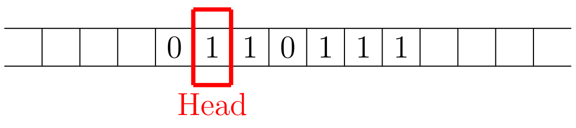
Computability and Turing machines

What computers can and can't do? *Computability* answers this question.
There are problems no computer or algorithm can ever solve. This is a bad news.
To prove such results, we need a more formal definition of computations.

Intuitive definition of computing (Turing, 1936) : a human being is asked a *question* ("input" or "instance", e.g. "123+479=?"), under the form of finitely many symbols from a finite alphabet, written on a sheet of paper.
The (human) computer can write down intermediate results, with unlimited supply of paper, using again symbols from a finite alphabet. The brain follows blindly, non-creatively, a finite list of elementary instructions. The next instruction is decided from the current instruction and the intermediate results currently under scruting by the computer on the paper.
There is an instruction called `STOP`. At this moment, the answer ("output") is written unambiguously on the paper (e.g."602 is the answer").

Formal definition of computing (Turing, 1936) : a *Turing machine* (T.M.) is composed of :

- a *tape*, which is bi-infinite, divided into cells, each being blank or containing a symbol from a finite alphabet A (e.g. $A = \{0, 1\}$);
- a *head*, which reads one cell at a time, can write a new symbol on this cell, shift the tape to the left or right.



- a finite set of instructions, each being numbered $(1, 2, 3, \ldots)$ of the form :
  - `STOP`
  - "`if CurrentCell=blank/0/1`
    `then Write(blank/0/1)`
    `and ShiftLeft/ShiftRight/NoShift`
    `and goto Instruction k.`"

So :

	1936		>1948	
algorithm performed by a human brain	⟷	Turing machine	⟷	modern computer
paper	⟷	tape	⟷	memory
(finite) brain	⟷	head	⟷	CPU
instructions	⟷	instructions	⟷	instructions, software

A *problem* f is a (partial) mapping.

$$\begin{aligned} f :& A^* \to A^* \\ & w \to f(w) \text{ or undefined} \\ A =& \text{ finite alphabet} \\ A^* =& \text{ set of finite words, e.g. } A^* = \{\emptyset, 0, 1, 00, 01, 11, 10, 000, 001, \ldots\} \\ w =& \text{ "input" or "instance"} \\ f(w) =& \text{"output"} \end{aligned}$$

A problem f is *computed* or *solved* by a Turing machine T if whenever we write w on an otherwise blank tape (and head is in initial instruction, at the beginning of w), T runs and eventually stops if and only if $f(w)$ is defined, in which case the tape contains $f(w)$ only (if $f(w)$ is undefined, T runs forever : infinite loop).

The problem f is *computable* if it is solved by at least a Turing machine.

A more *modern machine* : Matlab machine : a Matlab machine is the code of a Matlab function (without "rand" instruction) running on an ideal infinite-memory computer.

Theorem 1
A problem is solved by a Turing machine if and only if it is solved by a Matlab machine.

Proof

$\boxed{\Leftarrow}$ We have to convert any Matlab function into a T.M.=Matlab-to-TM compiler : admitted;

$\boxed{\Rightarrow}$ We have to write a TM-to-Matlab compiler : easier, admitted.

We say that Matlab is "Turing-equivalent". So are Python, C, C^{++} and Java.

Thesis 1 (Church-Turing)
Any "obviously algorithmic" procedure (in the intuitive sense) corresponds to an equivalent T.M./Matlab program.

This is *not* a theorem, because it is not formally defined. It is widely accepted because :

- of Turing's argument about human computer;
- every time we tried to convert an "obviously algorithmic" procedure into a T.M., we succeeded (with sweat and pain).

It says by instance that the Euclidean division algorithm, "obviously algorithmic", can be solved by a T.M./a Matlab program (although the T.M. is not trivial to find).
Of course in this specific case we can prove formally that this T.M. exists, by constructing it.

With Church-Turing thesis, the theorem that Matlab is Turing-equivalent is trivial : Matlab instructions are clearly algorithmic.

Pay attention not confuse it with "strong Church-Turing thesis" : anything the brain can do (writing poetry, proving theorems, ...) can be simulated by a T.M.

This one is highly controversial!

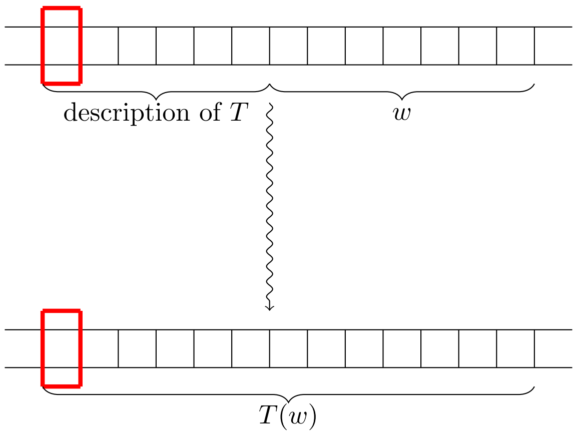
Now that we know what a formal algorithm is, we can prove *theorems* about them.

Universal T.M.

Theorem 2 (Turing, 1936)
The following problem is computable :

- Input *: description of a T.M. T + description of finite word on initial tape, "w";*
- Output *:*
  - *If T stops when initial tape is w,* the finite content of the tape*;*
  - *If t never stops,* undefined.

A T.M. solving it is called a "*universal T.M.*".



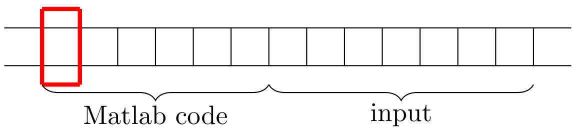
Proof

- construct explicitely universal Turing Machine (thank you Alan)
- OR construct explicitely universal Matlab machine.
- OR by C-T thesis, simulating T is obviously algorithmic (in theintuitive sense) thus there is a Turing Machine doing it.

<u>NB:</u> Variante:

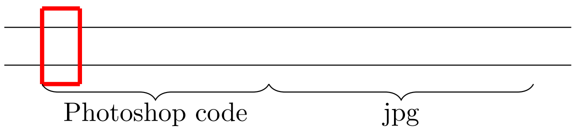
Proof

- There's a Matlab program simulating any other Matlab program on any input.
- There's a Turing Machine simulating any Matlab program on any input.



=Matlab-to-TM interpreter

- Universal Turing Machine means that code is just data, written on memory, on the same level as the input.
  e.g.



Modern computer are universal Turing Machines, i.e., they are (re-)programmable.
$\equiv$ Turing's fundamental insight.
(Pocket) 4-operation calculator = non universal T.M.

The Harting problem is undecidable

A decision problem is a total mapping: $A^* \rightarrow \{\text{YES,NO}\}$
E.g. "Is number n prime?"
A decision problem is decidable if it's computable.
The Harting problem:

Input:

$$\begin{cases} \text{Matlab code f.m.} + \text{input for f} \\ \text{a T.M. description} \end{cases} \tag{1}$$

Output:

- YES if f eventually stops on input x
- NO if f never stops ("infinite loop")