# Theory of Computation Notes: The Recursion Theorem

Arjun Chandrasekhar

To motivate this section, ask yourself the following questions:

- Can we build a robot that builds another robot?
- Can we build a robot that builds another robot-building robot?
- Can we build a robot that builds an identical robot-building robot?

The answer to these questions is: yes! We will see that we can build a program that replicates itself. In fact, we can even build a program that analyzes itself. Furthermore, we will show that this is doable even with the Turing machine model of computation, meaning we don't need a gimmick such as reading your own source code file. We will simply create a program that does normal TM operations, and ends up with its own description on the tape.

## 1 A Self-reproducing TM

**Example 1.1.** Consider the following 'program':

Print the following sentence twice, the second time in quotes
"Print the following sentence twice, the second time in quotes"

What does it do? Well, ideally, it will print Print the following sentence twice, the second time in quotes, and then it will print it again with quotes, which yields "Print the following sentence twice, the second time in quotes". But this is exactly the same as the original program!

The trick is, we have to make sure the program knows that Print the following sentence twice, the second time in quotes, i.e. the version without the quotes, is what 'the following sentence' refers to. As long as we can do that, the program will clearly reproduce itself.

While this is a nice example, for this to be useful we need to give a precise description of a Turing machine that can reproduce itself.

**Theorem 1.1.** There exists a TM that prints out its own description.

*Proof.* First, we will use the notation $P_w$ to refer to a machine $P$ that erases its input, writes out the string $w$, and halts. For example, $P_{Turing}$ will ignore its inut and print the string 'Turing' onto the tape. $P_{1001}$ is almost identical, but instead of printing 'Turing' it will print the string '1001'. It does this no matter what input it gets.

Before going forward, convince yourself that for any string, you know how to write the program $P_w$.

Next, recall that $\langle P_w \rangle$ is the *description* (i.e. source code) of the machine $P_w$. Keep in mind that $\langle P_w \rangle$ is itself a string. This means we could make a machine $P_{\langle P_w \rangle}$. This is a machine that ignores its input and prints out $\langle P_w \rangle$, i.e. it prints out the source code of *another* machine that ignores its input and prints out a string.

Now, convince yourself that you can write a program that takes as input $w$ and creates $\langle P_w \rangle$. You know exactly how the source code for this program should look; you just need to substitute the string $w$ that you receive at runtime into the appropriate place in the source code.

Once you have convinced yourself of this, we will define a machine $Q$, which does the following:

1. $Q$ takes a string $w$ as input
2. Construct the machine $P_w$
3. Write the description $\langle P_w \rangle$ onto the tape and exit

Next, we will define a machine $B$ that does the following:

1. $B$ takes a TM description $\langle M \rangle$ as input
2. Run $Q$ on $\langle M \rangle$. This produces $\langle P_{\langle M \rangle} \rangle$
3. Construct a machine $M^*$ which is a combination of $P_{\langle M \rangle}$ and $M$. That is, if we were to run $M^*$, it would first run $P_{\langle M \rangle}$ and then $M$. But note that we do not run $M^*$ - we simply construct the machine.
4. Write $\langle M^* \rangle$ onto the tape and exit.

Now we are ready to define our machine SELF, which reproduces its own source code. SELF consists of two parts: $P_{\langle B \rangle}$ and $B$. It runs $P_{\langle B \rangle}$, and then runs $B$. That's it! This is the TM that will reproduce its own source code! To see why, let's go step by step through how SELF operates:

1. SELF takes a string $w$ as input
2. Run $P_{\langle B \rangle}$. This will erase $w$, and print out $\langle B \rangle$ onto the tape.
3. Pass control to $B$
4. B reads $\langle B \rangle$ on the tape
5. Construct $P_{\langle B \rangle}$
6. Combine $P_{\langle B \rangle}$ and $B$ into one machine $M^*$
7. Write $\langle M^* \rangle$ to the tape and exit

So when SELF runs, it produces a description $\langle M^* \rangle$. Furthermore, $M^*$ is a combination of two machines: $P_{\langle B \rangle}$ and $B$. But wait - SELF is also a combination of $P_{\langle B \rangle}$ and $B$! This means SELF has reproduced its own source code. □

**Remark.** At first glance, this proof may appear circular. Because we run $B$ on $\langle B \rangle$, it looks like we assumed that $B$ could obtain its own description, which is the very thing we were trying to prove. But this is not the case. $B$ has access to its own description $\langle B \rangle$ because before running $B$, we ran $P_{\langle B \rangle}$, which printed $\langle B \rangle$ onto the tape so that $B$ could access it.

You may also wonder how we construct $P_{\langle B \rangle}$ - how do we know what $B$ is going to be before we run $B$? But remember, we defined $B$ before we defined SELF, so it is straightforward to construct $P_{\langle B \rangle}$ and combine it with $B$ into one machine SELF.

## 2 The Recursion Theorem

We have shown how to make a TM that always reproduces its own description. This is a quirky and interesting result, but how do we make it useful? One way would be if we could write a program that actually analyzes its own source code and rather than just printing it out. But can we actually do this? Is it possible to write a program that obtains its own description and then does something interesting with that description?

The answer is once again, yes.

**Theorem 2.1** (Kleene's recursion theorem)**.** Let $T$ be a Turing machine that computes a function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$. Then there exists a TM $R$ that computes a function $r : \Sigma^* \to \Sigma^*$, such that for all $w$:

$$r(w) = t(\langle R \rangle, w)$$

Here, $T$ is a TM that receives a machine description as one of its inputs, and analyzes that machine. The recursion theorem produces a machine $R$ that behaves exactly as $T$ behaves, but uses its own description in place of the input.

*Proof.* We construct $R$ using three components: $P_{\langle BT \rangle}, B, T$. Here, $P_{\langle BT \rangle}$ and $B$ are nearly identical, with a couple of minor tweaks.

The machine $R$ will behave as follows:

1. $R$ takes a string $w$ as input
2. Run $P_{\langle BT \rangle}$. This prints out $\langle BT \rangle$ onto the tape, where $\langle BT \rangle$ is a description of a machine that combines $B$ and then $T$. One important detail is that we don't erase $w$; instead we keep $w$ on the tape and write $\langle BT \rangle$ after $w$
3. Pass control to $B$
4. $B$ reads $\langle BT \rangle$ on the tape and constructs $P_{\langle BT \rangle}$. It then combines $P_{\langle BT \rangle}, B, T$ into one machine $M^*$
5. Write $\langle M^* \rangle$ onto the tape. Again, this is all done on the spaces following the original input $w$
6. Pass control to $T$
7. $T$ reads $w$ and $\langle M^* \rangle$. It computes $t(\langle M^* \rangle, w)$

Let's analyze what happens when we run $R$ on input $w$. First, we run $P_{\langle BT \rangle}$, which puts $\langle BT \rangle$ onto the tape (without erasing $w$). Next, $B$ finds $\langle BT \rangle$ on the tape, and constructs $P_{\langle BT \rangle}$. It then combines $P_{\langle BT \rangle}, B, T$ into one machine $M^*$. But wait - these are the same three components that comprise $R$! Thus, $R$ has managed to reproduce its own description, and when we write $\langle M^* \rangle$ to the tape we are actually writing $R$'s own description $\langle R \rangle$ to the tape. Finally, $T$ finds $w$ and $\langle R \rangle$ on the tape, and uses this to compute $t(\langle R \rangle, w)$, as desired. □

**Corollary 2.1.** When we construct a TM, we may have the TM obtain its own description. That is, we can put "obtain its own description" as part of the pseudocode, and then we can go on to analyze the description that we obtain.

# 3 Undecidability Proofs via the Recursion Theorem

Previously we have seen two techniques to prove that certain languages are undecidable: diagonalization and reduction. The recursion theorem opens up a third technique which often produces very short and sweet proofs.

**Theorem 3.1.** HALT $= \{\langle M, w \rangle | M \text{ halts on } w\}$ is undecidable.

*Proof.* AFSOC some machine $H$ decides HALT. We will construct a machine $M$ that does the following:

1. $M$ takes a string $w$ as input
2. Obtain its own description $\langle M \rangle$
3. Run $H$ on $\langle M, w \rangle$
4. If $H$ accepts $\langle M, w \rangle$, go into an infinite loop
   If $H$ rejects $\langle M, w \rangle$, immediately halt

If $H$ says $M$ should halt on $w$, $M$ goes into a loop. If $H$ says $M$ should loop on $w$, $M$ immediately halts. Thus we have arrived at a contradiction, and we conclude that $H$ is not a valid decider for HALT. □

**Remark.** This is actually quite similar to the diagonalization proof, in that we construct a machine $M$ that runs the decider $H$ and does the opposite, which leads to a contradiction when $M$ receives its own description as the input. The difference here is that $M$ doesn't need to be fed its own description - it knows how to obtain that. Put another way, $M$ knows how to find itself in the diagonalization grid and do the opposite of what it is supposed to do.

**Theorem 3.2.** $\mathrm{A_{TM}} = \{\langle M, w\rangle | w \in L(M)\}$ is undecidable

*Proof.* AFSOC some machine $A$ decides $\mathrm{A_{TM}}$. We will construct a machine $M$ that does the following:

1. $M$ takes a string $w$ as input
2. Obtain its own description $\langle M\rangle$
3. Run $A$ on $\langle M, w\rangle$
4. If $A$ accepts $\langle M, w\rangle$, reject $w$
   If $A$ rejects $\langle M, w\rangle$, accept $w$

If $A$ says $M$ should accept $w$, it rejects $w$. If $A$ says $M$ should reject $w$, it accepts $w$. Thus we have arrived at a contradiction, and we conclude that $A$ is not a valid decider for $\mathrm{A_{TM}}$. □

**Definition 3.1.** Consider the following language

$$\mathrm{REG_{TM}} = \{\langle M\rangle | L(M) \text{ is regular}\}$$

We receive a TM description as input, and we want to figure out if $M$ recognizes a regular language or not. If we could decide this language, we could determine whether certain TMs can be converted to an equivalent DFA.

**Theorem 3.3.** $\mathrm{REG_{TM}}$ is undecidable.

*Proof.* AFSOC some TM $R$ decides $\mathrm{REG_{TM}}$. We will construct a machine $M$ that does the following:

1. $M$ takes a string $w$ as input
2. Obtain its own description $\langle M\rangle$
3. Run $R$ on $\langle M\rangle$
4. If $R$ accepts $\langle M\rangle$, simulate a machine that recognizes $0^n1^n$ (a non-regular language)
5. If $R$ rejects $\langle M\rangle$, simulate a machine that recognizes $0^*1^*$ (a regular language)

If $R$ says that $L(M)$ is regular, it simulates a machine for $0^n1^n$, which is not regular. If $R$ says that $L(M)$ is not regular, it simulates a machine for $0^*1^*$, which is regular. So $L(M)$ is never what $R$ says it should be. Thus, we have arrived at a contradiction, and we conclude that $R$ is not deciding $\mathrm{REG_{TM}}$. □

**Definition 3.2.** We say a TM is **minimal** if there is no other machine $M_2$ such that

1. $L(M) = L(M_2)$
2. $|\langle M_2\rangle| < |\langle M\rangle|$

In other words, a machine $M$ is minimal if there is not another machine with a shorter description that recognizes the same language.

**Definition 3.3.** Consider the following language:

$$\text{MIN}_{\text{TM}} = \{\langle M \rangle | M \text{ is minimal}\}$$

We receive a TM description as input, and we want to determine if there is another machine $M_2$ that has a shorter description but still recognizes the same language as $M$. If we could decide this language, we could determine whether an arbitrary program could be re-written with fewer characters without changing the behavior of the program.

**Theorem 3.4.** $\text{MIN}_{\text{TM}}$ is not recursively enumerable.

*Proof.* AFSOC there is some enumerator $E$ that enumerates $\text{MIN}_{\text{TM}}$. We will construct a machine $M$ that does the following:

1. $M$ takes a string $w$ as input
2. Obtain its own description $\langle M \rangle$
3. Run $E$ until it prints out a machine $M_2$ such that $|\langle M \rangle| < |\langle M_2 \rangle|$
4. Simulate $M_2$ on $w$

$M$ obtains its own description, and then runs $E$ until it finds a machine $M_2$ with an even longer description. It then simulates $M_2$. So $L(M_1) = L(M_2)$ and $|\langle M \rangle| < |\langle M_2 \rangle|$. But this is a contradiction of the fact that $M_2$ is supposed to be a minimal TM! We conclude that $E$ is not a valid enumerator for $\text{MIN}_{\text{TM}}$. □