Turing machines

History:

- 1900 David Hilbert asked: Given an axiomatization of an area of mathematics (e.g. Hilbert's axiomatization of Geometry, or Frege's axiomatization of natural numbers), is there a *procedure* that will determine the truth of any conjecture?
- Hilbert did not mathematically define the concept of a procedure.
- 1930 Gödel showed that every "true" sentence in first-order predicate logic can be proved.
- It remained to develop a procedure of determining whether the a sentence in first order predicate logic is true or false.
- 1936 Alan Turing proposed the Turing machine to define the concept of a procedure.
- His conclusion: There is no procedure that can classify sentences of

A *Turing machine* represents an idealized human computer:

- Performs one kind of computation.
- Never makes a mistake.
- Never runs out of paper, pencils, erasers, or time.
- Finite brain power.

Syntax of a Turing Machine

A Turing Machine is a tuple $(S, Q, q_{\text{start}}, q_{\text{halt}}, T)$

- S is a finite alphabet set including the blank symbol $\not b$
- Q is a finite state set
- $q_{\text{start}} \in Q$ is the start state
- $q_{\text{halt}} \notin Q$ is the halt pseudostate.
- T is a set of transitions (see below)

Each transition is a tuple (q, a, r, b, d) where $q \in Q$, $a, b \in S$, $r \in Q \cup \{q_{\text{halt}}\}$,and $d \in \{L, R\}$.
We require that the Turing machine be deterministic in the sense that for each (q, a) pair, there is exactly one transition (q, a, b, r):

$$\forall q \in Q \cdot \forall a \in S \cdot 1 = |\{(r, b, d) \mid (q, a, r, b, d) \in T\}|$$

Operation of a Turing Machine:

The input and output of a Turing Machine is represented by a tape, which is a two way infinite sequence of symbols. Although the tape is considered to be infinite, we will restrict it to contain only a finite amount of information by insisting that all but a finite number of cells on the tape be blank.

We imagine there is a read/write head over the tape. In each step of execution, one symbol is read and that cell of the tape is then overwritten with a symbol. The the head moves either left or right.

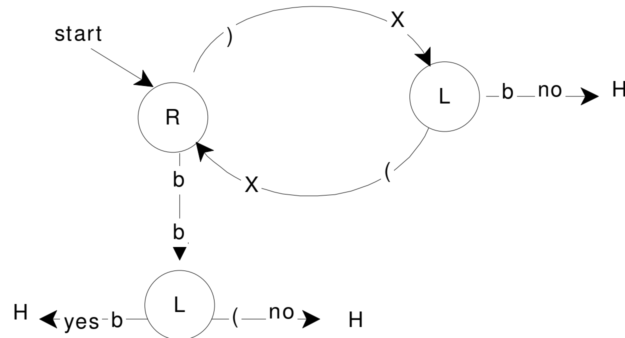
A configuration of a Turing machine consists of a state q, a value of the tape t, and a position of the head k.

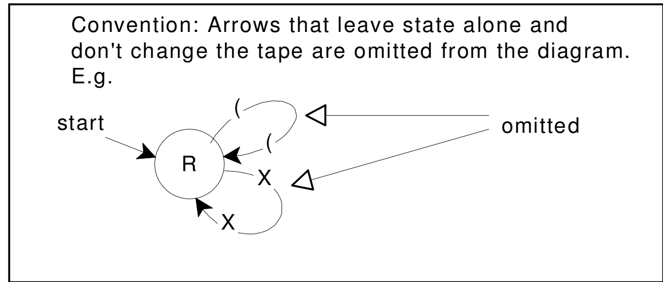
Let w be the input string.

var $t : \mathbb{Z} \xrightarrow{\text{tot}} S := [\ldots, \not{b}, \not{b}, \not{b}, ...]$
for $i \leftarrow \{0, .. \|w\|\}$ do $t(i) := w(i)$
var $k := 0$
var $q := q_{\text{start}}$
while true do (
    let $r, b, d \mid (q, t(k), r, b, d) \in T$
    $t(k) := b$;
    if $r = q_{\text{halt}}$ break
    if $d = L$ then $k := k - 1$ else $k := k + 1$;
    $q := r$)
output $t[k, ...]$

An example: Another parenthesis checker.

In the pictures states are marked with R or L.
All transitions into a state marked R have $d = R$.
All transitions into a state marked L state have $d = L$.
The halt pseudostate is H.

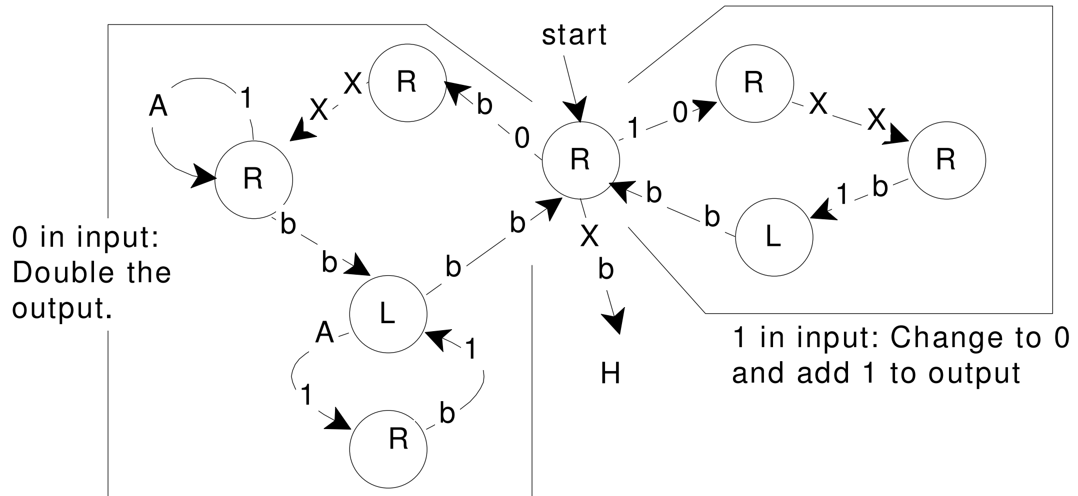




Notice that it overwrites its input with X's two matching parentheses at a time.

A Turing machine to convert binary to unary

Convert Binary to Unary.



Initial Config

start

...bbb110Xbbbbb...

Final config

H

...bbbbbbb111111bbb...

Halting

Note a Turing machine may not halt on some inputs. Thus a Turing machine does not define a function from strings to strings, but rather a *partial function* i.e. a function except that for some members of the domain there is no result.

Problem 0 (The halting problem): Given a Turing machine and an input tape, determine whether the TM will halt.

Problem 1: Given a Turing machine determine, if it will halt for all initial configurations.

Problem 2: (Busy Beaver): Given a number N, find the maximum number of 1's that can be output by a TM with N states, and an alphabet of $\{1, \not b\}$ when started on an all blank tape. (NB the TM must halt for this input tape.)

Equivalent formalisms.

- Idealized Programming languages
  - Suppose C's or Java's "new" command never failed to find new memory.
  - Suppose that in Haskell, LISP, or ML we can construct arbitrarily long lists.
- Idealized computers.
  - a computer in which a memory of finite words is indexed by the natural numbers (rather than a finite subset thereof)
  - a computer with a finite number of words (at least 2), but where every memory word contains an integer (rather than a 32-bit integer). (Register Machine)

  - ∗ Infinite number of integers (Random Access Machine)
  - ∗ a PC with an endless supply of rewritable disks and an operator to change them on request.

- Lambda Calculus (Church 1936). A simple Functional Programming Language.
- Partial recursive functions (Kleene 1936): Functions formed from addition, subtraction and a minimum operator:

$\min_x f(x) =$ the minimum natural x such that $f(x) = 0$.

- Grammars in which more than one symbol may appear on Left-hand-side

$$
\begin{aligned}
L &\rightarrow B \\
B &\rightarrow 0ABC2 \mid \epsilon \\
AC &\rightarrow 1 \\
A1 &\rightarrow 1A
\end{aligned}
$$

- 2PDAs — Like PDA's but with 2 stacks.
- The human computer.

Any limitation that applies to an idealized programming language or a idealized computer will also apply to real programming languages and real computers.

Effective computability

Defn: An "*effectively computable function*" is a function which can be evaluated by an algorithm.
N.B. This is not a mathematical definition as "algorithm" is not a well defined concept.

The Church-Turing Thesis: Any "effectively computable function" can be computed by some (always halting) Turing machine.
Note: This is not a mathematical theorem. It is a generally held belief.
Note: The converse is obviously true. Any function computed by a TM is effectively computable.
Corollary: Any effectively computable function can be computed by any of the above equivalent formalisms.

Why Turing Machines make good models

Why would we be interested in Turing Machines and similar models if all "real" devices have a finite amount of memory?

- We can not investigate time and space complexity unless we can consider inputs of unbounded size.
- Limitations on finite memory models can be overcome by enlarging the memory. Limitations on infinite memory models are fundamental.
- Running out of space in software is unpredictable. The model of memory used by (good) software developers is that of an unknown amount of memory. If an algorithm won't work on a machine with unlimited memory, it certainly won't work given an unknown amount of memory.

Limits on the power of Turing machines

Are there well-defined mathematical problems that no computer can solve?

We will show that the Halting Problem can not be solved by any algorithmic method

Claim: For any TM, there exists an equivalent TM with an alphabet of $\{0, 1, \not{b}\}$.

- We can encode the original alphabet with a binary code.

We can encode any such TM as a sequence of symbols from a small alphabet.

E.g. use binary encoding of the states.

Define

$$\textit{enc}(T) = \text{ the encoding of } T$$

Theorem *the halting problem can not be solved by any Turing Machine.*

Proof (By contradiction)

Assume to the contrary that we have a TM H that solves the halting problem. Specifically:

- If we start H in a configuration

  ... $\not{b}$ $\not{b}$ $\not{b}$ $\langle$encoding of a T$\rangle$ X $\langle$I$\rangle$ $\not{b}$ $\not{b}$ $\not{b}$...

  then

  * if T halts for input I, H halts in a configuration:

    ... $\not{b}$ $\not{b}$ $\not{b}$ $\langle$Yes$\rangle$ $\not{b}$ $\not{b}$ $\not{b}$...

    and

  * if T does not halt for input I, H halts in a configuration

    ... $\not{b}$ $\not{b}$ $\not{b}$ $\langle$No$\rangle$ $\not{b}$ $\not{b}$ $\not{b}$...

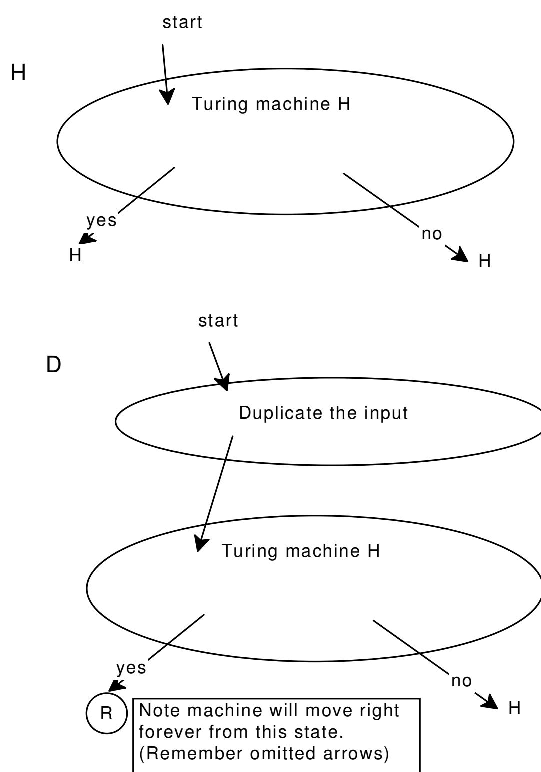
Thus H computes a function

$$H(enc(T), I) = \textbf{if } \text{T halts on I} \; \textbf{then} \text{ output yes and halt} \; \textbf{else} \text{ output no and halt}$$

We can build another TM D using H as a subroutine as follows

$$D(I) = \textbf{if } H(I, I) \textbf{ then } loop \textbf{ else } halt$$

I.e. D loops forever if H returns yes.



Consider running D with an input of $enc(D)$. This computation must halt or not. Furthermore H can tell us which.

- Suppose $D(enc(D))$ halts.

$$\begin{aligned} & D(enc(D)) \\ = \; & \textbf{if } H(enc(D), enc(D)) \textbf{ then } loop \textbf{ else } halt \\ = \; & \text{Supposition} \\ & \textbf{if } yes \textbf{ then } loop \textbf{ else } halt \\ = \; & loop \end{aligned}$$

  Thus $D(enc(D))$ does not halt. Contradiction.

- Suppose now that $D(enc(D))$ does not halt

$$\begin{aligned} & D(enc\, D) \\ = \; & \textbf{if } H(enc(D), enc(D)) \textbf{ then } loop \textbf{ else } halt \\ = \; & \text{Supposition} \\ & \textbf{if } no \textbf{ then } loop \textbf{ else } halt \\ = \; & halt \end{aligned}$$

  Thus $D(enc(D))$ does halt. Contradiction

Either way there is a contradiction. The culprit must be out assumption that H exists.

QED

Note: For many TMs and specific inputs, we *can* determine if they halt.

Corollary: Assuming the C-T thesis, there are problems no computer (and no human) can solve.

Other unsolvable problems.

- Given an arbitrary mathematical conjecture, does it have

a proof?

- Given an arbitrary polynomial such as

$$a^3 b^2 c^4 + a^2 b^3 = 0$$

Do there exists integer solutions?

That is unsolvable was proved in 1970: 70 years after Hilbert asked for a solution.

(Proof by showing that such equations can encode Turing Machines.)

The first says there is no universal theorem proving method. The second shows there are implications for everyday mathematics.

A universal TM (Optional)

Claim: There exists a TM (called U) with the following properties:

- Suppose we have a TM T (with alphabet $\{0, 1, \not b\}$) that only uses the left half of the tape.
- Suppose we have an input sequence I for T
- We will start it in a configuration
  ... $\not b$ $\not b$ $\not b$ $\langle$encoding of a T$\rangle$ X $\langle$input I$\rangle$ $\not b$ $\not b$ $\not b$...
- If T halts on input I with output O then U halts in a configuration
  ... $\not b$ $\not b$ $\not b$ $\langle$encoding of a T$\rangle$ X $\langle$output O$\rangle$ $\not b$ $\not b$ $\not b$...
- If T does not halt on input I then U does not halt.

Minsky (1967) describes a U with 23 states.

U is a stored-program computer: It is capable of executing any algorithm. T is the software representing the algorithm.
We only need one TM: U
We only need one kind of computing machine. The rest we can do with software.

The Chomsky Hierarchy (+ Others)

Type 3: (Finite space) Regular languages: Recognized by

- Deterministic Finite Recognizers (DFRs)
- Nondeterministic Finite Recognizers (NDFRs)
- Regular expressions & REFRs

Type ?: (Stack memory) LR(k)

- Deterministic Push-Down Automata

Type 2: (Stack memory) Context Free Languages

- Nondeteriministic Push-Down Automata
- Context-Free Grammars

Type 1 (Linear space) Context Sensitive Languages

- Turing Machines with memory $O(N)$ of input size N.

Type ? (Infinite space) Recursive Languages

- Turing Machines (etc) that halt

Type 0 (Infinite space.) Recursively Enumerable Languages

- Turing Machines, Unrestricted grammars, Idealized Computers and Languages, Lambda Calculus, General Recursive Functions