What can 'human computers' do?

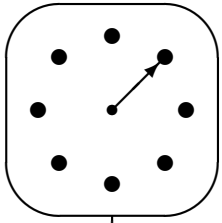
In 1936, **Alan Turing** made an attempt to formulate an abstract model of a **'human computer'** that would use a pencil and paper to solve some problem. Turing tried to decompose the operations of such a 'human computer' into a number of simple steps. He came to the conclusion that

- Any such step would consist of **erasing** a symbol on the paper at the tip of the pencil and **writing** a new one in its place.
- The decision as to which symbol should be written and which symbol should be at the tip of the pencil next would depend only
  - on the **symbol** currently at the tip of the pencil,
  - and the **'state of mind'** of the 'human computer'.

Based on these assumptions, Turing suggested a formal model of computation, known now as a **Turing machine**

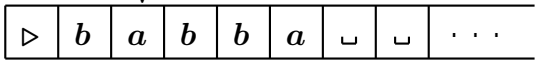
Turing machine

A **theoretical model** for computation, intended to capture the capabilities of **any procedure** or **program**



Finite control device:
at any moment can be in one of the **states**

Head: can **read** and **write**, move **left** and **right**



Input/output tape: it is divided into cells;
each cell may contain a symbol or be blank (␣).

How a Turing machine works

- The machine is supplied with input by inscribing the input string on the tape cells at the left end of the tape. The rest of the tape initially contains blank (␣) symbols. The head scans the leftmost cell of the input.

- At regular time intervals the machine performs two functions in a way
  dependent on the current state of its control device and
  the tape symbol currently scanned by the head:

  (1) Puts the control device in a new state.

  (2) Either (i) writes a symbol in the tape cell currently scanned,
  replacing the one already there

  or (ii) moves the head one tape cell to the left or right.

States and the tape of a Turing machine

States: represent the finite control device. There is always an **initial state** (s), where all computations of the machine start. Since the machine can write on the tape, it can leave an answer (output) on the tape at the end of computation. Therefore we don't need to provide special accepting states. On the other hand, since the head now can move back and forth, we do need a special **halting state** (h) to indicate the end of computation.

Tape: The tape has a left end and can be extended indefinitely to the right.
The leftmost cell contains a special marker $\triangleright$.
When the head scans this symbol, it **always** moves to the right.
(This way the machine never attempts to 'fall' from the left end of the tape.)

Transition function

It is the most important part of a Turing machine that describes its behaviour.

It can be given as a **transition** or **'instruction' table**:

Current state	Symbol scanned	Next state	Task
s	$\sqcup$	h	$\sqcup$
s	a	q	b
q	$\sqcup$	s	$\leftarrow$
s	$\triangleright$	s	$\rightarrow$
q	$\triangleright$	q	$\rightarrow$
q	a	h	a

In the 'Task' column a symbol like a (or $\sqcup$) means

'*replace the scanned symbol on the tape with* a (*or* $\sqcup$)'

$\rightarrow$ and $\leftarrow$ mean, respectively,

'*move the head one cell to the right*' or '*left*'

Turing machine: example 1

States of machine M_{eraser}: s, h, q.

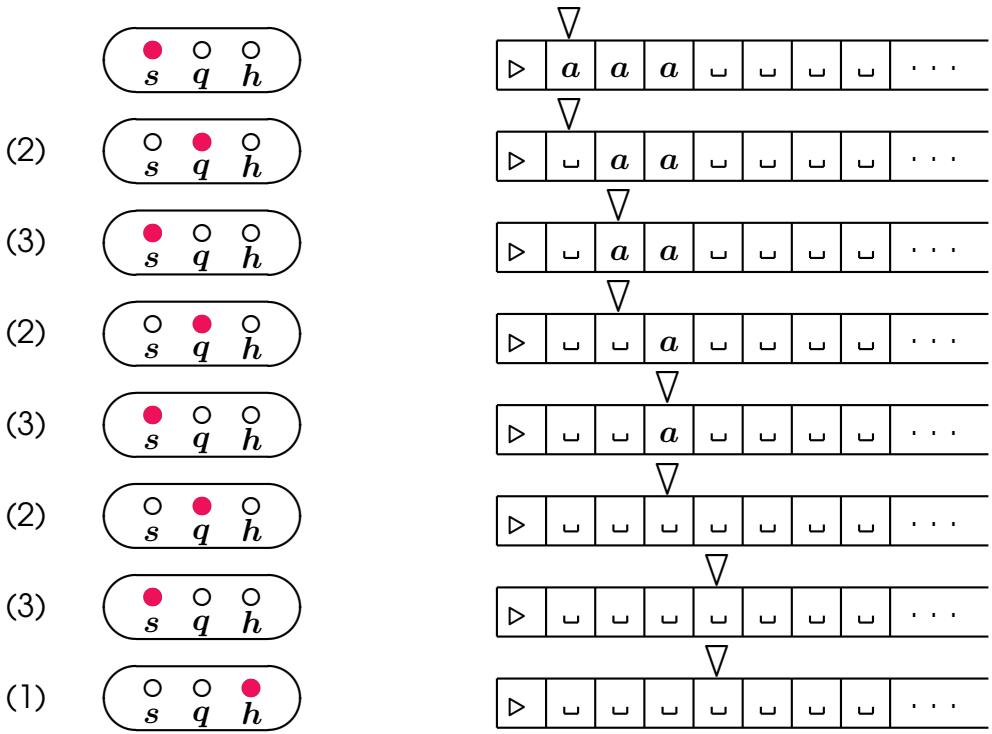
Symbols that can be written on the tape: $\triangleright, \sqcup, a$.

Transition table:

(1)	s	$\sqcup$	h	$\sqcup$
(2)	s	a	q	$\sqcup$
(3)	q	$\sqcup$	s	$\rightarrow$
(4)	s	$\triangleright$	s	$\rightarrow$
(5)	q	$\triangleright$	s	$\rightarrow$
(6)	q	a	h	a

- for each pair (non-halting state, symbol) there is a unique line in the table
- there are no lines starting with h
- if symbol $\triangleright$ is scanned then $\rightarrow$ should be the task to perform

How machine M_{eraser} works



Turing machine: formal definition

A **Turing machine** is a quintuple $M = (Q, \Sigma, \delta, s, h)$ where

- Q is a finite set of **states**, containing the **initial state** s and the **halting state** h,
- Σ is a finite set of symbols, the **tape alphabet**
  (containing the **left end marker** $\triangleright$ and the **blank** $\sqcup$, but **not** $\rightarrow$ and $\leftarrow$),
- δ is a **transition function** mapping
  each pair in $(Q - \{h\}) \times \Sigma$ to a pair in $Q \times ((\Sigma - \{\triangleright\}) \cup \{\rightarrow, \leftarrow\})$
  such that for all non-halting states q, $\delta(q, \triangleright) = (p, \rightarrow)$ for some state p.
  (This last bit guarantees that if M scans $\triangleright$ then the head always moves to the right.)

(non-halting state, tape alphabet symbol) $\overset{\delta}{\mapsto}$ (state, task)

Configurations of a Turing machine

As a Turing machine works, changes occur in the current state, the current tape contents, and the current head location.

A setting of these three items is called a **configuration** of the machine.

Since the input is always finite and the machine can move its head only one cell at a time, after a finite number of steps only finitely many tape cells can contain non-blank symbols. So we can represent a configuration by a pair like

$$(q,\ \triangleright ab\sqcup b\underline{a}a\sqcup\sqcup b)$$

where q is the current state, the current tape contents starts with $\triangleright ab\sqcup b\underline{a}a\sqcup\sqcup b$ and then all blank, and the head is currently scanning the underlined symbol.

A configuration of the form $(h, \dots)$ is called a **halting configuration**.

One step of a Turing machine

Say that configuration C_1 **yields** configuration C_2 **in one step** if the transition table shows that the Turing machine can 'legally go' from C_1 to C_2.

For example,

$(q_5,\ \triangleright u a \underline{b} v)$ yields $(q_7,\ \triangleright u \underline{a} b v)$ in one step

if the transition table of the machine contains the line

q_5	b	q_7	$\leftarrow$

$(s,\ \triangleright x y \underline{\sqcup} z z)$ yields $(q_3,\ \triangleright x y \underline{u} z z)$ in one step

if the transition table of the machine contains the line

s	$\sqcup$	q_3	u

Turing machine computations

The **computation of a Turing machine M on input word w** is the unique sequence of configurations

$$C_0, C_1, C_2, \ldots$$

such that

- C_0 is of the form $(s, \triangleright w)$ (where s is the initial state of M) and the head scans the leftmost symbol of w, and
- every C_i yields C_{i+1} in one step.

The Turing machine M **terminates** (or **halts**) **on input** w if there is a **finite** computation $C_0, C_1, \ldots, C_n$ on w such that

- C_n is a halting configuration.

We say that such a computation is of **length** n or has n **steps**

A computation of M_{eraser}

Recall (pages 6–7) that this machine just **'erases'** the input string from the tape.

For example, having started with the input aaa, M_{eraser} executes a cycle of using transitions (2) and (3) three times. When the tape is all blank, it applies transition (1) to halt.

The seven-step halting computation of M_{eraser} on input aaa is as follows:

$(s, \triangleright\underline{a}aa), (q, \triangleright\underline{\sqcup}aa), (s, \triangleright\sqcup\underline{a}a), (q, \triangleright\sqcup\underline{\sqcup}a), (s, \triangleright\sqcup\sqcup\underline{a}), (q, \triangleright\sqcup\sqcup\underline{\sqcup}),$

$(s, \triangleright\sqcup\sqcup\sqcup\underline{\sqcup}), (h, \triangleright\sqcup\sqcup\sqcup\underline{\sqcup})$

Turing machines are deterministic

Observe that for each non-halting configuration C of a Turing machine M, there is a **unique** configuration C' such that C yields C' in one step.
So starting from a non-halting configuration C_0, the machine M has a **unique** computation

$$C_0, C_1, C_2, \ldots$$

This computation may be **infinite** (when, starting with C_0, M does not terminate).

But if it ends with some configuration C_n, then C_n **must** be a halting configuration. Otherwise, there is always a line in the transition table of M that says how to continue (so a Turing machine never gets stuck).

Turing machine: example 2

Consider the Turing machine $M_2 = (Q, \Sigma, \delta, s, h)$ where

$$Q = \{s, h, q\}, \quad \Sigma = \{\triangleright, \sqcup, \square, \diamond\}$$ and

δ is given by the following transition table:

	s	$\sqcup$	q	$\square$
	s	$\square$	q	$\square$
	s	$\diamond$	h	$\square$
	s	$\triangleright$	s	$\rightarrow$
$(\bullet)$	q	$\sqcup$	q	$\leftarrow$
$(*)$	q	$\square$	q	$\rightarrow$
	q	$\diamond$	q	$\rightarrow$
	q	$\triangleright$	s	$\rightarrow$

Example 2 (cont.)

How the machine on the previous slide works on input $\square\diamond\square$:

- First, it changes its state to q.
- Then the head goes to the right end of the input word. When it reaches the first blank, the head starts to move back and forth indefinitely, alternating between transitions ($\bullet$) and ($*$).
- Thus the machine **never terminates** on input $\square\diamond\square$.

$$(s, \triangleright\underline{\square}\diamond\square),\ (q, \triangleright\underline{\square}\diamond\square),\ (q, \triangleright\square\underline{\diamond}\square),\ (q, \triangleright\square\diamond\underline{\square}),\ (q, \triangleright\square\diamond\square\underline{\sqcup}),$$
$$(q, \triangleright\square\diamond\underline{\square}),\ (q, \triangleright\square\diamond\square\underline{\sqcup}),\ (q, \triangleright\square\diamond\underline{\square}),\ (q, \triangleright\square\diamond\square\underline{\sqcup}),\ \ldots$$

There can be Turing machine computations that never stop

Example 2 (cont.)

How the machine on the previous slide works on input ◇□□□:

It stops in one step

$$(s, \triangleright\underline{\diamond}\square\square\square),\ (h, \triangleright\underline{\square}\square\square\square)$$

It can happen that
a Turing machine terminates on some inputs,
while runs forever on other inputs

Computational tasks

Turing machines are not only simple data processing devices.
They can perform many important computational tasks such as

- compute functions on strings
- recognise languages
- compute arithmetic functions
- ...

A function $f : \Sigma^* \to \Sigma^*$, for an alphabet Σ, is called a **function on Σ-strings**
(in other words, f maps each string w over Σ to a string $f(w)$ over Σ)

(1) $\Sigma = \{a, b\}$; for each word w over Σ, let $\boxed{f(w) = wa}$

Then $f(aaabb) = aaabba$, $f(\varepsilon) = a$, $f(ba) = baa$, ...

(2) $\Sigma = \{\square, \diamond\}$; for each word w over Σ, let $\boxed{f(w) = \begin{cases} w\square w, & \text{if } |w| \text{ is even} \\ w, & \text{if } |w| \text{ is odd} \end{cases}}$

Then $f(\square\diamond) = \square\diamond\square\square\diamond$, $f(\varepsilon) = \square$, $f(\diamond) = \diamond$, ...

Computing functions on strings

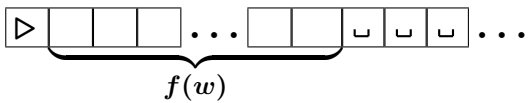
Let Σ be an alphabet and f a function on Σ-strings

(that is, f maps each string w over Σ to a string $f(w)$ over Σ).

Let M be a Turing machine with tape alphabet $\Sigma \cup \{\triangleright, \sqcup\}$.

Say that Turing machine M **computes the function** f if, for every string w over Σ

- M halts on input w → means that there is no infinite loop
- after halting the tape looks like



that is, $f(w)$ is the **output** of M on input w

A function f on strings is called **Turing computable** if there exists a Turing machine M that computes it

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

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

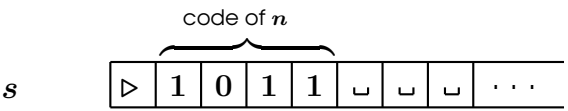
binary codes for natural numbers.

For example,

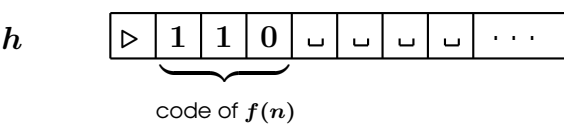
$$1101 \text{ codes } 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13.$$

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, $\mathbf{00}\textvisiblespace\mathbf{1}$) then we do not care what the machine does.

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.

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; }
}
```

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 **?**

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.