Introduction to the Theory of Computation
Models of Computation, Undecidability
Complexity Classes ($\mathcal{P}$ and $\mathcal{NP}$)
Slides for CIS511

Jean Gallier

March 22, 2021

Chapter 1

RAM Programs, Turing Machines, and the Partial Computable Functions

In this chapter we address the fundamental question

What is a computable function?

Nowadays computers are so pervasive that such a question may seem trivial.

Isn't the answer that a function is computable if we can write a program computing it!

This is basically the answer so what more can be said that will shed more light on the question?

The first issue is that we should be more careful about the kind of functions that we are considering.

Are we restricting ourselves to *total functions* or are we allowing *partial functions* that may not be defined for some of their inputs?

It turns out that if we consider functions computed by programs, then partial functions must be considered.

In fact, we will see that "deciding" whether a program terminates for all inputs is impossible. But what does deciding mean?

To be mathematically precise requires a fair amount of work.

One of the key technical points is the ability to design a program U that takes other programs P as input, and then executes P on any input x. In particular, U should be able to take U itself as input!

Of course a compiler does exactly the above task.

But fully describing a compiler for a "real" programming language such as JAVA, PYTHON, C++, *etc.* is a complicated and lengthy task.

So a simpler (still quite complicated) way to proceed is to develop a toy programming language and a toy computation model (some kind of machine) capable of executing programs written in our toy language.

Then we show how programs in this toy language can be coded so that they can be given as input to other programs.

Having done this we need to demonstrate that our language has *universal computing power*. This means that we need to show that a "real" program, say written in JAVA, could be translated into a possibly much longer program written in our toy language.

This step is typically an act of faith, in the sense that the details that such a translation can be performed are usually not provided.

A way to be precise regarding universal computing power is to define mathematically a family of functions that should be regarded as "obviously computable," and then to show that the functions computed by the programs written either in our toy programming language or in any modern progamming language are members of this mathematically defined family of computable functions.

This step is usually technically very involved, because one needs to show that executing the instructions of a program can be mimicked by functions in our family of computable functions.

Conversely, we should prove that every computable function in this family is indeed computable by a program written in our toy programming language or in any modern progamming language.

Then we will be have the assurance that we have captured the notion of universal computing power.

Remarkably, *Herbrand, Gödel, and Kleene defined such a family of functions in 1934-1935*.

This is a family of numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$ containing a subset of very simple functions called base functions, and this family is the smallest family containing the base functions closed under three operations:

1. Composition
2. Primitive recursion
3. Minimization.

Historically, the first two models of computation are the *λ-calculus* of Church (1935) and the *Turing machine* (1936) of Turing.

Kleene proved that the λ-definable functions are exactly the (total) computable functions in the sense of Herbrand–Gödel–Kleene in 1936, and Turing proved that the functions computed by Turing machines are exactly the computable functions in the sense of Herbrand–Gödel–Kleene in 1937.

Therefore, the λ-calculus and Turing machines have the same "computing power," and both compute exactly the class of computable functions in the sense of Herbrand–Gödel–Kleene.

In those days these results were considered quite surprising because the formalism of the λ-calculus has basically nothing to do with the formalism of Turing machines.

Once again we should be more precise about the kinds of functions that we are dealing with.

Until Turing (1936), only numerical functions $f\colon \mathbb{N}^m \to \mathbb{N}$ were considered.

In order to compute numerical functions in the λ-calculus, Church had to encode the natural numbers are certain λ-terms, which can be viewed as iterators.

Turing assumes that what he calls his *a-machines* (for automatic machines) make use of the symbols 0 and 1 for the purpose of input and output, and if the machine stops, then the output is a *string* of 0s and 1s.

Thus a Turing machine can be viewed as computing a function $f\colon (\{0,1\}^*)^m \to \{0,1\}^*$ *on strings*.

By allowing a more general alphabet Σ, we see that a Turing machine computes a function $f\colon (\Sigma^*)^m \to \Sigma^*$ on strings over Σ.

At first glance it appears that Turing machines compute a larger class of functions, but this is not so because there exist mutually invertible computable coding functions $C\colon \Sigma^* \to \mathbb{N}$ and decoding functions $D\colon \mathbb{N} \to \Sigma^*$.

Using these coding and decoding functions, it suffices to consider numerical functions.

However, Turing machines can also very naturally be viewed as devices for defining computable languages in terms of acceptance and rejection; some kinds of generalized DFA's of NFA's.

In this role, it would be very awkward to limit ourselves to sets of natural numbers, although this is possible in theory.

We should also point out that the notion of computable language can be handled in terms of a computation model for functions by considering the characteristic functions of languages.

Indeed, a language A is computable (we say decidable) iff its characteristic function χ_A is computable.

The above considerations motivate the definition of the computable functions in the sense of Herbrand–Gödel–Kleene to functions $f\colon (\Sigma^*)^m \to \Sigma^*$ operating *on strings*.

However, it is technically simpler to work out all the undecidabilty results for numerical functions of for subsets of $\mathbb{N}$.

Since there is no loss of generally in doing so in view of the computable bijections $C\colon \Sigma^* \to \mathbb{N}$ and $D\colon \mathbb{N} \to \Sigma^*$, we will do so.

Nevertherless, in order to deal with languages, it is important to develop a fair amount of computabilty theory about functions computing on strings, so we will present another computation model, the *RAM program model*, which computes functions defined on strings.

This model was introduced around 1963 (although it was introduced earlier by Post in a different format). It has the advantage of being closer to actual computer architecture, because the RAM model consists of programs operating on a fixed set of registers.

This model is equivalent to the Turing machine model, and the translations, although tedious, are not that bad.

The RAM program model also has the technical advantage that coding up a RAM program as a natural number is not that complicated.

The λ-calculus is a very elegant model but it is more abstract than the RAM program model and the Turing machine model so we will not discuss it in this course.

Another very interesting computation model particularly well suited to deal with decidable sets of natural numbers is *Diophantine definability*.

This model, arising from the work involved in proving that Hilbert's tenth problem is undecidable, will be discussed in Chapter 6.

1.1 Partial Functions and RAM Programs

We define an abstract machine model for computing functions

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_k\}$ is some input alphabet.

Numerical functions $f\colon \mathbb{N}^n \to \mathbb{N}$ can be viewed as functions defined over the one-letter alphabet $\{a_1\}$, using the bijection $m \mapsto a_1^m$.

Since programs are not guaranteed to terminate for all inputs, we are forced to deal with partial functions so we recall their definition.

A binary relation $R \subseteq A \times B$ between two sets A and B is *functional* iff, for all $x \in A$ and $y, z \in B$,

$$(x, y) \in R \quad \text{and} \quad (x, z) \in R \quad \text{implies that} \quad y = z.$$

Definition 1.1. A *partial function* is a triple $f = \langle A, G, B \rangle$, where A and B are arbitrary sets (possibly empty) and G is a functional relation (possibly empty) between A and B, called the *graph* of f.

Hence, a partial function is a functional relation such that every argument *has at most one image under f*.

The graph of a function f is denoted as $graph(f)$. When no confusion can arise, a function f and its graph are usually identified.

A partial function $f = \langle A, G, B \rangle$ is often denoted as $f \colon A \to B$.

The *domain* $dom(f)$ of a partial function $f = \langle A, G, B \rangle$ is the set

$$dom(f) = \{x \in A \mid \exists y \in B,\ (x, y) \in G\}.$$

For every element $x \in dom(f)$, the unique element $y \in B$ such that $(x, y) \in graph(f)$ is denoted as $f(x)$. We say that *$f(x)$ is defined*, also denoted as $f(x) \downarrow$.

If $x \in A$ and $x \notin dom(f)$, we say that *$f(x)$ is undefined*, also denoted as $f(x) \uparrow$.

Intuitively, if a function is partial, it does not return any output for any input not in its domain. This corresponds to an infinite computation.

A partial function $f \colon A \to B$ is a *total function* iff $dom(f) = A$. It is customary to call a total function simply a function.

We now define a model of computation know as the *RAM programs*, or *Post machines*.

RAM programs are written in a sort of assembly language involving simple instructions manipulating strings stored into registers.

Every RAM program uses a fixed and finite number of *registers* denoted as $R1, \ldots, Rp$, with no limitation on the size of strings held in the registers.

RAM programs can be defined either in flowchart form or in linear form. Since the linear form is more convenient for coding purposes, we present RAM programs in linear form.

A RAM program P (in linear form) consists of a finite sequence of *instructions* using a finite number of registers $R1, \ldots, Rp$.

Instructions may optionally be labeled with line numbers denoted as $N1, \ldots, Nq$.

It is neither mandatory to label all instructions, nor to use distinct line numbers!

Thus, the same line number can be used in more than one line. As we will see later on, this makes it easier to concatenate two different programs without performing a renumbering of line numbers.

Every instruction has *four fields*, not necessarily all used. The main field is the **op-code**.

Here is an example of a RAM program to concatenate two strings x_1 and x_2.

$$
\begin{array}{llll}
 & R3 & \leftarrow & R1 \\
 & R4 & \leftarrow & R2 \\
N0 & R4 & \texttt{jmp}_a & N1b \\
 & R4 & \texttt{jmp}_b & N2b \\
 & & \texttt{jmp} & N3b \\
N1 & & \texttt{add}_a & R3 \\
 & & \texttt{tail} & R4 \\
 & & \texttt{jmp} & N0a \\
N2 & & \texttt{add}_b & R3 \\
 & & \texttt{tail} & R4 \\
 & & \texttt{jmp} & N0a \\
N3 & R1 & \leftarrow & R3 \\
 & & \texttt{continue} &
\end{array}
$$

Definition 1.2. *RAM programs* are constructed from seven types of *instructions* shown below:

(1_j)	N		$\texttt{add}_j$	Y
(2)	N		$\texttt{tail}$	Y
(3)	N		$\texttt{clr}$	Y
(4)	N	Y	$\leftarrow$	X
$(5a)$	N		$\texttt{jmp}$	$N1a$
$(5b)$	N		$\texttt{jmp}$	$N1b$
(6_ja)	N	Y	$\texttt{jmp}_j$	$N1a$
(6_jb)	N	Y	$\texttt{jmp}_j$	$N1b$
(7)	N		$\texttt{continue}$	

An instruction of type (1_j) concatenates the letter a_j to the right of the string held by register Y $(1 \leq j \leq k)$. The effect is the assignment

$$Y := Ya_j$$

An instruction of type (2) deletes the leftmost letter of the string held by the register Y. This corresponds to the function *tail*, defined such that

$$\begin{aligned} tail(\epsilon) &= \epsilon, \\ tail(a_j u) &= u. \end{aligned}$$

The effect is the assignment

$$Y := tail(Y)$$

An instruction of type (3) clears register Y, i.e., sets its value to the empty string ϵ. The effect is the assignment

$$Y := \epsilon$$

An instruction of type (4) assigns the value of register X to register Y. The effect is the assignment

$$Y := X$$

An instruction of type (5a) or (5b) is an unconditional jump.

The effect of (5a) is to jump to the closest line number $N1$ occurring above the instruction being executed, and the effect of (5b) is to jump to the closest line number $N1$ occurring below the instruction being executed.

An instruction of type $(6_j a)$ or $(6_j b)$ is a conditional jump. Let *head* be the function defined as follows:

$$head(\epsilon) = \epsilon,$$
$$head(a_j u) = a_j.$$

The effect of $(6_j a)$ is to jump to the closest line number $N1$ occurring above the instruction being executed iff $head(Y) = a_j$, else to execute the next instruction (the one immediately following the instruction being executed).

The effect of $(6_j b)$ is to jump to the closest line number $N1$ occurring below the instruction being executed iff $head(Y) = a_j$, else to execute the next instruction.

When computing over $\mathbb{N}$, instructions of type $(6_j a)$ or $(6_j b)$ jump to the closest $N1$ above or below iff Y is nonnull.

An instruction of type (7) is a no-op, i.e., the registers are unaffected. If there is a next instruction, then it is executed, else, the program stops.

Obviously, a program is syntactically correct only if certain conditions hold.

Definition 1.3. A *RAM program* P is a finite sequence of instructions as in Definition 1.2, and satisfying the following conditions:

(1) For every jump instruction (conditional or not), the line number to be jumped to must exist in P.

(2) The last instruction of a RAM program is a `continue`.

The reason for allowing multiple occurences of line numbers is to make it easier to concatenate programs without having to perform a renaming of line numbers.

The technical choice of jumping to the closest address $N1$ above or below comes from the fact that it is easy to search up or down using primitive recursion, as we will see later on.

For the purpose of computing a function $f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$ using a RAM program P, we assume that P has at least n registers called *input registers*, and that these registers $R1, \ldots, Rn$ are initialized with the input values of the function f.

We also assume that *the output is returned in register $R1$*.

Example 1.1. The following RAM program concatenates two strings x_1 and x_2 held in registers $R1$ and $R2$.

Since $\Sigma = \{a, b\}$, for more clarity, we wrote $\mathtt{jmp}_a$ instead of $\mathtt{jmp}_1$, $\mathtt{jmp}_b$ instead of $\mathtt{jmp}_2$, $\mathtt{add}_a$ instead of $\mathtt{add}_1$, and $\mathtt{add}_b$ instead of $\mathtt{add}_2$.

$$
\begin{array}{llcl}
 & R3 & \leftarrow & R1 \\
 & R4 & \leftarrow & R2 \\
N0 & R4 & \mathtt{jmp}_a & N1b \\
 & R4 & \mathtt{jmp}_b & N2b \\
 & & \mathtt{jmp} & N3b \\
N1 & & \mathtt{add}_a & R3 \\
 & & \mathtt{tail} & R4 \\
 & & \mathtt{jmp} & N0a \\
N2 & & \mathtt{add}_b & R3 \\
 & & \mathtt{tail} & R4 \\
 & & \mathtt{jmp} & N0a \\
N3 & R1 & \leftarrow & R3 \\
 & & \mathtt{continue} &
\end{array}
$$

Definition 1.4. A RAM program P *computes the partial function* $\varphi\colon (\Sigma^*)^n \to \Sigma^*$ if the following conditions hold: For every input $(x_1, \ldots, x_n) \in (\Sigma^*)^n$, having initialized the input registers $R1, \ldots, Rn$ with $x_1, \ldots, x_n$, the program eventually halts iff $\varphi(x_1, \ldots, x_n)$ is defined, and if and when P halts, the value of $R1$ is equal to $\varphi(x_1, \ldots, x_n)$.

A partial function φ is *RAM-computable* iff it is computed by some RAM program.

For example, the following program computes the *erase function E* defined such that

$$E(u) = \epsilon$$

for all $u \in \Sigma^*$:

```
clr       R1
continue
```

The following program computes the *jth successor function S_j* defined such that

$$S_j(u) = ua_j$$

for all $u \in \Sigma^*$:

$$\begin{array}{ll} \texttt{add}_j & R1 \\ \texttt{continue} & \end{array}$$

The following program (with n input variables) computes the *projection function P_i^n* defined such that

$$P_i^n(u_1, \ldots, u_n) = u_i,$$

where $n \geq 1$, and $1 \leq i \leq n$:

$$\begin{array}{ll} R1 \leftarrow & Ri \\ \texttt{continue} & \end{array}$$

Note that P_1^1 is the identity function.

Having a programming language, we would like to know how powerful it is, that is, we would like to know what kind of functions are RAM-computable.

At first glance, RAM programs don't do much, but this is not so. Indeed, we will see shortly that the class of RAM-computable functions is quite extensive.

One way of getting new programs from previous ones is via composition. Another one is by primitive recursion.

We will investigate these constructions after introducing another model of computation, *Turing machines*.

Remarkably, the classes of (partial) functions computed by RAM programs and by Turing machines are identical.

This is the class of *partial computable functions*, also called *partial recursive functions*, a term which is now considered old-fashion.

This class can be given several other definitions.

The following Lemma will be needed to simplify the encoding of RAM programs as numbers.

Lemma 1.1. *Every RAM program can be converted to an equivalent program only using the following type of instructions:*

(1_j)	N		$\texttt{add}_j$	Y
(2)	N		$\texttt{tail}$	Y
$(6_j a)$	N	Y	$\texttt{jmp}_j$	$N1a$
$(6_j b)$	N	Y	$\texttt{jmp}_j$	$N1b$
(7)	N		$\texttt{continue}$	

The proof is fairly simple. For example, instructions of the form

$$Ri \leftarrow Rj$$

can be eliminated by transferring the contents of Rj into an auxiliary register Rk, and then by transferring the contents of Rk into Ri and Rj.

1.2 Definition of a Turing Machine

We define a Turing machine model for computing functions

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*,$$

where $\Sigma = \{a_1, \ldots, a_k\}$ is some input alphabet. In this section, since we are primarily interested in computing functions we only consider deterministic Turing machines.

There are many variants of the Turing machine model.

The main decision that needs to be made has to do with the kind of tape used by the machine.

We opt for a single *finite tape* that is both an input and a storage mechanism.

This tape can be viewed as a string over *tape alphabet* Γ such that $\Sigma \subseteq \Gamma$.

There is a read/write head pointing to some symbol on the tape, symbols on the tape can be overwritten, and the read/write head can move one symbol to the left or one symbol to the right, also causing a state transition.

When the write/read head attempts to move past the rightmost or the leftmost symbol on the tape, the tape is allowed to grow.

To accomodate such a move, the tape alphabet contains some special symbol $B \notin \Sigma$, the *blank*, and this symbol is added to the tape as the new leftmost or rightmost symbol on the tape.

A common variant uses a tape which is infinite at both ends, but only has finitely many symbols not equal to B, so effectively it is equivalent to a finite tape allowed to grow at either ends.

Another variant uses a semi-infinite tape infinite to the right, but with a left end.

We find this model cumbersome because it requires shifting right the entire tape when a left move is attempted from the left end of the tape.

Another decision that needs to be made is the format of the instructions.

Does an instruction cause *both* a state transition *and* a symbol overwrite, or do we have separate instructions for a state transition and a symbol overwrite.

In the first case, an instruction can be specified as a quintuple, and in the second case by a quadruple. We opt for quintuples.

Definition 1.5. A (deterministic) *Turing machine* (or *TM*) M is a sextuple $M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$, where

- K is a finite set of *states*;
- Σ is a finite *input alphabet*;
- Γ is a finite *tape alphabet*, s.t. $\Sigma \subseteq \Gamma$, $K \cap \Gamma = \emptyset$, and with blank $B \notin \Sigma$;
- $q_0 \in K$ is the *start state* (or *initial state*);
- δ is the *transition function*, a (finite) set of quintuples
$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$
such that for all $(p, a) \in K \times \Gamma$, there is at most one triple $(b, m, q) \in \Gamma \times \{L, R\} \times K$ such that $(p, a, b, m, q) \in \delta$.

A quintuple $(p, a, b, m, q) \in \delta$ is called an *instruction*. It is also denoted as

$$p, a \rightarrow b, m, q.$$

The effect of an instruction is to switch from state p to state q, overwrite the symbol currently scanned a with b, and move the read/write head either left or right, according to m.

Here is an example of a Turing machine.

Example 1.2. $K = \{q_0, q_1, q_2, q_3\}$;

$\Sigma = \{a, b\}$;

$\Gamma = \{a, b, B\}$;

The instructions in δ are:

$$
\begin{gathered}
q_0, B \to B, R, q_3, \\
q_0, a \to b, R, q_1, \\
q_0, b \to a, R, q_1, \\
q_1, a \to b, R, q_1, \\
q_1, b \to a, R, q_1, \\
q_1, B \to B, L, q_2, \\
q_2, a \to a, L, q_2, \\
q_2, b \to b, L, q_2, \\
q_2, B \to B, R, q_3.
\end{gathered}
$$

1.3 Computations of Turing Machines

To explain how a Turing machine works, we describe its action on *Instantaneous descriptions*. We take advantage of the fact that $K \cap \Gamma = \emptyset$ to define instantaneous descriptions.

Definition 1.6. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

an *instantaneous description* (for short an *ID*) is a (nonempty) string in $\Gamma^* K \Gamma^+$, that is, a string of the form

$$upav,$$

where $u, v \in \Gamma^*$, $p \in K$, and $a \in \Gamma$.

The intuition is that an ID $upav$ describes a snapshot of a TM in the current state p, whose tape contains the string uav, and with the read/write head pointing to the symbol a.

Thus, in $upav$, the state p is just to the left of the symbol presently scanned by the read/write head.

We explain how a TM works by showing how it acts on ID's.

Definition 1.7. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

the *yield relation (or compute relation)* $\vdash$ is a binary relation defined on the set of ID's as follows. For any two ID's ID_1 and ID_2, we have $ID_1 \vdash ID_2$ iff either

(1) $(p, a, b, R, q) \in \delta$, and either

  (a) $ID_1 = upacv$, $c \in \Gamma$, and $ID_2 = ubqcv$, or
  (b) $ID_1 = upa$ and $ID_2 = ubqB$;

or

(2) $(p, a, b, L, q) \in \delta$, and either

  (a) $ID_1 = ucpav$, $c \in \Gamma$, and $ID_2 = uqcbv$, or
  (b) $ID_1 = pav$ and $ID_2 = qBbv$.

Note how the tape is extended by one blank after the rightmost symbol in case (1)(b), and by one blank before the leftmost symbol in case (2)(b).

As usual, we let $\vdash^+$ denote the transitive closure of $\vdash$, and we let $\vdash^*$ denote the reflexive and transitive closure of $\vdash$.

We can now explain how a Turing machine computes a partial function

$$f\colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*.$$

Since we allow functions taking $n \geq 1$ input strings, we assume that Γ contains the special delimiter $\mathbf{,}$ not in Σ, used to separate the various input strings.

It is convenient to assume that a Turing machine "cleans up" its tape when it halts before returning its output.

What this means is that when the Turing machine halts, the output should be clearly identifiable, so all symbols not in $\Sigma \cup \{B\}$ that may have been used during the computation must be erased.

Thus when the TM stops the tape must consist of a string $w \in \Sigma^*$ possibly surrounded by blanks (the symbol B).

Actually, if the output is ϵ, the tape must contain a nonempty string of blanks. To achieve this technically, we define proper ID's.

Definition 1.8. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

where Γ contains some delimiter **,** not in Σ in addition to the blank B, a *starting ID* is of the form

$$q_0 w_1 , w_2 , \ldots , w_n$$

where $w_1, \ldots, w_n \in \Sigma^*$ and $n \geq 2$, or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$.

A *blocking (or halting) ID* is an ID *upav* such that there are no instructions $(p, a, b, m, q) \in \delta$ for any $(b, m, q) \in \Gamma \times \{L, R\} \times K$.

A *proper ID* is a halting ID of the form

$$B^h p w B^l,$$

where $w \in \Sigma^*$, and $h, l \geq 0$ (with $l \geq 1$ when $w = \epsilon$).

Computation sequences are defined as follows.

Definition 1.9. Given a Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0),$$

a *computation sequence (or computation)* is a finite or infinite sequence of ID's

$$ID_0, ID_1, \ldots, ID_i, ID_{i+1}, \ldots,$$

such that $ID_i \vdash ID_{i+1}$ for all $i \geq 0$.

A computation sequence *halts* iff it is a finite sequence of ID's, so that

$$ID_0 \vdash^* ID_n,$$

and ID_n is a halting ID.

A computation sequence *diverges* if it is an infinite sequence of ID's.

We now explain how a Turing machine computes a partial function.

Definition 1.10. A Turing machine

$$M = (K, \Sigma, \Gamma, \{L, R\}, \delta, q_0)$$

computes the partial function

$$f \colon \underbrace{\Sigma^* \times \cdots \times \Sigma^*}_{n} \to \Sigma^*$$

iff the following conditions hold:

(1) For every $w_1, \ldots, w_n \in \Sigma^*$, given the starting ID

$$ID_0 = q_0 w_1 , w_2 , \ldots , w_n$$

or $q_0 w$ with $w \in \Sigma^+$, or $q_0 B$, the computation sequence of M from ID_0 halts in a proper ID iff $f(w_1, \ldots, w_n)$ is defined.

(2) If $f(w_1, \ldots, w_n)$ is defined, then M halts in a proper ID of the form

$$ID_n = B^h p f(w_1, \ldots, w_n) B^l,$$

which means that it computes the right value.

A function f (over Σ^*) is *Turing computable* iff it is computed by some Turing machine M.

Note that by (1), the TM M may halt in an improper ID, in which case $f(w_1, \ldots, w_n)$ must be undefined. This corresponds to the fact that *we only accept to retrieve the output of a computation if the TM has cleaned up its tape*, i.e., produced a proper ID. In particular, intermediate calculations have to be erased before halting.

Example 1.3. $K = \{q_0, q_1, q_2, q_3\}$;
$\Sigma = \{a, b\}$;
$\Gamma = \{a, b, B\}$;
The instructions in δ are:

$$\begin{aligned}
q_0, B &\to B, R, q_3,\\
q_0, a &\to b, R, q_1,\\
q_0, b &\to a, R, q_1,\\
q_1, a &\to b, R, q_1,\\
q_1, b &\to a, R, q_1,\\
q_1, B &\to B, L, q_2,\\
q_2, a &\to a, L, q_2,\\
q_2, b &\to b, L, q_2,\\
q_2, B &\to B, R, q_3.
\end{aligned}$$

The reader can easily verify that this machine exchanges the a's and b's in a string. For example, on input $w = aaababb$, the output is $bbbabaa$.

1.4 Equivalence of RAM Programs and Turing Machines

Turing machines can simulate RAM programs, and as a result, we have the following Theorem.

Theorem 1.2. *Every RAM-computable function is Turing-computable. Furthermore, given a RAM program P, we can effectively construct a Turing machine M computing the same function.*

The idea of the proof is to represent the contents of the registers $R1, \ldots Rp$ on the Turing machine tape by the string

$$\#r1\#r2\# \cdots \#rp\#,$$

where # is a special marker and ri represents the string held by Ri. We also use Lemma 1.1 to reduce the number of instructions to be dealt with.

The Turing machine M is built of blocks, each block simulating the effect of some instruction of the program P. The details are a bit tedious, and can be found in the notes or in Machtey and Young.

RAM programs can also simulate Turing machines.

Theorem 1.3. *Every Turing-computable function is RAM-computable. Furthermore, given a Turing machine M, one can effectively construct a RAM program P computing the same function.*

The idea of the proof is to design a RAM program containing an encoding of the current ID of the Turing machine M in register $R1$, and to use other registers $R2, R3$ to simulate the effect of executing an instruction of M by updating the ID of M in $R1$.

The details are tedious and can be found in the notes.

Another proof can be obtained by proving that the class of Turing computable functions coincides with the class of *partial computable functions* (formerly called *partial recursive functions*).

Indeed, it turns out that both RAM programs and Turing machines compute precisely the class of partial recursive functions.

For this, we need to define the *primitive recursive functions*.

Informally, a primitive recursive function is a total recursive function that can be computed using only **for** loops, that is, loops in which the number of iterations is fixed (unlike a **while** loop).

A formal definition of the primitive functions is given in Section 1.7.

Definition 1.11. Let $\Sigma = \{a_1, \ldots, a_N\}$. The class of *partial computable functions* also called *partial recursive functions* is the class of partial functions (over Σ^*) that can be computed by RAM programs (or equivalently by Turing machines).

The class of *computable functions* also called *recursive functions* is the subset of the class of partial computable functions consisting of functions defined for every input (i.e., total functions).

Turing machines can also be used as acceptors to define languages so we introduce the basic relevant definitions.

1.5 Listable Languages and Computable Languages

We define the computably enumerable languages, also called listable languages, and the computable languages.

The old-fashion terminology for listable languages is recursively enumerable languages, and for computable languages is recursive languages.

When operating as an acceptor, a Turing machine takes a single string as input and either goes on forever or halts with the answer "accept" or "reject."

One way to deal with acceptance or rejection is to assume that the TM has a set of final states.

Another way more consistent with our view that machines compute functions is to assume that the TM's under consideration have a tape alphabet containing the special symbols 0 and 1.

Then acceptance is signaled by the output 1, and rejection is signaled by the output 0.

Note that with our convention that in order to produce an output a TM must halt in a proper ID, the TM must erase the tape before outputing 0 or 1.

Definition 1.12. Let $\Sigma = \{a_1, \ldots, a_N\}$. A language $L \subseteq \Sigma^*$ is *(Turing) listable* or *(Turing) computably enumerable (for short, a c.e. set)* (or *recursively enumerable (for short, a r.e. set)*) iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, either M halts in a proper ID with the output 0, or it runs forever.

A language $L \subseteq \Sigma^*$ is *(Turing) computable* (or *recursive*) iff there is some TM M such that for every $w \in L$, M halts in a proper ID with the output 1, and for every $w \notin L$, M halts in a proper ID with the output 0.

Thus, given a computably enumerable language L, for some $w \notin L$, it is possible that a TM accepting L runs forever on input w. On the other hand, for a computable (recursive) language L, a TM accepting L always halts in a proper ID.

When dealing with languages, it is often useful to consider *nondeterministic Turing machines*. Such machines are defined just like deterministic Turing machines, except that their transition function δ is just a (finite) set of quintuples

$$\delta \subseteq K \times \Gamma \times \Gamma \times \{L, R\} \times K,$$

with no particular extra condition.

It can be shown that every nondeterministic Turing machine can be simulated by a deterministic Turing machine, and thus, nondeterministic Turing machines also accept the class of c.e. sets.

It can be shown that a computably enumerable language is the range of some computable (recursive) function. It can also be shown that a language L is computable (recursive) iff both L and its complement are computably enumerable. There are computably enumerable languages that are not computable (recursive).

1.6 A Simple Function Not Known to be Computable

The “$3n + 1$ problem” proposed by Collatz around 1937 is the following:

Given any positive integer $n \geq 1$, construct the sequence $c_i(n)$ as follows starting with $i = 1$:

$$\begin{aligned} c_1(n) &= n \\ c_{i+1}(n) &= \begin{cases} c_i(n)/2 & \text{if } c_i(n) \text{ is even} \\ 3c_i(n) + 1 & \text{if } c_i(n) \text{ is odd.} \end{cases} \end{aligned}$$

Observe that for $n = 1$, we get the infinite periodic sequence

$$1 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1 \Longrightarrow \cdots ,$$

so we may assume that we stop the first time that the sequence $c_i(n)$ reaches the value 1 (if it actually does).

Such an index i is called the *stopping time* of the sequence. And this is the problem:

Conjecture (Collatz):

For any starting integer value $n \geq 1$, the sequence $(c_i(n))$ always reaches 1.

Starting with $n = 3$, we get the sequence

$$3 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 5$, we get the sequence

$$5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 6$, we get the sequence

$$6 \Longrightarrow 3 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.$$

Starting with $n = 7$, we get the sequence

$$\begin{aligned}
&7 \Longrightarrow 22 \Longrightarrow 11 \Longrightarrow 34 \Longrightarrow 17 \Longrightarrow 52 \Longrightarrow 26 \\
&\Longrightarrow 13 \Longrightarrow 40 \Longrightarrow 20 \Longrightarrow 10 \Longrightarrow 5 \Longrightarrow 16 \\
&\Longrightarrow 8 \Longrightarrow 4 \Longrightarrow 2 \Longrightarrow 1.
\end{aligned}$$

One might be surprised to find that for $n = 27$, it takes 111 steps to reach 1, and for $n = 97$, it takes 118 steps.

I computed the stopping times for n up to 10^7 and found that the largest stopping time, 686 (685 steps) is obtained for $n = 8400511$.

The terms of this sequence reach values over 1.5×10^{11}. The graph of the sequence $c(8400511)$ is shown in Figure 1.1.

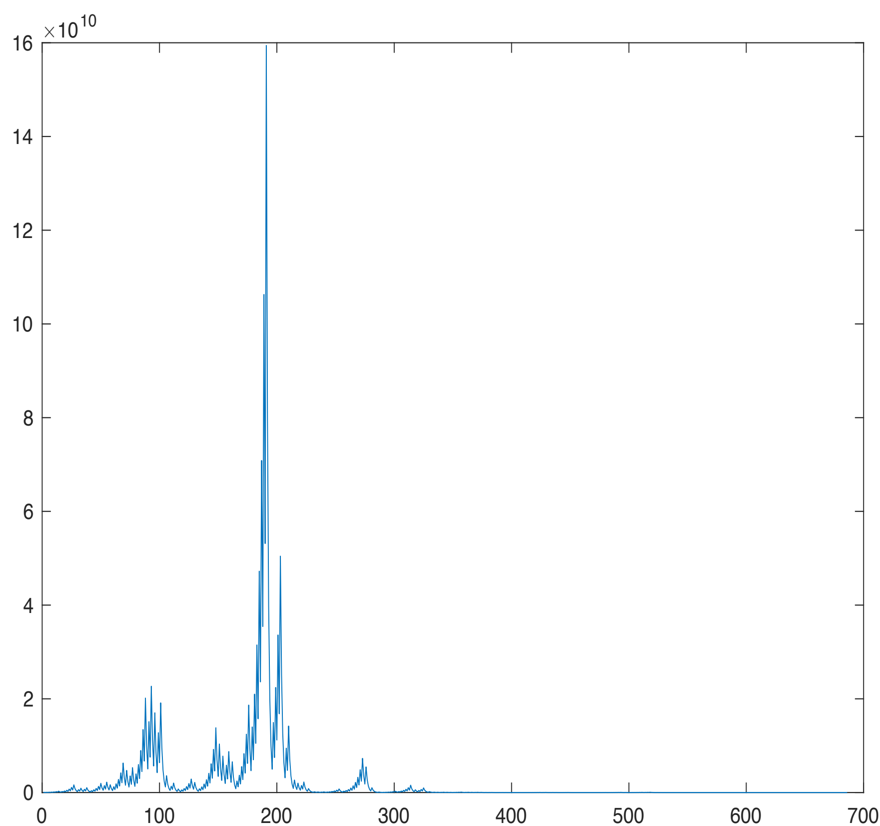

Figure 1.1: Graph of the sequence for $n = 8400511$.

We can define the partial computable function C (with positive integer inputs) defined by

$$C(n) = \text{the smallest } i \text{ such that } c_i(n) = 1 \text{ if it exists.}$$

Then the Collatz conjecture is equivalent to asserting that the function C is (total) computable.

The graph of the function C for $1 \leq n \leq 10^7$ is shown in Figure 1.2. So far, the conjecture remains open. It has

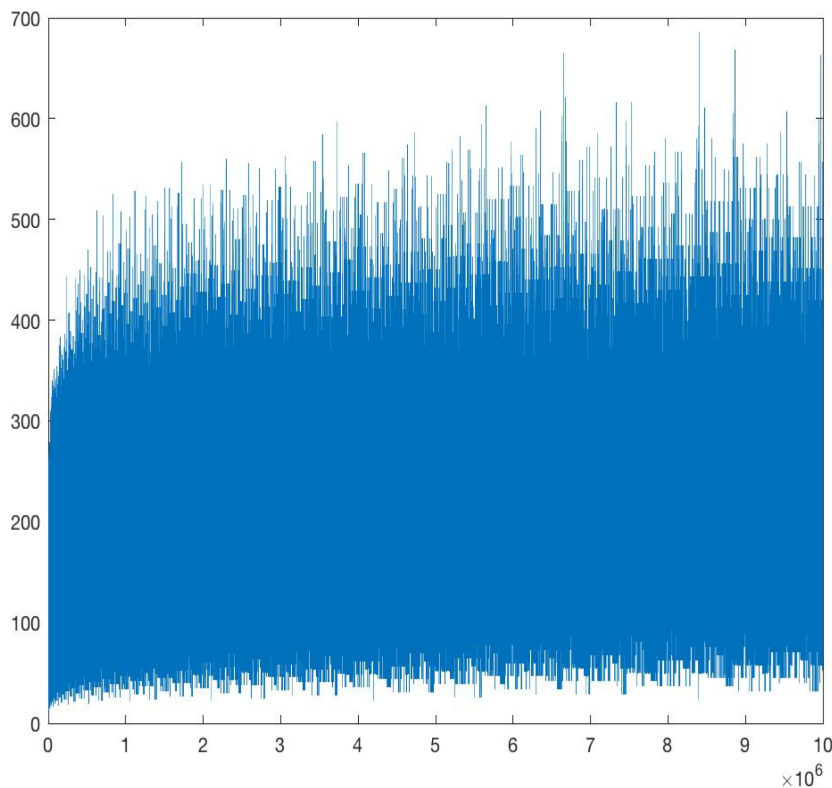

Figure 1.2: Graph of the function C for $1 \leq n \leq 10^7$.

been checked by computer for all integers $\leq 87 \times 2^{60}$.