Turing Machines and Mapping Reductions

Nikhil Sardana

December 2019

It's been a few weeks since our last lecture, and I'm sure you don't remember everything we talked about. So, Section 1 is a review of last meeting's material. Skip to Section 2 for material we didn't cover last meeting, with a few new diagrams. Skip to Section 3 for new material that wasn't on last meeting's handout. The schedule and materials are online at

`https://nikhilsardana.github.io/lectures`

1 Recap

We began our first lecture by asking "What is computation, anyway?" Computation is a vague notion—computers are everywhere, and we use them constantly, but they come in all different shapes and sizes, and they solve different problems. We would like to know if there are problems no computers can solve. And, we'd like to know which problems we can solve efficiently. So, it's important to find a general way of modeling computers, and simple way of framing computational problems. We'd like to think about a single model, and ask ourselves "Can this model solve problem x?", and have our answer generalize to all our real-world computers.

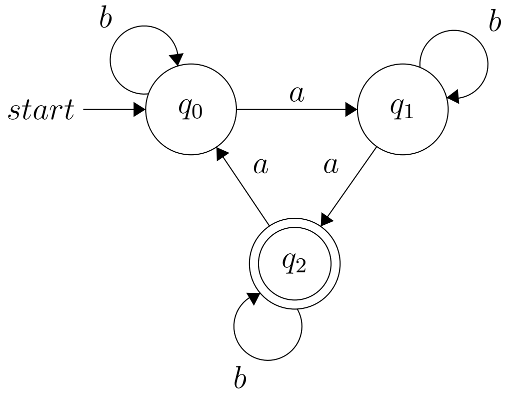
In our first lecture, we framed computational problems as a *language membership* problems: Given a set S and an input x, we want a model that can answer "Is x in S?" These sets can be quite complicated, so we can imagine any yes/no problem anyone could possibly ask a computer can be encoded as a question of language membership.

Recall the following definitions around languages:

- **Alphabets** are finite, nonempty sets of characters. For example, $\Sigma = \{a, b, c\}$.
- **Strings** are finite sequences of characters. For example, aa.
- **Languages** are sets of strings. For example, $\{a, aa, aaa, \dots\}$.
- ϵ denotes the empty string.
- A **Language over the alphabet** Σ is a set of strings made of characters from Σ.
- Σ^* is the set of all strings made from characters in Σ (this includes ϵ).

1.1 Finite Automata

We then introduced finite automata as a simple model for answering language membership queries. A language L has a corresponding automata D that *recognizes* L if D accepts a string if and only if the string is in the language L.



Every automata consists of a set of states, and a set of transitions between the states. We begin at the start state, and read a character of our input string s. We move to the appropriate state defined by a transition. We repeat this process, moving from state to state, consuming characters of s. If we run out of characters and end in an accepting state, we say that the automata accepts s, or equivalently, s is in the *language of the automata*. Otherwise, we say our automata rejects s, or s is not in the in the language of the automata.

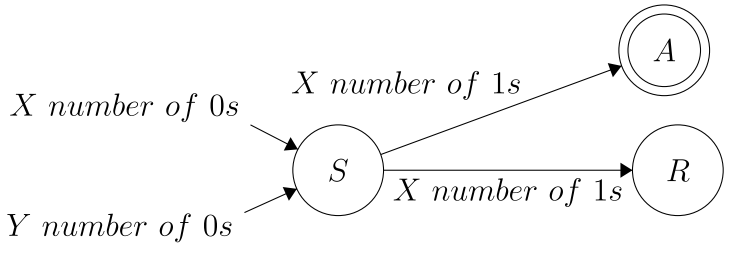
We introduced two types of automata last lecture. **Deterministic finite automata** have every transition defined, and every input string takes an explicit computation path. **Nondeterministic finite automata** do not have every transition explicitly defined, and an NFA computes language membership by taking every single possible path of computation for a given input simultaneously, and accepting if even a single path accepts.

Exercise 1.1. What is the language of the above DFA? What alphabet is this language over?

We proved that NFAs and DFAs are equivalently powerful—if there is an NFA that recognizes a language L, we can build a DFA that also recognizes L, and vice-versa.

1.2 Regular vs. Non-regular

We also saw that some languages do not have a corresponding finite automata. Recall that we looked at the language $L = \{0^n1^n \mid n \in \mathbb{N}\}$, and we realized that if there existed a DFA D for L, then D would need a different state for $0^1, 0^2, 0^3, \ldots$ and every 0^n for $n \in \mathbb{N}$. Since we can only have finitely many states in any DFA, there can't exist one for L.



Languages with corresponding finite automata are called **regular languages**, and the languages that have no corresponding finite automata are called **non-regular**. We can think about non-regular languages as having infinite equivalence classes, which is why they have no recognizing DFA.

The class of non-regular languages is huge—much larger than the class of regular languages, and it includes some languages we'd really like to be able to answer questions about. For example, HTML. I'd like to know if a file contains valid HTML or not, but HTML is famously non-regular. These are problems, basic problems, that our real-world computers can answer, but our finite automata model cannot. So, it's clear that finite automata are not the best model for our computers. To model non-regular problems, we need a more powerful model of computation. And so, we introduced the **Turing machine**.

1.3 Turing Machines

A Turing machine consists of a **tape**, which is infinitely long in one direction (we start at the left end) and a tape **head**. Initially, our input is written on the tape, followed by infinitely many blank tape cells, denoted by $\square$. Our input is a finite string, made up of characters from the **input alphabet** Σ.

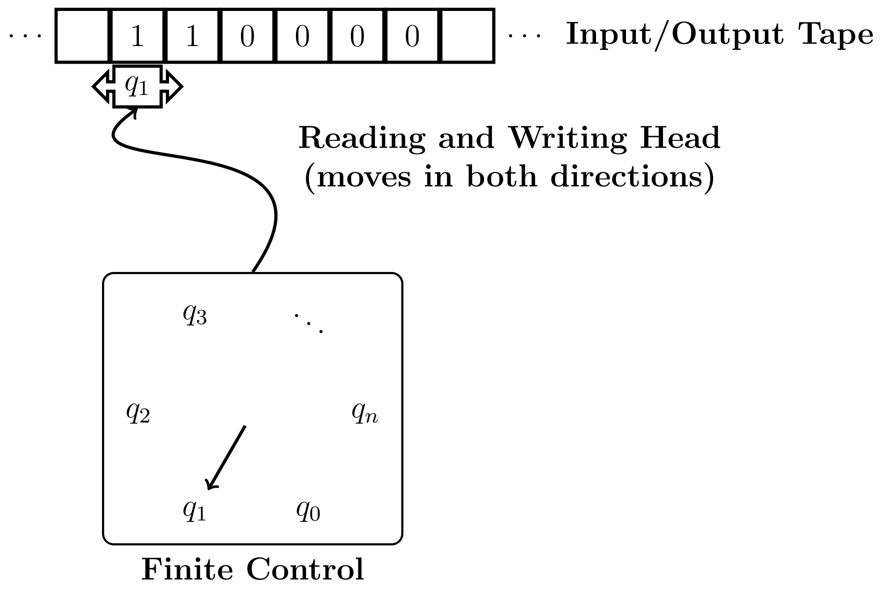
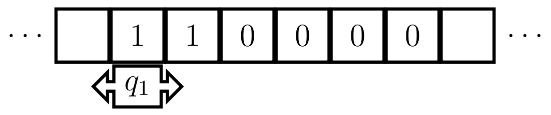


Figure 1: A Turing Machine with input alphabet $\Sigma = \{0, 1\}$ and head states $\{q_0, \ldots, q_n\}$.

The head of the Turing machine has finitely many states (just like the states of the DFA).

Our head starts at the left-most tape cell. At each step of our computation, the tape head reads the contents of a single cell, and changes state based on the cell's contents. Then, the head writes a character to the cell. Finally, the head moves either right or left.



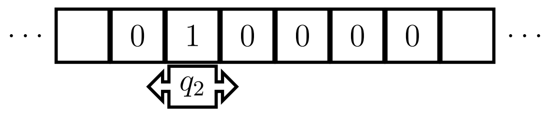


Figure 2: One step of the Turing machine. Between the first and second diagrams, the head read 1, changed state to q_2, wrote 0, and then moved right.

If our head ever reaches an accepting state q_a, then we immediately accept and our computation terminates. (This is unlike a DFA, in which we must finish our input before we accept.) Similarly, if we reach a rejecting state, then we immediately reject. If a Turing machine accepts or rejects an input, we say it **halts** or terminates. However, it is possible we never reach an accepting or rejecting state. Since we have to move right or left at every step, and our head never has to move to an accepting state, it's possible for a Turing machine to compute forever. We say a Turing machine **loops** when it fails to terminate.

Note: the head can write any character in the **tape alphabet** Γ, which contains Σ but can contain additional characters. This will allow us to write special symbols that we know don't occur naturally in the input, and let us know we've already seen a certain cell.

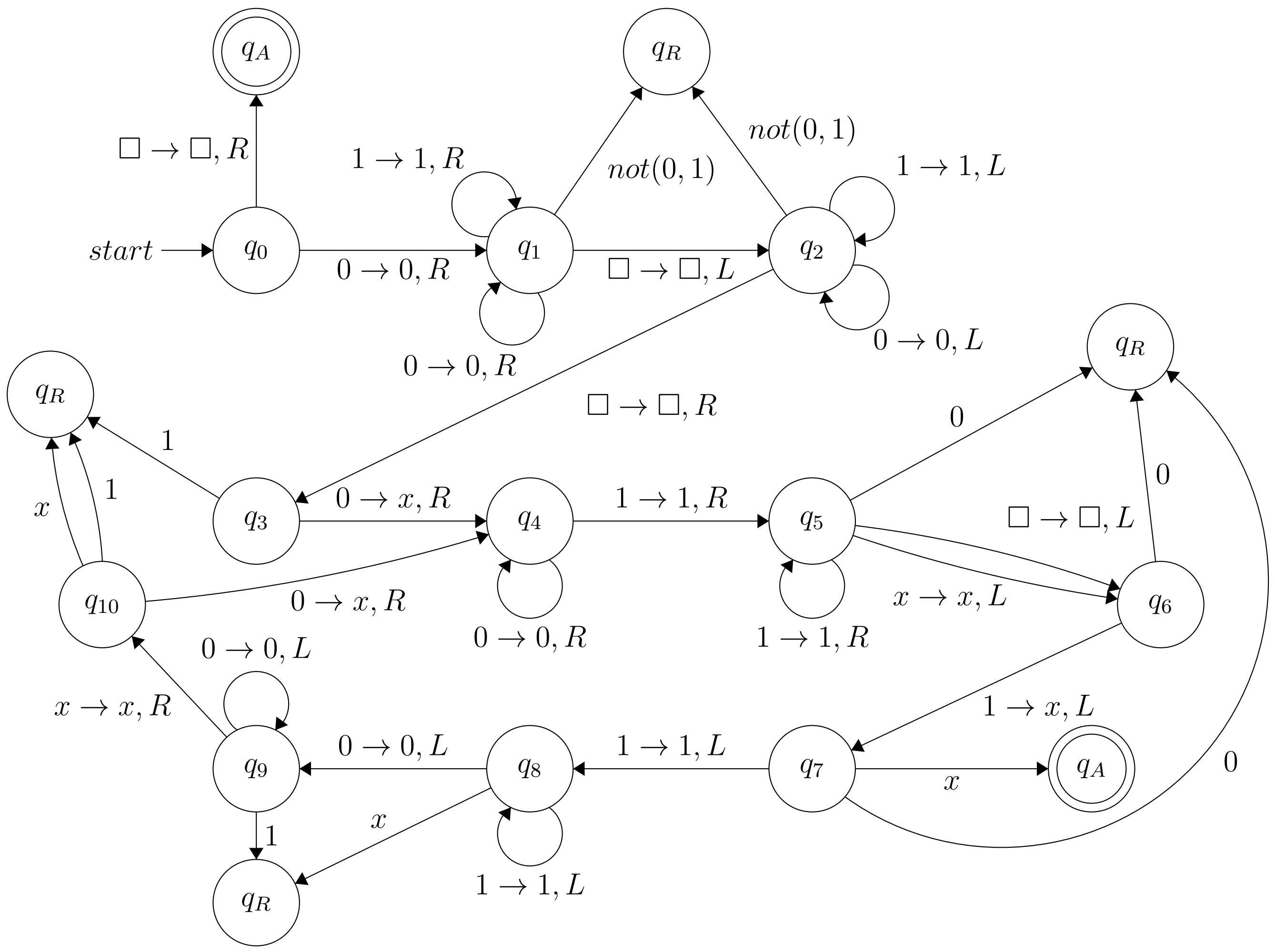


Figure 3: A transition function for a Turing machine that decides $\{0^n 1^n \mid n \in \mathbb{N}\}$.

We saw last time that Turing machines are more powerful than DFAs and NFAs, because they can decide non-regular languages. Recall our example Turing machine with the tape alphabet $\Gamma = \{0, 1, x\}$ and the above transition function (shown in Figure 3) to decide $\{0^n1^n \mid n \in \mathbb{N}\}$.

The diagram above shows the entire computation of the Turing machine. It sort of looks like a DFA, but with different transitions, because a Turing machine does more than just read characters. The diagram shows how the tape head writes and moves depending on what it reads and what state it's currently in.

We start off with the tape head at q_0 and read the very first character of the input. At each step of the computation, we take a different transition depending on the character we read.

Each transition is of the form $(A \to B, L/R)$, meaning at that state, if the head reads A, it writes B, and moves left or right. For transitions where there is only a single character listed (e.g. $x, 0, 1$ or $not(0, 1)$), this is just shorthand for reading $x, 0, 1$ or not 0 or 1 and signifies that the character we write and direction we move don't matter. Some trivial transitions are missing for clarity.

As soon as we reach an accept state (q_A) we accept, and as soon as we reach a reject state (q_R) we reject.

Exercise 1.2. Compare Figure 3 above to the algorithm we covered last lecture for deciding $\{0^n1^n \mid n \in \mathbb{N}\}$. Which states in the diagram correspond to which steps of the algorithm?

1.4 Practice Questions

1. Prove that there exists a Turing Machine which accepts all strings in the language $L = \{ww \mid w \in \{0, 1\}^*\}$ and rejects all strings not in L. Is there a DFA which recognizes L?

2. For every regular language R, is there a Turing machine which accepts strings if they are in R and rejects them if they are not?

2 Decidable vs. Recognizable

A language L is **decidable** if there exists a Turing machine M such that M accepts every string in L and M rejects every string not in L. Here, we say M *decides* L.

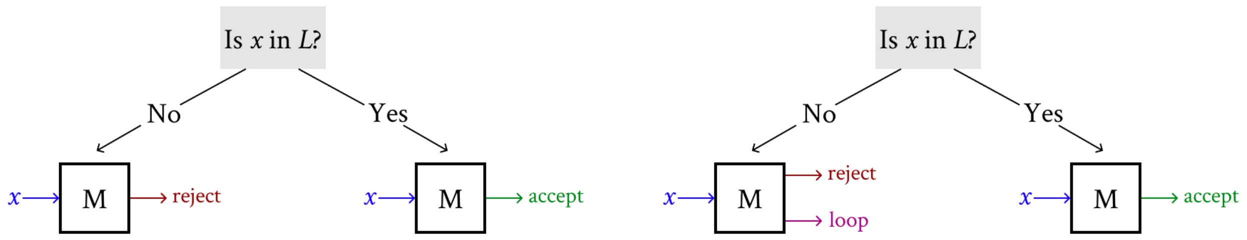


Figure 4: Decidable vs. Recognizable.

A language L is **recognizable** if there exists a Turing machine M such that M accepts every string in L and M rejects **or loops** on every string not in L. Here, we say M *recognizes* L.

We see that if and only if L and $\neg L$ ($\neg L$ is the language of all strings not in L) are both recognizable, then L is decidable.

We have already seen $0^n 1^n$ is decidable. Now, we will give examples of undecidable, recognizable, and unrecognizable languages.

Consider the following language

$$L = \{(M, w) \mid M \text{ is a Turing machine and } w \text{ is a string}\}$$

One first glance, this looks quite odd. Languages consist of strings. However, this language consists of (M, w) tuples, where M is a Turing machine. Think about it! Every Turing machine can be encoded as a string. We can always write down the finitely many transitions between states, and the alphabet and all the parameters, encoding it somehow.

Consider the following, somewhat similar language.

$$A_{TM} = \{(M, w) \mid M \text{ is a Turing machine and } w \text{ is a string and } M \text{ accepts } w\}$$

Theorem 1. *A_{TM} is recognizable.*

Proof. In order to recognize the language, we construct a Turing machine, which, given an input (M, w), simulates M on w. Such a Turing machine is called the Universal Turing machine, and is denoted U. U takes in (M, w), simulates M on w, and if M accepts w, then U accepts, and if M rejects w, then U rejects. Consequently, if M loops on w, then U loops on (M, w). So, A_{TM} is recognizable. □

Theorem 2. *A_{TM} is undecidable.*

Proof. Assume there is a Turing machine T which decides A_{TM}. In other words, T accepts (M, w) if $(M, w) \in A_{TM}$ and rejects (M, w) if $(M, w) \notin A_{TM}$.

$$T((M, w)) = \begin{cases} T \text{ accepts } (M, w) & \text{if } (M, w) \in A_{TM} \ \ (M \text{ accepts } w) \\ T \text{ rejects } (M, w) & \text{if } (M, w) \notin A_{TM} \ \ (M \text{ does not accept } w) \end{cases}$$

We will show this leads to a contradiction.

Now, remember, w is any string, and any Turing machine can be encoded as a string. What happens if we run T on (Turing machine, Turing machine) tuples, where the input strings w are Turing machines? We might get a table that looks something like this:

	M_1	M_2	M_3	$\dots$
M_1	accept	reject	accept	$\dots$
M_2	reject	accept	accept	$\dots$
M_3	accept	reject	reject	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

where the the cell in the ith row and jth column represents the output of $T(M_i, M_j)$, and $M_1, M_2, \ldots$ is some ordering of Turing machines. We don't actually know what the values are for the cells in the above table, but it turns out that won't matter.

Now, consider the following Turing machine D:

$$D(M) = \begin{cases} D \text{ accepts } M & \text{if } T \text{ rejects } (M, M) \\ D \text{ rejects } M & \text{if } T \text{ accepts } (M, M) \end{cases}$$

Once, more consider the table for T on tuples where both elements are Turing machines:

	M_1	M_2	M_3	$\ldots$	D
M_1	accept	reject	accept	$\ldots$	accept
M_2	reject	accept	accept	$\ldots$	reject
M_3	accept	reject	reject	$\ldots$	accept
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
D	reject	accept	accept	$\ldots$	reject

We see that the output of D simply reverses the diagonal, because $D(M_1)$ is the opposite of $T(M_1, M_1)$, and $D(M_2)$ is the opposite of $T(M_2, M_2)$, and so forth. The bold values (flipped diagonal) in the table below are the outputs of D.

	M_1	M_2	M_3	$\ldots$	D
M_1	**reject**	reject	accept	$\ldots$	accept
M_2	reject	**reject**	accept	$\ldots$	reject
M_3	accept	reject	**accept**	$\ldots$	accept
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
D	reject	accept	accept	$\ldots$	**???**

However, what happens at $D(D)$? By construction, D rejects D if T accepts (D, D), but T accepts (D, D) when D accepts D. Also by construction, D accepts D if T rejects (D, D), but this only happens when D does not accept D.

Therefore, we have

$$D(D) = \begin{cases} D \text{ accepts } D & \text{if } D \text{ does not accept } D \\ D \text{ rejects } D & \text{if } D \text{ accepts } D \end{cases}$$

But, this is a contradiction! So, D cannot exist, so T cannot exist, so there can be no Turing machine T that decides A_{TM}. Thus, A_{TM} is undecidable. □

2.1 Practice Questions

1. Prove $\neg A_{TM}$ is unrecognizable.

2. Prove the Halting problem $HALT = \{(M, w) \mid M \text{ is a Turing machine and } M \text{ halts on } w\}$ is recognizable and undecidable. What can you conclude about $\neg HALT$?

3 Reductions

Now, you could prove question 2 above by deriving a contradiction like we showed for A_{TM}. But, that's a lot of work. Diagonalization quickly becomes complicated when we start working with complex problems. Instead, let's use what we already know about A_{TM} to prove $HALT$ is undecidable.

Theorem 3. *HALT is undecidable.*

Proof. We will prove that if $HALT$ is decidable, then A_{TM} is decidable. This will lead to a contradiction—we already know A_{TM} is undecidable!

Suppose we have a magic function f that takes as input (M, w), and outputs (N, x), where M and N are Turing machines, and w and x are strings.

The function is magical because it is defined as follows:

$$f(M,w) = \begin{cases} (N,x) \text{ where N is a TM that halts on } x & \text{if M accepts } w \\ (N,x) \text{ where N is a TM that loops on } x & \text{if M does not accept } w \end{cases}$$

Suppose this function exists. Furthermore, suppose this function is computable, that is, it can be run on a Turing machine.

Now, assume $HALT$ is decidable. Then, there exists a Turing machine A that decides $HALT$. In other words, A accepts every input (N, x) such that N halts on x, and A rejects all (N, x) where N loops on x.

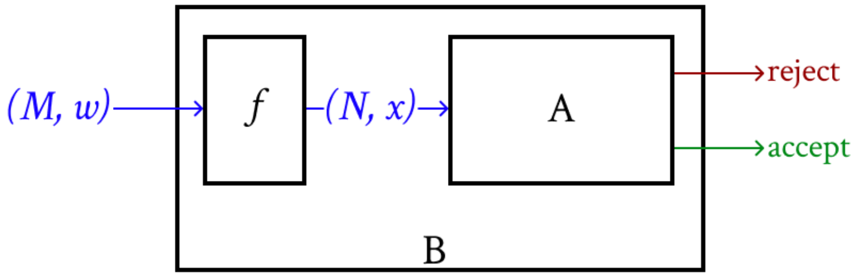


Figure 5: A Turing machine B that decides A_{TM} based on a TM A that decides $HALT$.

Using f and A, we construct a Turing machine B that decides A_{TM}, as shown in Figure 5 above.

On some input (M, w), B runs the magic function f from above on (M, w). This produces some output (N, x). B then simulates running (N, x) on A to see if (N, x) halts or not, because A decides $HALT$. B then outputs the result of A.

Why does B decide A_{TM}? Well, if our input to B (M, w) is in A_{TM}, then M accepts w, so $f(M, w) = (N, x) \in HALT$, so A accepts and thus B accepts. Otherwise, if $(M, w) \notin A_{TM}$, M does not accept w, so $f(M, w) = (N, x) \notin HALT$, so A rejects and B rejects.

And there we have it. Assuming the magic function f exists, B decides A_{TM}. However, A_{TM} is undecidable, so B cannot exist. Therefore, no Turing machine A that decides $HALT$ can exist. Thus, $HALT$ is undecidable.

Now, all that's left to do is to prove the existence of f. Consider the following function.

$$f(M, w) = (N, (M, w))$$

where N is a Turing machine that runs M on w and then accepts.

This is a computable function; it can be run on a Turing machine. Furthermore, if $(M, w) \in HALT$, then M halts on w, so N will accept, and thus $(N, (M, w)) \in A_{TM}$.

If $(M, w) \notin HALT$, then M loops on w, so N will loop simulating M on w, so $(N, (M, w)) \notin A_{TM}$. Thus, f satisfies all the properties of our "magic function" from earlier.

$$f(M, w) = \begin{cases} (N, (M, w)) \text{ where N is a TM that halts on } (M, w) & \text{if M accepts } w \\ (N, (M, w)) \text{ where N is a TM that loops on } (M, w) & \text{if M does not accept } w \end{cases}$$

Thus, $HALT$ is undecidable. $\square$

This method of solving problems—showing that if language L is decidable, then A_{TM} is decidable, is called a reduction. We reduced A_{TM} to the language $HALT$.

Could we do this with any two languages A and B? Given two languages A and B, when can I use B's decidability to prove A's decidability?

It turns out that a language A is mapping reducible to B precisely when a "magic" function exists between the A and B. Let's define this formally.

Definition 3.1. A is mapping reducible to B ($A \leq_m B$) if there exists a computable function f such that for all $x \in A$, $f(x) \in B$, and for all $x \in \neg A$, $f(x) \in \neg B$.

In other words, there exists a function that "maps" elements of A to elements of B. Computable essentially means that f can be calculated on a Turing machine.

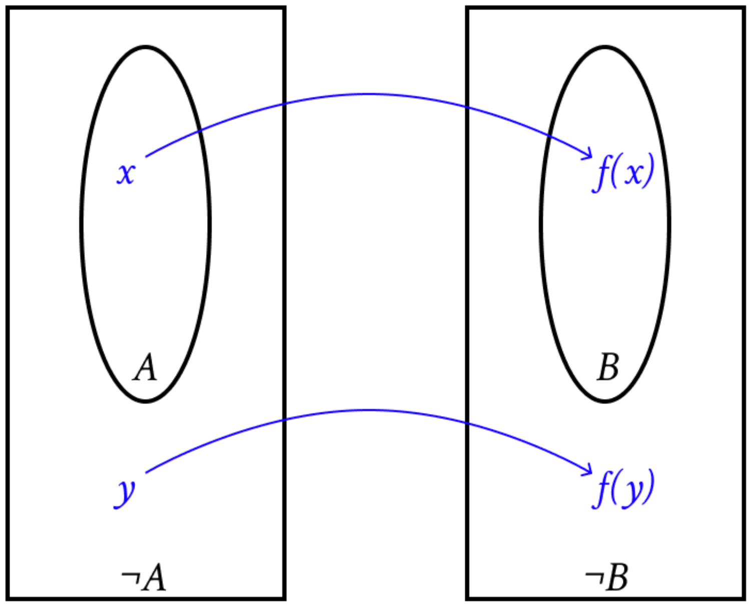
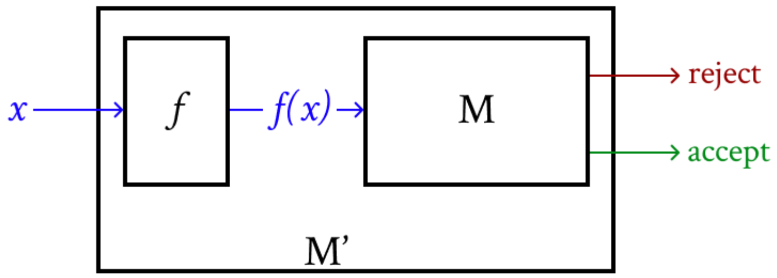


Figure 6: A mapping from A to B. Note f is not necessarily a bijection. There could be points in B or $\neg B$ that no x maps to. For example, suppose $f : \mathbb{R}^+ \to \mathbb{R}^+$, $A = (0, 1)$, and $B = (2, 4)$. Then, let $f(x) = \frac{1}{x}$ if $x \geq 1$ and $f(x) = x + 2$ otherwise. This is a valid computable mapping from $A \to B$ but is no $x \in \mathbb{R}^+$ such that $f(x) \in [3, \infty)$.

We can use this definition of a reduction to prove a few very useful theorems relating two languages. These should look very similar to how we proved the undecidability of $HALT$ given A_{TM}'s undecidability.

Theorem 4. *If $A \leq_m B$ and B is decidable, then A is decidable.*



Proof. If $A \leq_m B$, then there is a computable function f such that for all $x \in A$, $f(x) \in B$, and for all $x \in \neg A$, $f(x) \in \neg B$.

Suppose B is decidable by some Turing machine M. We construct the following Turing machine M' (represented in the figure above) to decide A. We simply run any input x through the function f, check if $f(x) \in B$ using its decider M, and output the result for M'.

If $x \in A$, then $f(x) \in B$, so M accepts $f(x)$, and M' accepts x.

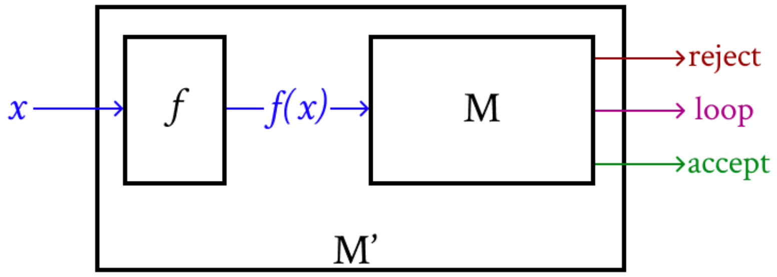
If $x \notin A$, then $f(x) \notin B$, so M rejects $f(x)$, and thus M' rejects x.

Thus, M' decides A. □

Corollary 1. *If $A \leq_m B$ and A is undecidable, then B is undecidable.*

Proof. This is just the contrapositive of Theorem 4. □

Theorem 5. *If $A \leq_m B$ and B is recognizable, then A is recognizable.*



Proof. If $A \leq_m B$, then there is a computable function f such that for all $x \in A$, $f(x) \in B$, and for all $x \in \neg A$, $f(x) \in \neg B$.

Suppose B is recognizable by some Turing machine M. We construct a the following Turing machine M' (represented by the diagram above) to recognize A. We simply run the input x through the function f, and output the result of $M(f(x))$ as our result of $M'(x)$.

If $x \in A$, then $f(x) \in B$, so M accepts $f(x)$, and thus M' accepts x.

If $x \notin A$, then $f(x) \notin B$, so M either rejects or loops on $f(x)$. So, M' rejects or loops on x.

Thus, M' recognizes A. □

Corollary 2. *If $A \leq_m B$ and A is unrecognizable, then B is unrecognizable.*

Proof. This is just the contrapositive of Theorem 5. $\square$

It's important to remember that not every function is computable! For example, consider the function $f(M, w) = (N, x)$, where N is a Turing machine that accepts x if M loops on w. How can we compute this? We can never know if M loops on w, because at any time M might halt on w in one more step, or two more steps. I would have to run M on w infinitely, so this isn't computable.

3.1 Examples

Let's go through a few examples. For each example, I'll present a language, and we'll construct a computable function to show that the language is reducible to a language we've already proven results about. The hard part is almost always coming up with a proper function that satisfies everything we want.

Example 3.1. Consider the language $EMPTY_{TM} = \{M \mid M \text{ is a TM and } M \text{ accepts } \emptyset\}$. $EMPTY_{TM}$ is unrecognizable.

We can prove this by reducing $\neg A_{TM} = \{(M, w) \mid M \text{ is a TM that does not accept } w\}$ to $EMPTY_{TM}$.

Consider the function $f(M, w) = N$ where N is a TM that rejects all strings x if $x \neq w$ and $N(w) = M(w)$.

This function is computable, we can easily create a Turing machine that takes in (M, w) and outputs N.

If M accepts w, then N accepts only w. So, $L(N) \neq \emptyset$, and thus $N \notin EMPTY_{TM}$. If M does not accept w, then N accepts nothing. So, $L(N) = \emptyset$, and $N \in EMPTY_{TM}$. Thus, $(M, w) \in \neg A_{TM} \iff N \in EMPTY_{TM}$.

Since $\neg A_{TM} \leq_m EMPTY_{TM}$, and $\neg A_{TM}$ is unrecognizable, $EMPTY_{TM}$ is unrecognizable.

Example 3.2. Consider the language $REGULAR_{TM} = \{M \mid M \text{ is a TM and } L(M) \text{ is regular}\}$.

We will prove that $REGULAR_{TM}$ is unrecognizable.

Consider the following computable function:

$$f(M, w) = N$$

where N is Turing machine that rejects every input except 0^n1^n. $N(x) = M(w)$ for all $x \in \{0^n1^n \mid n \in \mathbb{N}\}$.

We see that if M does not accept w, then N accepts nothing. Since the empty language is regular, $N \in REGULAR_{TM}$. If M accepts w, then $L(N) = \{0^n1^n \mid n \in \mathbb{N}\}$, so $N \notin REGULAR_{TM}$. Thus, $(M, w) \in \neg A_{TM} \iff N \in REGULAR_{TM}$. Since $\neg A_{TM}$ is unrecognizable, $REGULAR_{TM}$ is unrecognizable.

Example 3.3. Assuming only $HALT$ is undecidable, we can prove A_{TM} is undecidable. We can construct the following function:

$$f(M, w) = (N, x)$$

where N is a Turing machine that runs M on w and then accepts. This is a computable function. Furthermore, if $(M, w) \in HALT$, then M halts on w, so N will accept x, so $(N, x) \in A_{TM}$. If $(M, w) \notin HALT$, then M loops on w, so N will loop on x simulating M on w, so $(N, x) \notin A_{TM}$. Thus, $HALT \leq_m A_{TM}$.

Since $HALT$ and A_{TM} can both be reduced to each other, $HALT \equiv_m A_{TM}$! The problems are equivalent!

3.2 Practice Questions

Questions 1–4 have been borrowed from the Autumn 2019 offering of CS 154 at Stanford. You can view the lectures online at the course site for more problems.

1. Prove

   $$\textsc{OneHalts} := \{(M, x, y) \mid \text{the TM } M \text{ halts on precisely one of the two inputs } x, y\}$$

   is unrecognizable.

2. Is the following language decidable?

   $$\textsc{Symbol} = \{(M, w, \sigma) \mid M \text{ encodes a TM that writes the symbol } \sigma \in \Gamma \text{ in some step, while running on the input } w\}$$

3. Prove $\neg$OneHalts is also unrecognizable.

4. Let $S = \{M \mid M$ is a Turing machine, and the language recognized by M is $\{M\}\}$. Prove that S and $\neg S$ are unrecognizable. Recall that Turing machines can access their own descriptions.

5. Why is the following answer to the first part of question 4 incorrect?

   > We shall show $\neg A_{TM} \leq_m S$. Consider some $(M, w) \in \neg A_{TM}$. We construct a function $f(M, w) = M'$, where M' is a Turing machine that rejects every input x except $x = M'$. If $x = M'$, M' accepts x if M does not accept w, and M' rejects x if M accepts w.
   >
   > Thus, if $(M, w) \in \neg A_{TM}$, we see that M does not accept w so the language of M' is M' and so $M' \in S$. Going the other way, if the representation of $M' \in S$, then M must not accept w so $(M, w) \in \neg A_{TM}$. Therefore, S is not recognizable as $\neg A_{TM}$ is not recognizable. Note that M' can check if $x = M'$ as it has access to its own description.
   >
   > — *Courtesy of a friend from CS 154.*

4 Resources

- Some definitions and explanations from CS 103, Stanford's introductory theory course.
  `http://web.stanford.edu/class/archive/cs/cs103/cs103.1202/`
- We've borrowed multiple practice questions and examples (Section 3) from Stanford's CS 154 class. It's a great resource for learning more about Turing machines and theory.
  `https://cs154.stanford.edu`
- Automata drawing software:
  `http://madebyevan.com/fsm`
- Turing Machine diagram:
  `http://www.texample.net/tikz/examples/turing-machine-2/`