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

Remember that strings over the alphabet $\{0, 1\}$ can be regarded as

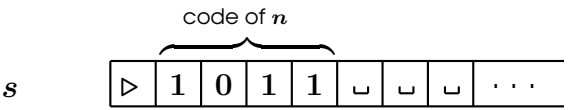
binary codes for natural numbers.

For example,

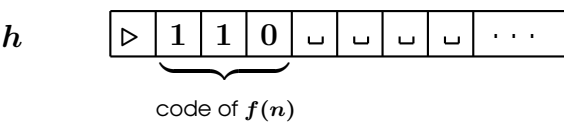
$$1101 \text{ codes } 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13.$$

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

(with both input and output represented in binary)



computation on $code(n)$ .....



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, $f(15) = 16$, $f(0) = 1$, $f(3) = 4$, $f(39) = 40, \ldots$)

How does it work in **binary**?

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

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, $\mathbf{00␣1}$) 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 $L = \{b,\ aa\}$

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

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

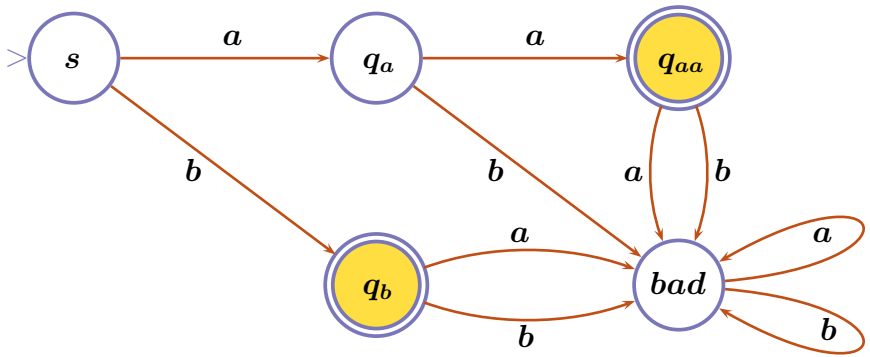
  to the transition table of the Turing machine.

- 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

$\ldots$

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:

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

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

The tape alphabet of the Turing machine T_L deciding L is

$$\triangleright \quad \sqcup \quad a \quad 0 \quad 1$$

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**'

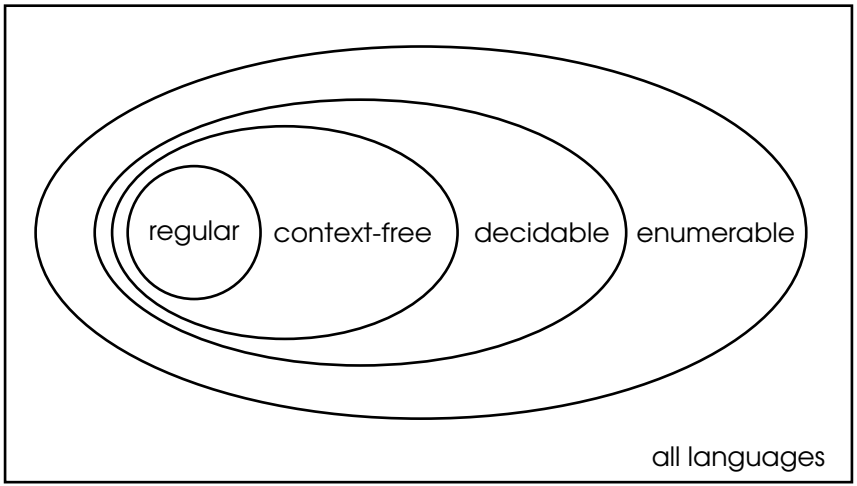
(3) if at stage (1) the number of a's on the tape was odd and >1, then '**reject**'

(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

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

some problems **cannot** be solved by **any** algorithm

Or, in other words, may be

there are problems that **no** 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.

BBC: Chess champion loses to computer

(Tuesday, 5 December 2006, 22:10 GMT)

- Deep Fritz, a chess-playing computer, has beaten human counterpart world chess champion Vladimir Kramnik in a six-game battle in Bonn, Germany.
- Deep Fritz won by four points to two, after taking the last game in 47 moves in a match lasting almost five hours.

In 1948, Turing, working with his former undergraduate colleague, Champernowne, began writing a chess program for a computer that didn't yet exist. In 1952, lacking a computer powerful enough to execute the program, Turing played a game in which he simulated the computer, taking about half an hour per move. The game was recorded; the program lost to Turing's colleague Alick Glennie, although it is said that it won a game against Champernowne's wife.

Quote from a software magazine:

> "Put the right kind of software into a computer,
> and it will do whatever you want it to.
> There may be limits on what you can do with the machines themselves,
> but **there are no limits on what you can do with software**"

Really hard decision problems

Halting problem Is there an algorithm that, given **any** (Java, C++, etc.) program will eventually stop running on a given input **?**

Remember **Collatz**

```
c = n;
while (c != 1) {
    if (c % 2 == 0) { c = c / 2; }
    else { c = 3*c + 1; }
}
```

Diophantine equations Is there an algorithm that can decide whether any given Diophantine equation (a polynomial equation such as $x^n + y^n = z^n$) has a solution in integer numbers **?**

Equivalence of CFGs Is there an algorithm that can decide whether any two context-free grammars define the same language **?**

. . .

The Halting Problem

Decide, for any given Turing machine M and input word w,
whether M will eventually halt on input w

Is there a general algorithmic method solving this problem?

If yes, then — according to the Church–Turing thesis — there should be
a Turing machine solving it.

The following result obtained by A. Turing in 1937 is illustrated by the video at
https://www.youtube.com/watch?v=92WHN-pAFCs

There is **no Turing machine** solving the halting problem

Computers Ltd.?

Does the unsolvability of the Halting Problem **really** mean that the capabilities of computers are limited?

One might say **'the stupid Turing machines can't solve it, but look here, here is an amazingly powerful supercomputer, with an incredibly sophisticated programming language and a very very smart programmer equipped with modern state-of-the-art methodologies, they can surely solve this dumb problem'**

Maybe. But even if you do believe this (which means that you don't believe in the Church–Turing thesis), think about the following:

The amazing new stuff surely would be able to imitate all the 'tricks' we did with old fashioned Turing machines: copying and coding, diagonalisation, etc.

So the very same argument would show that the **Halting Problem of Amazing Supercomputers** is unsolvable by amazing supercomputers.

More unsolvable problems

- **Empty-Tape Halting Problem:**

  Given any Turing machine M, does M halt on the all-blank tape as input?

- **Some-Input Halting Problem:**

  Given any Turing machine M, is there any input string on which M halts?

- **All-Input Halting Problem:**

  Given any Turing machine M, does M halt on every input string?

- **Turing machine Equivalence Problem:**

  Given any two Turing machines M_1 and M_2, do they halt on the same input strings?

- **Tiling Problem:**

  Given any set T of tiles, decide whether an arbitrary large room can be tiled using only the available tile types

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.