§7. The halting problem. We now revisit the discussion of §1.4, using the tools we have built up in the meantime. We will develop rigorous versions of the ideas and proofs that we saw earlier. The technical arguments should always be seen in the light of the intuition provided by the earlier informal arguments.

Recall that a *binary* Turing machine is one that has input alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ and tape alphabet $\Gamma = \{\mathtt{0}, \mathtt{1}, \text{ƀ}\}$. In Note 6 it was claimed that we lose nothing in generality by restricting our attention to binary Turing machines. Before proceeding further we should substantiate that claim.

Lemma 7.1 *For every Turing machine M with input alphabet $\{\mathtt{0}, \mathtt{1}\}$, there is a binary Turing machine $\widehat{M}$ that is equivalent to M: on every input, $\widehat{M}$ halts if and only if M halts, and $\widehat{M}$ accepts if and only if M accepts. (Note that the tape alphabet of M is unrestricted.)*

proof. Suppose the tape alphabet of M is Γ, and let $k = \lceil \lg |\Gamma| \rceil$. The basic idea of the proof is to encode each element of Γ (other than the blank symbol) as a binary string of length k. The blank symbol itself will be encoded as a string of k blank symbols[19]. Thus, we can imagine the tape of $\widehat{M}$ as being divided into k-square *blocks*, each of which represents a single tape square of the machine M. We assume that the symbol $\mathtt{0}$ receives code $\mathtt{0}^k$, and the symbol $\mathtt{1}$ receives code $\mathtt{0}^{k-1}\mathtt{1}$.

The first action of $\widehat{M}$ is to encode its input, using the substitutions $\mathtt{0} \to \mathtt{0}^k$ and $\mathtt{1} \to \mathtt{0}^{k-1}\mathtt{1}$. To encode a single symbol on the tape, $\widehat{M}$ positions its head over that symbol and performs the following operation $k-1$ times: shift the tape contents, from the tape head up to the first blank symbol, right one place, and fill the resulting gap with the symbol $\mathtt{0}$. To encode the entire input, $\widehat{M}$ simply repeats this procedure for each symbol of the input word. (The first symbol of the input must be marked in some way so that $\widehat{M}$ can later return its head to the leftmost square of the tape in readiness for the simulation proper.)

The simulation of M can now commence in earnest. To simulate a single move of M, the machine $\widehat{M}$ proceeds as follows. First, $\widehat{M}$ performs a sequence of $k-1$ steps in which it remembers the k symbols in the current block (the block that encodes the scanned symbol of M). This symbol, together with the current state of M (which $\widehat{M}$ has remembered in its finite control), determines the transition of M. In a further $2k-1$ steps, $\widehat{M}$ can overwrite the current block with an encoding of the new symbol of M, move to the adjacent block (left or right as appropriate), and remember the new state of M in its finite control. If M ever accepts then so does $\widehat{M}$; if M ever sticks then so does $\widehat{M}$. □

[19] We assume, without loss of generality, that $|\Gamma| \geq 2$ so that $k \geq 1$. The case $|\Gamma| = 1$ means that the input alphabet Σ is empty! Alternatively we can take $k = \max(1, \lceil \lg |\Gamma| \rceil)$.

In NOTE 6, a technique was presented for encoding binary Turing machines as words over the alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{B}, \mathtt{*}\}$. (That encoding did not specify the final state; this issue may be dealt with by insisting that the final state always receives the binary code for 1.) By replacing each of these four symbols by binary codes according to the correspondences $\mathtt{0} \to \mathtt{00}$, $\mathtt{1} \to \mathtt{01}$, $\mathtt{B} \to \mathtt{10}$, and $\mathtt{*} \to \mathtt{11}$, it is clear that we can encode binary Turing machines as words over the alphabet $\{\mathtt{0}, \mathtt{1}\}$. The point here is that (encodings of) Turing machines and their inputs are formally identical objects: words over the alphabet $\{\mathtt{0}, \mathtt{1}\}$. (Of course a modern digital computer displays the same phenomenon; programs and data are all represented by suitably formatted sequences of bits.). We shall use the notation $\langle M \rangle$ to stand for the binary encoding of machine M.

The *halting problem* is the following:

INSTANCE: A binary Turing machine M, and an input $x \in \{\mathtt{0}, \mathtt{1}\}^*$.

QUESTION: Does M halt on input x?

This way of presenting a decision problem as a yes/no question is intuitively appealing, but for complete precision we may re-express the halting problem as a problem in language recognition. Define the language $L_{\text{halt}} \subset \{\mathtt{0}, \mathtt{1}, \mathtt{\$}\}^*$ by

$$L_{\text{halt}} = \{\langle M \rangle \mathtt{\$} x \mid x \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } M \text{ halts on input } x\}.$$

Thus, a typical word in the language L_{halt} is formed from a pair of binary sub-words separated by a dollar symbol; the first sub-word is interpreted as the encoding of a Turing machine M, and the second as an input x to that machine. A word with the appropriate format is deemed to be a member of L_{halt} if M halts on input x. Notice that many words in $\{\mathtt{0}, \mathtt{1}, \mathtt{\$}\}^*$ are excluded from L_{halt} for the trivial reason that they have the wrong format: the number of dollar symbols is other than one, or the first binary word is not the encoding of a valid Turing machine (of course it is a simple matter to recognize such excluded words).

LEMMA 7.2 *The language L_{halt} is recursively enumerable.*

PROOF. In fact L_{halt} is accepted by a Turing machine M'_u which is only a slight modification of the universal Turing machine M_u of NOTE 6. On input $w \in \{\mathtt{0}, \mathtt{1}, \mathtt{\$}\}^*$, the machine M'_u first checks that w is of the form $w = \langle M \rangle \mathtt{\$} x$ for some valid Turing machine M. If the format of w is incorrect, M'_u immediately halts without accepting; otherwise, M'_u simulates the computation of M on input x and accepts if the computation terminates. □

Unfortunately, the machine M'_u is not a convincing solution to the halting problem: if the machine M does *not* halt on input x then M'_u simply loops forever, never

providing a definite answer. What we really seek is a machine that halts on all inputs, accepting if M halts on input x, and rejecting if M does not halt on input x.

We say that a language L is *recursive* if L is accepted by a Turing machine that halts on all inputs. Then, a yes/no problem is *decidable* if its associated language (the set of all yes-instances) is recursive, and *undecidable* otherwise. The question of whether there exists an effective solution to the halting problem can thus be phrased in two ways: (i) is the halting problem decidable? (ii) is the language L_{halt} recursive? These are equivalent, but the latter is more exact since the language L_{halt} was precisely defined, whereas the statement of the halting problem left open the detailed encoding of problem instances. The *decision problem* (i.e., yes/no problem) view is convenient precisely because it omits this often unnecessary detail.

The theorem which follows puts the discussion of §1.4 on a rigorous footing and dashes all hope of an effective solution to the halting problem. It is the most important result in the *Computability and Intractability* module.

THEOREM 7.1 *The language L_{halt} is not recursive.*

PROOF. We shall assume to the contrary that L_{halt} *is* recursive, and derive a contradiction.

So let M be a Turing machine that accepts the language L_{halt} and halts on all inputs. First we observe that there is a Turing machine M', with input alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{\$}\}$, which has the following behaviour:

$$M' \text{ halts on input } x \iff M \text{ rejects input } x. \tag{1}$$

The machine M' is obtained from M by replacing the accepting state of M by a non-accepting 'looping state'. Once M' enters the looping state, it continues in that state for ever. The behaviour of M' is exactly the same as M except that, at the very point that M would accept its input, M' enters an infinite loop. Thus the behaviour of M' satisfies (1).

Now observe that there is a Turing machine M'', with input alphabet $\{\mathtt{0}, \mathtt{1}\}$, which behaves as follows:

$$M'' \text{ halts on input } x \iff M \text{ rejects input } x\mathtt{\$}x. \tag{2}$$

Again, the machine M'' is a simple modification of the previous machine M'. When presented with input x, the machine M'' duplicates the input to produce a tape that reads $x\mathtt{\$}x$. Then M'' positions its head over the leftmost tape square and behaves exactly like M'. Since M' satisfies (1), it must be the case that M'' satisfies (2).

Using Lemma 7.1, we may transform M'' into an equivalent binary Turing machine. Once this is done, we may legitimately run M'' on its own description $\langle M'' \rangle$. What happens if we do this? From (2) we see immediately that

$$M'' \text{ halts on input } \langle M'' \rangle \iff M \text{ rejects input } \langle M'' \rangle \mathtt{\$} \langle M'' \rangle.$$

We now have the required contradiction. On the one hand, if M'' halts on input $\langle M'' \rangle$ then M rejects $\langle M'' \rangle \$ \langle M'' \rangle$, which is a contradiction since $\langle M'' \rangle \$ \langle M'' \rangle \in L_{\text{halt}}$ and M is supposed to accept the language L_{halt}. On the other hand, if M'' does *not* halt on input $\langle M'' \rangle$, then M accepts $\langle M'' \rangle \$ \langle M'' \rangle$, which is again a contradiction, since $\langle M'' \rangle \$ \langle M'' \rangle \notin L_{\text{halt}}$. We are therefore forced to reject our initial assumption, which was that L_{halt} is recursive. □

An interesting feature of the above proof is its *constructive* nature. Suppose a software company came to you with a Turing machine M, claiming that it 'solved' the halting problem. You could immediately construct a problem instance on which M is guaranteed to fail. This counterexample is simply the instance $\langle M'' \rangle \$ \langle M'' \rangle$, where M'' is constructed from M by the procedure described in the proof. (It is important to observe, in this context, that Lemma 7.1, too, is constructive.) Of course such a company is more likely to offer you a program written in a high level language but the same construction applies (as we saw in §1.4; indeed the observation follows from the Theorem together with Turing's thesis).

§7.1. An explosive function. Suppose M is a binary Turing machine and $x \in \{0, 1\}^*$ an input. If M halts on input x, let $T(M, x)$ denote the number of transitions made by M on input x before halting. If M does not halt on input x, then $T(M, x)$ is undefined. Define the function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(n) = \max \{T(M, x) \mid M \text{ halts on input } x, \text{ and } \langle M \rangle \$x \text{ has length } n\}.$$

For completeness, set $f(0) = 0$. Note that the function f is perfectly well defined.

Now suppose that there is a Turing machine transducer that computes the function $f(n)$. Then it would be possible to construct a Turing machine M_{halt} that accepts the language L_{halt}, and halts on all inputs. The proposed machine M_{halt} has two tapes. On input $\langle M \rangle \$x$, the machine M_{halt} writes n, the length of the input, on its second tape. It computes $f(n)$, leaving the answer on its second tape. Then M_{halt} proceeds to simulate, on its first tape, the computation of M on input x; after each step of the simulation, M_{halt} decrements the counter on its second tape. If the simulation terminates before the counter reaches zero, then M_{halt} accepts its input; if the simulation is still in progress when the counter reaches zero, then M_{halt} halts without accepting. Observe that if M has made $f(n)$ transitions without halting, then M will never halt.

But the language L_{halt} is not recursive, and the machine M_{halt} cannot exist. So we must discard our assumption that the function f is computable. Now suppose that $\widehat{f} : \mathbb{N} \to \mathbb{N}$ is any function that *dominates* f, i.e., that satisfies $\widehat{f}(n) \geq f(n)$ for all n. The argument used above to establish that f is not computable applies equally to the function $\widehat{f}$; thus we deduce that *no* function which dominates f can be computable. Stated more dramatically, the function f *grows faster than any computable function*!