CSE 303: Introduction to the Theory of Computation
(Algorithmic Solvability)

Pramod Ganapathhi
Department of Computer Science
State University of New York at Stony Brook

January 14, 2020
### Problem

- What is an algorithm?
## How do we compute?

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• What is an algorithm?</td>
<td>• An algorithm is an effective/systematic/mechanical method for achieving the desired result for a given problem.</td>
</tr>
<tr>
<td>• What are the properties of an algorithm?</td>
<td></td>
</tr>
</tbody>
</table>
### How do we compute?

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is an algorithm?</td>
<td>An algorithm is an <strong>effective/systematic/mechanical method</strong> for achieving the desired result for a given problem.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
</table>
| What are the properties of an algorithm? | It has a **finite number of instructions**.  
If carried out without error, it produces the **desired result** in a **finite number of steps**.  
It can be carried out by a human with only **paper and pen**.  
It requires **no insight, intuition, or ingenuity**, on the part of the human carrying out the method. |
### Problem

- Are Turing machines powerful enough to model any conceivable algorithm?

### Approach

- To solve this problem, we need to formally define algorithm.
- Before attempting to define algorithm, we need to understand the capabilities and limitations of Turing machines.
What are the types of computational problems?

Types

- **Decision problems:**
  Problems with input \( w \) and output “yes” or “no” answer. (“yes”: \( w \in L \). “no”: \( w \notin L \).)
  
  e.g.: Given a specific chess configuration and it is your turn, can you win the chess game?

- **Function computation:**
  Problems with input \( w \) and output \( f(w) \).
  
  e.g.: Given the Facebook graph, what is the minimum number of people connected between you and your role model?
What are Turing-decidable languages?

### Definitions

- A Turing machine $M$ accepts (or rejects) a given input string $w$ iff the initial configuration yields the accepting (or rejecting) configuration for the given string $w$.
- A Turing machine $M$ decides a language $L \subseteq \Sigma^*$ iff for all strings $w \in \Sigma^*$,

$$
\begin{cases}
M \text{ accepts } w, & \text{ if } w \in L, \\
M \text{ rejects } w, & \text{ if } w \notin L.
\end{cases}
$$

- A language is called **Turing-decidable** or **recursive** iff there exists a TM that decides it.

- Does this mean that a Turing machine that decides a language never enters an infinite loop?
What are Turing-decidable languages?

| $\Sigma^*$ | $\checkmark, \times$ | $N$ | IsPrime? | $\checkmark, \times$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1 \in L$</td>
<td>$\checkmark$</td>
<td>2</td>
<td>$\checkmark$</td>
<td></td>
</tr>
<tr>
<td>$w_2 \notin L$</td>
<td>$\times$</td>
<td>3</td>
<td>$\checkmark$</td>
<td></td>
</tr>
<tr>
<td>$w_3 \notin L$</td>
<td>$\times$</td>
<td>4</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>$w_4 \in L$</td>
<td>$\checkmark$</td>
<td>5</td>
<td>$\checkmark$</td>
<td></td>
</tr>
<tr>
<td>$w_5 \in L$</td>
<td>$\checkmark$</td>
<td>6</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td></td>
</tr>
<tr>
<td>$w_n \notin L$</td>
<td>$\times$</td>
<td>97</td>
<td>$\checkmark$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td></td>
</tr>
</tbody>
</table>

Note: $\checkmark$ indicates membership in the language $L$, and $\times$ indicates non-membership.
What are Turing-decidable languages?

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>● All regular languages</td>
</tr>
<tr>
<td>● All context-free languages</td>
</tr>
<tr>
<td>● Several non-context-free languages such as:</td>
</tr>
<tr>
<td>[ L = {a^n b^n c^n \mid n \geq 0} ]</td>
</tr>
<tr>
<td>[ L = {w \mid w = w^R \text{ and } w \in {a, b}^*} ]</td>
</tr>
<tr>
<td>[ L = {ww \mid w \in {a, b}^*} ]</td>
</tr>
<tr>
<td>[ L = {p \mid p \text{ is a prime}} ]</td>
</tr>
</tbody>
</table>

● What languages are Turing-undecidable languages?
What are Turing-computable functions?

<table>
<thead>
<tr>
<th>Definitions</th>
</tr>
</thead>
</table>
| • The **output** of a TM for input string \( w \) is string \( w' \) iff  <br> \[
(q_0, \triangleright w) \vdash^* (q_{\text{acc}}, \triangleright w')
\]  <br> • Let function \( f : \Sigma^* \rightarrow \Sigma^* \)  <br> • A Turing machine **computes a function** \( f \) iff  <br> for all strings \( w \in \Sigma^* \),  <br> \[
M \text{ outputs } f(w), \text{ i.e., } \ <br>(q_0, \triangleright w) \vdash^* (q_{\text{acc}}, \triangleright f(w))
\]  <br> • A function \( f : \Sigma^* \rightarrow \Sigma^* \) is called **Turing-computable** or **recursive** iff there exists a TM that computes it.  <br> • Why do we use the term recursive to describe both the languages decided by and the functions computed by Turing machines? |
What are Turing-computable functions?

<table>
<thead>
<tr>
<th>$\Sigma^*$</th>
<th>$f$</th>
<th>$\Sigma^*$</th>
<th>$\mathbb{N}$</th>
<th>Cube</th>
<th>$\mathbb{N}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>$\rightarrow$</td>
<td>$f(w_1)$</td>
<td>1</td>
<td>$\rightarrow$</td>
<td>1</td>
</tr>
<tr>
<td>$w_2$</td>
<td>$\rightarrow$</td>
<td>$f(w_2)$</td>
<td>2</td>
<td>$\rightarrow$</td>
<td>8</td>
</tr>
<tr>
<td>$w_3$</td>
<td>$\rightarrow$</td>
<td>$f(w_3)$</td>
<td>3</td>
<td>$\rightarrow$</td>
<td>27</td>
</tr>
<tr>
<td>$w_4$</td>
<td>$\rightarrow$</td>
<td>$f(w_4)$</td>
<td>4</td>
<td>$\rightarrow$</td>
<td>64</td>
</tr>
<tr>
<td>$w_5$</td>
<td>$\rightarrow$</td>
<td>$f(w_5)$</td>
<td>5</td>
<td>$\rightarrow$</td>
<td>125</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$w_n$</td>
<td>$\rightarrow$</td>
<td>$f(w_n)$</td>
<td>10</td>
<td>$\rightarrow$</td>
<td>1000</td>
</tr>
</tbody>
</table>
What are Turing-semidecidable languages?

**Definitions**

- A Turing machine $M$ semidesides a language $L \in \Sigma^*$ iff for all strings $w \in \Sigma^*$,
  
  \[
  \begin{cases}
  M \text{ accepts } w, & \text{if } w \in L, \\
  M \text{ rejects } w \text{ or runs forever,} & \text{if } w \notin L.
  \end{cases}
  \]

- A language is called Turing-semidecidable or recursively enumerable iff there exists a TM that semidesides it.

- Does this mean that a Turing machine that semidesides a language can enter an infinite loop?
What are Turing-semidecidable languages?

<table>
<thead>
<tr>
<th>Programs</th>
<th>Correctness?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1 \in L$</td>
<td>$\checkmark$, run forever</td>
</tr>
<tr>
<td>$w_2 \notin L$</td>
<td>$\times$, run forever</td>
</tr>
<tr>
<td>$w_3 \notin L$</td>
<td>$\checkmark$, run forever</td>
</tr>
<tr>
<td>$w_4 \in L$</td>
<td>$\checkmark$, run forever</td>
</tr>
<tr>
<td>$w_5 \in L$</td>
<td>$\times$, run forever</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$w_n \notin L$</td>
<td>$\checkmark$, run forever</td>
</tr>
</tbody>
</table>

| $P_1$ | $\checkmark$, run forever |
| $P_2$ | $\times$, run forever |
| $P_3$ | run forever |
| $P_4$ | $\checkmark$, run forever |
| $P_5$ | $\times$, run forever |
| $\vdots$ | $\vdots$ |
| $P_n$ | run forever |
The three types of computational problems solved by TM’s are:
- Turing-decidable languages
- Turing-computable functions
- Turing-semidecidable languages

Can we formalize the notion of an algorithm using the computation ideas described above?
## What might be algorithms?

<table>
<thead>
<tr>
<th>Properties of algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuitively, an algorithm has the following properties:</td>
</tr>
<tr>
<td>1. It is a sequence of steps that gives the correct result to</td>
</tr>
<tr>
<td>a computational problem.</td>
</tr>
<tr>
<td>2. It should work for all input instances from a given domain.</td>
</tr>
</tbody>
</table>

Describing algorithms

The properties imply that an algorithm always halts or an algorithm makes a total function.

### Type of computation

- **Always halt?** / **Total function?**
- **TM’s for decidable languages**
- **TM’s for computable functions**
- **TM’s for semidecidable languages**

A TM for a Turing-decidable language or a Turing-computable function formalizes the intuitive notion of an algorithm.
What might be algorithms?

Properties of algorithms

Intuitively, an algorithm has the following properties:
1. It is a sequence of steps that gives the **correct result** to a computational problem.
2. It should work for **all input instances** from a given domain.

Describing algorithms

- The properties imply that an algorithm always halts or an algorithm makes a total function.

<table>
<thead>
<tr>
<th>Type of computation</th>
<th>Always halt? / Total function?</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM’s for decidable languages</td>
<td>✓</td>
</tr>
<tr>
<td>TM’s for computable functions</td>
<td>✓</td>
</tr>
<tr>
<td>TM’s for semidecidable languages</td>
<td>✗</td>
</tr>
</tbody>
</table>

- A TM for a Turing-decidable language or a Turing-computable function formalizes the intuitive notion of an algorithm.
What are algorithms?

**Definitions**

- **Algorithm:**
  Turing machine for a Turing-decidable language or Turing machine for a Turing-computable function.

- **Algorithmic solvability:**
  Turing-decidability or Turing-computability

- **Algorithmic unsolvability:**
  Turing-undecidability or Turing-noncomputability i.e.,
  Turing-semidecidability and not Turing-semidecidability
Examples of algorithms?

Examples

- Thousands of algorithms taught in the courses such as algorithms, data structures, programming, operating systems, networking, security, operations research, computer graphics, computer vision, etc.
- The notion of algorithm is extended to include randomized algorithms, parallel algorithms, distributed algorithms, machine learning (or self-learning) algorithms, self-improving algorithms, quantum algorithms, etc.

- Are Turing machines powerful enough to model any conceivable algorithm?
What is Church-Turing thesis?

Hypothesis

- Any algorithm can be executed by a Turing machine.
- Anything that can be computed can be computed by a Turing machine.
- A function on the natural numbers can be calculated by an effective method iff it is computable by a Turing machine.
- Turing machines can do anything that could be described as “purely mechanical”.

<table>
<thead>
<tr>
<th>Some questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Why do we call Turing-decidable and Turing-semidecidable languages as <strong>recursive</strong> and <strong>recursively enumerable</strong>, respectively?</td>
</tr>
<tr>
<td>• What is the <strong>intuition</strong> behind algorithmic unsolvability?</td>
</tr>
<tr>
<td>• What is the <strong>relationship</strong> between recursive and recursively enumerable languages?</td>
</tr>
<tr>
<td>• What are the <strong>techniques to prove algorithmic unsolvability</strong>?</td>
</tr>
<tr>
<td>• What are some <strong>real-world</strong> problems that <strong>cannot be solved</strong> by human minds or real computers (from past, present, future)?</td>
</tr>
</tbody>
</table>
Chomsky hierarchy

Regular languages
Context-free languages
Context-sensitive languages
Turing-decidable languages
Turing-semidecidable languages
Not Turing-semidecidable languages

Algorithmically solvable (finite time)
Algorithmically unsolvable (infinite time)
Properties

- If $L$ is a Turing-decidable language, then $\overline{L}$ is a Turing-decidable language, too.
- If $L$ is both Turing-semidecidable and Turing-undecidable (algorithmically unsolvable), then $\overline{L}$ is not Turing-semidecidable.
How can we prove algorithmic unsolvability?

**Problem**

- How can we prove that there are some computational problems that are algorithmically unsolvable?

**Directions**

A. **Show that there are languages that are Turing-semidecidable but not Turing-decidable:**

B. **Show that there are languages that are not Turing semidecidable:**

**Approach**

<table>
<thead>
<tr>
<th>Approach</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show hypothetical examples</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Prove that the set of decision problems/languages is bigger than the set of computer programs/TM’s using uncountability</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Prove that the set of decision problems/languages is bigger than the set of computer programs/TM’s using diagonalization</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>Show real-world practical examples</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
**Problem**

- Let’s construct three non-halting Turing machines for \( \Sigma = \{a\} \) and \( \Gamma = \Sigma \cup \{\rhd, \square\} \) with the following transition tables. Explain the working of these non-halting TM’s \( M_1 \), \( M_2 \), and \( M_3 \).

<table>
<thead>
<tr>
<th>Current state</th>
<th>Current symbol (( \Gamma ))</th>
<th>( \Gamma )</th>
<th>( a )</th>
<th>( \square )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( (q_0, \rhd) )</td>
<td>( (q_0, \rightarrow) )</td>
<td>( (q_0, \rightarrow) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current state</th>
<th>Current symbol (( \Gamma ))</th>
<th>( \Gamma )</th>
<th>( a )</th>
<th>( \square )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( (q_0, \rhd) )</td>
<td>( (q_0, a) )</td>
<td>( (q_0, \square) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current state</th>
<th>Current symbol (( \Gamma ))</th>
<th>( \Gamma )</th>
<th>( a )</th>
<th>( \square )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( (q_0, \rhd) )</td>
<td>( (q_0, \leftarrow) )</td>
<td>( (q_0, \leftarrow) )</td>
<td></td>
</tr>
</tbody>
</table>
Simple Turing machines that run forever

Solution for $M_1$

$$(\{\triangleright, a, \square\}, \rightarrow)$$

start $\rightarrow q_0$

<table>
<thead>
<tr>
<th>Time</th>
<th>State</th>
<th>Tape</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q_0$</td>
<td>$\triangleright$ a a a □ □ ⋯</td>
</tr>
<tr>
<td>1</td>
<td>$q_0$</td>
<td>$\triangleright$ a a a □ □ ⋯</td>
</tr>
<tr>
<td>2</td>
<td>$q_0$</td>
<td>$\triangleright$ a a a □ □ ⋯</td>
</tr>
<tr>
<td>3</td>
<td>$q_0$</td>
<td>$\triangleright$ a a a □ □ ⋯</td>
</tr>
<tr>
<td>4</td>
<td>$q_0$</td>
<td>$\triangleright$ a a a □ □ ⋯</td>
</tr>
<tr>
<td>5</td>
<td>$q_0$</td>
<td>$\triangleright$ a a a □ □ ⋯</td>
</tr>
</tbody>
</table>

- The TM’s tape head keeps moving right on the tape that has an infinite amount of memory.
- The TM never halts for any input string.
Simple Turing machines that run forever

Solution for $M_2$

$(\triangleright, \rightarrow), (a, a), (\square, \square)$

![Diagram of Turing machine]

<table>
<thead>
<tr>
<th>Time</th>
<th>State</th>
<th>Tape</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q_0$</td>
<td>△ $a$ $a$ $a$ $\square$ $\square$ $\cdots$</td>
</tr>
<tr>
<td>1</td>
<td>$q_0$</td>
<td>△ $a$ $a$ $a$ $\square$ $\square$ $\cdots$</td>
</tr>
<tr>
<td>2</td>
<td>$q_0$</td>
<td>△ $a$ $a$ $a$ $\square$ $\square$ $\cdots$</td>
</tr>
<tr>
<td>3</td>
<td>$q_0$</td>
<td>△ $a$ $a$ $a$ $\square$ $\square$ $\cdots$</td>
</tr>
<tr>
<td>4</td>
<td>$q_0$</td>
<td>△ $a$ $a$ $a$ $\square$ $\square$ $\cdots$</td>
</tr>
<tr>
<td>5</td>
<td>$q_0$</td>
<td>△ $a$ $a$ $a$ $\square$ $\square$ $\cdots$</td>
</tr>
</tbody>
</table>

- The TM’s tape head does not move, replaces the first character by itself, and stays in the same state.
- The TM never halts for any input string.
Simple Turing machines that run forever

Solution for \( M_3 \)

\[(\triangleright, \rightarrow), (a, \leftarrow), (\Box, \leftarrow)\]

\[\text{start} \rightarrow q_0\]

<table>
<thead>
<tr>
<th>Time</th>
<th>State</th>
<th>Tape</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( q_0 )</td>
<td>( \triangleright a a a \Box \Box \cdots )</td>
</tr>
<tr>
<td>1</td>
<td>( q_0 )</td>
<td>( \triangleright a a a \Box \Box \cdots )</td>
</tr>
<tr>
<td>2</td>
<td>( q_0 )</td>
<td>( \triangleright a a a \Box \Box \cdots )</td>
</tr>
<tr>
<td>3</td>
<td>( q_0 )</td>
<td>( \triangleright a a a \Box \Box \cdots )</td>
</tr>
<tr>
<td>4</td>
<td>( q_0 )</td>
<td>( \triangleright a a a \Box \Box \cdots )</td>
</tr>
<tr>
<td>5</td>
<td>( q_0 )</td>
<td>( \triangleright a a a \Box \Box \cdots )</td>
</tr>
</tbody>
</table>

- The TM’s tape head oscillates between the left end symbol and the first character.
- The TM never halts for any input string.
Most problems are algorithmically unsolvable

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Prove that the set of all decision problems or languages is bigger than the set of Turing machines or computer programs using countability/uncountability.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Prove that the set of decision problems is uncountable.</td>
</tr>
<tr>
<td>2. Prove that the number of Turing machines is countable.</td>
</tr>
</tbody>
</table>

This proves that most decision problems or languages are not Turing-semidecidable.
Part 1. Prove that the set of decision problems is uncountable.

- A decision problem can be represented as a number in $[0, 1]$.
- E.g.: The function below represents $0.0110001 \ldots$.

<table>
<thead>
<tr>
<th>Strings</th>
<th>${0, 1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>$0$</td>
</tr>
<tr>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$00$</td>
<td>$0$</td>
</tr>
<tr>
<td>$01$</td>
<td>$0$</td>
</tr>
<tr>
<td>$10$</td>
<td>$0$</td>
</tr>
<tr>
<td>$11$</td>
<td>$1$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>

- Set of all decision problems (or functions $\Sigma^* \rightarrow \{0, 1\}$) can be represented by the set of all real problems in $[0, 1]$.
- The set of all real numbers in $[0, 1]$ is uncountable.
- Hence, the set of all decision problems is uncountable.
### Solution (continued)

**Part 2. Prove that the set of all Turing machines is countable.**

- A TM can be represented as a finite string.
- A finite string in ASCII can be represented as a binary string.
- The set of all TM’s represents the set of all binary strings.
- The set of all binary strings is countable.
- Hence, the set of all TM’s is countable.
<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Prove that the set of all decision problems or languages is bigger than the set of Turing machines or computer programs using diagonalization.</td>
</tr>
</tbody>
</table>
Most problems are algorithmically unsolvable

Problem

- Prove that the set of all decision problems or languages is bigger than the set of Turing machines or computer programs using diagonalization.

Solution

- Suppose $M_1, M_2, M_3, \ldots$ are the TM’s.
- Suppose $w_1, w_2, w_3, \ldots$ are strings in $\Sigma^*$.
- Construct a table with TM’s as rows and strings as columns.

<table>
<thead>
<tr>
<th>TM</th>
<th>Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$w_1$</td>
</tr>
<tr>
<td>$M_1$</td>
<td>1</td>
</tr>
<tr>
<td>$M_2$</td>
<td>0</td>
</tr>
<tr>
<td>$M_3$</td>
<td>0</td>
</tr>
<tr>
<td>$M_4$</td>
<td>1</td>
</tr>
<tr>
<td>$M_5$</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Solution (continued)

- Construct a TM that accepts language
  \[ L_d = \{ w_i \mid w_i \notin L(M_i) \} \] i.e., \( L_d = d_1 d_2 d_3 \ldots \), where
  \[ d_i = \begin{cases} 
  1 & \text{if } \text{table}_{ii} = 0, \\
  0 & \text{if } \text{table}_{ii} = 1. 
  \end{cases} \]
- For the example below, \( L_d = 01001 \ldots \)

<table>
<thead>
<tr>
<th>TM</th>
<th>Strings</th>
</tr>
</thead>
</table>
| \( M_1 \) | \[ \begin{array}{cccccc} 
  w_1 & w_2 & w_3 & w_4 & w_5 & \ldots \\
  1 & 0 & 0 & 1 & 0 & \ldots 
  \end{array} \] |
| \( M_2 \) | \[ \begin{array}{cccccc} 
  w_1 & w_2 & w_3 & w_4 & w_5 & \ldots \\
  0 & 0 & 1 & 0 & 0 & \ldots 
  \end{array} \] |
| \( M_3 \) | \[ \begin{array}{cccccc} 
  w_1 & w_2 & w_3 & w_4 & w_5 & \ldots \\
  0 & 1 & 1 & 1 & 1 & \ldots 
  \end{array} \] |
| \( M_4 \) | \[ \begin{array}{cccccc} 
  w_1 & w_2 & w_3 & w_4 & w_5 & \ldots \\
  1 & 1 & 0 & 1 & 0 & \ldots 
  \end{array} \] |
| \( M_5 \) | \[ \begin{array}{cccccc} 
  w_1 & w_2 & w_3 & w_4 & w_5 & \ldots \\
  0 & 1 & 0 & 0 & 0 & \ldots 
  \end{array} \] |
| \( \vdots \) | \[ \begin{array}{cccccc} 
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots 
  \end{array} \] |
| \( M_d \) | \[ \begin{array}{cccccc} 
  w_1 & w_2 & w_3 & w_4 & w_5 & \ldots \\
  0 & 1 & 0 & 0 & 1 & \ldots 
  \end{array} \] |
Most problems are algorithmically unsolvable

Solution (continued)

Proof by contradiction.

- Suppose $L_d$ is Turing-semidecidable. Then there exists TM $M_k$ such that $L_d = L(M_k)$.
- Case 1. $M_k$ accepts $w_k$.
  \[ \implies w_k \not\in L_d \quad (\because \text{defn. of } L_d) \]
  \[ \implies w_k \not\in L(M_k) \quad (\because L_d = L(M_k)) \]
  \[ \implies M_k \text{ does not accept } w_k \quad (\because \text{defn. of } L(M_k)) \]
- Case 2. $M_k$ does not accept $w_k$.
  \[ \implies w_k \in L_d \quad (\because \text{defn. of } L_d) \]
  \[ \implies w_k \in L(M_k) \quad (\because L_d = L(M_k)) \]
  \[ \implies M_k \text{ accepts } w_k \quad (\because \text{defn. of } L(M_k)) \]
- Contradiction! Hence, $L_d$ is not Turing-semidecidable.

There is a decision problem or language that is not Turing-semidecidable.
Simulate program is algorithmically impossible

Problem

- Prove that it is impossible to design an algorithm to simulate the working of a given computer program on a given input string.
## Problem

- Prove that it is impossible to design an algorithm to simulate the working of a given computer program on a given input string.

## Solution

**Language**

\[ Language = \{ \langle M, w \rangle \mid \text{TM } M \text{ accepts input string } w \} \]

Let's call the hypothetical method as `SIMULATE`.

1. Prove that `SIMULATE` is Turing-semidecidable.
2. Prove that `SIMULATE` is algorithmically impossible.
Part 1. Prove that `SIMULATE` is Turing-semidecidable.

- Consider the following generic procedure.

<table>
<thead>
<tr>
<th><code>SIMULATE(⟨M, w⟩)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Simulate TM <code>M</code> on input string <code>w</code></td>
</tr>
<tr>
<td>2. if <code>M</code> accepts <code>w</code> then</td>
</tr>
<tr>
<td>3. accept</td>
</tr>
<tr>
<td>4. elseif <code>M</code> rejects <code>w</code> then</td>
</tr>
<tr>
<td>5. reject</td>
</tr>
</tbody>
</table>

- Case 1: If `M` accepts `w`, then `SIMULATE` accepts.
- Case 2: If `M` rejects `w`, then `SIMULATE` rejects.
- Case 3: If `M` runs forever on `w`, then `SIMULATE` runs forever.

- So, `SIMULATE` is Turing-semidecidable.
Part 2. Prove that \textsc{Simulate} is algorithmically impossible.

Proof by contradiction.

1. Let’s assume that \textsc{Simulate} is algorithmically possible i.e., \textsc{Simulate} always halts giving a correct answer.
2. Then, we construct the \textsc{Paradox} algorithm as follows.

\textsc{Paradox}(\langle M \rangle)

1. \texttt{result} \leftarrow \textsc{Simulate}(\langle M, \langle M \rangle \rangle)
2. if \texttt{result} = \texttt{accept} then reject
3. elseif \texttt{result} = \texttt{reject} then accept
Part 2. Prove that \textsc{Simulate} is algorithmically impossible.

\texttt{Paradox}(\langle M \rangle)

\begin{itemize}
  \item \textbf{Input}: Source code of a computer program
  \item \textbf{Output}: Accept or reject
  \item \textbf{Require}: Invoke \texttt{Paradox}(\langle \texttt{Paradox} \rangle)
  \begin{enumerate}
    \item \texttt{result} ← \texttt{Simulate}(\langle M, \langle M \rangle \rangle)
    \item if \texttt{result} = accept then reject
    \item elseif \texttt{result} = reject then accept
  \end{enumerate}
\end{itemize}

\begin{itemize}
  \item Case 1. \texttt{Paradox} accepts \langle \texttt{Paradox} \rangle
    \begin{itemize}
      \item \implies \texttt{Simulate} rejects \langle \texttt{Paradox}, \langle \texttt{Paradox} \rangle \rangle
      \item \implies \texttt{Paradox} rejects \langle \texttt{Paradox} \rangle.
    \end{itemize}
  \item Case 2. \texttt{Paradox} rejects \langle \texttt{Paradox} \rangle
    \begin{itemize}
      \item \implies \texttt{Simulate} accepts \langle \texttt{Paradox}, \langle \texttt{Paradox} \rangle \rangle
      \item \implies \texttt{Paradox} accepts \langle \texttt{Paradox} \rangle.
    \end{itemize}
  \item Contradiction! Hence, \textsc{Simulate} is algorithmically impossible.
\end{itemize}
Problem

- Prove that it is impossible to design an algorithm to check if a given computer program halts on a given input string.
## Problem

- Prove that it is impossible to design an algorithm to check if a given computer program halts on a given input string.

## Solution

**Language**

\[ \{ \langle M, w \rangle \mid \text{TM } M \text{ halts on input string } w \} \]

Let’s call the hypothetical method as **HALT**.

1. Prove that **HALT** is Turing-semidecidable.
2. Prove that **HALT** is algorithmically impossible.
Solution (continued)

Part 1. Prove that \( \text{HALT} \) is Turing-semidecidable.

- Consider the following generic procedure.

\[
\begin{align*}
\text{HALT}(\langle M, w \rangle) & \\
1. & \text{Simulate TM } M \text{ on input string } w \\
2. & \text{if } M \text{ accepts } w \text{ or } M \text{ rejects } w \text{ then} \\
3. & \quad \text{accept} \\
4. & \text{else if } M \text{ runs forever then} \\
5. & \quad \text{reject}
\end{align*}
\]

- Case 1: If \( M \) accepts \( w \), then \( \text{HALT} \) accepts.
- Case 2: If \( M \) rejects \( w \), then \( \text{HALT} \) accepts.
- Case 3: If \( M \) runs forever on \( w \), then \( \text{HALT} \) runs forever.

- So, \( \text{HALT} \) is Turing-semidecidable.
Part 2. Prove that $\text{Halt}$ is algorithmically impossible.
Proof by contradiction.
- Let’s assume that $\text{Halt}$ is algorithmically possible i.e., $\text{Halt}$ always halts giving a correct answer.
- Then, we construct the $\text{Paradox}$ algorithm as follows.

$\text{Paradox}(\langle M \rangle)$

1. $\text{result} \leftarrow \text{Halt}(\langle M, \langle M \rangle \rangle)$
2. if $\text{result} = \text{accept}$ then run forever
3. elseif $\text{result} = \text{reject}$ then accept
Part 2. Prove that \textsc{Halt} is algorithmically impossible.

\begin{itemize}
\item Case 1. \textsc{Paradox} accepts \langle\textsc{Paradox}\rangle
\quad \implies \textsc{Halt} rejects \langle\textsc{Paradox}, \langle\textsc{Paradox}\rangle\rangle
\quad \implies \textsc{Paradox} runs forever on \langle\textsc{Paradox}\rangle.
\item Case 2. \textsc{Paradox} runs forever on \langle\textsc{Paradox}\rangle
\quad \implies \textsc{Halt} accepts \langle\textsc{Paradox}, \langle\textsc{Paradox}\rangle\rangle
\quad \implies \textsc{Paradox} accepts \langle\textsc{Paradox}\rangle.
\item Contradiction! Hence, \textsc{Halt} is algorithmically impossible.
\end{itemize}
What is reduction?

**Definition**

- Given two languages $L_{\text{old}}, L_{\text{new}} \in \Sigma^*$, we say that $L_{\text{old}}$ **reduces to** $L_{\text{new}}$, meaning $L_{\text{new}}$ is at least as hard as $L_{\text{old}}$, denoted as $L_{\text{old}} \leq_m L_{\text{new}}$ if there exists a **computable function** $f$ such that for all $x \in \Sigma^*$

  $$x \in L_{\text{old}} \iff f(x) \in L_{\text{new}}$$

```
Σ*       f       Σ*
```

```
L_{\text{old}}
```

```
L_{\text{old}}^c
```

```
L_{\text{new}}
```

```
L_{\text{new}}^c
```
What is reduction?

### Properties

- **Notation.** In $L_{\text{old}} \leq_m L_{\text{new}}$, the ‘$m$’ letter in $\leq_m$ represents many-to-one function.

- **Meaning.** If $L_{\text{old}} \leq_m L_{\text{new}}$, then $L_{\text{new}}$ is at least as hard as $L_{\text{old}}$.

- **Intuition.** If $L_{\text{old}} \leq_m L_{\text{new}}$, then the reduction should turn:
  - Instance of $L_{\text{old}}$ with yes to instance of $L_{\text{new}}$ with yes.
  - Instance of $L_{\text{old}}$ with no to instance of $L_{\text{new}}$ with no.

- **Consequences.** If $L_{\text{old}} \leq_m L_{\text{new}}$, then
  - If $L_{\text{old}}$ is undecidable, then so is $L_{\text{new}}$.
  - If $L_{\text{old}}$ is not Turing-semidecidable, then so is $L_{\text{new}}$.
  - If $L_{\text{new}}$ is decidable, then so is $L_{\text{old}}$. 
Problem

- Prove that it is impossible to design an algorithm to check if a given computer program halts on a given input string, using reduction.
Problem

• Prove that it is impossible to design an algorithm to check if a given computer program halts on a given input string, using reduction.

Solution

• \( L_{\text{sim}} = \{ \langle M, w \rangle \mid \text{TM } M \text{ accepts input string } w \} \)
• \( L_{\text{halt}} = \{ \langle M, w \rangle \mid \text{TM } M \text{ halts on input string } w \} \)
• Proof by contradiction and proof by reduction. Let’s call the hypothetical method as \( \text{HALT} \).
  We show that if \( \text{HALT} \) is algorithmically possible, then \( \text{SIMULATE} \) is algorithmically possible, too.
Halt program is algorithmically impossible

Solution (continued)

Prove that \textsc{Halt} is algorithmically impossible.

• Let’s assume that \textsc{Halt} is algorithmically possible. Then, we construct the \textsc{Simulate} algorithm as follows.

\begin{verbatim}
\textsc{Simulate}(\langle M, w \rangle)
1. \text{result} \leftarrow \text{Halt}(\langle M, w \rangle)
2. \textbf{if} \ \text{result} = \text{reject} \ \textbf{then} \ \text{reject} \quad \triangleright \ M \ \text{runs forever on} \ w
3. \textbf{elseif} \ \text{result} = \text{accept} \ \textbf{then}
4. \ \text{Simulate} \ M \ \text{on} \ w
5. \ \textbf{if} \ M \ \text{accepts} \ w \ \textbf{then} \ \text{accept} \quad \triangleright \ M \ \text{accepts} \ w
6. \ \textbf{elseif} \ M \ \text{rejects} \ w \ \textbf{then} \ \text{reject} \quad \triangleright \ M \ \text{rejects} \ w
\end{verbatim}

• If \textsc{Halt} is an algorithm, then \textsc{Simulate} is an algorithm too, which terminates in all cases.

• We know that \textsc{Simulate} is algorithmically impossible. Hence, \textsc{Halt} is algorithmically impossible, too.