Notes on Computer Theory Last updated: September, 2015

Turing Machines, diagonalization, the halting problem, reducibility

1 Turing Machines

A Turing machine is a state machine, similar to the ones we have seen until now, but with the addition of an infinite memory space on which it can read and write. The memory is modeled as an infinite tape of individual cells, and, in addition to the machine states that we have so far seen, we will also include in our state information the location of a tape "head." When determining each state transition, the Turing machine can read the character at the location of the tape head, choose to replace the character in that cell with some other character, and move the head right or left.

Formally, a Turing machine is made up of $(\Gamma, Q, q_{\mathsf{start}}, q_{\mathsf{a}}, q_{\mathsf{r}}, \delta)$ where Q is the finite set of states, Γ is the alphabet, which we assume includes a special "blank" symbol, used to represent empty cells on the tape, $q_{\mathsf{start}} \in Q$ is the start state, and $q_{\mathsf{a}}, q_{\mathsf{r}} \in Q$ are special accept states and reject states. δ is a transition function, $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. The input is the current state of the machine, and the character currently in the cell where the tape head is found. The output is a new state, a character that replaces the content of the current tape cell, and a direction for the head to move.

Note that we require explicit accept and reject states for Turing machines, because, unlike with automata, there is no specific halting condition with a Turing machine; the machine is allowed to repeatedly scan the the memory tape, including the input to the computation. The accept and reject states are trap states, and the machine halts its computation if it ever enters one of these two states. As before, we are interested in the languages that are decided by Turing machines. However, because the machine could possibly get stuck in an infinite loop, never entering either halting state, a subtlety arises here that did not occur in the case of automata. If there exists any input on which the Turing machine M fails to halt, then M does not decide any language. That is, we say M decides the language L if and only if 1) M enters the accept state for every $x \in L$, and 2) M enters the reject state for every $x \notin L$. That said, even if M does run forever on some inputs, it is still sometimes useful to describe the set of strings that M halts and accepts. We say that M *recognizes* L, if M halts and accepts every $x \in L$. Note that if M decides L, it also recognizes L, but the opposite is not necessarily true.

It is also useful to note that Turing machines are capable of more than just deciding whether a string belongs to a language. The machine can also halt with certain output on its tape. The output of M on input x, denoted $M(x)$, is the binary string contained on the tape (before the trailing blank symbols) when the machine halts. (When there is no output tape, then the output is '1' if M halts in state q_{a} and the output is '0' is M halts in state q_{r}.) A Turing machine M *computes a function* $f : \{0,1\}^* \to \{0,1\}^*$ if $M(x) = f(x)$ for all x. Assuming f is a total function, and so is defined on all inputs, this in particular means that M halts on all inputs. We say a function f is *computable*, if there exists some Turing machine that computes f.

1.1 Comments on the Model

(Jonathan Katz) Turing machines are *not* meant as a model of modern computer systems. Rather, they were introduced (before computers were even built!) as a mathematical model of *what computation is*. Explicitly, the axiom is that "any function that can be computed in the physical world, can be computed by a Turing machine"; this is the so-called *Church-Turing thesis*. (The thesis cannot be proved unless one can formally define what it means to "compute a function in the physical world" without reference to Turing machines. In fact, several alternate notions of computation have been defined and shown to be equivalent to computation by a Turing machine; there are no serious candidates for alternate notions of computation that are *not* equivalent to computation by a Turing machine. See [1] for further discussion.) In fact, an even stronger axiom known as the *strong Church-Turing thesis* is sometimes assumed to hold: this says that "any function that can be computed in the physical world, can be computed with at most a polynomial reduction in efficiency by a Turing machine". This thesis is challenged by notions of *randomized* computation that we will discuss later. In the past 15 years or so, however, this axiom has been called into question by results on *quantum computing* that show polynomial-time algorithms in a quantum model of computation for problems not known to have polynomial-time algorithms in the classical setting.

There are several variant definitions of Turing machines that are often considered; none of these contradict the strong Church-Turing thesis. (That is, any function that can be computed on any of these variant Turing machines, including the variant defined earlier, can be computed on any other variant with at most a polynomial increase in time/space.) Without being exhaustive, we list some examples (see [1, 2] for more):

- One may fix Γ to *only* include $\{0, 1\}$ and a blank symbol.
- One can allow the machine to have k tapes, rather than 1 tape, with k tape heads reading at each step. One can also allow a specially designated input tape and output tape.
- One may allow the tape heads to stay in place, rather than only moving left or right.
- One can allow the tapes to be infinite in both directions, or two-dimensional.
- One can allow random access to the work tapes (so that the contents of the ith cell of some tape can be read in one step). This gives a model of computation that fairly closely matches real-world computer systems, at least at an algorithmic level.

The upshot of all of this is that it does not matter much which model one uses, as long as one is ok with losing polynomial factors in the runtime. On the other hand, if one is concerned about "low level" time/space complexities then it is important to fix the exact model of computation under discussion. (We will discuss runtime and space requirements later.) For example, the problem of deciding whether an input string is a palindrome can be solved in time $O(n)$ on a two-tape Turing machine, but requires time $\Omega(n^2)$ on a one-tape Turing machine.

1.2 An example

We take an example directly out of Sipser's book [3]. The turing machine M_L accepts the language $L = \{0^{2^n} | n \geq 0\}$. M_L uses the alphabet $\Gamma = \{\sqcup, 0, x\}$. Recall that the transition function maps from $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. Make sure you understand how to map this formalism onto the notation in the Figure below.

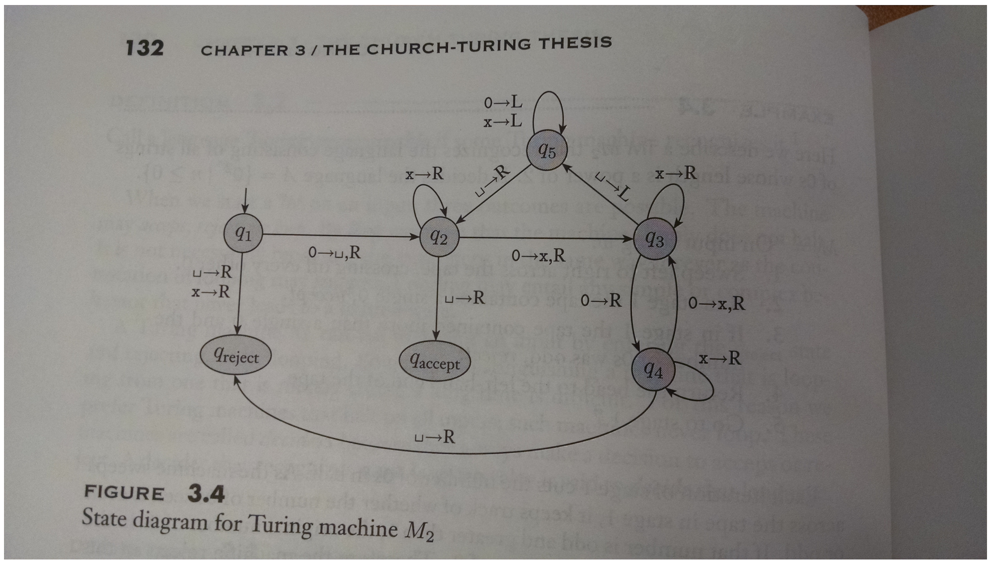

Figure 1: M_L accepting language $L = \{0^{2^n} | n \geq 0\}$, taken from Sipser's book [3].

Intuitively, what this machine does is: a) it checks if there are no "0"s on the input tape (q_1) and rejects if this is the case. Otherwise, it scrolls right until it finds the second "0" on the tape (q_2) and crosses this off by replacing it with an x. If it cannot find a second "0", then there is exactly one "0" remaining on the tape, and it accepts (also q_2). After that, it crosses off every other "0" (q_3 and q_4). If it reaches the end (i.e. by finding a $\sqcup$) when there are an odd number of "0"s, it rejects (q_4), otherwise, it rewinds to the first "0" and begins again (q_5). Note: there is one confusing thing in this state diagram! When q_1 replaces the first "0" with a $\sqcup$, this is to help the machine find the beginning of the tape during the rewinding. But you should think of this first "0" as still remaining, and *not* as thought it has been crossed off.

2 Diagonalization

We will use a proof technique called *diagonalization* to demonstrate that there are some languages that cannot be decided by a turing machine. This techniques was introduced in 1873 by Georg Cantor as a way of showing that the (infinite) set of real numbers is larger than the (infinite) set of integers. We will define what this means more precisely in a moment.

Definition 1 *Given two sets, A and B, and a function $f : A \to B$, we say that f is* injective *(one-to-one) if it never maps two different elements from A onto the same element from B: $\forall a, b \in A, a \neq b \Rightarrow f(a) \neq f(b)$*

Definition 2 *Given two sets, A and B, and a function $f : A \to B$, we say that f is* surjective *(onto) if every value in B has a pre-image under f: $\forall b \in B, \exists a \in A$ s.t. $f(a) = b$.*

If a function is both one-to-one and onto, then we say it is *bijective*, or a *correspondence*. Cantor proposed that we can use a correspondence to compare the sizes of infinite sets, the same way would finite sets. In particular, we are interested in determining whether some infinite set is the same size, or larger than the set of natural numbers $\mathcal{N} = \{1, 2, 3, \ldots\}$. If a set S has a correspondence with the natural numbers, i.e. $f : \mathcal{N} \to S$, we say that the set is *infinitely countable.*

Consider, for example, the set of even numbers $\{2, 4, 6, \ldots\}$. We might think that this set is smaller than the set of natural numbers, since it is fully contained within the set of natural numbers. But we can also see that there is a correspondence mapping from $\mathcal{N}$ to the set of even numbers: $f(a) = 2a$. (Verify for yourself that f is both one-to-one and onto.)

How about the set of positive rational numbers $\mathcal{Q} = \{\frac{m}{n} \mid m, n, \in \mathcal{N}\}$? Our first intuition might be that this set is "larger" than $\mathcal{N}$, but in fact we can again build a correspondence between the two sets. To do that, let's start by building a matrix of all positive rational numbers, placing any number with i in the numerator in the ith row, and any number with j in the denominator in the jth column.

$$\begin{pmatrix} \frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots \\ \frac{2}{1} & \frac{2}{2} & \frac{2}{3} & \frac{2}{4} & \frac{2}{5} & \cdots \\ \frac{3}{1} & \frac{3}{2} & \frac{3}{3} & \frac{3}{4} & \frac{3}{5} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Then, we go through the array, mapping each new item to one of the natural numbers. However, when doing that, we don't simply iterate over them row by row, or column by column, because each row and column has infinite size, and we'd never get past the first row or column! Instead, we move through the matrix along diagonals that go up from left to right: $\frac{1}{1}, \frac{2}{1}, \frac{1}{2}, \frac{3}{1}, \frac{1}{3}, \frac{4}{1}, \frac{3}{2}, \frac{2}{3}, \frac{1}{4}, \frac{5}{1}, \frac{1}{5}$. Note that in order to ensure that our mapping is one-to-one, we have to skip any repeated values. So, for example, we skip $\frac{2}{2}$, because we already included $\frac{1}{1} = \frac{2}{2}$. We also skipped $\frac{4}{2}, \frac{3}{3}$ and $\frac{2}{4}$ for the same reasons. The correspondence, f, simply maps the $a \in \mathcal{N}$ onto the ath element on this list. f is one-to-one because we have only included one "copy" of each equivalent rational number on the list, so no two natural numbers are mapped onto the same rational number. f is onto because there is no (positive) rational number left off of our list.

Theorem 1 *$\mathcal{R}$ is not countable.*

Proof We show this by contradiction. Suppose that $\mathcal{R}$ were countable, and let f be the correspondence with $\mathcal{N}$. We will show that there exists some $b \in \mathbb{R}$ that is not in the image of f, violating the assumption that f is onto. To do this, we start by ordering all the elements of $\mathbb{R}$ according to the values of their pre-images: $f(1), f(2), f(3), f(4), \ldots$. Because f is a correspondence, *all* elements of $\mathbb{R}$ must appear somewhere on this list. We now define a real number $b \in \mathbb{R}$ that does not appear on this list, demonstrating a contradiction. Recall that real numbers can be infinitely long. Choose the ith digit of b such that it is different from the ith digit of $f(i)$. Suppose that b appears in the jth position of this list (i.e. that $f(j) = b$). But this cannot be true, because we know that b differs from $f(j)$ in the jth position! This is called diagonalization, because we define the contradictory value by appropriately setting its value along the diagonal. ■

2.1 Unrecognizable languages

We use a similar technique to prove that there exists some language that cannot be decided. Actually, we can prove something even stronger. Recall, we say that a machine M *recognizes* L if it halts and accepts for every $x \in L$, and for every $x \notin L$, it either halts and rejects, or it runs forever. We will show that there exist languages that aren't even *recognizable.* We show this by proving that there are more languages than Turing machines: the set of Turing machines is countable, but the set of languages is not! We use the following lemmas and corollaries in our proof.

Lemma 2 *Any set $X \subseteq \mathcal{N}$ is countable.*

Corollary 3 *For any alphabet Σ, the set of strings Σ^* is countable.*

Corollary 4 *The set of all valid Turing machines is countable.*

Lemma 5 *The set of infinite binary sequences is uncountable.*

The proof of this lemma is essentially identical to the proof that $\mathbb{R}$ is not countable, so we omit it. Note that the key difference between the statement of Corollary 3 and this one is that is that no string in the set described in Corollary 3 can have infinite length, even though we place no limit on the size that any string in that set *can* have. In other words, for any string $w \in \Sigma^*$, there exists some constant c such that $w = \Sigma^c$.

Theorem 6 *There exist languages that are not Turing recognizable.*

Proof We already observed that the set of all Turing machines is countable. To show that the set of all languages is not countable, we give a correspondence to the set of infinite binary strings, and rely on the second lemma above to complete the proof. To describe this correspondence, we define a function f mapping each infinite binary string onto some language. We start with a fixed ordering on all binary strings: we list all strings of length 1 in lexicographic order, followed by all strings of length 2 in lexicographic order, etc. Then, for some infinite binary string w, we define the language $f(w)$ by including the jth binary string on this list in the language $f(w)$ if and only if the jth bit of w is 1. To see that f is one-to-one, consider two infinite binary strings $w \neq w'$, and, suppose they differ on the jth bit. $f(w) \neq f(w')$, since the two languages differ on whether they include the jth string from our canonical list of all strings. It is easy to verify that every language has a pre-image under f by simply defining the inverse function: given some language L, we can find w such that $f(w) = L$ by starting with the same canonical ordering on all strings, and letting the jth bit of w be 1 iff the jth string is in L.

Since there are more languages than there are Turing machines, it follows, at least intuitively, that there exists some language that is not recognized by any Turing machine. To formalize this, we actually require an additional lemma.

Lemma 7 *Let S be some infinite set, and let f be a surjective (onto) function $f : \mathcal{N} \to S$. Then there exists a bijection $g : \mathcal{N} \to S$.*

Proof f is onto S, but it is not necessarily one-to-one, which means there exist $a, b \in \mathcal{N}$ s.t. $a \neq b$ and $f(a) = f(b)$. Intuitively, we want to "re-map" f on one of these values so that they no longer collide. Since we have infinite space to deal with, we can simply choose the next available value, and "shift" everything down. More formally, to define g, we start by ordering the elements of S as follows. For every $s \in S$, let $m(s) = \min(\{j \mid f(j) = s\})$. Sort the elements of S according to $m(s)$. This ordering is complete, because f is a function, each element of $\mathcal{N}$ can only be mapped to a single element of S under f: if $s_i \neq s_j$, but $m(s_i) = m(s_j)$, it would follow that there exists some k such that $f(k) = s_i$, and $f(k) = s_j$. After ordering the elements of S this way, denote the ordered list as $s_1, s_2, \ldots$. We define $g(i) = s_i$.

We need to show that g is a bijection. To see that it is injective, note that none of the elements in S appear in our ordered list more than once, since $m(s)$ is unique for each s. To see that g is surjective, note that because f is surjective, $m(s)$ is well defined for every $s \in S$, so every element in S is included somewhere on our list; it follows that every element in S has a pre-image under g. ■

Consider the obvious function mapping the set of Turing machines onto the set of recognizable languages. (This function is not a correspondence — can you see why? — but it will suffice for our proof.) This function cannot be onto the set of *all* languages, or else, by the previous lemma, it would follow that the set of languages is countable. It follows that there exists some language without any pre-image under this function; this language is not recognized by any Turing machine. ■

2.2 Universal Turing Machines and Uncomputable Functions

(Jonathan Katz) An important observation (one that is, perhaps, obvious nowadays but was revolutionary in its time) is that *Turing machines can be represented by binary strings.* In other words, we can view a "program" (i.e., a Turing machine) equally well as "data", and run one Turing machine on (a description of) another. As a powerful example, a *universal* Turing machine is one that can be used to simulate any other Turing machine. We define this next.

Fix some representation of Turing machines by binary strings, and assume for simplicity that every binary string represents some Turing machine (this is easy to achieve by mapping badly formed strings to some fixed Turing machine). We will use $\langle M \rangle$ to denote the binary representation of the machine M. Consider the function $f(\langle M \rangle, x) = M(x)$. Is f computable? Perhaps surprisingly, f is computable. We stress that here we require there to be a *fixed* Turing machine U, with a fixed alphabet and a fixed set of states, that can simulate the behavior of an *arbitrary* Turing machine M that may use any number of states, and any size alphabet! A Turing machine computing f is called a *universal* Turing machine.

Note that the function $f(\langle M \rangle, x) = M(x)$ is a partial function, since in this context the given Turing machine M may not halt on the given input x and we leave f undefined in that case. To simplify things, we will consider the total function

$$f'(\langle M \rangle, x, 1^t) = \begin{cases} 1 & \text{if } M(x) \text{ halts and accepts within } t \text{ steps,} \\ 0 & \text{otherwise} \end{cases}$$

whose computability is closely linked to that of f.

Theorem 8 *There exists a Turing machine U such that (1) $U(\langle M \rangle, x, 1^t) = M(x)$ if M halts on x within t steps, and 0 otherwise. Furthermore, (2) for every M there exists a constant c such that the following holds: for all x, if $M(x)$ halts within t steps, then $U(\langle M \rangle, x, 1^t)$ halts within $c \cdot t \log t$ steps.*

We do not prove this theorem here, and instead refer to Arora and Barak [1]. Intuitively, though, the idea is that U can maintain the current state of M on its own work tape. U is given the binary representation of M as input, which includes M's transition function, δ. At each step,

U scans its work tape to fetch the current state of M, and the character currently pointed to by M's tape head. U can then scan the description of δ, and compute how M would update its own state. It then returns to the portion of the tape maintaining M's state and updates it accordingly. After each step, U will decrement a counter that starts at t: if the counter ever reaches 0, U will halt and reject.

Another natural possibility is to consider the (total) function

$$f_{halt}(\langle M\rangle, x) = \begin{cases} 1 & \text{if } M(x) \text{ halts with output 1} \\ 0 & \text{otherwise} \end{cases};$$

What about f_{halt}? Is it computable? By again viewing Turing machines as data, we can show that this function is *not* computable!

Theorem 9 *The function f_{halt} is not computable.*

Proof Say there is some Turing machine M_{halt} computing f_{halt}. Then we can define the following machine $M^*(\langle M\rangle)$:

> On input $\langle M\rangle$, compute $M_{halt}(\langle M\rangle, \langle M\rangle)$. If the result is 1, output 0; otherwise output 1.

Intuitively, M^* is given the description of a machine, $\langle M\rangle$. It duplicates the input, running M_{halt} on the duplicate input. Recall that M_{halt} looks at its first input, interprets it as a machine, and runs that interpreted machine on the 2nd input. In this case, the second input happens to *also* be the description of a machine. So running M_{halt} on the input $(\langle M\rangle, \langle M\rangle)$ amounts to asking "does M halt and accept its own description?" A strange, but perfectly reasonable question!

Now, consider and even stranger, but still perfectly reasonable question. What happens when we run M^* on *itself*? Consider the possibilities for the result $M^*(\langle M^*\rangle)$:

- Say $M^*(\langle M^*\rangle) = 1$. By the definition of M^*, this implies that $M_{halt}(\langle M^*\rangle, \langle M^*\rangle) = 0$. But, by the definition of M_{halt}, that means the first input, $\langle M^*\rangle$, does not halt and accept the second input, $\langle M^*\rangle$. In other words, $M^*(\langle M^*\rangle)$ does not halt with output 1, a contradiction.
- Say $M^*(\langle M^*\rangle) = 0$. By the definition of M^*, this implies that $M_{halt}(\langle M^*\rangle, \langle M^*\rangle) = 1$. But, by the definition of M_{halt}, that means the first input, $\langle M^*\rangle$, does halt and accept the second input, $\langle M^*\rangle$. In other words, $M^*(\langle M^*\rangle)$ halts with output 1, a contradiction.
- It is not possible for $M^*(\langle M^*\rangle)$ to never halt, since $M_{halt}(\langle M^*\rangle, \langle M^*\rangle)$ is a total function, and therefore halts on all inputs.

We have reached a contradiction in all cases, implying that M_{halt} as described cannot exist. ■

Remark: The fact that f_{halt} is not computable does *not* mean that the halting problem cannot be solved "in practice". In fact, checking termination of programs is done all the time in industry. Of course, they are not using algorithms that are solving the halting problem – this would be impossible! Rather, they use programs that may give *false negatives*, i.e., that may claim that some other program does not halt when it actually does. The reason this tends to work in practice is that the programs that people want to reason about in practice tend to have a form that makes them amenable to analysis.

We can actually visualize the proof that the halting problem cannot be decided as an example of diagonalization. To prove that L_{halt} is not decidable, we begin by assuming that it M_{halt} decides it, and we list the output of this machine on all possible inputs. Recall that it takes two inputs: $\langle M \rangle$ and w, where the first is the description of a turing machine, and the second is interpreted as the input to M. We will list the outputs of M_{halt} by listing all possible Turing machines on one side of a matrix, and all of their descriptions on the other side of a matrix. (Note that there is a correspondence mapping the set of Turing machines to the set of finite-length strings, so the second input to M_{halt} can always be interpreted as the description of some Turing machine. However, this point is not essential for the proof: we're simply only interested in the strings that represent Turing machines and therefore only list those along the top row.)

We can now list the output of M_{halt}:

$$\left(\begin{array}{cccccc} & \langle M_1 \rangle & \langle M_2 \rangle & \langle M_3 \rangle & \langle M_4 \rangle & \dots \\ M_1: & \mathit{accept} & \mathit{accept} & \mathit{reject} & \mathit{reject} & \dots \\ M_2: & \mathit{accept} & \mathit{reject} & \mathit{reject} & \mathit{reject} & \dots \\ M_3: & \mathit{reject} & \mathit{accept} & \mathit{reject} & \mathit{accept} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}\right)$$

We defined M^* to run $M_{halt}(\langle M \rangle, \langle M \rangle)$ and do the opposite. Since M^* is a valid turing machine, it must appear somewhere on the list we described above. We fill in its output in the table below:

$$\left(\begin{array}{ccccccc} & \langle M_1 \rangle & \langle M_2 \rangle & \langle M_3 \rangle & \langle M_4 \rangle & \dots & \dots \\ M_1: & \underline{\mathit{accept}} & \mathit{accept} & \mathit{reject} & \mathit{reject} & \dots & \mathit{accept} \\ M_2: & \mathit{accept} & \underline{\mathit{reject}} & \mathit{reject} & \mathit{reject} & \dots & \mathit{reject} \\ M_3: & \mathit{reject} & \mathit{accept} & \underline{\mathit{reject}} & \mathit{accept} & \dots & \mathit{accept} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \mathit{reject} \\ M^*: & \mathit{reject} & \mathit{accept} & \mathit{accept} & \mathit{reject} & \dots & \underline{???} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}\right)$$

Finally, we give a constructive proof that some languages cannot be recognized. (Note that our previous proof, which used the fact that the set of Turing machines is countable, did not leave us with the description of an actual language; it was only a proof of existence of such a language.) Consider the language $\overline{L}_{halt}$, which is the complement of L_{halt}. In other words, $\overline{L}_{halt} = \{\langle M \rangle, x \mid M \text{ does not halt and output 1 on input } x\}$. We will prove that this language is not recognizable by any Turing machine. We begin with the following theorem.

Theorem 10 *A language is decidable if and only if both it and its complement are Turing recognizable.*

Proof Let L be some decidable language. It follows immediately from the definitions that L is recognizable. To see that $\overline{L}$ is also recognizable, let M_L be the machine that decides L, and define $\overline{M}_L$ as the machine that runs M and reverses its output. In the other direction, suppose L and $\overline{L}$ are recognizable by M_L and $\overline{M}_L$ respectively. To decide L, we can construct a Turing machine that runs both of these machines in parallel, alternating the steps of each computation. (The alternation of steps is important, in case either one of these machines never terminates.) On input x, if $M_L(x)$ halts with output 1, then halt and output 1. If $\overline{M}_L$ halts and outputs 1, halt and output 0. ■

We have proven that L_{halt} is not decidable. However, note that it is recognizable, since on input $(\langle M\rangle, x)$ we can always simulate M on x: if M halts and outputs 1, then we do the same. (The fact that it is not decidable means that it might not halt when M does *not* accept x; in this case, our simulation would not halt either.) It follows that $\overline{L}_{halt}$ is not recognizable: otherwise, by the previous theorem, L_{halt} would be decidable.

3 Reducibility

Consider the following language $L_{A/R} = \{\langle M\rangle, x \mid M \text{ halts on input } x \text{ and outputs either 0 or 1}\}$. This looks very similar to L_{halt}, which we know is undecidable, so it is not surprising that this language is also undecidable. But how do we prove that? We can do it through a *reduction*: we demonstrate that if there is a Turing machine $M_{A/R}$ that decides $L_{A/R}$, then there is a Turing machine M_{halt} that decides L_{halt}. Since we already know that the latter statement is untrue, it follows that the former statement is untrue.

Theorem 11 *The language $L_{A/R}$ is undecidable.*

Proof We prove it by constructing M_{halt} that does the following.

```
$\underline{M_{halt}(\langle M\rangle, x) :}$
    1. Simulate $M_{A/R}(\langle M\rangle, x)$.
        (a) If it outputs 0, halt and output 0.
        (b) If it outputs 1, simulate $M$ on input $x$ until it halts. Output whatever $M$ outputs.
```

■

Theorem 12 *The language $L_{\mathsf{some}} = \{\langle M\rangle \mid M \text{ is a Turing machine that accepts some string }\}$ is undecidable.*

Proof We again assume that there exists some TM M_{some} deciding L_{some}, and demonstrate that this gives rise to a TM M_{halt} that decides L_{halt}. We describe M_{halt} as follows.

```
$\underline{M_{halt}(\langle M\rangle, x) :}$
    1. Write down a description of a TM $M'$ that modifies the behavior of $M$ as follows.
       $\underline{M' :}$
```

    (a) On input $y \neq x$, reject.
    (b) On input $y = x$, run $M(y)$ and output whatever it outputs.

2. Run $M_{\mathsf{some}}(\langle M' \rangle)$, and output whatever it outputs.

■

Note that $\langle M' \rangle$ described in the previous reduction has a particular property of interest. If $(\langle M \rangle, x) \in L_{halt}$, then $\langle M' \rangle$ accepts some string (namely, x), whereas if $(\langle M \rangle, x) \notin L_{halt}$, then $\langle M' \rangle$ rejects every string. There is a nice way of generalizing the reductions described above, using what is called a *mapping reduction.*

Definition 3 *Let L_1 and L_2 be two languages over alphabet Σ. A function $f : \Sigma^* \to \Sigma^*$ is a mapping reduction from $L_1 \leq_m L_2$ if f is computable by a Turing machine, and $\forall x \in \Sigma^*$, $f(x) \in L_2 \Leftrightarrow x \in L_1$.*

Now, to reduce L_1 to L_2, we can just describe a mapping reduction, f. The rest of the proof follows generically: suppose that M_2 decides L_2. Then, define M_1 that runs $M_2(f(x))$ and outputs the result. It is clear that M_1 decides L_1. If it is known that L_1 is undecidable, then, since f is shown to exist, it follows that M_2 must not exist, and that L_2 is not decidable either.

Let's revisit the reduction from L_{halt} to $L_{A/R}$, and this time prove the same result using a mapping reduction, f. We need to give a computable function f such that

$$(\langle M \rangle, x) \in L_{halt} \Leftrightarrow f(\langle M \rangle, x) \in L_{A/R}$$

Note that f in this case outputs the description of a Turing machine and its input. Specifically, on input $(\langle M \rangle, x)$, f outputs a $(\langle M' \rangle, x)$, where M' does the following.
$\underline{M'(\langle M \rangle, x):}$

1. Simulate M on input x.
    (a) If it outputs 1, halt and output 1.
    (b) If it outputs 0, loop forever.

To argue that reduction function, f, is computable by a Turing machine, we simply have to argue that a Turing machine can take $(\langle M \rangle, x)$ as input, and reliably halt with $\langle M' \rangle$ as output on its tape. Note that the Turing machine that does this, thereby computing f, does not have to simulate M. It just needs to take the *description* of M, (denoted $\langle M \rangle$), and modify the description of M to contain the extra loop in the case of rejection. Modifying the code of $\langle M \rangle$ can be done reliably, without risk of running forever, so f is computable. (Simulating M would give no such guarantee, since M might run forever.)

Let's now do the same with the reduction we built from L_{halt} to L_{some}. $f(\langle M \rangle, x) = \langle M' \rangle$, where M' is exactly as previously described: it rejects on all strings $y \neq x$, and runs $M(y)$ when $y = x$. If $(\langle M \rangle, x) \in L_{halt}$, then $\langle M' \rangle \in L_{\mathsf{some}}$, and if $(\langle M \rangle, x) \notin L_{halt}$, then $\langle M' \rangle \notin L_{\mathsf{some}}$. To argue that f is computable, note that it only needs to modify the description of $\langle M \rangle$ to perform the extra comparison, $x == y$, before proceeding to execute the original code of M.

Theorem 13 *The language $L_{EQ} = \{(\langle M_1 \rangle, \langle M_2 \rangle) \mid L(M_1) = L(M_2)\}$ is undecidable. (Here, we are $L(M)$ to denote the language* decided *by machine M.)*

Proof We first define a language $\mathsf{L}_\emptyset = \overline{L}_{\mathsf{some}}$. That is, $L_\emptyset = \{\langle M \rangle \mid M \text{ does not accept any string}\}$. Note that $L_\emptyset$ is not decidable (prove this to yourself). We show that L_{EQ} is not decidable by providing a mapping reduction from $L_\emptyset$ to L_{EQ}. Since $L_\emptyset$ is undecidable, it follows that L_{EQ} is also undecidable. We have to give a computable function f that maps $x \in L_\emptyset$ to $x' \in L_{EQ}$, and maps $x \notin L_\emptyset$ to $x' \notin L_{EQ}$. The function is as follows. Let $\langle M_2 \rangle$ be the description of a turing machine that rejects all strings. $f(\langle M_1 \rangle) = (\langle M_1 \rangle, \langle M_2 \rangle)$. To see that this is a good mapping reduction, first we argue that f is computable. This follows because f simply needs to write the description of 2 Turing machines to its output tape: one machine description it copies from its own input tape, and the other is a hard-coded description, $\langle M_2 \rangle$, which can just be stored in the states of the machine computing f.

We now argue that it preserves membership as it is supposed to. Suppose $x \in L_\emptyset$. This means that x describes a machine that accepts no inputs. In that case, the output of f is the description of two machines, both of which accept no input, so $f(x) \in L_{EQ}$. On the other hand, suppose $x \notin L_\emptyset$. In that case, x describes a machine that accepts *some* input. Since f outputs $(x, \langle M_2 \rangle)$, where x describes a machine that accepts some input, and $\langle M_2 \rangle$ describes a machine that accepts no input, it follows that the output of f is not in L_{EQ}. ∎

The existence of a mapping reduction suffices to show that L_{EQ} is not decidable. To repeat why this is, note that we can complete the argument generically, as follows. Suppose M_{EQ} decides L_{EQ}. Then, we claim that the following machine, $M_\emptyset$, decides $L_\emptyset$.

$\underline{M_\emptyset(\langle M \rangle) :}$

1. Run $M_{EQ}(f(\langle M \rangle))$, and output whatever it outputs.

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[2] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[3] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.

[4] R. Kannan. Towards separating nondeterminisn from determinisn. *Math. Systems Theory* 17(1): 29–45, 1984.

[5] W. Paul, N. Pippenger, E. Szemeredi, and W. Trotter. On determinism versus non-determinisn and related problems. FOCS 1983.

[6] R. Santhanam. On separators, segregators, and time versus space. *IEEE Conf. Computational Complexity* 2001.