### Problem
- What is an algorithm?
### Problem
- What is an algorithm?

### Solution
- An algorithm is an **effective/systematic/mechanical method** for achieving the desired result for a given problem.

### Problem
- What are the properties of 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 <strong>effective/systematic/mechanical method</strong> for achieving the desired result for a given problem.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem</td>
<td>Solution</td>
</tr>
<tr>
<td>• What are the properties of an algorithm?</td>
<td>• It has a <strong>finite number of instructions</strong>.</td>
</tr>
<tr>
<td></td>
<td>• If carried out without error, it produces the <strong>desired result</strong> in a finite number of steps.</td>
</tr>
<tr>
<td></td>
<td>• It can be carried out by a human with only <strong>paper and pen</strong>.</td>
</tr>
</tbody>
</table>
|                                              | • It requires **no insight, intuition, or ingenuity**, on the part of the human carrying out the method.
One big question

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Are Turing machines powerful enough to model any conceivable algorithm?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>• To solve this problem, we need to formally define algorithm.</td>
</tr>
<tr>
<td>• Before attempting to define algorithm, we need to understand the capabilities and limitations of Turing machines.</td>
</tr>
</tbody>
</table>
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”: $w \not\in L$.)
  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 $L \in \Sigma^*$ iff for all strings $w \in \Sigma^*$,
  
  \[
  \begin{cases} 
  M \text{ accepts } w, & \text{if } w \in L, \\
  M \text{ rejects } w, & \text{if } w \notin L. 
  \end{cases}
  \]

- 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?

<table>
<thead>
<tr>
<th>$\Sigma^*$</th>
<th>$\checkmark,\times$</th>
<th>$\mathbb{N}$</th>
<th>IsPrime?</th>
<th>$\checkmark,\times$</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>$\vdots$</td>
</tr>
<tr>
<td>$w_n \notin L$</td>
<td>$\times$</td>
<td>97</td>
<td>$\checkmark$</td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>
What are Turing-decidable languages?

Examples

- All regular languages
- All context-free languages
- Several non-context-free languages such as:
  \[ L = \{ a^n b^n c^n \mid n \geq 0 \} \]
  \[ L = \{ w \mid w = w^R \text{ and } w \in \{a, b\}^* \} \]
  \[ L = \{ww \mid w \in \{a, b\}^* \} \]
  \[ L = \{p \mid p \text{ is a prime} \} \]

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

**Definitions**

- The **output** of a TM for input string $w$ is string $w'$ iff
  $$ (q_0, w) \vdash^* (q_{\text{acc}}, w') $$

- Let function $f : \Sigma^* \rightarrow \Sigma^*$

- A Turing machine **computes a function** $f$ iff for all strings $w \in \Sigma^*$,
  $$ M \text{ outputs } f(w), \text{ i.e., } (q_0, w) \vdash^* (q_{\text{acc}}, f(w)) $$

- A function $f : \Sigma^* \rightarrow \Sigma^*$ is called **Turing-computable** or **recursive** iff there exists a TM that computes it.

- 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>$f(w_1)$</td>
<td>$1$</td>
<td>$\to$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>$w_2$</td>
<td>$f(w_2)$</td>
<td>$2$</td>
<td>$\to$</td>
<td>$8$</td>
<td></td>
</tr>
<tr>
<td>$w_3$</td>
<td>$f(w_3)$</td>
<td>$3$</td>
<td>$\to$</td>
<td>$27$</td>
<td></td>
</tr>
<tr>
<td>$w_4$</td>
<td>$f(w_4)$</td>
<td>$4$</td>
<td>$\to$</td>
<td>$64$</td>
<td></td>
</tr>
<tr>
<td>$w_5$</td>
<td>$f(w_5)$</td>
<td>$5$</td>
<td>$\to$</td>
<td>$125$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\to$</td>
<td>$\vdots$</td>
<td></td>
</tr>
<tr>
<td>$w_n$</td>
<td>$f(w_n)$</td>
<td>$10$</td>
<td>$\to$</td>
<td>$1000$</td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\to$</td>
<td>$\vdots$</td>
<td></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>(P_1)</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>(P_2)</td>
<td>(\times)</td>
</tr>
<tr>
<td>(P_3)</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>(P_4)</td>
<td>(\times)</td>
</tr>
<tr>
<td>(P_5)</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
<tr>
<td>(P_n)</td>
<td>(\times)</td>
</tr>
</tbody>
</table>

\(\Sigma^*\):

- \(w_1 \in L\) runs forever
- \(w_2 \notin L\) runs forever
- \(w_3 \notin L\) runs forever
- \(w_4 \in L\) runs forever
- \(w_5 \in L\) runs forever
- \(\ldots\)
- \(w_n \notin L\) runs forever
Types of computational problems solved by TM’s

<table>
<thead>
<tr>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>The three types of computational problems solved by TM’s are:</td>
</tr>
<tr>
<td>• Turing-decidable languages</td>
</tr>
<tr>
<td>• Turing-computable functions</td>
</tr>
<tr>
<td>• Turing-semidecidable languages</td>
</tr>
</tbody>
</table>

• Can we formalize the notion of an algorithm using the computation ideas described above?
### 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.
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.

<table>
<thead>
<tr>
<th>Type of computation</th>
<th>Always halt?</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?

<table>
<thead>
<tr>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm:</strong> Turing machine for a Turing-decidable language or Turing machine for a Turing-computable function.</td>
</tr>
<tr>
<td><strong>Algorithmic solvability:</strong> Turing-decidability or Turing-computability</td>
</tr>
<tr>
<td><strong>Algorithmic unsolvability:</strong> Turing-undecidability or Turing-noncomputability i.e., Turing-semidecidability and not Turing-semidecidability</td>
</tr>
</tbody>
</table>
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".
Some questions about algorithmic unsolvability

- Why do we call Turing-decidable and Turing-semidecidable languages as **recursive** and **recursively enumerable**, respectively?
- What is the **intuition** behind algorithmic unsolvability?
- What is the **relationship** between recursive and recursively enumerable languages?
- What are the **techniques to prove algorithmic unsolvability**?
- What are some **real-world** problems that **cannot be solved** by human minds or real computers (from past, present, future)?
Chomsky hierarchy

- Regular
- Context-free
- Context-sensitive
- Turing-decidable
- Turing-semidecidable
- Algorithmically solvable
- Algorithmically unsolvable
  (finite time)
  (infinite time)
  co-Turing-semidecidable
Chomsky hierarchy
Some properties of languages

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?

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• How can we prove that there are some computational problems that are algorithmically unsolvable?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Directions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Show that there are languages that are Turing-semidecidable but not Turing-decidable:</td>
</tr>
<tr>
<td>B. Show that there are languages that are not Turing semidecidable:</td>
</tr>
</tbody>
</table>

<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>
Let’s construct three non-halting Turing machines for $\Sigma = \{a\}$ and $\Gamma = \Sigma \cup \{>, \square\}$ 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 (Γ)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>▶</td>
<td>a</td>
</tr>
<tr>
<td>$q_0$</td>
<td>$(q_0, \rightarrow)$</td>
<td>$(q_0, \rightarrow)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current state</th>
<th>Current symbol (Γ)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>▶</td>
<td>a</td>
</tr>
<tr>
<td>$q_0$</td>
<td>$(q_0, \rightarrow)$</td>
<td>$(q_0, a)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Current state</th>
<th>Current symbol (Γ)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>▶</td>
<td>a</td>
</tr>
<tr>
<td>$q_0$</td>
<td>$(q_0, \rightarrow)$</td>
<td>$(q_0, \leftarrow)$</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></td>
<td></td>
<td>$q_0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\triangleright$</td>
</tr>
<tr>
<td>0</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>1</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>2</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>3</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>4</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>5</td>
<td>$q_0$</td>
<td>$a$</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 $M_2$

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

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>$a$</td>
</tr>
<tr>
<td>1</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>2</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>3</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>4</td>
<td>$q_0$</td>
<td>$a$</td>
</tr>
<tr>
<td>5</td>
<td>$q_0$</td>
<td>$a$</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 $M_3$

\[(\uparrow, \rightarrow), (a, \leftarrow), (\square, \leftarrow)\]

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

<table>
<thead>
<tr>
<th>Time</th>
<th>State</th>
<th>Tape</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q_0$</td>
<td>△</td>
</tr>
<tr>
<td>0</td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>0</td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>0</td>
<td>$a$</td>
<td>$a$</td>
</tr>
<tr>
<td>0</td>
<td>$\square$</td>
<td>$\square$</td>
</tr>
<tr>
<td>0</td>
<td>$\square$</td>
<td>$\square$</td>
</tr>
<tr>
<td>0</td>
<td>$\square$</td>
<td>$\square$</td>
</tr>
<tr>
<td>0</td>
<td>$\square$</td>
<td>$\square$</td>
</tr>
<tr>
<td>0</td>
<td>$\square$</td>
<td>$\square$</td>
</tr>
<tr>
<td>0</td>
<td>$\square$</td>
<td>$\square$</td>
</tr>
<tr>
<td>0</td>
<td>$\square$</td>
<td>$\square$</td>
</tr>
<tr>
<td>0</td>
<td>$\square$</td>
<td>$\square$</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.
Problem

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

Solution


1. Prove that the set of decision problems is uncountable.
2. Prove that the number of Turing machines is countable.

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 $\Sigma^* \to \{0, 1\}$) 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.
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 diagonalization.</td>
</tr>
</tbody>
</table>
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>\vdots</td>
<td>\vdots</td>
</tr>
</tbody>
</table>
• Construct a TM that accepts language
  \[ L_d = \{ w_i \mid w_i \notin L(M_i) \} \]
i.e., \[ L_d = d_1d_2d_3 \ldots, \]
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>
<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>( \vdots )</td>
<td>\vdots</td>
</tr>
<tr>
<td>( M_d )</td>
<td>\boxed{0}</td>
</tr>
</tbody>
</table>
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 \( w_k \).
  \[ \implies w_k \notin L_d \quad (\because \text{defn. of } L_d) \]
  \[ \implies w_k \notin L(M_k) \quad (\because L_d = L(M_k)) \]
  \[ \implies M_k \text{ does not accept } w_k \quad (\because \text{defn. of } L(M_k)) \]
- 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.
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**
  \[ \{\langle M, w \rangle \mid \text{TM } M \text{ accepts input string } w \} \]
- Let’s call the hypothetical method as **SIMULATE**.
  1. Prove that **SIMULATE** is Turing-semidecidable.
  2. Prove that **SIMULATE** is algorithmically impossible.
Solution (continued)

Part 1. Prove that SIMULATE is Turing-semidecidable.

- Consider the following generic procedure.

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

Solution (continued)

Part 2. Prove that `Simulate` is algorithmically impossible.
Proof by contradiction.
- Let’s assume that `Simulate` is algorithmically possible
  i.e., `Simulate` always halts giving a correct answer.
- Then, we construct the `Paradox` algorithm as follows.

<table>
<thead>
<tr>
<th>Paradox(⟨M⟩)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>result ← Simulate(⟨M, ⟨M⟩⟩)</code></td>
</tr>
<tr>
<td>2. if <code>result = accept</code> then reject</td>
</tr>
<tr>
<td>3. elseif <code>result = reject</code> then accept</td>
</tr>
</tbody>
</table>

![Diagram of Paradox and Simulate algorithms](image-url)
Part 2. Prove that \text{Simulate} is algorithmically impossible.

<table>
<thead>
<tr>
<th>\text{Paradox}(⟨M⟩)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Source code of a computer program</td>
</tr>
<tr>
<td><strong>Output:</strong> Accept or reject</td>
</tr>
<tr>
<td><strong>Require:</strong> Invoke \text{Paradox}(⟨\text{Paradox}⟩)</td>
</tr>
<tr>
<td>1. result ← \text{Simulate}(⟨M, ⟨M⟩⟩)</td>
</tr>
<tr>
<td>2. if result = accept then reject</td>
</tr>
<tr>
<td>3. elseif result = reject then accept</td>
</tr>
</tbody>
</table>

- **Case 1.** \text{Paradox} accepts ⟨\text{Paradox}⟩
  \[ \Rightarrow \text{Simulate} \text{ rejects} \langle \text{Paradox, } ⟨\text{Paradox}⟩ \rangle \]
  \[ \Rightarrow \text{Paradox} \text{ rejects} \langle \text{Paradox} \rangle. \]

- **Case 2.** \text{Paradox} rejects ⟨\text{Paradox}⟩
  \[ \Rightarrow \text{Simulate} \text{ accepts} \langle \text{Paradox, } ⟨\text{Paradox}⟩ \rangle \]
  \[ \Rightarrow \text{Paradox} \text{ accepts} \langle \text{Paradox} \rangle. \]

- **Contradiction!** Hence, \text{Simulate} is algorithmically impossible.
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 = \{⟨M, w⟩ \mid \text{TM } M \text{ halts on input string } w\}

Let’s call the hypothetical method as \text{HALT}.

1. Prove that \text{HALT} is Turing-semidecidable.
2. Prove that \text{HALT} is algorithmically impossible.
Part 1. Prove that \texttt{Halt} is Turing-semidecidable.

- Consider the following generic procedure.

\begin{tabular}{|l|}
  \hline
  \texttt{Halt}(⟨M, w⟩)
  \\
  1. Simulate TM \(M\) on input string \(w\)
  2. if \(M\) accepts \(w\) or \(M\) rejects \(w\) then
  3. \hspace{1em} accept
  4. \hspace{1em} else if \(M\) runs forever then
  5. \hspace{1em} reject
  \hline
\end{tabular}

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

So, \texttt{Halt} is Turing-semidecidable.
Solution (continued)

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.

$\text{Paradox}(⟨M⟩)$

1. $\text{result} ← \text{Halt}(⟨M, ⟨M⟩⟩)$
2. if $\text{result} = \text{accept}$ then run forever
3. elseif $\text{result} = \text{reject}$ then accept
Part 2. Prove that \texttt{HALT} is algorithmically impossible.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{PARADOX}(\langle M \rangle) \\
\hline
\textbf{Input:} Source code of a computer program \\
\textbf{Output:} Accept or reject \\
\textbf{Require:} Invoke \texttt{PARADOX}(\langle \texttt{PARADOX} \rangle) \\
1. \texttt{result} \leftarrow \texttt{HALT}(\langle M, \langle M \rangle \rangle) \\
2. if \texttt{result} = accept then run forever \\
3. elseif \texttt{result} = reject then accept \\
\hline
\end{tabular}
\end{table}

- **Case 1.** \texttt{PARADOX} accepts \langle \texttt{PARADOX} \rangle
  \[\rightarrow \texttt{HALT} \text{ rejects } \langle \texttt{PARADOX}, \langle \texttt{PARADOX} \rangle \rangle\]
  \[\rightarrow \texttt{PARADOX} \text{ runs forever on } \langle \texttt{PARADOX} \rangle.\]

- **Case 2.** \texttt{PARADOX} runs forever on \langle \texttt{PARADOX} \rangle
  \[\rightarrow \texttt{HALT} \text{ accepts } \langle \texttt{PARADOX}, \langle \texttt{PARADOX} \rangle \rangle\]
  \[\rightarrow \texttt{PARADOX} \text{ accepts } \langle \texttt{PARADOX} \rangle.\]

- Contradiction! Hence, \texttt{HALT} is algorithmically impossible.
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 $x \in \Sigma^*$

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

```markdown
<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notation.</td>
</tr>
<tr>
<td>Meaning.</td>
</tr>
<tr>
<td>Intuition.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Consequences.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
## 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.

$$L_{\text{sim}} = \{\langle M, w \rangle | \text{TM } M \text{ accepts input string } w\}$$

$$L_{\text{halt}} = \{\langle M, w \rangle | \text{TM } M \text{ halts on input string } w\}$$

Proof by contradiction and proof by reduction.

Let's call the hypothetical method as $Halt$. We show that if $Halt$ is algorithmically possible, then $Simulate$ is algorithmically possible, too.
### 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.
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.

<table>
<thead>
<tr>
<th>\textsc{Simulate}(⟨M, w⟩)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. result ← \textsc{Halt}(⟨M, w⟩)</td>
</tr>
<tr>
<td>2. if result = reject then reject \quad \triangleright \text{ M runs forever on } w</td>
</tr>
<tr>
<td>3. elseif result = accept then</td>
</tr>
<tr>
<td>4. \text{ Simulate } M \text{ on } w</td>
</tr>
<tr>
<td>5. if } M \text{ accepts } w \text{ then accept } \quad \triangleright \text{ M accepts } w</td>
</tr>
<tr>
<td>6. elseif } M \text{ rejects } w \text{ then reject } \quad \triangleright \text{ M rejects } w</td>
</tr>
</tbody>
</table>

- 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.
Hofstadter’s $ac$ puzzle

Problem

Starting with the string $ab$, can you derive $ac$, using the following productions?

1. Add a $c$ to the end of any string ending in $b$.
   i.e., $xb \rightarrow xbc$

2. Double the string after the first character $a$.
   i.e., $ax \rightarrow axx$

3. Replace any $bbb$ with a $c$.
   i.e., $xbbby \rightarrow xcy$

4. Remove any $cc$.
   i.e., $xccy \rightarrow xy$
Hofstadter’s $ac$ puzzle

Solution (core idea)

- The problem cannot be solved.
- **Invariant:** $n = (\#b's \text{ in a string})$ is not divisible by 3.
- The invariant is true for the starting string $ab$.
  - The invariant is true for all strings derivable from $ab$.
- The invariant is false for $ac$.
- Hence, the string $ac$ cannot be derived from the string $ab$. 
**Hofstadter’s \(ac\) puzzle**

### Solution (continued)

- **[Starting string.]** For the starting string \(ab\), \(n = 1\). Hence, the invariant is true.
- **[Rules 1 and 4.]** The rules do not change \(#b\)'s. Hence, the invariant is true.
- **[Rule 2.]** Doubling a number that is not divisible by 3 does not make it divisible by 3. Hence, the invariant is true.
- **[Rule 3.]** Subtracting 3 from a number that is not divisible by 3 does not make it divisible by 3. Hence, the invariant is true.
- The desired string \(ac\) cannot be derived because \(n = 0\). And 0 is divisible by 3.

### Reference:

https://en.wikipedia.org/wiki/MU_puzzle
**Decision problems involving TM’s**

<table>
<thead>
<tr>
<th>Algorithmically solvable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Given a TM $M$, does $M$ have at least 481 states?</td>
</tr>
<tr>
<td>- Given a TM $M$, does $M$ take more than 481 steps on input $\epsilon$?</td>
</tr>
<tr>
<td>- Given a TM $M$, does $M$ take more than 481 steps on some input?</td>
</tr>
<tr>
<td>- Given a TM $M$, does $M$ take more than 481 steps on all inputs?</td>
</tr>
<tr>
<td>- Given a TM $M$, does $M$ ever move its head more than 481 tape cells away from the left endmarker on input $\epsilon$?</td>
</tr>
</tbody>
</table>
Decision problems involving TM’s

**Algorithmically unsolvable.**
- Given a TM $M$ and an input string $w$, is $w \in L(M)$?
- Given a TM $M$, is $L(M)$ nonempty?
- Given a TM $M$, is $L(M) = \Sigma^*$?
- Given a TM $M$, is $L(M)$ a regular language?
- Given a TM $M$, is $L(M)$ a CFL?
- Given a TM $M$, is $L(M)$ a recursive language?
- Given a TM $M$, is $L(M)$ recursively enumerable?
- Given two TM’s $M_1$ and $M_2$, is $L(M_1) = L(M_2)$?
- Given two TM’s $M_1$ and $M_2$, is $L(M_1) \subseteq L(M_2)$?
- Given two TM’s $M_1$ and $M_2$, is $L(M_1) \cap L(M_2)$ nonempty?
- Given a TM $M$ and an input string $w$, does $M$ use a finite amount of tape?
Post correspondence problem (PCP)

**Problem**
- Given the set of dominos, is it possible to list these dominos (repetitions permitted) so that the string of symbols on top is the same as the string of symbols on the bottom?

\[(D_1, D_2, D_3, D_4) = (b \text{ ca}, a \text{ ab}, ca \text{ a}, abc \text{ c})\]

**Solution**
- Yes!
- A solution:
  
  \[
  \begin{align*}
  D_2 & \quad D_1 \\
  a & \quad \text{ab} \\
  \text{ca} & \quad a \\
  \text{ca} & \quad \text{abc} \\
  \end{align*}
  \]

- \(= \text{abca}\text{abc}\)
**Problem**

- Given the set of dominos, is it possible to list these dominos (repetitions permitted) so that the string of symbols on top is the same as the string of symbols on the bottom?

\[(D_1, D_2, D_3, D_4) = \begin{pmatrix} b & a & ca & abc \\
                               ca & ab & a & c \end{pmatrix} \]

**Solution**

- Yes!
- A solution: \([D_2D_1D_3D_2D_4].\)

\[
\begin{array}{cccc}
a & b & ca & a & abc \\
ab & ca & a & ab & c \\
\end{array} = \begin{array}{c}
abcaaabc \\
abcaaaabc \\
\end{array} = \text{match}
\]
Post correspondence problem (PCP)

Problem

- Given the set of dominos, is it possible to list these dominos (repetitions permitted) so that the string of symbols on top is the same as the string of symbols on the bottom?

\[(D_1, D_2, D_3) = \left( \begin{array}{c}
\text{abc} \\
\text{ab} \\
\end{array}, \quad \begin{array}{c}
\text{ca} \\
\text{a} \\
\end{array}, \quad \begin{array}{c}
\text{acc} \\
\text{ba} \\
\end{array} \right)\]
Post correspondence problem (PCP)

Problem

- Given the set of dominos, is it possible to list these dominos (repetitions permitted) so that the string of symbols on top is the same as the string of symbols on the bottom?

\[ (D_1, D_2, D_3) = \left( \begin{array}{c}
abc \\
ab \\
a
\end{array} \right), \quad
\begin{array}{c}
ca \\
a
\end{array}, \quad
\begin{array}{c}
acc \\
ba
\end{array} \right) \]

Solution

- No!

  Every top string is greater than its bottom string.
Problem

Given the set of dominos, is there a match?

- \((D_1, D_2, D_3) = (a \quad ab \quad bba, baa \quad aa \quad bb)\)
- \((D_1, D_2, D_3, D_4) = (b \quad abb \quad aba \quad bbaaa, bab \quad b \quad a \quad babaa)\)
- \((D_1, D_2, D_3) = (bb \quad ab \quad c, b \quad ba \quad bc)\)
- \((D_1, D_2, D_3) = (bbab \quad abba \quad b, b \quad bb \quad bba)\)
Post correspondence problem (PCP)

Problem
Given the set of dominos, is there a match?

- \((D_1, D_2, D_3) = (\begin{array}{c}
  a \\
  \text{baa}
\end{array}, \begin{array}{c}
  ab \\
  \text{aa}
\end{array}, \begin{array}{c}
  bba \\
  \text{bb}
\end{array})\)

- \((D_1, D_2, D_3, D_4) = (\begin{array}{c}
  b \\
  \text{bab}
\end{array}, \begin{array}{c}
  ab \\
  \text{ba}
\end{array}, \begin{array}{c}
  c \\
  \text{bc}
\end{array}, \begin{array}{c}
  \text{bbaa}
\end{array})\)

- \((D_1, D_2, D_3) = (\begin{array}{c}
  bb \\
  \text{b}
\end{array}, \begin{array}{c}
  ab \\
  \text{ba}
\end{array}, \begin{array}{c}
  b \\
  \text{bba}
\end{array})\)

- \((D_1, D_2, D_3) = (\begin{array}{c}
  bbab \\
  \text{b}
\end{array}, \begin{array}{c}
  ab \\
  \text{bb}
\end{array}, \begin{array}{c}
  b \\
  \text{bba}
\end{array})\)

Solution

- Yes! Infinite solutions: \([D_3 D_2 D_3 D_1]^+\).
- Yes! Infinite solutions: \([D_1 D_2 D_3 D_1]^+\).
- Yes! Infinite solutions: \([D_1 D_2^+ D_3]^+\).
- Yes! Shortest solution has length 252.
Post correspondence problem (PCP)

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given the set of dominos, is there a match?</td>
</tr>
<tr>
<td>$(D_1, D_2, D_3) = (\begin{array}{c} bb \ abb \end{array}, \begin{array}{c} ab \ a \end{array}, \begin{array}{c} aab \ bba \end{array})$</td>
</tr>
</tbody>
</table>

Start index $= 1$

- $(D_1, D_2, D_3) = (\begin{array}{c} bb \\ abb \end{array}, \begin{array}{c} ab \\ a \end{array}, \begin{array}{c} aab \\ bba \end{array})$

Solution: No!
Post correspondence problem (PCP)

**Problem**
Given the set of dominos, is there a match?

- \((D_1, D_2, D_3) = (\begin{array}{c} bb \\ abb \end{array}, \begin{array}{c} ab \\ a \end{array}, \begin{array}{c} aab \\ bba \end{array})\)

**Solution**
- No!

```
  Start
     /   \
   /     \
  /       \
 index = 1
  \begin{array}{c} bb \\ abb \end{array} X

  index = 1
  \begin{array}{c} ab \\ a \end{array} X

  index = 1
  \begin{array}{c} aab \\ bba \end{array} X

  index = 2
  \begin{array}{c} bb \\ abb \end{array} X

  index = 2
  \begin{array}{c} ab \\ a \end{array} X

  index = 2
  \begin{array}{c} aab \\ bba \end{array} X
```
Post correspondence problem (PCP)

**Problem**

- Discovered by Emil Post in 1940’s.
- You are given the set $P$ of dominos
  
  $$(t_1 b_1, t_2 b_2, \ldots, t_k b_k)$$

  A match is a sequence $i_1 i_2 \ldots i_\ell$, where
  
  $$t_{i_1} t_{i_2} \cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}.$$  

- Is there an algorithm to determine if $P$ has a match?
Post correspondence problem (PCP)

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Discovered by Emil Post in 1940’s.</td>
</tr>
<tr>
<td>• You are given the set $P$ of dominos</td>
</tr>
<tr>
<td>$$(t_1, b_1, \ldots, t_k, b_k)$$</td>
</tr>
<tr>
<td>A match is a sequence $i_1i_2\ldots i_\ell$, where</td>
</tr>
<tr>
<td>$t_{i_1}t_{i_2}\cdots t_{i_\ell} = b_{i_1} b_{i_2} \cdots b_{i_\ell}$.</td>
</tr>
<tr>
<td>• Is there an algorithm to determine if $P$ has a match?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• No! The problem is algorithmically unsolvable.</td>
</tr>
<tr>
<td>• Let $\text{PCP} = {\langle P \rangle \mid P$ is a domino set that has a match$}$.</td>
</tr>
<tr>
<td>PCP is not Turing-decidable.</td>
</tr>
</tbody>
</table>