COMPUTABILITY THEORY

Formal and Mathematical Logic Project

Author
Shahd Elmahallawy - 900194441
Yehia Elkasas - 900202395
Gehad Ahmed - 900205068

Contents

1 Automata and Languages

1.1 Introduction to Computation and Finite Automata

The journey into the theory of computation begins with a seemingly simple question: What exactly constitutes a computer? While modern computers are highly complex and multifaceted, for the purposes of theoretical exploration, we often rely on simplified models known as computational models. Among these, the **finite automaton** is particularly noted for its simplicity and practicality, serving as an ideal model for systems with minimal memory. Despite their straightforward nature, finite automata are essential in the operation of numerous everyday devices and systems, highlighting their significance beyond basic theoretical constructs [1].

1.2 Understanding Finite Automata

A finite automato is a theoretical model that represents systems with a limited number of states, transitioning between them based on external inputs via a defined transition function. A practical example is the automatic door controller that switches between OPEN and CLOSED states depending on whether a person is detected, as illustrated in Figure 1. This controller demonstrates how finite automata apply a standardized method to depict the simple memory requirements and state transitions typical in such systems. While an automatic door controller uses minimal memory to track its two states, more complex systems like elevator or appliance controllers require additional memory to manage multiple states corresponding to different operational conditions. This showcases the value of finite automata in designing and understanding the functionality of various everyday devices using basic computational models.

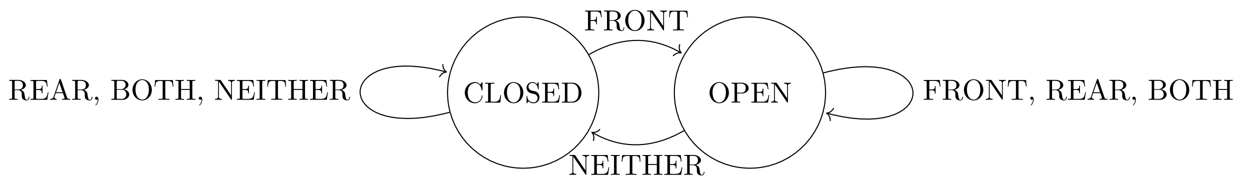


Figure 1: State diagram for an automatic door controller

In the theoretical exploration of finite automata, specific applications are often abstracted to focus on the general mathematical models. Consider the finite automaton named M1, illustrated below in Figure 2. , which features a simplified yet robust setup:

The finite automaton, M1,consists of three states: q_1, q_2, and q_3. State q_1 is the initial state, indicated by an incoming arrow. State q_2 is designated as the accepting state, marked by a double circle. Transitions between states are triggered by input symbols that dictate the path of state changes in a left-to-right sequence of input processing.

When M1 processes an input string such as “1101”, it follows these steps:

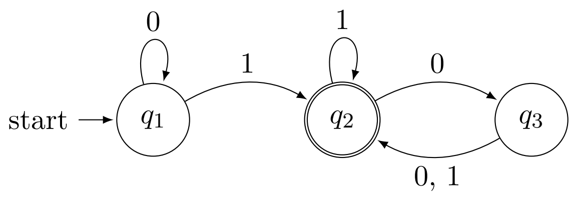


Figure 2: State diagram of finite automaton M1.

1. Starts in state q_1.
2. Transitions to q_2 upon reading the first '1'.
3. Remains in q_2 after reading the second '1'.
4. Moves to q_3 with a '0'.
5. Returns to q_2 with another '1', and accepts the input as it ends in the accepting state.

This automaton efficiently recognizes strings that terminate with the digit '1', such as "1", "01", "11", and patterns extending to any length ending with an even number of '0's following the last '1'. It rejects strings like "0", "10", and "101000", showcasing its capability to discern specific patterns. This functionality highlights the practical utility of finite automata in various computational contexts, particularly in pattern recognition and sequence analysis.

1.3 Formal Definition Finite Automata

After exploring finite automata through intuitive state diagrams, we now transition to a formal definition which is essential for precision and effective communication of concepts.

Definition 1.1. Finite Automaton [1]

It is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set called the *states*,
- Σ is a finite set called the *alphabet*,
- $\delta : Q \times \Sigma \to Q$ is the *transition function*,
- $q_0 \in Q$ is the *start state*,
- $F \subseteq Q$ is the *set of accept states*.

The transition function δ is central to the automaton's operation, determining the state transitions based on the current state and the input symbol. For example, if the automaton is in state x and reads an input symbol 1, and there is a transition labeled 1 to state y, this is formally represented as $\delta(x, 1) = y$. This definition encapsulates the essential elements of a finite automaton, providing a structured framework that supports both theoretical analysis and practical implementation.

Using the formal definition framework, we can precisely describe specific finite automata, such as M1, by specifying its five components. For the finite automaton M1, we showed previously2. We can represent it formally as:

$$M1 = (Q, \Sigma, \delta, q1, F)$$

where:

- $Q = \{q1, q2, q3\}$ – the set of states,
- $\Sigma = \{0, 1\}$ – the input alphabet,
- δ – the transition function, described in the table below,

Table 1:

Current State	**Input 0**	**Input 1**
$q1$	$q1$	$q2$
$q2$	$q3$	$q2$
$q3$	$q2$	$q2$

- $q1$ – the start state,
- $F = \{q2\}$ – the set of accept states.

This configuration details how M1 responds to input symbols by transitioning between states according to the rules specified in the transition table, thereby defining its operational behavior.

If A is the set of all strings that machine M accepts, then A is called the language of machine M and we write $L(M) = A$. We say that M recognizes or accepts A. To avoid confusion, since "accept" can refer to both strings and languages, we prefer to say that M recognizes a language. A machine may accept multiple strings, but it only recognizes one language. If the machine accepts no strings, it still recognizes one language—the empty language $\emptyset$.

Definition 1.2. Language [1]

A language A is a set of strings

So, in the previous example, let A be the set of strings where each string w contains at least one '1', followed by an even number of '0's after the last '1'. Then $L(M1) = A$, meaning $M1$ recognizes A.

1.4 Regular Languages

Finite automata have been described both informally, using state diagrams, and formally as a 5-tuple. While informal descriptions provide intuitive understanding, formal definitions clarify ambiguities and establish precise computational frameworks. We extend this precision to define how finite automata compute and accept strings, thereby defining languages.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is an element of the alphabet Σ. The automaton M accepts the string w if there exists a sequence of states $r_0, r_1, \dots, r_n$ in Q satisfying the following conditions:

1. $r_0 = q_0$ (the computation starts in the initial state),
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for each $i = 0, \dots, n-1$ (the machine transitions from state to state according to the transition function),
3. $r_n \in F$ (the computation ends in an accepted state, indicating that w is accepted).

We say that M recognizes a language A if $A = \{w \mid M \text{ accepts } w\}$. This leads us to a broader definition within the theory of computation [1]:

Definition 1.3

A language is called a **regular language** if it can be recognized by some finite automaton.

Proof. Consider the finite automaton M1 that we showed earlier in Figure 2 and let $w = 1101$. According to the formal computation rules, M1 accepts w because it follows the sequence of states:

$$q_0, q_2, q_2, q_3, q_2$$

which meets the specified conditions for acceptance:

1. The sequence starts at the initial state q_0.
2. Transitions between states are consistent with the transition function δ, particularly:
   - $\delta(q_0, 1) = q_2$
   - $\delta(q_2, 1) = q_2$
   - $\delta(q_2, 0) = q_3$
   - $\delta(q_3, 1) = q_2$
3. The sequence ends in the accept state q_2, which is part of the set of accept states F.

Therefore, M1 recognizes a language $L(M1)$ defined by all strings that can be accepted by following such state transitions, thus characterizing it as a regular language according to its computational capabilities. ■

1.5 Limitations of Finite Automata

While finite automata are instrumental in various computational tasks involving pattern recognition and basic data processing, they are significantly limited by their inability to manage complex memory and perform tasks requiring advanced computational capabilities. Finite automata are constrained to operate with a fixed, limited number of states and lack the ability to use additional memory for processing. This inherent limitation restricts their utility in problems requiring more than simple state transitions, such as those involving counting, recursion, or context-sensitive decisions. Such tasks demand a computational model with the ability to manage dynamic and extensive memory, leading us to consider Turing machines, which will be discussed in detail in section 2.

2 Turing Machines

2.1 Introduction to Turing Machine

Alan Turing developed the Turing machine as a more concrete and powerful computational model in response to Hilbert's Entscheidungsproblem, which highlighted the need for a formal definition of algorithms [2]. Unlike finite automata, which are limited by a fixed number of states and do not modify input strings, Turing machines operate with a finite alphabet on an unbounded tape divided into cells. Each cell can hold a symbol, including a special blank symbol for indicating emptiness. The tape head's ability to read, write, and move across the tape allows for complex computations through direct manipulation of symbol sequences. This capability significantly extends the computational power of Turing machines beyond finite automata, enabling them to address a broader range of problems.

In this case, the Turing machine can be thought of as a device that reads and writes symbols on the tape based on its current state and the symbol it is reading. The finite state machine within the Turing machine embodies the algorithm. The machine follows a set of rules, represented by the finite state machine, which can be described informally as:

- Read the current symbol under the tape head.
- Based on the current state and the read symbol, the finite state machine determines the next action:
  - Write a new symbol in the current cell.
  - Move the tape head left or right.
  - Transition to a new state.
- Repeat the process until a specified condition is met, which might include halting the machine either by accepting or rejecting the input. Note that the machine may also continue indefinitely if the specified condition for halting is never met.

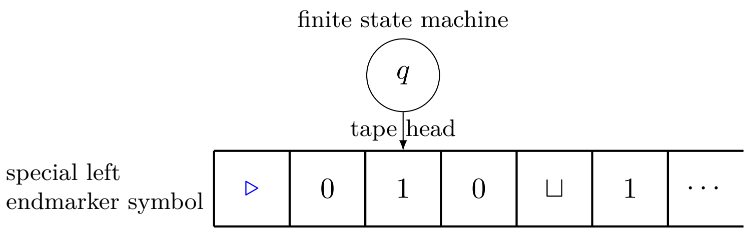


Figure 3: Turing Machine

Definition 2.1. Turing Machine [2]

A Turing machine is formally defined as a quintuple (Q, Σ, s, δ) where:

- Q: A finite set of machine states
- Σ: A finite set of tape symbols including special symbols such as the left end-marker $\triangleright$ and the blank symbol $\sqcup$. Importantly, Σ is disjoint from Q, ensuring no overlap between the sets of tape symbols and machine states.
- Q: is an initial such that $s \in Q$
- $\delta : (Q \times \Sigma) \to (Q \cup \{\text{acc}, \text{rej}\}) \times \Sigma \times \{L, R, S\}$: A transition function satisfies the following: for every state $q \in Q$, there exists a state $q_0 \in Q \cup \{\text{acc}, \text{rej}\}$ for which $\delta(q, \triangleright) = (q_0, \triangleright, R)$. This ensures that the left endmarker is never overwritten and the machine consistently moves to the right when this symbol is encountered.

Example. Consider the Turing machine defined by $M = (Q, \Sigma, s, \delta)$ where:

- $Q = \{s, q_1, q_2\}$ — the set of states,
- $\Sigma = \{\triangleright, \sqcup, 0, 1\}$ — the set of tape symbols that includes:
  - $\triangleright$ — a special left endmarker,
  - $\sqcup$ — a blank symbol representing an empty cell on the tape,
  - $0, 1$ — binary symbols used on the tape,
- s — the initial state, here assumed to be state s,
- δ — the transition function, mapping the current state and current tape symbol to a new state, a symbol to write, and a direction to move the tape head (left L, right R, or stay S).

The transition function δ can be described by the following table:

δ	$\triangleright$	$\sqcup$	0	1
s	$(s, \triangleright, R)$	$(q_1, \sqcup, S)$	$(rej, 0, R)$	$(rej, 1, S)$
q_1	$(rej, \triangleright, R)$	$(q_2, 0, L)$	$(q_1, 1, R)$	$(q_1, 1, R)$
q_2	$(rej, \triangleright, R)$	$(acc, \sqcup, S)$	$(rej, 0, S)$	$(q_2, 1, L)$

2.2 Turing Machine Computation

The computation of a Turing machine on a given input can be described as a series of configurations. Each configuration consists of a sequence of symbols (representing the tape's contents at a specific point in the computation), a number indicating the position of the read/write head, and a state. By utilizing these configurations, we can define what the Turing machine M computes for a given input.

Definition 2.2. Configuration [2]

Turing machine configuration is represented as a triple (q, w, u) where:

- $q \in Q \cup \{\text{acc}, \text{rej}\}$ denotes the current state of the machine.
- w is a non-empty string containing the tape symbols located under and to the left of the tape head. The symbol under the head is the last element of w.
- u is a string (possibly empty) of tape symbols to the right of the tape head, which becomes entirely blank $\sqcup$ beyond a certain point.

A Turing machine begins in the **initial configuration** $(s, \triangleright, u)$.

Simply, a Turing machine configuration consists of the current state, the symbols on the tape under and to the left of the head, and the symbols to the right of the head. It describes the machine's status at any point in its computation. Building on this understanding, the computation of a Turing machine describes how the machine processes input and transitions between different states. It involves a sequence of configurations that represent the machine's status at each step.

Definition 2.3. Turing Machine Computation [2]

The process of computation in a Turing machine M consists of a series (which can either be finite or infinite) of configurations $c_0, c_1, c_2, \ldots$, where:

- $c_0 = (s, \triangleright, u)$ represents the starting configuration,
- $c_i \rightarrow_M c_{i+1}$ The transition from one configuration c_i to the next c_{i+1} follows the rule $c_i \rightarrow_M c_{i+1}$ for each i starting from 0.

The computation can be finite or infinite depending on whether the Turing machine reaches a halting state or continues to operate indefinitely. In simpler terms, this definition lays out how a Turing machine starts from an initial setup and evolves step by step, following specific rules, to process information or perform calculations.

Example. In the previous example of turning machine, we claim that the computation of M, beginning with the configuration $(s, \triangleright, 1^n0)$, will eventually halt in the configuration $(\text{acc}, \triangleright, 1^{n+1}0)$. Thus,

$$\begin{aligned}
(s, \triangleright, 1^n 0) &\to_M (s, \triangleright, 1^n 0) \\
&\to_M (q, \triangleright 1, 1^{n-1} 0) \\
&\quad\vdots \\
&\to_M (q, \triangleright 1^n, 0) \\
&\to_M (q, \triangleright 1^n 0, \epsilon) \\
&\to_M (q, \triangleright 1^{n+1} \triangleright, \epsilon) \\
&\to_M (q', \triangleright 1^{n+1}, 0) \\
&\quad\vdots \\
&\to_M (q', \triangleright 1^{n+1} 0) \\
&\to_M (acc, \triangleright 1^{n+1} 0)
\end{aligned}$$

Definition 2.4. Turing-recognizable [1]

A language L is Turing-recognizable if there exists a Turing machine that accepts all words in L. This means that for every word in the language L, the Turing machine will eventually enter an accepting state.

In summary, Turing machines offer a comprehensive framework for understanding computation, far surpassing the capabilities of finite automata. This allows us to recognize languages and determine how Turing machines process inputs. With this foundation in place, we can now transition to exploring computable functions, where we will investigate the types of problems Turing machines can solve and the limits of their computational power.

3 Computable Functions

3.1 Encoding Numbers on a Turing Machine Tape

It has been established that a register machine can implement the computation of a Turing machine. Conversely, a Turing machine can also implement the computation of a register machine. To understand this fully, we must define how to represent register machine configurations on a tape, as well as how to encode numbers and lists of numbers.

Definition 3.1. [2]

A tape over$\Sigma = \{\triangleright, \sqcup, 0, 1\}$ codes a list of numbers if precisely two cells contain 0 and the only cells containing 1 occur between these.

Such tapes look like:
which corresponds to the list $[n_1, n_2, \ldots, n_k]$.

$$\triangleright \sqcup \cdots \sqcup 0\,\underbrace{1\cdots 1}_{n_1} \sqcup \underbrace{1\cdots 1}_{n_2} \sqcup \cdots \sqcup \underbrace{1\cdots 1}_{n_k}\, 0 \sqcup \cdots$$

3.2 Turing Computable Function

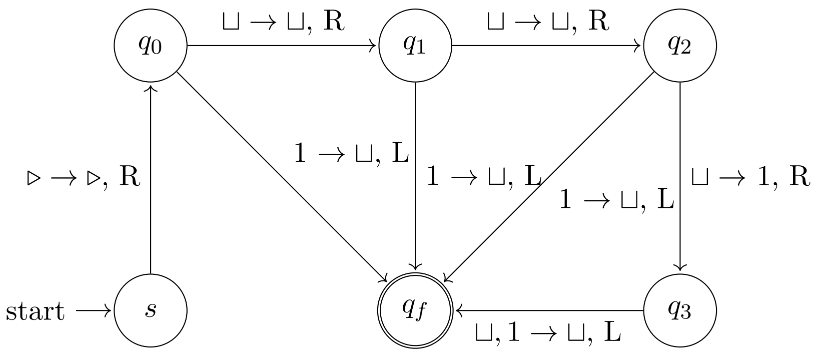
Definition 3.2. [2]

A function $f \in \mathbb{N}^n \to \mathbb{N}$ is Turing computable if and only if there is a Turing machine M with the following property: Starting M from its initial state with the tape head on the left endmarker of a tape coding $[0, x_1, \ldots, x_n]$, M halts if and only if $f(x_1, \ldots, x_n) \downarrow$, and in that case the final tape codes a list (of length ≥ 1) whose first element is y where $f(x_1, \ldots, x_n) = y$.

Definition 3.3. Another definition

A function $f \in \mathbb{N}^n \to \mathbb{N}$ is a **computable function** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

All standard arithmetic operations on integers can be computed by functions. For instance, we can create a machine that accepts the input ⟨m,n⟩ and outputs m+n, the sum of m and n.


$$\begin{tikzcd}
 & q_0 \arrow[r, "\sqcup \to \sqcup, R"] \arrow[dr, "1 \to \sqcup, L"] & q_1 \arrow[r, "\sqcup \to \sqcup, R"] \arrow[d, "1 \to \sqcup, L"] & q_2 \arrow[dl, "1 \to \sqcup, L"] \arrow[d, "\sqcup \to 1, R"] \\
\text{start} \arrow[r] & s \arrow[u, "\triangleright \to \triangleright, R"] & q_f & q_3 \arrow[l, "{\sqcup, 1 \to \sqcup, L}"]
\end{tikzcd}$$

Example.

4 Decidability

Consider the problem of verifying a computer program against a precise specification of its intended functionality, such as ensuring a program correctly sorts a list of numbers. Despite the mathematical precision of both the program and its specification. The general problem of software verification is not solvable by computers.

4.1 Undecidability

Computers often seem so powerful that they can eventually solve any problem. However, unsolvable problems extend beyond the theoretical challenges posited by academicians; they encompass practical issues that people seek to address. Consider the problem of verifying a computer program against a precise specification of its intended functionality, such as ensuring a program correctly sorts a list of numbers. Despite the mathematical precision of both the program and its specification. The general problem of software verification is not solvable by computers.

Now, we show the undecidability of a specific language: the problem of problem of determining whether a Turing machine accepts a given input string. We call it A_{TM}.

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

Theorem 4.1 A_{TM} is undecidable.

It is important to observe that A_{TM} is Turing-recognizable. This means that Turing recognizers *are* more capable than Turing deciders. The restriction of a Turing Machine (TM) to halt on all inputs limits the variety of languages it can recognize. Consider the following Turing machine U that recognizes A_{TM}:

```
$U$ = "On input $\langle M, w\rangle$, where $M$ is a TM and $w$ is a string:
    1. Simulate $M$ on input $w$.
    2. If $M$ enters its accept state, *accept*; if $M$ enters its reject state, *reject*."
```

Note that this machine may loop on input $\langle M, w\rangle$ if M does not halt on w, illustrating why it does not decide A_{TM}. If there were an algorithm to determine the non-halting of M on w, it could *reject*; however, such a determination is not possible, as we will demonstrate.

The Turing machine U serves as a significant example of the *universal Turing machine*, originally proposed by Alan Turing in 1936 [3]. This machine is termed "universal" because it can simulate any other Turing machine using the description of that machine.

To prove the undecidability of A_{TM}, we take a step back into set theory revising few concepts and definitions.

Definition 4.2
A set A is **countable** if either it is finite or if there exists a bijection from the set of natural numbers $\mathbb{N}$ to the set A.

Theorem 4.3 The set of real numbers $\mathbb{R}$ is uncountable.

This theorem was proven by Georg Cantor in 1873 using the *diagonalization* technique he proposed. Additionally, this theorem provides an important argument to show that some languages aren't decidable or even Turing-recognizable, as there are uncountable many languages yet only countably many Turning machines. Since each Turing machine can recognize a single language, then there are more languages than Turing machine, hence some languages can't be recognized by any Turing machine.

Corollary 4.4 ([1]). Some languages are not Turing-recognizable.

Proof. To show that the set of all Turing machines is countable, we first observe that the set of all strings Σ^* is countable for any alphabet Σ. With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, length 1, length 2, and so on.

The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M\rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

To show that the set of all languages is uncountable, we first observe that the set of all infinite binary sequences is uncountable. An infinite binary sequence is an unending sequence of 0s and 1s. Let B be the set of all infinite binary sequences. We can show that B is uncountable by using a proof by diagonalization similar to the one we used in Theorem 4.17 to show that $\mathbb{R}$ is uncountable.

Let L be the set of all languages over alphabet Σ. We show that L is uncountable by giving a correspondence with B, thus showing that the two sets are the same size. Let $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Each language $A \in L$ has a unique sequence in B. The i-th bit of that sequence is a 1 if $s_i \in A$ and is a 0 if $s_i \notin A$, which is called the characteristic sequence of A. For example, if A were the language of all strings starting with a 0 over the alphabet $\{0, 1\}$, its characteristic sequence χ_A would be

$$\begin{aligned}\Sigma^* &= \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}\\ A &= \{0, 00, 01, 000, 001, \ldots\}\\ \chi_A &= 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ \ \ldots\end{aligned}$$

The function $f : L \to B$, where $f(A)$ equals the characteristic sequence of A, is one-to-one and onto, and hence is a correspondence. Therefore, as B is uncountable, L is uncountable as well.

Thus we have shown that the set of all languages cannot be put into a correspondence with the set of all Turing machines. We conclude that some languages are not recognized by any Turing machine. ■

Corollary 4.5. The language $\overline{A_{TM}}$, which represents the complement of the Turing machine language, is not Turing-recognizable.

Proof. It is established that A_{TM}, the set of all Turing machines that halt, is Turing-recognizable. If $\overline{A_{TM}}$ were also Turing-recognizable, then A_{TM} would be a decidable language. However, according to Theorem 4.1, A_{TM} is undecidable, leading to the conclusion that $\overline{A_{TM}}$ cannot be Turing-recognizable. ■

5 Reducibility

A **reduction** is to convert one problem to another, such as a solution to the second problem can also be used to solve the first problem. In our context, this can be used when we know that a certain problem is computationally unsolvable, and then any problem that can be reduced to this problem is also unsolvable (which is easier than proving the unsolvability of the problem from scratch).

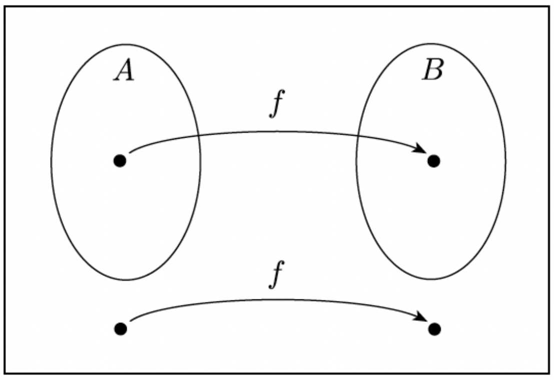
Definition 5.1. [1]

Language A is **mapping reducible** to language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w,

$$w \in A \Leftrightarrow f(w) \in B.$$

The function f is called the **reduction** from A to B.

The following figure illustrates mapping reducibility.



Theorem 5.2 If $A \leq_m B$ and B is decidable, then A is decidable.

Proof. We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B. Thus, M accepts $f(w)$ whenever $w \in A$. Therefore, N works as desired. ■

Corollary 5.3. If $A \leq_m B$ and A is undecidable, then B is undecidable.

Mapping reducibility is crucial in proving the non-recognizability of certain languages through complementation. It is also utilized in demonstrating that some problems are not Turing-recognizable. The following theorem is similar to Theorem 5:

Theorem 5.4 If $A \leq_m B$, and B is Turing-recognizable, then A is also Turing-recognizable.

This proof follows the structure of Theorem 5, with the modification that both M and N are recognizers rather than deciders.

Corollary 5.5. If $A \leq_m B$ and A is not Turing-recognizable, then B cannot be Turing-recognizable either.

This corollary is commonly applied by setting A as the complement of the Turing machine language A_{TM} (denoted as $\overline{A_{TM}}$), which is known from Corollary 4.1 to be not Turing-recognizable. The nature of mapping reducibility ensures that if $A \leq_m B$, then the non-recognizability of A implies the non-recognizability of B.

5.1 Oracle Turing Machines

Consider the languages A_{TM} and $\overline{A_{TM}}$. They appear to be reducible to each other since solving one seemingly provides the solution to the other through simple inversion. However, this is misleading as A_{TM} is Turing-recognizable but $\overline{A_{TM}}$ is not. This discrepancy leads us to a more refined concept of reducibility, *Turing reducibility*, which aligns more closely with our intuitive understanding of problem-solving through reducibility.

Definition 5.6. An Oracle Turing Machine [1]

An **oracle** for a language B is defined as an external mechanism capable of confirming whether a string w belongs to B. An **oracle Turing machine** is an enhanced Turing machine equipped with the capability to consult this oracle. We denote an oracle Turing machine that accesses an oracle for language B as M^B.

Consider the application of an oracle for the language A_{TM}. An Oracle Turing Machine equipped with an oracle for A_{TM} can decide more languages than a standard Turing machine, as it can directly decide A_{TM} and also address the emptiness problem for Turing machines (E_{TM}), denoting the set of all Turing machines with empty language:

$$E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}.$$

This is done through a procedure we'll call $T_{A_{TM}}$.

Procedure $T_{A_{TM}}$

On input $\langle M \rangle$, where M is a Turing machine:

1. Construct a new Turing machine N:
   (a) On any input, run M in parallel on all strings in Σ^*.
   (b) Accept if M accepts any string.
2. Query the oracle whether $\langle N, 0 \rangle \in A_{TM}$.

3. Accept if the oracle says NO, otherwise reject.

If M's language is non-empty, N will accept all inputs, leading the oracle to respond YES, and T_{ATM} will reject. If M's language is empty, T_{ATM} accepts. This illustrates that ETM is decidable relative to ATM.

Definition 5.7. [1]

Language A is **Turing reducible** to language B, written as $A \leq_T B$, if A is decidable relative to B.

This concept is highlighted by showing that E_{TM} is Turing reducible to A_{TM}, enhancing our understanding of reducibility.

Theorem 5.8. I $A \leq_T B$ and B is decidable, then A is also decidable.

Proof. If B is decidable, then substituting the oracle for B with a procedure that decides B allows the oracle Turing machine for A to become a standard Turing machine that decides A. ■

Turing reducibility is a broader framework than mapping reducibility, enabling the use of an oracle Turing machine with an oracle for A_{TM} to tackle complex decision problems. Although these machines are powerful, they cannot solve every language problem, which underscores the limitations of computability.

6 Hypercomputation

In the previous section, we defined a new variation of turning machines which is oracle turning machines, we will define other models for computations that are more capable than the standard turning machines proposed in 1939.

6.1 Infinite Time Turing Machines

An **Infinite Time Turing Machine (ITTM)** is a theoretical model of computation that extends the concept of a traditional Turing machine by allowing computations to run for an infinite amount of time. This machine operates over transfinite time periods, indexed by ordinal numbers.

It consists of the following components:

- **Tape:** An infinite tape, similar to that in a standard Turing machine, capable of storing symbols from a finite alphabet.
- **Head:** A tape head that can read and write symbols on the tape and move left or right.
- **State register:** A register that holds the current state of the machine, including at least one halting state.
- **Transition function:** A function that determines the next state of the machine,

the symbol to write, and the movement of the tape head, based on the current state and the symbol being read.
- **Time structure:** The execution time of the machine is indexed by ordinal numbers, allowing the machine to execute through every ordinal number.

Computation Steps

- **Successor ordinals:** At each successor ordinal ($\alpha + 1$), the machine performs a standard Turing machine step according to its transition function.
- **Limit ordinals:** At each limit ordinal (an ordinal with no immediate predecessor), the machine enters a special configuration, often defined by a limit rule such as taking the liminf or limsup of the configurations at earlier stages.

Halting Condition. The machine halts if it enters a designated halting state during any step of its computation. If this occurs at some ordinal time β, then the output can be read off the tape.

Output. The output of an ITTM can be complex, including sequences determined by behaviors at limit stages, extending beyond the capabilities of classical Turing machines.

Theorem 6.1. Infinite Computability A function is **infinite-time computable** if there exists an ITTM program that, for any input, leads to a halting state wherein the output tape displays the function's value.

Theorem 6.2. Decidability in ITTMs A set is **infinite-time decidable** if its characteristic function can be computed by an ITTM, thereby extending the concept of Turing decidability to the transfinite.

ITTMs not only challenge our conventional understanding of computability and decidability but also enrich the theoretical landscape by integrating advanced concepts from ordinal and cardinal arithmetic. These machines represent a significant step in exploring the computational limits proposed by the Church-Turing thesis.

Theorem 6.3 Every halting computation by an infinite-time Turing machine is countable.

A detailed snapshot of a computation describes fully the machine's state at any specific moment, including the program, the state and position of the head, and all tape contents. It is feasible that such a machine might enter a permanent loop, continuously operating or halting only if the limit of its repeated operations differs from any specific operational state.

Definition 6.4. [4]

A computation is said to repeat itself if identical descriptions occur at two distinct limit-ordinal stages and no transitions from 0 to 1 are observed between these stages. This definition permits transitions from 1 to 0.

Corollary 6.5. An infinite-time computation either halts or indefinitely repeats itself in a countable number of steps.

The primary question about infinite-time Turing machines is their relative computational power compared to standard Turing machines. It is evident that such machines can settle the halting problem, traditionally undecidable, by simply simulating a standard Turing machine's computation transfinetely and halting either upon completion or reaching a repeating state. Therefore, infinite-time Turing machines possess the capability to resolve problems unmanageable by conventional Turing machines.

Theorem 6.6 All arithmetic sets are decidable by an infinite-time Turing machine.

This statement underlines the significantly enhanced computational power of infinite-time Turing machines compared to their traditional counterparts.

References

[1] M. Sipser, "Introduction to the theory of computation (3rd international ed.)," *Cengage Learning*, 2013.

[2] T. K. Andrew Pitts, *Lecture notes in Computation Theory.* University of Cambridge, UK, April 2022.

[3] A. M. Turing *et al.*, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[4] A. Syropoulos, *Hypercomputation: computing beyond the Church-Turing barrier.* Springer Science & Business Media, 2008.