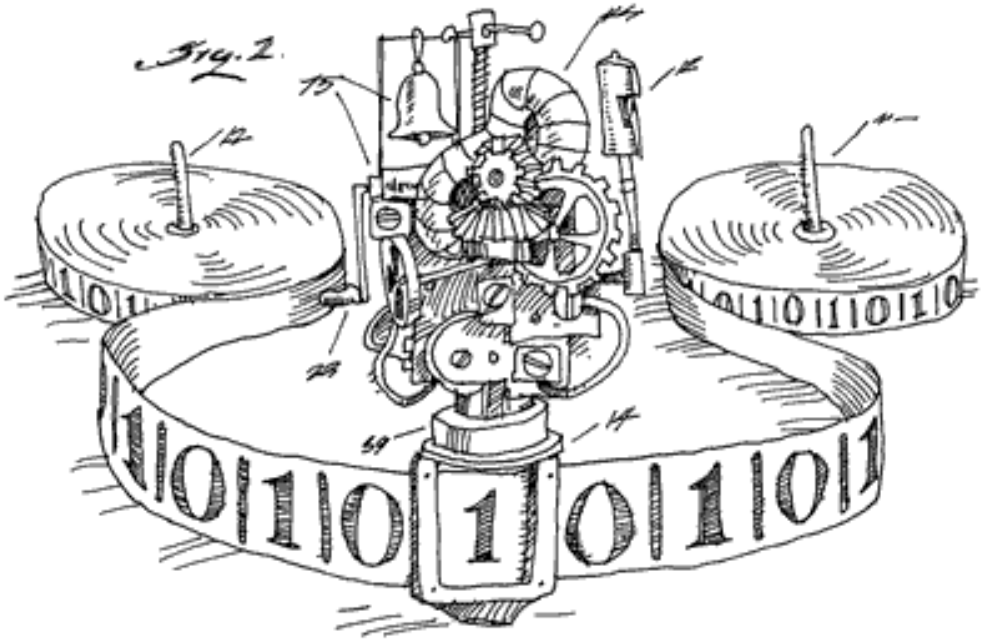
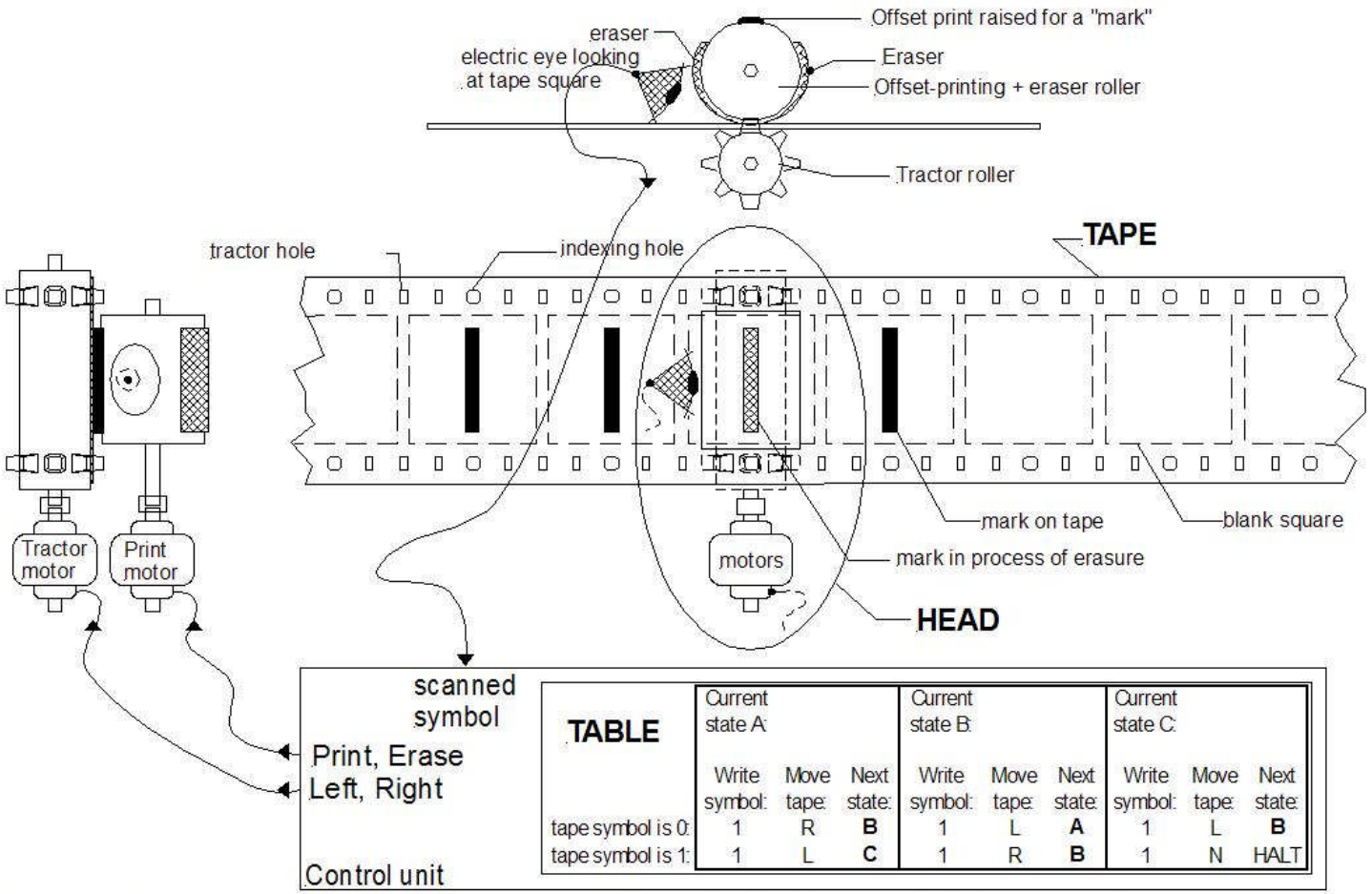
LA MACHINE DE TURING



COMPILATION

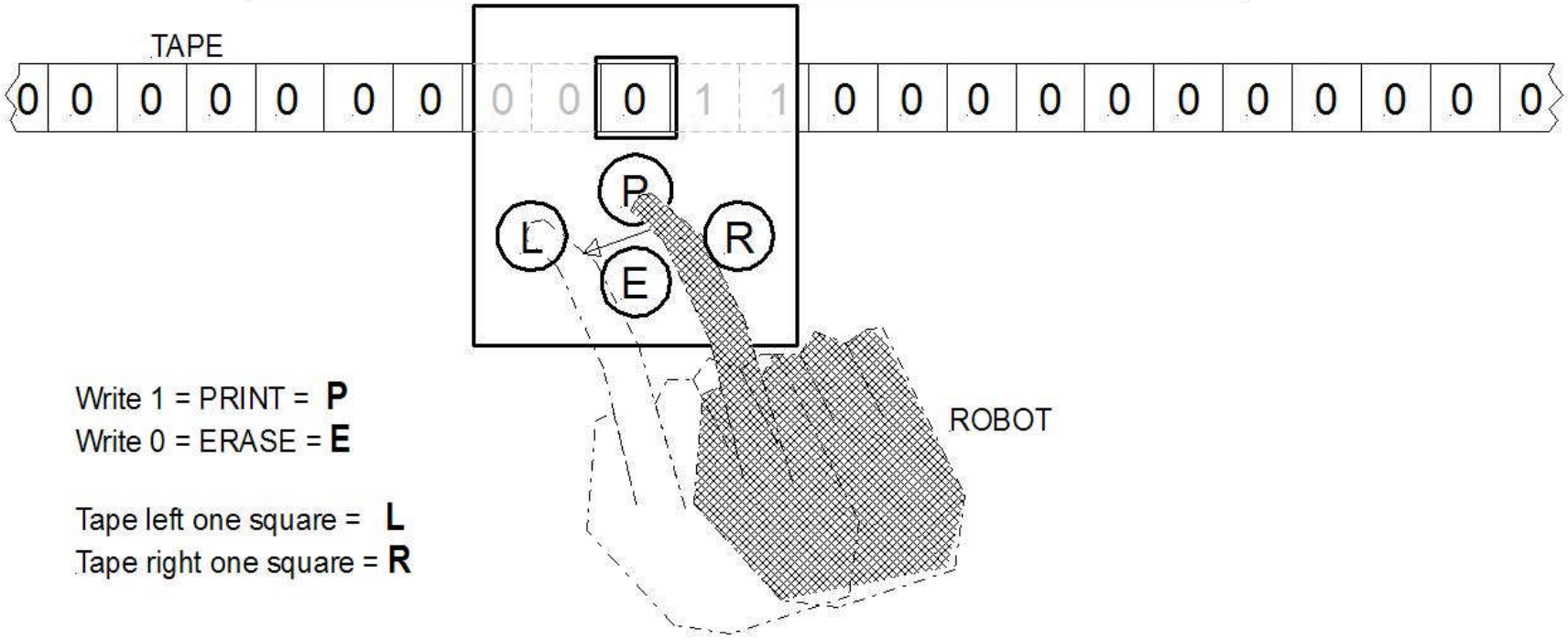


TABLE

	Current state A:			Current state B:			Current state C:		
	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:
tape symbol is 0:	1	R	**B**	1	L	**A**	1	L	**B**
tape symbol is 1:	1	L	**C**	1	R	**B**	1	N	HALT

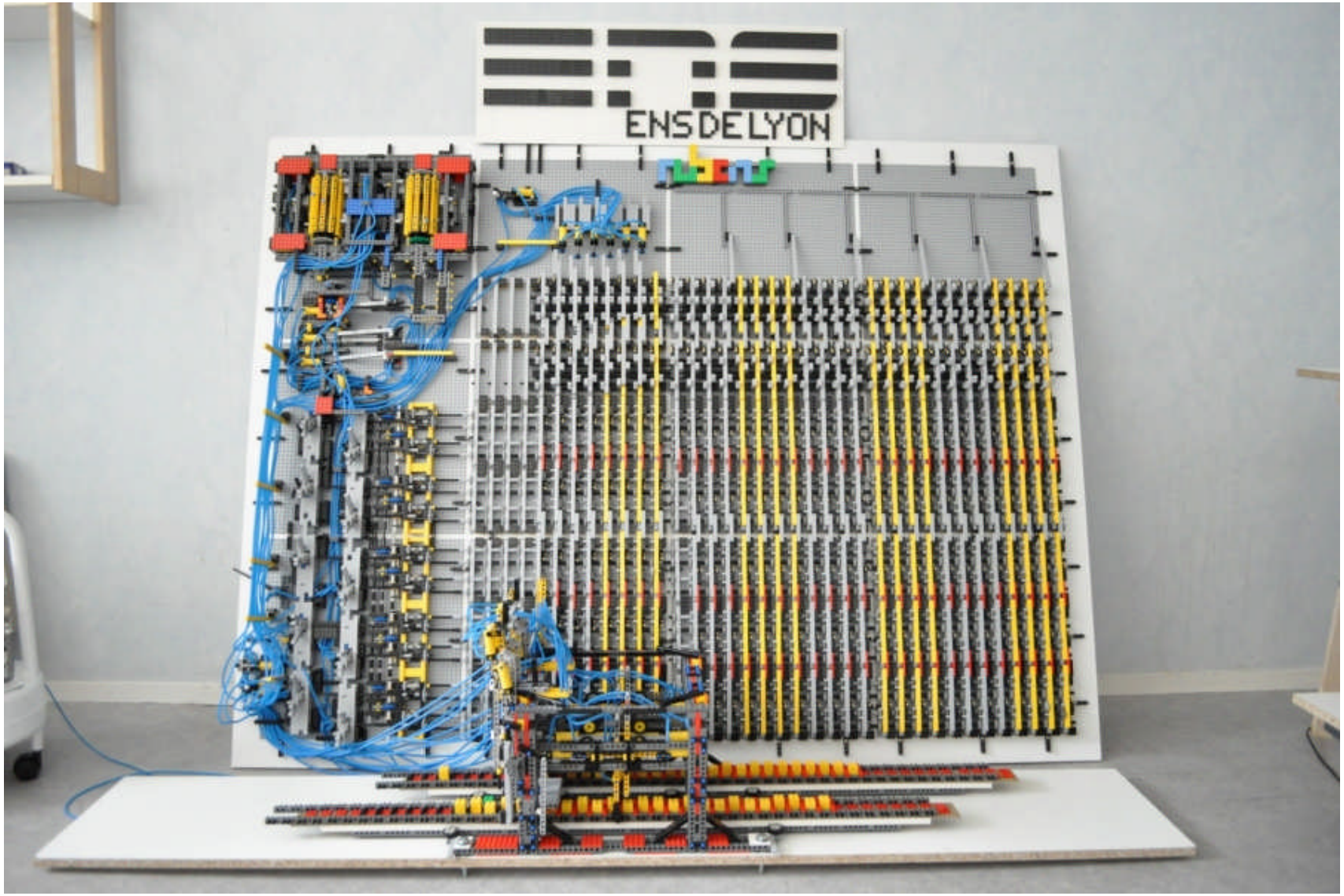
A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

TABLE	Current state A:			Current state B:			Current state C:		
	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:
tape symbol is 0:	1	R	**B**	1	L	**A**	1	L	**B**
tape symbol is 1:	1	L	**C**	1	R	**B**	1	N	HALT



Write 1 = PRINT = **P**
Write 0 = ERASE = **E**

Tape left one square = **L**
Tape right one square = **R**



REALISATION A L’AIDE DE LEGOS

Turing machine

This article is about the symbol manipulator. For the deciphering machine, see Bombe. For the test of artificial intelligence, see Turing test. For the instrumental rock band, see Turing Machine (band).
Classes of automata

A **Turing machine** is an abstract machine[1] that manipulates symbols on a strip of tape according to a table of rules; to be more exact, it is a mathematical model of computation that defines such a device.[2] Despite the model's simplicity, given any computer algorithm, a Turing machine can be constructed that is capable of simulating that algorithm's logic.[3]

The machine operates on an infinite[4] memory tape divided into *cells*.[5] The machine positions its *head* over a cell and "reads" (scans[6]) the symbol there. Then as per the symbol and its present place in a *finite table*[7] of user-specified instructions the machine (i) writes a symbol (e.g. a digit or a letter from a finite alphabet) in the cell (some models allowing symbol erasure[8] and/or no writing), then (ii) either moves the tape one cell left or right (some models allow no motion, some models move the head),[9] then (iii) (as determined by the observed symbol and the machine's place in the table) either proceeds to a subsequent instruction or halts[10] the computation.

The Turing machine was invented in 1936 by Alan Turing,[11][12] who called it an *a-machine* (automatic machine).[13] With this model Turing was able to answer two questions in the negative: (1) Does a machine exist that can determine whether any arbitrary machine on its tape is "circular" (e.g. freezes, or fails to continue its computational task); similarly, (2) does a machine exist that can determine whether any arbitrary machine on its tape ever prints a given symbol.[14] Thus by providing a mathematical description of a very simple device capable of arbitrary computations, he was able to prove properties of computation in general - and in particular, the uncomputability of the Hilbert Entscheidungsproblem ("decision problem").[15]

Thus, Turing machines prove fundamental limitations on the power of mechanical computation.[16] While they can express arbitrary computations, their minimalistic design makes them unsuitable for computation in practice: real-world computers are based on different designs that, unlike Turing machines, use random-access memory.

Turing completeness is the ability for a system of instructions to simulate a Turing machine. A programming language that is Turing complete is theoretically capable of expressing all tasks accomplishable by computers; nearly all programming languages are Turing complete if the limitations of finite memory are ignored.

1 Overview

A Turing machine is a general example of a CPU that controls all data manipulation done by a computer, with the canonical machine using sequential memory to store data. More specifically, it is a machine (automaton) capable of enumerating some arbitrary subset of valid strings of an alphabet; these strings are part of a recursively enumerable set.

Assuming a black box, the Turing machine cannot know whether it will eventually enumerate any one specific string of the subset with a given program. This is due to the fact that the halting problem is unsolvable, which has major implications for the theoretical limits of computing.

The Turing machine is capable of processing an unrestricted grammar, which further implies that it is capable of robustly evaluating first-order logic in an infinite number of ways. This is famously demonstrated through lambda calculus.

A Turing machine that is able to simulate any other Turing machine is called a universal Turing machine (**UTM**, or simply a **universal machine**). A more mathematically oriented definition with a similar "universal" nature was introduced by Alonzo Church, whose work on lambda calculus intertwined with Turing's in a formal theory of computation known as the Church–Turing thesis. The thesis states that Turing machines indeed capture the informal notion of effective methods in logic and mathematics, and provide a precise definition of an algorithm or "mechanical procedure". Studying their abstract properties yields many insights into computer science and complexity theory.

1.1 Physical description

In his 1948 essay, "Intelligent Machinery", Turing wrote that his machine consisted of:

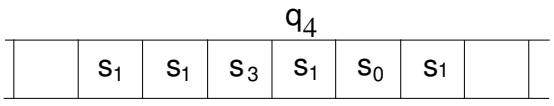
> ...an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol

in the machine; it is called the scanned symbol. The machine can alter the scanned symbol, and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.[17] (Turing 1948, p. 3[18])

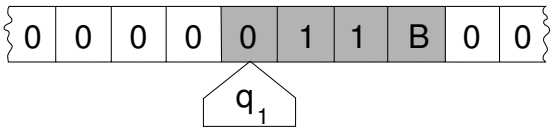
2 Informal description

For visualizations of Turing machines, see Turing machine gallery.

The Turing machine mathematically models a machine that mechanically operates on a tape. On this tape are symbols, which the machine can read and write, one at a time, using a tape head. Operation is fully determined by a finite set of elementary instructions such as "in state 42, if the symbol seen is 0, write a 1; if the symbol seen is 1, change into state 17; in state 17, if the symbol seen is 0, write a 1 and change to state 6;" etc. In the original article ("On computable numbers, with an application to the Entscheidungsproblem", see also references below), Turing imagines not a mechanism, but a person whom he calls the "computer", who executes these deterministic mechanical rules slavishly (or as Turing puts it, "in a desultory manner").



The head is always over a particular square of the tape; only a finite stretch of squares is shown. The instruction to be performed (q_4) is shown over the scanned square. (Drawing after Kleene (1952) p.375.)



Here, the internal state (q_1) is shown inside the head, and the illustration describes the tape as being infinite and pre-filled with "0", the symbol serving as blank. The system's full state (its complete configuration*) consists of the internal state, any non-blank symbols on the tape (in this illustration "11B"), and the position of the head relative to those symbols including blanks, i.e. "011B". (Drawing after Minsky (1967) p. 121).*

More precisely, a Turing machine consists of:

1. A **tape** divided into cells, one next to the other. Each cell contains a symbol from some finite alphabet. The alphabet contains a special *blank* symbol (here written as '0') and one or more other symbols. The tape is assumed to be arbitrarily extendable to the left and to the right, i.e., the Turing machine is always supplied with as much tape as it needs for its computation. Cells that have not been written before are assumed to be filled with the blank symbol. In some models the tape has a left end marked with a special symbol; the tape extends or is indefinitely extensible to the right.

2. A **head** that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time. In some models the head moves and the tape is stationary.

3. A **state register** that stores the state of the Turing machine, one of finitely many. Among these is the special *start state* with which the state register is initialized. These states, writes Turing, replace the "state of mind" a person performing computations would ordinarily be in.

4. A finite **table**[19] of instructions[20] that, given the *state*(q_i) the machine is currently in *and* the *symbol*(a_j) it is reading on the tape (symbol currently under the head), tells the machine to do the following in sequence (for the 5-tuple models):

   - Either erase or write a symbol (replacing a_j with a_{j1}), *and then*
   - Move the head (which is described by d_k and can have values: 'L' for one step left *or* 'R' for one step right *or* 'N' for staying in the same place), *and then*
   - Assume the same or a *new state* as prescribed (go to state q_{i1}).

   In the 4-tuple models, erasing or writing a symbol (a_{j1}) and moving the head left or right (d_k) are specified as separate instructions. Specifically, the table tells the machine to (ia) erase or write a symbol *or* (ib) move the head left or right, *and then* (ii) assume the same or a new state as prescribed, but not both actions (ia) and (ib) in the same instruction. In some models, if there is no entry in the table for the current combination of symbol and state then the machine will halt; other models require all entries to be filled.

Note that every part of the machine (i.e. its state, symbol-collections, and used tape at any given time) and its actions (such as printing, erasing and tape motion) is *finite*, *discrete* and *distinguishable*; it is the unlimited amount of tape and runtime that gives it an unbounded amount of storage space.

3 Formal definition

Following Hopcroft and Ullman (1979, p. 148), a (one-tape) Turing machine can be formally defined as a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$ where

- Q is a finite, non-empty set of *states*
- Γ is a finite, non-empty set of *tape alphabet symbols*
- $b \in \Gamma$ is the *blank symbol* (the only symbol allowed to occur on the tape infinitely often at any step during the computation)
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of *input symbols*
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a partial function called the *transition function*, where L is left shift, R is right shift. (A relatively uncommon variant allows "no shift", say N, as a third element of the latter set.) If δ is not defined on the current state and the current tape symbol, then the machine halts.[21]
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *final* or *accepting states*. The initial tape contents is said to be *accepted* by M if it eventually halts in a state from F .

Anything that operates according to these specifications is a Turing machine.

The 7-tuple for the 3-state busy beaver looks like this (see more about this busy beaver at Turing machine examples):

- $Q = \{\text{A}, \text{B}, \text{C}, \text{HALT}\}$
- $\Gamma = \{0, 1\}$
- $b = 0$ ("blank")
- $\Sigma = \{1\}$
- $q_0 =$ A (the initial state)
- $F = \{\text{HALT}\}$
- $\delta =$ see state-table below

Initially all tape cells are marked with 0 .

4 Additional details required to visualize or implement Turing machines

In the words of van Emde Boas (1990), p. 6: "The set-theoretical object [his formal seven-tuple description similar to the above] provides only partial information on how the machine will behave and what its computations will look like."

For instance,

- There will need to be many decisions on what the symbols actually look like, and a failproof way of reading and writing symbols indefinitely.
- The shift left and shift right operations may shift the tape head across the tape, but when actually building a Turing machine it is more practical to make the tape slide back and forth under the head instead.
- The tape can be finite, and automatically extended with blanks as needed (which is closest to the mathematical definition), but it is more common to think of it as stretching infinitely at both ends and being pre-filled with blanks except on the explicitly given finite fragment the tape head is on. (This is, of course, not implementable in practice.) The tape *cannot* be fixed in length, since that would not correspond to the given definition and would seriously limit the range of computations the machine can perform to those of a linear bounded automaton.

4.1 Alternative definitions

Definitions in literature sometimes differ slightly, to make arguments or proofs easier or clearer, but this is always done in such a way that the resulting machine has the same computational power. For example, changing the set $\{L, R\}$ to $\{L, R, N\}$, where N ("None" or "No-operation") would allow the machine to stay on the same tape cell instead of moving left or right, does not increase the machine's computational power.

The most common convention represents each "Turing instruction" in a "Turing table" by one of nine 5-tuples, per the convention of Turing/Davis (Turing (1936) in *Undecidable*, p. 126-127 and Davis (2000) p. 152):

(definition 1): **(q_i, S_j, S_k/E/N, L/R/N, q_m)**

> **(** current state **q_i ,** symbol scanned **S_j ,** print symbol **S_k**/erase **E**/none **N ,** move_tape_one_square left **L**/right **R**/none **N ,** new state **q_m)**

Other authors (Minsky (1967) p. 119, Hopcroft and Ullman (1979) p. 158, Stone (1972) p. 9) adopt a different convention, with new state **q_m** listed immediately after the scanned symbol S_j:

(definition 2): **(q_i, S_j, q_m, S_k/E/N, L/R/N)**

> **(** current state **q_i ,** symbol scanned **S_j ,** new state **q_m ,** print symbol **S_k**/erase **E**/none **N ,** move_tape_one_square left **L**/right **R**/none **N)**

For the remainder of this article “definition 1” (the Turing/Davis convention) will be used.

In the following table, Turing’s original model allowed only the first three lines that he called N1, N2, N3 (cf Turing in *Undecidable*, p. 126). He allowed for erasure of the “scanned square” by naming a 0th symbol S_0 = “erase” or “blank”, etc. However, he did not allow for non-printing, so every instruction-line includes “print symbol S_k" or “erase” (cf footnote 12 in Post (1947), *Undecidable* p. 300). The abbreviations are Turing’s (*Undecidable* p. 119). Subsequent to Turing’s original paper in 1936–1937, machine-models have allowed all nine possible types of five-tuples:

Any Turing table (list of instructions) can be constructed from the above nine 5-tuples. For technical reasons, the three non-printing or “N” instructions (4, 5, 6) can usually be dispensed with. For examples see Turing machine examples.

Less frequently the use of 4-tuples are encountered: these represent a further atomization of the Turing instructions (cf Post (1947), Boolos & Jeffrey (1974, 1999), Davis-Sigal-Weyuker (1994)); also see more at Post–Turing machine.

4.2 The “state”

The word “state” used in context of Turing machines can be a source of confusion, as it can mean two things. Most commentators after Turing have used “state” to mean the name/designator of the current instruction to be performed—i.e. the contents of the state register. But Turing (1936) made a strong distinction between a record of what he called the machine’s “m-configuration”, (its internal state) and the machine’s (or person’s) “state of progress” through the computation - the current state of the total system. What Turing called “the state formula” includes both the current instruction and *all* the symbols on the tape:

> Thus the state of progress of the computation at any stage is completely determined by the note of instructions and the symbols on the tape. That is, the **state of the system** may be described by a single expression (sequence of symbols) consisting of the symbols on the tape followed by Δ (which we suppose not to appear elsewhere) and then by the note of instructions. This expression is called the 'state formula'.
> — *Undecidable*, p.139–140, emphasis added

Earlier in his paper Turing carried this even further: he gives an example where he placed a symbol of the current “m-configuration”—the instruction’s label—beneath the scanned square, together with all the symbols on the tape (*Undecidable*, p. 121); this he calls “the *complete configuration*" (*Undecidable*, p. 118). To print the “complete configuration” on one line, he places the state-label/m-configuration to the *left* of the scanned symbol.

A variant of this is seen in Kleene (1952) where Kleene shows how to write the Gödel number of a machine’s “situation": he places the “m-configuration” symbol q_4 over the scanned square in roughly the center of the 6 non-blank squares on the tape (see the Turing-tape figure in this article) and puts it to the *right* of the scanned square. But Kleene refers to “q_4" itself as “the machine state” (Kleene, p. 374-375). Hopcroft and Ullman call this composite the “instantaneous description” and follow the Turing convention of putting the “current state” (instruction-label, m-configuration) to the *left* of the scanned symbol (p. 149).

Example: total state of 3-state 2-symbol busy beaver after 3 “moves” (taken from example “run” in the figure below):

1**A**1

This means: after three moves the tape has ... 000110000 ... on it, the head is scanning the right-most 1, and the state is **A**. Blanks (in this case represented by “0"s) can be part of the total state as shown here: **B**01; the tape has a single 1 on it, but the head is scanning the 0 (“blank”) to its left and the state is **B**.

“State” in the context of Turing machines should be clarified as to which is being described: (*i*) the current instruction, or (*ii*) the list of symbols on the tape together with the current instruction, or (*iii*) the list of symbols on the tape together with the current instruction placed to the left of the scanned symbol or to the right of the scanned symbol.

Turing’s biographer Andrew Hodges (1983: 107) has noted and discussed this confusion.

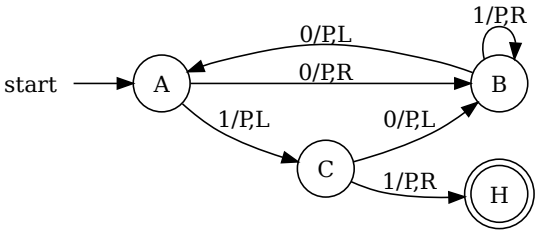
4.3 Turing machine “state” diagrams

To the right: the above TABLE as expressed as a “state transition” diagram.

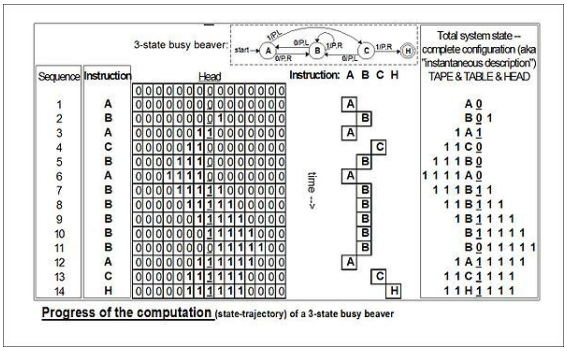
Usually large TABLES are better left as tables (Booth, p. 74). They are more readily simulated by computer in tabular form (Booth, p. 74). However, certain concepts—e.g. machines with “reset” states and machines with repeating patterns (cf Hill and Peterson p. 244ff)—can be more readily seen when viewed as a drawing.

Whether a drawing represents an improvement on its TABLE must be decided by the reader for the particular context. See Finite state machine for more.

The reader should again be cautioned that such diagrams represent a snapshot of their TABLE frozen in time, *not* the course (“trajectory”) of a computation *through* time



The "3-state busy beaver" Turing machine in a finite state representation. Each circle represents a "state" of the TABLE—an "m-configuration" or "instruction". "Direction" of a state transition *is shown by an arrow. The label (e.g.. **0/P,R**) near the outgoing state (at the "tail" of the arrow) specifies the scanned symbol that causes a particular transition (e.g. **0**) followed by a slash **/**, followed by the subsequent "behaviors" of the machine, e.g. "**P** Print" then move tape "**R** Right". No general accepted format exists. The convention shown is after McClusky (1965), Booth (1967), Hill, and Peterson (1974).*



The evolution of the busy-beaver's computation starts at the top and proceeds to the bottom.

and/or space. While every time the busy beaver machine "runs" it will always follow the same state-trajectory, this is not true for the "copy" machine that can be provided with variable input "parameters".

The diagram "Progress of the computation" shows the 3-state busy beaver's "state" (instruction) progress through its computation from start to finish. On the far right is the Turing "complete configuration" (Kleene "situation", Hopcroft–Ullman "instantaneous description") at each step. If the machine were to be stopped and cleared to blank both the "state register" and entire tape, these "configurations" could be used to rekindle a computation anywhere in its progress (cf Turing (1936) *Undecidable* pp. 139–140).

5 Models equivalent to the Turing machine model

See also: Turing machine equivalents, Register machine, and Post–Turing machine

Many machines that might be thought to have more computational capability than a simple universal Turing machine can be shown to have no more power (Hopcroft and Ullman p. 159, cf Minsky (1967)). They might compute faster, perhaps, or use less memory, or their instruction set might be smaller, but they cannot compute more powerfully (i.e. more mathematical functions). (Recall that the Church–Turing thesis *hypothesizes* this to be true for any kind of machine: that anything that can be "computed" can be computed by some Turing machine.)

A Turing machine is equivalent to a pushdown automaton that has been made more flexible and concise by relaxing the last-in-first-out requirement of its stack.

At the other extreme, some very simple models turn out to be Turing-equivalent, i.e. to have the same computational power as the Turing machine model.

Common equivalent models are the multi-tape Turing machine, multi-track Turing machine, machines with input and output, and the *non-deterministic* Turing machine (NDTM) as opposed to the *deterministic* Turing machine (DTM) for which the action table has at most one entry for each combination of symbol and state.

Read-only, right-moving Turing machines are equivalent to NDFAs (as well as DFAs by conversion using the NDFA to DFA conversion algorithm).

For practical and didactical intentions the equivalent register machine can be used as a usual assembly programming language.

6 Choice c-machines, Oracle o-machines

Early in his paper (1936) Turing makes a distinction between an "automatic machine"—its "motion ... completely determined by the configuration" and a "choice machine":

> ...whose motion is only partially determined by the configuration ... When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems.
> — *Undecidable*, p. 118

Turing (1936) does not elaborate further except in a footnote in which he describes how to use an a-machine to "find all the provable formulae of the [Hilbert] calculus" rather than use a choice machine. He "suppose[s] that the choices are always between two possibilities 0 and 1. Each proof will then be determined by a sequence of choices i_1, i_2, ..., i_n (i_1 = 0 or 1, i_2 = 0 or 1, ..., i_n

= 0 or 1), and hence the number $2^n + i_1 2^{n-1} + i_2 2^{n-2} + \ldots + i_n$ completely determines the proof. The automatic machine carries out successively proof 1, proof 2, proof 3, ..." (Footnote ‡, *Undecidable*, p. 138)

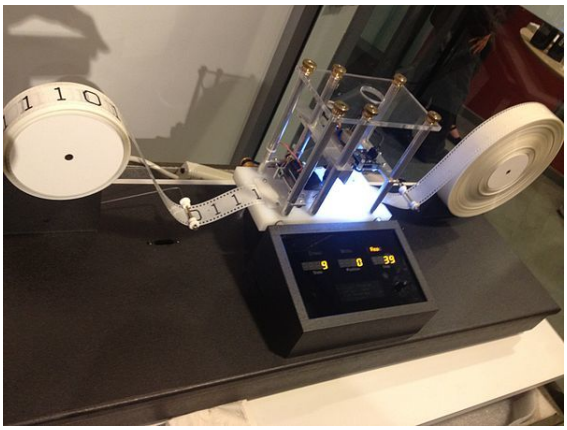
This is indeed the technique by which a deterministic (i.e. a-) Turing machine can be used to mimic the action of a nondeterministic Turing machine; Turing solved the matter in a footnote and appears to dismiss it from further consideration.

An oracle machine or o-machine is a Turing a-machine that pauses its computation at state "**o**" while, to complete its calculation, it "awaits the decision" of "the oracle"—an unspecified entity "apart from saying that it cannot be a machine" (Turing (1939), Undecidable p. 166–168). The concept is now actively used by mathematicians.

7 Universal Turing machines

Main article: Universal Turing machine

As Turing wrote in *Undecidable*, p. 128 (italics added):



An implementation of a Turing machine

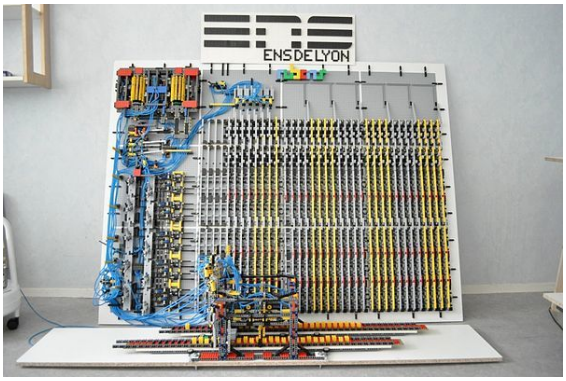
> It is possible to invent a *single machine* which can be used to compute *any* computable sequence. If this machine **U** is supplied with the tape on the beginning of which is written the string of quintuples separated by semicolons of some computing machine **M**, then **U** will compute the same sequence as **M**.

This finding is now taken for granted, but at the time (1936) it was considered astonishing. The model of computation that Turing called his "universal machine"—"**U**" for short—is considered by some (cf Davis (2000)) to have been the fundamental theoretical breakthrough that led to the notion of the stored-program computer.

> Turing's paper ... contains, in essence, the invention of the modern computer and some of the programming techniques that accompanied it.
> — Minsky (1967), p. 104

In terms of computational complexity, a multi-tape universal Turing machine need only be slower by logarithmic factor compared to the machines it simulates. This result was obtained in 1966 by F. C. Hennie and R. E. Stearns. (Arora and Barak, 2009, theorem 1.9)

8 Comparison with real machines



A Turing machine realisation in Lego

It is often said that Turing machines, unlike simpler automata, are as powerful as real machines, and are able to execute any operation that a real program can. What is neglected in this statement is that, because a real machine can only have a finite number of *configurations*, this "real machine" is really nothing but a linear bounded automaton. On the other hand, Turing machines are equivalent to machines that have an unlimited amount of storage space for their computations. However, Turing machines are not intended to model computers, but rather they are intended to model computation itself. Historically, computers, which compute only on their (fixed) internal storage, were developed only later.

There are a number of ways to explain why Turing machines are useful models of real computers:

1. Anything a real computer can compute, a Turing machine can also compute. For example: "A Turing machine can simulate any type of subroutine found in programming languages, including recursive procedures and any of the known parameter-passing mechanisms" (Hopcroft and Ullman p. 157). A large enough FSA can also model any real computer, disregarding IO. Thus, a statement about the limitations of Turing machines will also apply to real computers.
2. The difference lies only with the ability of a Turing machine to manipulate an unbounded amount

of data. However, given a finite amount of time, a Turing machine (like a real machine) can only manipulate a finite amount of data.

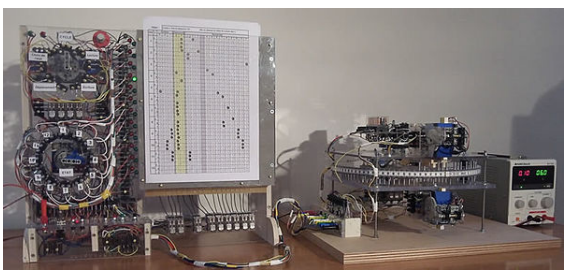
3. Like a Turing machine, a real machine can have its storage space enlarged as needed, by acquiring more disks or other storage media. If the supply of these runs short, the Turing machine may become less useful as a model. But the fact is that neither Turing machines nor real machines need astronomical amounts of storage space in order to perform useful computation. The processing time required is usually much more of a problem.

4. Descriptions of real machine programs using simpler abstract models are often much more complex than descriptions using Turing machines. For example, a Turing machine describing an algorithm may have a few hundred states, while the equivalent deterministic finite automaton (DFA) on a given real machine has quadrillions. This makes the DFA representation infeasible to analyze.

5. Turing machines describe algorithms independent of how much memory they use. There is a limit to the memory possessed by any current machine, but this limit can rise arbitrarily in time. Turing machines allow us to make statements about algorithms which will (theoretically) hold forever, regardless of advances in *conventional* computing machine architecture.

6. Turing machines simplify the statement of algorithms. Algorithms running on Turing-equivalent abstract machines are usually more general than their counterparts running on real machines, because they have arbitrary-precision data types available and never have to deal with unexpected conditions (including, but not limited to, running out of memory).

One way in which Turing machines are a poor model for programs is that many real programs, such as operating systems and word processors, are written to receive unbounded input over time, and therefore do not halt. Turing machines do not model such ongoing computation well (but can still model portions of it, such as individual procedures).



An experimental prototype to achieve Turing machine

8.1 Limitations of Turing machines

8.1.1 Computational complexity theory

Further information: Computational complexity theory

A limitation of Turing machines is that they do not model the strengths of a particular arrangement well. For instance, modern stored-program computers are actually instances of a more specific form of abstract machine known as the random access stored program machine or RASP machine model. Like the Universal Turing machine the RASP stores its "program" in "memory" external to its finite-state machine's "instructions". Unlike the universal Turing machine, the RASP has an infinite number of distinguishable, numbered but unbounded "registers"—memory "cells" that can contain any integer (cf. Elgot and Robinson (1964), Hartmanis (1971), and in particular Cook-Rechow (1973); references at random access machine). The RASP's finite-state machine is equipped with the capability for indirect addressing (e.g. the contents of one register can be used as an address to specify another register); thus the RASP's "program" can address any register in the register-sequence. The upshot of this distinction is that there are computational optimizations that can be performed based on the memory indices, which are not possible in a general Turing machine; thus when Turing machines are used as the basis for bounding running times, a 'false lower bound' can be proven on certain algorithms' running times (due to the false simplifying assumption of a Turing machine). An example of this is binary search, an algorithm that can be shown to perform more quickly when using the RASP model of computation rather than the Turing machine model.

8.1.2 Concurrency

Another limitation of Turing machines is that they do not model concurrency well. For example, there is a bound on the size of integer that can be computed by an always-halting nondeterministic Turing machine starting on a blank tape. (See article on unbounded nondeterminism.) By contrast, there are always-halting concurrent systems with no inputs that can compute an integer of unbounded size. (A process can be created with local storage that is initialized with a count of 0 that concurrently sends itself both a stop and a go message. When it receives a go message, it increments its count by 1 and sends itself a go message. When it receives a stop message, it stops with an unbounded number in its local storage.)

9 History

See also: Algorithm and Church–Turing thesis

They were described in 1936 by Alan Turing.

9.1 Historical background: computational machinery

Robin Gandy (1919–1995)—a student of Alan Turing (1912–1954) and his lifelong friend—traces the lineage of the notion of "calculating machine" back to Babbage (circa 1834) and actually proposes "Babbage's Thesis":

> *That the whole of development and operations of analysis are now capable of being executed by machinery.*
> — (italics in Babbage as cited by Gandy, p. 54)

Gandy's analysis of Babbage's Analytical Engine describes the following five operations (cf p. 52–53):

1. The arithmetic functions +, −, × where − indicates "proper" subtraction x − y = 0 if y ≥ x
2. Any sequence of operations is an operation
3. Iteration of an operation (repeating n times an operation P)
4. Conditional iteration (repeating n times an operation P conditional on the "success" of test T)
5. Conditional transfer (i.e. conditional "goto").

Gandy states that "the functions which can be calculated by (1), (2), and (4) are precisely those which are Turing computable." (p. 53). He cites other proposals for "universal calculating machines" including those of Percy Ludgate (1909), Leonardo Torres y Quevedo (1914), Maurice d'Ocagne (1922), Louis Couffignal (1933), Vannevar Bush (1936), Howard Aiken (1937). However:

> ... the emphasis is on programming a fixed iterable sequence of arithmetical operations. The fundamental importance of conditional iteration and conditional transfer for a general theory of calculating machines is not recognized ...
> — Gandy p. 55

9.2 The Entscheidungsproblem (the "decision problem"): Hilbert's tenth question of 1900

With regard to Hilbert's problems posed by the famous mathematician David Hilbert in 1900, an aspect of problem #10 had been floating about for almost 30 years before it was framed precisely. Hilbert's original expression for #10 is as follows:

> **10. Determination of the solvability of a Diophantine equation**. Given a Diophantine equation with any number of unknown quantities and with rational integral coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.
>
> The Entscheidungsproblem [decision problem for first-order logic] is solved when we know a procedure that allows for any given logical expression to decide by finitely many operations its validity or satisfiability ... The Entscheidungsproblem must be considered the main problem of mathematical logic.
>
> — quoted, with this translation and the original German, in Dershowitz and Gurevich, 2008

By 1922, this notion of "Entscheidungsproblem" had developed a bit, and H. Behmann stated that

> ... most general form of the Entscheidungsproblem [is] as follows:
>
> > A quite definite generally applicable prescription is required which will allow one to decide in a finite number of steps the truth or falsity of a given purely logical assertion ...
> > — Gandy p. 57, quoting Behmann

> Behmann remarks that ... the general problem is equivalent to the problem of deciding which mathematical propositions are true.
> — *ibid.*

> If one were able to solve the Entscheidungsproblem then one would have a "procedure for solving many (or even all) mathematical problems".
> — *ibid.*, p. 92

By the 1928 international congress of mathematicians, Hilbert "made his questions quite precise. First, was mathematics *complete* ... Second, was mathematics *consistent* ... And thirdly, was mathematics *decidable*?" (Hodges p. 91, Hawking p. 1121). The first two questions were answered in 1930 by Kurt Gödel at the very same meeting where Hilbert delivered his retirement speech (much to the chagrin of Hilbert); the third—the Entscheidungsproblem—had to wait until the mid-1930s.

The problem was that an answer first required a precise definition of "*definite general applicable prescription*", which Princeton professor Alonzo Church would come to call "effective calculability", and in 1928 no such definition existed. But over the next 6–7 years Emil Post developed his definition of a worker moving from room to room writing and erasing marks per a list of instructions (Post 1936), as did Church and his two students Stephen Kleene and J. B. Rosser by use of Church's lambda-calculus and Gödel's recursion theory (1934). Church's paper (published 15 April 1936) showed that the Entscheidungsproblem was indeed "undecidable" and beat Turing to the punch by almost a year (Turing's paper submitted 28 May 1936, published January 1937). In the meantime, Emil Post submitted a brief paper in the fall of 1936, so Turing at least had priority over Post. While Church refereed Turing's paper, Turing had time to study Church's paper and add an Appendix where he sketched a proof that Church's lambda-calculus and his machines would compute the same functions.

> But what Church had done was something rather different, and in a certain sense weaker. ... the Turing construction was more direct, and provided an argument from first principles, closing the gap in Church's demonstration.
> — Hodges p. 112

And Post had only proposed a definition of calculability and criticized Church's "definition", but had proved nothing.

9.3 Alan Turing's a- (automatic-)machine

In the spring of 1935, Turing as a young Master's student at King's College Cambridge, UK, took on the challenge; he had been stimulated by the lectures of the logician M. H. A. Newman "and learned from them of Gödel's work and the Entscheidungsproblem ... Newman used the word 'mechanical' ... In his obituary of Turing 1955 Newman writes:

> To the question 'what is a "mechanical" process?' Turing returned the characteristic answer 'Something that can be done by a machine' and he embarked on the highly congenial task of analysing the general notion of a computing machine.
> — Gandy, p. 74

Gandy states that:

> I suppose, but do not know, that Turing, right from the start of his work, had as his goal a proof of the undecidability of the Entscheidungsproblem. He told me that the 'main idea' of the paper came to him when he was lying in Grantchester meadows in the summer of 1935. The 'main idea' might have either been his analysis of computation or his realization that there was a universal machine, and so a diagonal argument to prove unsolvability.
> — *ibid.*, p. 76

While Gandy believed that Newman's statement above is "misleading", this opinion is not shared by all. Turing had a lifelong interest in machines: "Alan had dreamt of inventing typewriters as a boy; [his mother] Mrs. Turing had a typewriter; and he could well have begun by asking himself what was meant by calling a typewriter 'mechanical'" (Hodges p. 96). While at Princeton pursuing his PhD, Turing built a Boolean-logic multiplier (see below). His PhD thesis, titled "Systems of Logic Based on Ordinals", contains the following definition of "a computable function":

> It was stated above that 'a function is effectively calculable if its values can be found by some purely mechanical process'. We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a machine. It is possible to give a mathematical description, in a certain normal form, of the structures of these machines. The development of these ideas leads to the author's definition of a computable function, and to an identification of computability with effective calculability. It is not difficult, though somewhat laborious, to prove that these three definitions [the 3rd is the λ-calculus] are equivalent.
> — Turing (1939) in *The Undecidable*, p. 160

When Turing returned to the UK he ultimately became jointly responsible for breaking the German secret codes created by encryption machines called "The Enigma"; he also became involved in the design of the ACE (Automatic Computing Engine), "[Turing's] ACE proposal was effectively self-contained, and its roots lay not in the EDVAC [the USA's initiative], but in his own universal machine" (Hodges p. 318). Arguments still continue concerning the origin and nature of what has been named by Kleene (1952) Turing's Thesis. But what Turing *did prove* with his computational-machine model appears in his paper *On Computable Numbers, With an Application to the Entscheidungsproblem* (1937):

> [that] the Hilbert Entscheidungsproblem can have no solution ... I propose, therefore to show that there can be no general process

> for determining whether a given formula U of the functional calculus K is provable, i.e. that there can be no machine which, supplied with any one U of these formulae, will eventually say whether U is provable.
> — from Turing's paper as reprinted in *The Undecidable*, p. 145

Turing's example (his second proof): If one is to ask for a general procedure to tell us: "Does this machine ever print 0", the question is "undecidable".

9.4 1937–1970: The "digital computer", the birth of "computer science"

In 1937, while at Princeton working on his PhD thesis, Turing built a digital (Boolean-logic) multiplier from scratch, making his own electromechanical relays (Hodges p. 138). "Alan's task was to embody the logical design of a Turing machine in a network of relay-operated switches ..." (Hodges p. 138). While Turing might have been just initially curious and experimenting, quite-earnest work in the same direction was going in Germany (Konrad Zuse (1938)), and in the United States (Howard Aiken) and George Stibitz (1937); the fruits of their labors were used by the Axis and Allied military in World War II (cf Hodges p. 298–299). In the early to mid-1950s Hao Wang and Marvin Minsky reduced the Turing machine to a simpler form (a precursor to the Post-Turing machine of Martin Davis); simultaneously European researchers were reducing the new-fangled electronic computer to a computer-like theoretical object equivalent to what was now being called a "Turing machine". In the late 1950s and early 1960s, the coincidentally parallel developments of Melzak and Lambek (1961), Minsky (1961), and Shepherdson and Sturgis (1961) carried the European work further and reduced the Turing machine to a more friendly, computer-like abstract model called the counter machine; Elgot and Robinson (1964), Hartmanis (1971), Cook and Reckhow (1973) carried this work even further with the register machine and random access machine models—but basically all are just multi-tape Turing machines with an arithmetic-like instruction set.

9.5 1970–present: the Turing machine as a model of computation

Today, the counter, register and random-access machines and their sire the Turing machine continue to be the models of choice for theorists investigating questions in the theory of computation. In particular, computational complexity theory makes use of the Turing machine:

> Depending on the objects one likes to manipulate in the computations (numbers like nonnegative integers or alphanumeric strings), two models have obtained a dominant position in machine-based complexity theory:
>
> *the off-line multitape Turing machine...*, which represents the standard model for string-oriented computation, and
>
> the *random access machine (RAM)* as introduced by Cook and Reckhow ..., which models the idealized Von Neumann style computer.
> — van Emde Boas 1990:4

> Only in the related area of analysis of algorithms this role is taken over by the RAM model.
> — van Emde Boas 1990:16

10 See also

11 Notes

[1] Minsky 1967:107 "In his 1936 paper, A. M. Turing defined the class of abstract machines that now bear his name. A Turing machine is a finite-state machine associated with a special kind of environment -- its tape -- in which it can store (and later recover) sequences of symbols", also Stone 1972:8 where the word "machine" is in quotation marks.

[2] Stone 1972:8 states "This "machine" is an abstract mathematical model", also cf Sipser 2006:137ff that describes the "Turing machine model". Rogers 1987 (1967):13 refers to "Turing's characterization", Boolos Burgess and Jeffrey 2002:25 refers to a "specific kind of idealized machine".

[3] Sipser 2006:137 "A Turing machine can do everything that a real computer can do".

[4] cf Sipser 2002:137. Also Rogers 1987 (1967):13 describes "a paper tape of infinite length in both directions". Minsky 1967:118 states "The tape is regarded as infinite in both directions". Boolos Burgess and Jeffrey 2002:25 include the possibility of "there is someone stationed at each end to add extra blank squares as needed".

[5] cf Rogers 1987 (1967):13. Other authors use the word "square" e.g. Boolos Burgess Jeffrey 2002:35, Minsky 1967:117, Penrose 1989:37.

[6] The word used by e.g. Davis 2000:151

[7] This table represents an algorithm or "effective computational procedure" which is necessarily finite; see Penrose 1989:30ff, Stone 1972:3ff.

[8] Boolos Burgess and Jeffrey 2002:25

[9] Boolos Burgess Jeffry 2002:25 illustrate the machine as moving along the tape. Penrose 1989:36-37 describes himself as "uncomfortable" with an infinite tape observing that it "might be hard to shift!"; he "prefer[s] to think of the tape as representing some external environment through which our finite device can move" and after observing that the " 'movement' is a convenient way of picturing things" and then suggests that "the device receives all its input from this environment.

[10] "Also by convention one of the states is distinguished as the stopping state and is given the name HALT" (Stone 1972:9). Turing's original description did not include a HALT instruction but he did allow for a "circular" condition, a "configuration from which there is no possible move" (see Turing 1936 in *The Undecidable* 1967:119); this notion was added in the 1950s; see more at Halting problem.

[11] Andrew Hodges (2012). *Alan Turing: The Enigma (THE CENTENARY EDITION)*. Princeton University Press. ISBN 978-0-691-15564-7.

[12] The idea came to him in mid-1935 (perhaps, see more in the History section) after a question posed by M. H. A. Newman in his lectures: "Was there a definite method, or as Newman put it, a *mechanical process* which could be applied to a mathematical statement, and which would come up with the answer as to whether it was provable" (Hodges 1983:93). Turing submitted his paper on 31 May 1936 to the London Mathematical Society for its *Proceedings* (cf Hodges 1983:112), but it was *published* in early 1937 and offprints were available in February 1937 (cf Hodges 1983:129).

[13] See footnote in Davis 2000:151.

[14] Turing 1936 in *The Undecidable* 1965:132-134; Turing's definition of "circular" is found on page 119.

[15] Turing 1936 in *The Undecidable* 1965:145

[16] Sipser 2006:137 observes that "A Turing machine can do everything that a real computer can do. Nevertheless, even a Turing machine cannot solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation."

[17] See the definition of "innings" on Wiktionary

[18] A.M. Turing (1948). "Intelligent Machinery (manuscript)". The Turing Archive. p. 3.

[19] Occasionally called an **action table** or **transition function**

[20] Usually quintuples [5-tuples]: $q_i a_j \rightarrow q_{i1} a_{j1} d_k$, but sometimes quadruples [4-tuples].

[21] p.149; in particular, Hopcroft and Ullman assume that δ is undefined on all states from F

12 References

12.1 Primary literature, reprints, and compilations

- B. Jack Copeland ed. (2004), *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life plus The Secrets of Enigma,* Clarendon Press (Oxford University Press), Oxford UK, ISBN 0-19-825079-7. Contains the Turing papers plus a draft letter to Emil Post re his criticism of "Turing's convention", and Donald W. Davies' *Corrections to Turing's Universal Computing Machine*

- Martin Davis (ed.) (1965), *The Undecidable*, Raven Press, Hewlett, NY.

- Emil Post (1936), "Finite Combinatory Processes—Formulation 1", *Journal of Symbolic Logic*, 1, 103–105, 1936. Reprinted in *The Undecidable* pp. 289ff.

- Emil Post (1947), "Recursive Unsolvability of a Problem of Thue", *Journal of Symbolic Logic*, vol. 12, pp. 1–11. Reprinted in *The Undecidable* pp. 293ff. In the Appendix of this paper Post comments on and gives corrections to Turing's paper of 1936–1937. In particular see the footnotes 11 with corrections to the universal computing machine coding and footnote 14 with comments on Turing's first and second proofs.

- Turing, A.M. (1936). "On Computable Numbers, with an Application to the Entscheidungs problem". *Proceedings of the London Mathematical Society*. 2 (published 1937). **42**: 230–265. doi:10.1112/plms/s2-42.1.230. (and Turing, A.M. (1938). "On Computable Numbers, with an Application to the Entscheidungsproblem: A correction". *Proceedings of the London Mathematical Society*. 2 (published 1937). **43** (6): 544–6. doi:10.1112/plms/s2-43.6.544.). Reprinted in many collections, e.g. in *The Undecidable* pp. 115–154; available on the web in many places.

- Alan Turing, 1948, "Intelligent Machinery." Reprinted in "Cybernetics: Key Papers." Ed. C.R. Evans and A.D.J. Robertson. Baltimore: University Park Press, 1968. p. 31. Reprinted in Turing, A. M. (1996). "Intelligent Machinery, A Heretical Theory". *Philosophia Mathematica*. **4** (3): 256. doi:10.1093/philmat/4.3.256.

- F. C. Hennie and R. E. Stearns. *Two-tape simulation of multitape Turing machines*. JACM, 13(4):533–546, 1966.

12.2 Computability theory

- Boolos, George; Richard Jeffrey (1999) [1989]. *Computability and Logic* (3rd ed.). Cambridge UK:

Cambridge University Press. ISBN 0-521-20402-X.

- Boolos, George; John Burgess; Richard Jeffrey (2002). *Computability and Logic* (4th ed.). Cambridge UK: Cambridge University Press. ISBN 0-521-00758-5. Some parts have been significantly rewritten by Burgess. Presentation of Turing machines in context of Lambek "abacus machines" (cf Register machine) and recursive functions, showing their equivalence.
- Taylor L. Booth (1967), *Sequential Machines and Automata Theory*, John Wiley and Sons, Inc., New York. Graduate level engineering text; ranges over a wide variety of topics, Chapter IX *Turing Machines* includes some recursion theory.
- Martin Davis (1958). *Computability and Unsolvability*. McGraw-Hill Book Company, Inc, New York.. On pages 12–20 he gives examples of 5-tuple tables for Addition, The Successor Function, Subtraction (x ≥ y), Proper Subtraction (0 if x < y), The Identity Function and various identity functions, and Multiplication.
- Davis, Martin; Ron Sigal; Elaine J. Weyuker (1994). *Computability, Complexity, and Languages and Logic: Fundamentals of Theoretical Computer Science* (2nd ed.). San Diego: Academic Press, Harcourt, Brace & Company. ISBN 0-12-206382-1.
- Hennie, Fredrick (1977). *Introduction to Computability*. Addison–Wesley, Reading, Mass. QA248.5H4 1977.. On pages 90–103 Hennie discusses the UTM with examples and flow-charts, but no actual 'code'.
- John Hopcroft and Jeffrey Ullman, (1979). *Introduction to Automata Theory, Languages, and Computation* (1st ed.). Addison–Wesley, Reading Mass. ISBN 0-201-02988-X. A difficult book. Centered around the issues of machine-interpretation of "languages", NP-completeness, etc.
- Hopcroft, John E.; Rajeev Motwani; Jeffrey D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Reading Mass: Addison–Wesley. ISBN 0-201-44124-1. Distinctly different and less intimidating than the first edition.
- Stephen Kleene (1952), *Introduction to Metamathematics*, North–Holland Publishing Company, Amsterdam Netherlands, 10th impression (with corrections of 6th reprint 1971). Graduate level text; most of Chapter XIII *Computable functions* is on Turing machine proofs of computability of recursive functions, etc.
- Knuth, Donald E. (1973). *Volume 1/Fundamental Algorithms: The Art of computer Programming* (2nd ed.). Reading, Mass.: Addison–Wesley Publishing Company.. With reference to the role of Turing machines in the development of computation (both hardware and software) see 1.4.5 *History and Bibliography* pp. 225ff and 2.6 *History and Bibliography*pp. 456ff.
- Zohar Manna, 1974, *Mathematical Theory of Computation*. Reprinted, Dover, 2003. ISBN 978-0-486-43238-0
- Marvin Minsky, *Computation: Finite and Infinite Machines*, Prentice–Hall, Inc., N.J., 1967. See Chapter 8, Section 8.2 "Unsolvability of the Halting Problem." Excellent, i.e. relatively readable, sometimes funny.
- Christos Papadimitriou (1993). *Computational Complexity* (1st ed.). Addison Wesley. ISBN 0-201-53082-1. Chapter 2: Turing machines, pp. 19–56.
- Hartley Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, The MIT Press, Cambridge MA, paperback edition 1987, original McGraw-Hill edition 1967, ISBN 0-262-68052-1 (pbk.)
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. Chapter 3: The Church–Turing Thesis, pp. 125–149.
- Stone, Harold S. (1972). *Introduction to Computer Organization and Data Structures* (1st ed.). New York: McGraw–Hill Book Company. ISBN 0-07-061726-0.
- Peter van Emde Boas 1990, *Machine Models and Simulations*, pp. 3–66, in Jan van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, The MIT Press/Elsevier, [place?], ISBN 0-444-88071-2 (Volume A). QA76.H279 1990. Valuable survey, with 141 references.

12.3 Church's thesis

- Nachum Dershowitz; Yuri Gurevich (September 2008). "A natural axiomatization of computability and proof of Church's Thesis" (PDF). *Bulletin of Symbolic Logic*. **14** (3). Retrieved 2008-10-15.
- Roger Penrose (1990) [1989]. *The Emperor's New Mind* (2nd ed.). Oxford University Press, New York. ISBN 0-19-851973-7.

12.4 Small Turing machines

- Rogozhin, Yurii, 1998, "A Universal Turing Machine with 22 States and 2 Symbols", *Romanian*

Journal Of Information Science and Technology, 1(3), 259–265, 1998. (surveys known results about small universal Turing machines)

- Stephen Wolfram, 2002, *A New Kind of Science*, Wolfram Media, ISBN 1-57955-008-8
- Brunfiel, Geoff, Student snags maths prize, *Nature*, October 24. 2007.
- Jim Giles (2007), Simplest 'universal computer' wins student $25,000, New Scientist, October 24, 2007.
- Alex Smith, Universality of Wolfram's 2, 3 Turing Machine, Submission for the Wolfram 2, 3 Turing Machine Research Prize.
- Vaughan Pratt, 2007, "Simple Turing machines, Universality, Encodings, etc.", FOM email list. October 29, 2007.
- Martin Davis, 2007, "Smallest universal machine", and Definition of universal Turing machine FOM email list. October 26–27, 2007.
- Alasdair Urquhart, 2007 "Smallest universal machine", FOM email list. October 26, 2007.
- Hector Zenil (Wolfram Research), 2007 "smallest universal machine", FOM email list. October 29, 2007.
- Todd Rowland, 2007, "Confusion on FOM", Wolfram Science message board, October 30, 2007.
- Olivier and Marc RAYNAUD, 2014, A programmable prototype to achieve Turing machines" LIMOS Laboratory of Blaise Pascal University (Clermont-Ferrand in France).

12.5 Other

- Martin Davis (2000). *Engines of Logic: Mathematicians and the origin of the Computer* (1st ed.). W. W. Norton & Company, New York. ISBN 0-393-32229-7.
- Robin Gandy, "The Confluence of Ideas in 1936", pp. 51–102 in Rolf Herken, see below.
- Stephen Hawking (editor), 2005, *God Created the Integers: The Mathematical Breakthroughs that Changed History*, Running Press, Philadelphia, ISBN 978-0-7624-1922-7. Includes Turing's 1936–1937 paper, with brief commentary and biography of Turing as written by Hawking.
- Rolf Herken (1995). *The Universal Turing Machine—A Half-Century Survey*. Springer Verlag. ISBN 3-211-82637-8.
- Andrew Hodges, *Alan Turing: The Enigma*, Simon and Schuster, New York. Cf Chapter "The Spirit of Truth" for a history leading to, and a discussion of, his proof.
- Ivars Peterson (1988). *The Mathematical Tourist: Snapshots of Modern Mathematics* (1st ed.). W. H. Freeman and Company, New York. ISBN 0-7167-2064-7.
- Roger Penrose, *The Emporer's New Mind: Concerning Computers, Minds, and the Laws of Physics*, Oxford University Press, Oxford and New York, 1989 (1990 corrections), ISBN 0-19-851973-7.
- Paul Strathern (1997). *Turing and the Computer—The Big Idea*. Anchor Books/Doubleday. ISBN 0-385-49243-X.
- Hao Wang, "A variant to Turing's theory of computing machines", *Journal of the Association for Computing Machinery* (JACM) 4, 63–92 (1957).
- Charles Petzold, Petzold, Charles, *The Annotated Turing*, John Wiley & Sons, Inc., ISBN 0-470-22905-5
- Arora, Sanjeev; Barak, Boaz, "Complexity Theory: A Modern Approach", Cambridge University Press, 2009, ISBN 978-0-521-42426-4, section 1.4, "Machines as strings and the universal Turing machine" and 1.7, "Proof of theorem 1.9"
- Kantorovitz, Isaiah Pinchas (December 1, 2005). "A note on turing machine computability of rule driven systems". *SIGACT News*. ACM. **36** (4): 109–110. doi:10.1145/1107523.1107525. Retrieved April 6, 2014.

13 External links

- Hazewinkel, Michiel, ed. (2001), "Turing machine", *Encyclopedia of Mathematics*, Springer, ISBN 978-1-55608-010-4
- Turing Machine on Stanford Encyclopedia of Philosophy
- Detailed info on the Church–Turing Hypothesis (Stanford Encyclopedia of Philosophy)
- Web Turing Machine Web application to construct and execute Turing machines (Javascript)
- Turing Machine-Like Models in Molecular Biology, to understand life mechanisms with a DNA-tape processor.
- The Turing machine—Summary about the Turing machine, its functionality and historical facts

- The Wolfram 2,3 Turing Machine Research Prize—Stephen Wolfram's $25,000 prize for the proof or disproof of the universality of the potentially smallest universal Turing Machine. The contest has ended, with the proof affirming the machine's universality.
- "Turing Machine Causal Networks" by Enrique Zeleny, Wolfram Demonstrations Project.
- Turing Machines at DMOZ
- Purely mechanical Turing Machine
- JSTMSimulator: an open source Turing Machine simulator, written in JavaScript. (source code on GitHub)
- How Alan Turing Cracked The Enigma Code Imperial War Museums

14 Text and image sources, contributors, and licenses

14.1 Text

- **Turing machine** *Source:* https://en.wikipedia.org/wiki/Turing_machine?oldid=740893405 *Contributors:* Damian Yerrick, AxelBoldt, Derek Ross, LC~enwiki, Brion VIBBER, Bryan Derksen, Robert Merkel, The Anome, Jan Hidders, Andre Engels, XJaM, Arvindn, Roadrunner, SimonP, Heron, Olivier, Stevertigo, Edward, Patrick, Chas zzz brown, Tillwe, Michael Hardy, Wshun, Dominus, Cole Kitchen, Lousyd, Rp, Kku, Liftarn, Zeno Gantner, TakuyaMurata, Loisel, Ellywa, Ahoerstemeier, Nanshu, Snoyes, Александър, LittleDan, IMSoP, Tristanb, Ehn, Barak~enwiki, Alex Vinokur~enwiki, Ec5618, Eszett, Timwi, Dcoetzee, Paul Stansifer, Fuzheado, Doradus, Populus, Head, Spikey, Shizhao, Topbanana, Khym Chanur, Raul654, Cheran~enwiki, BenRG, Phil Boswell, Robbot, Jaredwf, Kadin2048, Altenmann, MathMartin, Saforrest, Nitishkorula, Nikitadanilov, Tea2min, David Koller, Centrx, Giftlite, Brouhaha, Drunkasian, Mousomer, Meursault2004, MSGJ, Dratman, Wikiwikifast, Tom-, Sundar, Golbez, Gubbubu, CryptoDerk, Gdr, Knutux, Slowking Man, LucasVB, Antandrus, Q17, Oneiros, Kntg, Kevin143, B.d.mills, Creidieki, Karl Dickman, D6, Mormegil, Gachet, DanielCristofani, Smimram, RossPatterson, Jayc, Byrial, Plumpy, Kaisershatner, Alamino, Ferkel, Obradovic Goran, Nynexman4464~enwiki, Blahma, Opticon, Sligocki, Yipdw, MoraSique, Cal 1234, Artur adib, GabrielF, Gene Nygaard, Blaxthos, Oleg Alexandrov, TShilo12, Nuno Tavares, Shreevatsa, LoopZilla, MattGiuca, Ruud Koot, -Ril-, Slike2, Smmurphy, Graham87, BD2412, Qwertyus, BorgHunter, Rjwilmsi, Zbxgscqf, Isaac Rabinovitch, Strangethingintheland, SpNeo, Bensin, Penumbra2000, Alejo2083, FlaBot, Chris Pressey, RobertG, Mathbot, Nihiltres, Harmil, Pexatus, David H Braun (1964), Jidan, Chobot, Ahpook, Metaeducation, YurikBot, Wavelength, Schadel, Hairy Dude, Ventolin, Arado, Iamfscked, Shell Kinney, Gaius Cornelius, Member, AdamPeterman, Trovatore, R.e.s., Hzenilc, ArmadniGeneral, Cholmes75, Jpbowen, DYLAN LENNON~enwiki, King mike, Ott2, LarryLACa, K.Nevelsteen, Glome83, Abune, Claygate, GrinBot~enwiki, Syko, Jinwicked, That Guy, From That Show!, GrafZahl, SmackBot, Mmernex, Reedy, InverseHypercube, Hydrogen Iodide, Verne Equinox, Gilliam, Sviemeister, Ben.c.roberts, DroEsperanto, Droll, J. Spencer, Calibwam, Dzonatas, Can't sleep, clown will eat me, Readams, Frap, Allan McInnes, Grover cleveland, Bigmantonyd, Polymath69, Pfhyde, HarisM, Jon Awbrey, Wvbailey, Edetic, Yan Kuligin, RCX, JorisvS, Tarcieri, Makyen, Dicklyon, Therebelcountry, Hu12, Iridescent, JMK, Rob-nick, Blehfu, Tawkerbot2, JRSpriggs, DKqwerty, Merzbow, JForget, CRGreathouse, CmdrObot, CBM, Maester mensch, Kris Schnee, Lmcelhiney, Ieee8023, WeggeBot, Cydebot, Krauss, Gtxfrance, Marco.caminati, Pascal.Tesson, UberScienceNerd, Malleus Fatuorum, Jeph paul, Gioto, QuiteUnusual, Prolog, Apocalyps956, Ad88110, Vineetgupta, VictorAnyakin, P.L.A.R., JAnDbot, XyBot, Asmeurer, Davewho2, NapoliRoma, Thenub314, Jheiv, Gavia immer, Eus Kevin, Theroadislong, Gwythoff, David Eppstein, Martynas Patasius, GermanX, Pet5, Ftiercel, Gwern, MartinBot, Pagw, Alain Vey, Aliazimi, Ramesh Chandra, GrahamDavies, Brest, Mikael Häggström, Tarotcards, Policron, Pleasantville, Satyr9, Philip Trueman, Lingwitt, TXiKiBoT, MDoggNoGFresh, Hqb, Anonymous Dissident, Ask123, Ocolon, TedColes, Ilia Kr., Wuffe~enwiki, Sheerfirepower, Duncan.Hull, Ewakened, Eubulides, Napoleon Dynamite42, Billinghurst, Wolfrock, Alaniaris, Calliopejen1, Parhamr, Mrengy, WikiTony999, Reinderien, Chridd, Svick, Cloversmate, ClueBot, Matman132, DavisSta, Supertouch, Laminatrix, Greenmatter, Whatsthatpicture, BearMachine, Sun Creator, Cgay88, Psinu, Jpmelos, Johnuniq, Wednesday Next, XLinkBot, Mitch Ames, Addbot, Kne1p, Ender2101, Rjpryan, Abovechief, Ld100, Tide rolls, س ع ى, Luckas-bot, Yobot, Themfromspace, OrgasGirl, JSimmonz, Pcap, Carleas, Punctilius, AnomieBOT, Jim1138, JackieBot, ErikTheBikeMan, Materialscientist, Citation bot, Diego Queiroz, HenryCorp, Miym, GrouchoBot, Pritamworld, False vacuum, RibotBOT, Anton203, IShadowed, Cocteau834, Kirsted, TobiasKlaus, Howard McCay, Jsorr, Altg20April2nd, Machine Elf 1735, Cannolis, HamburgerRadio, OgreBot, Citation bot 1, Pinethicket, Jonesey95, Three887, Tom.Reding, Heooo, Dac04, Jauhienij, TobeBot, Trappist the monk, Lotje, Vrenator, Lokentaren, DARTH SIDIOUS 2, TomT0m, Ashutosh y0078, DASHBot, EmausBot, Jurvetson2, BillyPreset, Dcirovic, Thecheesykid, ZéroBot, HeyStopThat, Mvanveen, L Kensington, Nagato, ClueBot NG, Accelerometer, Yourmomblah, Frietjes, Thepigdog, ScottSteiner, Widr, Helpful Pixie Bot, Strike Eagle, Jaberrio, Vasyaivanov, BG19bot, MusikAnimal, 1xdd0ufhgnlsoprfgd, Glacialfox, Duxwing, Rbkillea, BattyBot, Ronin712, Streakofhope, Dexbot, Klickagent, Fleuryeric, Jochen Burghardt, Jiawhein, 7804j, Asierog, Ggwine, Wenzchen, Alexakh, Jakenath, JpurvisUM, Chrsimon, Haeisen, Ahasn, Szhu008, Zicane, Melcous, Monkbot, Kalyanam17, WholeWheatBagel, Raynaudmarc, J.V.Cimrman, Kyle1009, Anastasiapapadopoulou, MasaComp, Velvel2, Allo312, Federico.aponte, Anaszt5, KasparBot, Seventhorbitday, Franciscolopezsanchoabraham, GreenC bot, Fmadd, Sube wiki, Arkenidar and Anonymous: 438

14.2 Images

- **File:Commons-logo.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/4/4a/Commons-logo.svg *License:* CC-BY-SA-3.0 *Contributors:* ? *Original artist:* ?
- **File:Lego_Turing_Machine.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/7/7b/Lego_Turing_Machine.jpg *License:* CC BY 3.0 *Contributors:* Projet Rubens, ENS Lyon, http://rubens.ens-lyon.fr/fr/gallery/ *Original artist:* Projet Rubens, ENS Lyon
- **File:Model_of_a_Turing_machine.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/a/ad/Model_of_a_Turing_machine.jpg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* GabrielF
- **File:Question_book-new.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/9/99/Question_book-new.svg *License:* Cc-by-sa-3.0 *Contributors:* ? *Original artist:* ?
- **File:State_diagram_3_state_busy_beaver_2B.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/4/4b/State_diagram_3_state_busy_beaver_2B.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Diego Queiroz
- **File:State_diagram_3_state_busy_beaver_4.JPG** *Source:* https://upload.wikimedia.org/wikipedia/commons/2/26/State_diagram_3_state_busy_beaver_4.JPG *License:* CC-BY-SA-3.0 *Contributors:* wvbaileyWvbailey 12:33, 10 August 2006 (UTC), original drawing by wvbailey in Autosketch *Original artist:* Wvbailey at English Wikipedia
- **File:Turing_machine_2a.svg** *Source:* https://upload.wikimedia.org/wikipedia/en/0/09/Turing_machine_2a.svg *License:* Cc-by-sa-3.0 *Contributors:*
  self-made
  Original artist:
  User:Nynexman4464
- **File:Turing_machine_2b.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/a/a2/Turing_machine_2b.svg *License:* Public domain *Contributors:* Own work (Original text: *self-made*) *Original artist:* User:Nynexman4464
- **File:Turingmachine.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/8/8f/Turingmachine.jpg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Raynaudmarc

14.3 Content license

Chapitre 14

Machines de Turing

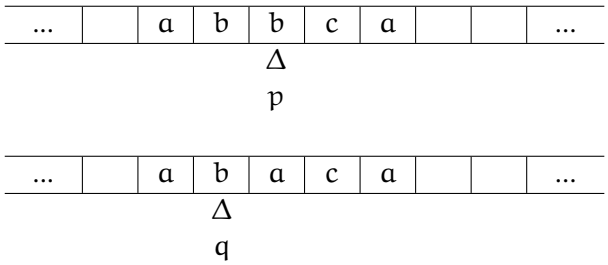
Dans ce chapitre on présente un modèle de calcul introduit dans les années 30 par Turing, les *machines de Turing*. Ces machines formalisent la notion de calculabilité. La thèse de Church affirme que tout calcul par un algorithme effectif peut-être effectué par une machine de Turing.

14.1. Définition et fonctionnement

Les automates représentent un modèle de calcul utilisant une mémoire finie correspondant à l'ensemble des états de la machine. Les machines de Turing sont également des machines "finies" (correspondant à des algorithmes effectifs), mais elles utilisent une mémoire plus élaborée. Elles utilisent un (ou des) ruban "infini" de cases mémoires qu'elles lisent et réécrivent.

Le fonctionnement d'une machine de Turing peut se représenter de manière informelle par une tête de lecture se trouvant sur un ruban qui à la lecture de la case mémoire correspondante change d'état, modifie la case mémoire et se déplace sur le ruban à droite ou à gauche

Exemple. Machine qui lisant un b dans l'état p, passe à l'état q, écrit un a à la place du b et déplace sa tête de lecture à gauche.

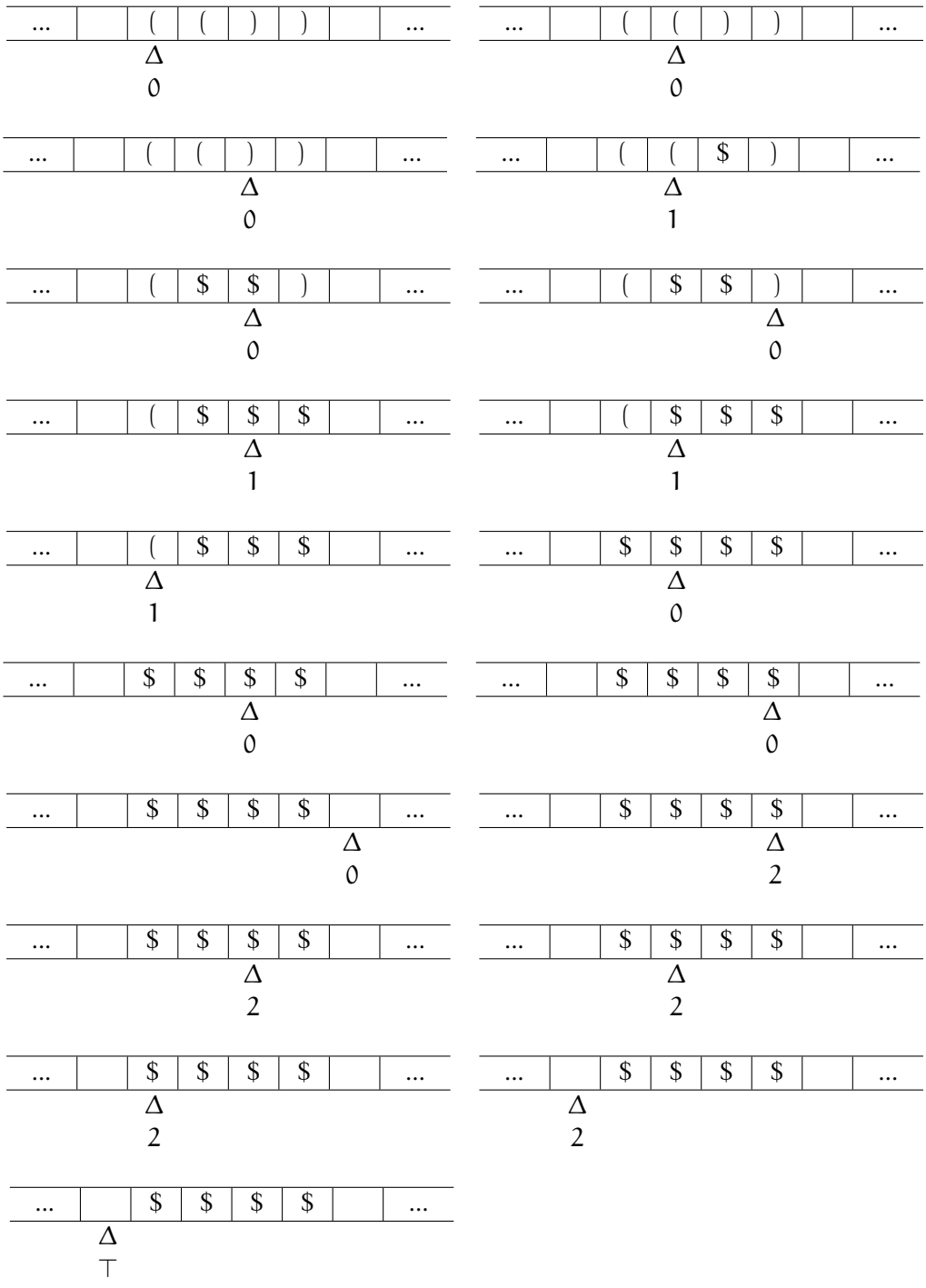


14.1.1. Définition. Une *machine de Turing* (à un ruban) est la donnée
- d'un alphabet fini Σ auquel l'on ajoute un symbole $\square$ représentant les cases vides, formant ainsi un alphabet Σ_+ ;
- d'un ensemble fini d'états Q auquel l'on ajoute deux états finaux, un état acceptant $\top$ et un état refusant $\bot$, formant ainsi un ensemble d'états Q_+ ;
- d'une fonction de changement d'état $\delta : Q \times \Sigma_+ \to Q_+$;
- d'une fonction d'écriture $\sigma : Q \times \Sigma_+ \to \Sigma_+$;
- d'une fonction de déplacement $\Delta : Q \times \Sigma_+ \to \{G, D\}$;
- d'un état initial $q_0 \in Q$.

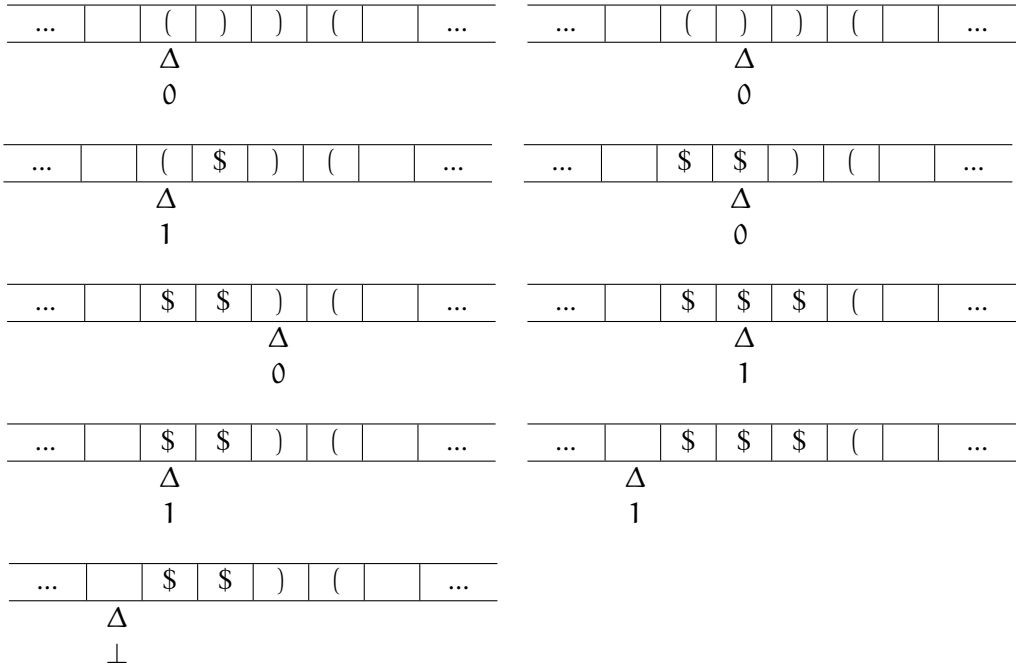
14.1.2. Exemple. Le tableau ci-dessous représente les transitions d'une machine de Turing d'alphabet $\{(,), \$\}$ et d'états $\{0, 1, 2\}$.

		(	)	□	$
→	0	0, (, D	1, $, G	2, □, G	0, $, D
	1	0, $, D	⊥	⊥	1, $, G
	2	⊥	⊥	⊤	2, $, G

Cette machine de Turing permet en fait de reconnaître le langage bien parenthésé L. Voici son fonctionnement sur le mot $(())$:



Son fonctionnement sur le mot $())($:



14.2. UTILISATION : DÉCIDABILITÉ ET CALCULABILITÉ

Afin de formaliser le fonctionnement d'une machine de Turing, on utilise la notation $u \uparrow v$ pour représenter la configuration du ruban sur lequel est écrit le mot uv avec sa tête de lecture placée au début du mot v et où toutes las autres cases du ruban sont vides. On note

$$(p, u \uparrow v) \vdash (q, u' \uparrow v')$$

la transition de l'état p avec la configuration de ruban $u \uparrow v$ à l'état q avec la configuration de ruban $(q, u' \uparrow v')$.

14.2.1. Mots acceptés, mots refusés. Soient M une machine de Turing d'alphabet Σ et u un mot sur Σ. On dit M accepte (respectivement refuse) u si le fonctionnement de M à partir de la configuration $(p_0, \uparrow u)$ mène en un nombre fini de transitions à l'état acceptant $\top$ (respectivement l'état refusant $\bot$).

Remarque. Une machine de Turing peut ni accepter, ni refuser un mot donné, mais ne pas s'arrêter de fonctionner à partir de la configuration associée à ce mot. Voici un exemple de machine qui accepte les mots formés d'un nombre impair de a mais ne s'arrête pas sur les mots formés d'un nombre pair de a.

		a	$\square$
$\rightarrow$	0	$1, a, D$	$2, a, G$
	1	$0, a, D$	$\top$
	2	$2, a, G$	$1, a, D$

14.2.2. Langages (ou ensembles) décidables et semi-décidables. Soit L un langage sur Σ. On dit que L est *décidable* s'il existe une machine de Turing d'alphabet Σ qui accepte tous les mots de L et refuse tous les mots qui ne sont pas dans L. On dit que L est *semi-décidable* s'il existe une machine de Turing qui n'accepte que les mots de L.

Exemples. Les langages rationnels sont tous décidables (un automate fini est une machine de Turing particulière). Nous verrons plus loin que les langages algébriques sont également décidables. Le complémentaire de tout langage décidable est décidable. La langage non-algébrique $\{a^n b^n c^n : n \in \mathbb{N}\}$ est décidable.

14.2.3. Fonctions calculables. Une fonction partielle $f : \Sigma \to \Sigma$ est *calculable* s'il existe une machine de Turing d'alphabet Σ qui n'accepte que les mots du domaine de définition de f et qui à partir de la configuration $(p_0, \uparrow w)$ où w est un mot du domaine de définition termine son calcul avec le mot $f(w)$ sur son ruban.

On étend cette définition aux fonctions de plusieurs variables en représentant les éléments $(u_1, .., u_k) \in \Sigma^k$ par le mot $u_1 \square u_2 \square ... \square u_k \in \Sigma_+$.

Exemples. Toutes les fonctions calculées par des machines séquentielles sont calculables. La fonction de concaténation qui à $(u_1, u_2, ..., u_k)$ associe $u_1 u_2 .. u_k$ est calculable. La somme, le produit de nombres représentés dans une base donnée sont calculables. Pour la somme en base unaire, il suffit d'utiliser la concaténation. De manière générale, pour calculer ces fonctions il est plus pratique d'utiliser plusieurs rubans mémoires. On se convaincra ensuite que l'on peut simuler de telles machines avec une machine à un seul ruban.

14.3. GÉNÉRALISATIONS

Une généralisation naturelle des machines ci-dessus est l'utilisation de plusieurs rubans. Chaque ruban possède alors une tête de lecture et à chaque pas de calcul la transition dépend de la lecture des caractères sur chaque ruban.

14.3.1. Machines de Turing à plusieurs rubans. Une *machine de Turing* à k rubans est la donnée
- d'un alphabet fini Σ auquel l'on ajoute un symbole $\square$ représentant les cases vides, formant ainsi un alphabet Σ_+ ;
- d'un ensemble fini d'états Q auquel l'on ajoute deux états finaux, un état acceptant $\top$ et un état refusant $\bot$, formant ainsi un ensemble d'états Q_+ ;
- d'une fonction de changement d'état $\delta : Q \times \Sigma_+^k \to Q_+$;
- d'une fonction d'écriture $\sigma : Q \times \Sigma_+^k \to \Sigma_+^k$;
- d'une fonction de déplacement $\Delta : Q \times \Sigma_+^k \to \{G, S, D\}^k$ (S correspondant à l'absence de déplacement la tête de lecture) ;
- d'un état initial $q_0 \in Q$.

Notons $\sigma_1, ..., \sigma_k$ et $\Delta_1, ..., \Delta_k$ les fonctions coordonnées de σ et Δ. La transition à partir d'une configuration $(p, u_1 \uparrow a_1 v_1, u_2 \uparrow a_2 v_2, ..., u_k \uparrow a_k v_k)$ consiste donc à :
- passer à l'état $\delta(p, a_1, .., a_k)$,
- remplacer chaque a_i par $\sigma_i(p, a_1, .., a_k)$,
- déplacer la tête de lecture du i-ème ruban suivant la valeur de $\Delta_i(p, a_1, .., a_k)$.

14.3.2. Exemples.

1. **Automates à pile** : Les automates à piles déterministes peuvent être représentées par des machines à deux rubans, le second ruban utilisé par la pile.
2. **Addition en représentation binaire** : on considère une machine de Turing à deux rubans. Cette machine commence par placer ses têtes de lectures à la fin des représentations binaires de deux nombres placés sur ces rubans. Ensuite elle fonctionne comme l'algorithme classique de l'addition

posée. Elle lit les chifres sur chaque ruban, les additionnent avec la retenue et écrit le résultat correspondant sur le premier ruban et utilise son état pour conserver la retenue qui vaut 0 ou 1. (Rappelons que ceci peut également s'effectuer par une machine séquentielle).

3. **Multiplication en représentation binaire** : on utilise une machine à trois rubans. Le premier sert au résultat, les deux autres aux donnés. On simule alors l'algorithme clssique de la multiplication posée, en utilisant l'addition à chaque étape.
4. **Passage d'une base à une autre pour la réprésentation d'un nombre** : il suffit pour cela de passer par la base 1. Le passage de la base 1 à la base n consiste à itérer la division euclidienne par n. On compte le nombre de bloc de 1 de longueur n sur un ruban auxiliaire. Le reste se lit alors directement sur le premier ruban que l'on écrit sur un ruban résultat. On itère ensuite sur le quotient que l'on a recopié sur le premier ruban. Le passage de la base n à 1 consiste à représenter le premier chiffre en base n par un bloc de 1 correspondant sur le ruban résultat, puis à dupliquer n fois le nombre de 1 sur le ruban résultat avant de passer au chiffre suivant.

14.3.3. Simulation d'une Machine de Turing à 2 ubans par une machine de Turing à un seul ruban. Nous allons voir rapidement que le calcul d'une machine de Turing à plusieurs rubans peut-être simulé par une machine de Turing à un seul ruban. Faisons le cas de deux rubans. Le principe est de coderles configurations sur deux rubans par une configuration sur un seul ruban. Pour cela on peut par exemple utiliser alternativement deux cases mémoires du ruban de la nouvelle machine pour chacune des cases des rubans de la machine de départ, de la manières suivante.

Configuration d'une machine à 2 rubans :

...	a	b	c	c	b	...
		Δ				

...	c	a	a	a	a	...
				Δ		
				p		

Codage de cette configuration sur un ruban :

...		a		c	Δ	b		a		c		a		c	Δ	a		b		a	...
					Δ																
					p																

Dans la nouvelle machine, le ruban est décomposé par bloc de 4 cases, la première case codant le fait que la tête de lecture du premier ruban originel se trouve ici ou non, la seconde la case mémoire correspondant à celle du premier ruban originel et de même les deux cases suivantes pour le second ruban originel. Le fonctionnement de la nouvelle machine à ruban consistera à détecter les cases à lire en utilisant les marqueurs des têtes de lecture puis à simuler les transitions sur les deux rubans de la machine originelle.

Remarques. La simulation ci-dessus se généralise facilement à k rubans. Par ailleurs si l'on considère un calcul sur une machine à k-rubans partant de la configuration $(\uparrow u_1, \uparrow u_2, ..., \uparrow u_k)$ il est facile de voir via la simulation ci-dessus que l'on peut faire le même calcul avec une machine à un ruban partant de la configuration $\uparrow u_1 \square u_2 \square ... \square u_k$. Donc tous les exemples de fonctions calculées par des machines à plusieurs rubans sont calculables au senn de la définition donnée précedemment.

14.3.4. Machines de Turing non-déterministes. Une autre généralisation naturelle des machines de Turing est d'avoir un nombre fini de transitions possibles à chaque pas de calcul. Il est assez facile de montrer que de telles machines peuvent être simulées par des machines déterministes. Pour cela on

énumère au fur et à mesure l'ensemble des configurations possibles sur un ruban auxiliaire. A l'aide de telle machine, on remarque par exemple facilement que tout automate à pile (non nécessairement déterministe) peut-être représenté par une machine de Turing. En particulier les langages algébriques sont bien décidables (au sens des machines de Turing).

14.4. EXEMPLES ET CONTRE-EXEMPLES

14.4.1. Réduction de l'alphabet. En pratique la mémoire physique dans une machine correspond à des bits. Il est assez simple de voir que toute machine de Turing peut être simulée par une machine de Turing utilisant un ruban constitué de bits, c'est-à-dire d'alphabet $\Sigma_+ = \{\square, 1\}$ que l'on notera plus en général $\Sigma_+ = \{0, 1\}$.

Pour cela il suffit de remarquer que tout ensemble fini se code par un nombre fini d'entiers ayant des représentations binaire d'une longueur fixée. Prenons par exemple une machine de Turing M d'alphabet $\Sigma = \{a, b, c\}$ que l'on codera par des représentations binaires de longueur 2 :

$$\begin{array}{rcl} \square & : & 00 \\ a & : & 01 \\ b & : & 10 \\ c & : & 11 \end{array}$$

Ainsi à chaque configuration de ruban de M on associe une configuration d'une machine simulante M' en précisant que l'on place la tête de lecture sur la première des deux cases représentant l'élément de l'alphabet de M. Puis M' simule la transition définie par M en lisant la case suivante afin de déterminer le caractère lu par M. Voici un exemple :

Configuration de M (état p et lecture de a) :

...		b		a	c	a		...
				Δ				
				p				

Représentation sur le ruban de M' :

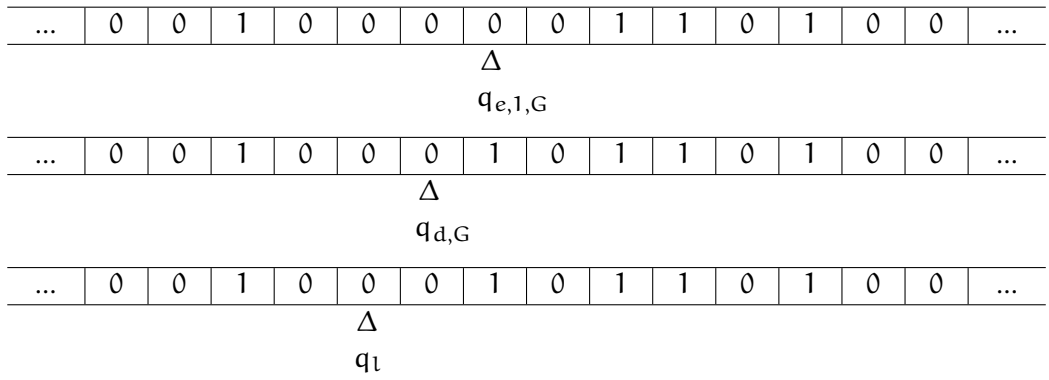
...	0	0	1	0	0	0	0	1	1	1	0	1	0	0	...
							Δ								
							p_1								

Transition de M : passage à l'état q, écriture de b et déplacement à gauche.

...		b		b	c	a		...
			Δ					
			q					

Simulation par M' en utilisant des états auxiliaires :

...	0	0	1	0	0	0	0	1	1	1	0	1	0	0	...
								Δ							
								$p_{1,0}$							



On remarque ainsi qu'à des codages simples près (codage calculables par des machines séquentielles) toute fonction calculable peut être calculée par une machine de Turing à un seul ruban d'alphabet $\Sigma_+ = \{0, 1\}$ où 0 est utilisé pour représenter les cases mémoires vides.

14.4.2. Machine de Turing universelle. Les exemples de machine de Turing que nous avons vu précédemment ne sont pas programmables ; elles effectuent un calcul prédéterminé. Nous allons maintenant décrire une machine de Turing "programmable" qui prendra en entrée sur un ruban le code d'une machine de Turing M d'alphabet $\{0, 1\}$ et simulera sur un autre ruban le fonctionnement de cette machine.

Notre machine universelle U sera constitué de trois rubans.

Premier ruban : sur ce ruban on simulera le calcul de la machine M sur un mot sur l'alphabet $\{0, 1\}$.

Second ruban : sur ce ruban on mettra sous forme codé l'ensemble des transitions de M ce qui correspondra en quelque sorte au "programme M".

Troisième ruban : On mettra le code correspondant à l'état courant de M.

Pour cela il est nécessaire de coder le tableau des transitions de M (ce code correspondra au "programme" qui sera écrit sur le premier ruban d'une machine de Turing, *machine universelle*. Nous supposerons que les états sont représentés par des entiers que l'état acceptant correspond à 1 et l'état refusant à 0. On peut alors représenter une transition $(p, l) \to (q, e, \Delta)$ de l'état $p \in \mathbb{N}$ avec la lecture de $l \in \{0, 1\}$ qui passe à l'état $q \in \mathbb{N}$, écrit $e \in \{0, 1\}$ et effectue le déplacement $\Delta \in \{G, D\}$ par

$$\#\text{bin}(p)\#\ l\ \#\text{bin}(q)\#\ e\Delta$$

où $\text{bin}(p)$ et $\text{bin}(q)$ sont les représentations binaires des entiers p et q.

On représente ensuite l'ensemble des transitions de M par

$$\$T_1T_2T_3T_4...T_n\$.$$

Notre machine universelle U est d'alphabet $\{0, 1, G, D, \$, \#\}$. À partir de la donnée de l'ensemble des transitions de M sur le second ruban, de la représentation binaire de l'état initial de M sur le troisième ruban et de la donnée d'un mot sur le premier ruban, la machine U simule le calcul de M de la façon suivante :

1. A la lecture de 0 ou 1 sur le premier ruban elle recherche en lisant sons troisième ruban la transition correspondante au calcul par M.
2. Une fois cette transition trouvée, elle simule cette transition en écrivant sur son troisième ruban l'état d'arrivée, en écrivant le caractère 0 ou 1 defini par la transition sur son premier ruban et en déplaçant la tête de lecture sur son premier ruban suivant toujours cette transition. Dans le cas où l'état d'arrivée est 0 (respectivement 1) elle se met dans son état refusant (respectivement acceptant) et s'arrête.

Remarque. Les choix ci-dessus du codage, de l'alphabet pour cela et du nombre de rubans utilisés ne sont qu'un exemple possible. Par des codages supplémentaires, on peut considérer une machine universelle à un seul ruban et qui a elle-même l'alphabet $\{0, 1\}$.

14.4.3. Problème de l'arrêt. Le problème de l'arrêt est le suivant : étant donnée une machine de Turing M et un mot w sur son alphabet, est-ce que le calcul de M à partir de ce mot w s'arrête ou non. Nous allons voir que ce problème est indécidable mais tout d'abord montrons qu'il est semi-décidable. Pour cela nous devons choisir une formalisation de ce problème : nous n'allons considérer que des machines de Turing d'alphabet $\{0, 1\}$ et considérer leurs codes (codes des ensembles des transitions) vu précédemment. Notons que l'ensemble de ces codes qui sont sur l'alphabet $\Sigma = \{0, 1, G, D, \$, \#\}$ forme un langage rationnel. Pour tout tel code e, l'on note M_e la machine de Turing correspondante. Sous cette formalisation le problème de l'arrêt correspond au langage sur Σ

$$L_{ar} = \{ex : \text{ le calcul de la machine de Turing } M_e \text{ s'arrête en partant de la configuration } \uparrow x\}.$$

Il est assez clair que la machine universelle précédente semi-décide ce langage.

Montrons maintenant que le problème de l'arrêt est indécidable. Nous allons pour cela utiliser un procédé diagonal. Tout d'abord il est facile de se rammener au langage $\{0, 1\}$, par exemple en codant les lettres de $\Sigma = \{0, 1, G, D, \$, \#\}$ de la façon suivante :

$$\begin{array}{rcl} 0 & : & 000 \\ 1 & : & 001 \\ G & : & 010 \\ D & : & 011 \\ \$ & : & 100 \\ \# & : & 101 \end{array}$$

Notons ϕ la fonction de codage. Supposons que L_{ar} soit décidable. Alors nous obtenons une contradiction par un procédé diagonal classique, en construisant une machine de Turing M' sur l'alphabet $\{0, 1\}$ qui à partir de la configuration initiale $\uparrow u$, s'arrête si et seulement si les deux conditions suivantes sont réalisées :

1. le mot u s'écrit sous la forme $\phi(e)u'$ pour un mot e qui est le code d'une machine de Turing ;
2. si la première condition est vérifiée avec le code e alors la machine M_e ne s'arrête pas sur le mot $\phi(e)$ (c'est-à-dire, $e\phi(e)$ n'appartient pas au langage L_{ar}).

La contradiction est alors obtenue en utilisant le code e' de la machine M' et en faisant opérer M' sur la configuration $\uparrow \phi(e')$. En effet :

1. ou bien M' atteint un état d'arrêt. Dans ce cas la seconde condtion n'est pas vérifiée et donc M' n'aurait pas du s'arrêter ;
2. ou bien M' ne s'arrête pas. Mais alors les deux condtions sont vérifiées et cette machine aurait du s'arrêter.

Pour construire M', commençons par l'introduction d'une telle machine à plusieurs rubans. Le premier ruban utilise l'alphabet $\{0, 1\}$ (où 0 correspond aux cases vides). Voici son fonctionnement :

1. On met la donnée u sur le premier ruban ;
2. Cette machine décode par bloc de trois lettres de u, le code e d'une machine Turing qu'elle écrit sur un second ruban. Si elle tombe sur un bloc de trois lettres ne correspondant pas à aucune lettre de Σ, elle déplace alors indéfiniment sa tête de lecture vers la droite.

3. La reconnaissance éventuelle d'un code e se fait simultanément sur le second ruban par la simulation de l'automate correspondant. La machine passe à l'étape suivant si elle a reconnu un code, sinon elle ne s'arrête pas.
4. Si la machine se trouve à cette étape en ayant trouvé le code e d'une machine de Turing, alors sur son second ruban se trouve e et sur le premier ruban un mot $\psi(e)u'$. Elle utilise alors la machine décidant le langage L_{ar} pour déterminer si M_e s'rarrête sur $\psi(e)$ ou non. Dans le cas négatif notre machine s'arrête, dans le cas positif, elle déplace indéfiniment sa tête de lecture à droite.

Remarquons maintenant que cette machine peut être simulée par une machine sur l'alphabet $\{0,1\}$ et avec un seul ruban, cela donne notre machine M'. (Il suffit d'utiliser un procédé analogue à la réduction de deux rubans à un seul. Notons qu'il faut faire un peu attention dans cette simulation qui consiste à dupliquer les cases d'un ruban, pour conserver entrée la donnée sous la même forme.)

14.4.4. Le castor affairé. On considère dans cette partie uniquement des machines de Turing d'alphabet $\Sigma_+ = \{0,1\}$. Nous mettons ces machines en compétition : le but est d'écrire le plus grand nombre de 1 et de s'arrêter ceci en partant de la configuration initiale où le ruban est vide. Pour cette compétition, on range ces machines par catégories selon leurs nombres d'états. Pour tout entier n, il n'y a, à numérotation près des états, qu'un nombre fini de machine de Turing avec n états sur l'alphabet $\{0,1\}$. Il existe donc une (ou plusieurs) machine à n états qui écrit le plus grand nombre de 1 en s'arrêtant : on notera $\mathrm{Cast}(n)$ ce nombre maximal de 1.

Nous allons montrer que l'on obtient ainsi une fonction qui n'est pas calculable.

Tout d'abord vérifions que cette fonction est strictement croissante : soient n et $\mathtt{Castor}_n$ une machine de Turing écrivant $\mathrm{Cast}(n)$ 1 sur ruban puis s'arrêtant. Avec un état supplémentaire, on peut considérer une machine qui simule la machine $\mathtt{Castor}_n$ et quand elle se trouve dans l'état final de celle-ci, reste dans cette état tant qu'elle lit des 1 en se déplaçant à droite. Dès qu'elle lit un 0 eslle écrit un 1 et se place dans son nouvel état qui remplace l'état final précédent. Cette machine à $n+1$ états écrira donc un 1 de plus.

14.4.5 Proposition. *Pour toute fonction calculable* $f : \mathbb{N} \to \mathbb{N}$, $f(n) < \mathrm{Cast}(n)$ *pour* n *assez grand. Donc* Cast *n'est pas calculable.*

Démonstration. Remarquons tout d'abord que la fonction $g(n) = \sum_{i=0}^{n} f(n)$ est également calculable. On peut donc supposer que f est croissante. Comme la fonction produit est calculable, c'est également le cas de la fonction carrée. Il suit que la fonction qui à n associe $f(n^2)$ est calculable. Considérons maintenant une machine de Turing M calculant cette fonction représentée en base 1. Soit k son nombre d'états. Soit n un entier quelconque. Il existe une machine de Turing à $n+1$ qui, à partir de la configuration vide, aboutit à la configuration $\uparrow$ 111...111 avec exactement n 1, c'est-à-dire la représentation base 1 de n sur son ruban. En composant cette machine avec M, on obtient une machine de Turiing à $n+k+1$ états qui, à partir de la configuration vide, écrit $f(n^2)$ 1. Par définition de Cast on a donc

$$f(n^2) \leq \mathrm{Cast}(n+k+1).$$

Maintenant, pour n assez grand on a

$$n+k+1 < (n-1)^2$$

d'où

$$f(n^2) < \mathrm{Cast}((n-1)^2).$$

Par croissance des deux fonctions, il suit que pour tout m assez grand

$$f(m) < \mathrm{Cast}(m).$$

□

Turing Machines

Computation has been around a very long time. Computer programs, after all, are a rather recent creation. So, we shall take what seems like a brief detour back in time in order to examine another system or model of computation. We shall not wander too far back though, merely to the mid 1930's. After all, one could go back to Aristotle, who was possibly the first Western person to develop formal computational systems and write about them.

Well before the advent of modern computing machinery, a British logician named A. M. Turing (who later became a famous World War II codebreaker) developed a computing system. In the 1930's, a little before the construction of the first electrical computer, he and several other mathematicians (including Church, Markov, Post, and Turing) independently considered the problem of specifying a system in which computation could be defined and studied.

Turing focused upon human computation and thought about the way that people compute things by hand. With this examination of human computation he designed a system in which computation could be expressed and carried out. He claimed that any nontrivial computation required:

- a simple sequence of computing instructions,
- scratch paper,
- an implement for writing and erasing,
- a reading device, and
- the ability to remember which instruction is being carried out.

Turing then developed a mathematical description of a device possessing all of the above attributes. Today, we would recognize the device that he defined as a special purpose computer. In his honor it has been named the *Turing machine*.

This heart of this machine is a *finite control box* which is wired to execute a specific *list of instructions* and thus is precisely a special purpose computer or computer chip. The device records information on a *scratch tape* during computation and has a *two-way head* that *reads* and *writes* on the tape as it moves along. Such a machine might look like that pictured in figure 1.

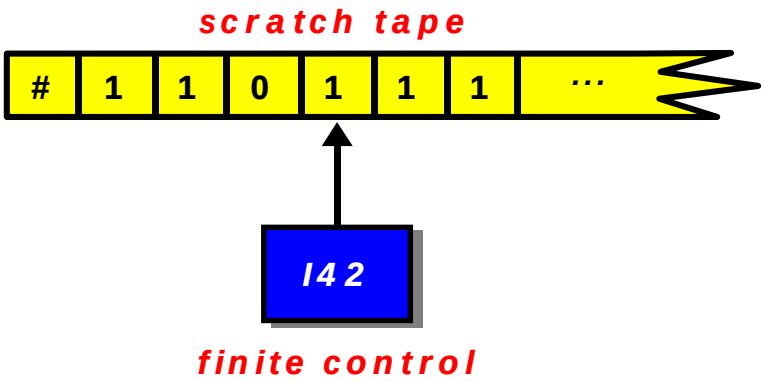


Figure 1 - A Turing Machine

A *finite control* is a simple memory device that remembers which *instruction* should be executed next. The *tape*, divided into *squares* (each of which may hold a *symbol),* is provided so that the machine may record results and refer to them during the computation. In order to have enough space to perform computation, we shall say that the tape is *arbitrarily long.* By this we mean that a machine never runs out of tape or reaches the right end of its tape. This does *NOT* mean that the tape is infinite - just long enough to do what is needed. A *tape head* that can *move* to the left and right as well as *read* and *write* connects the finite control with the tape.

If we again examine figure 1, it is evident that the machine is about to execute instruction *I42* and is reading a 1 from the tape square that is fifth from the left end of the tape. Note that we only show the portion of the tape that contains non-blank symbols and use three dots (. . .) at the right end of our tapes to indicate that the remainder is blank.

That is fine. But, what runs the machine? What exactly are these instructions which govern its every move? A *Turing machine instruction* commands the machine to perform the sequence of several simple steps indicated below.

a) read the tape square under the tape head,
b) write a symbol on the tape in that square,
c) move its tape head to the left or right, and
d) proceed to a new instruction

Steps (b) through (d) depend upon what symbol appeared on the tape square being scanned before the instruction was executed.

An instruction shall be presented in a chart that enumerates outcomes for all of the possible input combinations. Here is an example of an instruction for a machine which uses the symbols 0, 1, #, and blank.

	symbol read	*symbol written*	*head move*	*next instruction*
I93	**0**	**1**	**left**	**next**
	1	**1**	**right**	**I17**
	b	**0**	**halt**	
	#	**#**	**right**	**same**

This instruction (I93) directs a machine to perform the actions described in the fragment of *NICE* language code provided below.

```
case (symbol read) of:
      0: begin
           print a 1;
           move one tape square left;
           goto the next instruction (I94)
         end;
      1: begin
           print a 1;
           move right one square;
           goto instruction I17
         end;
  blank: begin print a 0; halt end;
      #: begin
           print #;
           move to the right;
           goto this instruction (I93)
         end
endcase
```

Now that we know about instructions, we need some conventions concerning machine operation. *Input strings* are written on the tape prior to computation and will always consist of the symbols 0, 1, and blank. Thus we may speak of inputs as binary numbers when we wish. This may seem arbitrary, and it is. But the reason for this is so that we can describe Turing machines more easily later on. Besides, we shall discuss other input symbol alphabets in a later section.

When several binary numbers are given to a machine they will be separated by blanks (denoted as *b).* A sharp sign (#) always marks the left end of the tape at the beginning of a computation. Usually a machine is never allowed to change this marker. This is so that it can always tell when it is at the left end of its tape and thus not fall off the tape unless it wishes to do so. Here is an input tape with the triple <5, 14, 22> written upon it.

In order to depict this tape as a string we write: #101*b*1110*b*10110 and obviously omit the blank fill on the right.

Like programs, Turing machines are designed by coding sequences of instructions. So, let us design and examine an entire machine. The sequence of instructions in figure 2 describes a Turing machine that receives a binary number as input, adds one to it and then halts. Our strategy will be to begin at the lowest order bit (on the right end of the tape) and travel left changing ones to zeros until we reach a zero. This is then changed into a one.

One small problem arises. If the endmarker (#) is reached before a zero, then we have an input of the form 111...11 (the number $2^n - 1$) and must change it to 1000...00 (or 2^n).

sweep right to end of input

	read	*write*	*move*	*goto*
I1	**0**	**0**	**right**	**same**
	1	**1**	**right**	**same**
	#	**#**	**right**	**same**
	b	***b***	**left**	**next**

change 1's to 0's on left sweep, then change 0 to 1

I2	**0**	**1**	**halt**	
	1	**0**	**left**	**same**
	#	**#**	**right**	**next**

input = 11...1, so sweep right printing 1000...0 (print leading 1, add 0 to end)

I3	**0**	**1**	**right**	**next**

I4	**0**	**0**	**right**	**same**
	b	**0**	**halt**	

Figure 2 - Successor Machine

In order to understand this computational process better, let us examine, or in elementary programming terms, trace, a computation carried out by this Turing machine. First, we provide it with the input 1011 on its tape, place its head on the left endmarker (the #), and turn it loose.

Have a peek at figure 3. It is a sequence of snapshots of the machine in action. One should note that in the last snapshot (step 9) the machine is *not* about to execute an instruction. This is because it has halted.

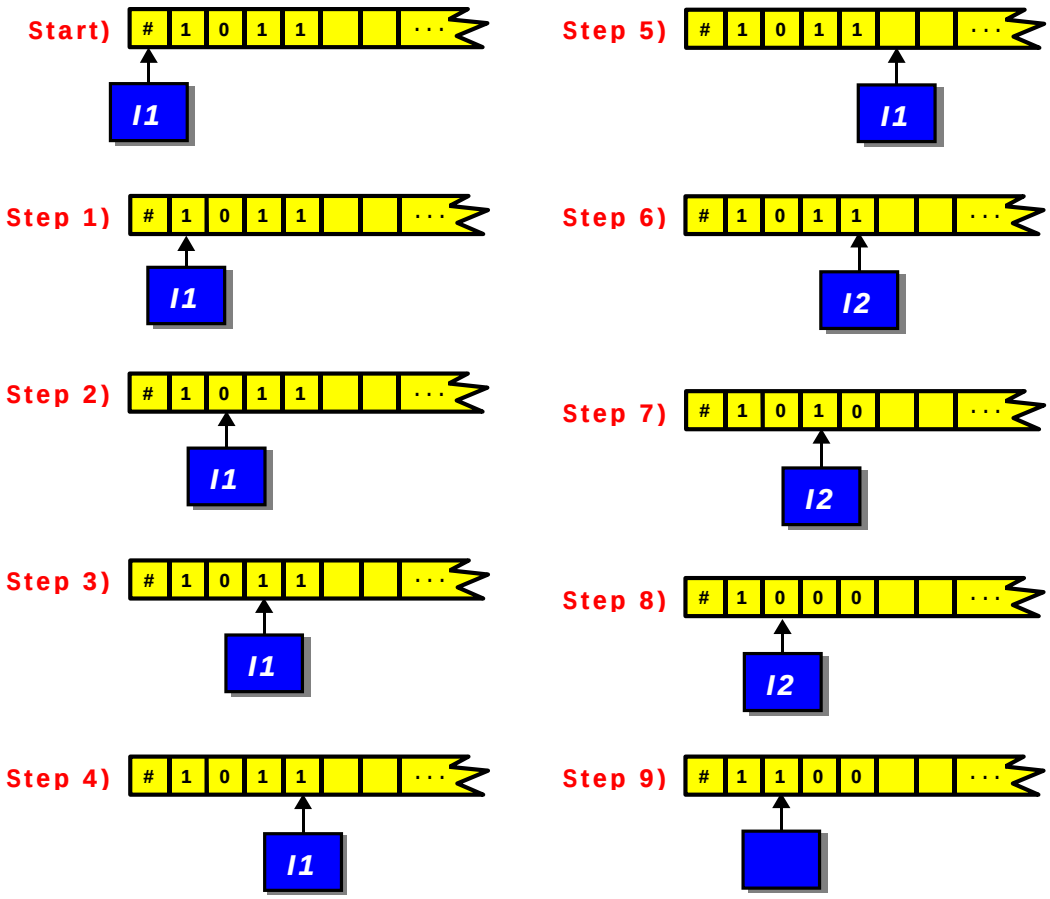


Figure 3 - Turing Machine Computation

Now that we have seen a Turing machine in action let us note some features, or properties of this class of computational devices.

a) There are no space or time constraints.
b) They may use numbers (or strings) of any size.
c) Their operation is quite simple - they read, write, and move.

In fact, Turing machines are *merely programs written in a very simple language.* Everything is finite and in general rather uncomplicated So, there is not too much to learn if we wish to use them as a computing device. Well, maybe we should wait a bit before believing that!

For a moment we shall return to the previous machine and discuss its efficiency. If it receives an input consisting only of ones (for example: 111111111), it must:

1) Go to the right end of the input,
2) Return to the left end marker, and
3) Go back to the right end of the input.

This means that *it runs for a number of steps more than three times the length of its input.* While one might complain that this is fairly slow, the machine does do the job! One might ask if a more efficient machine could be designed to accomplish this task? Try that as an amusing exercise.

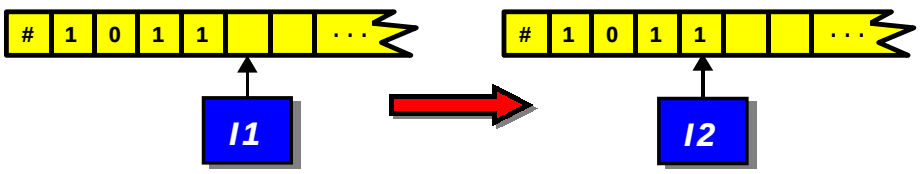
Another thing we should note is that when we present the machine with a blank tape it runs for a few steps and gets stuck on instruction I3 where no action is indicated for the configuration it is in since it is reading a blank instead of a zero. Thus it cannot figure out what to do. We say that this in an *undefined computation* and we shall examine situations such as this quite carefully later.

Up to this point, our discussion of Turing machines has been quite intuitive and informal. This is fine, but if we wish to study them as a model of computation and perhaps even prove a few theorems about them we must be a bit more precise and specify exactly what it is we are discussing. Let us begin.

A *Turing machine instruction* (we shall just call it an *instruction*) is a box containing *read-write-move-next* quadruples. A *labeled instruction* is an instruction with a label (such as I46) attached to it. Here is the entire machine.

> **Definition.** *A **Turing machine** is a finite sequence of labeled instructions with labels numbered sequentially from I1.*

Now we know precisely what Turing machines are. But we have yet to define what they do. Let's begin with pictures and then describe them in our definitions. Steps five and six of our previous computational example (figure 3) were the *machine configurations:*



If we translate this picture into a string, we can discuss what is happening in prose. We must do so if we wish to define precisely what Turing machines accomplish. So, place the instruction to be executed next to the symbol being read and we have an encoding of this change of configurations that looks like:

$$\#1011(I1)b... \rightarrow \#101(I2)1b...$$

This provides the same information as the picture. It is almost as if we took a snapshot of the machine during its computation. Omitting trailing blanks from the description we now have the following computational step

$$\#1011(I1) \rightarrow \#101(I2)1$$

Note that we shall always assume that there is an arbitrarily long sequence of blanks to the right of any Turing machine configuration.

> **Definition.** *A **Turing machine configuration** is a string of the form x(In)y or x where n is an integer and both x and y are (possibly empty) strings of symbols used by the machine.*

So far, so good. Now we need to describe how a machine goes from one configuration to another. This is done, as we all know by applying the instruction mentioned in a configuration to that configuration thus producing another configuration. An example should clear up any problems with the above verbosity. Consider the following instruction.

I17

0	**1**	**right**	**next**
1	***b***	**right**	**I3**
b	**1**	**left**	**same**
#	**#**	**halt**	

Now, observe how it transforms the following configurations.

a) #1101(I17)01 → #11011(I18)1
b) #110(I17)101 → #110*b*(I3)01
c) #110100(I17) → #11010(I17)01
d) (I17)#110100 → #110100

Especially note what took place in (c) and (d). Case (c) finds the machine at the beginning of the blank fill at the right end of the tape. So, it jots down a 1 and moves to the left. In (d) the machine reads the endmarker and halts. This is why the instruction disappeared from the configuration.

> **Definition.** *For Turing machine configurations C_i and C_j, C_i **yields** C_j (written $C_i \rightarrow C_j$) if and only if applying the instruction in C_i produces C_j .*

In order to be able to discuss a sequence of computational steps or an entire computation at once, we need additional notation.

Definition. *If C_i and C_j are Turing machine configurations then C_i* ***eventually yields*** *C_j (written $C_i \Rightarrow C_j$) if and only if there is a finite sequence of configurations $C_1, C_2, \ldots, C_k$ such that:*

$$C_i = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_k = C_j .$$

At the moment we should be fairly at ease with Turing machines and their operation. The concept of computation taking place when a machine goes through a sequence of configurations should also be comfortable.

Let us turn to something quite different. What about configurations which do not yield other configurations? They deserve our attention also. These are called *terminal configurations* because they terminate a computation). For example, given the instruction:

I3				
	0	**1**	**halt**	
	1	***b***	**right**	**next**
	#	**#**	**left**	**same**

what happens when the machine gets into the following configurations?

a) (I3)#01101
b) #1001(I3)*b*10
c) #100110(I3)
d) #101011

Nothing happens - right? If we examine the configurations and the instruction we find that the machine cannot continue for the following reasons (one for each configuration).

a) The machine moves left and falls off of the tape.
b) The machine does not know what to do.
c) Same thing. A trailing blank is being scanned.
d) Our machine has halted.

Thus none of those configurations lead to others. Furthermore, any computation or sequence of configurations containing configurations like them must terminate immediately.

By the way, configuration (d) is a favored configuration called a *halting configuration* because it was reached when the machine wished to halt. For example, if our machine was in the configuration #10(*I3*)0011 then the next configuration would be #101011 and no other configuration could follow. These halting configurations will pop up later and be of very great interest to us.

We name individual machines so that we know exactly which machine we are discussing at any time. We will often refer to them as M_1, M_2, M_3, or M_i and M_k. The notation $M_i(x)$ means that Turing machine M_i has been presented with x as its input. We shall use the name of a machine as the function it computes.

> *If the Turing machine M_i is presented with x as its input and eventually halts (after computing for a while) with z written on its tape, we think of M_i as a function whose value is z for the input x.*

Let us now examine a machine that expects the integers x and y separated by a blank as input. It should have an initial configuration resembling #x*b*y.

erase x, find first symbol of y

I1	**#**	**#**	**right**	**same**
	0	***b***	**right**	**same**
	1	***b***	**right**	**same**
	b	***b***	**right**	**next**

get next symbol of y - mark place

I2	**0**	*****	**left**	**next**
	1	*****	**left**	**I5**
	b	***b***	**halt**	

find right edge of output - write 0

I3	**b**	**b**	**left**	**same**
	#	**#**	**right**	**next**
	0	**0**	**right**	**next**
	1	**1**	**right**	**next**
I4	**b**	**0**	**right**	**I7**

find right edge of output - write 1

I5	**b**	**b**	**left**	**same**
	#	**#**	**right**	**next**
	0	**0**	**right**	**next**
	1	**1**	**right**	**next**
I6	**b**	**1**	**right**	**next**

find the * and resume copying

I7	**b**	**b**	**right**	**same**
	*****	**b**	**right**	**I2**

Figure 4 - Selection Machine

The Turing machine in figure 4 is what we call a *selection* machine. These receive several numbers (or strings) as input and select one of them as their output. This one computes the function: M(x, y) = y and selects the second of its inputs. This of course generalizes to any number of inputs, but let us not get too carried away.

Looking carefully at this machine, it should be obvious that it:

1) erases x, and
2) copies y next to the endmarker (#).

But, what might happen if either x or y happens to be blank? Figure it out! Also determine exactly how many steps this machine takes to erase x and copy y. (The answer is about n^2 steps if x and y are each n bits in length.)

Here is another Turing machine.

find the right end of the input

I1	**0**	**0**	**right**	**same**
	1	**1**	**right**	**same**
	#	**#**	**right**	**same**
	b	**b**	**left**	**next**

is low order bit is 0 or 1?

I2	**0**	**b**	**left**	**next**
	1	**b**	**left**	**I5**
	#	**#**	**right**	**I6**

erase input and print 1

I3	**0**	**b**	**left**	**same**
	1	**b**	**left**	**same**
	#	**#**	**right**	**next**
I4	**b**	**1**	**halt**	

erase input and print 0

I5	**0**	**b**	**left**	**same**
	1	**b**	**left**	**same**
	#	**#**	**right**	**next**
I6	**b**	**0**	**halt**	

Figure 5 - Even Integer Acceptor

It comes from a very important family of functions, one which contains functions that compute relations (or predicates) and membership in sets. These are known as *characteristic functions,* or *0-1 functions* because they only take values of zero and one which denote false and true.

An example is the characteristic function for the set of even integers computed by the Turing machine of figure 5. It may be described:

$$\text{even}(x) = \begin{cases} 1 \text{ if } x \text{ is even} \\ 0 \text{ otherwise} \end{cases}$$

This machine leaves a one upon its tape if the input ended with a zero (thus an even number) and halts with a zero on its tape otherwise (for a blank or odd integers). It should not be difficult to figure out how many steps it takes for an input of length n.

Now for a quick recap and a few more formal definitions. We know that Turing machines *compute functions.* Also we have agreed that if a machine receives x as an input and halts with z written on its tape, or in our notation:

$$(I1)\#x \Rightarrow \#z$$

then we say that M(x) = z. When machines never halt (that is: run forever or reach a non-halting terminal configuration) for some input x we claim that the value of M(x) is *undefined* just as we did with programs. Since output and halting are linked together, we shall precisely define halting.

> **Definition.** *A Turing machine **halts** if and only if it encounters a halt instruction during computation and **diverges** otherwise.*

So, we have machines that always provide output and some that do upon occasion. Those that always halt compute what we shall denote the ***total functions*** while the others merely compute ***partial functions***.

We now relate functions with sets by discussing how Turing machines may characterize the set by deciding which inputs are members of the set and which are not.

> **Definition.** *The Turing machine M **decides membership** in the set A if and only if for every x, if $x \in A$ then M(x) = 1, otherwise M(x) = 0.*

There just happens to be another method of computing membership in sets. Suppose you only wanted to know about members in some set and did not care

at all about elements that were not in the set. Then you could build a machine which halted when given a member of the set and *diverged* (ran forever or entered a non-halting terminal configuration) otherwise. This is called *accepting* the set.

> **Definition.** *The Turing machine M **accepts** the set A if and only if for all x, M(x) halts for x in A and diverges otherwise.*

This concept of acceptance may seem a trifle bizarre but it will turn out to be of surprising importance in later chapters.

Caveat lector: This is the zeroth (draft) edition of this lecture note. In particular, some topics still need to be written. Please send bug reports and suggestions to jeffe@illinois.edu.

Think globally, act locally.

— Attributed to Patrick Geddes (c.1915), among many others.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.

— Alan Turing, "Computing Machinery and Intelligence" (1950)

Never worry about theory as long as the machinery does what it's supposed to do.

— Robert Anson Heinlein, *Waldo & Magic, Inc.* (1950)

6 Turing Machines

In 1936, a few months before his 24th birthday, Alan Turing launched computer science as a modern intellectual discipline. In a single remarkable paper, Turing provided the following results:

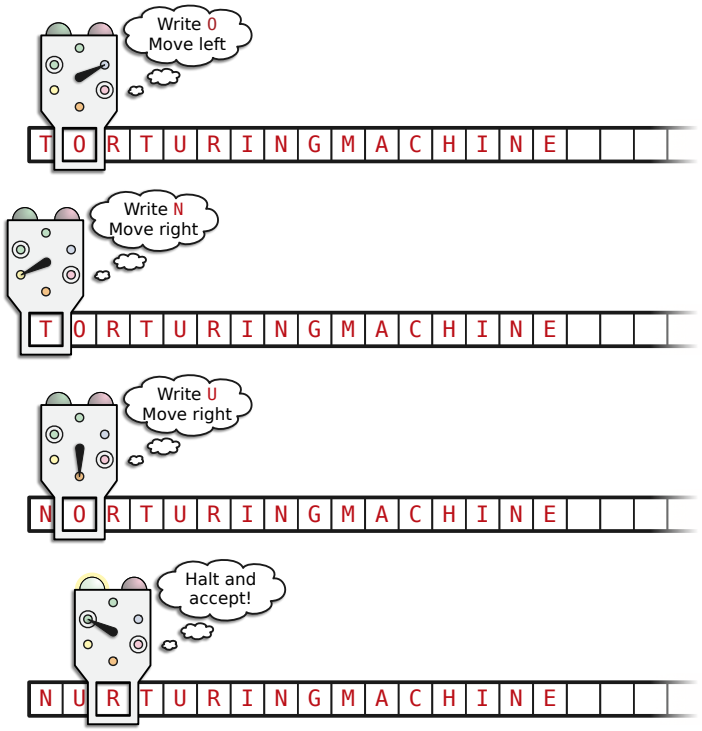
- A simple formal model of mechanical computation now known as *Turing machines*.
- A description of a single *universal* machine that can be used to compute *any* function computable by *any* other Turing machine.
- A proof that no Turing machine can solve the *halting problem*—Given the formal description of an arbitrary Turing machine M, does M halt or run forever?
- A proof that no Turing machine can determine whether an arbitrary given proposition is provable from the axioms of first-order logic. This Hilbert and Ackermann's famous *Entscheidungsproblem* ("decision problem")
- Compelling arguments[1] that his machines can execute *arbitrary* "calculation by finite means".

Turing's paper was not the first to prove that the *Entscheidungsproblem* had no algorithmic solution. Alonzo Church published the first proof just a new months earlier, using a very different model of computation, now called the ***untyped λ-calculus***. Turing and Church developed their results independently; indeed, Turing rushed the submission of his own paper immediately after receiving a copy of Church's paper, pausing only long enough to prove that any function computable via λ-calculus can also be computed by a Turing machine and vice versa. Church was the referee for Turing's paper; between the paper's submission and its acceptance, Turing was admitted to Princeton, where he became Church's PhD student. He finished his PhD two years later.

Informally, Turing described a device with a finite number of ***internal states*** that has access to memory in the form of a ***tape***. The tape consists of a semi-infinite sequence of ***cells***, each

[1] As Turing put it, "All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically." The claim that anything that can be computed can be computing using Turing machines is now known as the ***Church-Turing thesis***.

containing a single symbol from some arbitrary finite alphabet. The Turing machine can access the tape only through its ***head***, which is positioned over a single cell. Initially, the tape contains an arbitrary finite ***input string*** followed by an infinite sequence of ***blanks***, and the head is positioned over the first cell on the tape. In a single iteration, the machine reads the symbol in that cell, possibly write a new symbol into that cell, possibly changes its internal state, possibly moves the head to a neighboring cell, and possibly halts. The precise behavior of the machine at each iteration is entirely determined by its internal state and the symbol that it reads. When the machine halts, it indicates whether it has ***accepted*** or ***rejected*** the original input string.



A few iterations of a six-state Turing machine.

6.1 Why Bother?

Students used to thinking of computation in terms of higher-level operations like random memory accesses, function calls, and recursion may wonder why we should even consider a model as simple and constrained as Turing machines. Admittedly, Turing machines are a terrible model for thinking about ***fast*** computation; simple operations that take constant time in the standard random-access model can require *arbitrarily* many steps on a Turing machine. Worse, seemingly minor variations in the precise definition of "Turing machine" can have significant impact on problem complexity. As a simple example (which will make more sense later), we can reverse a string of n bits in $O(n)$ time using a two-tape Turing machine, but the same task provably requires $\Omega(n^2)$ time on a single-tape machine.

But here we are not interested in finding *fast* algorithms, or indeed in finding algorithms at all, but rather in proving that some problems cannot be solved by *any* computational means. Such a bold claim requires a formal definition of "computation" that is simple enough to support formal argument, but still powerful enough to describe arbitrary algorithms. Turing machines are ideal for this purpose. In particular, Turing machines are powerful enough to *simulate other*

Turing machines, while still simple enough to let us build up this self-simulation from scratch, unlike more complex but efficient models like the standard random-access machine

(Arguably, self-simulation is even simpler in Church's λ-calculus, or in Schönfinkel and Curry's combinator calculus, which is one of many reasons those models are more common in the design and analysis of programming languages than Turing machines. Those models much more abstract; in particular, they are harder to show equivalent to standard iterative models of computation.)

6.2 Formal Definitions

Formally, a Turing machine consists of the following components. (Hang on; it's a long list.)

- An arbitrary finite set Γ with at least two elements, called the ***tape alphabet***.
- An arbitrary symbol $\square \in \Gamma$, called the ***blank symbol*** or just the ***blank***.
- An arbitrary nonempty subset $\Sigma \subseteq (\Gamma \setminus \{\square\})$, called the ***input alphabet***.
- Another arbitrary finite set Q whose elements are called ***states***.
- Three distinct special states $\mathsf{start}, \mathsf{accept}, \mathsf{reject} \in Q$.
- A ***transition*** function $\delta : (Q \setminus \{\mathsf{accept}, \mathsf{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

A ***configuration*** or ***global state*** of a Turing machine is represented by a triple $(q, x, i) \in Q \times \Gamma^* \times \mathbb{N}$, indicating that the machine's internal state is q, the tape contains the string x followed by an infinite sequence of blanks, and the head is located at position i. Trailing blanks in the tape string are ignored; the triples (q, x, i) and $(q, x\square, i)$ describe exactly the same configuration.

The transition function δ describes the evolution of the machine. For example, $\delta(q, a) = (p, b, -1)$ means that when the machine reads symbol a in state q, it changes its internal state to p, writes symbol b onto the tape at its current location (replacing a), and then decreases its position by 1 (or more intuitively, moves one step to the left). If the position of the head becomes negative, no further transitions are possible, and the machine ***crashes***.

We write $\boldsymbol{(p, x, i) \Rightarrow_M (q, y, j)}$ to indicate that Turing machine M transitions from the first configuration to the second in one step. (The symbol $\Rightarrow$ is often pronounced "yields"; I will omit the subscript M if the machine is clear from context.) For example, $\delta(p, a) = (q, b, \pm 1)$ means that

$$(p, xay, i) \;\Rightarrow\; (q, xby, i \pm 1)$$

for any non-negative integer i, any string x of length i, and any string y. The evolution of any Turing machine is ***deterministic***; each configuration C yields a *unique* configuration C'. We write $\boldsymbol{C \Rightarrow^* C'}$ to indicate that there is a (possibly empty) sequence of transitions from configuration C to configuration C'. (The symbol $\Rightarrow^*$ can be pronounced "eventually yields".)

The initial configuration is $(w, \mathsf{start}, 0)$ for some arbitrary (and possibly empty) ***input string*** $w \in \Sigma^*$. If M eventually reaches the accept state—more formally, if $(w, \mathsf{start}, 0) \Rightarrow^* (x, \mathsf{accept}, i)$ for some string $x \in \Gamma^*$ and some integer i—we say that M ***accepts*** the original input string w. Similarly, if M eventually reaches the reject state, we say that M ***rejects*** w. We must emphasize that "rejects" and "does not accept" are *not* synonyms; if M crashes or runs forever, then M neither accepts nor rejects w.

We distinguish between two different senses in which a Turing machine can "accept" a language. Let M be a Turing machine with input alphabet Σ, and let $L \subseteq \Sigma^*$ be an arbitrary language over Σ.

- M ***recognizes*** or ***accepts*** L if and only if M accepts every string in L but nothing else. A language is ***recognizable*** (or *semi-computable* or *recursively enumerable*) if it is recognized by some Turing machine.
- M ***decides*** L if and only if M accepts every string in L and rejects every string in $\Sigma^* \setminus L$. Equivalently, M decides L if and only if M recognizes L and halts (without crashing) on all inputs. A language is ***decidable*** (or *computable* or *recursive*) if it is decided by some Turing machine.

Trivially, every decidable language is recognizable, but (as we will see later), not every recognizable language is decidable.

6.3 A First Example

Consider the language $L = \{0^n1^n0^n \mid n \geq 0\}$. This language is neither regular nor context-free, but it can be decided by the following six-state Turing machine. The alphabets and states of the machine are defined as follows:

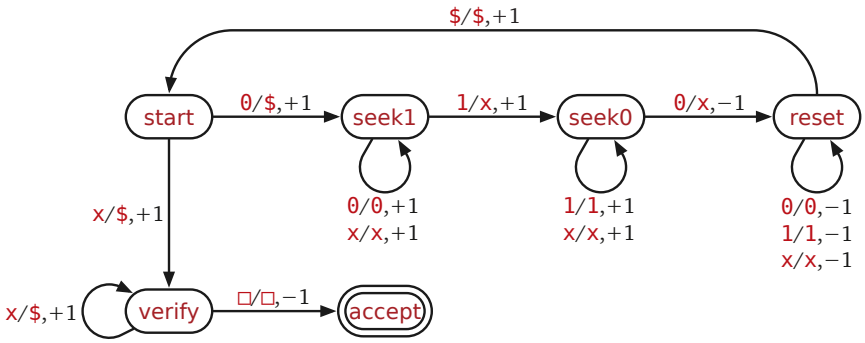
$$\Gamma = \{0, 1, \$, x, \square\}$$
$$\Sigma = \{0, 1\}$$
$$Q = \{\text{start}, \text{seek1}, \text{seek0}, \text{reset}, \text{verify}, \text{accept}, \text{reject}\}$$

The transition function is described in the following table; all unspecified transitions lead to the reject state. We also give a graphical representation of the same machine, which resembles a drawing of a DFA, but with output symbols and actions specified on each edge. For example, we indicate the transition $\delta(p, 0) = (q, 1, +1)$ by writing $0/1, +1$ next to the arrow from state p to state q.

$\delta(\ p\ , a) = (\ q\ , b, \Delta)$	explanation
$\delta($ start , 0$) = ($ seek1 , $, +1)$	mark first 0 and scan right
$\delta($ start , x$) = ($ verify , $, +1)$	looks like we're done, but let's make sure
$\delta($seek1, 0$) = ($ seek1 , 0, +1$)$	scan rightward for 1
$\delta($seek1, x$) = ($ seek1 , x, +1$)$	
$\delta($seek1, 1$) = ($ seek0 , x, +1$)$	mark 1 and continue right
$\delta($seek0, 1$) = ($ seek0 , 1, +1$)$	scan rightward for 0
$\delta($seek0, x$) = ($ seek0 , x, +1$)$	
$\delta($seek0, 0$) = ($ reset , x, +1$)$	mark 0 and scan left
$\delta($ reset , 0$) = ($ reset , 0, −1$)$	scan leftward for $
$\delta($ reset , 1$) = ($ reset , 1, −1$)$	
$\delta($ reset , x$) = ($ reset , x, −1$)$	
$\delta($ reset , $\$$$) = ($ start , $, +1$)$	step right and start over
$\delta($verify, x$) = ($ verify , $, +1$)$	scan right for any unmarked symbol
$\delta($verify, $\square$$) = ($accept, $\square$, −1$)$	success!

The transition function for a Turing machine that decides the language $\{0^n1^n0^n \mid n \geq 0\}$.

Finally, we trace the execution of this machine on two input strings: $001100 \in L$ and $00100 \notin L$. In each configuration, we indicate the position of the head using a small triangle



A graphical representation of the example Turing machine

instead of listing the position explicitly. Notice that we automatically add blanks to the tape string as necessary. Proving that this machine actually decides L—and in particular, that it never crashes or infinite-loops—is a straightforward but tedious exercise in induction.

$$
\begin{aligned}
(\text{start}, \underset{\blacktriangle}{0}01100) &\Rightarrow (\text{seek1}, \$\underset{\blacktriangle}{0}1100) \Rightarrow (\text{seek1}, \$0\underset{\blacktriangle}{1}100) \Rightarrow (\text{seek0}, \$0x\underset{\blacktriangle}{1}00) \Rightarrow (\text{seek0}, \$0x\underset{\blacktriangle}{1}00) \\
&\Rightarrow (\text{reset}, \$0x\underset{\blacktriangle}{1}x0) \Rightarrow (\text{reset}, \$0\underset{\blacktriangle}{x}1x0) \Rightarrow (\text{reset}, \$\underset{\blacktriangle}{0}x1x0) \Rightarrow (\text{reset}, \underset{\blacktriangle}{\$}0x1x0) \\
&\Rightarrow (\text{start}, \$\underset{\blacktriangle}{0}x1x0) \\
&\Rightarrow (\text{seek1}, \$\$\underset{\blacktriangle}{x}1x0) \Rightarrow (\text{seek1}, \$\$x\underset{\blacktriangle}{1}x0) \Rightarrow (\text{seek0}, \$\$xx\underset{\blacktriangle}{x}0) \Rightarrow (\text{seek0}, \$\$xxx\underset{\blacktriangle}{0}) \\
&\Rightarrow (\text{reset}, \$\$xx\underset{\blacktriangle}{x}x) \Rightarrow (\text{reset}, \$\$x\underset{\blacktriangle}{x}xx) \Rightarrow (\text{reset}, \$\$\underset{\blacktriangle}{x}xxx) \Rightarrow (\text{reset}, \underset{\blacktriangle}{\$}\$xxxx) \\
&\Rightarrow (\text{verify}, \$\underset{\blacktriangle}{\$}xxxx) \Rightarrow (\text{verify}, \$\$\underset{\blacktriangle}{\$}xxx) \Rightarrow (\text{verify}, \$\$\$\underset{\blacktriangle}{x}xx) \\
&\Rightarrow (\text{verify}, \$\$\$\$\underset{\blacktriangle}{x}x) \Rightarrow (\text{verify}, \$\$\$\$\$\underset{\blacktriangle}{□}) \Rightarrow (\text{accept}, \$\$\$\$\$\underset{\blacktriangle}{\$}) \Rightarrow \textbf{accept!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string $001100 \in L$

$$
\begin{aligned}
(\text{start}, \underset{\blacktriangle}{0}0100) &\Rightarrow (\text{seek1}, \$\underset{\blacktriangle}{0}100) \Rightarrow (\text{seek1}, \$0\underset{\blacktriangle}{1}00) \Rightarrow (\text{seek0}, \$0x\underset{\blacktriangle}{0}0) \\
&\Rightarrow (\text{reset}, \$0\underset{\blacktriangle}{x}x0) \Rightarrow (\text{reset}, \$\underset{\blacktriangle}{0}xx0) \Rightarrow (\text{reset}, \underset{\blacktriangle}{\$}0xx0) \\
&\Rightarrow (\text{start}, \$\underset{\blacktriangle}{0}xx0) \\
&\Rightarrow (\text{seek1}, \$\$\underset{\blacktriangle}{x}x0) \Rightarrow (\text{seek1}, \$\$x\underset{\blacktriangle}{x}0) \Rightarrow (\text{seek1}, \$\$xx\underset{\blacktriangle}{0}) \Rightarrow \textbf{reject!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string $00100 \notin L$

6.4 Variations

There are actually several formal models that all fall under the name "Turing machine", each with small variations on the definition we've given. Although we do need to be explicit about *which* variant we want to use for any particular problem, the differences between the variants are relatively unimportant. For any machine defined in one model, there is an equivalent machine in each of the other models; in particular, all of these variants recognize the same languages and decide the same languages. For example:

- **Halting conditions.** Some models allow multiple accept and reject states, which (depending on the precise model) trigger acceptance or rejection either when the machine enters the state, or when the machine has no valid transitions out of such a state. Others include only explicit accept states, and either equate crashing with rejection or do not define a rejection mechanism at all. Still other models include halting as one of the possible *actions* of the machine, in addition to moving left or moving right; in these models, the machine accepts/rejects its input if and only if it halts in an accepting/non-accepting state.

- **Actions.** Some Turing machine models allow transitions that do not move the head, or that move the head by more than one cell in a single step. Others insist that a single step of the machine *either* writes a new symbol onto the tape *or* moves the head one step. Finally, as mentioned above, some models include halting as one of the available actions.

- **Transition function.** Some models of Turing machines, including Turing's original definition, allow the transition function to be undefined on some state-symbol pairs. In this formulation, the transition function is given by a set $\delta \subset Q \times \Gamma \times Q \times \Gamma \times \{+1, -1\}$, such that for each state q and symbol a, there is at most one transition $(q, a, \cdot, \cdot, \cdot) \in \delta$. If the machine enters a configuration from which there is no transition, it halts and (depending on the precise model) either crashes or rejects. Others define the transition function as $\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{-1, +1\})$, allowing the machine to *either* write a symbol to the tape *or* move the head in each step.

- **Beginning of the tape.** Some models forbid the head to move past the beginning of the tape, either by starting the tape with a special symbol that cannot be overwritten and that forces a rightward transition, or by declaring that a leftward transition at position 0 leaves the head in position 0, or even by pure fiat—declaring any machine that performs a leftward move at position 0 to be invalid.

To prove that any two of these variant "species" of Turing machine are equivalent, we must show how to transform a machine of one species into a machine of the other species that accepts and rejects the same strings. For example, let $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ be a Turing machine with explicit accept and reject states. We can define an equivalent Turing machine M' that halts only when it moves left from position 0, and accepts only by halting while in an accepting state, as follows. We define the set of accepting states for M' as $A = \{\mathsf{accept}\}$ and define a new transition function

$$\delta'(q, a) := \begin{cases} (\mathsf{accept}, a, -1) & \text{if } q = \mathsf{accept} \\ (\mathsf{reject}, a, -1) & \text{if } q = \mathsf{reject} \\ \delta(q, a) & \text{otherwise} \end{cases}$$

Similarly, suppose someone gives us a Turing machine $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ whose transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ allows the machine to transition without moving its head. We can construct an equivalent Turing machine $M' = (\Gamma, \square, \Sigma, Q', s, \mathsf{accept}, \mathsf{reject}, \delta')$ that moves its head at every transition by defining $Q' := Q \times \{0, 1\}$ and

$$\delta'((p, 0), a) := \begin{cases} ((q, 1), b, +1) & \text{if } \delta(p, a) = (q, b, 0), \\ ((q, 0), b, \Delta) & \text{if } \delta(p, a) = (q, b, \Delta) \text{ and } \Delta \neq 0, \end{cases}$$

$$\delta'((p, 1), a) := ((p, 0), a, -1).$$

6.5 Computing Functions

Turing machines can also be used to compute functions from strings to strings, instead of just accepting or rejecting strings. Since we don't care about acceptance or rejection, we replace the explicit accept and reject states with a single halt state, and we define the ***output*** of the Turing machine to be the contents of the tape when the machine halts, after removing the infinite sequence of trailing blanks. More formally, for any Turing machine M, any string $w \in \Sigma^*$, and any string $x \in \Gamma^*$ that does not end with a blank, we write $\boldsymbol{M(w) = x}$ if and only if $(w, s, 0) \Rightarrow_M^* (x, \text{halt}, i)$ for some integer i. If M does not halt on input w, then we write $\boldsymbol{M(w)\nearrow}$, which can be read either "M diverges on w" or "$M(w)$ is undefined." We say that M ***computes*** the function $f : \Sigma^* \to \Sigma^*$ if and only if $M(w) = f(w)$ for every string w.

6.5.1 Shifting

One basic operation that is used in many Turing machine constructions is ***shifting*** the input string a constant number of steps to the right or to the left. For example, given any input string $w \in \{0,1\}^*$, we can compute the string $0w$ using a Turing machine with tape alphabet $\Gamma = \{0, 1, \square\}$, state set $Q = \{0, 1, \text{halt}\}$, start state 0, and the following transition function:

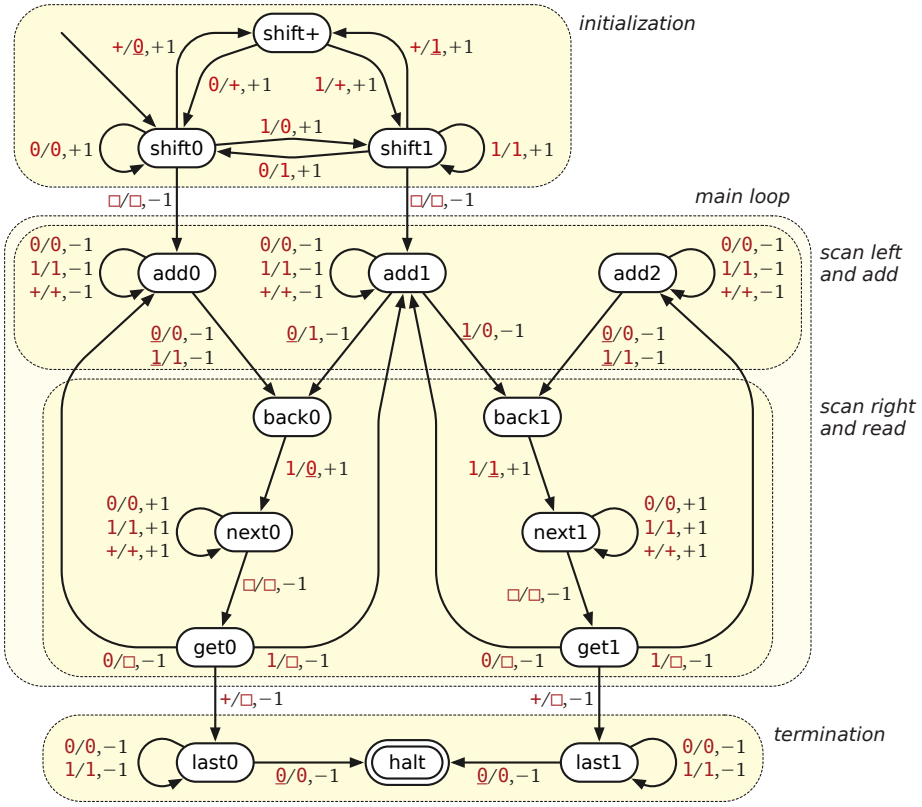
$$\begin{array}{l}
\delta(p,\ a) = (\ q\ ,\ b,\ \Delta\ ) \\
\hline
\delta(0,\ 0) = (\ 0\ ,\ 0,\ +1) \\
\delta(0,\ 1) = (\ 1\ ,\ 0,\ +1) \\
\delta(0,\ \square) = (\text{halt},\ 0,\ +1) \\
\hdashline
\delta(1,\ 0) = (\ 0\ ,\ 1,\ +1) \\
\delta(1,\ 1) = (\ 1\ ,\ 1,\ +1) \\
\delta(1,\ \square) = (\text{halt},\ 1,\ +1)
\end{array}$$

By increasing the number of states, we can build a Turing machine that shifts the input string any fixed number of steps in either direction. For example, a machine that shifts its input to the left by five steps might read the string from right to left, storing the five most recently read symbols in its internal state. A typical transition for such a machine would be $\delta(12345, 0) = (01234, 5, -1)$.

6.5.2 Binary Addition

With a more complex Turing machine, we can implement binary addition. The input is a string of the form $w{+}x$, where $w, x \in \{0,1\}^n$, representing two numbers in binary; the output is the binary representation of $w + x$. To simplify our presentation, we assume that $|w| = |x| > 0$; however, this restrictions can be removed with the addition of a few more states. The following figure shows the entire Turing machine at a glance. The machine uses the tape alphabet $\Gamma = \{\square, 0, 1, +, \underline{0}, \underline{1}\}$; the start state is shift0. All missing transitions go to a fail state, indicating that the input was badly formed.

Execution of this Turing machine proceeds in several phases, each with its own subset of states, as indicated in the figure. The initialization phase scans the entire input, shifting it to the right to make room for the output string, marking the rightmost bit of w, and reading and erasing the last bit of x.



A Turing machine that adds two binary numbers of the same length.

$$\frac{\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)}{\begin{array}{l}\delta(\text{shift0},\ \texttt{0}) = (\text{shift0},\ \texttt{0},\ +1)\\ \delta(\text{shift0},\ \texttt{1}) = (\text{shift1},\ \texttt{0},\ +1)\\ \delta(\text{shift0},\ \texttt{+}) = (\text{shift+},\ \underline{\texttt{0}},\ +1)\\ \delta(\text{shift0},\ \square) = (\text{add0},\ \square,\ -1)\\ \hdashline \delta(\text{shift1},\ \texttt{0}) = (\text{shift0},\ \texttt{1},\ +1)\\ \delta(\text{shift1},\ \texttt{1}) = (\text{shift1},\ \texttt{1},\ +1)\\ \delta(\text{shift1},\ \texttt{+}) = (\text{shift+},\ \underline{\texttt{1}},\ +1)\\ \delta(\text{shift1},\ \square) = (\text{add1},\ \square,\ -1)\\ \hdashline \delta(\text{shift+},\ \texttt{0}) = (\text{shift0},\ \texttt{+},\ +1)\\ \delta(\text{shift+},\ \texttt{1}) = (\text{shift1},\ \texttt{+},\ +1)\end{array}}$$

The first part of the main loop scans left to the marked bit of w, adds the bit of x that was just erased plus the carry bit from the previous iteration, and records the carry bit for the next iteration in the machines internal state.

$$\frac{\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)}{\begin{array}{l}\delta(\text{add0},\ \texttt{0}) = (\text{add0},\ \texttt{0},\ -1)\\ \delta(\text{add0},\ \texttt{1}) = (\text{add0},\ \texttt{0},\ -1)\\ \delta(\text{add0},\ \texttt{+}) = (\text{add0},\ \texttt{0},\ -1)\\ \delta(\text{add0},\ \underline{\texttt{0}}) = (\text{back0},\ \texttt{0},\ -1)\\ \delta(\text{add0},\ \underline{\texttt{1}}) = (\text{back0},\ \texttt{1},\ -1)\end{array}} \qquad \frac{\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)}{\begin{array}{l}\delta(\text{add1},\ \texttt{0}) = (\text{add1},\ \texttt{0},\ -1)\\ \delta(\text{add1},\ \texttt{1}) = (\text{add1},\ \texttt{0},\ -1)\\ \delta(\text{add1},\ \texttt{+}) = (\text{add1},\ \texttt{0},\ -1)\\ \delta(\text{add1},\ \underline{\texttt{0}}) = (\text{back0},\ \texttt{1},\ -1)\\ \delta(\text{add1},\ \underline{\texttt{1}}) = (\text{back1},\ \texttt{0},\ -1)\end{array}} \qquad \frac{\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta)}{\begin{array}{l}\delta(\text{add2},\ \texttt{0}) = (\text{add2},\ \texttt{0},\ -1)\\ \delta(\text{add2},\ \texttt{1}) = (\text{add2},\ \texttt{0},\ -1)\\ \delta(\text{add2},\ \texttt{+}) = (\text{add2},\ \texttt{0},\ -1)\\ \delta(\text{add2},\ \underline{\texttt{0}}) = (\text{back1},\ \texttt{0},\ -1)\\ \delta(\text{add2},\ \underline{\texttt{1}}) = (\text{back1},\ \texttt{1},\ -1)\end{array}}$$

The second part of the main loop marks the previous bit of w, scans right to the end of x, and then reads and erases the last bit of x, all while maintaining the carry bit.

$\delta($ p , $a) = ($ q , b, $\Delta)$	$\delta($ p , $a) = ($ q , b, $\Delta)$
$\delta($back0, 0$) = ($next0, $\underline{0}$, $+1$)$	$\delta($back1, 0$) = ($next1, $\underline{0}$, $+1$)$
$\delta($back0, 1$) = ($next0, $\underline{1}$, $+1$)$	$\delta($back1, 1$) = ($next1, $\underline{1}$, $+1$)$
$\delta($next0, 0$) = ($next0, 0, $+1$)$	$\delta($next1, 0$) = ($next1, 0, $+1$)$
$\delta($next0, 1$) = ($next0, 0, $+1$)$	$\delta($next1, 1$) = ($next1, 0, $+1$)$
$\delta($next0, +$) = ($next0, 0, $+1$)$	$\delta($next1, +$) = ($next1, 0, $+1$)$
$\delta($next0, □$) = ($ get0 , □, $-1)$	$\delta($next1, □$) = ($ get1 , □, $-1)$
$\delta($ get0 , 0$) = ($add0, □, $-1)$	$\delta($ get1 , 0$) = ($add1, □, $-1)$
$\delta($ get0 , 1$) = ($add1, □, $-1)$	$\delta($ get1 , 1$) = ($add2, □, $-1)$
$\delta($ get0 , +$) = ($last0, □, $-1)$	$\delta($ get1 , +$) = ($last1, □, $-1)$

Finally, after erasing the + in the last iteration of the main loop, the termination phase adds the last carry bit to the leftmost output bit and halts.

$\delta($ p , $a) = ($ q , b, $\Delta)$
$\delta($last0, 0$) = ($last0, 0, $-1)$
$\delta($last0, 1$) = ($last0, 0, $-1)$
$\delta($last0, $\underline{0}$$) = ($ halt , 0, $\quad)$
$\delta($last1, 0$) = ($last1, 0, $-1)$
$\delta($last1, 1$) = ($last1, 0, $-1)$
$\delta($last1, $\underline{0}$$) = ($ halt , 1, $\quad)$

6.6 Variations on Tracks, Heads, and Tapes

Multiple Tracks

It is sometimes convenient to endow the Turing machine tape with multiple ***tracks***, each with its own tape alphabet, and allow the machine to read from and write to the same position on all tracks simultaneously. For example, to define a Turing machine with three tracks, we need three tape alphabets Γ_1, Γ_2, and Γ_3, each with its own blank symbol, where (say) Γ_1 contains the input alphabet Σ as a subset; we also need a transition function of the form

$$\delta : Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \to Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \times \{-1, +1\}$$

Describing a configuration of this machine requires a quintuple (q, x_1, x_2, x_3, i), indicating that each track i contains the string x_i followed by an infinite sequence of blanks. The initial configuration is $(\text{start}, w, \varepsilon, \varepsilon, 0)$, with the input string written on the first track, and the other two tracks completely blank.

But any such machine is equivalent (if not *identical*) to a single-track Turing machine with the (still finite!) tape alphabet $\Gamma := \Gamma_1 \times \Gamma_2 \times \Gamma_3$. Instead of thinking of the tape as three infinite sequences of symbols, we think of it as a single infinite sequence of "records", each containing three symbols. Moreover, there's nothing special about the number 3 in this construction; a Turing machine with *any* constant number of tracks is equivalent to a single-track machine.

Doubly-Infinite Tape

It is also sometimes convenient to allow the tape to be infinite in both directions, for example, to avoid boundary conditions. There are several ways to simulate a doubly-infinite tape on a machine with only a semi-infinite tape. Perhaps the simplest method is to use a semi-infinite tape with two tracks, one containing the cells with positive index and the other containing the cells

with negative index in reverse order, with a special marker symbol at position zero to indicate the transition.

0	+1	+2	+3	+4	⋯
►	−1	−2	−3	−4	⋯

Another method is to shuffle the positive-index and negative-index cells onto a single track, and add additional states to allow the Turing machine to move two steps in a single transition. Again, we need a special symbol at the left end of the tape to indicate the transition:

►	0	−1	+1	−2	+2	−3	+3	⋯

A third method maintains two sentinel symbols ► and ◄ that surround all other non-blank symbols on the tape. Whenever the machine reads the right sentinel ◄, we write a blank, move right, write ◄, move left, and then proceed as if we had just read a blank. On the other hand, when the machine reads the left sentinel ►, we shift the entire contents of the tape (up to and including the right sentinel) one step to the right, then move back to the left sentinel, move right, write a blank, and finally proceed as if we had just read a blank. Since the Turing machine does not actually have access to the position of the head *as an integer*, shifting the head and the tape contents one step right has no effect on its future evolution.

►	−3	−2	−1	0	+1	+2	+3	+4	+5	◄	⋯

Using either of the first two methods, we can simulate t steps of an arbitrary Turing machine with a doubly-infinite tape using only $O(t)$ steps on a standard Turing machine. The third method, unfortunately, requires $\Theta(t^2)$ steps in the worst case.

Insertion and Deletion

We can also allow Turing machines to insert and delete cells on the tape, in addition to simply overwriting existing symbols. We've already seen how to insert a new cell: Leave a special mark on the tape (perhaps in a second track), shift everything to the right of this mark one cell to the right, scan left to the mark, erase the mark, and finally write the correct character into the new cell. Deletion is similar: Mark the cell to be deleted, shift everything to the right of the mark one step to the left, scan left to the mark, and erase the mark. We may also need to maintain a mark in some cell to the right every non-blank symbol, indicating that all cells further to the right are blank, so that we know when to stop shifting left or right.

Multiple Heads

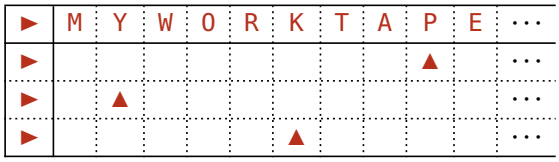
Another convenient extension is to allow machines simultaneous access to more than one position on the tape. For example, to define a Turing machine with *three* heads, we need a transition function of the form

$$\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{-1, +1\}^3.$$

Describing a configuration of such a machine requires a quintuple (q, x, i, j, k), indicating that the machine is in state q, the tape contains string x, and the three heads are at positions i, j, k. The transition function tells us, given q and the three symbols $x[i], x[j], x[k]$, which three symbols to write on the tape and which direction to move each of the heads.

We can simulate this behavior with a single head by adding additional tracks to the tape that record the positions of each head. To simulate a machine M with three heads, we use a

tape with four tracks: track 0 is the actual work tape; each of the remaining tracks has a single non-blank symbol recording the position of one of the heads. We also insert a special marker symbols at the left end of the tape.



We can simulate any single transition of M, starting with our single head at the left end of the tape, as follows. Throughout the simulation, we maintain the internal state of M as one of the components of our current state. First, for each i, we read the symbol under the ith head of M as follows:

> Scan to the right to find the mark on track i, read the corresponding symbol from track 0 into our internal state, and then return to the left end of the tape.

At this point, our internal state records M's current internal state and the three symbols under M's heads. After one more transition (using M's transition function), our internal state records M's *next* state, the symbol to be written by each head, and the direction to move each head. Then, for each i, we write with and move the ith head of M as follows:

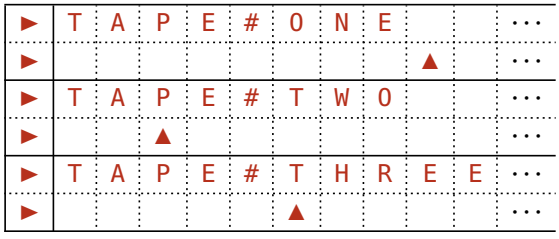
> Scan to the right to find the mark on track i, write the correct symbol onto on track 0, move the mark on track i one step left or right, and then return to the left end of the tape.

Again, there is nothing special about the number 3 here; we can simulate machines with *any* fixed number of heads.

Careful analysis of this technique implies that for any integer k, we can simulate t steps of an arbitrary Turing machine with k independent heads in $\Theta(t^2)$ time on a standard Turing machine with only one head. Unfortunately, this quadratic blowup is unavoidable. It is relatively easy to recognize the language of *marked palindromes* $\{w \bullet w^R \mid w \in \{0,1\}^*\}$ in $O(n)$ time using a Turing machine with two heads, but recognizing this language provably requires $\Omega(n^2)$ time on a standard machine with only one head. On the other hand, with much more sophisticated techniques, it is possible to simulate t steps of a Turing machine with k head, for any fixed integer k, using only $O(t \log t)$ steps on a Turing machine with just *two* heads.

Multiple Tapes

We can also allow machines with multiple independent tapes, each with its own head. To simulate such a machine with a single tape, we simply maintain each tape as an independent track with its own head. Equivalently, we can simulate a machine with k tapes using a single tape with $2k$ tracks, half storing the contents of the k tapes and half storing the positions of the k heads.



Just as for multiple tracks, for any constant k, we can simulate t steps of an arbitrary Turing machine with k independent tapes in $\Theta(t^2)$ steps on a standard Turing machine with one tape, and this quadratic blowup is unavoidable. Moreover, it is possible to simulate t steps on a k-tape Turing machine using only $O(t \log t)$ steps on a *two*-tape Turing machine using more sophisticated techniques. (This faster simulation is easier to obtain for multiple independent tapes than for multiple heads on the same tape.)

By combining these tricks, we can simulate a Turing machine with any fixed number of tapes, each of which may be infinite in one or both directions, each with any fixed number of heads and any fixed number of tracks, with at most a quadratic blowup in the running time.

6.7 Simulating a Real Computer

6.7.1 Subroutines and Recursion

★★★

Use a second tape/track as a "call stack". Add save and restore actions. In the simplest formulation, subroutines do not have local memory. To call a subroutine, save the current state onto the call stack and jump to the first state of the subroutine. To return, restore (and remove) the return state from the call stack. We can simulate t steps of any recursive Turing machine with $O(t)$ steps on a multitape standard Turing machine, or in $O(t^2)$ steps on a standard Turing machine.

More complex versions of this simulation can adapt to

6.7.2 Random-Access Memory

★★★

Keep [address•data] pairs on a separate "memory" tape. Write address to an "address" tape; read data from or write data to a "data" tape. Add new or changed [address•data] pairs at the end of the memory tape. (Semantics of reading from an address that has never been written to?)

Suppose all memory accesses require at most ℓ address and data bits. Then we can simulate the kth memory access in $O(k\ell)$ steps on a multitape Turing machine or in $O(k^2\ell^2)$ steps on a single-tape machine. Thus, simulating t memory accesses in a random-access machine with ℓ-bit words requires $O(t^2\ell)$ time on a multitape Turing machine, or $O(t^3\ell^2)$ time on a single-tape machine.

6.8 Universal Turing Machines

With all these tools in hand, we can now describe the pinnacle of Turing machine constructions: the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w \rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w \rangle$ if and only if M accepts w.
- U rejects $\langle M, w \rangle$ if and only if M rejects w.

In the next few pages, I will sketch a universal Turing machine U that uses the input alphabet {0, 1, [,], •, |} and a somewhat larger tape alphabet (via marks on additional tracks). However, I do *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|) \rceil$ bits. Thus, if $\Gamma = \{0, 1, \$, x, \square\}$, we might use the following encoding:

$$\langle 0 \rangle = 001, \qquad \langle 1 \rangle = 010, \qquad \langle \$ \rangle = 011, \qquad \langle x \rangle = 100, \qquad \langle \square \rangle = 000.$$

The input string w is encoded by its sequence of symbol encodings, with separators • between every pair of symbols and with brackets [and] around the whole string. For example, with this encoding, the input string *001100* would be encoded on the input tape as

$$\langle 001100 \rangle = [001\bullet001\bullet010\bullet010\bullet001\bullet001]$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg|Q| \rceil$ bits. Without loss of generality, we encode the start state with all 1s and the reject state with all 0s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start} \rangle = 111 \qquad \langle \mathit{seek1} \rangle = 010 \qquad \langle \mathit{seek0} \rangle = 011 \qquad \langle \mathit{reset} \rangle = 100$$
$$\langle \mathit{verify} \rangle = 101 \qquad \langle \mathit{accept} \rangle = 110 \qquad \langle \mathit{reject} \rangle = 000$$

We encode the machine M itself as the string $\langle M \rangle = [\langle \mathit{reject} \rangle \bullet \langle \square \rangle] \langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $[\langle p \rangle \bullet \langle a \rangle | \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle]$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq \mathit{reject}$. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := 0$ and $\langle +1 \rangle := 1$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \$) = (\mathit{start}, \$, +1)$ would be encoded as

$$[100\bullet011|001\bullet011\bullet1].$$

Our first example Turing machine for recognizing $\{0^n 1^n 0^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
          [010•001|010•001•1][010•100|010•100•1]
          [010•010|011•100•1][011•010|011•010•1]
          [011•100|011•100•1][011•001|100•100•1]
          [100•001|100•001•0][100•010|100•010•0]
          [100•100|100•100•0][100•011|001•011•1]
          [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $[\langle q \rangle \bullet \langle a \rangle]$ indicating that (1) the cell contains symbol a; (2) if $q \neq \mathit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \mathit{reject}$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets [and].

For example, with the encodings described above, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{0}01100, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{0}01100, 0 \rangle = \underbrace{[[111\bullet001][000\bullet001][000\bullet010][000\bullet010][000\bullet001][000\bullet001]}_{\textbf{start } 0 \quad \mathit{reject}\ 0 \quad \mathit{reject}\ 1 \quad \mathit{reject}\ 1 \quad \mathit{reject}\ 0 \quad \mathit{reject}\ 0}]$$

Similarly, the intermediate configuration (*reset*, *$0x1x0*, 3) would be encoded as follows:

$$\langle \textit{reset}, \$\$x1x0, 3\rangle = \underbrace{\texttt{[[000•011]}}_{\textit{reject \$}}\underbrace{\texttt{[000•011]}}_{\textit{reject 0}}\underbrace{\texttt{[000•100]}}_{\textit{reject x}}\underbrace{\texttt{[010•010]}}_{\textbf{\textit{reset 1}}}\underbrace{\texttt{[000•100]}}_{\textit{reject x}}\underbrace{\texttt{[000•001]]}}_{\textit{reject 0}}$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings `[[` and `]]` each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle start, x, 0\rangle$ of the initial configuration of M with input x. First, we write `[[`$\langle start\rangle$`•`. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write `[[`$\langle start\rangle$`•`.
- Instead of copying each separator `•`, write `][`$\langle reject\rangle$`•`
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle start\rangle$ and $\langle reject\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle start\rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell `[`$\langle p\rangle$`•`$\langle a\rangle$`]` such that $p \neq reject$. Then we scan through the encoded transition function $\langle\delta\rangle$ to find the unique encoded tuple `[`$\langle p\rangle$`•`$\langle a\rangle$`|`$\langle q\rangle$`•`$\langle b\rangle$`•`$\langle\Delta\rangle$`]` whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle$`•`$\langle b\rangle$ of the tuple to the work tape. Now if $q = accept$, then U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq reject$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with `[`$\langle reject\rangle$`•`$\langle\square\rangle$`]]` (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks `[`$\langle reject\rangle$`•`$\langle\square\rangle$`]` instead of *actual* blanks □.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string *001100*. The execution of M on input w eventually reaches the configuration (*seek1*, *$$x1x0*, 3). At the start of the corresponding step in U's simulation, U is in the following configuration:

`[[000•011][000•011][000•100][010•010][000•100][000•001]]`

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix `[`$\langle reject\rangle$`•` of the machine encoding $\langle M\rangle$ on the input tape. U finds a match in the fourth encoded cell.

`[[000•011][000•011][000•100][010•010][000•100][000•001]]`

Next, U scans the machine encoding $\langle M \rangle$ for the substring `[010•010` matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition `[010•010|011•100•1]`. U copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][011•100][000•100][000•001]]`

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][000•100][011•100][000•001]]`

Finally, U scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration (*seek0*, *$$xxx0*, 4), as required.

`[[000•011][000•011][000•100][000•100][011•100][000•001]]`

Exercises

1. Describe Turing machines that decide each of the following languages:
   (a) Palindromes over the alphabet $\{0, 1\}$
   (b) $\{ww \mid w \in \{0, 1\}^*\}$
   (c) $\left\{0^a 1^b 0^{ab} \mid a, b \in \mathbb{N}\right\}$

2. Let $\langle n \rangle_2$ denote the binary representation of the non-negative integer n. For example, $\langle 17 \rangle_2 = 10001$ and $\langle 42 \rangle_2 = 101010$. Describe Turing machines that compute the following functions from $\{0, 1\}^*$ to $\{0, 1\}^*$:
   (a) $w \mapsto www$
   (b) $1^n 0 1^m \mapsto 1^{mn}$
   (c) $1^n \mapsto 1^{2^n}$
   (d) $1^n \mapsto \langle n \rangle_2$
   (e) $0^* \langle n \rangle_2 \mapsto 1^n$
   (f) $\langle n \rangle_2 \mapsto \langle n^2 \rangle_2$

3. Describe Turing machines that write each of the following infinite streams of bits onto their tape. Specifically, for each integer n, there must be a finite time after which the first n symbols on the tape always match the first n symbols in the target stream.
   (a) An infinite stream of 1s
   (b) 0101101110111101111101111110…, where the nth block of 1s has length n.
   (c) The stream of bits whose nth bit is 1 if and only if n is prime.

(d) The ***Thue-Morse sequence*** $T_0 \bullet T_1 \bullet T_2 \bullet T_3 \cdots$, where

$$T_n := \begin{cases} \texttt{0} & \text{if } n = 0 \\ \texttt{1} & \text{if } n = 1 \\ T_{n-1} \bullet \overline{T_{n-1}} & \text{otherwise} \end{cases}$$

where $\overline{w}$ indicates the binary string obtained from w by flipping every bit. Equivalently, the nth bit of the Thue Morse sequence if `0` if the binary representation of n has an even number of `1`s and `1` otherwise.

`0 1 10 1001 10010110 1001011001101001 10010110011010010110100 10...`

(e) The ***Fibonacci*** sequence $F_0 \bullet F_1 \bullet F_2 \bullet F_3 \cdots$, where

$$F_n := \begin{cases} \texttt{0} & \text{if } n = 0 \\ \texttt{1} & \text{if } n = 1 \\ F_{n-2} \bullet F_{n-1} & \text{otherwise} \end{cases}$$

`0 1 01 101 01101 10101101 0110110101101 1010110101101101 10...`

4. A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

   Prove formally that any standard Turing machine can be simulated by a two-stack machine. That is, given any standard Turing machine M, describe a two-stack machine M' that accepts and rejects exactly the same input strings as M.

★★★

Counter machines. Configuration consists of k rational numbers and an internal state (from some finite set Q). Transition function $\delta : Q \times \{=0, >0\}^k \to Q \times \{-1, 0, +1\}^k$ takes internal state and signs of counters as input, and produces new internal state and changes to counters as output.

- Prove that any Turing machine can be simulated by a three-counter machine. One counter holds the binary representation of the tape after the head; another counter holds the reversed binary representation of the tape before the head. Implement transitions via halving, doubling, and parity, using the third counter for scratch work.
- Prove that two counters can simulate three. Store $2^a 3^b 5^c$ in one counter, use the other for scratch work.
- Prove that a three-counter machine can compute any computable function: Given input $(n, 0, 0)$, we can compute $(f(n), 0, 0)$ for *any* computable function f. First transform $(n, 0, 0)$ to $(2^n, 0, 0)$ using all three counters; then run two- (or three-)counter TM simulation to obtain $(2^{f(n)}, 0, 0)$; and finally transform $(2^{f(n)}, 0, 0)$ to $(f(n), 0, 0)$ using all three counters.
- **HARD:** Prove that a two-counter machine cannot transform $(n, 0)$ to $(2^n, 0)$. [Barzdin' 1963, Yao 1971, Schröpel 1972, Ibarra+Trân 1993]

★★★

FRACTRAN [Conway 1987]: One-counter machine whose "program" is a sequence of rational numbers. The counter is initially 1. At each iteration, multiply the counter by the first rational number that yields an integer; if there is no such number, halt.

- Prove that for any computable function $f : \mathbb{N} \to \mathbb{N}$, there is a FRACTRAN program that transforms 2^{n+1} into $3^{f(n)+1}$, for all natural numbers n.
- Prove that every FRACTRAN program, given the integer 1 as input, either outputs 1 or loops forever. It follows that there is no FRACTRAN program for the increment function $n \mapsto n + 1$.

5. A ***tag-***Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

6. (a) Prove that any standard Turing machine can be simulated by a Turing machine with only three states. *[Hint: Use the tape to store an encoding of the state of the machine yours is simulating.]*

   ★(b) Prove that any standard Turing machine can be simulated by a Turing machine with only *two* states.

7. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

   (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

   (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

   - Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
   - Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
   - Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
   - Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

   Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

8. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current

cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

9. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

   Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem 4!)

10. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

    Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

★★★

- Tape alphabet = $\mathbb{N}$.
  - Read: zero or positive. Write: $+1$, -1
  - Read: even or odd. Write: $+1$, -1, $\times 2$, $\div 2$
  - Read: positive, negative, or zero. Write: $x + y$ (merge), $x - y$ (merge), 1, 0
- Never three times in a row in the same direction
- Hole-punch TM: tape alphabet $\{\square, \blacksquare\}$, and only $\square \mapsto \blacksquare$ transitions allowed.

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Machine de Turing

Nous avons vu qu'un programme peut être considéré comme la **décomposition** de la tâche à réaliser en une séquence d'instructions **élémentaires** (manipulant des données élémentaires) **compréhensibles** par l'automate programmable que l'on désire utiliser.

Cependant, chaque automate programmable (en pratique chaque processeur) se caractérise par un jeu d'instructions élémentaires qui lui est propre.

Dans la pratique, cette diversité est résolue par l'existence de compilateurs ou d'interpréteurs qui traduisent les instructions du langage (évolué) mis à disposition des programmeurs en les instructions élémentaires (langage machine) du processeur utilisé.

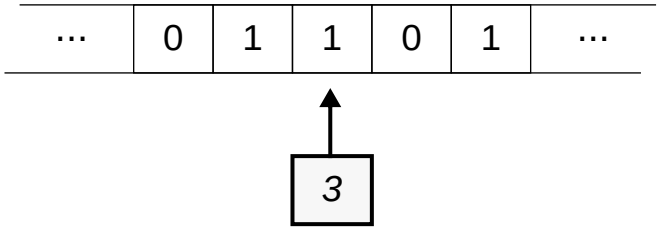
Cependant, cette solution n'est pas suffisante pour les besoins de l'informatique théorique, qui requière une représentation unifiée de la notion d'automate programmable, permettant de démontrer des résultats généraux (*décidabilité*, *complexité*, ...) vrais pour l'ensemble des automates programmables concrets que l'on peut envisager.

A cette fin a été développée la notion de ***machine de Turing***, qui constitue une abstraction (et une formalisation) de la notion d'automate programmable.

Machine de Turing: définition

Une *machine de Turing* est un **automate abstrait**, constitué des éléments suivants:

- une bande infinie, décomposée en cellules au sein desquelles peuvent être stockés des caractères (issus d'un ensemble fini).
- une tête de lecture/écriture, pouvant:
  - lire et modifier le caractère stocké dans la cellule correspondant à la position courante de la tête (le *caractère courant*)
  - se déplacer d'une cellule vers la gauche ou vers la droite (modifier la position courante)
- un ensemble fini d'états internes permettant de conditionner le fonctionnement de la machine
- une table de transitions indiquant, pour chaque couple (*état interne, caractère courant*) les nouvelles valeurs pour ce couple, ainsi que le déplacement de la tête de lecture/écriture.
  Dans la table de transisitions, chaque couple est donc associé à un triplet:
  (*état interne*$_{[nouveau]}$, *caractère*$_{[nouveau]}$, *déplacement*)



état courant \ caractère courant	**0**	**1**	ε
1	(1,0,+)	(1,0,+)	(2,ε,-)
2	(2,0,-)	(2,0,-)	(3,ε,+)

avec + respect. - indiquant un déplacement vers la droite, respect. vers la gauche.

Machine de Turing: fonctionnement

Une machine de Turing fonctionne selon le principe suivant:

1) la bande est initialisée avec la séquence de caractères correspondant aux données d'entrées;

2) la tête de lecture/écriture est positionnées sur la première cellule de la bande, et l'état interne est positionné à sa valeur initiale (par exemple 1)

3) tant que le couple courant (*état interne*, *caractère courant*) est présent dans la table de transitions, le triplet ($\textit{état interne}_{[\text{nouveau}]}$, $\textit{caractère}_{[\text{nouveau}]}$, *déplacement*) est utilisé pour mettre à jour l'état interne, le caractère courant, puis le déplacement de la tête de lecture/écriture est effectué.

4) lorsqu'est rencontré un couple (état interne, caractère courant) non recensé dans la table de transitions, la machine s'arrête et la séquence de caractères stockée à ce moment-là sur la bande est considérée comme le résultat du traitement.

Machine de Turing: exemple

Exemple: une machine de Turing déterminant si un nombre est pair

Entrée: le nombre à tester, sous forme binaire

Sortie: 1 si le nombre est pair, 0 sinon

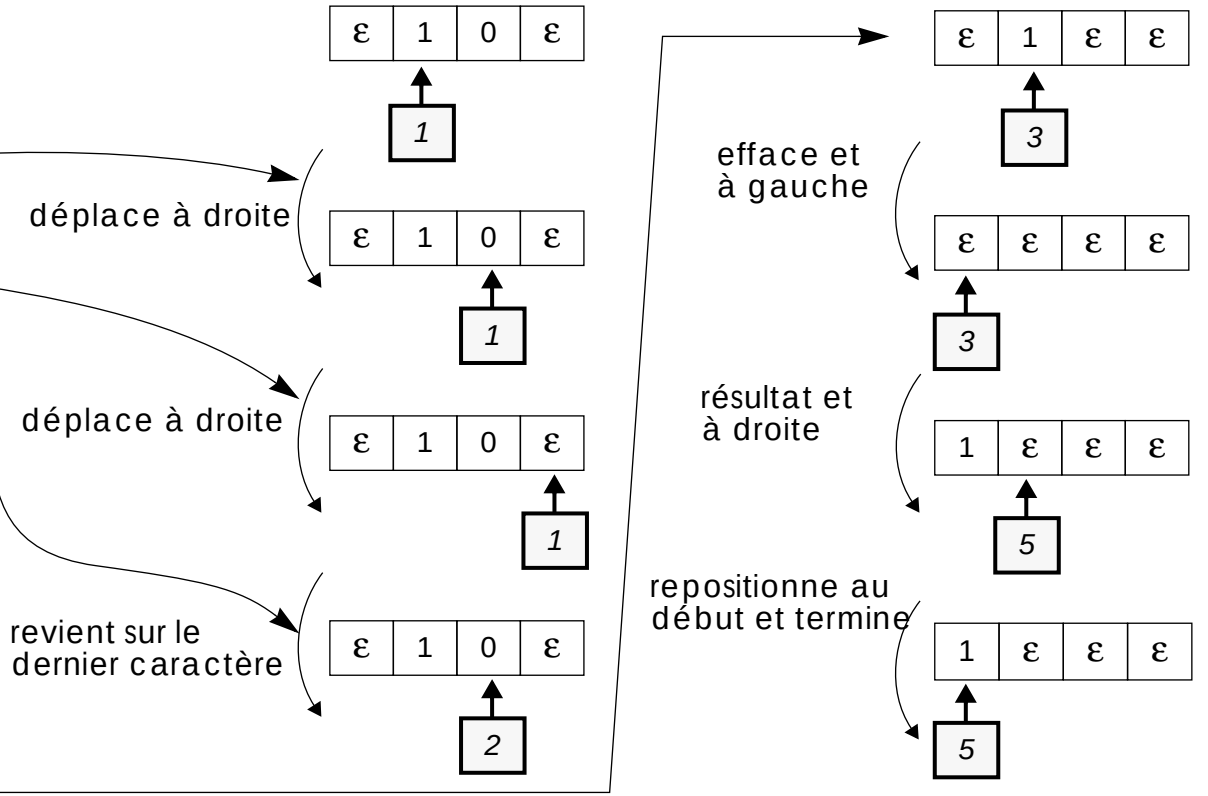
déplace à droite jusqu'à la fin

détecte la fin droite, revient sur le dernier caractère

état \ caractère	0	1	ε
1	(1,0,+)	(1,1,+)	(2,ε,-)
2	(3,ε,-)	(4,ε,-)	x
3	(3,ε,-)	(3,ε,-)	(5,1,+)
4	(4,ε,-)	(4,ε,-)	(5,0,+)
5	x	x	(6,ε,-)

revient au début en effaçant tout

Exemple pour l'entrée 2



Machine de Turing: forme canonique

On voit qu'une machine de Turing est caractérisée par

1. sa logique de fonctionnement;
2. le codage de ses entrées et sorties, sous forme de séquences de caractères;
3. la table de transitions décrivant son fonctionnement.

Si l'on impose de coder les entrées et les sorties sous forme binaire et d'indiquer l'absence de caractère dans une cellule par le caractère ε, on obtient une représentation uniforme des machines de Turing, appelée **représentation canonique**.

D'autres choix sont possibles pour la définition des machines de Turing (plusieurs bandes, autres opérations élémentaires, autres alphabets de caractères, ...) mais on peut montrer que les résultats théoriques obtenus avec de telles machines sont équivalents à ceux obtenus à l'aide d'une machine de Turing canonique.

Machine de Turing universelle (1)

Le fonctionnement d'une machine de Turing
est conditionné par sa table de transition.

une machine de Turing constitue donc une abstraction
pour la notion **d'automate *modifiable***...

... mais pas encore pour celle d'**automate *programmable***,
pour laquelle le programme doit faire partie des données d'entrée de la machine,
et non pas représenter un élément constitutif de l'automate (comme c'est le cas
pour la table de transitions d'une machine de Turing)

Si l'on désire qu'une machine de Turing constitue une abstraction pour la notion d'automate **programmable**, il faut donc que sa table de transitions soit fixe, et que le conditionnement de son fonctionnement soit **entièrement imposé par ses données d'entrées**.

mais comment construire une telle machine,
alors appelée ***machine de Turing universelle*** ?

Machine de Turing universelle (2)

On peut montrer que l'on peut effectivement
construire une *machine de Turing universelle* permettant
de simuler le fonctionnement d'une machine de Turing quelconque,
i.e. d'avoir le même comportement, pour ce qui est des
entrées et des sorties, tout en ayant
une table de transitions fixe.

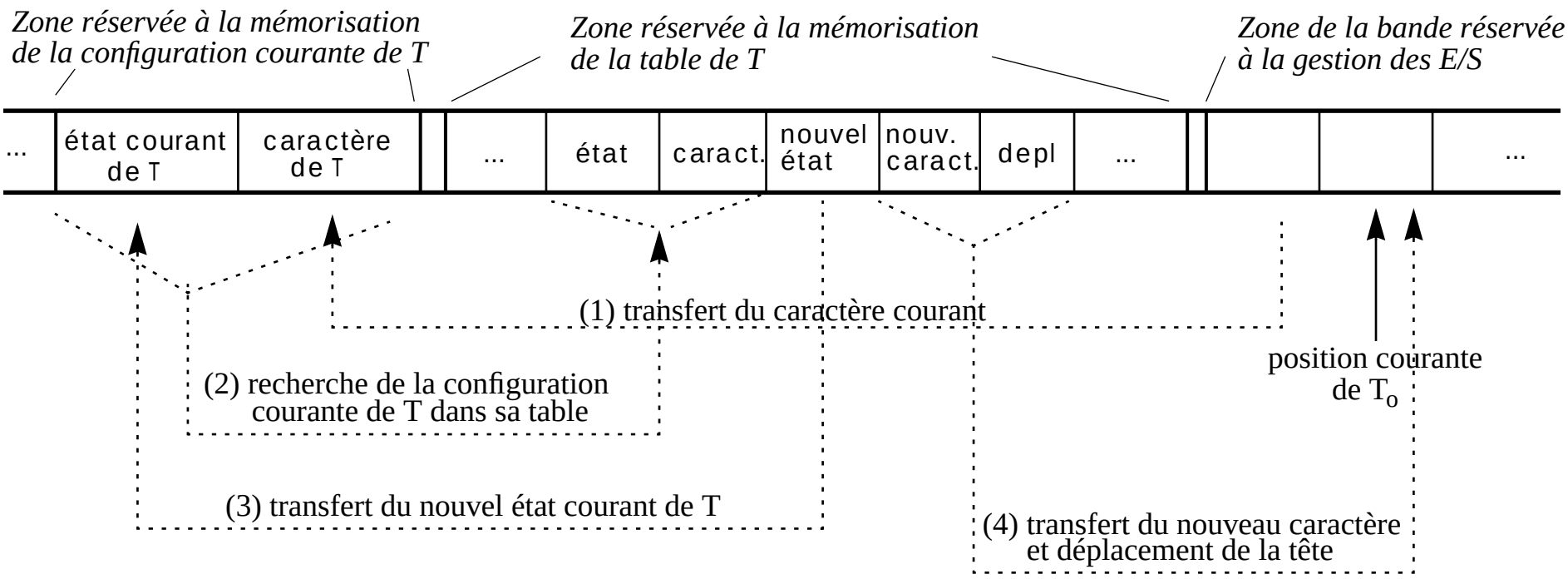
L'idée permettant de construire une telle machine T_o est la suivante:

1. la table de transitions de la machine T à simuler est codée sous la forme d'une séquence binaire;

2. la bande de la machine universelle est séparée en deux zones distinctes, l'une permettant de stocker la séquence binaire représentant la table de la machine T, ainsi que sa configuration (état interne, caractère courant), et l'autre permettant de gérer les véritables entrées/sorties.

Machine de Turing universelle (3)

Le fonctionnement de la machine universelle peut alors être schématisé ainsi:



Machine de Turing universelle (4)

La notion de machine de Turing universelle T_o
permet d'introduire les notations utiles suivantes:

Soit `T` une machine de Turing, on notera `(T)` la valeur entière
correspondant à la séquence binaire codant la table de transitions de `T`

Si `i` est une valeur entière correspondant au codage d'une table de transitions,
on notera T_i la machine de Turing universelle correspondante.
On a donc: $T = T_{i(T)}$

Si `i` est une valeur entière correspondant au codage d'une séquence d'entrée
pour une machine de Turing `T`, on notera `T(i)` la séquence de sortie produite par
l'application de `T` à la séquence d'entrée `i` (**attention**: `T(i)` peut
ne pas être définie, si `T` ne termine pas sur `i`)

Si T_o est la machine de Turing universelle, `T` une machine de Turing et `j` une séquence
d'entrée pour `T`, on notera $T_o(i(T),j)$ le résultat de la **simulation** de `T` sur `j` par T_o
On a donc: $T_o(i(T),j) = T(j)$
(lorsque ces valeurs sont définies)

Intérêt des machines de Turing

Les machines de Turing constituent une notion centrale en informatique, car:

- elles permettent de donner une base théorique solide aux notions importantes de ***décidabilité*** et de ***complexité*** (notions évoquées plus en détails plus loin dans ce cours);

- elles permettent de donner un contenu précis à la notion informelle d'**algorithme**

Algorithme: définition (1)

Algorithme: (définition informelle)

un algorithme, pour une tâche donnée, est la décomposition de cette tâche en une séquence finies d'étapes élémentaires permettant de la réaliser.

Si la tâche peut être **formalisée** sous la forme d'une association entre des entrées et des sorties, alors produire une machine de Turing permettant de réaliser cette association peut être vu comme une façon possible de résoudre la tâche.

La **table de transitions** de la machine produite est donc une **représentation particulière possible de l'algorithme** de résolution associé.

Algorithme: définition (2)

Les machines de Turing fournissent donc,
par définition, une représentation possible pour
les algorithmes de résolution des tâches pouvant
être traitées par des machines de Turing.

De plus, l'expérience acquise dans le domaine de
l'informatique invite à penser que les machines de Turing
constituent en fait un **outil encore plus puissant**:

Thèse de Church: **tout** algorithme peut être représenté
sous la forme d'une table de transitions
d'une machine de Turing.

L'idée sous-jacente est que les machines de Turing, malgré leur apparente simplicité, possèdent en fait une puissance suffisante pour permettre de résoudre tout problème pouvant être traité de façon automatique.

une formulation savante de la thèse de Church est doc:

«tout problème calculable est Turing calculable »

Algorithmique

Un ***algorithme*** est:

une **suite finie** de règles à appliquer,
dans un **ordre déterminé**,
à un **nombre fini** de données,

pour arriver, en un nombre fini d'étapes, à un résultat,
et cela quelque soit les données traitées. *[Encyclopedia Universalis]*

Un algorithme correspond donc à la partie **conceptuelle** d'un programme, indépendante du langage de programmation.

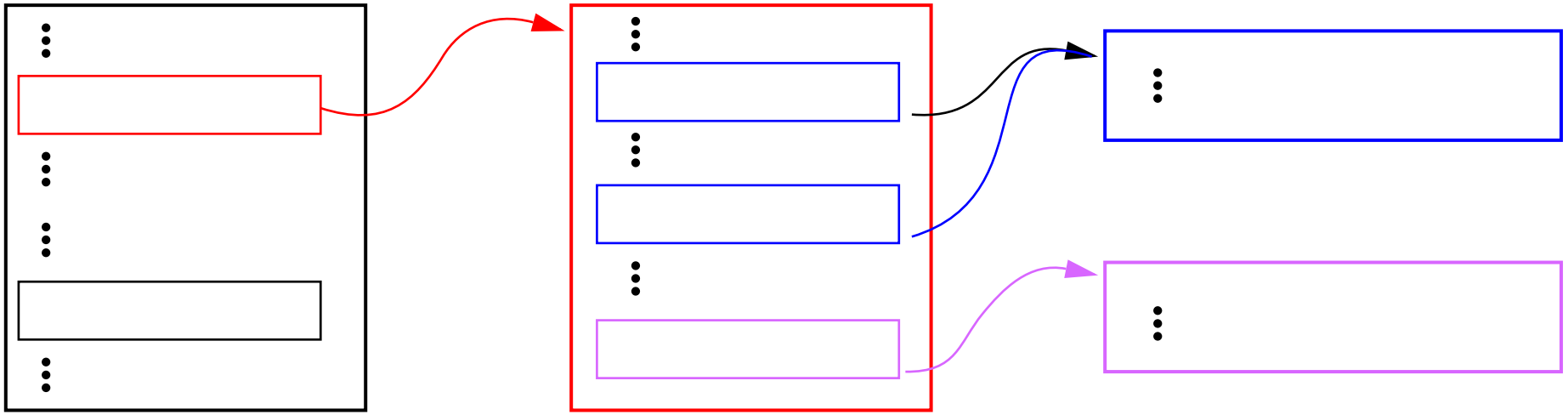
Le programme est alors **l'implémentation** de l'algorithme, dans un langage de programmation (et sur un système) particulier.

Approche descendante

Méthode de résolution: *(élaboration de solutions algorithmiques)*

On résoud généralement un problème par une ***analyse descendante***, c'est-à-dire une analyse **du plus général au plus spécifique**.

Cette analyse se fait à l'aide de **blocs imbriqués**,[1] correspondant à des traitements de plus en plus spécifiques, eux aussi décrits à l'aide **d'algorithmes**.[2]



1. Notons le parallèle évident avec l'implémentation du programme décomposé en fonctions
2. La méthode de résolution est donc itérative: on résoud par analyses successives, à des niveaux de détails de plus en plus précis.

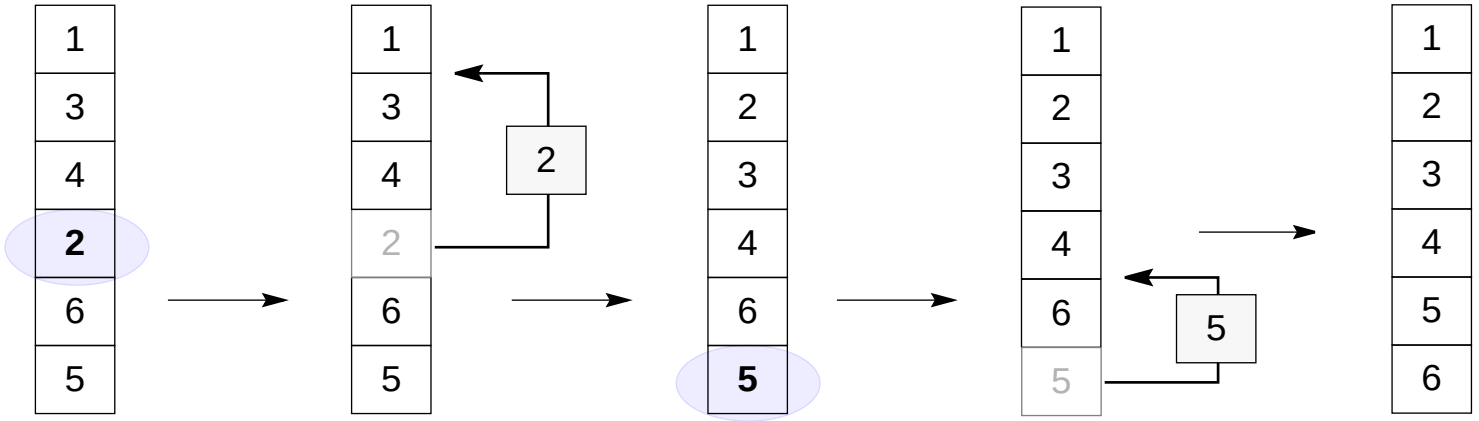
Exemple: tri par insertion

Un algorithme: **tri (d'un tableau) par insertion**

le principe du tri par insertion est extrêmement simple:

«tout élément *mal placé* dans le tableau va être déplacé vers (inséré à) *sa bonne place*»

- Par «élément *mal placé*», on entend ici un élément pour lequel la relation d'ordre (utilisée pour le tri) avec l'élément prédécesseur n'est pas satisfaite... par exemple, un élément strictement plus petit que son prédécesseur, pour un tri suivant l'ordre croissant.
- Par «sa bonne place», on entend ici une position –parmis les éléments déjà triés– pour laquelle la relation d'ordre entre le prédécesseur et l'élément est satisfaite, ainsi qu'entre l'élément et son successeur.



tri par insertion: résolution globale

Le schéma général de l'algorithme de tri par insertion est le suivant:

tri insertion

---

entrée: un tableau d'entiers quelconque

sortie: un tableau d'entiers trié (mêmes éléments que l'entrée)

---

chercher un élément *mal placé*

tant qu'un tel élément existe:

    chercher *sa bonne place*

    déplacer l'élément vers sa bonne place

    chercher un élément *mal placé*

fin tant que

tri par insertion: résolution détaillée (1)

- Le bloc *mal placé* prend en entrée un tableau `tab`, et donne en sortie l'indice du premier élément de `tab` de prédécesseur strictement plus grand. Par convention, si'il n'existe pas d'élément admettant un tel sucesseur, la sortie sera `MAX`, la taille du tableau.

- Comme le 1er élément de `tab` ne peut être mal placé (puisque sans prédécesseur), le bloc *mal placé* doit donc parcourir les éléments de `tab`, à la recherche d'un élément mal placé, en partant du second élément.

- Le bloc *mal placé* correspond donc à une itération sur les éléments de tab, du deuxième élément au dernier (d'indice `MAX-1`).

tris par insertion: résolution détaillée (2)

Un algorithme pour le bloc *mal placé* sera:

mal placé

entrée: un tableau d'entiers

sortie: l'indice du premier élément mal placé

pour chaque élément du tableau, à partir du deuxième:
    comparer cet élément au précédent
    s'il est plus petit, alors il est mal placé:
        retourner l'indice de cet élément (et **terminer**)
    fin si
fin pour
retourner MAX, la taille du tableau (et **terminer**)

tri par insertion: résolution détaillée (3)

- Le bloc *bonne place* prend en entrée un tableau `tab` et l'indice `pos` d'un élément mal placé, et donne en sortie l'indice `newpos` de la «bonne place» de l'élement dans le tableau.

- La «bonne place» correspond à la position d'indice `newpos` (<`pos`) le plus grand tel que `tab[newpos-1] <= tab[pos]`, où 0 si cette relation n'est pas satisfaite pour `newpos` = 1.

- Le bloc *bonne place* doit donc parcourir les positions de tab, entre le premier élément et (au maximum) pos, à la recherche de la «bonne position»

- Le bloc *bonne place* correspond donc à une itération sur les éléments de `tab`, du premier élément à celui d'indice `pos`.

tri par insertion: résolution détaillée (4)

Un algorithme pour le bloc *bonne place* sera:[3]

bonne place

---

entrée: un tableau d'entiers `tab`, l'indice `pos` d'un élément mal placé

sortie: l'indice de la bonne place de l'élément

---

pour chaque élément du tableau, de 0 à `pos`:

    si l'élément est plus grand que `tab[pos]`:

        retourner l'indice de cet élément (et **terminer**)

    fin si

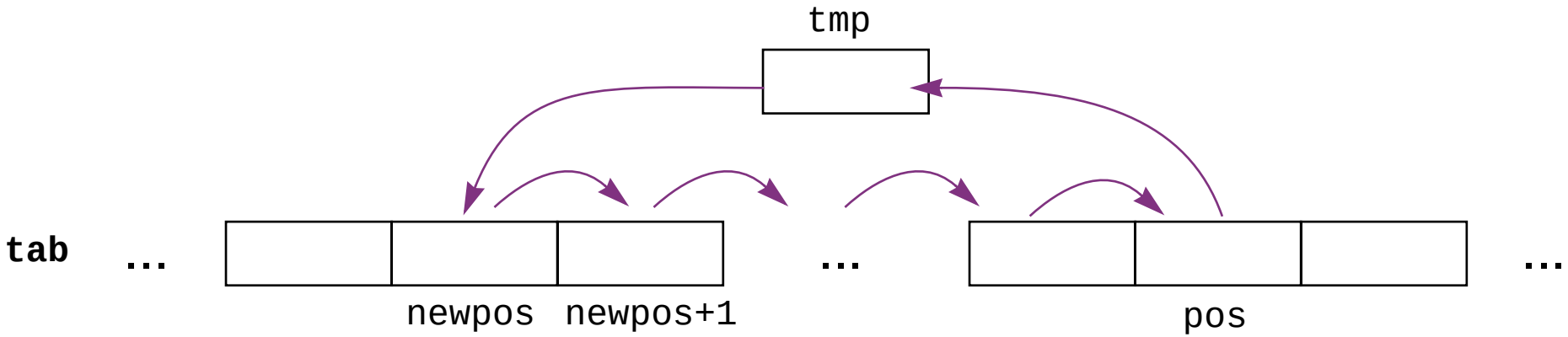
fin pour

3. L'algorithme correspond à une interprétation (équivalente en terme de résultat) de la description du transparent précédent !

tri par insertion: résolution détaillée (5)

- Le bloc *déplacer* prend en entrée un tableau `tab`, l'indice `pos` d'origine d'un élément, et l'indice `newpos` de destination de cet élément, et doit déplacer ledit élément de sa position d'origine à sa positition de destination.

On peut effectuer cette opération par **décalages successifs**
(en utilisant un stockage temporaire `tmp`)



tri par insertion: résolution détaillée (6)

Un algorithme pour le bloc *déplacer* sera

déplacer

entrée: un tableau, un indice d'origine `pos`, un indice final `newpos`

sortie: le tableau dans lequel l'élement d'indice d'origine à été déplacé et possède maintenant l'indice final (l'ordre des autres éléments est conservé)

mémoriser l'élément d'indice `pos`

pour chaque élément du tableau entre `pos-1` et `newpos`

    copier cet élément à la position d'indice `courant+1`

fin pour

copier l'élément mémorisé à la position d'indice `newpos`

Paradigme «diviser pour résoudre» (1)

Il n'existe pas de méthode miracle pour fabriquer les solutions algorithmiques requises pour les problèmes auxquels l'informaticien peut être confronté.

Cependant, un certain nombre de schémas généraux de résolution peuvent être utilisés.

Parmi ces schémas, on peut citer le paradigme «***diviser pour résoudre***» (*divide and conquer*), qui est fréquemment utilisé.

Diviser pour résoudre (2)

Le schéma général d'une approche «diviser pour résoudre» de résolution d'un problème P appliqué à des données X est le suivant:

P(X):

si X est suffisement simple ou petit, appliquer *adhoc*(X),
sinon:
décomposer X en instances plus petites, x_1, ..., x_n
puis pour tout x_i (1 <= i <= n):
résoudre P(x_i) pour obtenir une solution partielle y_i
recombiner les y_i pour obtenir une solution globale Y pour P(X)

adhoc(X)

est un algorithme permettant de résoudre P pour des données X simples.

Diviser pour résoudre: exemple (1)

Un premier exemple d'application du paradigme «diviser pour résoudre» à déjà été présenté, à l'occasion de l'étude des fonctions récursives, avec le tri récursif par insertion:

tri(X):

si X est vide ou réduit à 1 seul élément, X est solution
sinon:
décomposer X en:
x_1 = {X[1]} où X[1] est le 1er élement de X,
x_2 = X \ X[1]
résoudre *tri*(x_1) (cas trivial) -> y_1
résoudre *tri*(x_2) (appel récursif) -> y_2
produire une solution pour X en insérant y_1 à *sa bonne place* dans y_2, et retourner la combinaison obtenue.

Diviser pour résoudre: exemple (2)

Recherche par dichotomie:

Objectif: déterminer si un élément *x* appartient à un ensemble E d'éléments $E = \{x_1, ..., x_n\}$

Entrée: *x* et E (sous la forme d'un tableau trié)

Sortie: *vrai* ou *faux* selon que *x* appartient ou non à E.

Schéma de résolution:

appartient(*x*,E):
si E est vide,
la solution est *faux*
sinon si $E = \{x_1\}$,
la solution est vrai si $x=x_1$, faux sinon
sinon ($E = \{x_{1, ..., } x_n\}$)
décomposer E en:
$E_1 = \{x_{n/2}\}$ avec n/2 la division entière de n par 2 !
$E_2 = \{x_1, ..., x_{(n/2)-1}\}$
$E_3 = \{x_{(n/2)+1}, ..., x_n\}$
si y_1 = *appartient*(*x*,E_1) est *vrai*, alors Y est *vrai*
sinon si $x < x_{n/2}$ alors $Y = y_2$ = *appartient*(*x*, E_2)
sinon $Y = y_3$ = *appartient*(*x*, E_3)
(la solution est Y)