IV.3 (a) Definition of the Turing Machine

IV.3 (b) Equivalence of URM computable and Turing computable functions

IV.3 (c) Undecidability of the Turing Halting Problem

IV.3 (d) The Church-Turing Thesis
(a) Definition of the Turing Machine

- There are two problems with the model of a URM:
- Execution of a single URM instruction might take arbitrarily long:
  - Consider $\text{succ}(n)$.
  - If $R_n$ contains in binary $\underbrace{111 \cdots 111}_k$, this instruction replaces it by $\underbrace{1000 \cdots 000}_k$.
  - We have to replace $k$ symbols 1 by 0.
  - $k$ is arbitrary
    → this single step might take arbitrarily long time.
That incrementing a number by one takes arbitrarily many steps happens on a real computer as well:

- If we want to represent arbitrary big numbers on the computer, we have to represent them by multiple machine integers
  - Then incrementing a number by one will correspond to arbitrarily many machine instructions (although usually only a few).
- However, often in complexity theory this problem is ignored because the effect is marginal in real applications.
  - The exception are applications in which very big integers occur, e.g. tests for primality. There this effect cannot be ignored any more.
First Problem of URMs

- If one takes this effect into account, one needs in many examples to multiply the running time by a factor of $\ln(n)$, where $n$ is the largest number occurring.

- Therefore URMs unsuitable as a basis for defining the precise complexity of algorithms.

- However, there are theorems linking complexity of URMs to actual complexities of algorithms.
Second Problem of URMs

- We aim at a notion of computability, which covers all possible ways of computing something, independently of any concrete machine.
- URMs are a model of computation which covers current standard computers.
- However, there might be completely different notions of computability, based on symbolic manipulations of a sequence of characters, where it might be more complicated to see directly that all such computations can be simulated by a URM.
- It is more easy to see that such notions are covered by the Turing machine model of computation.
Idea of a Turing Machine

Idea of a Turing machine (TM):
Analysis of a computation carried out by a human being (agent) on a piece of paper.

\[
\begin{array}{c}
15 \\
15 \\
90 \\
240 \\
\end{array}
\]
Idea of a Turing Machine

Steps in this formulation:

- Algorithm should be deterministic.
- The agent will use only finitely many symbols, put at discrete positions on the paper.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can replace a two-dimensional piece of paper by one potentially infinite tape, by using a special symbol for a line break. Each entry on this tape is called a cell:

\[
\begin{array}{cccccc}
\cdots & 1 & 5 & . & 1 & 6 \quad = \quad \text{CR} \quad \cdots \\
& 1 & 5 & \text{CR} & 1 & 5 & \text{CR} \\
\end{array}
\]
Steps in Formalising TMs

- In the real situation, agent can look at several cells at the same time, but bounded by his physical capability. Can be simulated by looking at one cell only at any time, and moving around in order to get information about neighbouring cells.
Steps in Formalising TMs

- In the real situation, an agent can make arbitrary jumps, but bounded by the physical ability of the agent. Each such jump can be simulated by finitely many one-step jumps.
  - Restriction to one-step movements.
Steps in Formalising TMs

- In the real situation, an agent can make arbitrary jumps, but bounded by the physical ability of the agent. Each such jump can be simulated by finitely many one-step jumps.
  - Restriction to one-step movements.

| ... | 1 | 5 | . | 1 | 6 | = | CR | ... | 1 | 5 | CR | ... |

↑ Head
Steps in Formalising TMs

- In the real situation, an agent can make arbitrary jumps, but bounded by the physical ability of the agent. Each such jump can be simulated by finitely many one-step jumps.

→ Restriction to one-step movements.

\[
\begin{array}{ccccccccc}
\vdots & 1 & 5 & . & 1 & 6 & = & \text{CR} & \text{CR} & \text{CR} & \text{CR} & \vdots \\
\end{array}
\]

↑

Head
Steps in Formalising TMs

In the real situation, an agent can make arbitrary jumps, but bounded by the physical ability of the agent. Each such jump can be simulated by finitely many one-step jumps. → Restriction to one-step movements.

| ⋯ | 1 | 5 | . | 1 | 6 | = | CR | ⋯ |

↑

Head
Steps in Formalising TMs

- In the real situation, an agent can make arbitrary jumps, but bounded by the physical ability of the agent. Each such jump can be simulated by finitely many one-step jumps. → Restriction to one-step movements.

<table>
<thead>
<tr>
<th>...</th>
<th>1</th>
<th>5</th>
<th>.</th>
<th>1</th>
<th>6</th>
<th>=</th>
<th>CR</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Head</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↑

...
Steps in Formalising TMs

- Agent operates purely mechanistically:
  Reads a symbol, and depending on it changes it and makes a movement.
  Agent himself will have only finite memory.
  → There is a finite state of the agent, and, depending on the state and the symbol at the head, a next state, a new symbol, and a movement is chosen.
Agent operates purely mechanistically:
Reads a symbol, and depending on it changes it and makes a movement.
Agent himself will have only finite memory.
→ There is a finite state of the agent, and, depending on the state and the symbol at the head, a next state, a new symbol, and a movement is chosen.
Agent operates purely mechanistically:
Reads a symbol, and depending on it changes it and makes a movement.
Agent himself will have only finite memory.
→ There is a finite state of the agent, and, depending on the state and the symbol at the head, a next state, a new symbol, and a movement is chosen.

\[
\begin{array}{cccc|c|c|c}
\cdots & 1 & 5 & \cdot & 0 & 6 & = & \text{CR} & \cdots \\
\uparrow & & & & & & & 1 & 5 & \text{CR} & \cdots \\
\end{array}
\]
Steps in Formalising TMs

- Agent operates purely mechanistically:
  Reads a symbol, and depending on it changes it and makes a movement.
  Agent himself will have only finite memory.
  → There is a finite state of the agent, and, depending on the state and the symbol at the head, a next state, a new symbol, and a movement is chosen.

\[ \cdots | 1 | 5 | . | 0 | 7 | = | \text{CR} | 1 | 5 | \text{CR} | \cdots \]

\[ \uparrow \]

\[ S_2 \]
Steps in Formalising TMs

- Agent operates purely mechanistically: Reads a symbol, and depending on it changes it and makes a movement.
  Agent himself will have only finite memory.
  → There is a finite state of the agent, and, depending on the state and the symbol at the head, a next state, a new symbol, and a movement is chosen.

\[
\begin{array}{cccccccc}
\cdots & 1 & 3 & . & 3 & 7 & = & \text{CR} & 1 & 5 & \text{CR} & \cdots \\

\uparrow
\end{array}
\]
Definition of TMs

A **Turing machine** is a five tuple (or quintuple) $(\Sigma, S, I, \downarrow \downarrow, s_0)$, where

- $\Sigma$ is a finite set of symbols, called the **alphabet** of the Turing machine. On the tape, the symbols in $\Sigma$ will be written.
  - $\Sigma$ is the Greek capital letter “Sigma”.
- $S$ is a finite set of states.
Definition of TMs

... 1 5 . 1 6 = CR ... 1 5 CR ...

\[ \uparrow \]

\[ s_0 \]

- \( I \) is a finite sets of quintuples \((s, a, s', a', D)\), where
  - \( s, s' \in S \),
  - \( a, a' \in \Sigma \),
  - \( D \in \{L, R\} \),

s.t. for every \( s \in S, a \in \Sigma \) there is \textbf{at most one} \( s', a', D \) s.t. \((s, a, s', a', D) \in I\).

The elements of \( I \) are called \textit{instructions}.

- \( \_ \_ \_ \in \Sigma \) (a symbol for blank).

- \( s_0 \in S \) (the initial state).
An instruction \((s, a, s', a', D) \in I\) means the following:

- If the Turing machine is in state \(s\), and the symbol at position of the head is \(a\), then
  - the state is changed to \(s'\),
  - the symbol at this position is changed to \(a'\),
  - if \(D = L\), the head moves left,
  - if \(D = R\), the head moves right.
IV.3 (a) Definition of the Turing Machine

Meaning of Instructions

▷ An instruction \((s, a, s', a', D) \in I\) means the following:
  ▷ If the Turing machine is in state \(s\), and the symbol at position of the head is \(a\), then
    ▷ the state is changed to \(s'\),
    ▷ the symbol at this position is changed to \(a'\),
    ▷ if \(D = L\), the head moves left,
    ▷ if \(D = R\), the head moves right.

Example:
\((s_0, 1, s_1, 0, R)\)
\((s_1, 6, s_2, 7, L)\)
Meaning of Instructions

- An instruction \((s, a, s', a', D) \in I\) means the following:
  - If the Turing machine is in state \(s\), and the symbol at position of the head is \(a\), then
    - the state is changed to \(s'\),
    - the symbol at this position is changed to \(a'\),
    - if \(D = L\), the head moves left,
    - if \(D = R\), the head moves right.

Example:

\[(s_0, 1, s_1, 0, R)\]
\[(s_1, 6, s_2, 7, L)\]

\[
\begin{array}{cccccccccc}
\cdots & 1 & 5 & . & 1 & 6 & = & \text{CR} & \text{CR} & \text{CR} & \cdots \\
& \uparrow & & & & & & & & & \\
& s_0 & & & & & & & & & \\
\end{array}
\]
Meaning of Instructions

- An instruction $(s, a, s', a', D) \in I$ means the following:
  - If the Turing machine is in state $s$, and the symbol at position of the head is $a$, then
    - the state is changed to $s'$,
    - the symbol at this position is changed to $a'$,
    - if $D = L$, the head moves left,
    - if $D = R$, the head moves right.

Example:

$(s_0, 1, s_1, 0, R)$
$(s_1, 6, s_2, 7, L)$

\[
\begin{array}{ccccccc}
\cdots & 1 & 5 & . & 0 & 6 & = & \text{CR} & \cdots \\
\uparrow & \text{s}_1 & \ \end{array}
\]
Meaning of Instructions

- An instruction \((s, a, s', a', D) \in I\) means the following:
  - If the Turing machine is in state \(s\), and the symbol at position of the head is \(a\), then
    - the state is changed to \(s'\),
    - the symbol at this position is changed to \(a'\),
    - if \(D = L\), the head moves left,
    - if \(D = R\), the head moves right.

Example:
\((s_0, 1, s_1, 0, R)\)
\((s_1, 6, s_2, 7, L)\)

\[
\begin{array}{cccccccc}
\cdots & 1 & 5 & . & 0 & 7 & = & \text{CR} & \cdots \\
\uparrow & & & & & & & & \\
S_2
\end{array}
\]
Meaning of Instructions

- Note that for the above it is important that for every \( s \in S, \ a \in \Sigma \) there is at most one \( s', a', D \) s.t. \((s, a, s', a', D) \in S\).
  - Without this condition, there might be more than one choice of selecting a new tape symbol, next state and direction.
  - If we omit this condition, we obtain a **non-deterministic TM**. In this case the machine selects in each step one of the possible choices (provided there exist one) at random.

- If the Turing machine is in a state \( s \) and reads symbol \( a \) at his head, and there are no \( s', a', D \) s.t. \((s, a, s', a', D) \in I\), then the Turing machine stops.
TM Architecture vs. TM Program

As for URMs a TM means both the TM architecture and the TM program.

- The TM architecture describes that a TM has a tape, a head, a state, and how it is executed.
- The TM program consists of the alphabet on the tape, the set of states, the instructions, the symbol for blank and the initial state.

When asked to define a TM which has a certain behaviour one usually actually asks for a TM program, such that a TM with this program has this behaviour.
A TM

\[(\Sigma, S, I, \sqcup, s_0)\]

can be visualised by a labelled graph as follows:

- Vertices: states (i.e. \(S\)).
- Edges: If \((s, a, t, b, D) \in I\), then there is an edge

\[s \overset{a/b, D}{\rightarrow} t\]

Furthermore we write an arrow to the initial state coming from nowhere.

If there are several vertices from \(s\) to \(s'\), one draws only one arrow with one label for each vertex.
Example

The Turing machine with initial state $s_0$ and instructions

$$\{ (s_0, 0, s_0, 0, R),
(s_0, 1, s_0, 0, R),
(s_0, \square, s_1, \square, L),
(s_1, 0, s_1, 0, L),
(s_1, \square, s_2, \square, R) \}$$

is visualised as follows (we write $B$ instead of $\square$):
Example

- The TM on the previous slide sets the binary number the head is pointing to to zero, provided to the left of the head there are is a blank.

**Exercise:**
- This example assumes that the TM points to the left most digit of a binary number.
- Modify this TM, so that it works as well if the TM points initially to any digit of a binary number.
IV.3 (a) Definition of the Turing Machine

Equivalent Representations

- The pictorial representation is equivalent to the set of instructions plus an initial state.
- Therefore a TM can both be given by listing its instructions and by the pictorial representation.
- Furthermore the only relevant sets of instructions are those occurring in the pictorial representation. Similarly for the set of symbols on the tape.
- Therefore, assuming that the blank symbol is canonical, we can take the pictorial representation as the complete definition of a TM (with states being the set of states occurring in the diagram, and alphabet consisting of the canonical blank symbol and the states occurring in the diagram).
We write $(b_0, \ldots, b_{k-1})_2$ for the natural number having binary representation $b_0, \ldots, b_{k-1}$, e.g.

$$(01010)_2 = 10$$

Note that we allow for leading zeros, so

$$(001010)_2 = (01010)_2 = (1010)_2 = 10$$

We allow as well the empty string

$$(*)_2 = 0$$
Example of a TM

- Development of a TM with $\Sigma = \{0, 1, \sqcup\}$,
  - where $\sqcup$ is the symbol for the blank entry.
- Functionality of the TM:
  - Assume initially the following:
    - The tape contains binary number,
    - The rest of the tape contains $\sqcup$.
    - The head points to any digit of the number.
    - The TM in state $s_0$.
  - Then the TM stops after finitely many steps and then
    - the tape contains the original number incremented by one,
    - the rest of tape contains $\sqcup$,
    - the head points to most significant bit.
Example

Initially

```
... 1 0 1 0 0 1 0 0 1 1 1 ...
```

↑

S₀
Example

Initially

\[
\begin{array}{cccccccccccc}
\cdots & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & \cdots \\
\end{array}
\]

\[\uparrow \]

\[s_0\]

Finally

\[
\begin{array}{cccccccccccc}
\cdots & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & \cdots \\
\end{array}
\]

\[\uparrow \]

\[s_3\]
Construction of the TM

- TM is \((\{0, 1, \sqcup \}, S, I, \sqcup, s_0)\).
- States \(S\) and instructions \(I\) developed in the following.
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is \(\_\_\), move head left, leave symbol again as it is.

- Achieved by the following instructions:

\[
(s_0, 0, s_0, 0, R) \\
(s_0, 1, s_0, 1, R) \\
(s_0, \_\_, s_1, \_\_, L)
\]

- At the end TM is in state \(s_1\).
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is \( \underline{\underline{\_}} \), move head left, leave symbol again as it is.
- Achieved by the following instructions:
  
  \[
  \begin{align*}
  (s_0, 0, s_0, 0, R) \\
  (s_0, 1, s_0, 1, R) \\
  (s_0, \underline{\underline{\_}}, s_1, \underline{\underline{\_}}, L)
  \end{align*}
  \]
- At the end TM is in state \( s_1 \).
**Step 1**

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is \(\square\), move head left, leave symbol again as it is.

- Achieved by the following instructions:

  \[(s_0, 0, s_0, 0, R)\]
  \[(s_0, 1, s_0, 1, R)\]
  \[(s_0, \square, s_1, \square, L)\]

- At the end TM is in state \(s_1\).

---

- There is a transition diagram showing the states \(s_0\) and \(s_1\) with transitions labeled by symbols and directions: \(B/B, L\) for 0/0, R for 1/1.
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is \[\text{\textdblbar}\text{\textdblbar}\], move head left, leave symbol again as it is.

- Achieved by the following instructions:
  - \((s_0, 0, s_0, 0, R)\)
  - \((s_0, 1, s_0, 1, R)\)
  - \((s_0, \text{\textdblbar}, s_1, \text{\textdblbar}, L)\)

- At the end TM is in state \(s_1\).
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is □□, move head left, leave symbol again as it is.
- Achieved by the following instructions:

\[
\begin{align*}
& (s_0, 0, s_0, 0, \text{R}) \\
& (s_0, 1, s_0, 1, \text{R}) \\
& (s_0, \□□, s_1, \□□, \text{L})
\end{align*}
\]

- At the end TM is in state \( s_1 \).
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is `⌜⌟`, move head left, leave symbol again as it is.

- Achieved by the following instructions:
  
  - (s₀, 0, s₀, 0, R)
  - (s₀, 1, s₀, 1, R)
  - (s₀, ⌜⌟, s₁, ⌜⌟, L)

- At the end TM is in state s₁.

```
⋯ 1 0 1 0 0 1 0 0 1 1 1 ⋯
```

![Diagram of the Turing Machine]

- s₀
  - B/B, L
  - 0/0, R
  - 1/1, R

- s₁
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is \[\text{⌞⌟}\], move head left, leave symbol again as it is.

- Achieved by the following instructions:

  \[(s_0, 0, s_0, 0, R)\]
  \[(s_0, 1, s_0, 1, R)\]
  \[(s_0, \text{⌞⌟}, s_1, \text{⌞⌟}, L)\]

- At the end TM is in state \(s_1\).
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is \( \text{□□} \), move head left, leave symbol again as it is.

- Achieved by the following instructions:
  \[(s_0, 0, s_0, 0, R)\]
  \[(s_0, 1, s_0, 1, R)\]
  \[(s_0, \text{□□}, s_1, \text{□□}, L)\]

- At the end TM is in state \( s_1 \).
Step 1

- Initially, move head to least significant bit.
  - i.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is □□, move head left, leave symbol again as it is.
- Achieved by the following instructions:
  - \((s_0, 0, s_0, 0, R)\)
  - \((s_0, 1, s_0, 1, R)\)
  - \((s_0, □□, s_1, □□, L)\)
- At the end TM is in state \(s_1\).
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is $\sqcap\sqcup$, move head left, leave symbol again as it is.

- Achieved by the following instructions:
  
  $(s_0, 0, s_0, 0, R)$
  $(s_0, 1, s_0, 1, R)$
  $(s_0, \sqcap\sqcup, s_1, \sqcap\sqcup, L)$

- At the end TM is in state $s_1$. 

```
... 1 0 1 0 0 1 0 0 1 1 1 ...
```

![Diagram of a Turing machine state transition](image)
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is $\Box$, move head left, leave symbol again as it is.

- Achieved by the following instructions:
  
  $$(s_0, 0, s_0, 0, R)$$
  $$(s_0, 1, s_0, 1, R)$$
  $$(s_0, \Box, s_1, \Box, L)$$

- At the end TM is in state $s_1$. 

```
... 1 0 1 0 0 1 0 0 1 1 1 ...
```

```
  +---+
  |   |
  |   |
  +---+
      ➔ S0
```
Step 1

- Initially, move head to least significant bit.
  - I.e. as long as symbol at head is 0 or 1, move right, leave symbol as it is.
  - If symbol is \(\underline{\underline{\text{\_\_}}\text{\_\_}}\), move head left, leave symbol again as it is.

- Achieved by the following instructions:
  
  \[(s_0, 0, s_0, 0, R)\]
  \[(s_0, 1, s_0, 1, R)\]
  \[(s_0, \underline{\underline{\text{\_\_}}\text{\_\_}}, s_1, \underline{\underline{\text{\_\_}}\text{\_\_}}, L)\]

- At the end TM is in state \(s_1\).
Step 2

Increasing a binary number $b$ done as follows:

- **Case number consists of 1 only:**
  - I.e. $b = (111 \cdots 111)_2$.
  - $b + 1 = (1000 \cdots 000)_2$.
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.
  - That’s what happens when we add by hand:

    \[
    \begin{array}{cccccccc}
    1 & 0 & 0 & 1 & 1 & 1 & 1 \\
    + & & & & & & 1 \\
    \end{array}
    \]

    \[
    \hline
    \end{array}
    \]
Step 2

Increasing a binary number $b$ done as follows:

- **Case number consists of 1 only:**
  - I.e. $b = (111 \cdots 111)_2$.
  - \[ b + 1 = (1000 \cdots 000)_2. \]
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.
  - That’s what happens when we add by hand:

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
+ & & & & & & 1 & \\
\hline
& & & & & & 1 & 0
\end{array}
\]
Step 2

Increasing a binary number $b$ done as follows:

- **Case number consists of 1 only:**
  - I.e. $b = (111 \cdots 111)_2$.
  - $b + 1 = (1000 \cdots 000)_2$.
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.
  - That’s what happens when we add by hand:
Step 2

Increasing a binary number $b$ done as follows:

- **Case number consists of 1 only:**
  - I.e. $b = (111 \cdots 111)_2$.
  - $b + 1 = (1000 \cdots 000)_2$.
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.
  - That’s what happens when we add by hand:
Step 2

Increasing a binary number $b$ done as follows:

- **Case number consists of 1 only:**
  - I.e. $b = (111 \cdots 111)_2$.
  - \( b + 1 = (1000 \cdots 000)_2 \).
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.
  - That’s what happens when we add by hand:

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 \\
+ & & & & & & 1 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{array}
\]
Step 2

Increasing a binary number $b$ done as follows:

- **Case number consists of 1 only:**
  - I.e. $b = (111 \cdots 111)_2$.
  - $b + 1 = (1000 \cdots 000)_2$.
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.
  - That’s what happens when we add by hand:

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 \\
+ & & & & & & 1 \\
\hline
1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0
\end{array}
\]
Step 2

Increasing a binary number $b$ done as follows:

- **Case number consists of 1 only:**
  - i.e. $b = (111 \cdots 111)_2$.
  - $b + 1 = (1000 \cdots 000)_2$.
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.
  - That’s what happens when we add by hand:

\[
\begin{array}{ccccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
+ & & & & & & 1 \\
\hline
1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}
\]
Step 2

Increasing a binary number $b$ done as follows:

- **Case number consists of 1 only:**
  - I.e. $b = (111 \cdots 111)_2$.
  - $b + 1 = (1000 \cdots 000)_2$.
  - Obtained by replacing all ones by zeros and then replacing the first blank symbol by 1.

That’s what happens when we add by hand:

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
+ & & & & & & & \\
\hline
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Otherwise:

- Then the representation of the number contains at the end one 0 followed by ones only.
  Includes case where the least significant digit is 0.
  - Example 1: \( b = (010001011) \), one 0 followed by 3 ones.
  - Example 2: \( b = (0100010010) \), least significant digit is 0.

- Let \( b = (b_0 b_1 \cdots b_k 0 111 \cdots 111)_2 \).
  \( / \) times

- \( b + 1 \) obtained by replacing the final block of ones by 0 and the 0 by 1:
  \( b + 1 = (b_0 b_1 \cdots b_k 1000 \cdots 000)_2 \).
  \( / \) times
Step 2 – General Situation

- We have to replace, as long as we find ones, the ones by zeros, and move left, until we encounter a 0 or a □, which is replaced by a 1.
- So we need a new state $s_2$, and the following instructions:
  
  $$(s_1, 1, s_1, 0, L)$$
  $$(s_1, 0, s_2, 1, L)$$
  $$(s_1, □, s_2, 1, L)$$

- At the end the head will be one field to the left of the 1 written, and the state will be $s_2$. 
Step 2 – General Situation

- We have to replace, as long as we find ones, the ones by zeros, and move left, until we encounter a 0 or a □, which is replaced by a 1.

- So we need a new state $s_2$, and the following instructions:
  - $(s_1, 1, s_1, 0, L)$
  - $(s_1, 0, s_2, 1, L)$
  - $(s_1, □, s_2, 1, L)$

- At the end the head will be one field to the left of the 1 written, and the state will be $s_2$.

```
   ... 1 0 1 0 0 1 0 0 1 1 1  ... 
   ↑   ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ↓  ...
Step 2 – General Situation

- We have to replace, as long as we find ones, the ones by zeros, and move left, until we encounter a 0 or a $\sqcup \sqcup$, which is replaced by a 1.
- So we need a new state $s_2$, and the following instructions
  
  $$(s_1, 1, s_1, 0, L)$$  
  $$(s_1, 0, s_2, 1, L)$$  
  $$(s_1, \sqcup \sqcup, s_2, 1, L)$$

- At the end the head will be one field to the left of the 1 written, and the state will be $s_2$.  

```
⋯ | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | ⋯
```

↑  

$s_1$
Step 2 – General Situation

- We have to replace, as long as we find ones, the ones by zeros, and move left, until we encounter a 0 or a \(\text{⌞⌟}\), which is replaced by a 1.
- So we need a new state \(s_2\), and the following instructions

\[
\begin{align*}
(s_1, 1, s_1, 0, L) \\
(s_1, 0, s_2, 1, L) \\
(s_1, \text{⌞⌟}, s_2, 1, L)
\end{align*}
\]

- At the end the head will be one field to the left of the 1 written, and the state will be \(s_2\).
We have to replace, as long as we find ones, the ones by zeros, and move left, until we encounter a 0 or a □, which is replaced by a 1.

So we need a new state $s_2$, and the following instructions:

- $(s_1, 1, s_1, 0, L)$
- $(s_1, 0, s_2, 1, L)$
- $(s_1, □, s_2, 1, L)$

At the end the head will be one field to the left of the 1 written, and the state will be $s_2$. 

```
... 1 0 1 0 0 1 0 0 0 0 0 0 0 ...

↑

S_1
```
Step 2 – General Situation

- We have to replace, as long as we find ones, the ones by zeros, and move left, until we encounter a 0 or a □□, which is replaced by a 1.
- So we need a new state $s_2$, and the following instructions:
  
  $(s_1, 1, s_1, 0, L)$
  $(s_1, 0, s_2, 1, L)$
  $(s_1, □□, s_2, 1, L)$

- At the end the head will be one field to the left of the 1 written, and the state will be $s_2$. 

$\begin{array}{cccccccccccc}
\cdots & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & \cdots \\
\end{array}$

\uparrow

$s_2$
Finally, we have to move the most significant bit, which is done as follows:

- \((s_2, 0, s_2, 0, L)\)
- \((s_2, 1, s_2, 1, L)\)
- \((s_2, \bot, s_3, \bot, R)\)

The program terminates in state \(s_3\).
Step 3

Finally, we have to move the most significant bit, which is done as follows

\[(s_2, 0, s_2, 0, L)\]
\[(s_2, 1, s_2, 1, L)\]
\[(s_2, \_, \_, s_3, \_, R)\]

The program terminates in state \(s_3\).
Finally, we have to move the most significant bit, which is done as follows:

\[(s_2, 0, s_2, 0, L)
(s_2, 1, s_2, 1, L)
(s_2, \_\_, s_3, \_\_, R)\]

The program terminates in state \(s_3\).
Finally, we have to move the most significant bit, which is done as follows:

\[(s_2, 0, s_2, 0, L)\]
\[(s_2, 1, s_2, 1, L)\]
\[(s_2, \_\_\_\_, s_3, \_\_\_\_, R)\]

The program terminates in state \(s_3\).
Finally, we have to move the most significant bit, which is done as follows:

\[(s_2, 0, s_2, 0, L)\]
\[(s_2, 1, s_2, 1, L)\]
\[(s_2, \_\_, s_3, \_\_, R)\]

The program terminates in state \(s_3\).
Finally, we have to move the most significant bit, which is done as follows

\[(s_2, 0, s_2, 0, L)\]
\[(s_2, 1, s_2, 1, L)\]
\[(s_2, \text{~}, s_3, \text{~}, R)\]

The program terminates in state \(s_3\).

```
... 0 1 0 0 1 0 0 1 0 1 0 0 0 ...
```

↑

\(s_2\)
Finally, we have to move the most significant bit, which is done as follows

\[(s_2, 0, s_2, 0, L)\]
\[(s_2, 1, s_2, 1, L)\]
\[(s_2, \_\_, s_3, \_\_, R)\]

The program terminates in state \(s_3\).
Finally, we have to move the most significant bit, which is done as follows

\[(s_2, 0, s_2, 0, L)\]
\[(s_2, 1, s_2, 1, L)\]
\[(s_2, \_\_\_, s_3, \_\_\_, R)\]

The program terminates in state \(s_3\).
Finally, we have to move the most significant bit, which is done as follows:

\[(s_2, 0, s_2, 0, L)\]
\[(s_2, 1, s_2, 1, L)\]
\[(s_2, \_\_\_, s_3, \_\_\_, R)\]

The program terminates in state \(s_3\).
Step 3

Finally, we have to move the most significant bit, which is done as follows

\[(s_2, 0, s_2, 0, L)\]
\[(s_2, 1, s_2, 1, L)\]
\[(s_2, \_\_\_, s_3, \_\_\_, R)\]

The program terminates in state \(s_3\).
The complete TM is as follows:

\[
\left(\{0, 1, \sqcup, \sqcap\}, \\{s_0, s_1, s_2, s_3\}, \\{(s_0, 0, s_0, 0, R), (s_0, 1, s_0, 1, R), (s_0, \sqcup, s_1, \sqcup, L), (s_1, 1, s_1, 0, L), (s_1, 0, s_2, 1, L), (s_1, \sqcup, s_2, 1, L), (s_2, 0, s_2, 0, L), (s_2, 1, s_2, 1, L), (s_2, \sqcup, s_3, \sqcup, R)\}, \sqcup, s_0\right)
\]
Complete TM

- State s0:
  - 0/0, R
  - 1/1, R

- State s1:
  - 0/0, R
  - 1/1, L

- State s2:
  - 0/0, L
  - 1/1, R

- State s3:
  - B/B, R
Notation: bin

- TMs usually operate on binary numbers.
- Therefore we define for a natural number \( \text{bin}(n) \) as the string in \( \{0, 1\}^* \) representing the unique normalised binary representation of \( n \).
- **Normalised** means that the string has no leading zeros, except for the string "0" representing 0.
- Furthermore the empty string is not normalised (but is considered as a non-normalised representation of 0).
- Examples:
  - \( \text{bin}(0) = "0" \),
  - \( \text{bin}(1) = "1" \),
  - \( \text{bin}(2) = "10" \),
  - \( \text{bin}(3) = "11" \),
  - \( \text{bin}(4) = "100" \), etc.
IV.3 (a) Definition of the Turing Machine

Function Computed by a TM

Definition (3.1)

Let $T = (\Sigma, S, I, \bot, s_0)$ be a Turing machine with $\{0, 1\} \subseteq \Sigma$. Define for every $k \in \mathbb{N}$ $T^{(k)} : \mathbb{N}^k \hookrightarrow \mathbb{N}$, where $T^{(k)}(a_0, \ldots, a_{k-1})$ is computed as follows:

- **Initialisation:**
  - We write on the tape $\text{bin}(a_0) \bot \text{bin}(a_1) \bot \cdots \bot \text{bin}(a_{k-1})$.
  - E.g. if $k = 3$, $a_0 = 0$, $a_1 = 3$, $a_2 = 2$ then we write $0 \bot 11 \bot 10$.
  - All other cells contain $\bot$.
  - The head is at the left most bit of the arguments written on the tape.
  - The state is set to $s_0$.

- **Iteration:** Run the TM, until it stops.
Definition (Cont) (3.1)

- **Output:**
  - **Case 1:** The TM stops. Only finitely many cells are non-blank.
  
  Let tape, starting from the head-position, contain $b_0 b_1 \cdots b_{k-1} c$ where $b_i \in \{0, 1\}$ and $c \not\in \{0, 1\}$.

  $(k$ might be $0)$.

  Let

  $$ a = (b_0 b_1 \cdots b_{k-1})_2 $$

  Then

  $$ T^{(k)}(a_0, \ldots, a_{k-1}) \simeq a $$
**Example:** Let $\Sigma = \{0, 1, a, b, \sqcup\}$ where 0, 1, a, b, $\sqcup$ are different.

- If the tape starting with the head is as follows:
  - $01001\sqcup\sqcup0101\sqcup\sqcup$
  - or $01001a\sqcup\sqcup$,

  output is $(01001)_2 = 9$.

- If tape starting with the head is as follows:
  - $ab\sqcup\sqcup$
  - or $a$,
  - or $\sqcup$,

  the output is $(\_\_\_)_2 = 0$.

**Definition (Cont) (3.1)**

- **Case 2:** Otherwise.

Then $T^{(k)}(a_0, \ldots, a_{k-1}) \uparrow$, i.e. $T^{(k)}(a_0, \ldots, a_{k-1}) \simeq \bot$. 
Remark

- If the TM terminates with the head in the middle of a binary number, only the portion of this number starting with the head counts.

- Example: Assume the TM terminates with the following configuration:

  1 0 1 1

  Then the output is \((011)_2\) which is 3.
Definition Turing Computable Function

**Definition (3.2)**

\[ f : \mathbb{N}^k \sim \mathbb{N} \text{ is Turing-computable}, \text{ in short TM-computable}, \text{ if } f = T^{(k)} \text{ for some TM } T, \text{ the alphabet of which contains } \{0, 1\} \].

**Example:** That succ : \( \mathbb{N} \sim \mathbb{N} \) and zero : \( \mathbb{N} \sim \mathbb{N} \) are Turing-computable was shown above.
zero can be defined in a simpler way by defining a TM which writes a blank and moves right, then moves back (left) and stops with the head pointing to this blank:

The final state of this TM, run with input some binary number, is as follows (\( x \) is 0, 1 or \( \_\)):
Simpler Solution for zero

- The output of $T^{(1)}(x)$ is the value of largest binary string in the final configuration starting with the head position.
- This string is the empty string, which is interpreted as 0.
Even Simpler Solution

- There are even simpler TMs for defining zero:
  - One which uses only 2 states.
  - and one which uses only 1 state.
Remark

- If the tape of the Turing machine initially contains only finitely many cells which are not blank, then at any step during the execution of the TM only finitely many cells are non blank.
  - Follows since in each step at most one cell can be modified to become non-blank.
  - So in finitely many steps only finitely many cells can be converted from blank to non-blank.
IV.3 (a) Definition of the Turing Machine

IV.3 (b) Equivalence of URM computable and Turing computable functions

IV.3 (c) Undecidability of the Turing Halting Problem

IV.3 (d) The Church-Turing Thesis
(b) Equivalence of URM computable and Turing computable functions

Theorem (3.3)

\[ f : \mathbb{N}^n \xrightarrow{\sim} \mathbb{N} \text{ is URM-computable iff it is Turing-computable by a TM with alphabet } \{0, 1, \sqsubseteq\}. \]
Proof Idea URM-Computable $\Rightarrow$ TM-Computable

The idea that URM computable functions are TM computable is as follows:

- A URM changes only finitely many registers.
- Therefore it suffices to simulate a URM with only finitely many registers $R_0, \ldots, R_n$.
- If $R_0, \ldots, R_n$ contain values $x_0, \ldots, x_n$, then this state of the URM can be represented by having

  $$ \text{bin}(x_0) \quad \text{bin}(x_1) \quad \cdots \quad \text{bin}(x_n) $$

  on the tape (surrounded by blanks) and the head pointing to the left most digit of bin($x_0$).

- We can now write TM instructions which take this configuration and executes one URM instruction.
Proof Idea URM-Computable $\Rightarrow$ TM-Computable

- For $\text{succ}(k)$ can be simulated by
  - moving the head to the $k$th number
  - incrementing it by 1
  - moving the head back to the left most digit of the first number,
  - and continuing with the simulation of the next instruction following this instruction
    (or terminating if there is no such instruction).

- It might happen that the number of digits of the number incremented increases.
  - In this case first shift the numbers to the left once to the left.
Proof Idea URM-Computable $\Rightarrow$ TM-Computable

- $\text{pred}(k)$ can be simulated similarly.
- $\text{ifzero}(k, l)$ can be simulated by checking whether the $k$th number is zero or not.
  - If it is zero continue executing the simulation of instruction $l$.
  - If it is not zero continue executing the next instruction.
  - If in one of these cases the instruction doesn’t exist, terminate.
Proof Idea URM-Computable $\Rightarrow$ TM-Computable

- Let the original URM be $U$ and the resulting TM be $T$.
- $T^{(k)}$ will write the arguments in binary on the tape.
  - The arguments will just be written in the register positions.
- Then $T$ will simulate $U$.
- $T$ will terminate iff $U$ terminates.
- If $T$ terminates $T^{(k)}(x_0, \ldots, x_{k-1})$ will return the binary value of the first number of the tape which is the content of $R_0$ and therefore the output of the $U^{(k)}(x_0, \ldots, x_{k-1})$.
- So $T^{(k)}$ and $U^{(k)}$ return the same results.
- Details can be found in the proof of Lemma 3.4 below.
Proof Idea $\text{TM-Computable} \Rightarrow \text{URM-Computable}$

- At any time during the execution of a TM only a finite portion of the tape is non-blank.
- Therefore the state of a TM can be encoded by giving:
  - the finite portion of the tape which is non-blank,
  - the position of the head in this portion,
  - the state of the TM
- There are techniques for encoding this in a computable way as a natural number.
- Now simulate the TM by a URM in a similar way as the simulation of a URM by a TM.
Formal Proof

A formal proof of one direction (URM-computable functions are Turing computable) can be found in the additional material (Lemma 3.4).
Extension to Arbitrary Alphabets

- Let $A$ be a finite alphabet s.t. $\square \not\in A$, and $B := A^*$.
- To a Turing machine $T = (\Sigma, S, I, \square, s_0)$ with $A \subseteq \Sigma$ corresponds a partial function $T^{(A,n)} : B^n \rightarrow B$, where $T^{(A,n)}(a_0, \ldots, a_{n-1})$ is computed as follows:
  - Initially write $a_0 \square \cdots \square a_{n-1}$ on the tape, otherwise $\square \square$.
  - Start in state $s_0$ on the left most position of $a_0$.
  - Iterate TM as before.
  - In case of termination, the output of the function is $c_0 \cdots c_{l-1}$, if the tape contains, starting with the head position $c_0 \cdots c_{l-1}d$ with $c_i \in A$, $d \not\in A$.
  - Otherwise, the function value is undefined.
In order to introduce the notion of Turing-decidable we need to remind us of the following definition:

Let \( M \subseteq \mathbb{N}^n \) be a predicate. The characteristic function \( \chi_M : \mathbb{N}^n \to \mathbb{N} \) for \( M \) is defined as follows:

\[
\chi_M(\vec{x}) := \begin{cases} 
1 & \text{if } M(\vec{x}) \text{ holds,} \\
0 & \text{otherwise}
\end{cases}
\]

(Here \( \vec{x} \) stands for arguments \( x_1, \ldots, x_n \)).

If we treat true as 1 and false as 0, then the characteristic function is nothing but the Boolean valued function which decides whether \( M(\vec{x}) \) holds or not:

\[
\chi_M(\vec{x}) = \begin{cases} 
\text{true} & \text{if } M(\vec{x}) \text{ holds,} \\
\text{false} & \text{otherwise}
\end{cases}
\]
Extension to Arbitrary Alphabets

- Notion is modulo encoding of $A^*$ into $\mathbb{N}$ equivalent to the notion of Turing-computability on $\mathbb{N}$.
- However, when considering complexity bounds, this notation might be more appropriate.
  - Avoids encoding/decoding into $\mathbb{N}$. 
IV.3 (b) Equivalence of URM computable and Turing computable functions

Turing-Computable Predicates

- A predicate $A$ is Turing-decidable, iff $\chi_A$ is Turing-computable.
- Instead of simulating $\chi_A$
  - means to write the output of $\chi_A$ (a binary number 0 or 1) on the tape.
  - it is more convenient, to take TM with two additional special states $s_{\text{true}}$ and $s_{\text{false}}$ corresponding to truth and falsity of the predicate.
Then a predicate is Turing decidable, if, when we write initially the inputs as before on the tape and start executing the TM, it always terminates in $s_{\text{true}}$ or $s_{\text{false}}$, and it terminates in $s_{\text{true}}$, iff the predicate holds for the inputs, and in $s_{\text{false}}$, otherwise.

The latter notion is equivalent to the first notion.

Usually the latter one is taken as basis for complexity considerations.
IV.3 (a) Definition of the Turing Machine

IV.3 (b) Equivalence of URM computable and Turing computable functions

IV.3 (c) Undecidability of the Turing Halting Problem

IV.3 (d) The Church-Turing Thesis
Undecidability of the Halting Problem first proved 1936 by Alan Turing.

In this Section, we will identify computable with Turing-computable.

This will later be justified by the Church-Turing thesis.
History of Computability Theory

Alan Mathison Turing  
(1912 – 1954)  
Introduced the Turing machine.  
Proved the undecidability of the Turing-Halting problem.
Definition (3.5)

(a) A **problem** is an $n$-ary predicate $M(\vec{x})$ of natural numbers, i.e. a property of $n$-tuples of natural numbers.

(b) A problem (or predicate) $M$ is (Turing-)**decidable**, if the **characteristic function** $\chi_M$ of $M$ is (Turing-)computable.

(The characteristic function $\chi_M$ was defined at the End of Subsect. 3 (b)).
Example of Decidable Problems

- The binary predicate

\[
\text{Multiple}(x, y) : \iff x \text{ is a multiple of } y
\]

is a predicate and therefore a problem.

- \(\chi_{\text{Multiple}}(x, y)\) decides, whether \(\text{Multiple}(x, y)\) holds (then it returns 1 for yes), or not:

\[
\chi_{\text{Multiple}}(x, y) = \begin{cases} 
1 & \text{if } x \text{ is a multiple of } y, \\
0 & \text{if } x \text{ is not a multiple of } y.
\end{cases}
\]

- \(\chi_{\text{Multiple}}\) is intuitively computable, therefore Multiple is decidable.
Need of Encoding of TMs

- We want to show that it is not decidable whether a Turing Machine terminates or not.
- For this we need to be able to talk about programs which have as input a Turing Machine.
- For this we need to give a formalisation of what a Turing Machine is.
- Since we are restricting ourselves to functions having as arguments elements of \( \mathbb{N}^k \), we need to encode a TM as an element of \( \mathbb{N}^k \) for some \( k \).
- We will actually encode TMs as elements of \( \mathbb{N} \).
A Turing Machine is a quintuple (or five-tuple) \((\Sigma, S, \iota, \downarrow, s_0)\).

We can assume that \(\downarrow\), each symbol of the alphabet, and each state can be represented by a string of letters and numbers.

Then this quintuple can be written as a string of ASCII-symbols.

Turing machines can be represented as elements of \(A^*\), where \(A\) = set of ASCII-symbols.

There are computable functions, which allow to encode strings as natural numbers and corresponding computable decoding functions.

- Taught in an extended module on computability theory.

Turing machines can be encoded as natural numbers.

- Of course more efficient encoding exist.
Let for a Turing machine $T$, $\text{encode}(T) \in \mathbb{N}$ be its code.

It is intuitively decidable, whether a string of ASCII symbols is a Turing machine.
- One can show that this can be decided by a Turing machine.

$\Rightarrow$ It is intuitively decidable, whether $n = \text{encode}(T)$ for a Turing machine $T$. 
Assume $e \in \mathbb{N}$. We define a partial function $\{e\}^k : \mathbb{N}^k \sim \mathbb{N}$, by

$$\{e\}^k(\vec{x}) \sim \begin{cases} m & \text{if } e = \text{encode}(T) \text{ for some Turing machine } T \\
\quad \text{and } T^{(k)}(\vec{x}) \sim m, \quad & \perp \quad \text{otherwise.} \end{cases}$$

So if $e = \text{encode}(T)$, $\{e\}^k = T^{(k)}$.

- Roughly speaking, $\{e\}^k$ is the function computed by the $e$th Turing machine.
- So for every computable (more precisely Turing-computable) function $f : \mathbb{N}^k \sim \mathbb{N}$ there exists an $e$ s.t. $f = \{e\}^k$. 
The notation $\{e\}_k$ is due to Stephen Kleene.

$\{\}$ are called **Kleene-Brackets**.

We write $\{e\}$ for $\{e\}^1$. 

\[ \{e\}_k \]
Stephen Cole Kleene

Probably the most influential computability theorist up to now. Introduced the partial recursive functions.
Definition of the Halting Problem

Definition (3.6)

The Halting Problem is the following binary predicate:

\[ \text{Halt}(e, n) \iff \{e\}(n) \downarrow \]

We will show that Halt is undecidable.
Example

- Let \( e = \text{encode}(T) \), where \( T \) is the Turing machine \( T \) which translates the URM program consisting of only one instruction

\[
I_0 = \text{ifzero}(0, 0)
\]

- If this TM is run with arguments written on the tape, it loops if the first argument is 0, and terminates otherwise with its first argument unchanged.

- So we have

\[
\{ e \}(k) \simeq T^{(1)}(k) \simeq \begin{cases} 
  k & \text{if } k > 0 \\
  \bot & \text{otherwise.}
\end{cases}
\]

- Therefore \( \text{Halt}(e, k) \) holds for \( k > 0 \) and does not hold for \( k = 0 \).
Question

If we fix $e = \text{encode}(T)$ for the Turing machine above, can we decide, for which $k$ we have that $\text{Halt}(e, k)$ holds?
Remark

- Below we will see: Halt is undecideable.
- However, the following function WeakHalt is computable:

\[
\text{WeakHalt}(e, n) \equiv \begin{cases} 
1 & \text{if } \{e\}(n) \downarrow \\
\bot & \text{otherwise}
\end{cases}
\]

- Computed as follows:
  First check whether \( e = \text{encode}(T) \) for some Turing machine \( T \).
  If not, enter an infinite loop.
  Otherwise, simulate \( T \) with input \( n \).
  If simulation stops, output 1, otherwise the program loops for ever.
Question

- What is \(\text{WeakHalt}(e, n)\), where \(e\) is a code for the Turing machine corresponding to the URM program

\[ I_0 = \text{ifzero}(0, 0) ? \]
Theorem (3.7)

The halting problem is undecidable.

Proof:

- **Assume** the Halting problem were decidable i.e. assume that we can decide using a Turing machine whether \{e\}(n)↓ holds.
- We will define below a computable function \( f : \mathbb{N} \rightarrow \mathbb{N} \), s.t. for all \( e \in \mathbb{N} \) we have \( f \neq \{e\} \).
- Therefore \( f \) cannot be computed by the Turing machine with code \( e \) for any \( e \), i.e. \( f \) is noncomputable.
- Therefore we obtain a **contradiction**.
Proof of Theorem 3.7

We define $f(e)$ in such a way that $f \neq \{e\}$ since $f(e) \not\in \{e\}(e)$.

- If $\{e\}(e)\downarrow$, then we let $f(e)\uparrow$.
- If $\{e\}(e)\uparrow$, we let $f(e)\downarrow$, e.g. by defining $f(e) \simeq 0$ (any other defined result would be appropriate as well).
- So we define

$$f(e) \simeq \begin{cases} \bot, & \text{if } \{e\}(e)\downarrow \\ 0, & \text{if } \{e\}(e)\uparrow \end{cases} \simeq \begin{cases} \bot, & \text{if } \text{Halt}(e, e) \\ 0, & \text{if } \neg\text{Halt}(e, e) \end{cases}$$
Proof of Theorem 3.7

\[
f(e) \sim \begin{cases} 
\bot, & \text{if } \{e\}(e) \downarrow \\
0, & \text{if } \{e\}(e) \uparrow 
\end{cases} \sim \begin{cases} 
\bot, & \text{if } \text{Halt}(e, e) \\
0, & \text{if } \neg \text{Halt}(e, e)
\end{cases}
\]

- Since we assumed Halt to be decidable, \( f \) is computable (Exercise: show that \( f \) is computable by a Turing machine, assuming a Turing machine for Halt).
- Furthermore \( f(e) \downarrow \iff \{e\}(e) \uparrow \), therefore \( f \neq \{e\} \).
- But then \( f \) is not computable, since if it were computable it would be computable by a TM with code \( e \), so would have \( f = \{e\} \) for some \( e \).
- So we obtain a contradiction, and obtain therefore that the assumption that Halt is decidable was false.
Proof of Theorem 3.7

The complete proof on one slide is as follows:

- Assume Halt were decidable.
- Define
  \[ f(e) \equiv \begin{cases} \perp, & \text{if } \{e\}(e) \downarrow \\ 0, & \text{if } \{e\}(e) \uparrow \end{cases} \]
- By Halt decidable, we obtain \( f \) is computable, so \( f = \{e\} \) for some \( e \).
- But then
  \[ f(e) \downarrow \iff \text{Def of } f \implies \{e\}(e) \uparrow \iff f = \{e\} \implies f(e) \uparrow \]
Remark

- The above proof can easily be adapted to any reasonable programming language, in which one can define all intuitively computable functions.
- Such programming languages are called Turing-complete languages.
  - Babbage’s machine was, if one removes the restriction to finite memory, Turing-complete, since it had a conditional jump.
- For standard Turing complete languages, the unsolvability of the Turing-halting problem means: it is not possible to write a program, which checks, whether a program on given input terminates.
IV.3 (a) Definition of the Turing Machine

IV.3 (b) Equivalence of URM computable and Turing computable functions

IV.3 (c) Undecidability of the Turing Halting Problem

IV.3 (d) The Church-Turing Thesis
We have introduced two models of computations:

- The URM-computable functions.
- The Turing-computable functions.

Further we have indicated why the two models of computation compute the same partial functions.
Lots of other models of computation have been studied:

- The partial recursive functions.
- The while programs.
- Symbol manipulation systems by Post and by Markov.
- Equational calculi by Kleene and by Gödel.
- The $\lambda$-definable functions.
- Any of the programming languages Pascal, C, C++, Java, Prolog, Haskell, ML (and many more).
- Lots of other models of computation.
One can show that the partial functions computable in these models of computation are again exactly the Turing computable functions.

So all these attempts to define a complete model of computation result in the same set of partial recursive functions.

Therefore we arrive at the Church-Turing Thesis, also called Church’s thesis.
The Church-Turing Thesis:

The (in an intuitive sense) computable partial functions are exactly the Turing-computable functions

(or equivalently the URM-computable functions or equivalently the functions computable in any other known Turing-complete model of computation).
IV.3 (d) The Church-Turing Thesis

Philosophical Thesis

- This thesis is **not a mathematical theorem**.
- It is a **philosophical thesis**.
- Therefore the Church-Turing thesis **cannot be proven**.
- We can only provide **philosophical evidence** for it.
- This evidence comes from the following **considerations and empirical facts**:
Empirical Facts

- All complete models of computation suggested by researchers define the same set of partial functions.
- Many of these models were carefully designed in order to capture intuitive notions of computability:
  - The Turing machine model captures the intuitive notion of computation on a piece of paper in a general sense.
  - The URM machine model captures the general notion of computability by a computer.
  - Symbolic manipulation systems capture the general notion of computability by manipulation of symbolic strings.
Empirical Facts

- No intuitively computable partial function, which is not partial recursive, has been found, despite lots of researchers trying it.
- A strong intuition has been developed that in principal programs in any programming language can be simulated by Turing machines and URMs.

Because of this, only few researchers doubt the correctness of the Church-Turing thesis.
Decidable Sets

- A predicate $A$ is URM-/Turing-decidable iff $\chi_A$ is URM-/Turing-computable.
- A predicate $A$ is decidable iff $\chi_A$ is computable.
- By the Church-Turing thesis to be computable is the same as to be URM-computable or to be Turing-computable.
- So the decidable predicates are exactly the URM-decidable and exactly the Turing-decidable predicates.
Halting Problem

Because of the equivalence of the models of computation, the halting problem for any of the above mentioned models of computation is undecidable.

Especially it is undecidable, whether a program in one of the programming languages mentioned terminates:

Assume we had a decision procedure for deciding whether or not say a Java program terminates for given input.

Then we could, using a translation of URMs into Java programs, decide the halting problem for URMs, which is impossible.