

4.1 The definition of a Turing machine

A Turing machine consists of a finite control, a tape and a head that can be used for reacting or writing on that tape

Turing machines are designed to satisfy simultaneously these three criteria:

1. They should be automata; that is, their construction and function should be in the same general spirit as the devices previously studied.
2. They should be as simple as possible to describe, define formally, and reason about.
3. They should be as general as possible in terms of the computations they can carry out.

A Turing machine consists of a finite state control unit and a tape. Communication between the two is provided by a single head, which reads symbols from the tape and is also used to change the symbols on the tape. The control unit operates in discrete steps; at each step it performs two functions in a way that depends on the current state and the tape symbol currently scanned by the read/write head.

1. Put the control unit in a new state
2. Either:
 1. write a symbol in the tape square currently scanned, replacing the one already there, or
 2. move the read/write head on the tape square to the left or right

The tape has a left end but it extends indefinitely to the right. To prevent the machine from moving its head off the left end of the tape, we assume that the left most end of the tape is always marked by a special symbol denoted by \sqcup ; we assume further that all our Turing machines are so designed that, whenever the head reads a \sqcup , it immediately moves to the right. Also we shall use the distinct symbols \leftarrow and \rightarrow to denote movement of the head to the left or right.

Definition 4.1.1:

A Turing machine is a quintuple $(K, \Sigma, \delta, s, H)$ where:

- K is a finite set of states
- Σ is an alphabet, containing the blank symbol \sqcup and the left end symbol \sqcup , but not containing the symbol \sqcup and \sqcup
- $s \in K$ is the initial state.
- $H \subseteq K$ is the set of halting states
- δ the transition function is a function from $(K-H) \times \Sigma$ to $K \times (\Sigma \cup \{\sqcup, \leftarrow, \rightarrow\})$ such that
 1. for all $q \in K-H$, if $\delta(q, \sqcup) = (p, b)$ then $b = \sqcup$
 2. for all $q \in K-H$ and $a \in \Sigma$ if $\delta(q, a) = (p, b)$ then $b \neq \sqcup$

Definition 4.1.2:

a configuration of a Turing machine $M = (K, \Sigma, \delta, s, H)$ is a member of $K \times \Sigma^* \times (\Sigma^* (\Sigma \cup \{\sqcup\}) \cup \{e\})$

That is, all configurations are assumed to start with the left end symbol, and never end with a blank - unless the blank is currently scanned. A configuration whose state component is in H will be called a halted configuration.

Definition 4.1.3:

let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine and consider two configurations of M $(q_1, w_1 a_1 u_1)$ and $(q_2, w_2 a_2 u_2)$, where $a_1, a_2 \in \Sigma$. Then

$$(q_1, w_1 a_1 u_1) \vdash_M (q_2, w_2 a_2 u_2)$$

if and only if, for some $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$, $\delta(q_1, a_1) = (q_2, b)$, and either

1. $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$ and $a_2 = b$, or
2. $b = \leftarrow$, $w_1 = w_2 a_2$ and either
 1. $u_2 = a_1 u_1$, if $a_1 \neq \epsilon$ or $u_1 \neq \epsilon$, or
 2. $u_2 = \epsilon$, if $a_1 = \epsilon$ and $u_1 = \epsilon$; or
3. $b = \rightarrow$, $w_2 = w_1 a_1$, and either
 1. $u_1 = a_2 u_2$, or
 2. $u_1 = u_2 = \epsilon$, and $a_2 = \epsilon$.

Definition 4.1.4:

For any Turing machine M , let $\overset{*}{\sim}_M$ be the reflexive, transitive closure of \sim_M , we say that configuration C_1 yields configuration C_2 if $C_1 \overset{*}{\sim}_M C_2$. A computation by M is a sequence of configurations C_0, C_1, \dots, C_n for some $n \geq 0$ such that

$$C_0 \overset{M}{\sim} C_1 \overset{M}{\sim} C_2 \overset{M}{\sim} \dots \overset{M}{\sim} C_n$$

We say that the computation is of length n or that it has n steps, and we write $C_0 \overset{n}{\sim}_M C_n$

If $a \in \Sigma$, then the a -writing machine will be denoted simply as a . The head moving machines M_{\leftarrow} and M_{\rightarrow} will be abbreviated as L and R

4.2 Computing with Turing machines

Input is presented to Turing machines in the following way. The input string, with no blank symbol in it, is written to the right of the leftmost symbol ϵ , with a blank to its left and a blank to its right, the head is positioned at the tape square containing the blank between the ϵ and the input, and the machine starts operating in its initial state. If $M = (K, \Sigma, \delta, s, H)$ is a Turing machine and $w \in (\Sigma - \{\epsilon, \leftarrow, \rightarrow\})^*$, then the initial configuration of M on input w is $(s, \epsilon w)$. With this convention we can now define how Turing machines are used as language recognizers

Definition 4.2.1:

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine such that $H = \{y, n\}$ consists of two distinguished halting states (y and n for yes and no). Any halting state whose state component is y is called an accepting configuration while a halting configuration whose state component is n is called a rejecting configuration. Let $\Sigma_0 \subseteq \Sigma - \{\epsilon, \leftarrow, \rightarrow\}$ be an alphabet, called the input alphabet of M ; by fixing Σ_0 to be a subset of $\Sigma - \{\epsilon, \leftarrow, \rightarrow\}$, we allow our Turing machines to use extra symbols during their computation, besides those appearing in their input. We say that M decides a language $L \subseteq \Sigma_0^*$ if for any string $w \in \Sigma_0^*$ the following is true: if $w \in L$ then M accepts w ; and if $w \notin L$ then M rejects w . Finally call a language L recursive if there is a Turing machine that decides it.

That is, a Turing machine decides a language L if, when started with input w , it always halts, and does so in a halt state that is correct response to the input: y if $w \in L$, n if $w \notin L$. Notice that no guarantees are given about what happens if the input to the machine contains blanks or the left end symbol.

A Turing machine, even if it has only two halt states y and n , always has the option of evading an answer (yes or no) by falling to halt.

Definition 4.2.2:

Let $M = (K, \Sigma, \delta, s, \{h\})$ be a Turing machine, let $\Sigma_0 \subseteq \Sigma - \{ _, \}$ be an alphabet, and let $w \in \Sigma_0^*$. Suppose that M halts on input w , and that $(s, _w) \xrightarrow{*}_M (h, _y)$ for some $y \in \Sigma_0^*$. Then u is called the output of M on input w , and is denoted $M(w)$. Notice that $M(w)$ is defined only if M halts on input w , and in fact does so at a configuration of the form $(h, _y)$ with $y \in \Sigma_0^*$. Now let f be any function from Σ_0^* to Σ_0^* . We say that M computes function f if for all $w \in \Sigma_0^*$, $M(w) = f(w)$. That is, for all $w \in \Sigma_0^*$ M eventually halts on input w , and when it does halt, its tape contains the string $f(w)$. A function is called recursive if there is a Turing machine M that computes f .

Definition 4.2.3:

Let $M = (K, \Sigma, \delta, s, \{h\})$ be a Turing machine such that $0, 1 \in \Sigma$ and let f be any function from \mathbb{N}^k to \mathbb{N} for some $k \geq 1$. We say that M computes function f if for all $w_1, \dots, w_k \in 0 \cup 1 \{0, 1\}^*$ (that is for any k strings that are binary encodings of integers), $\text{num}(M(w_1, \dots, w_k)) = f(\text{num}(w_1), \dots, \text{num}(w_k))$. That is if M is started with the binary representations of the integers n_1, \dots, n_k as input, then it eventually halts, and when it does halt, its tape contains a string that represents numbers $f(n_1, \dots, n_k)$ - the value of the function. A function $f: \mathbb{N}^k \rightarrow \mathbb{N}$ is called recursive if there is a Turing machine M that computes f .

If a Turing machine decides a language or computes a function, it can be reasonably thought of as an algorithm that performs correctly and reliably some computational task. We next introduce a third subtler way in which a Turing machine can define a language:

Definition 4.2.4:

Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine let $\Sigma_0 \subseteq \Sigma - \{ _, \}$ be an alphabet, and let $L \subseteq \Sigma_0^*$ be a language. We say that M semidecides L if for any string $w \in \Sigma_0^*$ the following is true $w \in L$ if and only if M halts on input w . A language L is recursively enumerable if and only if there is a Turing machine M that semidecides L .

Extending the "functional" notation of Turing machines that we introduced previously (which allows us to write equations such as $M(w) = u$), we shall write $M(w) = ?$ if M fails to halt on input w . In this notation we can restate the definition of semidecision of a language $L \subseteq \Sigma_0^*$ by Turing machine M as follows: For all $w \in \Sigma_0^*$ $M(w) = ?$ if and only if $w \in L$.

A Turing machine that semidecides a language L cannot be usefully employed for telling whether a string w is in L , because if $w \notin L$, then we will never know when we have waited enough for an answer. Turing machines that semidecide languages are not algorithms.

Any recursive language is also recursively enumerable. All it takes to construct another Turing machine that semidecides instead of decides, the language is to make the rejecting state a nonhalting state, from which the machine is guaranteed to never halt.

Theorem 4.2.1:

If a language is recursive, then it is recursively enumerable.

There are recursively enumerable languages that are not recursive.

Theorem 4.2.2:

If L is a recursive language, then its complement \bar{L} is also recursive.

4.5 Nondeterministic Turing machines

A nondeterministic Turing machine is a quintuple $(K, \Sigma, \Delta, s, H)$, where K, Σ, s and H as for standard Turing machines and Δ is a subset of $((K-H) \times \Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$ rather than a function from $(K-H) \times \Sigma$ to $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$. Configurations and the relations \rightarrow_M and \rightarrow_M^* are defined in the natural way. But now \rightarrow_M need not be single valued. One configuration may yield several others in one step

Definition 4.5.1:

Let $M = (K, \Sigma, \Delta, s, H)$ be a nondeterministic Turing machine. We say that M accepts an input $w \in (\Sigma - \{_, \blacksquare\})^*$ if $(s, _w) \rightarrow_M^* (h, u\underline{a}v)$ for some $h \in H$ and $a \in \Sigma, u, v \in \Sigma^*$. Notice that a nondeterministic machine accepts an input even though it may have many nonhalting computations on this input - as long as at least one halting computation exists. We say that M semidecides a language $L \subseteq (\Sigma - \{_, \blacksquare\})^*$ if the following holds for all $w \in (\Sigma - \{_, \blacksquare\})^*$: $w \in L$ if and only if M accepts w .

Definition 4.5.2:

Let $M = (K, \Sigma, \Delta, s, \{y, n\})$ be a nondeterministic Turing machine. We say that M decides a language $L \subseteq (\Sigma - \{_, \blacksquare\})^*$ if the following two conditions hold for all $w \in (\Sigma - \{_, \blacksquare\})^*$:

1. There is a natural number N , dependent on M and w , such that there is no configuration C satisfying $(s, _w) \rightarrow_M^n C$
 2. $w \in L$ if and only if $(s, _w) \rightarrow_M^* (y, u\underline{a}v)$ for some $u, v \in \Sigma^*, a \in \Sigma$
- Finally, we say that M computes a function $f: (\Sigma - \{_, \blacksquare\})^* \rightarrow (\Sigma - \{_, \blacksquare\})^*$ if the following two conditions hold for all $w \in (\Sigma - \{_, \blacksquare\})^*$:

1. There is an N depending on M and w , such that there is no configuration C satisfying $(s, _w) \rightarrow_M^n C$.
2. $(s, _w) \rightarrow_M^* (h, u\underline{a}w)$ if and only if $ua = _w$ and $v = f(w)$

Theorem 4.5.1:

If a nondeterministic Turing machine M semidecides or decides a language, or computes a function, then there is a standard Turing machine M' semideciding or deciding the same language, or computing the same function.

4.7 Numerical Functions

Definition 4.7.1:

We start by defining certain extremely simple functions from \mathbb{N}^k to \mathbb{N} , for various values of $k \geq 0$ (a 0-ary function is, of course, a constant, and it has nothing on which to depend). The basic functions are the following:

1. For any $k \geq 0$, the k -ary zero function is defined as $\text{zero}_k(n_1, \dots, n_k) = 0$ for all $n_1, \dots, n_k \in \mathbb{N}$
2. For any $k \geq j \geq 0$, the j^{th} k -ary identity function is simply the function $\text{id}_{k,j}(n_1, \dots, n_k) = n_j$ for all $n_1, \dots, n_k \in \mathbb{N}$
3. The successor function is defined as $\text{succ}(n) = n+1$ for all $n \in \mathbb{N}$

Next we introduce two simple ways of combining functions to get slightly more complex functions.

1. Let $k, l \geq 0$ let $g: \mathbb{N}^k \rightarrow \mathbb{N}$ be a k -ary function, and let n_1, \dots, n_k be l -ary functions. Then the composition of g with h_1, \dots, h_k is the l -ary function defined as $f(n_1, \dots, n_l) = g(h_1(n_1, \dots, n_l), \dots, h_k(n_1, \dots, n_l))$

2. Let $k \geq 0$ let g be a k -ary function, and let h be a $(k+2)$ -ary function. Then the function defined recursively by g and h is the $(k+1)$ -ary function defined as

$$f_0 = g(n_1, \dots, n_k)$$

$$f(n_1, \dots, n_k, m+1) = h(n_1, \dots, n_k, m, f(n_1, \dots, n_k, m))$$

for all $n_1, \dots, n_k, m \in \mathbb{N}$

The primitive recursive functions are all basic functions, and all functions that can be obtained by them by any number of successive applications of composition and recursive definition

Definition 4.7.2:

Let g be a $(k+1)$ -ary function, for some $k \geq 0$. The minimalization of g is the k -ary function defined as follows:

$$f(n_1, \dots, n_k)$$

$$\left\{ \begin{array}{l} \text{the least } m \text{ such that } g(n_1, \dots, n_k, m) = 1, \text{ if such an } m \text{ exists;} \\ 0 \text{ otherwise} \end{array} \right.$$

*** the $f(n_1, \dots, n_k)$ should be in front of the bracket***

We shall denote the minimalization of a function g is always well-defined, there is no obvious method for computing it - even if we know how to compute g . The obvious method

```

m:=0
while g(n1,...,nk,m) ≠ 1 do m:= m+1;
output m

```

is not an algorithm, because it may fail to terminate. Let us then call a function g minimalizable if the above method always terminates. That is a $(k+1)$ -ary function g is minimalizable if it has the following property. For every $n_1, \dots, n_k \in \mathbb{N}$, there is an $m \in \mathbb{N}$ such that $g(n_1, \dots, n_k, m) = 1$.

Finally, call a function μ -recursive if it can be obtained from the basic functions by the operations of composition: recursive definition, and minimalization of minimalizable functions.

Theorem 4.7.1: A function $f: \mathbb{N}^k \rightarrow \mathbb{N}$ is μ -recursive if and only if it is recursive (that is, computable by a Turing machine)