1 Definition of a Turing machine

Turing machines are an abstract model of computation. They provide a precise, formal definition of what it means for a function to be computable. Many other definitions of computation have been proposed over the years — for example, one could try to formalize precisely what it means to run a program in Java on a computer with an infinite amount of memory — but it turns out that all known definitions of computation agree on what is computable and what is not. The Turing Machine definition seems to be the simplest, which is why we present it here. The key features of the Turing machine model of computation are:

1. A finite amount of internal state.
2. An infinite amount of external data storage.
3. A program specified by a finite number of instructions in a predefined language.
4. Self-reference: the programming language is expressive enough to write an interpreter for its own programs.

Models of computation with these key features tend to be equivalent to Turing machines, in the sense that the distinction between computable and uncomputable functions is the same in all such models.

A Turing machine can be thought of as a finite state machine sitting on an infinitely long tape containing symbols from some finite alphabet Σ. Based on the symbol it's currently reading, and its current state, the Turing machine writes a new symbol in that location (possibly the same as the previous one), moves left or right or stays in place, and enters a new state. It may also decide to halt and, optionally, to output "yes" or "no" upon halting. The machine's *transition function* is the "program" that specifies each of these actions (overwriting the current symbol, moving left or right, entering a new state, optionally halting and outputting an answer) given the current state and the symbol the machine is currently reading.

Definition 1. A Turing machine is specified by a finite alphabet Σ, a finite set of states K with a special element s (the starting state), and a *transition function* $\delta : K \times \Sigma \to (K \cup \{\text{halt, yes, no}\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. It is assumed that Σ, K, {halt,yes,no}, and $\{\leftarrow, \rightarrow, -\}$ are disjoint sets, and that Σ contains two special elements $\triangleright, \sqcup$ representing the start and end of the tape, respectively. We require that for every $q \in K$, if $\delta(q, \triangleright) = (p, \sigma, d)$ then $\sigma = \triangleright$ and $d \neq \leftarrow$. In other words, the machine never tries to overwrite the leftmost symbol on its tape nor to move to the left of it.[1]

[1] One could instead define Turing machines as having a doubly-infinite tape, with the ability to move arbitrarily far both left and right. Our choice of a singly-infinite tape makes certain definitions more convenient, but does not limit the computational power of Turing machines. A Turing machine as defined here can easily simulate one with a doubly-infinite tape by using the even-numbered positions on its tape to simulate the non-negative positions on the doubly-infinite tape, and using the odd-numbered positions on its tape to simulate the negative positions on the doubly-infinite tape. This simulation requires small modifications to the transition diagram of the Turing machine. Essentially, every move on the doubly-infinite tape gets simulated with two consecutive moves on the single-infinite tape, and this requires a couple of extra "bridging states" that are used for passing through the middle location while doing one of these two-step moves. The interested reader may fill in the details.

Note that a Turing machine is not prevented from overwriting the rightmost symbol on its tape or moving to the right of it. In fact, this capability is necessary in order for Turing machines to perform computations that require more space than is given in their original input string.

Having defined the *specification* of a Turing machine, we must now pin down a definition of *how they operate*. This has been informally described above, but it's time to make it formal. That begins with formally defining the configuration of the Turing machine at any time (the state of its tape, as well as the machine's own state and its position on the tape) and the rules for how its configuration changes over time.

Definition 2. The set Σ^* is the set of all finite sequences of elements of Σ. When an element of Σ^* is denoted by a letter such as x, then the elements of the sequence x are denoted by $x_0, x_1, x_2, \ldots, x_{n-1}$, where n is the length of x. The length of x is denoted by $|x|$.

A *configuration* of a Turing machine is an ordered triple $(x, q, k) \in \Sigma^* \times K \times \mathbb{N}$, where x denotes the string on the tape, q denotes the machine's current state, and k denotes the position of the machine on the tape. The string x is required to begin with $\triangleright$ and end with $\sqcup$. The position k is required to satisfy $0 \leq k < |x|$.

If M is a Turing machine and (x, q, k) is its configuration at any point in time, then its configuration (x', q', k') at the following point in time is determined as follows. Let $(p, \sigma, d) = \delta(q, x_k)$. The string x' is obtained from x by changing x_k to σ, and also appending $\sqcup$ to the end of x, if $k = |x| - 1$. The new state q' is equal to p, and the new position k' is equal to $k-1, k+1$, or k according to whether d is $\leftarrow, \rightarrow$, or $-$, respectively. We express this relation between (x, q, k) and (x', q', k') by writing $(x, q, k) \xrightarrow{M} (x', q', k')$.

A *computation* of a Turing machine is a sequence of configurations (x_i, q_i, k_i), where i runs from 0 to T (allowing for the case $T = \infty$) that satisfies:

- The machine starts in a valid starting configuration, meaning that $q_0 = s$ and $k_0 = 0$.
- Each pair of consecutive configurations represents a valid transition, i.e. for $0 \leq i < T$, it is the case that $(x_i, q_i, k_i) \xrightarrow{M} (x_{i+1}, q_{i+1}, k_{i+1})$.
- If $T = \infty$, we say that the computation *does not halt*.
- If $T < \infty$, we require that $q_T \in \{\text{halt,yes,no}\}$ and we say that the computation *halts*.

If $q_T = \text{yes}$ (respectively, $q_T = \text{no}$) we say that the computation outputs "yes" (respectively, outputs "no"). If $q_T = \text{halt}$ then the output of the computation is defined to be the string obtained by removing $\triangleright$ and $\sqcup$ from the start and end of x_T. In all three cases, the output of the computation is denoted by $M(x)$, where x is the input, i.e. the string x_0 without the initial $\triangleright$ and final $\sqcup$. If the computation does not halt, then its output is undefined and we write $M(x) = \nearrow$.

The upshot of this discussion is that one must standardize on either a singly-infinite or doubly-infinite tape, for the sake of making a precise definition, but the choice has no effect on the computational power of the model that is eventually defined. As we go through the other parts of the definition of Turing machines, we will encounter many other examples of this phenomenon: details that must be standardized for the sake of precision, but where the precise choice of definition has no bearing on the distinction between computable and uncomputable.

2 Examples of Turing machines

Example 1. As our first example, let's construct a Turing machine that takes a binary string and appends 0 to the left side of the string. The machine has four states: s, r_0, r_1, ℓ. State s is the starting state, in state r_0 and r_1 it is moving right and preparing to write a 0 or 1, respectively, and in state ℓ it is moving left.

The state s will be used only for getting started: thus, we only need to define how the Turing machine behaves when reading $\triangleright$ in state s. The states r_0 and r_1 will be used, respectively, for writing 0 and writing 1 while remembering the overwritten symbol and moving to the right. Finally, state ℓ is used for returning to the left side of the tape without changing its contents. This plain-English description of the Turing machine implies the following transition function. For brevity, we have omitted from the table the lines corresponding to pairs (q, σ) such that the Turing machine can't possibly be reading σ when it is in state q.

		$\delta(q,\sigma)$		
q	σ	state	symbol	direction
s	$\triangleright$	r_0	$\triangleright$	$\rightarrow$
r_0	0	r_0	0	$\rightarrow$
r_0	1	r_1	0	$\rightarrow$
r_0	$\sqcup$	ℓ	0	$\leftarrow$
r_1	0	r_0	1	$\rightarrow$
r_1	1	r_1	1	$\rightarrow$
r_1	$\sqcup$	ℓ	1	$\leftarrow$
ℓ	0	ℓ	0	$\leftarrow$
ℓ	1	ℓ	1	$\leftarrow$
ℓ	$\triangleright$	halt	$\triangleright$	–

Example 2. Using similar ideas, we can design a Turing machine that takes a binary integer n (with the digits written in order, from the most significant digits on the left to the least significant digits on the right) and outputs the binary representation of its successor, i.e. the number $n + 1$.

It is easy to see that the following rule takes the binary representation of n and outputs the binary representation of $n + 1$. Find the rightmost occurrence of the digit 0 in the binary representation of n, change this digit to 1, and change every digit to the right of it from 1 to 0. The only exception is if the binary representation of n does not contain the digit 0; in that case, one should change every digit from 1 to 0 and prepend the digit 1.

		$\delta(q,\sigma)$		
q	σ	state	symbol	direction
s	$\triangleright$	r	$\triangleright$	$\rightarrow$
r	0	r	0	$\rightarrow$
r	1	r	1	$\rightarrow$
r	$\sqcup$	ℓ	$\sqcup$	$\leftarrow$
ℓ	0	t	1	$\leftarrow$
ℓ	1	ℓ	0	$\leftarrow$
ℓ	$\triangleright$	prepend::s	$\triangleright$	–
t	0	t	0	$\leftarrow$
t	1	t	1	$\leftarrow$
t	$\triangleright$	halt	$\triangleright$	–
prepend::s	$\triangleright$	prepend::s	$\triangleright$	$\rightarrow$
prepend::s	0	prepend::r	1	$\rightarrow$
prepend::r	0	prepend::r	0	$\rightarrow$
prepend::r	$\sqcup$	prepend::ℓ	0	$\leftarrow$
prepend::ℓ	0	prepend::ℓ	0	$\leftarrow$
prepend::ℓ	1	prepend::ℓ	1	$\leftarrow$
prepend::ℓ	$\triangleright$	halt	$\triangleright$	–

Thus, the Turing machine for computing the binary successor function works as follows: it uses one state r to initially scan from left to right without modifying any of the digits, until it encounters the symbol $\sqcup$. At that point it changes into a new state ℓ in which it moves to the left, changing any 1's that it encounters to 0's, until the first time that it encounters a symbol other than 1. (This may happen before encountering any 1's.) If that symbol is 0, it changes it to 1 and enters a new state t that moves leftward to $\triangleright$ and then halts. On the other hand, if the symbol is $\triangleright$, then the original input consisted exclusively of 1's. In that case, it prepends

a 1 to the input using a subroutine very similar to Example 1. We'll refer to the states in that subroutine as prepend::s, prepend::r, prepend::ℓ.

Example 3. We can compute the binary predecessor function in roughly the same way. We must take care to define the value of the predecessor function when its input is the number 0, since we haven't yet specified how negative numbers should be represented on a Turing machine's tape using the alphabet $\{\triangleright, \sqcup, 0, 1\}$. Rather than specify a convention for representing negative numbers, we will simply define the value of the predecessor function to be 0 when its input is 0. Also, for convenience, we will allow the output of the binary predecessor function to have any number of initial copies of the digit 0.

		$\delta(q,\sigma)$		
q	σ	state	symbol	direction
s	$\triangleright$	z	$\triangleright$	$\rightarrow$
z	0	z	0	$\rightarrow$
z	1	r	1	$\rightarrow$
z	$\sqcup$	t	$\sqcup$	$\leftarrow$
r	0	r	0	$\rightarrow$
r	1	r	1	$\rightarrow$
r	$\sqcup$	ℓ	$\sqcup$	$\leftarrow$
ℓ	0	ℓ	1	$\leftarrow$
ℓ	1	t	0	$\leftarrow$
t	0	t	0	$\leftarrow$
t	1	t	1	$\leftarrow$
t	$\triangleright$	halt	$\triangleright$	–

Our program to compute the binary predecessor of n thus begins with a test to see if n is equal to 0. The machine moves to the right, remaining in state z unless it sees the digit 1. If it reaches the end of its input in state z, then it simply rewinds to the beginning of the tape. Otherwise, it decrements the input in much the same way that the preceding example incremented it: in this case we use state ℓ to change the trailing 0's to 1's until we encounter the rightmost 1, and then we enter state t to rewind to the beginning of the tape.