6

ADVANCED TOPICS IN COMPUTABILITY THEORY

In this chapter we delve into four deeper aspects of computability theory: (1) the recursion theorem, (2) logical theories, (3) Turing reducibility, and (4) descriptive complexity. The topic covered in each section is mainly independent of the others, except for an application of the recursion theorem at the end of the section on logical theories. Part Three of this book doesn't depend on any material from this chapter.

6.1 THE RECURSION THEOREM

The recursion theorem is a mathematical result that plays an important role in advanced work in the theory of computability. It has connections to mathematical logic, the theory of self-reproducing systems, and even computer viruses.

To introduce the recursion theorem, we consider a paradox that arises in the study of life. It concerns the possibility of making machines that can construct replicas of themselves. The paradox can be summarized in the following manner.

1. Living things are machines.
2. Living things can self-reproduce.
3. Machines cannot self-reproduce.

Statement 1 is a tenet of modern biology. We believe that organisms operate in a mechanistic way. Statement 2 is obvious. The ability to self-reproduce is an essential characteristic of every biological species. For statement 3, we make the following argument that machines cannot self-reproduce. Consider a machine that constructs other machines, such as an automated factory that produces cars. Raw materials go in at one end, the manufacturing robots follow a set of instructions, and then completed vehicles come out the other end.

We claim that the factory must be more complex than the cars produced, in the sense that designing the factory would be more difficult than designing a car. This claim must be true because the factory itself has the car's design within it, in addition to the design of all the manufacturing robots. The same reasoning applies to any machine A that constructs a machine B: A must be *more* complex than B. But a machine cannot be more complex than itself. Consequently, no machine can construct itself, and thus self-reproduction is impossible.

How can we resolve this paradox? The answer is simple: Statement 3 is incorrect. Making machines that reproduce themselves *is* possible. The recursion theorem demonstrates how.

SELF-REFERENCE

Let's begin by making a Turing machine that ignores its input and prints out a copy of its own description. We call this machine *SELF*. To help describe *SELF*, we need the following lemma.

LEMMA 6.1

There is a computable function $q\colon \Sigma^* \longrightarrow \Sigma^*$, where if w is any string, $q(w)$ is the description of a Turing machine P_w that prints out w and then halts.

PROOF Once we understand the statement of this lemma, the proof is easy. Obviously, we can take any string w and construct from it a Turing machine that has w built into a table so that the machine can simply output w when started. The following TM Q computes $q(w)$.

```
Q = "On input string $w$:
    1. Construct the following Turing machine $P_w$.
         $P_w$ = "On any input:
             1. Erase input.
             2. Write $w$ on the tape.
             3. Halt."
    2. Output $\langle P_w \rangle$."
```

The Turing machine *SELF* is in two parts: A and B. We think of A and B as being two separate procedures that go together to make up *SELF*. We want *SELF* to print out $\langle SELF \rangle = \langle AB \rangle$.

Part A runs first and upon completion passes control to B. The job of A is to print out a description of B, and conversely the job of B is to print out a description of A. The result is the desired description of *SELF*. The jobs are similar, but they are carried out differently. We show how to get part A first.

For A we use the machine $P_{\langle B \rangle}$, described by $q(\langle B \rangle)$, which is the result of applying the function q to $\langle B \rangle$. Thus, part A is a Turing machine that prints out $\langle B \rangle$. Our description of A depends on having a description of B. So we can't complete the description of A until we construct B.

Now for part B. We might be tempted to define B with $q(\langle A \rangle)$, but that doesn't make sense! Doing so would define B in terms of A, which in turn is defined in terms of B. That would be a *circular* definition of an object in terms of itself, a logical transgression. Instead, we define B so that it prints A by using a different strategy: *B computes A* from the output that A produces.

We defined $\langle A \rangle$ to be $q(\langle B \rangle)$. Now comes the tricky part: If B can obtain $\langle B \rangle$, it can apply q to that and obtain $\langle A \rangle$. But how does B obtain $\langle B \rangle$? It was left on the tape when A finished! So B only needs to look at the tape to obtain $\langle B \rangle$. Then after B computes $q(\langle B \rangle) = \langle A \rangle$, it combines A and B into a single machine and writes its description $\langle AB \rangle = \langle SELF \rangle$ on the tape. In summary, we have:

$A = P_{\langle B \rangle}$, and

$B =$ "On input $\langle M \rangle$, where M is a portion of a TM:

1. Compute $q(\langle M \rangle)$.
2. Combine the result with $\langle M \rangle$ to make a complete TM.
3. Print the description of this TM and halt."

This completes the construction of *SELF*, for which a schematic diagram is presented in the following figure.

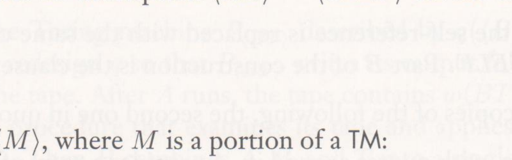

FIGURE 6.2
Schematic of *SELF*, a TM that prints its own description

If we now run $SELF$, we observe the following behavior.

1. First A runs. It prints $\langle B \rangle$ on the tape.
2. B starts. It looks at the tape and finds its input, $\langle B \rangle$.
3. B calculates $q(\langle B \rangle) = \langle A \rangle$ and combines that with $\langle B \rangle$ into a TM description, $\langle SELF \rangle$.
4. B prints this description and halts.

We can easily implement this construction in any programming language to obtain a program that outputs a copy of itself. We can even do so in plain English. Suppose that we want to give an English sentence that commands the reader to print a copy of the same sentence. One way to do so is to say:

Print out this sentence.

This sentence has the desired meaning because it directs the reader to print a copy of the sentence itself. However, it doesn't have an obvious translation into a programming language because the self-referential word "this" in the sentence usually has no counterpart. But no self-reference is needed to make such a sentence. Consider the following alternative.

Print out two copies of the following, the second one in quotes:
"Print out two copies of the following, the second one in quotes:"

In this sentence, the self-reference is replaced with the same construction used to make the TM $SELF$. Part B of the construction is the clause:

Print out two copies of the following, the second one in quotes:

Part A is the same, with quotes around it. A provides a copy of B to B so B can process that copy as the TM does.

The recursion theorem provides the ability to implement the self-referential *this* into any programming language. With it, any program has the ability to refer to its own description, which has certain applications, as you will see. Before getting to that, we state the recursion theorem itself. The recursion theorem extends the technique we used in constructing $SELF$ so that a program can obtain its own description and then go on to compute with it, instead of merely printing it out.

THEOREM **6.3**

Recursion theorem Let T be a Turing machine that computes a function $t\colon \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. There is a Turing machine R that computes a function $r\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

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

The statement of this theorem seems a bit technical, but it actually represents something quite simple. To make a Turing machine that can obtain its own description and then compute with it, we need only make a machine, called T

in the statement, that receives the description of the machine as an extra input. Then the recursion theorem produces a new machine R, which operates exactly as T does but with R's description filled in automatically.

PROOF The proof is similar to the construction of $SELF$. We construct a TM R in three parts, A, B, and T, where T is given by the statement of the theorem; a schematic diagram is presented in the following figure.

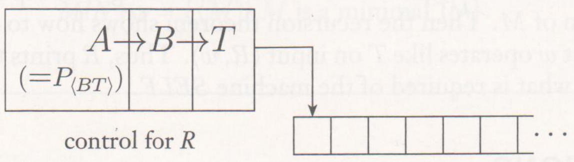

FIGURE 6.4
Schematic of R

Here, A is the Turing machine $P_{\langle BT \rangle}$ described by $q(\langle BT \rangle)$. To preserve the input w, we redesign q so that $P_{\langle BT \rangle}$ writes its output following any string preexisting on the tape. After A runs, the tape contains $w\langle BT \rangle$.

Again, B is a procedure that examines its tape and applies q to its contents. The result is $\langle A \rangle$. Then B combines A, B, and T into a single machine and obtains its description $\langle ABT \rangle = \langle R \rangle$. Finally, it encodes that description together with w, places the resulting string $\langle R, w \rangle$ on the tape, and passes control to T.

TERMINOLOGY FOR THE RECURSION THEOREM

The recursion theorem states that Turing machines can obtain their own description and then go on to compute with it. At first glance, this capability may seem to be useful only for frivolous tasks such as making a machine that prints a copy of itself. But, as we demonstrate, the recursion theorem is a handy tool for solving certain problems concerning the theory of algorithms.

You can use the recursion theorem in the following way when designing Turing machine algorithms. If you are designing a machine M, you can include the phrase "obtain own description $\langle M \rangle$" in the informal description of M's algorithm. Upon having obtained its own description, M can then go on to use it as it would use any other computed value. For example, M might simply print out $\langle M \rangle$ as happens in the TM $SELF$, or it might count the number of states in $\langle M \rangle$, or possibly even simulate $\langle M \rangle$. To illustrate this method, we use the recursion theorem to describe the machine $SELF$.

```
SELF = "On any input:
    1. Obtain, via the recursion theorem, own description ⟨SELF⟩.
    2. Print ⟨SELF⟩."
```

The recursion theorem shows how to implement the "obtain own description" construct. To produce the machine $SELF$, we first write the following machine T.

```
T = "On input ⟨M, w⟩:
    1. Print ⟨M⟩ and halt."
```

The TM T receives a description of a TM M and a string w as input, and it prints the description of M. Then the recursion theorem shows how to obtain a TM R, which on input w operates like T on input $\langle R, w\rangle$. Thus, R prints the description of R—exactly what is required of the machine $SELF$.

APPLICATIONS

A ***computer virus*** is a computer program that is designed to spread itself among computers. Aptly named, it has much in common with a biological virus. Computer viruses are inactive when standing alone as a piece of code. But when placed appropriately in a host computer, thereby "infecting" it, they can become activated and transmit copies of themselves to other accessible machines. Various media can transmit viruses, including the Internet and transferable disks. In order to carry out its primary task of self-replication, a virus may contain the construction described in the proof of the recursion theorem.

Let's now consider three theorems whose proofs use the recursion theorem. An additional application appears in the proof of Theorem 6.17 in Section 6.2.

First we return to the proof of the undecidability of A_{TM}. Recall that we earlier proved it in Theorem 4.11, using Cantor's diagonal method. The recursion theorem gives us a new and simpler proof.

THEOREM 6.5

A_{TM} is undecidable.

PROOF We assume that Turing machine H decides A_{TM}, for the purpose of obtaining a contradiction. We construct the following machine B.

```
B = "On input w:
    1. Obtain, via the recursion theorem, own description ⟨B⟩.
    2. Run H on input ⟨B, w⟩.
    3. Do the opposite of what H says. That is, accept if H rejects and
       reject if H accepts."
```

Running B on input w does the opposite of what H declares it does. Therefore, H cannot be deciding A_{TM}. Done!

The following theorem concerning minimal Turing machines is another application of the recursion theorem.

DEFINITION 6.6

If M is a Turing machine, then we say that the *length* of the description $\langle M \rangle$ of M is the number of symbols in the string describing M. Say that M is ***minimal*** if there is no Turing machine equivalent to M that has a shorter description. Let

$$MIN_{\mathsf{TM}} = \{\langle M \rangle |\ M \text{ is a minimal TM}\}.$$

THEOREM 6.7

MIN_{TM} is not Turing-recognizable.

PROOF Assume that some TM E enumerates MIN_{TM} and obtain a contradiction. We construct the following TM C.

```
C = "On input w:
    1. Obtain, via the recursion theorem, own description ⟨C⟩.
    2. Run the enumerator E until a machine D appears with a longer
       description than that of C.
    3. Simulate D on input w."
```

Because MIN_{TM} is infinite, E's list must contain a TM with a longer description than C's description. Therefore, step 2 of C eventually terminates with some TM D that is longer than C. Then C simulates D and so is equivalent to it. Because C is shorter than D and is equivalent to it, D cannot be minimal. But D appears on the list that E produces. Thus, we have a contradiction.

Our final application of the recursion theorem is a type of fixed-point theorem. A ***fixed point*** of a function is a value that isn't changed by application of the function. In this case, we consider functions that are computable transformations of Turing machine descriptions. We show that for any such transformation, some Turing machine exists whose behavior is unchanged by the transformation. This theorem is called the fixed-point version of the recursion theorem.

THEOREM 6.8

Let $t\colon \Sigma^* \longrightarrow \Sigma^*$ be a computable function. Then there is a Turing machine F for which $t(\langle F\rangle)$ describes a Turing machine equivalent to F. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.

In this theorem, t plays the role of the transformation, and F is the fixed point.

PROOF Let F be the following Turing machine.

```
F = "On input w:
    1. Obtain, via the recursion theorem, own description ⟨F⟩.
    2. Compute t(⟨F⟩) to obtain the description of a TM G.
    3. Simulate G on w."
```

Clearly, $\langle F\rangle$ and $t(\langle F\rangle) = \langle G\rangle$ describe equivalent Turing machines because F simulates G.

6.2

DECIDABILITY OF LOGICAL THEORIES

Mathematical logic is the branch of mathematics that investigates mathematics itself. It addresses questions such as: What is a theorem? What is a proof? What is truth? Can an algorithm decide which statements are true? Are all true statements provable? We'll touch on a few of these topics in our brief introduction to this rich and fascinating subject.

We focus on the problem of determining whether mathematical statements are true or false and investigate the decidability of this problem. The answer depends on the domain of mathematics from which the statements are drawn. We examine two domains: one for which we can give an algorithm to decide truth, and another for which this problem is undecidable.

First, we need to set up a precise language to formulate these problems. Our intention is to be able to consider mathematical statements such as

1. $\forall q\, \exists p\, \forall x,y\, \big[\, p{>}q \wedge (x,y{>}1 \rightarrow xy{\neq}p)\,\big]$,
2. $\forall a,b,c,n\, \big[(a,b,c{>}0 \wedge n{>}2) \rightarrow a^n{+}b^n{\neq}c^n \,\big]$, and
3. $\forall q\, \exists p\, \forall x,y\, \big[\, p{>}q \wedge (x,y{>}1 \rightarrow (xy{\neq}p \wedge xy{\neq}p{+}2))\,\big]$.

Statement 1 says that infinitely many prime numbers exist, which has been known to be true since the time of Euclid, about 2,300 years ago. Statement 2 is *Fermat's last theorem*, which has been known to be true only since Andrew Wiles proved it in 1994. Finally, statement 3 says that infinitely many prime pairs[1] exist. Known as the *twin prime conjecture*, it remains unsolved.

[1] ***Prime pairs*** are primes that differ by 2.