## mac.1 Introduction

tur:mac:int:sec

What does it mean for a function, say, from $\mathbb{N}$ to $\mathbb{N}$ to be *computable*? Among the first answers, and the most well known one, is that a function is computable if it can be computed by a Turing machine. This notion was set out by Alan Turing in 1936. Turing machines are an example of *a model of computation*—they are a mathematically precise way of defining the idea of a "computational procedure." What exactly that means is debated, but it is widely agreed that Turing machines are one way of specifying computational procedures. Even though the term "Turing machine" evokes the image of a physical machine with moving parts, strictly speaking a Turing machine is a purely mathematical construct, and as such it idealizes the idea of a computational procedure. For instance, we place no restriction on either the time or memory requirements of a Turing machine: Turing machines can compute something even if the computation would require more storage space or more steps than there are atoms in the universe.

explanation

It is perhaps best to think of a Turing machine as a program for a special kind of imaginary mechanism. This mechanism consists of a *tape* and a *read-write head*. In our version of Turing machines, the tape is infinite in one direction (to the right), and it is divided into *squares*, each of which may contain a symbol from a finite *alphabet*. Such alphabets can contain any number of different symbols, say, but we will mainly make do with three: $\triangleright$, 0, and 1. When the mechanism is started, the tape is empty (i.e., each square contains the symbol 0) except for the leftmost square, which contains $\triangleright$, and a finite number of squares which contain the *input*. At any time, the mechanism is in one of a finite number of *states*. At the outset, the head scans the leftmost square and in a specified *initial state*. At each step of the mechanism's run, the content of the square currently scanned together with the state the mechanism is in and the Turing machine program determine what happens next. The Turing machine program is given by a partial function which takes as input a state $q$ and a symbol $\sigma$ and outputs a triple $\langle q', \sigma', D\rangle$. Whenever the mechanism is in state $q$ and reads symbol $\sigma$, it replaces the symbol on the current square with $\sigma'$, the head moves left, right, or stays put according to whether $D$ is $L$, $R$, or $N$, and the mechanism goes into state $q'$.

For instance, consider the situation in section mac.1. The visible part of the tape of the Turing machine contains the end-of-tape symbol $\triangleright$ on the leftmost square, followed by three 1's, a 0, and four more 1's. The head is reading the third square from the left, which contains a 1, and is in state $q_1$—we say "the machine is reading a 1 in state $q_1$." If the program of the Turing machine returns, for input $\langle q_1, 1\rangle$, the triple $\langle q_2, 0, N\rangle$, the machine would now replace the 1 on the third square with a 0, leave the read/write head where it is, and switch to state $q_2$. If then the program returns $\langle q_3, 0, R\rangle$ for input $\langle q_2, 0\rangle$, the machine would now overwrite the 0 with another 0 (effectively, leaving the content of the tape under the read/write head unchanged), move one square to the right, and enter state $q_3$. And so on.

We say that the machine *halts* when it encounters some state, $q_n$, and sym-

tur:mac:int:
fig:tm

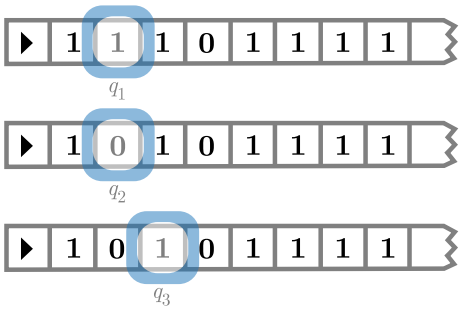


Figure 1: A Turing machine executing its program.

bol, $\sigma$ such that there is no instruction for $\langle q_n, \sigma \rangle$, i.e., the transition function for input $\langle q_n, \sigma \rangle$ is undefined. In other words, the machine has no instruction to carry out, and at that point, it ceases operation. Halting is sometimes represented by a specific halt state $h$. This will be demonstrated in more detail later on.

digression

The beauty of Turing's paper, "On computable numbers," is that he presents not only a formal definition, but also an argument that the definition captures the intuitive notion of computability. From the definition, it should be clear that any function computable by a Turing machine is computable in the intuitive sense. Turing offers three types of argument that the converse is true, i.e., that any function that we would naturally regard as computable is computable by such a machine. They are (in Turing's words):

1. A direct appeal to intuition.

2. A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

3. Giving examples of large classes of numbers which are computable.

Our goal is to try to define the notion of computability "in principle," i.e., without taking into account practical limitations of time and space. Of course, with the broadest definition of computability in place, one can then go on to consider computation with bounded resources; this forms the heart of the subject known as "computational complexity."

**Historical Remarks** Alan Turing invented Turing machines in 1936. While his interest at the time was the decidability of first-order logic, the paper has been described as a definitive paper on the foundations of computer design. In the paper, Turing focuses on computable real numbers, i.e., real numbers whose decimal expansions are computable; but he notes that it is not hard to adapt his notions to computable functions on the natural numbers, and so on. Notice that this was a full five years before the first working general

purpose computer was built in 1941 (by the German Konrad Zuse in his parent's living room), seven years before Turing and his colleagues at Bletchley Park built the code-breaking Colossus (1943), nine years before the American ENIAC (1945), twelve years before the first British general purpose computer—the Manchester Small-Scale Experimental Machine—was built in Manchester (1948), and thirteen years before the Americans first tested the BINAC (1949). The Manchester SSEM has the distinction of being the first stored-program computer—previous machines had to be rewired by hand for each new task.

## Photo Credits

## Bibliography