Turing Machines

Richard Cole

November 21, 2011

Alan Turing completed the invention of what became known as Turing Machines in 1936. He was seeking to describe in a precise way what it means to systematically calculate or compute. Before going further let me note that Turing Machines are conceptual devices; while they are easy to describe on paper, and in principle could be built, Turing had no intention of building them or having others build them. (Turing was not simply an “ivory-tower” mathematician; in the late 1940s he was one of the earliest users of one of the few computers then in existence — at Manchester University in England).

Let us consider what systematic computation meant in 1936. At the time there were machines and tools for helping with calculations: adding machines and slide rules; slide rules continued to be in widespread use until the early 1970s when calculators became relatively cheap. Also, supporting data management, were card readers and sorters (IBM was a leading supplier). There were even computers; but these were people whose job was to compute (a description of a computing room at Los Alamos in 1944 full of human computers can be found in Richard Feynman’s semi-autobiography “Surely You’re Joking Mr. Feynman,” in the chapter “Los Alamos from Below”).

What Turing was trying to formalize were the processes followed when calculating something: the orbit of a planet, the load a bridge could bear, the next move to play in a game of chess, etc., though subsequently he proposed a much broader view of what could be computed (at least in principle).

The key question Turing asked was how do people compute? Essentially people could do three things: they could think in their minds, they could write down something (on a sheet of paper), or they could read something they had written down previously. As we all know, with too many sheets of paper it is hard to keep track of what is where, and consequently it is helpful to number or label the pages. One way of organizing this is to keep the pages bound in a book in numbered order. However, for reasons of conceptual simplicity, Turing preferred to think of the pages as being on a roll, or tape as he called it, with the pages being accessed in consecutive order by rolling or unrolling the roll (this is the way in which long documents were made and read in ancient times).

The actions with the roll or tape, as we shall call it henceforth, were very simple: one could read the current page, one could decide to erase the current page and write something new (pencil not ink), one could decide to advance to the next page or go back to the previous page. But does this really capture everything? Suppose one wanted to go to page 150 and one was currently on page 28: well, then one needs to advance one page 122 times — a tedious process, for sure, but one that achieves the desired result in the end. The implicit
assumption is that one can hold the count in one’s mind. But what if the numbers are too large to hold in mind? (Incidentally, this implies that even a whole page is too large to hold in mind). What one would like to do is to write them on scraps of paper; however, this is cheating. What one has to do is use the tape pages as the scraps of paper. How this might be done is left as an exercise to the reader (see exercise 2 below).

How did Turing abstract the notion of thinking? The crucial hypothesis was that a mind can hold only a finite number of different thoughts, albeit extremely large, not an infinite number. Why is this plausible? At the level of neurons (not that this was properly understood in Turing’s day) simplifying a bit, each neuron is either sending a signal or not; as their number is finite, this yields only a finite number of possibilities — albeit a ridiculously large number of possibilities. At the level of elementary particles (electrons, protons, etc.), quantum physics states that there are only a finite number of states for each particle even if they are not fully knowable, and again this yields only a finite number of possibilities. So abstractly, thinking can be viewed as follows: the mind has a (large) finite number of states in which it could currently be, conventionally denoted by \( q_1, q_2, \ldots, q_k \). Computation takes the following form: when in state \( q \), on reading the current page take an action as already described (rewrite the page, move to an adjacent page) and enter a new state \( q' \) (possibly \( q' = q \)); this action is fully determined by the state \( q \) and the contents of the current page.

Turing proposed one more conceptual simplification. Just as the states of a mind are finite, what one can write on a single page is also finite. Accordingly he proposed representing the possible one-page writings by a finite set of symbols \( a_1, a_2, \ldots, a_r \), which he called characters; he called the set \( A = \{a_1, a_2, \ldots, a_r\} \) the alphabet. Now that “only” a single letter was being written on each page, he called them cells. So an action consists of

1. reading the character written in the current cell;
2. possibly writing a new character in the current cell;
3. possibly moving one position to the left or right on the tape thereby changing the current cell accordingly;
4. possibly changing states.

A computational process can then be described by giving the particular rules to be followed for the calculation at hand. To specify this fully, a few more details are needed. One begins with the information on which one is calculating, the input, written on a leftmost segment of the tape, and the mind (now called the finite control) in an initial state (conventionally \( q_1 \)). The computation finishes with the result, the output, written on a leftmost segment of the tape, or sometimes right after the input, and the finite control in a final state (conventionally \( q_f \)).

Turing’s thesis was that these machines captured everything that could be done computationally. Certainly, anything that can be done on a computer can in principle be done on a Turing Machine, albeit really inefficiently (the slowdown is by “only” a polynomial factor). But why study them if they are horribly inefficient? The reason is that they are conceptually much simpler, and so they are very useful in understanding what is computationally possible, both in terms of whether something can be computed at all, and whether something can be computed efficiently (e.g. polynomial rather than exponential time).
There is one more detail to mention. As Turing did not want to put any fixed bound on the length of a computation, although any computation to be useful would have to end eventually, he imagined that the tape of cells was unending or infinite to the right, and computation would use as much of the tape as needed.

At this point we have described a class of computing devices, one for each task we might wish to perform. Turing’s next critical observation was that one Turing Machine, called the Universal Machine, suffices. Suppose we want to carry out the computation of Turing Machine \( M \) on some input \( x \), say. Then the Universal Machine, \( U \), receives two pieces of input: a description of \( M \) (i.e. its states and alphabet together with the rules for its actions or moves), and \( M \)’s input \( x \). Then \( U \) carries out \( M \)’s computation as follows: at each step it consults the description of \( M \) to decide the next action \( M \) would take and then it does it. One detail to note is that \( U \) has to write down \( M \)’s current state as its finite control will not be large enough to remember the states of every possible Turing Machine \( M \).

\( U \) is effectively a general purpose computer: \( M \)’s description amount to a program, and \( M \)’s input corresponds to the input to the program.

**Exercises**

1. Show that a binary alphabet suffices. That is, given Turing Machine \( M \) with alphabet \( A \), create a new Turing Machine \( M_b \) with a binary alphabet that carries out the same computation as \( M \).

   Hint. Encode each character of \( A \) using \( \log A \) binary symbols, and make \( M_b \)’s finite control sufficiently larger than that of \( M \) so that it can “read” and “write” a character of \( M \).

2. This problem shows how to count out a number larger than can be kept in mind.

   Design a Turing Machine with a 4 character alphabet \( A = \{0, 1, *, b\} \) (\( b \) for blank, representing an unwritten location). Suppose the input is a number \( n \) written in binary at the left end of the tape; at this point the rest of the tape is blank, i.e. every other cell stores a \( b \). Design a Turing Machine to write \( n * \) characters after the input number. You may alter the input number as the computation proceeds. Also assume you are given a procedure (a collection of states and actions) that subtracts 1 from a positive number written in consecutive cells at the left end of the tape. Finally, describe the Turing Machine in reasonably high level terms such as: move to the right while reading cells with a ‘*’ until a cell without a ‘*’ is reached.