (i) $A = \mathrm{L}(M)$, and (ii) M never runs forever (i.e., it either accepts or rejects each input string $w \in \Sigma^*$).

It is evident from the definition that every decidable language is also Turing recognizable—we've added an additional condition on the DTM M for the case of decidability. We will see later that the other containment does not hold: there are languages that are Turing recognizable but not decidable. Understanding which languages are decidable, which are Turing recognizable but not decidable, and what relationships hold among the two classes will occupy us for several lectures, and we'll see many interesting results and ideas about computation along the way.

From time to time you may hear about these concepts by different names. Sometimes the term *recursive* is used in place of *decidable* and *recursively enumerable* is used in place of *Turing recognizable*. These are historical terms based on different, but equivalent, ways of defining these classes of languages.

12.3 A simple example of a Turing machine

Let us now see an example of a DTM. For this first example, we will describe the DTM using a state diagram. The syntax is a bit different, but the idea is similar to state diagrams for DFAs. In the DTM case, we represent the property that the transition function satisfies $\delta(p, a) = (q, b, \rightarrow)$ with a transition of the form



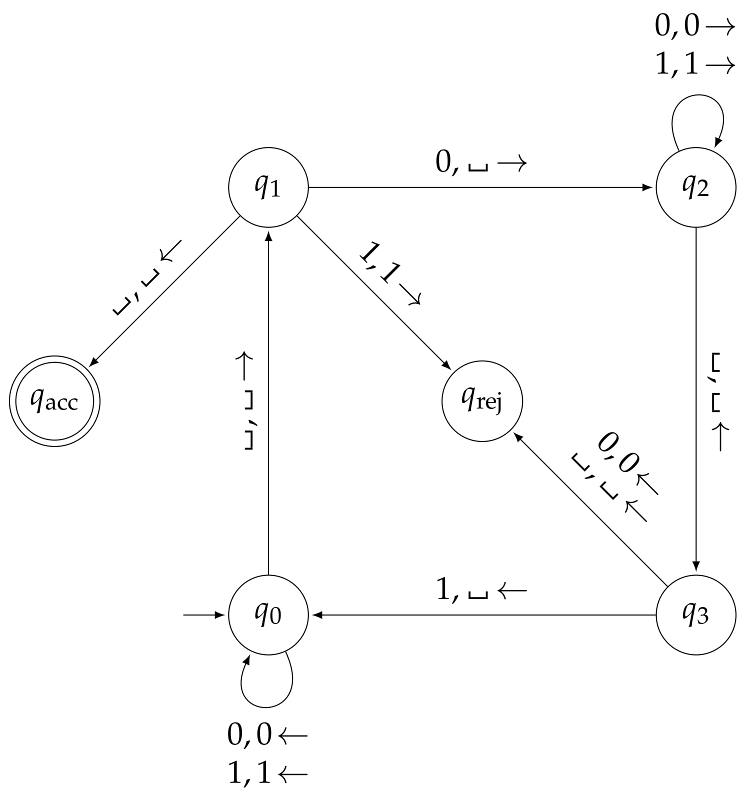


Figure 12.3: A DTM M for the language $\{0^n1^n : n \in \mathbb{N}\}$.

and similarly we represent the property that $\delta(p, a) = (q, b, \leftarrow)$ with a transition of the form

$$\begin{tikzcd} p \arrow[r, "{a, b \leftarrow}"] & q \end{tikzcd}$$

The state diagram for the example is given in Figure 12.3. The DTM M described by this diagram is for the language

$$A = \{0^n1^n : n \in \mathbb{N}\}. \tag{12.21}$$

To be more precise, M accepts every string in A and rejects every string in $\overline{A}$, and therefore the language A is *decidable*.

The specific way that the DTM M works can be summarized as follows. The DTM M starts out with its tape head scanning the blank symbol immediately to the left of its input. It moves the tape head right, and if it sees a 1 it rejects: the input string must not be of the form 0^n1^n if this happens. On the other hand, if it sees another blank symbol, it accepts: the input must be the empty string, which corresponds to the $n = 0$ case in the description of A. Otherwise, it must have seen the symbol 0, and in this case the 0 is erased (meaning that it replaces it with the blank symbol), the tape head repeatedly moves right until a blank is found, and then it moves one square back to the left. If a 1 is not found at this location the DTM rejects: there weren't enough 1s at the right end of the string. Otherwise, if a 1 is found, it is erased, and the tape head moves all the way back to the left, where we essentially recurse on a slightly shorter string.

Of course, the summary just suggested doesn't tell you precisely how the DTM works—but if you didn't already have the state diagram from Figure 12.3, the summary would probably be enough to give you a good idea for how to come up with the state diagram (or perhaps a slightly different one operating in a similar way).

In fact, an even higher-level summary would probably be enough. For instance, we could describe the functioning of the DTM M as follows:

1. Accept if the input is the empty string.
2. Check that the left-most non-blank symbol on the tape is a 0 and that the right-most non-blank symbol is a 1. Reject if this is not the case, and otherwise erase these symbols and goto 1.

There will, of course, be several specific ways to implement this algorithm with a DTM, with the DTM M from Figure 12.3 being one of them.

Back to the Church–Turing thesis

The example of a DTM above was very simple, which makes it atypical. The DTMs we will be most interested in will almost always be much more complicated—so complicated, in fact, that the idea of representing them by state diagrams would be absurd. The reality is that state diagrams turn out to be almost totally useless for describing DTMS, and so we will almost never refer to them; this would be quite similar to describing complex programs using machine code.

The more typical way to describe DTMs will be in terms of *algorithms*, typically expressed in the form of pseudo-code or high-level descriptions like we did at the end of our first example above. As you build your understanding of how DTMs work and what they can do, you'll gain more confidence that these sorts of high-level descriptions actually could be implemented by DTMs. It may take some time for you to convince yourself of this, but perhaps your experience as programmers will help—you already know that very complex algorithms can be implemented through very primitive low-level instructions carried out by a computer, and the situation is similar for Turing machines.

This is connected to the idea that the Church–Turing thesis suggests. If you can implement an algorithm using your favorite programming language, then with enough motivation and stamina you could also implement the same algorithm on a Turing machine. (Of course we are talking about algorithms that take an input string and produce an output, not interactive, multi-media applications or things like this.) We will continue discussing this idea in the next couple of lectures.