Foundations of Computer Science
Logic, models, and computations

Chapter: Turing machines

Course CSC_41012_EP

of l'Ecole Polytechnique

Olivier Bournez

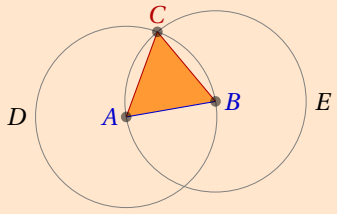
bournez@lix.polytechnique.fr

Version of August 20, 2024

Turing machines

We have used many times up to know the notion of algorithm but without having provided a formal definition. Intuitively, one can say that an algorithm is an automatic method, that can be implemented on some computer, to solve a given problem. For example, the familiar techniques to perform an addition, or multiplication or a division on numbers learned at elementary school are algorithms. The techniques discussed to evaluate the truth value of a propositional formula from the value of its variables are also algorithms. More generally, we have discussed some proof methods for propositional calculus or for predicate calculus that can be seen as algorithms.

Example 1 *The following example taken from the manual of package* TikZ-PGF *version 2.0, inspired in turn from the* Elements *of* Euclid*, can be considered as an algorithm. It provides a method to draw an equilateral triangle with edge AB.*



***Algorithm to construct a equilateral triangle with edge** AB: draw a circle of center A and radius AB; draw the circle of center B of radius AB. Name C one of the intersection of the two circles. The triangle ABC is the desired solution.*

We will see in the following chapters that not all the problems can be solved by algorithms, and even for problems very easy to formulate: For example,

- there is no algorithm to determine if a given closed formula of predicate calculus is valid in the general case;
- there is no algorithm to determine if a multivariate polynomial (i.e. with several variables) with integer coefficients has an integer root (*Hilbert* 10*th problem*).

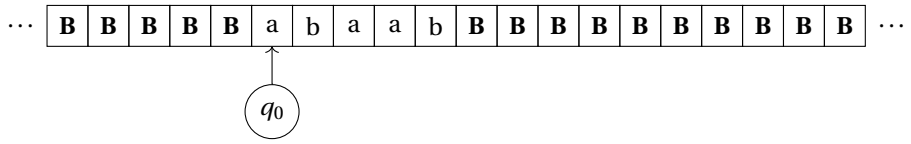


Figure 1: Turing machine. The machine is on the initial state of a computation on word $abaab$.

Historically, it was actually the formalization of the notion of proofs in mathematics and the question of limits of formal proof systems that led to the models that we will discuss. It was later understood that the notion captured by these models was more than simply the formalization of the notion of proof and that it was actually also a formalization of everything that can be computed by a digital machine. This remains true today since today's computers are digital.

Furthermore, several formalizations have been proposed, in an independent way, using at first sight very different notions: In particular, Alonzo Church in 1936, using the formalism of *λ-calculus*,, Alan Turing in 1936, using what is now called the *Turing machines*, or Emil Post in 1936, using systems based on very simple rules, called *Post systems*. Later, it was shown that these formalisms are all equivalent.

The objective of this chapter is to describe the Turing machine model. In the next chapter, we will define a few other models of computation and will show that they are equivalent to the Turing machine model.

We will then by talk about the *Church-Turing thesis*.

The models that we are going to describe can all be considered as very abstract, and might at first sight give the impression to be very limited, and far from being able to cover everything that can be programmed using today's languages such as Python, CAML or JAVA. The main objective of this chapter and of the next one is to convince the reader that this is NOT the case: Everything that can be programmed can actually be programmed using these "basic" models.

1 Turing machines

1.1 Ingredients

A (deterministic) Turing machine (See Figure 1) is composed of the following elements:

1. An infinite memory of the form of a *tape*. The tape is divided in cells. Each cell can contain an element of a set Σ (i.e. of some alphabet). We assume that the alphabet Σ is some finite set.

2. a (reading/writing) head that can move along the tape.

3. A program given as a *transition function* that, for every internal state q of the machine, among a finite number possible internal states Q, gives according to the symbol under the reading head:
   (a) the next internal state $q' \in Q$;
   (b) the new element of Σ to write in place of the element of M currently in front of the head;
   (c) a moving direction for the reading head.

The execution of a Turing machine on some word $w \in \Sigma^*$ can then be described as follows: initially, the input w is on the tape, and the head is positioned in front of the first letter of the word. The cells not corresponding to the input all contain the element **B** (blank symbol), that is a special letter. The machine is in its initial internal state q_0: See Figure 1.

At every execution step, the machine, reads the symbol in front of the head, and, according to its internal state and this symbol, following its program, it:

- replaces the symbol in front of the head by the one given by the transition function;
- (possibly) moves its head to the left or to the right, according to the direction given by the transition function;
- changes the internal state to the internal state given by the transition function.

The word w is said to be accepted when the execution of the Turing machine eventually reaches the accepting internal state, in which case the execution of the machine stops.

In the next section, we will give a formal definition of Turing machines and their execution.

1.2 Description

Definition 1 (Turing machine) A Turing machine *is an 8-uple*

$$M = (Q, \Sigma, \Gamma, \mathbf{B}, \delta, q_0, q_a, q_r)$$

where:

1. Q *is the finite set of internal states;*
2. Σ *is a* finite *alphabet;*
3. Γ *is the* finite *tape alphabet with* $\Sigma \subset \Gamma$*;*
4. $\mathbf{B} \in \Gamma$ *is the blank symbol;*
5. $q_0 \in Q$ *is the initial state;*

6. *$q_a \in Q$ is the accepting state;*
7. *$q_r \in Q$ is the rejecting state;*
8. *δ is the transition function: δ is a function (possibly partial) from $Q \times \Gamma$ to $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$. The symbol $\leftarrow$ is used to mean a move to the left, $|$ no move, $\rightarrow$ a move to the right.*

The language accepted by a Turing machine is defined using the notions of *configurations* and of successor relation between configurations of a Turing machine. A configuration consists of all the information required to describe the state of the machine at a given moment, and to determine the future states of the machine, namely:

- the internal state;
- the content of the tape;
- the position of the head.

We give a more formal definition.

Definition 2 (Configuration) A configuration *is given by the description of the tape, the position of the head, and the internal state.*

To write a configuration, one difficulty is that the tape is infinite, and thus consists of an infinite sequence of symbols of the tape alphabet Γ of the machine. However, we focus on finite executions, and consequently, at any moment of an execution, only a finite part of the tape has been visited by the machine. Indeed, initially the tape contains an input of finite length, and at every step the machine moves its head at most of one cell. As a consequence, after t steps, the head has moved at most t cells to the left or to the right from its initial position. Consequently, the content of the tape can be defined at any moment by a fixed sequence of symbols, the rest containing only the blank symbol **B**.

To denote the position of the head, we could use some integer $n \in \mathbb{Z}$. We will actually use the following trick that has the advantage of simplifying the coming definitions: Instead of seeing the tape as a finite sequence, we will represent it by two finite sequences: The content of what is on the right and what is on the left of the head. We will write the right prefix as usual from left to right. In contrast, we will write the prefix corresponding to what is on the left of the head from right to left: The interest is that the first letter of the left prefix is the letter immediately at the left of the head. A *configuration* will hence be an element of $Q \times \Gamma^* \times \Gamma^*$.

Formally:

Definition 3 (Denoting a configuration) *A configuration is denoted by $C = (q, u, v)$, with $u, v \in \Gamma^*$, $q \in Q$: u and v respectively denote the content of the tape respectively to the left and to the right of the head which is in front of the first letter of v. We assume that the letters of u and of v are not containing the blank symbol* **B**.

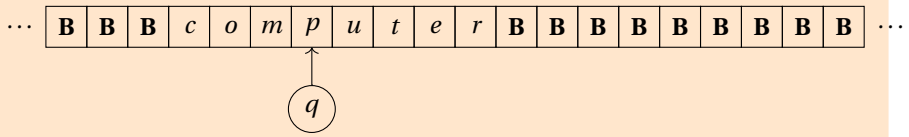
We make the convention that the word v is written from left to right (letter number $i+1$ of v corresponds to the cell at the right of the letter/cell number i) while the word u is written from right to left (the letter number $i+1$ of u corresponds to the content of tape at the left of letter/cell number i, the first letter of u being the cell immediately at the left of the head).

Example 2 *The configuration of the machine represented on Figure 1 is* $(q_0, \epsilon, abaab)$.

We will sometimes write configurations in an other way.

Definition 4 (Alternative notation) *The configuration (q,u,v) will also been seen/-denoted in some sections or chapters by $u\boldsymbol{q}v$, keeping u and v written from left to right.*

Example 3 *A configuration such as*



is encoded by configuration $(q, moc, puter)$, or sometimes by $com\boldsymbol{q}puter$.

A configuration is said to be *accepting* if $q = q_a$, *rejecting* if $q = q_r$.

For $w \in \Sigma^*$, the initial configuration corresponding to w is the configuration $C[w] = (q_0, \epsilon, w)$.

We write: $C \vdash C'$ if the configuration C' is the immediate successor of configuration C by the program (given by δ) of the Turing machine.

Formally, if $C = (q,u,v)$ and if a denotes the first letter [1] of v, and if $\delta(q,a) = (q',a',m')$ then $C \vdash C'$ if $C' = (q',u',v')$, and

- if $m' = |$, then $u' = u$, and v' is obtained by replacing the first letter a of v by a';
- if $m' = \leftarrow$, u' is obtained by deleting the first letter a'' of u, v' is obtained by concatenating a'' and the result of replacing the first letter a of v by a';
- if $m' = \rightarrow$, $u' = a'u$, and v' is obtained by deleting the first letter a of v.

[1] With the convention that the first letter of the empty word is the blank symbol **B**.

Remark 1 *The above rules are formalizing the changing of the cell in front of the head from a to a' and the corresponding potential shift of the tape to the right or to the left.*

Definition 5 (Accepted word) *A word $w \in \Sigma^*$ is said to be* accepted *(in time t) by the Turing machine, if there exists a sequence of configurations $C_1, \cdots, C_t$ with:*

1. *$C_0 = C[w]$;*
2. *$C_i \vdash C_{i+1}$ for all $i < t$;*
3. *none of the configurations C_i for $i < t$ is accepting or rejecting*
4. *C_t is accepting.*

Definition 6 (Rejected word) *A word $w \in \Sigma^*$ is said to be* rejected *(in time t) by the Turing machine, if there exists a sequence of configurations $C_1, \cdots, C_t$ with:*

1. *$C_0 = C[w]$;*
2. *$C_i \vdash C_{i+1}$ for all $i < t$;*
3. *none of the configurations C_i for $i < t$ is accepting or rejecting.*
4. *C_t is rejecting.*

Definition 7 (Machine that loops on a word) *We say that a Turing machine* loops *on a word w, if w is neither accepted nor rejected.*

Remark 2 *For every word w we thus have exactly one of the following three exclusive cases:*

1. *it is accepted by the Turing machine;*
2. *it is rejected by the Turing machine;*
3. *the machine loops on this word.*

Remark 3 *The terminology* loops *means simply that the machine is not halting on this word: This does not necessarily mean that one repeats for ever the same instructions. The machine can loop for several reasons. For example, since it reaches a configuration that has no successor configuration that is defined, or because it enters a complex behaviours that produces an infinite sequence of*

configurations not accepting nor rejecting.

More generally, one calls *computation of M on a word $w \in \Sigma^*$*, a (finite or infinite) sequence of configurations $(C_i)_{i\in\mathbb{N}}$ such that $C_0 = C[w]$ and for all i, $C_i \vdash C_{i+1}$, with the convention that an accepting or rejecting configuration has no successor.

Definition 8 (Language accepted by a machine) *The language $L \subset \Sigma^*$* accepted by Turing machine *M is the set of words w that are accepted by the machine. It is denoted by L(M). We also call $L(M)$ as the* language recognized *by M.*

A machine that does not halt is usually undesirable. Thus, we generally try to guarantee a stronger property:

Definition 9 (Langage decided by a machine) *One says that a language $L \subset \Sigma^*$ is* decided *by the machine M if:*

- *for every $w \in L$, w is accepted by M;*
- *for every $w \notin L$ (=otherwise), w is rejected by M.*

In other words, the machine accepts L and terminates on every input (i.e. it never loops).

One says in that case that the machine *M decides L.*

1.3 Programming with Turing machines

Programming with Turing machines is extremely low level. We will however see that one can really program many things with this model. The first step is to get convinced that we can program solutions to many problems with Turing machines. To say the truth, the only way to get convinced is to try to program by oneself with Turing machines, for example by trying to solve the following exercises.

Exercise 1 *Construct a Turing machine that accepts exactly the words w on alphabet $\Sigma = \{0,1\}$ of the form 0^n1^n, $n \in \mathbb{N}$.*

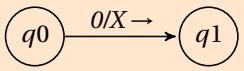
Here is a solution. Consider a machine $Q = \{q0, q1, q2, q3, q4\}$, $\Gamma = \{0, 1, X, Y, B\}$, the accepting state $q4$ and the transition function δ such that:

- $\delta(q0, 0) = (q1, X, \rightarrow)$;
- $\delta(q0, Y) = (q3, Y, \rightarrow)$;
- $\delta(q1, 0) = (q1, 0, \rightarrow)$;
- $\delta(q1, 1) = (q2, Y, \leftarrow)$;
- $\delta(q1, Y) = (q1, Y, \rightarrow)$;

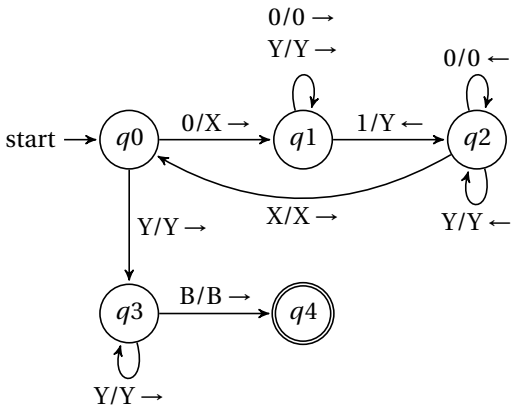
- $\delta(q2,0) = (q2,0,\leftarrow)$;
- $\delta(q2,X) = (q0,X,\rightarrow)$;
- $\delta(q2,Y) = (q2,Y,\leftarrow)$;
- $\delta(q3,Y) = (q3,Y,\rightarrow)$;
- $\delta(q3,B) = (q4,B,\rightarrow)$.

As one can see, a description of a Turing machine in this way is essentially unreadable. We thus prefer a representation of the program of a machine (the function δ) in the form of a graph: The vertices of the graph represent the states of the machine. Every transition $\delta(q,a) = (q',a',m)$ is represented with an arc from the state q to the state q' labeled by $a/a'\ m$. The initial state is marked with an incoming arc. The accepting state is marked with a double circle.

Example 4 *For example, the transition $\delta(q0,0) = (q1,X,\rightarrow)$ is represented graphically by:*



With these conventions, the previous program can be represented by:



How does this program work? During a computation, the part of the tape that the machine has visited will be of the form $X^*0^*Y^*1^*$. Every time that a 0 is read, it is replaced by X and one goes to state $q1$ which starts the following sub-procedure: One moves right as long as a 00 or a Y is read. As soon as a 1 is reached, it is transformed into a Y, and one goes back to the left until one reaches a X (the X that has been written) and then one does a one cell right shift.

Doing so, for each 0 that is erased (i.e. marked by a X), we will have erased a 1 (i.e. marked a Y). If all the 0 are marked and one reaches a Y, one goes to state $q3$, which has the effect of checking that what is on the right is indeed only consists of

Y's. Once everything has been read, i.e. a **B** is reached, one accepts, i.e. one goes to state $q4$.

Of course, a true proof of the correctness of this algorithm would consist in proving that if a word is accepted, it is necessarily of type 0^n1^n. We leave to the reader to get convinced of this.

Example 5 *Here is an example of accepting computation for M: $q_0 0011 \vdash X q_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1 \vdash q_2 X0Y1 \vdash Xq_0 0Y1 \vdash XXq_1 Y1 \vdash XXYq_1 1 \vdash XXq_2 YY \vdash Xq_2 XYY \vdash XXq_0 YY \vdash XXYq_3 Y \vdash XXYYq_3\mathbf{B} \vdash XXYY\mathbf{B}q_4\mathbf{B}$.*

Definition 10 (Space-time diagram) *One often represents a sequence of configurations line by line: The line number i represents the ith configuration of the computation, using the encoding of Definition 4. This representation is called a* space-time diagram *of a machine.*

Example 6 *Here is the space-time diagram corresponding to the previous computation on* 0011.

$\cdots$	**B**	**B**	**B**	**B**	q_0	0	0	1	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_1	0	1	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	0	q_1	1	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_2	0	Y	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	q_2	X	0	Y	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_0	0	Y	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_1	Y	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	q_1	1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_2	Y	Y	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	q_2	X	Y	Y	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	q_0	Y	Y	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	q_3	Y	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	Y	q_3	B	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$
$\cdots$	**B**	**B**	**B**	**B**	X	X	Y	Y	B	q_4	B	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	$\cdots$

Example 7 *Here is the space-time diagram of the computation of the machine on* 0010*:*

⋯	**B**	**B**	**B**	**B**	q_0	*0*	*0*	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	q_1	*0*	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	*0*	q_1	*1*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	q_2	*0*	*Y*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	q_2	*X*	*0*	*Y*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	q_0	*0*	*Y*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	*X*	q_1	*Y*	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	*X*	*Y*	q_1	*0*	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	*X*	*X*	*Y*	*0*	q_1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯

Observe that in the last configuration no continuation is possible, and hence that there is no accepting computation starting from 0010.

Exercise 2 *(solution on page 210) [Subtraction in unary] Construct a Turing machine program that realizes a subtraction in unary: Starting from a word of the form* 0^m10^n*, the machine stops with* $0^{m\ominus n}$ *on its tape surrounded by blanks (where* $m\ominus n$ *is* $\max(0, m-n)$*).*

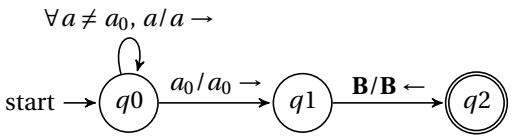
1.4 Some programming techniques

Here are a few programming techniques that are used for programming Turing machines.

The first consists in encoding some finite information in the internal state of the machine. We will illustrate this on an example, where we will store the first read character in the internal state. As long as the information to store is finite, this is possible.

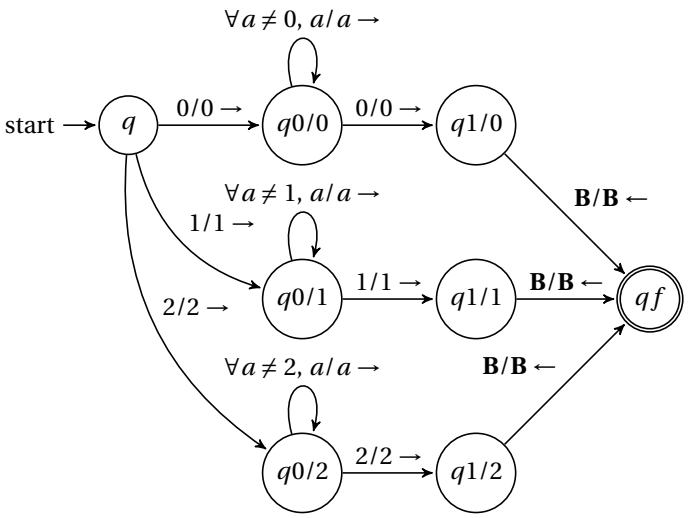
Exercise 3 *Construct a Turing machine that reads the symbol in front of the head and checks that this character letter is also the last letter of the input word but does not appear anywhere else in the input.*

If one fixes the symbol $a_0 \in \Sigma$ of alphabet Σ, it is easy to construct a program that checks that the symbol a_0 is appearing nowhere but as the last letter on the right. Indeed, consider:



where $\forall a \neq a_0$ means that one repeats the transition $a/a, \rightarrow$ for all symbols $a \neq a_0$.

Now, to solve our problem, it is sufficient to read the first letter a_0 and to copy this program as many times as the number of letters in alphabet Σ. If $\Sigma = \{0,1,2\}$ for example:



We are hence using the fact that this program is working on some states that can be ordered pairs: here we use ordered pair q_i/j with $i \in \{1,2\}$, and $j \in \Sigma$.

A second technique is using subprocedures. Once again, we will illustrate this with an example.

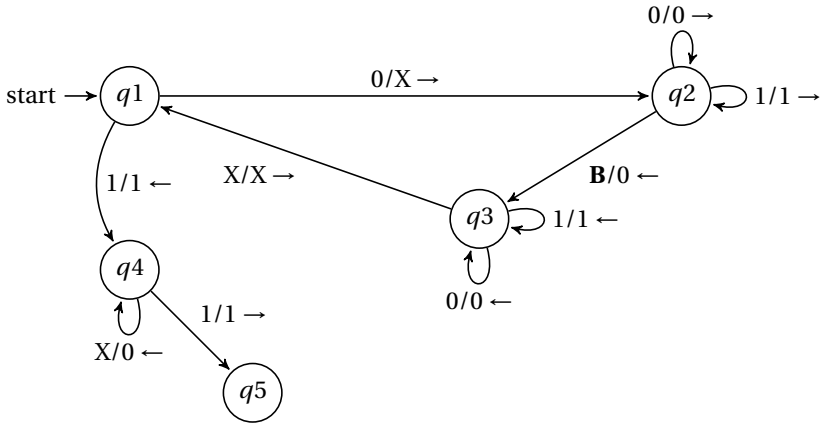
Exercise 4 *[Multiplication in unary] Construct a Turing machine that performs multiplication in unary: Starting from a word of the form 0^m10^n, the machine stops with $0^{m\cdot n}$ on its tape.*

A possible strategy is the following:

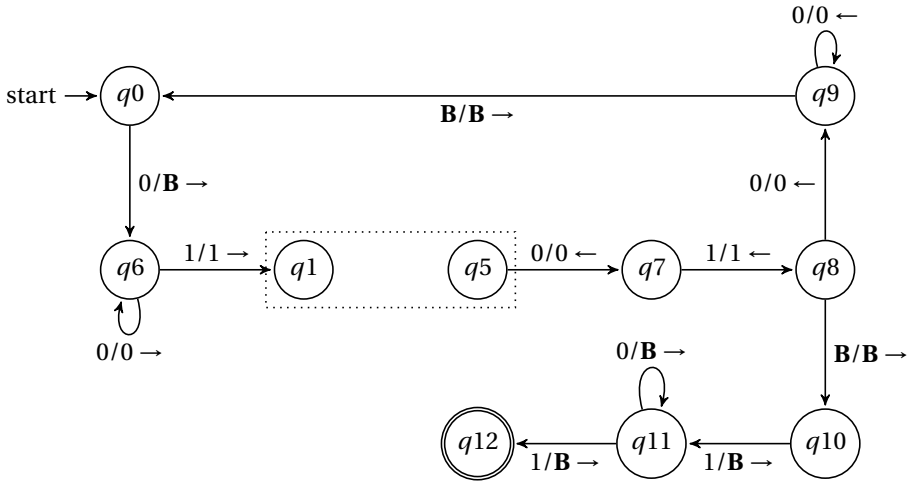
1. the tape will contain a word of the form $0^i10^n10^{kn}$ for an integer k;
2. at every step, one will change a 0 of the first group into a blank, and one will add n 0's to the first group, to obtain the word $0^{i-1}10^n10^{(k+1)n}$;
3. by doing so, we will copy the group of n 0's m times, once for every symbol of the first group set to blank. When there is no blank left in the first group of 0's, there will consequently be $m \cdot n$ 0's in the last group;
4. the last step is to overwrite the prefix 10^n1 with blanks.

The heart of the method is hence the subprocedure, that we will call $Copy$ that implements step 2: It transforms a configuration $0^{m-k}1q_10^n1^{(k-1)n}$ into $0^{m-k}1q_50^n1^{kn}$.

Here is a way to program this: If one starts in state q_1 with such an input, we will eventually go to state q_5 with the correct result.



Once we have this subprocedure, one can construct the global algorithm.



where the dashed rectangle means "paste the program just describe for the subprocedure here".

In this example we see that it is possible to program with Turing machines in a modular way, by using subprocedures. In practise, this means pasting pieces of program inside a program of a machine as in the example.

1.5 Applications

As we said before, the only way to understand all what can be programmed with a Turing machine is trying to program them.

So here are a few exercises.

Exercise 5 *Construct a Turing machine that adds* 1 *to the number written in binary (hence with* **0** *and* **1***'s) on its tape.*

Exercise 6 *Construct a Turing machine that subtracts* 1 *to the number written in binary (hence with* **0** *and* **1***'s) on its tape.*

Exercise 7 *Construct a Turing machine that accepts the words with the same number of* **0***'s and* **1***'s.*

1.6 Variants of the notion of Turing machine

The Turing machine model is extremely robust.

Indeed, there are many possible variations of the model, but they do not change what can be computed with these machines.

In this section, we will illustrate this by discussing several of these modifications.

Restriction to a binary alphabet

Proposition 1 *Every Turing machine working over some arbitrary alphabet* Σ *can be simulated by a Turing machine working over alphabet* $\Sigma = \Gamma$ *with only two letters (without counting the blank symbol).*

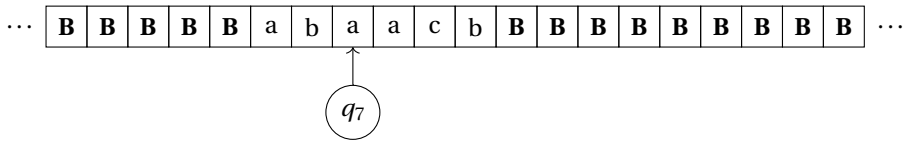
Sketch of proof: The idea is that one can always encode the letters of the alphabet by using a binary encoding. For example, if the alphabet Σ has 3 letters a, b, and c, one can decide to encode a by 00, b by 01 and c by 10: See Figure 2. In the general case, one just needs to use possibly more than 2 letters.

Of course, this may require first to initially transform the input in order to rewrite it using this encoding.

One can then transform the program of a Turing machine M that works over alphabet Σ into a program M' that works over this encoding.

For example, if the program of M contains an instruction that says that if M is in state q and that the head reads a one must write c and move to the right, the program of M' will consist in stating that if one is in state q and one reads 0 in front of the head, and 0 on its right (and so what is on the right of the head starts by 00, the encoding of a), then one must replace these two 0's by 10 (i.e. the encoding of c) and go to state q'. By doing so, every time that a computation of M produces a tape corresponding to some word w, then the machine M' will produce a tape corresponding to the encoding of w in binary letter by letter. □

Machine M on alphabet $\{a,b,c\}$



Machine M' simulating M on alphabet $\{0,1\}$.

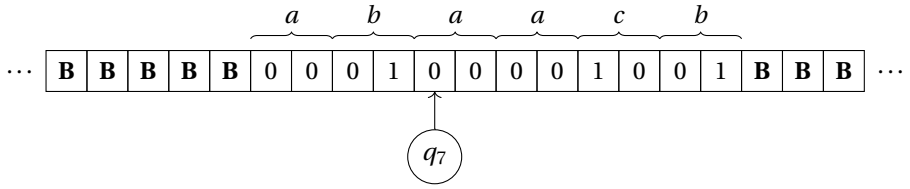


Figure 2: Illustration of the proof of Proposition 1.

Turing machines with several tapes

One can also consider some Turing machines with several tapes, say k, where k is some integer. Each of these k tapes has its own reading head. The machine still have a finite number of internal states Q. Simply, now the transition function δ is not any more a function of $Q \times \Gamma$ to $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$. but from $Q \times \Gamma^k$ to $Q \times \Gamma^k \times \{\leftarrow, |, \rightarrow\}^k$: Depending of the the internal state of the machine and the symbols read by each of the k heads, the transition function gives the new symbols to be written on each of the tapes, and the movements for each of the heads.

It is possible to formalize this model, but we will not do so as this does not really bring a new difficulty.

We will sketch the following result.

Proposition 2 *Every Turing machine with k tapes can be simulated by a Turing machine with a unique tape.*

Sketch of proof: The idea is that if a machine M works with k tapes on the alphabet Γ, one can simulate M by a machine M' with a unique tape that works on alphabet $(\Gamma \times \{0,1\} \cup \{\#\})$ (which is still a finite alphabet).

The tape of M' contains the concatenation of the contents of all the tapes of M, separated by some marker $\#$. We use $(\Gamma \times \{0,1\} \cup \{\#\})$ instead of $(\Gamma \cup \{\#\})$ in order to use 1 more bit of information for every cell to store the information "the read is in front this cell".

M' will simulate step by step the transitions of M: To simulate a transition of M, M' will scan from left to right its tape to determine the position of each of the

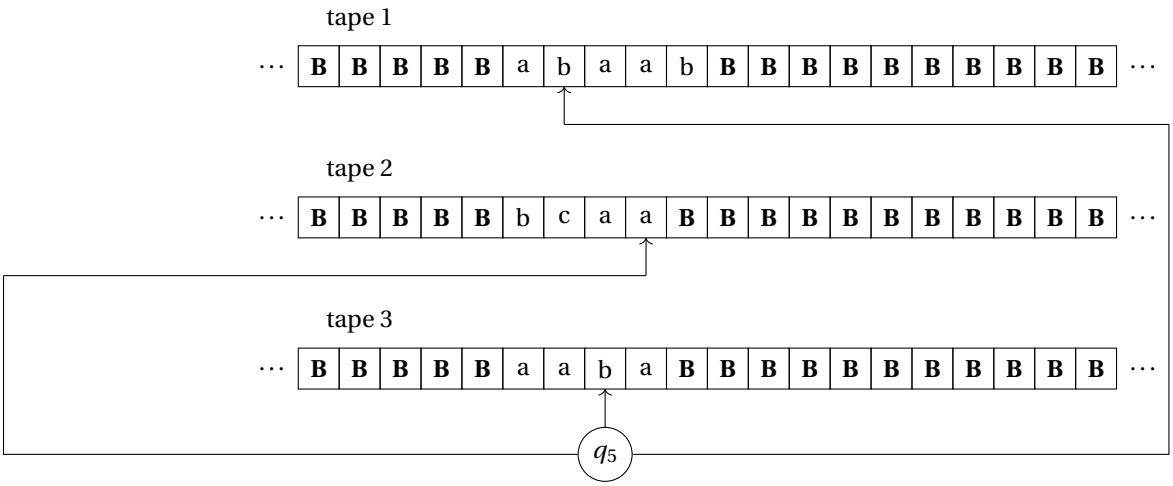


Figure 3: A Turing machine with 3 tapes

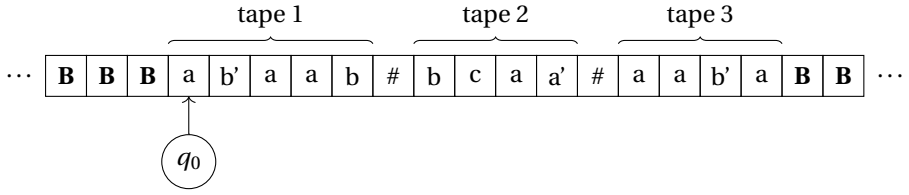


Figure 4: Illustration of the proof of Proposition 2: Graphical representation of a Turing machine with 1 tape simulating the machine with 3 tapes of Figure 3. On this graphical representation, we write a primed letter when the bit "the head is in front of this cell" is set to 1.

reading heads, and the symbol in front of each of the heads (by memorizing these symbols in its internal state). Once all the symbols in front of all the head known, M' knows the symbols to be written and the movements to be done for each of the heads: M' will scan again its tape from left to right to update its encoding of the of the configuration of M. By doing so systematically transition after transition, M' will perfectly simulate the evolution of M with unique tape: See Figure 1.6 □

Non-deterministic Turing machines

We can also introduce non-determinism in Turing machines: The definition of a Turing machine is exactly as the notion of (deterministic) Turing machine except for one point. δ is not a function from $Q \times \Gamma$ to $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$ anymore, but a relation of the form

$$\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}).$$

In other words, to a given internal state and a letter read in front of the head, δ is not defining a unique triple of $Q \times \Gamma \times \{\leftarrow, |, \rightarrow\}$, but possibly a set of triples. Intuitively, during an execution, the machine has the possibility to choose any triple.

Formally, this is expressed by the fact that one can go from configuration C to successor configuration C' if and only one can go from C to C' (we will write this $C \vdash C'$) with previous definitions, but replacing $\delta(q,a) = (q', a', m')$ by $((q,a),(q',a',m')) \in \delta$. The other definitions are unchanged, and the same as for the deterministic Turing machines.

The difference is that a non-deterministic Turing machine does not have a unique execution on some input w but possibly several: Actually, the executions of the machine on a word w give rise to a tree of possibilities, and the idea is that one accepts a word if one of the branches contains an accepting configuration.

The notion of word w accepted is (still) given by Definition 5.

The language $L \subset \Sigma^*$ *accepted by* M is (still) the set of words w that are accepted by the machine. We (still) denote it by $L(M)$. We (still) also call $L(M)$ *the language recognized* by M.

One avoids in general in this context to talk of *rejected word*.

We will however say that a language $L \subset \Sigma^*$ is *decided by* M if it is accepted by a machine that halts on every input: That is to say, for every $w \in L$, the machine has (at least) **ONE** accepting computation that leads to some accepting configuration as in Definition 5, and for every $w \notin L$, **ALL** the computations of the machine lead to some rejecting configuration.

One can prove the following result (we will do so in a following chapter).

Proposition 3 *Any non-deterministic Turing machine can be simulated by a deterministic Turing machine: A language L is accepted by a non-deterministic Turing machine if and only if it is accepted by some (deterministic) Turing machine.*

Obviously, one can consider a Turing machine as a particular non-deterministic Turing machine. The non-trivial direction of the proposition is that one can always simulate a non-deterministic machine by some deterministic one.

In other words, allowing non-determinism does not extend the power of the model, as long as one talks about *computability*, that is to say about the problems that one can solve. We will see that this is not so direct when talking about efficiency, i.e. *complexity*.

1.7 Locality of the notion of computation

We now give a fundamental property of the notion of computation that we will use at several occasions, and that we invite the reader to meditate on:

Proposition 4 (Locality of the notion of computation) *Consider the space-time diagram of a machine M. We consider the possible contents of a subrectangle of width* 3 *and height* 2 *in this diagram. For every machine M there is a finite number of possible contents that one can find in these rectangles. We call the possible contents for the machine M the* legal window *: See Figure 6 for an illustration.*

This even provides a characterization of the space-time diagrams of a given machine: An array is a space-time diagram of M for some initial configuration C_0 if and only if its first line corresponds to C_0, and furthermore in this array, the content of all the subrectangles of width 3 *and height* 2 *are among the legal windows.*

Proof: It is sufficient to look at all possible cases. This is very tedious, but not difficult. □

We will come back to this. Forget this for now, and let us come back for now to other models in next chapter.

Remark 4 *Similar results can also be shown for other models. However, they are often harder to formulate for them.*

2 Bibliographic notes

Suggested readings To go further with all the mentioned notions in this chapter, we suggest to read [Sipser, 1997], [Hopcroft & Ullman, 2000].

Bibliography This chapter has been written using the presentation of Turing machines in [Wolper, 2001], and discussions in [Hopcroft & Ullman, 2000] for the part about their programming. The part on the (S)RAM is inspired by [Papadimitriou, 1994] and [Jones, 1997].

(a)

⋯	**B**	**B**	**B**	**B**	q_0	0	0	1	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	q_1	0	1	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	0	q_1	1	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	q_2	0	Y	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	q_2	X	0	Y	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	q_0	0	Y	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	X	q_1	Y	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	X	Y	q_1	0	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯
⋯	**B**	**B**	**B**	**B**	X	X	Y	0	q_1	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	**B**	⋯

(b)

1	0	B
Y	0	B

Figure 5: (a). The space-time diagram of Example 7. We show one 3×2 subrectangle with gray background. (b) The corresponding (legal) window.

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Figure 6: Some legal windows for a Turing machine M: each of them can be observed on a 3×2 subrectangle of the space-time diagram of M .

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

Figure 7: Some illegal windows for a certain Turing machine M with $\delta(q_1, b) = (q_1, c, \leftarrow)$. One cannot observe these contents in a 3×2 subrectangle of the space-time diagram of M: Indeed, in (a), the middle symbol cannot be changed without the head being next to it. In (b), the symbol at the bottom right should be c but not a, according to the transition function. In (c), there cannot be two heads on the tape.

Index

Bibliography

[Hopcroft & Ullman, 2000] Hopcroft, J. E. & Ullman, J. D. (2000). *Introduction to Automata Theory, Languages and Computation, Second Edition* (2nd ed.). Addison-Wesley.

[Jones, 1997] Jones, N. D. (1997). *Computability and complexity - from a programming perspective*. Foundations of computing series. MIT Press. https://doi.org/10.7551/mitpress/2003.001.0001

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Sipser, 1997] Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing Company.

[Wolper, 2001] Wolper, P. (2001). *Introduction à la calculabilité: cours et exercices corrigés*. Dunod.