Computing $\mathbb{N} \to \mathbb{N}$ functions

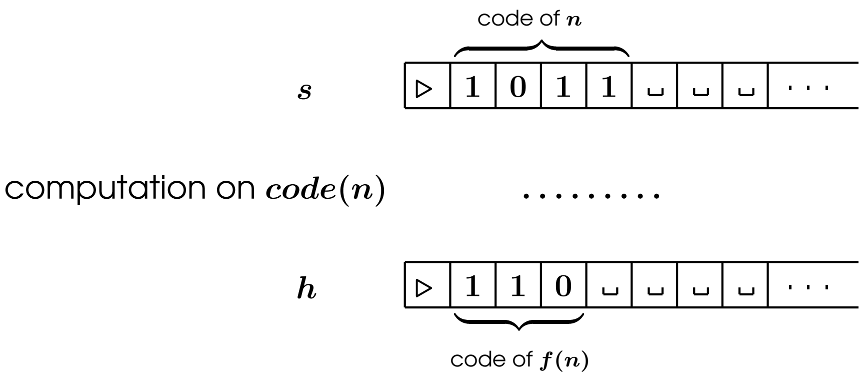
Remember that strings over the alphabet $\{0, 1\}$ can be regarded as **binary codes** for natural numbers.

For example,

$$1101 \text{ codes } 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13.$$

So Turing machines can compute $\mathbb{N} \to \mathbb{N}$ functions

(with both input and output represented in binary)



Example: computing the function $f(n) = n + 1$

Consider the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(n) = n + 1$.

(Then, say, $f(15) = 16$, $f(0) = 1$, $f(3) = 4$, $f(39) = 40, \dots$)

How does it work in **binary**?

$$n = 39 \quad \leadsto \quad \text{code of } n \;=\; \begin{array}{r} 100111 \\ +\qquad 1 \\ \hline \underbrace{101000}_{\text{code of } n+1} \end{array}$$

The Turing machine computing f 'imitates' how we do '$+1$' in binary.

Computing $f(n) = n + 1$: implementation

An implementation-level description of the Turing machine computing f is as follows:

(1) It wants to find the rightmost bit of the input. So it scans the tape to the right until it reaches a blank.

(2) It turns left and flips all $\mathbf{1}$'s to $\mathbf{0}$'s until it reaches the first $\mathbf{0}$.

(3) Then it flips this $\mathbf{0}$ to $\mathbf{1}$ and halts.

(4) If it never finds $\mathbf{0}$ (which means that the input string was all $\mathbf{1}$'s), it writes $\mathbf{1}$ to the leftmost cell, finds the first blank to the right, writes there $\mathbf{0}$, and halts.

The machine should work properly for all binary numbers. But if the input is not 'proper' (say, $\mathbf{00{\sqcup}1}$) then we do not care what the machine does.

Recognising languages

A language L over Σ is **Turing enumerable** (or simply **enumerable**) if there is a Turing machine M such that, for every input word $w \in \Sigma^*$,

- if $w \in L$ then M halts on input w with output 1 (or YES)
- if $w \notin L$ then M never stops on input w

Thus, if $w \in L$ then M will eventually tell us that this is the case; however, if $w \notin L$ then we will never get an answer

We can run M on every $w \in \Sigma^*$ and construct a (possibly infinite) list of words for which M returns 1. In this sense, M **enumerates** L

A language L over Σ is **Turing decidable** (or simply **decidable**) if there is a Turing machine M such that, for every input word $w \in \Sigma^*$,

- if $w \in L$ then M halts on input w with output 1 (or YES)
- if $w \notin L$ then M halts on input w with output 0 (or NO)

Thus, for **any** input w, machine M will eventually tell us whether $w \in L$ or not

Finite languages are always decidable

Consider, for instance, the language $\boxed{L = \{b,\ aa\}}$

It is very easy to design a Turing machine which accepts just these two words and rejects any other word over the alphabet $\{a, b\}$:

- It scans the tape from left to right, and if the first cell has b then
  - if the second is blank, turns to a state '**accept**'
  - if any other than blank comes, turns to a state '**reject**'
- If the first cell has a and the second also a, then
  - if the third is blank, turns to a state '**accept**'
  - if any other than blank comes third, turns to a state '**reject**'

At the end, erases the tape and writes 1 or 0, depending on whether its current state is '**accept**' or '**reject**'

Regular and context-free languages are decidable

This is because any deterministic finite automaton can always be transformed into a Turing machine accepting the same language:

– The first part of the transition table 'simulates' the automaton: e.g., if there is an a-arrow from a state q to a state r then add the line

$$\boxed{q \mid a \parallel r \mid \rightarrow}$$

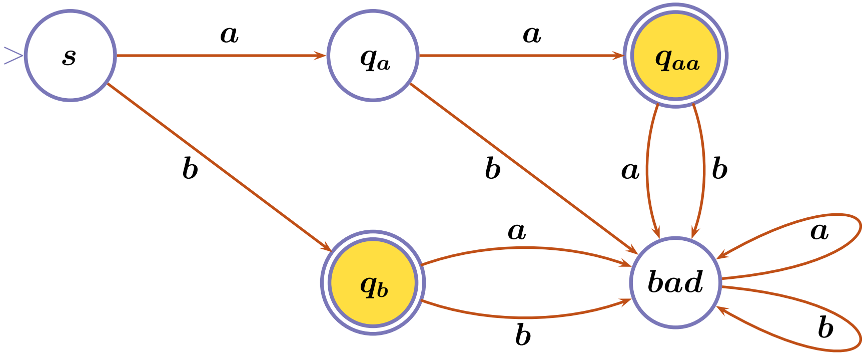
to the transition table of the Turing machine.

– The second part 'takes care' of giving the correct answer:

If the automaton completes reading the input in a favourable state, then the Turing machine turns to a state '**accept**', otherwise to a state '**reject**'. And then it erases the tape and writes 1 or 0, respectively, depending on whether its state is '**accept**' or '**reject**'.

Simulating finite automata: example

Deterministic finite automaton A such that $L(A) = \{b,\ aa\}$:



Turing machine deciding the language $\{b,\ aa\}$ (and 'simulating' A):

s	a	q_a	$\rightarrow$
s	b	q_b	$\rightarrow$
q_{aa}	a	bad	$\rightarrow$

q_{aa}	$\sqcup$	*accept*	$\leftarrow$
q_b	$\sqcup$	*accept*	$\leftarrow$
bad	$\sqcup$	*reject*	$\leftarrow$

accept	a	*accept*	$\sqcup$
accept	$\triangleright$	*accept*$^+$	$\rightarrow$
accept$^+$	$\sqcup$	h	1

$\dots$

Non-context-free languages can also be decidable

Example. Let L consist of all strings of a's whose length is a power of 2:

$$L = \{a^{2^n} \mid n \geq 0\} = \{a,\ aa,\ aaaa,\ aaaaaaaa, \dots\}.$$

L is **NOT** **context-free** (why?)

The tape alphabet of the Turing machine T_L deciding L is

$$\rhd \quad \sqcup \quad a \quad 0 \quad 1$$

We will use the symbol 1 not only for exhibiting the final answer,

but also for some kind of 'marking'.

Idea: We 'mark' (replace by 1) every 2nd a, then mark every 2nd of the remaining a's, then mark every 2nd of the remaining a's, etc. If finally one a remains, **accept**. Otherwise, **reject**.

‘Implementation-level’ description of T_L

On input string w the machine iterates the following steps:

(1) scans the tape from left to right by keeping track whether the number of a’s is one, or odd but > 1, or even, and replaces every second a with 1

(2) if at stage (1) the tape contained a single a, then ‘**accept**’

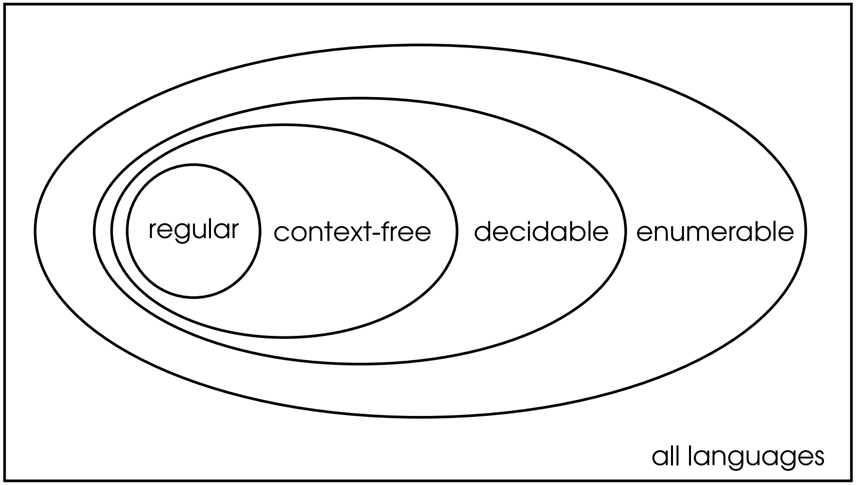
(3) if at stage (1) the number of a’s on the tape was odd and > 1, then ‘**reject**’

(4) return the head to the left end of the tape.

After leaving the loop, the machine is either in state **accept** or in state **reject**. In any case, erase the tape. Then write 1 in case of **accept**,
and 0 in case of **reject**.

Exercise. Give a Turing machine deciding the language $\{a^n b^n c^n \mid n \geq 0\}$

Language hierarchy



- The class of enumerable languages coincides with the class of languages generated by unrestricted grammars

Are there more powerful computational devices than Turing machines?

Extensions of Turing machines

Turing machines are clumsy, wasting a lot of operations on steps that can be carried out by a modern computer in just one step.

We may experiment with many different versions to extend the **computational power** of Turing machines:

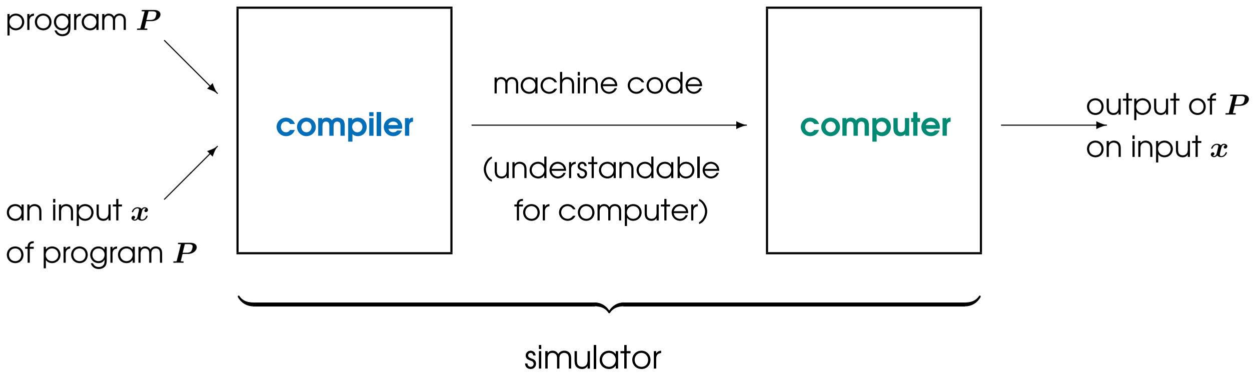
- multiple tape Turing machines
- multiple head Turing machines
- non-deterministic Turing machines
- random access Turing machines

It turns out, however, that none of these attempts yields a model of computation more powerful than the good old one-head one-tape Turing machine

‘Simulating’ programming languages

An important feature of any programming language is that programs in this language can be **‘simulated’:**

they are **compiled** and then **run** on a computer.


program P
an input x of program P
compiler
machine code
(understandable for computer)
computer
output of P on input x
simulator

‘Simulating’ Turing machines

- Like previously finite automata, Turing machines can also be regarded as a (primitive) programming language. So it makes sense to speak about a ‘Turing machine simulator’

- Compiling/simulating programming languages is a difficult computational task. Usually, compilers are written on some sophisticated high-level programming language.

$\rightsquigarrow$ So it is rather surprising that it is possible to design a **Turing machine** that does the job of a ‘Turing machine simulator’:

this machine is called the **Universal Turing machine**

We are about to design this Universal Turing machine...

Universal Turing machine: the problem of inputs

A compiler can take **any** program as input. So the Universal Turing machine should be able to take **any** Turing machine as input.

But Turing machines may have arbitrarily large numbers of states, tape symbols, transitions, while our universal machine (like any Turing machine) must have

- a fixed finite tape alphabet,
- a fixed finite set of states, and
- a fixed finite set of transitions.

We must find a way to **encode** any Turing machine by a string over some fixed **finite** alphabet. Moreover, the Universal Turing machine then must be able to **decode** the code.

Encoding an arbitrary tape alphabet

From the point of view of **what** a Turing machine actually **computes**, any finite tape alphabet Σ having k symbols can be considered as the subset $\{a_1, a_2, a_3, \dots, a_k\}$ of the infinite set

$$\{a_1, a_2, a_3, \dots\}.$$

(What matters for any Turing machine is the **size** of its tape alphabet, rather than the particular **names** of the symbols.)

We can encode **every** symbol by using just three symbols:

$$a,\ 0,\ 1.$$

Each symbol a_i will be encoded as a string $a\bar{i}$, where $\bar{i}$ denotes the binary code of the number i (e.g., $a101$ stands for a_5).

To make decoding easier, we assume that a_1 always represents the marker $\triangleright$, and a_2 always represents the $\llcorner\!\lrcorner$ symbol

Encoding a tape alphabet: example

Consider the Turing machine deciding the language $\{b, aa\}$

(see slides 5–7).

The tape alphabet of this machine consists of the symbols $\triangleright, \sqcup, a, b, 0, 1$.

Symbol:	Symbol renamed:	Code:
$\triangleright$	a_1	$a1$
$\sqcup$	a_2	$a10$
a	a_3	$a11$
b	a_4	$a100$
0	a_5	$a101$
1	a_6	$a110$

Encoding arbitrary states

Again, what matters for any Turing machine is the **number** of its states, rather than the particular **symbols** denoting these states.

So we may assume that the (non-halting) states of any Turing machine actually all belong to the infinite set

$$\{q_1, q_2, q_3, \dots\},$$

q_1 being the initial state.

Using the same trick as before, we can encode **every** state by using just three symbols:

$$q,\ 0,\ 1.$$

Each non-halting state q_i will be encoded as $q\bar{i}$, where $\bar{i}$ denotes the binary code of the number i (e.g., $q1101$ stands for q_{13}).

The halting state will be encoded by the **empty** string.

Encoding states: example

Again, consider the Turing machine deciding the language $\{b, aa\}$.

Its states are: s, q_a, q_{aa}, q_b, bad, $accept$, $reject$, $accept^+$, $reject^+$, h.

Non-halting state:	State renamed:	Code:
s	q_1	$q1$
q_a	q_2	$q10$
q_{aa}	q_3	$q11$
q_b	q_4	$q100$
bad	q_5	$q101$
$accept$	q_6	$q110$
$reject$	q_7	$q111$
$accept^+$	q_8	$q1000$
$reject^+$	q_9	$q1001$

Encoding an arbitrary transition

We use the symbols a, q, 0, and 1, plus the following five symbols:

$$(\qquad) \qquad \Rightarrow \qquad \Leftarrow \qquad ,$$

Transitions are encoded as strings over these symbols. For example,

- transition $\boxed{q_3 \mid a_{10} \parallel q_4 \mid \leftarrow}$ is encoded as $(q11, a1010, q100, \Leftarrow)$
- transition $\boxed{q_5 \mid \sqcup \parallel h \mid a_4}$ is encoded as $(q101, a10, , a100)$
- transition $\boxed{q_8 \mid \triangleright \parallel q_2 \mid \rightarrow}$ is encoded as $(q1000, a1, q10, \Rightarrow)$

Encoding Turing machines and input strings

Now every Turing machine can be represented as a

finite sequence of codes of its transitions

For example, this sequence may start as

$$(q1, a1, q100, \Leftarrow)(q1, a10, q110, a100)(q10, a100, , \Rightarrow)\dots$$

- Given any Turing machine M, we will denote its code by $\boxed{\langle M\rangle}$

Similarly, to code any input string w over a finite k-element alphabet Σ, we first rename the symbols of Σ as $a_3, a_4, \dots, a_{k+2}$, then simply list the codes of the symbols a_i occurring in w: e.g., the string $a_3a_5a_4$ will be coded as $a11a101a100$.

- Given any string w, we will denote its code by $\boxed{\langle w\rangle}$

The ‘encoding machine’

Since, as we have just seen, coding is a simple algorithmic task, it is even possible to design a **Turing machine** T_{code} performing this task:

given **any** input word over the 9 element alphabet

$$a \quad q \quad 0 \quad 1 \quad (\quad) \quad \Rightarrow \quad \Leftarrow \quad ,$$

T_{code} outputs the code of the input word.

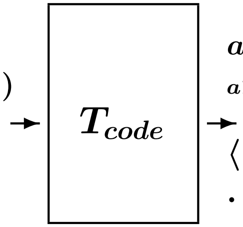
input:

$,)aq1$

$(q1, a11, q10, \Rightarrow)$

$\langle M_{eraser} \rangle$

$\dots$



output:

$a1011a1000a11a100a110$

$a111a100a110a1011a11a110a110a1011a100a110a101a1011a1001a1000$

$\langle\langle M_{eraser} \rangle\rangle$

$\dots$

Universal Turing machine: specification

We are now in a position to specify **what** we want the Universal Turing machine to do:

- The tape alphabet of the universal Turing machine $\boxed{U}$ is

$$\triangleright \quad \sqcup \quad a \quad q \quad 0 \quad 1 \quad (\quad) \quad , \quad \Rightarrow \quad \Leftarrow$$

- A 'legal' input of U is of the form $\boxed{\langle M\rangle\sqcup\langle w\rangle}$ where $\langle M\rangle$ is the code of a Turing machine M, and $\langle w\rangle$ is the code of a string w.

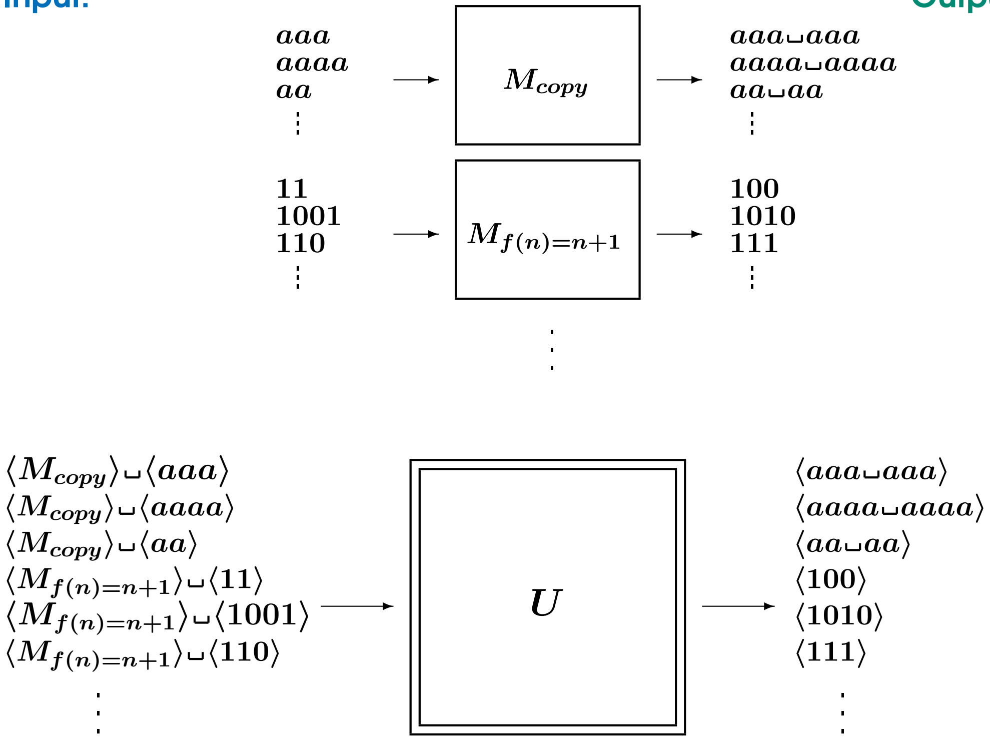
- Given such a legal input, U **simulates** the behaviour of M on input w in the following sense:
  - U halts if and only if M halts on input w, and
  - if U halts then it outputs the code of the output of M running on input w.

  (If the input is not legal, we don't care what U is doing.)

Universal Turing machine: specification (cont.)

Input:

Output:


aaa
$aaaa$
aa
M_{copy}
$aaa\sqcup aaa$
$aaaa\sqcup aaaa$
$aa\sqcup aa$
11
1001
110
$M_{f(n)=n+1}$
100
1010
111
$\langle M_{copy}\rangle\sqcup\langle aaa\rangle$
$\langle M_{copy}\rangle\sqcup\langle aaaa\rangle$
$\langle M_{copy}\rangle\sqcup\langle aa\rangle$
$\langle M_{f(n)=n+1}\rangle\sqcup\langle 11\rangle$
$\langle M_{f(n)=n+1}\rangle\sqcup\langle 1001\rangle$
$\langle M_{f(n)=n+1}\rangle\sqcup\langle 110\rangle$
U
$\langle aaa\sqcup aaa\rangle$
$\langle aaaa\sqcup aaaa\rangle$
$\langle aa\sqcup aa\rangle$
$\langle 100\rangle$
$\langle 1010\rangle$
$\langle 111\rangle$

Universal Turing Machines

For more details on UMTs and further references, consult

`http://en.wikipedia.org/wiki/Universal_Turing_machine`

The Church–Turing thesis

The Church-Turing thesis says that **any algorithmic procedure** that can be carried out by a human being or by a computer
can also be carried out by a Turing machine

This is **not a theorem** that can be **proved**.

Why is this? The Church-Turing thesis establishes a connection between the precise, formal notion of a Turing machine and the intuitive, informal notion of an algorithm.

It cannot be **proved** because it is not, in fact can't be, **stated** in a precise, formal way.

In can rather be considered as a **definition** of the term '**algorithm**':

an algorithmic procedure is what can be carried out by a Turing machine

Evidence for the Church–Turing thesis

Why should we believe that the thesis is true? Here are some arguments:

(1) Turing machines can do many different kinds of algorithmic procedures: compute functions, decide languages, do iterations, simulate finite automata, simulate Turing machines, etc.

In fact,

> no one has yet found an algorithm that happened to be unimplementable by a Turing machine

(2) Extensions with multiple tapes and/or heads, non-determinism, counters, etc. all turned out to have the same computational power as standard Turing machines.

(3) All other suggested theoretical models of computation have turned out to be provably equivalent to Turing machines.

Can the thesis be disproved?

In principle, yes.

It can happen in the future that somebody finds some kind of a procedure

- that looks intuitively algorithmic (not only to the 'finder' but to everybody else), and

- it is possible to prove in a precise way that the procedure cannot be implemented by a Turing machine.

So far no one has yet found a procedure that happened to be widely accepted as an algorithm and unimplementable by a Turing machine at the same time.

An important implication of the thesis

Since the thesis says that any solvable algorithmic problem can be solved by a Turing machine, it opens the possibility to show

(in a mathematically precise way) that

$\boxed{\text{some problems } \textbf{cannot} \text{ be solved by } \textbf{any} \text{ algorithm}}$

Or, in other words, may be

$\boxed{\text{there are problems that } \textbf{no} \text{ computer can solve}}$

– If we believe in the Church–Turing thesis, all we have to show is that there is no Turing machine that can solve the problem in question.

– And if we don't believe? . . . Doesn't matter.

We shall see that there are still problems that **no** computer can solve.

Historical notes

An **algorithm** is a procedure or formula for solving a problem.

The word **‘algorithm’** derives from the name of the Persian mathematician,
Mohammed ibn-Musa al-Khwarizmi,
who was part of the royal court in Baghdad and who lived in 780–850.
Al-Khwarizmi’s work is the likely source for the word **algebra** as well.