1 Introduction

We finally name the class of languages that are accepted by TMs – in fact, we name two such classes. We also consider a *universal Turing machine*, the closest thing to a 'real' computer that we have seen so far.

2 Recursive and recursively enumerable languages

A language, L is *recursively enumerable* if it is the language accepted by some Turing machine M. That is, $w \in L$ if and only if M halts on input w in an accepting state. Note that if we process some word $v \notin L$ with M then M might halt in a non-accepting state, might crash, or might simply run forever. This situation is rather unsatisfactory – we know that if a 'yes' answer is coming it will arrive eventually, but if the answer is 'no' we may never be sure.

That makes it natural to define a (possibly) stronger notion: a language, L, is *recursive* if it is the language accepted by some Turing machine M that *halts on all inputs*. Now we are guaranteed a yes or no answer.

Theorem 2.1. *If both L and $\Sigma^* \setminus L$ are recursively enumerable, then L is recursive.*

Proof. Let M_L accept L and M_{cL} accept $\Sigma^* \setminus L$. We use these two machines to put together a multi-tape machine. Given an input w we first copy w onto another tape. Then we run M_L on one tape in parallel with M_{cL} on the other (the states of this new machine correspond to pairs of states of the two given machines, so there are still only finitely many of them). As soon as one of the two simulations halts, we can halt and announce the status of w. Since one of the two simulations is guaranteed to halt (M_L if $w \in L$ and M_{cL} otherwise) this is sufficient. □

3 Language enumerators

There are certainly circumstances in which we might be more interested in making a list of the elements of a language L rather than recognizing membership in L (for instance, a

list of primes, a list of possible solutions to some optimization problem, ...). A machine M *enumerates* L if it produces on some output tape (which always moves to the right, though not necessarily at every computation step) a sequence:

$$u_1 \# u_2 \# u_3 \# \cdots$$

where $\#$ is an extra symbol (not from the alphabet of L) and $w \in L$ if and only if $w = u_i$ for some i. In other words, the machine just lists the elements of L in some order, and possibly with repetitions, but every element gets listed eventually.

Proposition 3.1. *If there is a TM that enumerates L, then there is a TM that enumerates L without repetitions.*

Proof. Take a TM enumerating L and augment it with another output tape – the 'real' output tape. Each time M writes a word u_k on its output tape, pause M and use a separate control module to check whether $u_k = u_j$ for some $j < k$. If so, just restart M. If not, write u_k to the 'real' output tape (followed by a $\#$) and then restart M. □

Of course the main result (and the reason for the name 'recursively enumerable') is:

Theorem 3.2. *A language L is enumerated by some TM if and only if it is recursively enumerable.*

Proof. Suppose that L is enumerated by M. To build a machine T that accepts M simply add an extra input tape on which an input word w is written. Then start M. Each time M produces a new word u_i check whether $u_i = w$. If so, halt and accept. If not, carry on. Thus, if $w \in L$ we halt and accept, and otherwise we do not.

Now suppose that L is recursively enumerable, accepted by some machine T. To build an enumerator M we use the 'simulate for a fixed number of steps on each possible input of a fixed length' idea. That is, we build a machine M that successively runs T for k steps on all input sequences of length at most k for k from 1 to $\cdots$. Each time we spot an acceptance we add it to the output tape. Thus, we eventually add all the words that T accepts to the output tape and no others. □

The problem with language enumerators (as with recursively enumerable languages) is that if we haven't seen a word yet we have no indication whether we ever will. If we could enumerate the strings of a language in order of their length, then that would be a different story – once we saw a string longer than the one we were interested in we would know that waiting any longer would be a forlorn hope.

Say that L *can be enumerated by length* if there is an enumerator M for L which first lists all the strings of length 0 in L, then all the strings of length 1, then of length 2, and so on, i.e.M enumerates L by length. Not surprisingly:

Theorem 3.3. *A language L can be enumerated by length if and only if it is recursive.*

Proof. If L is recursive, take a machine M that accepts it and halts on all inputs. Run it on each string of length 0, then each string of length 1 and so on. Each time a string is accepted, add it to an output tape. The resulting machine enumerates L by length.

Conversely, suppose that L can be enumerated by length. If L is finite then it is regular and so certainly recursive, and there is nothing to prove. Otherwise, take the enumerator by length and a desired input word w. Wait until either w or some longer word appears, and accept or reject accordingly. □

4 Representing machines as strings

In several of the preceding arguments we have been piecing together new machines from old in an *ad hoc* fashion as required. There's nothing wrong with that, but effectively we're treating machines as modules here – and it would seem sensible to perhaps represent them in some uniform fashion (then we might feel more comfortable about adding certain operations 'erase the working tape and write the first string of length 6 on it, then run M for 6 steps')

A first step in that process is to come up with a representation of (basic) TMs as strings – and there's nothing particularly special about the representation we might choose – in practice, anything vaguely sensible will do. Since we're worried more about theoretical rather than practical arguments we'll just take a very simple representation.

We'll use a set alphabet of $\{0, 1\}$. The 0 symbol will be used only as punctuation, so we'll effectively encode everything else in unary. What do we need to encode?

- An actual alphabet Γ. Call the elements of Γ, a_1 through a_n and encode a_i as 1^i (a string of i 1's).
- A set of states. Call the states q_1 through q_m and encode q_j as 1^j
- The directions. Call L, 1 and R, 11.
- The transitions. Transitions can be thought of as 5-tuples $(q_{\text{start}}, a_{\text{read}}, a_{\text{write}}, d_{\text{move}}, q_{\text{finish}})$. So, simply encode these as the encodings of the individual parts separated from one another by single 0's.
- The complete set of transitions. Take each transition, encode it, and separate it from the next by 00.
- The start/finish of the encoding. Add a 000 at the beginning and at the end.

For convenience, having done all this to a Turing machine M we'll call the resulting string the *representation* of M and denote it by $R(M)$.

5 A universal Turing machine

Now that we can represent a Turing machine by a string (think 'program') we can design a so called *universal Turing machine*, U. It's easiest to think of U as a multi-tape machine. Its input or program tape is initially assumed to be of the form $R(M)w$ where M is a TM and w an input word. The job that U is meant to do is to simulate the operation of M on w. This is easy enough. First, w is copied to some working tape. Then a third tape, the state register, is initialized to '1', representing the 'current state' (assumed to be q_1 of M). Now operation proper can begin – the program tape is scanned for a transition from the current state using the current symbol of w. Once that is found, an appropriate update is made to the working tape and the state register. Then the next step is simulated ...

If M halts on w then U will halt on $R(M)w$. In particular, if we assume that U accepts by halting, the language accepted by U is:

$$\textsc{Halt} = \{R(M)w \mid M \text{ halts on } w\}.$$

So, the language HALT, also known as the halting language, is recursively enumerable.

This makes sense – we can recognise *positive* instances of the problem "does M halt on w" (where both M and w are parameters) simply by simulating M and answering 'yes' if it halts. Of greater interest is whether we can also somehow recognise *negative* instances of the same problem.

6 Exercises

1. Consider the language $\{a^n \mid n \text{ is composite}\}$.
   - Describe a simple mechanism for accepting this language on a normal two-tape Turing machine, with an input tape and an oracle tape.
   - Do the same, but with a "non-deterministic transitions" two-tape machine.
2. Consider the following problem: given two input tapes, one containing a string b^n, the other containing input of the form $a^{m_1}\#a^{m_2}\#\cdots\#a^{m_k}$ (i.e. a sequence of blocks of a's), determine whether some subset of the blocks of a's is of total length n (this is called the SUBSET-SUM problem). Show that a three-tape TM with oracle tape makes this problem almost trivial. Likewise model it in a TM with non-deterministic transitions. Finally, consider the difficulties in dealing with it deterministically.
3. (Review) Remember that regular languages form a subset of the context-free languages, which in turn form a subset of the recursive languages, and those are a subset of the recursively enumerable languages (though, at the moment, we can't be sure that there are any recursively enumerable languages that are not recursive). To prove that a language is regular, we can either design a finite state automaton that accepts it, or show that it is generated by a regular language. To prove that a language is not regular, we usually use the pumping lemma for regular languages or the Myhill-Nerode theorem. Similarly, to prove that a language is context free, we either show that it is accepted by a push down automaton, or give a context free grammar for it. Again, to prove a language is not context free, we use the pumping lemma for a context free languages. To prove that a language is recursive, we design a TM which halts on all inputs and accepts it. So far, we have no mechanisms for proving that a language is not recursive. For each of the following languages determine exactly which type it is, and prove it (so, for instance if you are claiming it is context-free, you should both give a grammar/PDA and an argument that it is not regular.)

   (a) $L = \{a^{3k}ba^{2l} \mid 0 \leq k, l\}$

(b) $L = \{a^{3k}ba^{2k} \mid 0 \leq k\}$

(c) $L = \{a^{k^2}ba^{k} \mid 0 \leq k\}$