Algorithms on Strings

Maxime Crochemore
Christophe Hancart
Thierry Lecroq

Algorithms on Strings

Cambridge University Press

Algorithms on Strings – Maxime Crochemore, Christophe Hancart et Thierry Lecroq

Table of contents

Preface

This book presents a broad panorama of the algorithmic methods used for processing texts. For this reason it is a book on algorithms, but whose object is focused on the handling of texts by computers. The idea of this publication results from the observation that the rare books entirely devoted to the subject are primarily monographs of research. This is surprising because the problems of the field are known since the development of advanced operating systems, and the need for effective solutions becomes crucial since the massive use of data processing in office automation is crucial in many sectors of the society.

In a written or vocal form, text is the only reliable vehicle of abstract concepts. Therefore it remains the privileged support of information systems, despite of significant efforts towards the use of other media (graphic interfaces, systems of virtual reality, synthesis movies, etc). This aspect is still reinforced by the use of knowledge databases, legal, commercial or others, which develop on the Internet thanks in particular to the Web services.

The content of the book carry over into formal elements and technical bases required in the fields of Information retrieval, of automatic indexing for search engines, and more generally of software systems, which includes the edition, the treatment and the compression of texts. The methods that are described apply to the automatic processing of natural languages, to the treatment and analysis of genomic sequences, to the analysis of musical sequences, to problems of safety and security related to data flows, and to the management of the textual databases, to quote only some immediate applications.

The selected subjects address pattern matching, the indexing of textual data, the comparison of texts by alignment and the search for local regularities. In addition to their practical interest, these subjects have theoretical and combinatorial aspects which provide astonishing examples of algorithmic solutions.

The goal of this work is principally educational. It is initially aimed at graduate and undergraduate students. But it can also be used by

software designers.

We warmly thank the researchers who took time to read and comment on the preliminary outlines of this book. They are: Saïd Abdeddaïm, Marie-Pierre Béal, Christian Charras, Sabine Mercier, Laurent Mouchard, Gregory Kucherov, Johann Pelfrêne, Bruno Petazzoni, Mathieu Raffinot, Giuseppina Rindone, Marie-France Sagot. Remaining errors are ours.

Finally, extra elements to the contents of the book are accessible on the site `http://chl.univ-mlv.fr/` or from the web pages of the authors.

Maxime Crochemore
Christophe Hancart
Thierry Lecroq
London and Rouen,
April 2006

1 Tools

This chapter presents the algorithmic and combinatorial framework in which are developed the following chapters. It first specifies the concepts and notation used to work on strings, languages and automata. The rest is mainly devoted to the introduction of chosen data structures for implementing automata, to the presentation of combinatorial results, and to the design of elementary pattern matching techniques. This organization is based on the observation that efficient algorithms for text processing rely on one or the other of these aspects.

Section 1.2 provides some combinatorial properties of strings that occur in numerous correctness proofs of algorithms or in their performance evaluation. They are mainly periodicity results.

The formalism for the description of algorithms is presented in Section 1.3, which is especially centered on the type of algorithm presented in the book, and introduces some standard objects related to queues and automata processing.

Section 1.4 details several methods to implement automata in memory, these techniques contribute in particular to results of Chapters 2, 5 and 6.

The first algorithms for locating strings in texts are presented in Section 1.5. The sliding window mechanism, the notions of search automaton and of bit vectors that are described in this section are also used and improved in Chapters 2, 3 and 8, in particular.

Section 1.6 is the algorithmic jewel of the chapter. It presents two fundamental algorithmic methods used for text processing. They are used to compute the border table and the prefix table of a string that constitute two essential tables for string processing. They synthesize a part of the combinatorial properties of a string. Their utilization and adaptation is considered in Chapters 2 and 3, and also punctually come back in other chapters.

Finally, we can note that intuition for combinatorial properties or algorithms sometimes relies on figures whose style is introduced in this chapter and kept thereafter.

1.1 Strings and automata

In this section we introduce notation on strings, languages and automata.

Alphabet and strings

An ***alphabet*** is a finite non-empty set whose elements are called ***letters***. A ***string*** on an alphabet A is a finite sequence of elements of A. The zero letter sequence is called the ***empty string*** and is denoted by ε. For the sake of simplification, delimiters and separators usually employed in sequence notation are removed and a string is written as the simple juxtaposition of the letters that compose it. Thus, ε, `a`, `b` and `baba` are strings on any alphabet that contains the two letters `a` and `b`. The set of all the strings on the alphabet A is denoted by A^*, and the set of all the strings on the alphabet A except the empty string ε is denoted by A^+.

The ***length*** of a string x is defined as the length of the sequence associated with the string x and is denoted by $|x|$. We denote by $x[i]$, for $i = 0, 1, \ldots, |x| - 1$, the letter at index i of x with the convention that indices begin with 0. When $x \neq \varepsilon$, we say more specifically that each index $i = 0, 1, \ldots, |x| - 1$ is a ***position* on** x. It follows that the i-th letter of x is the letter at position $i - 1$ on x and that:

$$x = x[0]x[1] \ldots x[|x| - 1] \ .$$

Thus an elementary definition of the identity between any two strings x and y is:

$$x = y$$

if and only if

$$|x| = |y| \text{ and } x[i] = y[i] \text{ for } i = 0, 1, \ldots, |x| - 1 \ .$$

The set of letters that occur in the string x is denoted by $alph(x)$. For instance, if $x = \texttt{abaaab}$, we have $|x| = 6$ and $alph(x) = \{\texttt{a}, \texttt{b}\}$.

The ***product*** – we also say the ***concatenation*** – of two strings x and y is the string composed of the letters of x followed by the letters of y. It is denoted by xy or also $x \cdot y$ to show the decomposition of the resulting string. The neutral element for the product is ε. For every string x and every natural number n, we define the n-th ***power*** of the string x, denoted by x^n, by $x^0 = \varepsilon$ and $x^k = x^{k-1}x$ for $k = 1, 2, \ldots, n$. We denote respectively by zy^{-1} and $x^{-1}z$ the strings x and y when $z = xy$. The ***reverse*** – or ***mirror image*** – of the string x is the string $x^\sim$ defined by:

$$x^\sim = x[|x| - 1]x[|x| - 2] \ldots x[0] \ .$$

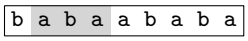


Figure 1.1 An occurrence of string `aba` in string `babaababa` at (left) position 1.

A string x is a ***factor*** of a string y if there exist two strings u and v such that $y = uxv$. When $u = \varepsilon$, x is a ***prefix*** of y; and when $v = \varepsilon$, x is a ***suffix*** of y. The string x is a ***subsequence***[1] of y if there exist $|x| + 1$ strings $w_0, w_1, \ldots, w_{|x|}$ such that $y = w_0x[0]w_1x[1]\ldots x[|x|-1]w_{|x|}$; in a less formal way, x is a string obtained from y by deleting $|y| - |x|$ letters. A factor or a subsequence x of a string y is ***proper*** if $x \neq y$. We denote respectively by $x \preceq_{fact} y$, $x \prec_{fact} y$, $x \preceq_{pref} y$, $x \prec_{pref} y$, $x \preceq_{suff} y$, $x \prec_{suff} y$, $x \preceq_{sseq} y$ and $x \prec_{sseq} y$ when x is a factor, a proper factor, a prefix, a proper prefix, a suffix, a proper suffix, a subsequence and a proper subsequence of y. One can verify that $\preceq_{fact}$, $\preceq_{pref}$, $\preceq_{suff}$ and $\preceq_{sseq}$ are orderings on A^*.

The ***lexicographic order***, denoted by $\leq$, is an order on the strings induced by an order on the letters and denoted by the same symbol. It is defined as follows. For $x, y \in A^*$, $x \leq y$ if and only if, either $x \preceq_{pref} y$, or x and y can be decomposed as $x = uav$ and $y = ubw$ with $u, v, w \in A^*$, $a, b \in A$ and $a < b$. Thus, `ababb` $<$ `abba` $<$ `abbaab` assuming `a` $<$ `b`.

Let x be a non-empty string and y be a string, we say that there is an ***occurrence*** of x in y, or, more simply, that x ***occurs in*** y, when x is a factor of y. Every occurrence, or every appearance, of x can be characterized by a position on y. Thus we say that an occurrence of x ***starts*** at the ***left position*** i on y when $y[i\,..\,i + |x| - 1] = x$ (see Figure 1.1). It is sometimes more suitable to consider the ***right position*** $i + |x| - 1$ at which this occurrence ***ends***. For instance, the left and right positions where the string $x =$ `aba` occurs in the string $y =$ `babaababa` are:

i	0	1	2	3	4	5	6	7	8
$y[i]$	`b`	`a`	`b`	`a`	`a`	`b`	`a`	`b`	`a`
left positions		1			4		6		
right positions				3			6		8

The ***position of the first occurrence*** $pos(x)$ of x in y is the minimal (left) position at which starts the occurrence of x in yA^*. With the notation on the languages recalled thereafter, we have:

$$pos(x) = \min\{|u| : \{ux\}A^* \cap \{y\}A^* \neq \emptyset\} \ .$$

The square bracket notation for the letters of strings is extended to

[1] We avoid the common use of "subword" because it has two definitions in literature: one of them is factor and the other is subsequence.

factors. We define the factor $x[i\,..\,j]$ of the string x by:

$$x[i\,..\,j] = x[i]x[i+1]\ldots x[j]$$

for all integers i and j satisfying $0 \leq i \leq |x|$, $-1 \leq j \leq |x|-1$ and $i \leq j+1$. When $i = j+1$, the string $x[i\,..\,j]$ is the empty string.

Languages

Any subset of A^* is a ***language*** on the alphabet A. The product defined on strings is extended to languages as follows

$$XY = X \cdot Y = \{xy : (x,y) \in X \times Y\}$$

for every languages X and Y. We extend as well the notion of power as follows $X^0 = \{\varepsilon\}$ and $X^k = X^{k-1}X$ for $k \geq 1$. The ***star*** of X is the language:

$$X^* = \bigcup_{n \geq 0} X^n \ .$$

We denote by X^+ the language defined by:

$$X^+ = \bigcup_{n \geq 1} X^n \ .$$

Note that these two latter notation are compatible with the notation A^* and A^+. In order not to overload the notation, a language that is reduced to a single string can be named by the string itself if it does not lead to any confusion. For instance, the expression $A^*\mathtt{abaaab}$ denotes the language of the strings in A^* having the string `abaaab` as suffix, assuming $\{\mathtt{a},\mathtt{b}\} \subseteq A$.

The notion of length is extended to languages as follows:

$$|X| = \sum_{x \in X} |x| \ .$$

In the same way, we define $alph(X)$ by

$$alph(X) = \bigcup_{x \in X} alph(x)$$

and $X^{\sim}$ by

$$X^{\sim} = \{x^{\sim} : x \in X\} \ .$$

The sets of factors, prefixes, suffixes and subsequences of the strings of a language X are particular languages that are often considered in the rest of the book; they are respectively denoted by $Fact(X)$, $Pref(X)$, $Suff(X)$ and $Subs(X)$.

The ***right context*** of a string y relatively to a language X is the language:

$$y^{-1}X = \{y^{-1}x : x \in X\} \ .$$

The equivalence relation defined by the identity of right contexts is denoted by $\equiv_X$, or simply[2] $\equiv$. Thus

$$y \equiv z \text{ if and only if } y^{-1}X = z^{-1}X$$

for $y, z \in A^*$. For instance, when $A = \{\mathtt{a}, \mathtt{b}\}$ and $X = A^*\{\mathtt{aba}\}$, the relation $\equiv$ admits four equivalence classes: $\{\varepsilon, \mathtt{b}\} \cup A^*\{\mathtt{bb}\}$, $\{\mathtt{a}\} \cup A^*\{\mathtt{aa}, \mathtt{bba}\}$, $A^*\{\mathtt{ab}\}$ and $A^*\{\mathtt{aba}\}$. For every language X, the relation $\equiv$ is an equivalence relation that is compatible with the concatenation. It is called the ***right syntactic congruence*** associated with X.

Regular expressions and languages

The ***regular expressions*** on an alphabet A and the languages they describe, the ***regular languages***, are recursively defined as follows:

- `0` and `1` are regular expressions that respectively describe $\emptyset$ (the empty set) and $\{\varepsilon\}$;
- for every letter $a \in A$, $\boldsymbol{a}$ is a regular expression that describes the singleton $\{a\}$;
- if $\boldsymbol{x}$ and $\boldsymbol{y}$ are regular expressions respectively describing the regular languages X and Y, then `(`$\boldsymbol{x}$`)+(`$\boldsymbol{y}$`)`, `(`$\boldsymbol{x}$`)·(`$\boldsymbol{y}$`)` and `(`$\boldsymbol{x}$`)*` are regular expressions that respectively describe the regular languages $X \cup Y$, $X \cdot Y$ and X^*.

The priority order of operations on the regular expressions is `*`, $\cdot$, then `+`. Possible writing simplifications allow to omit the symbol $\cdot$ and some parentheses pairs. The language described by a regular expression $\boldsymbol{x}$ is denoted by $\mathit{Lang}(\boldsymbol{x})$.

Automata

An ***automaton*** M on the alphabet A is composed of a finite set Q of ***states***, of an ***initial*** state[3] q_0, of a set $T \subseteq Q$ of ***terminal*** states and of a set $F \subseteq Q \times A \times Q$ of ***arcs*** – or ***transitions***. We denote the automaton M by the quadruplet:

$$(Q, q_0, T, F) \ .$$

[2] As in all the rest of the book, the notation is indexed by the object to which they refer only when it could be ambiguous.

[3] The standard definition of automata considers a set of initial states rather than a single initial state as we do in the entire book. We leave the reader to convince himself that it is possible to build a correspondence between any automaton defined in the standard way and an automaton with a single initial state that recognizes the same language.

We say of an arc (p, a, q) that it leaves the state p and that it enters the state q; state p is the ***source*** of the arc, letter a its ***label*** and state q its ***target***. The number of arcs outgoing a given state is called the ***outgoing degree*** of the state. The ***incoming degree*** of a state is defined in a dual way. By analogy with graphs, the state q is a ***successor*** by the letter a of the state p when $(p, a, q) \in F$; in the same case, we say that the pair (a, q) is a ***labeled successor*** of the state p.

A ***path*** of length n in the automaton $M = (Q, q_0, T, F)$ is a sequence of n consecutive arcs

$$\langle (p_0, a_0, p'_0), (p_1, a_1, p'_1), \ldots, (p_{n-1}, a_{n-1}, p'_{n-1}) \rangle \ ,$$

that satisfies

$$p'_k = p_{k+1}$$

for $k = 0, 1, \ldots, n-2$. The ***label*** of the path is the string $a_0 a_1 \ldots a_{n-1}$, its ***origin*** the state p_0, its ***end*** the state p'_{n-1}. By convention, there exists for each state p a path of null length of origin and of end p; the label of such a path is ε, the empty string. A path in the automaton M is ***successful*** if its origin is the initial state q_0 and if its end is in T. A string is ***recognized*** – or ***accepted*** – by the automaton if it is the label of a successful path. The language composed of the strings recognized by the automaton M is denoted by $Lang(M)$.

Often, more than its formal notation, a diagram illustrates how an automaton works. We represent the states by circles and the arcs by directed arrows from source to target, labeled by the corresponding letter. When several arcs have the same source and the same target, we merge the arcs and the label of the resulting arc becomes an enumeration of the letters. The initial state is distinguished by a short incoming arrow and the terminal states are double circled. An example is shown Figure 1.2.

A state p of an automaton $M = (Q, q_0, T, F)$ is ***accessible*** if there exists a path in M starting at q_0 and ending in p. A state p is ***co-accessible*** if there exists a path in M starting at p and ending in T.

An automaton $M = (Q, q_0, T, F)$ is ***deterministic*** if for every pair $(p, a) \in Q \times A$ there exists at most one state $q \in Q$ such that $(p, a, q) \in F$. In such a case, it is natural to consider the ***transition function***

$$\delta : Q \times A \to Q$$

of the automaton defined for every arc $(p, a, q) \in F$ by

$$\delta(p, a) = q$$

and not defined elsewhere. The function δ is easily extended to strings. It is enough to consider its extension $\bar{\delta} : Q \times A^* \to Q$ recursively defined by $\bar{\delta}(p, \varepsilon) = p$ and $\bar{\delta}(p, wa) = \delta(\bar{\delta}(p, w), a)$ for $p \in Q$, $w \in A^*$ and $a \in A$. It follows that the string w is recognized by the automaton M if and

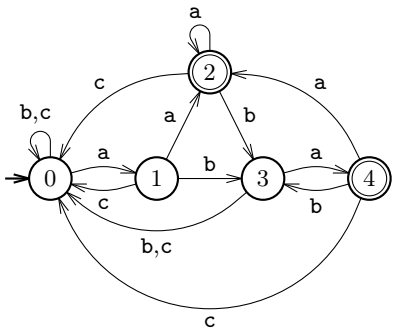


Figure 1.2 Representation of an automaton on the alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. The states of the automaton are numbered from 0 to 4, its initial state is 0 and its terminal states are 2 and 4. The automaton possesses $3 \times 5 = 15$ arcs. The language that it recognizes is described by the regular expression `(a+b+c)*(aa+aba)`, i.e. the set of strings on the three letter alphabet `a`, `b` and `c` ending by `aa` or `aba`.

only if $\bar{\delta}(q_0, w) \in T$. Generally, the function δ and its extension $\bar{\delta}$ are denoted in the same way.

The automaton $M = (Q, q_0, T, F)$ is ***complete*** when for every pair $(p, a) \in Q \times A$ there exists at least one state $q \in Q$ such that $(p, a, q) \in F$.

Proposition 1.1
For every automaton, there exists a deterministic and complete automaton that recognizes the same language. ∎

To complete an automaton is not difficult: it is enough to add to the automaton a ***sink*** state, then to make it the target of all undefined transitions. It is a bit more difficult to ***determinize*** an automaton, that is, to transform an automaton $M = (Q, q_0, T, F)$ into a deterministic automaton recognizing the same language. One can use the so-called method of ***construction by subsets***: let M' be the automaton whose states are the subsets of Q, the initial state is the singleton $\{q_0\}$, the terminal states are the subsets of Q that intersect T, and the arcs are the triplets (U, a, V) where V is the set of successors by the letter a of the states p belonging to U; then M' is a deterministic automaton that recognizes the same language as M. In practical applications, we do not construct the automaton M' entirely, but only its accessible part from the initial state $\{q_0\}$.

A language X is ***recognizable*** if there exists an automaton M such that $X = \mathit{Lang}(M)$. The statement of a fundamental theorem of automata theory that establishes the link between recognizable languages and regular languages on a given alphabet follows.

Theorem 1.2 (Kleene theorem)

A language is recognizable if and only if it is regular. ∎

If X is a recognizable language, the ***minimal automaton*** of X, denoted by $\mathcal{M}(X)$, is determined by the right syntactic congruence associated to X. It is the automaton whose set of states is $\{w^{-1}X : w \in A^*\}$, the initial state is X, the set of terminal states is $\{w^{-1}X : w \in X\}$, and the set of arcs is $\{(w^{-1}X, a, (wa)^{-1}X) : (w, a) \in A^* \times A\}$.

Proposition 1.3
The minimal automaton $\mathcal{M}(X)$ of a language X is the automaton having the smallest number of states among the deterministic and complete automata that recognize the language X. The automaton $\mathcal{M}(X)$ is the homomorphic image of every automaton recognizing X. ∎

We often say of an automaton that it is minimal though it is not complete. Actually, this automaton is indeed minimal if one takes care to add a sink state.

Each state of an automaton, or even sometimes each arc, can be associated with an ***output***. It is a value or a set of values associated with the state or the arc.

1.2 Some combinatorics

We consider the notion of periodicity on strings for which we give the basic properties. We begin with presenting two families of strings that have interesting combinatorial properties with regard to questions of periodicities and repeats examined in several chapters.

Some specific strings

Fibonacci numbers are defined by the recurrence:

$$\begin{aligned} F_0 &= 0 \ , \\ F_1 &= 1 \ , \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{for } n \geq 2 \ . \end{aligned}$$

These famous numbers satisfy properties all more remarkable than the others. Among those, we just give two:

- for every natural number $n \geq 2$, $\gcd(F_n, F_{n-1}) = 1$,
- for every natural number n, F_n is the nearest integer of $\Phi^n/\sqrt{5}$, where $\Phi = \frac{1}{2}(1 + \sqrt{5}) = 1{,}61803\ldots$ is the ***golden ratio***.

Fibonacci strings are defined on the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ by the following recurrence:

$$f_0 = \varepsilon \ ,$$

$$
\begin{aligned}
f_1 &= \texttt{b}\ , \\
f_2 &= \texttt{a}\ , \\
f_n &= f_{n-1}f_{n-2} \qquad \text{for } n \geq 3\ .
\end{aligned}
$$

Note that the sequence of lengths of the strings is exactly the sequence of Fibonacci numbers, that is, $F_n = |f_n|$. Here are the first ten Fibonacci numbers and strings:

n	F_n	f_n
0	0	ε
1	1	`b`
2	1	`a`
3	2	`ab`
4	3	`aba`
5	5	`abaab`
6	8	`abaababa`
7	13	`abaababaabaab`
8	21	`abaababaabaababaababa`
9	34	`abaababaabaababaababaabaababaabaab`

The interest in Fibonacci strings is that they satisfy many combinatorial properties and they contain a large number of repeats.

The de Bruijn strings considered here are defined on the alphabet $A = \{\texttt{a}, \texttt{b}\}$ and are parameterized by a non-null natural number. A non-empty string $x \in A^+$ is a ***de Bruijn string*** of order k if each string on A of length k occurs once and only once in x. A first example: `ab` and `ba` are the only two de Bruijn strings of order 1. A second example: the string `aaababbbaa` is a de Bruijn string of order 3 since its factors of length 3 are the eight strings of A^3, that is, `aaa`, `aab`, `aba`, `abb`, `baa`, `bab`, `bba` and `bbb`, and each of them occurs exactly once in it.

The existence of a de Bruijn string of order $k \geq 2$ can be verified with the help of the ***automaton*** defined by:

- states are the strings of the language A^{k-1},
- arcs are of the form (av, b, vb) with $a, b \in A$ and $v \in A^{k-2}$,

the initial state and the terminal states are not given (an illustration is shown Figure 1.3). We note that exactly two arcs exit each of the states, one labeled by `a`, the other by `b`; and that exactly two arcs enter each of the states, both labeled by the same letter. The graph associated with the automaton thus satisfies the Euler condition: the outgoing degree and the incoming degree of each state are identical. It follows that there exists an Eulerian circuit in the graph. Now, let

$$\langle (u_0, a_0, u_1), (u_1, a_1, u_2), \ldots, (u_{n-1}, a_{n-1}, u_0) \rangle$$

be the corresponding path. The string $u_0 a_0 a_1 \ldots a_{n-1}$ is a de Bruijn string of order k, since each arc of the path is identified with a factor

of length k. It follows in the same way that a de Bruijn string of order k has length $2^k + k - 1$ (thus $n = 2^k$ with the previous notation). It can also be verified that the number of de Bruijn strings of order k is exponential in k.

The de Bruijn strings are often used as examples of limit cases in the sense that they contain all the factors of a given length.

Periodicity and borders

Let x be a non-empty string. An integer p such that $0 < p \leq |x|$ is called a ***period*** of x if:

$$x[i] = x[i + p]$$

for $i = 0, 1, \ldots, |x| - p - 1$. Note that the length of a non-empty string is a period of this string, such that every non-empty string has at least one period. We define thus without any ambiguity ***the period*** of a non-empty string x as the smallest of its periods. It is denoted by $per(x)$. For instance, 3, 6, 7 and 8 are periods of the string $x = \texttt{aabaabaa}$ and the period of x is $per(x) = 3$.

We note that if p is a period of x, its multiples kp are also periods of x when k is an integer satisfying $0 < k \leq \lfloor |x|/p \rfloor$.

Proposition 1.4
Let x be a non-empty string and p an integer such that $0 < p \leq |x|$. Then the five following properties are equivalent:

1. *The integer p is a period of x.*
2. *There exist two unique strings $u \in A^*$ and $v \in A^+$ and an integer $k > 0$ such that $x = (uv)^k u$ and $|uv| = p$.*
3. *There exist a string t and an integer $k > 0$ such that $x \preceq_{pref} t^k$ and $|t| = p$.*
4. *There exist three strings u, v and w such that $x = uw = wv$ and $|u| = |v| = p$.*
5. *There exists a string t such that $x \preceq_{pref} tx$ and $|t| = p$.*

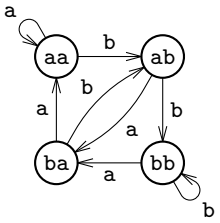


Figure 1.3 The order 3 de Bruijn automaton on the alphabet $\{\texttt{a}, \texttt{b}\}$. The initial state of the automaton is not given.

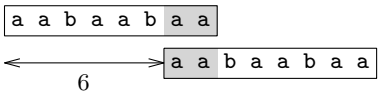


Figure 1.4 Duality between the notions of borders and periods. String `aa` is a border of string `aabaabaa`; it corresponds to period $6 = |\texttt{aabaabaa}| - |\texttt{aa}|$.

Proof $1 \Rightarrow 2$: if $v \neq \varepsilon$ and $k > 0$, then k is the quotient of the integer division of $|x|$ by p. Now, if the triplet (u', v', k') satisfies the same conditions than the triplet (u, v, k), we have $k' = k$ then, due to the equality of length, $|u'| = |u|$. It follows immediately that $u' = u$ and $v' = v$. This shows the uniqueness of the decomposition if it exists. Let k and r be respectively the quotient and the remainder of the Euclidean division of $|x|$ by p, then u and v be the two factors of x defined by $u = x[0\,..\,r-1]$ and $v = x[r\,..\,p-1]$. Thus $x = (uv)^k u$ and $|uv| = p$. This demonstrates the existence of the triplet (u, v, k) and ends the proof of the property.

$2 \Rightarrow 3$: it is enough to consider the string $t = uv$.

$3 \Rightarrow 4$: let w be the suffix of x defined by $w = t^{-1}x$. As $x \preceq_{pref} t^k$, w is also a prefix of x. Thus the existence of two strings u $(= t)$ and v such that $x = uw = wv$ and $|u| = |v| = |t| = p$.

$4 \Rightarrow 5$: since $uw \preceq_{pref} uwv$, we have $x \preceq_{pref} tx$ with $|t| = p$ by simply setting $t = u$.

$5 \Rightarrow 1$: let i be an integer such that $0 \leq i \leq |x| - p - 1$. Then:

$$\begin{aligned} x[i+p] &= (tx)[i+p] && (\text{since } x \preceq_{pref} tx) \\ &= x[i] && (\text{since } |t| = p)\ . \end{aligned}$$

This shows that p is a period of x. ∎

We note in particular that Property 3 can be expressed in a more general way by replacing $\preceq_{pref}$ by $\preceq_{fact}$ (Exercise 1.4).

A ***border*** of a non-empty string x is a proper factor of x that is both a prefix and a suffix of x. Thus, ε, `a`, `aa` and `aabaa` are the borders of the string `aabaabaa`.

The notions of borders and of periods are dual as shown by Property 4 of the previous proposition (see Figure 1.4). The proposition that follows expresses this duality in different terms.

We introduce the function $Border\colon A^* \to A^*$ defined for every non-empty string x by:

$Border(x) =$ the longest border of x .

We say of $Border(x)$ that it is ***the border*** of x. For instance, the border of every string of length 1 is the empty string and the border of the string `aabaabaa` is `aabaa`. Also note that, when defined, the border of a border of a given string x is also a border of x.

Proposition 1.5
Let x be a non-empty string and n be the largest integer k for which $Border^k(x)$ is defined (thus $Border^n(x) = \varepsilon$). Then

$$\langle Border(x), Border^2(x), \ldots, Border^n(x)\rangle \tag{1.1}$$

is the sequence of borders of x in decreasing order of length, and

$$\langle |x| - |Border(x)|, |x| - |Border^2(x)|, \ldots, |x| - |Border^n(x)|\rangle \tag{1.2}$$

is the sequence of periods of x in increasing order.

Proof We proceed by recurrence on the length of strings. The statement of the proposition is valid when the length of the string x is equal to 1: the sequence of borders is reduced to $\langle\varepsilon\rangle$ and the sequence of periods to $\langle|x|\rangle$.

Let x be a string of length greater than 1. Then every border of x different from $Border(x)$ is a border of $Border(x)$, and conversely. It follows by recurrence hypothesis that the sequence (1.1) is exactly the sequence of borders of x. Now, if p is a period of x, Proposition 1.4 ensures the existence of three strings u, v and w such that $x = uw = wv$ and $|u| = |v| = p$. Then w is a border of x and $p = |x| - |w|$. It follows that the sequence (1.2) is the sequence of periods of x. ∎

Lemma 1.6 (Periodicity lemma)
If p and q are periods of a non-empty string x and satisfy

$$p + q - \gcd(p, q) \leq |x| \ ,$$

then $\gcd(p, q)$ is also a period of x.

Proof By recurrence on $\max\{p, q\}$. The result is straightforward when $p = q = 1$ and, more generally when $p = q$. We can then assume in the rest that $p > q$.

From Proposition 1.4, the string x can be written both as uy with $|u| = p$ and y a border of x, and as vz with $|v| = q$ and z a border of x.

The quantity $p-q$ is a period of z. Indeed, since $p > q$, y is a border of x of length strictly less than the length of the border z. Thus y is a border of z. It follows that $|z| - |y|$ is a period of z. And $|z| - |y| = (|x| - q) - (|x| - p) = p - q$.

But q is also a period of z. Indeed, since $p > q$ and $\gcd(p, q) \leq p - q$, we have $q \leq p - \gcd(p, q)$. One the other hand we have $p - \gcd(p, q) = p + q - \gcd(p, q) - q \leq |x| - q = |z|$. It follows that $q \leq |z|$. This shows that the period q of x is also a period of its factor z.

Moreover, we have $(p - q) + q - \gcd(p - q, q) = p - \gcd(p, q)$, which, as can be seen above, is a quantity less than $|z|$.

We apply the recurrence hypothesis to $\max\{p-q, q\}$ relatively to the string z, and we obtain thus that $\gcd(p, q)$ is a period of z.

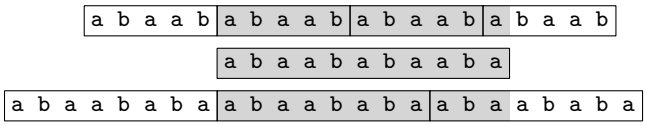


Figure 1.5 Application of the Periodicity lemma. String `abaababaaba` of length 11 possesses 5 and 8 as periods. It is not possible to extend them to the left nor to the right while keeping these two periods. Indeed, if 5 and 8 are periods of some string, but 1 – the greatest common divisor of 5 and 8 – is not, then this string is of length less than $5 + 8 - \gcd(5, 8) = 12$.

The conditions on p and q (those of the lemma and $\gcd(p, q) \leq p - q$) give $q \leq |x|/2$. And as $x = vz$ and z is a border of x, v is a prefix of z. It has moreover a length that is a multiple of $\gcd(p, q)$. Let t be the prefix of x of length $\gcd(p, q)$. Then v is a power of t and z is a prefix of a power of t. It follows then by Proposition 1.4 that x is a prefix of a power of t, and thus that $|t| = \gcd(p, q)$ is a period of x. Which ends the proof. ∎

To illustrate the periodicity lemma, let us consider a string x that admits both 5 and 8 as periods. Then, if we assume moreover that x is composed of at least two distinct letters, $\gcd(5, 8) = 1$ is not a period of x, and, by application of the lemma, the length of x is strictly less than $5 + 8 - \gcd(5, 8) = 12$. It is the case, for instance, for the four strings of length greater than 7 which are prefixes of the string `abaababaaba` of length 11. Another illustration of the result is proposed in Figure 1.5.

We wish to show in what follows that one cannot weaken the condition required on the periods in the statement of the periodicity lemma. More precisely, we give examples of strings x that have two periods p and q such that $p + q - \gcd(p, q) = |x| + 1$ but which do not satisfy the conclusion of the lemma. (See also Exercise 1.5.)

Let $\beta\colon A^* \to A^*$ be the function defined by

$$\beta(uab) = uba$$

for every string $u \in A^*$ and every letters $a, b \in A$.

Lemma 1.7
For every natural number $n \geq 3$, $\beta(f_n) = f_{n-2}f_{n-1}$.

Proof By recurrence on n. The result is straightforward when $3 \leq n \leq 4$. If $n \geq 5$, we have:

$$\begin{aligned}
\beta(f_n) &= \beta(f_{n-1}f_{n-2}) && \text{(by definition of } f_n\text{)} \\
&= f_{n-1}\beta(f_{n-2}) && \text{(since } |f_{n-2}| = F_{n-2} \geq 2\text{)} \\
&= f_{n-1}f_{n-4}f_{n-3} && \text{(by recurrence hypothesis)} \\
&= f_{n-2}f_{n-3}f_{n-4}f_{n-3} && \text{(by definition of } f_{n-1}\text{)}
\end{aligned}$$

$$\begin{aligned} &= f_{n-2}f_{n-2}f_{n-3} && \text{(by definition of } f_{n-2}\text{)} \\ &= f_{n-2}f_{n-1} && \text{(by definition of } f_{n-1}\text{) .} \end{aligned}$$ ■

For every natural number $n \geq 3$, we define the string g_n as the prefix of length $F_n - 2$ of f_n, that is, f_n with its last two letters chopped off.

Lemma 1.8
For every natural number $n \geq 6$, $g_n = {f_{n-2}}^2 g_{n-3}$.

Proof We have:

$$\begin{aligned} f_n &= f_{n-1}f_{n-2} && \text{(by definition of } f_n\text{)} \\ &= f_{n-2}f_{n-3}f_{n-2} && \text{(by definition of } f_{n-1}\text{)} \\ &= f_{n-2}\beta(f_{n-1}) && \text{(from Lemma 1.7)} \\ &= f_{n-2}\beta(f_{n-2}f_{n-3}) && \text{(by definition of } f_{n-1}\text{)} \\ &= {f_{n-2}}^2\beta(f_{n-3}) && \text{(since } |f_{n-3}| = F_{n-3} \geq 2\text{) .} \end{aligned}$$

The stated result immediately follows. ■

Lemma 1.9
For every natural number $n \geq 3$, $g_n \preceq_{pref} {f_{n-1}}^2$ and $g_n \preceq_{pref} {f_{n-2}}^3$.

Proof We have:

$$\begin{aligned} g_n &\preceq_{pref} f_n f_{n-3} && \text{(since } g_n \preceq_{pref} f_n\text{)} \\ &= f_{n-1}f_{n-2}f_{n-3} && \text{(by definition of } f_n\text{)} \\ &= {f_{n-1}}^2 && \text{(by definition of } f_{n-1}\text{) .} \end{aligned}$$

The second relation is valid when $3 \leq n \leq 5$. When $n \geq 6$, we have:

$$\begin{aligned} g_n &= {f_{n-2}}^2 g_{n-3} && \text{(from Lemma 1.8)} \\ &\preceq_{pref} {f_{n-2}}^2 f_{n-3}f_{n-4} && \text{(since } g_{n-3} \preceq_{pref} f_{n-3}\text{)} \\ &= {f_{n-2}}^3 && \text{(by definition of } f_{n-2}\text{) .} \end{aligned}$$ ■

Now, let n be a natural number, $n \geq 5$, so that the string g_n is both defined and of length greater than 2. It follows then:

$$\begin{aligned} |g_n| &= F_n - 2 && \text{(by definition of } g_n\text{)} \\ &= F_{n-1} + F_{n-2} - 2 && \text{(by definition of } F_n\text{)} \\ &\geq F_{n-1} && \text{(since } F_{n-2} \geq 2\text{) .} \end{aligned}$$

It results from this inequality, from Lemma 1.9 and from Proposition 1.4 that F_{n-1} and F_{n-2} are two periods of g_n. In addition note that, since $\gcd(F_{n-1}, F_{n-2}) = 1$, we also have:

$$\begin{aligned} F_{n-1} + F_{n-2} - \gcd(F_{n-1}, F_{n-2}) &= F_n - 1 \\ &= |g_n| + 1 \text{ .} \end{aligned}$$

Thus, if the conclusion of the periodicity lemma applied to the string g_n and its two periods F_{n-1} and F_{n-2}, g_n would be the power of a string of length 1. But the first two letters of g_n are distinct. This indicates that the condition of the periodicity lemma is in some sense optimal.

Powers, primitivity and conjugacy

Lemma 1.10
Let x and y be two strings. If there exist two natural non-null numbers m and n such that $x^m = y^n$, x and y are powers of some string z.

Proof It is enough to show the result in the non-trivial case where neither x nor y are empty strings. Two sub-cases can then be distinguished, whether $\min\{m, n\}$ is equal to 1 or not.

If $\min\{m, n\} = 1$, it is sufficient to consider the string $z = y$ if $m = 1$ and $z = x$ if $n = 1$.

Otherwise, $\min\{m, n\} \geq 2$. Then we note that $|x|$ and $|y|$ are periods of the string $t = x^m = y^n$ which satisfy the condition of the periodicity lemma: $|x| + |y| - \gcd(|x|, |y|) \leq |x| + |y| - 1 < |t|$. Thus it is sufficient to consider the string z defined as the prefix of t of length $\gcd(|x|, |y|)$ to get the stated result. ∎

A non-empty string is ***primitive*** if it is not the power of any other string. In other words, a string $x \in A^+$ is primitive if and only if every decomposition of the form $x = u^n$ with $u \in A^*$ and $n \in \mathbf{N}$ implies $n = 1$, and then $u = x$. For instance, the string `abaab` is primitive, while the strings ε and `bababa` $= (\mathtt{ba})^3$ are not.

Lemma 1.11 (Primitivity lemma)
A non-empty string is primitive if and only if it is a factor of its square only as a prefix and as a suffix. In other words, for every non-empty string x,

x primitive

if and only if

$yx \preceq_{pref} x^2$ implies $y = \varepsilon$ or $y = x$.

An illustration of this result is proposed in Figure 1.6.

Proof If x is a non-empty non-primitive string, there exist $z \in A^+$ and $n \geq 2$ such that $x = z^n$. Since x^2 can be decomposed as $z \cdot z^n \cdot z^{n-1}$, the string x occurs at the position $|z|$ on x^2. This shows that every non-empty non-primitive string is a factor of its square without being only a prefix and a suffix of it.

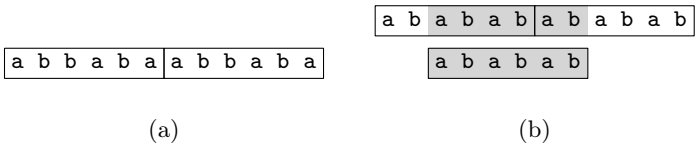


Figure 1.6 Application of the Primitivity lemma. **(a)** String $x = \texttt{abbaba}$ does not possess any "non trivial" occurrence in its square x^2 – i.e. that is neither a prefix nor a suffix of x^2 – since x is primitive. **(b)** String $x = \texttt{ababab}$ possesses a "non trivial" occurrence in its square x^2 since x is not primitive: $x = (\texttt{ab})^3$.

Conversely, let x be a non-empty string such that its square x^2 can be written as yxz with $y, z \in A^+$. Due to the length condition, it first follows that $|y| < |x|$. Then, and since $x \preceq_{pref} yx$, we obtain from Proposition 1.4 that $|y|$ is a period of x. Thus $|x|$ and $|y|$ are periods of yx. From the periodicity lemma, we deduce that $p = \gcd(|x|, |y|)$ is also a period of yx. Now, as $p \leq |y| < |x|$, p is also a period of x. And as p divides $|x|$, we deduce that x is of the form t^n with $|t| = p$ and $n \geq 2$. This shows that the string x is not primitive. ∎

Another way of stating the previous lemma is that the primitivity of x is equivalent to saying that $per(x^2) = |x|$.

Proposition 1.12
For every non-empty string, there exists one and only one primitive string which it is a power of.

Proof The proof of the existence comes from a trivial recurrence on the length of the strings. We now have to show the uniqueness.

Let x be a non-empty string. If we assume that $x = u^m = v^n$ for two primitive strings u and v and two natural non-null numbers m and n, then u and v are necessarily powers of a string $z \in A^+$ from Lemma 1.10. But their primitivity implies $z = u = v$, which shows the uniqueness and ends the proof. ∎

If x is a non-empty string, we say of the primitive string z which x is the power of that it is the ***root*** of x, and of the natural number n such that $x = z^n$ that it is the ***exponent***[4] of x.

Two strings x and y are ***conjugates*** if there exist two strings u and v such that $x = uv$ and $y = vu$. For instance, the strings $\texttt{abaab}$ and $\texttt{ababa}$ are conjugate. It is clear that conjugacy is an equivalence relation. It is not compatible with the product.

[4]More generally, the exponent of x is the quantity $|x|/per(x)$ which is not necessarily an integer (see Exercise 9.2).

Proposition 1.13
Two non-empty strings are conjugate if and only if their roots also are.

Proof The proof of the reciprocal is immediate.

For the proof of the direct implication, we consider two non-empty conjugate strings x and y and we denote by z and t then m and n their roots and exponents respectively. Since x and y are conjugate, there exist $z', z'' \in A^+$ and $p, q \in \mathbf{N}$ such that $z = z'z''$, $x = z^p z' \cdot z''z^q$, $y = z''z^q \cdot z^p z'$ and $m = p + q + 1$. We deduce that $y = (z''z')^m$. Now, as t is primitive, Lemma 1.10 implies that $z''z'$ is a power of t. This shows the existence of a natural non-null number k such that $|z| = k|t|$. Symmetrically, there exists a natural non-null number ℓ such that $|t| = \ell|z|$. It follows that $k = \ell = 1$, that $|t| = |z|$, then that $t = z''z'$. This shows that the roots z and t are conjugate. ∎

Proposition 1.14
Two non-empty strings x and y are conjugate if and only if there exists a string z such that $xz = zy$.

Proof $\Rightarrow$: x and y can be decomposed as $x = uv$ and $y = vu$ with $u, v \in A^*$, then the string $z = u$ suits since $xz = uvu = zy$.

$\Leftarrow$: in the non-trivial case where $z \in A^+$, we obtain by an immediate recurrence that $x^k z = zy^k$ for every $k \in \mathbf{N}$. Let n be the (non-null) natural number such that $(n-1)|x| \leq |z| < n|x|$. There exist thus $u, v \in A^*$ such that $x = uv$, $z = x^{n-1}u$ and $vz = y^n$. It follows that $y^n = vx^{n-1}u = (vu)^n$. Finally, since $|y| = |x|$, we have $y = vu$, which shows that x and y are conjugate. ∎

1.3 Algorithms and complexity

In this section, we present the algorithmic elements used in the rest of the book. They include the writing conventions, the evaluation of the algorithm complexity and some standard objects.

Writing conventions of algorithms

The style of the algorithmic language used here is relatively close to real programming languages but at a higher abstraction level. We adopt the following conventions:

- Indentation means the structure of blocks inherent to compound instructions.
- Lines of code are numbered in order to be referenced in the text.
- The symbol ▷ introduces a comment.
- The access to a specific attribute of an object is signified by the name

of the attribute followed by the identifier associated with the object between brackets.

- A variable that represents a given object (table, queue, tree, string, automaton) is a pointer to this object.
- The arguments given to procedures or to functions are managed by the "call by value" rule.
- Variables of procedures and of functions are local to them unless otherwise mentioned.
- The evaluation of boolean expressions is performed from left to right in a lazy way.

We consider, following the example of a language like the C language, the iterative instruction **do-while** – used instead of the traditional instruction **repeat-until** – and the instruction **break** which produces the termination of the most internal loop in which it is located.

Well adapted to the sequential processing of strings, we use the formulation:

```
1  for each letter a of u, sequentially do
2        processing of a
```

for every string u. It means that the letters $u[i]$, $i = 0, 1, \ldots, |u| - 1$, composing u are processed one after the other in the body of the loop: first $u[0]$, then $u[1]$, and so on. It means that the length of the string u is not necessarily known in advance, the end of the loop can be detected by a marker that ends the string. In the case where the length of the string u is known, this formulation is equivalent to a formulation of the type:

```
1  for i ← 0 to |u| − 1 do
2        a ← u[i]
3        processing of a
```

where the integer variable i is free (its use does not produce any conflict with the environment).

Pattern matching algorithms

A ***pattern*** represents a non-empty language not containing the empty string. It can be described by a string, by a finite set of strings, or by other means. The ***pattern matching*** problem is to search for occurrences of strings of the language in other strings – or in ***texts*** to be less formal. The notions of occurrence, of appearance and of position on the strings are extended to patterns.

According to the specified problem, the input of a pattern matching algorithm is a string x or a language X and a text y, together or not with their lengths.

The output can take several forms. Here are some of them:

- Booleans values: to implement an algorithm that tests whether the pattern occurs in the text or not, without specifying the positions of the possible occurrences, the output is simply the boolean value TRUE in the first situation and FALSE in the second.
- A string: during a sequential search, it is appropriate to produce a string $\bar{y}$ on the alphabet $\{\mathtt{0}, \mathtt{1}\}$ that encodes the existence of the right positions of occurrences. The string $\bar{y}$ is such that $|\bar{y}| = |y|$ and $\bar{y}[i] = \mathtt{1}$ if and only if i is the right position of an occurrence of the pattern on y.
- A set of positions: the output can also take the form of a set P of left – or right – positions of occurrences of the pattern on y.

Let e be a predicate having value TRUE if and only if an occurrence has just been detected. A function corresponding to the first form and ending as soon as an occurrence is detected should integrate in its code an instruction:

```
1  if e then
2       return TRUE
```

in the heart of its searching process, and return the value FALSE at the termination of this process. The second form needs to initialize the variable $\bar{y}$ with ε, the empty string, then to modify its value by an instruction:

```
1  if e then
2       ȳ ← ȳ · 1
3  else ȳ ← ȳ · 0
```

then to return it at the termination. It is identical for the third form, where the set P is initially emptied, then augmented by an instruction:

```
1  if e then
2       P ← P ∪ {the current position on y}
```

and finally returned.

In order to present only one variant of the code for an algorithm, we consider the following special instruction:

OUTPUT-IF(e) means, at the location where it appears, an occurrence of the pattern at the current position on the text is detected when the predicate e has value TRUE.

Expression of complexity

The model of computation for the evaluation of the algorithms complexity is the standard random access machine model.

In a general way, the algorithm complexity is an expression including the input size. This includes the length of the language represented by the pattern, the length of the string in which the search is performed, and the size of the alphabet. We assume that the letters of the alphabet are of size comparable to the machine word size, and, consequently, the comparison between two letters is an elementary operation that is performed in constant time.

We assume that every instruction OUTPUT-IF(e) is executed in constant time[5] once the predicate e has been evaluated.

We use the notation recommended by Knuth [72] to express the orders of magnitude. Let f and g be two functions from $\mathbf{N}$ to $\mathbf{N}$. We write "$f(n)$ is $O(g(n))$" to mean that there exists a constant K and a natural number n_0 such that $f(n) \leq Kg(n)$ for every $n \geq n_0$. In a dual way, we write "$f(n)$ is $\Omega(g(n))$" if there exists a constant K and a natural number n_0 such that $f(n) \geq Kg(n)$ for every $n \geq n_0$. We finally write "$f(n)$ is $\Theta(g(n))$" to mean that f and g are of the same order, that is to say that $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

The function $f: \mathbf{N} \to \mathbf{N}$ is ***linear*** if $f(n)$ is $\Theta(n)$, ***quadratic*** if $f(n)$ is $\Theta(n^2)$, ***cubic*** if $f(n)$ is $\Theta(n^3)$, ***logarithmic*** if $f(n)$ is $\Theta(\log n)$, ***exponential*** if there exists $a > 0$ for which $f(n)$ is $\Theta(a^n)$.

We say that a function with two parameters $f: \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ is linear when $f(m, n)$ is $\Theta(m + n)$ and quadratic when $f(m, n)$ is $\Theta(m \times n)$.

Some standard objects

Queues, states and automata are objects often used in the rest of the book. Without telling what their true implementations are – they can actually differ from one algorithm to the other – we indicate the minimal attributes and operations defined on these objects.

For queues, we only describe the basic operations.

EMPTY-QUEUE() creates then returns an empty queue.

QUEUE-IS-EMPTY(F) returns TRUE if the file F is empty, and FALSE otherwise.

ENQUEUE(F, x) adds the element x to the tail of the queue F.

HEAD(F) returns the element located at the head of the queue F.

DEQUEUE(F) deletes the element located at the head of the queue F.

DEQUEUED(F) deletes the element located at the head of the queue F then returns it;

LENGTH(F) returns the length of the queue F.

[5] Actually we can always come down to it even though the language represented by the pattern is not reduced to a single string. For that, it is sufficient to only produce one descriptor – previously computed – of the set of strings that occur at the current position (instead for instance, of producing explicitly the set of strings). It then remains to use a tool that develops the information if necessary.

States are objects that possess at least the two attributes *terminal* and *Succ*. The first attribute indicates if the state is terminal or not and the second is an implementation of the set of labeled successors of the state. The attribute corresponding to an output of a state is denoted by *output*. The two standard operations on the states are the functions NEW-STATE and TARGET. While the first creates then returns a non-terminal state with an empty set of labeled successors, the second returns the target of an arc given the source and the label of the arc, or the value special NIL if such an arc does not exist. The code for these two functions can be written in a few lines:

```
NEW-STATE()
1  allocate an object p of type state
2  terminal[p] ← FALSE
3  Succ[p] ← ∅
4  return p
```

```
TARGET(p, a)
1  if there exists a state q such that (a, q) ∈ Succ[p] then
2       return q
3  else return NIL
```

The objects of the type automaton possess at least the attribute *initial* that specifies the initial state of the automaton. The function NEW-AUTOMATON creates then returns an automaton with a single state. It constitues its initial state and has an empty set of labeled successors. The corresponding code is the following:

```
NEW-AUTOMATON()
1  allocate an object M of type automaton
2  q0 ← NEW-STATE()
3  initial[M] ← q0
4  return M
```

1.4 Implementation of automata

Some pattern matching algorithms rely on specific implementations of the deterministic automata they consider. This section details several methods, including the data structures and the algorithms, that can be used to implement these objects in memory.

Implementing a deterministic automaton (Q, q_0, T, F) consists of setting in memory, either the set F of its arcs, or the sets of the labeled successors of its states, or its transition function δ. Those are equivalent problems that fit in the general framework of representing ***partial functions*** (Exercise 1.15). We distinguish two families of implementations:

	a	b	c
0	1	0	0
1	2	3	0
2	2	3	0
3	4	0	0
4	2	3	0

Figure 1.7 The transition matrix of the automaton of Figure 1.2.

- the family of ***full*** implementations in which all the transitions are represented,
- the family of ***reduced*** implementations that use more or less elaborate techniques of compression in order to reduce the memory space of the representation.

The choice of the implementation influences the time necessary to compute a transition, i.e. to execute $\textsc{Target}(p, a)$, for a state $p \in Q$ and a letter $a \in A$. This computation time is called the ***delay*** since it measures also the time necessary for going from the current letter of the input to the next letter. Typically, two models can be opposed:

- The ***branching model*** in which δ is implemented with a $Q \times A$ matrix and where the delay is constant (in the random access model).
- The ***comparisons model*** in which the elementary operation is the comparison of letters and where the delay is typically $O(\log \operatorname{card} A)$ when any two letters can be compared in one unit of time (general assumption formulated in Section 1.3).

We also consider in the next section an elementary technique known as the "bit-vector model" whose application scope is restricted: it is especially interesting when the size of the automaton is very small.

For each of the implementation families, we specify the orders of magnitude of the necessary memory space and of the delay. There is always a trade-off to be found between these two quantities.

Full implementations

The most simple method for implementing the function δ is to store its values in a $Q \times A$ matrix, known as the ***transition matrix*** (an illustration is given Figure 1.7) of the automaton. It is a method of choice for a deterministic complete automaton on an alphabet of relatively small size and when the letters can be identified with indices on a table. Computing a transition reduces to a mere table look-up.

Proposition 1.15
In an implementation by transition matrix, the necessary memory space is $O(\operatorname{card} Q \times \operatorname{card} A)$ *and the delay* $O(1)$. ∎

In the case where the automaton is not complete, the representation

remains correct except that the execution of the automaton on the text given as an input can now stop on an undefined transition. The matrix can be initialized in time $O(\operatorname{card} F)$ only if we implement partial functions as proposed in Exercise 1.15. The above stated complexities for the memory space as well as for the delay remain valid.

An automaton can be implemented by means of an adjacency matrix as it is classical to do for graphs. We associate then to each letter of the alphabet a boolean $Q \times Q$ matrix. This representation is in general not adapted for the applications developed in this book. It is however related to the method that follows.

The method by ***list of transitions*** consists in implementing a list of triplets (p, a, q) that are arcs of the automaton. The required space is only $O(\operatorname{card} F)$. Having done this, we assume that this list is stored in a hash table in order to allow a fast computation of the transitions. The corresponding hash function is defined on the pairs $(p, a) \in Q \times A$. Given a pair (p, a), the access to the transition (p, a, q), if it is defined, is done in average constant time with the usual assumptions specific to this type of technique.

These first types of representations implicitly assume that the alphabet is fixed and known in advance, which opposes them to the representations in the comparison model considered by the method described below.

The method by ***sets of labeled successors*** consists in using a table t indexed on Q for which each element $t[p]$ gives access to an implementation of the set of the labeled successors of the state p. The required space is $O(\operatorname{card} Q + \operatorname{card} F)$. This method is valuable even when the only authorized operation on the letters is the comparison. Denoting by s the maximum outgoing degree of the states, the delay is $O(\log s)$ if we use an efficient implementation of the sets of labeled successors.

Proposition 1.16
In an implementation by sets of labeled successors, the space requirement is $O(\operatorname{card} Q + \operatorname{card} F)$ *and the delay* $O(\log s)$ *where* s *is the maximum outgoing degree of states.* ∎

Note that the delay is also $O(\log \operatorname{card} A)$ in this case: indeed, since the automaton is assumed to be deterministic, the outgoing degree of each of the states is less than $\operatorname{card} A$, thus $s \leq \operatorname{card} A$ with the notation used above.

Reduced implementations

When the automaton is complete, the space complexity can however be reduced by considering a ***successor by default*** for the computation of the transitions from any given state – the state occurring the most often in a set of labeled successors is the best possible candidate for being

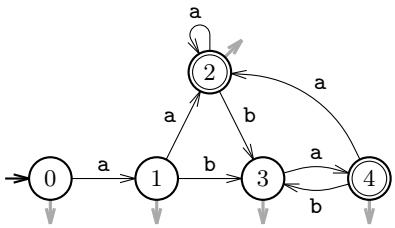


Figure 1.8 Reduced implementation by adjonction of successors by default. We consider the automaton of Figure 1.2 and we chose the initial state as unique successor by default (this choice parfectly suits for pattern matching problems). States that admit the initial state as successor by default (actually all here) are indicated by a short gray arrow. For example, the target of the transition from state 3 by the letter `a` is state 4, and by every other letter, here `b` or `c`, it is the initial state 0.

the successor by default. The delay can also be reduced since the size of the sets of labeled successors becomes smaller. For pattern matching problems, the choice of the initial state as successor by default perfectly suits. Figure 1.8 shows an example where short gray arrows mean that the state possesses the initial state as successor by default.

Another method to reduce the implementation space consists in using a failure function. The idea is here to reduce the necessary space for implementing the automaton, by redirecting, in most cases, the computation of the transition from the current state to the one from another state but by the same letter. This technique serves to implement deterministic automata in the model comparison. Its principal advantage is – generally – to provide linear size representations and to simultaneously get a linear time computation of series of transitions even when the computation of a single transition cannot be done in constant time.

Formally, let

$$\gamma\colon Q \times A \to Q$$

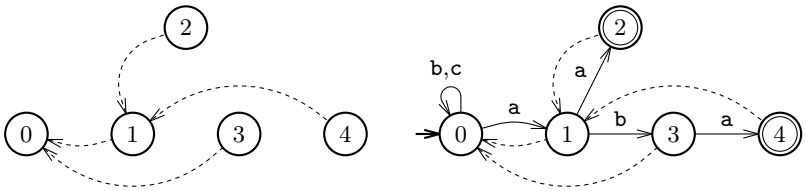
and

$$f\colon Q \to Q$$

be two functions. We say that the pair (γ, f) represents the transition function δ of a complete automaton having δ as transition function if and only if γ is a sub-function of δ, f defines an order on elements of Q, and for every pair $(p, a) \in Q \times A$

$$\delta(p,a) = \begin{cases} \gamma(p,a) & \text{if } \gamma(p,a) \text{ is defined} \ , \\ \delta(f(p),a) & \text{otherwise} \ . \end{cases}$$

When it is defined, we say of the state $f(p)$ that it is the ***failure state*** of the state p. We say of the functions γ and f that they are respectively, and jointly, a ***sub-transition*** and a ***failure function*** of δ.



(a) (b)

Figure 1.9 Reduced implementation by adjonction of a failure function. We take again the example of the automaton of Figure 1.2. **(a)** A failure function given under the form of an directed graph. As this graph does not possess any cycle, the function defines an order on the set of states. **(b)** The corresponding reduced automaton. Each link state-failure state is indicated by a dashed arrow. The computation of the transition from state 4 by the letter `c` is refered to state 1, then to state 0. State 0 is indeed the first among states 4, 1 et 0, in this order, to possess a transition defined by `c`. Finally, the target of the transition from state 4 by `c` is state 0.

We indicate the link state-failure state by a directed dash arrow in figures (see the example in Figure 1.9).

The space needed to represent the function δ by the functions γ and f is $O(\operatorname{card} Q + \operatorname{card} F')$ in the case of an implementation by sets of labeled successors where

$$F' = \{(p, a, q) \in F : \gamma(p, a) \text{ is defined}\} \ .$$

Note that γ is the transition function of the automaton (Q, q_0, T, F').

A complete example

The method presented here is a combination of the previous ones together with a fast computation of transitions and a compact representation of transitions due to the joint use of tables and of a failure function. It is known as "compression of transition table".

Two extra attributes, *fail* and *base*, are added to states, the first has values in Q and the second in $\mathbf{N}$. We consider also two tables indexed by $\mathbf{N}$ and with values in Q: *target* and *control*. For each pair $(p, a) \in Q \times A$, $base[p] + rank[a]$ is an index on both *target* and *control*, denoting by *rank* the function that associates with every letter of A its rank in a fixed ordered sequence of letters of A.

The computation of the successor of a state $p \in Q$ by a letter $a \in A$ proceeds as follows:

1. If $control[base[p]+rank[a]] = p$, $target[base[p]+rank[a]]$ is the target of the arc of source p and labeled by a.
2. Otherwise the process is repeated recursively on the state $fail[p]$ and the letter a (assuming that *fail* is a failure function).

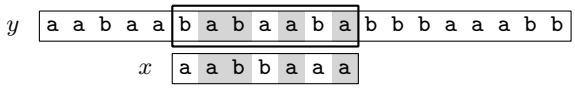


Figure 1.10 An attempt to locate string x = `aabbaaa` in text y = `aabaababaabablbbaaabb`. The attempt takes place at position 5 on y. The content of the window and the string matches in four positions.

The (non-recursive) code of the corresponding function follows.

TARGET-BY-COMPRESSION(p, a)
1 **while** $control[base[p] + rank[a]] \neq p$ **do**
2 $\quad p \leftarrow fail[p]$
3 **return** $target[base[p] + rank[a]]$

In the worst case, the space required by the implementation is $O(\operatorname{card} Q \times \operatorname{card} A)$ and the delay is $O(\operatorname{card} Q)$. This method allows us to reduce the space in $O(\operatorname{card} Q + \operatorname{card} A)$ with a constant delay in the best case.

1.5 Basic pattern matching techniques

We present in this section elementary approaches for the pattern matching problem. It includes the notion of sliding window common to many searching algorithms, the utilization of heuristics in order to reduce the computation time, the general method based on automata when the texts are to be processed in a sequential order, and the use of techniques that rely on the binary encoding of letters realized by machine words.

Notion of sliding window

When the pattern is a non-empty string x of length m, it is convenient to consider that the text y of length n in which the search is performed, is examined through a ***sliding window***. The window delimits a factor of the text – called the ***content of the window*** – which has, in most cases, the length of the string x. It slides along the text from the beginning to the end – from left to right.

The window being at a given position j on the text, the algorithm tests whether the string x occurs or not at this position, by comparing some letters of the content of the window with aligned letters of the string. We speak of an ***attempt*** at the position j (see an example in Figure 1.10). If the comparison is successful, an occurrence is signalled. During this phase of test, the algorithm acquires some information on the text which can be exploited in two ways:

- to set up the length of the next ***shift*** of the window according to rules that are specific to the algorithm,

- to avoid comparisons during next attempts by memorizing a part of the collected information.

When the shift slides the window from the position j to the position $j + d$ ($d \geq 1$), we say that the shift is of ***length*** d. To answer to the given problem, a shift of length d for an attempt at the position j must be ***valid***, i.e. it must ensure that, when $d \geq 2$, there is no occurrence of the searched string x from positions $j + 1$ to $j + d - 1$ on the text y.

The naive algorithm

The simplest implementation of the sliding window mechanism is the so-called naive algorithm. The strategy consists here in considering a window of length m and in sliding it one position to the right after each attempt. This leads, if the comparison of the content of the window and of the string is correctly implemented, to an obviously correct algorithm.

We give below the code of the algorithm. The variable j corresponds to the left position of the window on the text. It is clear that the comparison of the strings at Line 2 is supposed to be performed letter by letter according to a pre-established order.

NAIVE-SEARCH(x, m, y, n)
1 **for** $j \leftarrow 0$ **to** $n - m$ **do**
2 OUTPUT-IF$(y[j \,..\, j + m - 1] = x)$

In the worst case, the algorithm NAIVE-SEARCH executes in time $\Theta(m \times n)$, as for instance when x and y are powers of the same letter. In the average case,[6] its behavior is rather good, as the following proposition indicates.

Proposition 1.17
With the double assumption of an alphabet non-reduced to a single letter and of a uniform and independent distribution of letters of the alphabet, the average number of comparisons of letters performed by the operation NAIVE-SEARCH(x, m, y, n) *is* $\Theta(n - m)$.

Proof Let c be the size of the alphabet. The number of comparisons of letters necessary to determine if two strings u and v of length m are identical on average

$$1 + 1/c + \cdots + 1/c^{m-1} \ ,$$

independently of the permutation of positions considered for comparing compared between letters of the strings. When $c \geq 2$, this quantity is less than $1/(1 - 1/c)$, which is itself no more than 2.

[6] Even when the patterns and the texts considered in practice have no reason to be random, the average cases express what one can expect of a given pattern matching algorithm.

It follows that the average number of comparisons of letters counted during the execution of the operation is less than $2(n - m + 1)$. Thus the result holds since at least $n - m + 1$ comparisons are performed. ∎

Heuristics

Some elementary processes sensibly improve the global behavior of pattern matching algorithms. We detail here some of the most significant. They are described in connection with the naive algorithm. But most of the other algorithms can include them in their code, the adaptation being more or less easy. We speak of heuristics since we are not able to formally measure their contribution to the complexity of the algorithm.

When locating all the occurrences of the string x in the text y by the naive method, we can start by locating the occurrences of its first letter, $x[0]$, in the prefix $y[0\,..\,n-m+1]$ of y. It then remains to test, for each occurrence of $x[0]$ at a position j on y, the possible identity between the two strings $x[1\,..\,m-1]$ and $y[j+1\,..\,j+m-1]$. As the searching operation for the occurrence of a letter is generally a low level operation of operating systems, the reduction of the computation time is often noticeable in practice. This elementary search can still be improved in two ways:

- by positioning $x[0]$ as a sentinel at the end of the text y, in order to have to test less frequently the end of the text,
- by searching, non-necessarily $x[0]$, but the letter of x which has the smallest frequency of appearance in the texts of the family of y.

It should be noted that the first technique assumes that such an alteration of the memory is possible and that it can be performed in constant time. For the second, besides the necessity of having to know the frequency of letters, the choice of the position of the distinguished letter requires a precomputation on x.

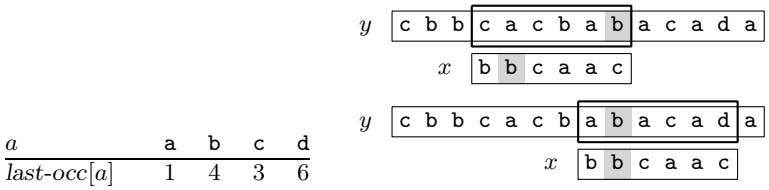
A different process consists in applying a shift that takes into account only the value of the rightmost letter of the window. Let j be the right position of the window. Two antagonist cases can be envisaged whether or not the letter $y[j]$ occurs in $x[0\,..\,m-2]$:

- in the case where $y[j]$ does not occur in $x[0\,..\,m-2]$, the string x cannot occur at right positions $j+1$ to $j+m-1$ on y,
- in the other case, if k is the maximal position of an occurrence of the letter $y[j]$ on $x[0\,..\,m-2]$, the string x cannot occur at right positions $j+1$ to $j+m-1-k-1$ on y.

Thus the valid shifts to apply in the two cases have lengths: m for the first, and $m-1-k$ for the second. Note that they do not depend on the letter $y[j]$ and in no way on its position j on y.

To formalize the previous observation, we introduce the table

last-occ: $A \to \{1, 2, \ldots, m\}$



a	a	b	c	d
last-occ$[a]$	1	4	3	6

(a) (b)

Figure 1.11 Shift of the sliding window with the table of the last occurrence, *last-occ*, when $x =$ `bbcaac`. **(a)** The values of the table *last-occ* on the alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. **(b)** The window on the text y is at right position 8. The letter at this position, $y[8] = \mathtt{b}$, occurs at the maximal position $k = 1$ on $x[0\,..\,|x| - 2]$. A valid shift consists in sliding the window of $|x| - 1 - k = 4 =$ *last-occ*$[\mathtt{b}]$ positions to the right.

defined for every letter $a \in A$ by

$$\textit{last-occ}[a] = \min(\{m\} \cup \{m - 1 - k : 0 \leq k \leq m - 2 \text{ and } x[k] = a\}) \ .$$

We call *last-occ* the ***table of the last occurrence***. It expresses a valid shift, *last-occ*$[y[j]]$, to apply after the attempt at the right position j on y. An illustration is proposed Figure 1.11. The code for the computation of *last-occ* follows. It executes in time $\Theta(m + \operatorname{card} A)$.

```
LAST-OCCURRENCE(x, m)
 1  for each letter a ∈ A do
 2        last-occ[a] ← m
 3  for k ← 0 to m − 2 do
 4        last-occ[x[k]] ← m − 1 − k
 5  return last-occ
```

We give now the complete code of the algorithm FAST-SEARCH obtained from the naive algorithm by adding the table *last-occ*.

```
FAST-SEARCH(x, m, y, n)
 1  last-occ ← LAST-OCCURRENCE(x, m)
 2  j ← m − 1
 3  while j < n do
 4        OUTPUT-IF(y[j − m + 1 .. j] = x)
 5        j ← j + last-occ[y[j]]
```

If the comparison of the strings at Line 4 starts in position $m-1$, the searching phase of the algorithm FAST-SEARCH executes in time $\Theta(n/m)$ in the best case. As for instance when no letter at positions congruent modulo m to $m - 1$ on y occurs in x; in this case, a single comparison

between letters is performed during each attempt[7] and the shift is always equal to m. The behavior of the algorithm on natural language texts is very good. One can show however that in the average case (with the double assumption of Proposition 1.17 and for a set of strings having the same length), the number of comparisons per text letter is asymptotically lower bounded by $1/\operatorname{card} A$. The bound is independent of the length of the pattern.

Search engine

Some automata can serve as a ***search engine*** for the online processing of texts. We describe in this part two algorithms based on an automaton for locating patterns. We assume the automata are given; Chapter 2 presents the construction of some of these automata.

Let us consider a pattern $X \subseteq A^*$ and a deterministic automaton M that recognizes the language A^*X (Figure 1.12(a) displays an example). The automaton M recognizes the strings that have a string of X as a suffix. For locating the strings of X that occur in a text y, it is sufficient to run the automaton M on the text y. When the current state is terminal, this means that the current prefix of y – the part of y already parsed by the automaton – belongs to A^*X; or, in other words, that the current position on y is the right position of an occurrence of a string of X. This remark leads to the algorithm whose code follows. An illustration of how the algorithm works is presented in Figure 1.12(b).

```
DET-SEARCH(M, y)
1  r ← initial[M]
2  for each letter a of y, sequentially do
3        r ← TARGET(r, a)
4        OUTPUT-IF(terminal[r])
```

Proposition 1.18
When M is a deterministic automaton that recognizes the language A^*X for a pattern $X \subseteq A^*$, the operation DET-SEARCH(M, y) *locates all the occurrences of strings of X in the text $y \in A^*$.*

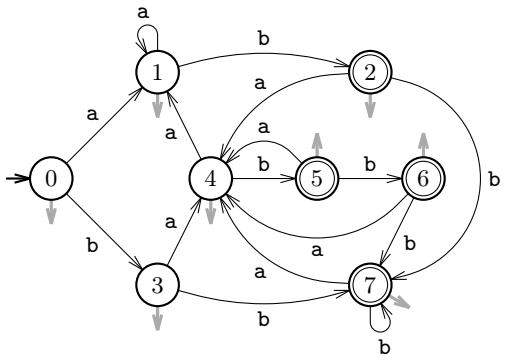
Proof Let δ be the transition function of the automaton M. As the automaton is deterministic, it follows immediately that

$$r = \delta(\mathit{initial}[M], u) \ , \tag{1.3}$$

where u is the current prefix of y, is satisfied after the execution of each of the instructions of the algorithm.

[7] Note that it is the best case possible for an algorithm detecting a string of length m in a text of length n; at least $\lfloor n/m \rfloor$ letters of the text must be inspected before the non-appearance of the searched string can be determined.

(a)



(b)

j	$y[j]$	state r	
		0	
0	c	0	
1	b	3	
2	a	4	
3	b	5	occurrence of ab
4	b	6	occurrences of babb and bb
5	a	4	

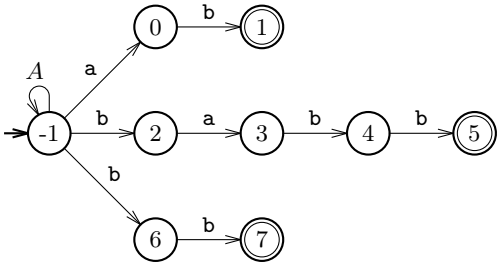
Figure 1.12 Search for occurrences of a pattern with a deterministic automaton (see also Figure 1.13). **(a)** With alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and pattern $X = \{\mathtt{ab}, \mathtt{babb}, \mathtt{bb}\}$, the deterministic automaton represented above recognizes language A^*X. The grey arrows exiting each state stand for arcs having for source these same states, for target the initial state 0, and labeled by a letter that is not already present. To locate occurrences of strings of X in a text y, it is sufficient to operate the automaton on y and to indicate each time that a terminal state is reached. **(b)** Parsing example with $y = \mathtt{cbabba}$. From the utilization of the automaton, it follows that there is at least one occurrence of a string of X at positions 3 and 4 on y, and none at other positions.

If an occurrence of a string of X ends at the current position, the current prefix u belongs to A^*X. And thus, by definition of M and after Property (1.3), the current state r is terminal. As the initial state is not terminal (since $\varepsilon \notin X$), it follows that the operation signals this occurrence.

Conversely, assume that an occurrence has just been signalled. The current state r is thus terminal, which, after Property (1.3) and by definition of M, implies that the current prefix u belongs to A^*X. An occurrence of a string of X ends thus at the current position, which ends the proof. ∎

The execution time and the extra space needed for running the algorithm DET-SEARCH uniquely depend on the implementation of the automaton M. For example, in an implementation by transition matrix, the time to parse the text is $\Theta(|y|)$, since the delay is constant,

(a)



(b)

j	$y[j]$	set of states R
		$\{-1\}$
0	c	$\{-1\}$
1	b	$\{-1, 2, 6\}$
2	a	$\{-1, 0, 3\}$
3	b	$\{-1, 1, 2, 4, 6\}$ occurrence of ab
4	b	$\{-1, 2, 5, 6, 7\}$ occurrences of babb and bb
5	a	$\{-1, 0, 3\}$

Figure 1.13 Search of occurrences of a pattern with a non deterministic automaton (see also Figure 1.12). **(a)** The non deterministic automaton recognizes the language A^*X, with alphabet $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and pattern $X = \{\mathtt{ab}, \mathtt{babb}, \mathtt{bb}\}$. To locate the occurrences of strings of X that occur in a text y, it is sufficient to operate the automaton on y and to signal an occurrence each time that a terminal state is reached. **(b)** Example when $y = \mathtt{cbabba}$. The computation amounts to simultaneously follow all possible paths. It results that the pattern occurs at right positions 3 and 4 on y and nowhere else.

and the extra space, in addition to the matrix, is also constant (see Proposition 1.15).

The second algorithm of this part applies when we dispose of an automaton N recognizing the language X itself, and no longer A^*X. By adding to the automaton an arc from its initial state to itself and labeled by a, for each letter $a \in A$, we simply get an automaton N' that recognizes the language A^*X. But the automaton N' is not deterministic, and therefore the previous algorithm cannot be applied. Figure 1.13(a) presents an example of automaton N' for the same pattern X as the one of Figure 1.12(a).

In such a situation, the retained solution usually consists in simulating the automaton obtained by the determinisation of N', following in parallel all the possible paths having a given label. Since only states that are the ends of paths may perform the occurrence test, we simply keep the set R of reached states. It is what realizes the algorithm NON-DET-SEARCH below. Actually, it is even not necessary to modify the automaton N since the loops on its initial state can also be simulated. This is realized in Line 4 of the algorithm by adding systematically the initial state to the set of states. During the execution of the automaton

on the input y, the automaton is not in a single state, but in a set of states, R. This subset of the set of states is recomputed after the analysis of the current letter of y. The algorithm calls the function TARGETS that performs a transition on a set of states, which function is an immediate extension of the function TARGET.

```
NON-DET-SEARCH(N, y)
 1  q_0 ← initial[N]
 2  R ← {q_0}
 3  for each letter a of y, sequentially do
 4        R ← TARGETS(R, a) ∪ {q_0}
 5        t ← FALSE
 6        for each state p ∈ R do
 7              if terminal[p] then
 8                    t ← TRUE
 9        OUTPUT-IF(t)
```

```
TARGETS(R, a)
1  S ← ∅
2  for each state p ∈ R do
3        for each state q such that (a, q) ∈ Succ[p] do
4              S ← S ∪ {q}
5  return S
```

Lines 5–8 of the algorithm NON-DET-SEARCH give the value TRUE to the boolean variable t when the intersection between the set of states R and the set of terminal states is non-empty. An occurrence is then signalled, Line 9, if the case arises. Figure 1.13(b) illustrates how the algorithm works.

Proposition 1.19
When N is an automaton that recognizes the language X for a pattern $X \subseteq A^*$, the operation NON-DET-SEARCH(N, y) *locates all the occurrences of strings of X in the text $y \in A^*$.*

Proof Let us denote by q_0 the initial state of the automaton N and, for every string $v \in A^*$, R_v the set of states defined by

$$R_v = \{q : q \text{ end of a path of origin } q_0 \text{ and of label } v\} \ .$$

One can verify, by recurrence on the length of the prefixes of y, that the assertion

$$R = \bigcup_{v \preceq_{suff} u} R_v \ , \tag{1.4}$$

where u is the current prefix of y, is satisfied after the execution of each of the instructions of the algorithm, except at Line 1.

If an occurrence of a string of X ends at the current position, one of the suffixes v of the current prefix u belongs to X. Therefore, by the definition of N, one of the states $q \in R_v$ is terminal, and by Property (1.4), one of the states of R is terminal. It follows that the operation signals this occurrence since no string of X is empty.

Conversely, if an occurrence has just been signalled, it means that one of the states $q \in R$ is terminal. Property (1.4) and the definition of N imply the existence of a suffix v of the current prefix u that belongs to X. It follows that an occurrence of a string of X ends at the current position. This ends the proof of the proposition. ∎

The complexity of the algorithm NON-DET-SEARCH depends both on the implementation retained for the automaton N and the realization chosen for manipulating the sets of states. If, for instance, the automaton is deterministic, its transition function is implemented by a transition matrix, and the sets of states are implemented by boolean vectors which indices are states, the function TARGETS executes in time and space $O(\operatorname{card} Q)$, where Q is the set of states. In this case, the analysis of the text y runs in time $O(|y| \times \operatorname{card} Q)$ and utilizes $O(\operatorname{card} Q)$ extra space.

In the following paragraphs, we consider an example of realization of the above simulation adapted to the case of a very small automaton that possesses a tree structure.

Bit-vector model

The ***bit-vector model*** refers to the possibility of using machine words for encoding the states of the automata. When the length of the language associated with the pattern is not larger than the size of a machine word counted in bits, this technique gives algorithms that are efficient and easy to implement. The technique is also used in Section 8.4.

Here, the principle is applied to the method that simulates a deterministic automaton and described the previous paragraphs. It encodes the set of reached states into a bit vector, and executes a transition by a simple shift controlled by a mask associated with the considered letter.

Let us start with specifying the notation used in the rest for bit vectors. We identify a bit vector with a string on the alphabet $\{\mathtt{0}, \mathtt{1}\}$. We denote respectively by $\vee$ and $\wedge$ the "or" and "and" bitwise operators. These are binary operations internal to the sets of bit vectors of identical lengths. The first operation, $\vee$, puts to `1` the bit of the result if one of the two bits at the same position of the two operands is equal to `1`, and to `0` otherwise. The second operation, $\wedge$, puts to `1` the bits of the result if the two bits at the same position of the two operands are equal to `1`, and to `0` otherwise. We denote by $\dashv$ the shift operation defined as follows: with a natural number k and a bit vector the result is the bit vector of same length obtained from the first one by shifting the bits to the right by k positions and by completing it to the left with k `0`'s.

Thus, $\mathtt{1001} \vee \mathtt{0011} = \mathtt{1011}$, $\mathtt{1001} \wedge \mathtt{0011} = \mathtt{0001}$, and $2 \dashv \mathtt{1101} = \mathtt{0011}$.

Let us consider a finite non-empty set X of non-empty strings. Let N be the automaton obtained from the card X elementary deterministic automata that recognizes the strings of X by merging their initial states into a single one, say q_0. Let N' be the automaton built on N by adding the arcs of the form (q_0, a, q_0), for each letter $a \in A$. The automaton N' recognizes the language A^*X. The search for the occurrences of strings of X in a text y is realized here as in the above paragraphs by simulating the determinized automaton of N' by means of N (see Figure 1.13(a)).

Let us set $m = |X|$ and let us number the states of N from -1 to $m-1$ using a depth-first traversal of the structure from the initial state q_0 – it is the numbering used in the example of Figure 1.13(a). Let us encode now each set of states $R \setminus \{-1\}$ by a vector r of m bits with the following convention:

$p \in R \setminus \{-1\}$ if and only if $r[p] = \mathtt{1}$.

Let r be the vector of m bits that encodes the current state of the search, $a \in A$ be the current letter of y, and s be the vector of m bits that encodes the next state. It is clear that the computation of s from r and a observes the following rule: $s[p] = \mathtt{1}$ if and only if there exists an arc of label a, either from the state -1 to the state p, or from the state $p-1$ to the state p with $r[p-1] = \mathtt{1}$. Let us consider *init* the vector of m bits defined by $\mathit{init}[p] = \mathtt{1}$ if and only if there exists an arc with state -1 as its source and state p as its target. Let us consider also the table *masq* indexed on A and with values in the set of vectors of m bits, defined for every letter $b \in A$ by $\mathit{masq}[b][p] = \mathtt{1}$ if and only if there exists an arc of label b and of target the state p. Then r, a and s satisfy the identity:

$s = (\mathit{init} \vee (1 \dashv r)) \wedge \mathit{masq}[a]$.

This latter expression translates the transition performed in Line 4 of algorithm NON-DET-SEARCH in terms of bitwise operations, except for the initial state. The bit vector *init* encodes the potential transitions from the initial state, and one-bit right shift from reached states. The table *masq* validates the transitions labeled by the current letter.

It only remains to indicate how to test whether one of the states represented by a vector r of m bits that encodes the current state of the search is terminal or not. To this goal, let *term* be the vector of m bits defined by $\mathit{term}[p] = \mathtt{1}$ if and only if the state p is terminal. Then one of the states represented by r is terminal if and only if:

$r \wedge \mathit{term} \neq \mathtt{0}^m$.

The code of the function SMALL-AUTOMATON that computes the vectors *init* and *term*, and the table *masq* follows, then the code of the pattern matching algorithm is given.

```
SMALL-AUTOMATON(X, m)
 1  init ← 0^m
 2  term ← 0^m
 3  for each letter a ∈ A do
 4       masq[a] ← 0^m
 5  p ← −1
 6  for each string x ∈ X do
 7       init[p + 1] ← 1
 8       for each letter a of x, sequentially do
 9            p ← p + 1
10            masq[a][p] ← 1
11       term[p] ← 1
12  return (init, term, masq)
```

```
SHORT-STRINGS-SEARCH(X, m, y)
1  (init, term, masq) ← SMALL-AUTOMATON(X, m)
2  r ← 0^m
3  for each letter a of y, sequentially do
4       r ← (init ∨ (1 ⊣ r)) ∧ masq[a]
5       OUTPUT-IF(r ∧ term ≠ 0^m)
```

An example of computation is treated in Figure 1.14.

Proposition 1.20
Running the operation SHORT-STRINGS-SEARCH(X, m, y) *takes a* $\Theta(m \times \operatorname{card} A + m \times |y|)$ *time. The required extra memory space is* $\Theta(m \times \operatorname{card} A)$.

Proof The time necessary for initializing the bit vectors *init*, *term* and *masq*$[a]$, for $a \in A$, is linear in their size, thus $\Theta(m \times \operatorname{card} A)$. The instructions at Lines 4 and 5 execute in $\Theta(m)$ time each. The stated complexities follow. ∎

Once this is established, when the length m is less than the number of bits of a machine word, every bit vector of m bits can be implemented with the help of a machine word whose first m bits only are significant. This gives the following result.

Corollary 1.21
When $m = |X|$ *is less than the length of a machine word, the operation* SHORT-STRINGS-SEARCH(X, m, y) *executes in time* $\Theta(|y| + \operatorname{card} A)$ *with an extra memory space* $\Theta(\operatorname{card} A)$. ∎

1.6 Borders and prefixes tables

We present in this section two fundamental methods for locating efficiently patterns or for searching for regularities in strings. There are

(a)

k	0	1	2	3	4	5	6	7
$init[k]$	1	0	1	0	0	0	1	0
$term[k]$	0	1	0	0	0	1	0	1
$masq$[a]$[k]$	1	0	0	1	0	0	0	0
$masq$[b]$[k]$	0	1	1	0	1	1	1	1
$masq$[c]$[k]$	0	0	0	0	0	0	0	0

(b)

j	$y[j]$	bit vector r	
		00000000	
0	c	00000000	
1	b	00100010	
2	a	10010000	
3	b	01101010	occurrence of ab
4	b	00100111	occurrences of babb and bb
5	a	10010000	

Figure 1.14 Using bit vectors to search for the occurrences of the pattern $X = \{\texttt{ab}, \texttt{babb}, \texttt{bb}\}$ (see Figure 1.13). **(a)** Vectors *init* and *term*, and table of vectors *masq* on the alphabet $A = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. These vectors are of length 8 since $|X| = 8$. The first vector encodes the potential transitions from the initial state. The second encodes the terminal states. The vectors of the table *masq* encode the occurrences of letters of the alphabet in the strings of X. **(b)** Successive values of the vector r that encodes the current state of the search for strings of X in the text $y =$ cbabba. The gray area that marks some bits indicates that a terminal state has been reached.

two tables, the table of borders and the table of prefixes, that both store occurrences of prefixes of a string that occur inside itself. The tables can be computed in linear time. The computation algorithms also provide methods for locating strings that are studied in details in Chapters 2 and 3 (a prelude is proposed in Exercise 1.24).

Table of borders

Let x be a string of length $m \geq 1$. We define the table

$$border\colon \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

by

$$border[k] = |Border(x[0\,..\,k])|$$

for $k = 0, 1, \ldots, m-1$. The table *border* is called the ***table of borders*** for the string x, meaning that they are borders of the non-empty prefixes of the string. Here is an example of the table of borders for the string $x =$ abbabaabbabaaaabbabbaa:

k	0	1	2	3	4	5	6	7	8	9	10	11
$x[k]$	a	b	b	a	b	a	a	b	b	a	b	a
$border[k]$	0	0	0	1	2	1	1	2	3	4	5	6

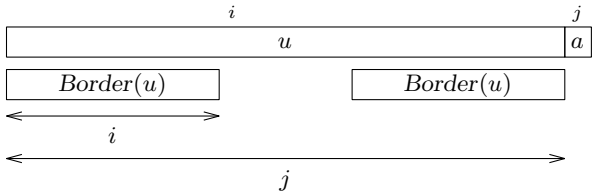


Figure 1.15 Schema showing the correspondence between variables i and j considered at Line 3 of function BORDERS and Lemma 1.22.

k	12	13	14	15	16	17	18	19	20	21
$x[k]$	a	a	a	b	b	a	b	b	a	a
$border[k]$	7	1	1	2	3	4	5	3	4	1

The following lemma provides the recurrence relation used by the function BORDERS, given thereafter, for computing the table *border*.

Lemma 1.22
For every $(u, a) \in A^+ \times A$, we have

$$Border(ua) = \begin{cases} Border(u)a & \text{if } Border(u)a \preceq_{pref} u \ , \\ Border(Border(u)a) & \text{otherwise} \ . \end{cases}$$

Proof We first note that if $Border(ua)$ is a non-empty string, it is of the form wa where w is a border of u.

If $Border(u)a \preceq_{pref} u$, the string $Border(u)a$ is then a border of ua, and the previous remark shows that it is the longest string of this kind. It follows that $Border(ua) = Border(u)a$ in this case.

Otherwise, $Border(ua)$ is both a prefix of $Border(u)$ and a suffix of $Border(u)a$. As it is of maximal length with this property, it is indeed the string $Border(Border(u)a)$. ∎

Figure 1.15 schematizes the correspondence between the variables i and j of the function *Borders*, which code follows, and the statement of Lemma 1.22.

```
BORDERS(x, m)
 1  i ← 0
 2  for j ← 1 to m − 1 do
 3        border[j − 1] ← i
 4        while i ≥ 0 and x[j] ≠ x[i] do
 5              if i = 0 then
 6                    i ← −1
 7              else i ← border[i − 1]
 8        i ← i + 1
 9  border[m − 1] ← i
10  return border
```

Proposition 1.23
The function BORDERS *applied to a string x and its length m produces the table of borders for x.*

Proof The table *border* is computed by the function BORDERS sequentially: it runs from the prefix of x of length 1 to x itself. During the execution of the **while** loop Lines 4–7 the sequence of borders of $x[0\,..\,j-1]$ is inspected, following Proposition 1.5. When exiting this loop, we have $|Border(x[0\,..\,j])| = |x[0\,..\,i]| = i+1$, in accordance with Lemma 1.22. The correctness of the code follows. ∎

Proposition 1.24
The operation BORDERS(x, m) *executes in time $\Theta(m)$. The number of comparisons between letters of the string x is within $m-1$ and $2m-3$ when $m \geq 2$. These bounds are tight.*

We say, in the rest, that the comparison between two given letters is ***positive*** when these two letters are identical, and is ***negative*** otherwise.

Proof Let us note that the execution time is linear in the number of comparisons performed between the letters of x. It is thus sufficient to establish the bound on the number of comparisons.

The quantity $2j-i$ increases by at least one unit after each comparison of letters: the variables i and j are both incremented after a positive comparison; the value of i is decreased by at least one and the value of j remains unchanged after a negative comparison. When $m \geq 2$, this quantity is equal to 2 for the first comparison ($i = 0$ and $j = 1$) and at most $2m-2$ during the last ($i \geq 0$ and $j = m-1$). The overall number of comparisons is thus bounded by $2m-3$ as stated.

The lower bound of $m-1$ is tight and is reached for $x = ab^{m-1}$. The upper bound of $2m-3$ comparisons is tight: it is reached for every string x of the form $a^{m-1}b$ with $a, b \in A$ and $a \neq b$. This ends the proof. ∎

Another proof of the bound $2m-3$ is proposed in Exercise 1.22.

Table of prefixes

Let x be a string of length $m \geq 1$. We define the table

$pref : \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

by

$pref[k] = |lcp(x, x[k\,..\,m-1])|$

for $k = 0, 1, \ldots, m-1$, where lcp(u,v) is the ***longest common prefix*** of strings u and v.

The table *pref* is called the ***table of prefixes*** for the string x. It memorizes the prefixes of x that occur inside the string itself. We note that $pref[0] = |x|$. The following example shows the table of prefixes for the string $x =$ `abbabaabbabaaaabbabbaa`.

k	0	1	2	3	4	5	6	7	8	9	10	11
$x[k]$	a	b	b	a	b	a	a	b	b	a	b	a
$pref[k]$	22	0	0	2	0	1	7	0	0	2	0	1

k	12	13	14	15	16	17	18	19	20	21
$x[k]$	a	a	a	b	b	a	b	b	a	a
$pref[k]$	1	1	5	0	0	4	0	0	1	1

Some string matching algorithms (see Chapter 3) use the table *suff* which is nothing but the analogue of the table of prefixes obtained by considering the reverse of the string x.

The method for computing *pref* that is presented below proceeds by determining $pref[i]$ by increasing values of the position i on x. A naive method would consist in evaluating each value $pref[i]$ independently of the previous values by direct comparisons; but it would then lead to a quadratic-time computation, in the case where x is the power of a single letter, for example. The utilization of already computed values yields a linear-time algorithm. For that, we introduce, the index i being fixed, two values g and f that constitute the key elements of the method. They satisfy the relations

$$g = \max\{j + pref[j] : 0 < j < i\} \tag{1.5}$$

and

$$f \in \{j : 0 < j < i \text{ and } j + pref[j] = g\} \ . \tag{1.6}$$

We note that g and f are defined when $i > 1$. The string $x[f\,..\,g-1]$ is then a prefix of x, thus also a border of $x[0\,..\,g-1]$. It is the empty string when $f = g$. We can note, moreover, that if $g < i$ we have then $g = i-1$, and that on the contrary, by definition of f, we have $f < i \leq g$.

The following lemma provides the justification for the correctness of the function Prefixes.

Lemma 1.25
If $i < g$, we have the relation

$$pref[i] = \begin{cases} pref[i-f] & \text{if } pref[i-f] < g-i \ , \\ g-i & \text{if } pref[i-f] > g-i \ , \\ g-i+\ell & \text{otherwise} \ , \end{cases}$$

where $\ell = |lcp(x[g-i\,..\,m-1], x[g\,..\,m-1])|$.

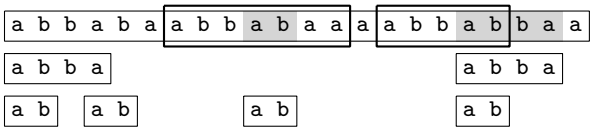


Figure 1.16 Illustration of the function PREFIXES. The framed factors $x[6\,..\,12]$ and $x[14\,..\,18]$ and the gray factors $x[9\,..\,10]$ and $x[17\,..\,20]$ are prefixes of string $x = \texttt{abbabaabbabaaaabbabbaa}$. For $i = 9$, we have $f = 6$ and $g = 13$. The situation at this position is the same that at position $3 = 9 - 6$. We have $pref[9] = pref[3] = 2$ which means that $\texttt{ab}$, of length 2, is the longest factor at position 9 that is a prefix of x. For $i = 17$, we have $f = 14$ and $g = 19$. As $pref[17 - 14] = 2 \geq 19 - 17$, we deduce that string $\texttt{ab} = x[i\,..\,g-1]$ is a prefix of x. Letters of x and $x[i\,..\,m-1]$ have to be compared from respective positions 2 and g for determining $pref[i] = 4$.

Proof Let us set $u = x[f\,..\,g-1]$. The string u is a prefix of x by the definition of f and g. Let us also set $k = pref[i-f]$. By the definition of $pref$, the string $x[i-f\,..\,i-f+k-1]$ is a prefix of x but $x[i-f\,..\,i-f+k]$ is not.

In the case where $pref[i-f] < g-i$, an occurrence of $x[i-f\,..\,i-f+k]$ starts at the position $i-f$ on u — thus also at the position i on x — which shows that $x[i-f\,..\,i-f+k-1]$ is the longest prefix of x starting at position i. Therefore, we get $pref[i] = k = pref[i-f]$.

In the case where $pref[i-f] > g-i$, $x[0\,..\,g-i-1] = x[i-f\,..\,g-f-1] = x[i\,..\,g-1]$ and $x[g-i] = x[g-f] \neq x[g]$. We have thus $pref[i] = g-i$.

In the case where $pref[i-f] = g-i$, we have $x[g-i] \neq x[g-f]$ and $x[g-f] \neq x[g]$, therefore we cannot decide on the result of the comparison between $x[g-i]$ and $x[g]$. Extra letter comparisons are necessary and we conclude that $pref[i] = g-i+\ell$. ∎

In the computation of $pref$, we initialize the variable g to 0 to simplify the writing of the code of the function PREFIXES, and we leave f initially undefined. The first step of the computation consists thus in determining $pref[1]$ by letter comparisons. The utility of the above statement comes for computing next values. An illustration of how the function works is given in Figure 1.16. A schema showing the correspondence between the variables of the function and the notation used in the statement of Lemma 1.25 and its proof is given in Figure 1.17.

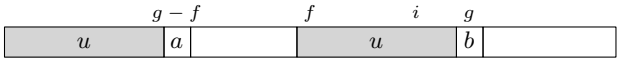


Figure 1.17 Variables i, f and g of the function PREFIXES. The main loop has for invariants: $u = lcp(x, x[f\,..\,m-1])$ and thus $a \neq b$ with $a, b \in A$, then $f < i$ when f is defined. The schema corresponds to the situation in which $i < g$.

```
PREFIXES(x, m)
 1  pref[0] ← m
 2  g ← 0
 3  for i ← 1 to m − 1 do
 4       if i < g and pref[i − f] ≠ g − i then
 5            pref[i] ← min{pref[i − f], g − i}
 6       else (g, f) ← (max{g, i}, i)
 7            while g < m and x[g] = x[g − f] do
 8                 g ← g + 1
 9            pref[i] ← g − f
10  return pref
```

Proposition 1.26
The function PREFIXES *applied to a string x and to its length m produces the table of prefixes for x.*

Proof We can verify that the variables f and g satisfy the relations (1.5) and (1.6) at each step of the execution of the loop.

We note then that, for i fixed satisfying the condition $i < g$, the function applies the relation stated in Lemma 1.25, which produces a correct computation. It remains thus to check that the computation is correct when $i \geq g$. But in this situation, Lines 6–8 compute $|lcp(x, x[i\,..\,m-1])| = |x[f\,..\,g-1]| = g - f$ which is, by definition, the value of $pref[i]$.

Therefore, the function produces the table $pref$. ∎

Proposition 1.27
The execution of the operation PREFIXES(x, m) *runs in time $\Theta(m)$. Less than $2m$ comparisons between letters of the string x are performed.*

Proof Comparisons between letters are performed at Line 7. Every comparison between equal letters increments the variable g. As the value of g never decreases and that it varies from 0 to at most m, there are at most m positive comparisons. Each negative comparison leads to the next step of the loop. Then there are at most $m - 1$ of them. Thus less than $2m$ comparisons on the overall.

The previous argument also shows that the total time of all the executions of the loop at Lines 7–8 is $\Theta(m)$. The other instructions of the

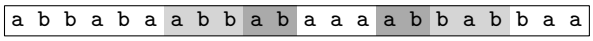


Figure 1.18 Relations between borders and prefixes. In string $x =$ abbabaabbabaaaabbabbaa, $pref[9] = 2$ and $border[9 + 2 - 1] = 5 \neq 2$. We also have $border[15] = 2$ and $pref[15 - 2 + 1] = 5 \neq 2$.

loop 3–9 take a constant time for each value of i giving again a global time $\Theta(m)$ for their execution and that of the function. ∎

The bound of $2m$ on the number of comparisons performed by the function PREFIXES is relatively tight. For instance, we get $2m - 3$ comparisons for a string of the form $a^{m-1}b$ with $m \geq 2$, $a, b \in A$ and $a \neq b$. Indeed, it takes $m - 1$ comparisons to compute $pref[1]$, then one comparison for each of the $m - 2$ values $pref[i]$ with $1 < i < m$.

Relation between borders and prefixes

The tables *border* and *pref*, whose computation is described above, both memorize occurrences of prefixes of x. We explicit here a relation between these two tables.

The relation is not immediate for the reason that follows, which is illustrated in Figure 1.18. When $pref[i] = \ell$, the factor $u = x[i\,..\,i+\ell-1]$ is a prefix of x but it is not necessarily the border of $x[0\,..\,i+\ell-1]$ because this border can be longer than u. In the same way, when $border[j] = \ell$, the factor $v = x[j-\ell+1\,..\,j]$ is a prefix of x but it is not necessarily the *longest* prefix of x occurring at position $j - \ell + 1$.

The proposition that follows shows how the table *border* is expressed using the table *pref*. One can deduce from the statement an algorithm for computing the table *border* knowing the table *pref*.

Proposition 1.28
Let $x \in A^+$ and j be a position on x. Then:

$$border[j] = \begin{cases} 0 & \text{if } I = \emptyset \ , \\ j - \min I + 1 & \text{otherwise} \ , \end{cases}$$

where $I = \{i : 0 < i \leq j \text{ and } i + pref[i] - 1 \geq j\}$.

Proof We first note that, for $0 < i \leq j$, $i \in I$ if and only if $x[i\,..\,j] \preceq_{pref} x$. Indeed, if $i \in I$, we have $x[i\,..\,j] \preceq_{pref} x[i\,..\,i+pref[i]-1] \preceq_{pref} x$, thus $x[i\,..\,j] \preceq_{pref} x$. Conversely, if $x[i\,..\,j] \preceq_{pref} x$, we deduce, by definition of $pref[i]$, $pref[i] \geq j - i + 1$. And thus $i + pref[i] - 1 \geq j$. Which shows that $i \in I$. We also note that $border[j] = 0$ if and only if $I = \emptyset$.

It follows that if $border[j] \neq 0$ (thus $border[j] > 0$) and $k = j - border[j] + 1$, we have $k \leq j$ and $x[k\,..\,j] \preceq_{pref} x$. No factor $x[i\,..\,j]$, $i < k$, satisfies the relation $x[i\,..\,j] \preceq_{pref} x$ by definition of $border[j]$. Thus $k = \min I$ by the first remark, and $border[j] = j - k + 1$ as stated. ∎

The computation of the table *pref* from the table *border* can lead to an iteration, and does not seem to give a simple expression, comparable to the one of the previous statement (see Exercise 1.23).

Notes

The chapter contains the basic elements for a precise study of algorithms on strings. Most of the notions that are introduced here are dispersed in different books. We cite here those that are often considered as references in their domains.

The combinatorial aspects on strings are dealt with in the collective books of Lothaire [73, 74, 75]. One can refer to the book of Aho, Hopcroft and Ullman [63] for algorithmic questions: expression of algorithms, data structures and complexity evaluation. We were inspired by the book of Cormen, Leiserson and Rivest [69] for the general presentation and the style of algorithms. Concerning automata and languages, one can refer to the book of Berstel [67] or the one of Pin [76]. The books of Berstel and Perrin [68] and of Béal [65] contain elements on the theory of codes (Exercises 1.10 and 1.11). Finally, the book of Aho, Sethi and Ullman [64] describes methods for the implementation of automata.

Section 1.5 on basic techniques contains elements frequently selected for the final development of software using algorithms that process strings. They are, more specifically, heuristics and utilization of machine words. This last technique is also tackled in Chapter 8 for approximate pattern matching. This type of technique has been initiated by Baeza-Yates and Gonnet [89] and by Wu and Manber [185]. The algorithm FAST-SEARCH is from Horspool [130]. The search for a string by means of a hash function is analyzed by Karp and Rabin [137].

The treatment of notions in Section 1.6 is original. The computation of the table of borders is classical. It is inspired by an algorithm of Morris and Pratt of 1970 (see [9]) that is at the origin of the first string matching algorithm running in linear time. The table of prefixes synthesizes differently the same information on a string as the previous table. The dual notion of table of suffixes is used in Chapter 3. Gusfield [5] makes it a fundamental element of string matching methods. (His Z algorithm corresponds to the algorithm SUFFIXES of Chapter 3).

Exercises

1.1 (Computation)
What is the number of prefixes, suffixes, factors and subsequences of a given string ? Discuss if necessary.

1.2 (Fibonacci morphism)

A ***morphism*** f on A^* is an application from A^* into itself that satisfies the rules:

$$\begin{aligned} f(\varepsilon) &= \varepsilon \ , \\ f(x \cdot y) &= f(x) \cdot f(y) \qquad \text{for } x, y \in A^* \ . \end{aligned}$$

For every natural number n and every string $x \in A^*$, we denote by $f^n(x)$ the string defined by $f^0(x) = x$ and $f^k(x) = f^{k-1}(f(x))$ for $k = 1, 2, \ldots, n$.

Let us consider the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$. Let φ be the morphism on A^* defined by $\varphi(\mathtt{a}) = \mathtt{ab}$ and $\varphi(\mathtt{b}) = \mathtt{a}$. Show that the string $\varphi^n(\mathtt{a})$ is identical to f_{n+2}, the Fibonacci string of index $n + 2$.

1.3 (Permutation)
We call a permutation on the alphabet A a string u that satisfies the condition $\operatorname{card} alph(u) = |u| = \operatorname{card} A$. This is thus a string in which all the letters of the alphabet occur exactly once.

For $k = \operatorname{card} A$, show that there exists a string of length less than $k^2 - 2k + 4$ that contains as subsequences all the permutations on A. Design a construction algorithm for such a string. [*Hint:* see Mohanty [157].]

1.4 (Period)
Show that the condition 3 of Proposition 1.4 can be replaced by the following condition: there exists a string t and an integer $k > 0$ such that $x \preceq_{fact} t^k$ and $|t| = p$.

1.5 (Limit case)
Show that the string $(\mathtt{ab})^k\mathtt{a}(\mathtt{ab})^k\mathtt{a}$ with $k \geq 1$ is the limit case for the periodicity lemma.

1.6 (Three periods)
On the triplets of sorted positive integers (p_1, p_2, p_3), $p_1 \leq p_2 \leq p_3$, we define the derivation by: the derivative of (p_1, p_2, p_3) is the triplet made of the integers p_1, $p_2 - p_1$ and $p_3 - p_1$. Let (q_1, q_2, q_3) be the first triplet obtained by iterating the derivation from (p_1, p_2, p_3) and such that $q_1 = 0$.

Show that if the string $x \in A^*$ has p_1, p_2 and p_3 as periods and that

$$|x| \geq \frac{1}{2}(p_1 + p_2 + p_3 - 2\gcd(p_1, p_2, p_3) + q_2 + q_3) \ ,$$

then it has also $\gcd(p_1, p_2, p_3)$ as period. [*Hint:* see Mignosi and Restivo [74].]

1.7 (Three squares)
Let u, v and w be three non-empty strings. Show that we have $2|u| < |w|$ if we assume that u is primitive and that $u^2 \prec_{pref} v^2 \prec_{pref} w^2$ (see Proposition 9.17 for a more precise consequence).

1.8 (Conjugates)
Show that two non-empty conjugate strings have the same exponent and conjugate roots.

Show that the conjugacy class of every non-empty string x contains $|x|/k$ elements where k is the exponent of x.

1.9 (Periods)
Let p be a period of x that is not a multiple of $per(x)$. Show that $p > |x| - per(x)$.

Let p and q be two periods of x such that $p < q$. Show that:

- $q - p$ is a period of $first_{|x|-p}(x)$ and of $(first_p(x))^{-1}x$,
- p and $q + p$ are periods of $first_q(x)x$.

(The definition of $first_k$ is given in Section 4.4.)

Show that if $x = uvw$, uv and vw have period p and $|v| \geq p$, then x has period p.

Let us assume that x has period p and contains a factor v of period r with r divisor of q. Show that r is also a period of x.

1.10 (Code)
A language $X \subseteq A^*$ is a ***code*** if every string of X^+ has a unique decomposition in strings of X.

Show that the ASCII codes of characters on the alphabet $\{\mathtt{0}, \mathtt{1}\}$ form a code according to this definition.

Show that the languages $\{\mathtt{a}, \mathtt{b}\}^*$, $\mathtt{ab}^*$, $\{\mathtt{aa}, \mathtt{ba}, \mathtt{b}\}$, $\{\mathtt{aa}, \mathtt{baa}, \mathtt{ba}\}$ and $\{\mathtt{a}, \mathtt{ba}, \mathtt{bb}\}$ are codes. Show that this is not the case of the languages $\{\mathtt{a}, \mathtt{ab}, \mathtt{ba}\}$ and $\{\mathtt{a}, \mathtt{abbba}, \mathtt{babab}, \mathtt{bb}\}$.

A language $X \subseteq A^*$ is prefix if the condition

$u \preceq_{pref} v$ implies $u = v$

is satisfied for every strings $u, v \in X$. The notion of a suffix language is defined in a dual way.

Show that every prefix language is a code. Do the same for suffix languages.

1.11 (Default theorem)
Let $X \subseteq A^*$ be a finite set that is not a code. Let $Y \subseteq A^*$ be a code for which Y^* is the smallest set of this form that contains X^*. Show that $\operatorname{card} Y < \operatorname{card} X$. [*Hint:* every string $x \in X$ can be written in the form $y_1 y_2 \dots y_k$ with $y_i \in Y$ for $i = 1, 2, \dots, k$; show that the function $\alpha: X \to Y$ defined by $\alpha(x) = y_1$ is surjective but is not injective; see [73].]

1.12 (Commutation)
Show by the default theorem (see Exercise 1.11), then by the Periodicity lemma that, if $uv = vu$, for two strings $u, v \in A^*$, u and v are powers of a same string.

1.13 (nlogn)
Let $f\colon \mathbf{N} \to \mathbf{N}$ be a function defined by:

$$\begin{aligned} f(1) &= a \ , \\ f(n) &= f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + bn \qquad \text{for } n \geq 2 \ , \end{aligned}$$

with $a \in \mathbf{N}$ and $b \in \mathbf{N} \setminus \{0\}$. Show that $f(n)$ is $\Theta(n \log n)$.

1.14 (Filter)
We consider a code for which characters are encoded on 8 bits. We want to develop a pattern matching algorithm using an automaton for strings written on the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

Describe data structures to realize the automaton with the help of a transition matrix of size $4 \times m$ (and not $256 \times m$), where m is the number of states of the automaton, possibly using an amount of extra space which is independent of m.

1.15 (Implementation of partial functions)
Let $f\colon E \to F$ be a partial function where E is a finite set. Describe an implementation of f able to perform each of the four following operations in constant time:

- initialize f, such that $f(x)$ is undefined for $x \in E$,
- set the value of $f(x)$ to $y \in F$, for $x \in E$,
- test whether $f(x)$ is defined or not, for $x \in E$,
- produce the value of $f(x)$, for $x \in E$.

One can use $O(\operatorname{card} E)$ space. [*Hint:* simultaneously use a table indexed by E and a list of elements x for which $f(x)$ is defined, with cross-references between the table and the list.]

Deduce that the implementation of such a function can be done in linear time in the number of elements of E whose images by f are defined.

1.16 (Not so naive)
We consider here a slightly more elaborate implementation for the sliding window mechanism that the one described for the naive algorithm. Among the strings x of length $m \geq 2$, it distinguishes two classes: one for which the first two letters are identical (thus $x[0] = x[1]$), and the antagonist class (thus $x[0] \neq x[1]$). This elementary distinction allows us to shift the window by two positions to the right in the following cases: string x belongs to the first class and $y[j+1] \neq x[1]$; string x belongs to the second class and $y[j+1] = x[1]$. On the other hand, if the comparison of the string x with the content of the window is always performed letter by letter, it considers positions on x in the following order $1, 2, \ldots, m-1, 0$.

Give the code of an algorithm that applies this method.

Show that the number of comparisons between text letters is on the average strictly less than 1 when the average is evaluated on the set

of strings of same length, that this length is more than 2 and that the alphabet contains at least four letters. [*Hint:* see Hancart [122].]

1.17 (End of window)
Let us consider the method that, as the algorithm FAST-SEARCH using the rightmost letter in the window for performing a shift, uses the two rightmost letters in the window (assuming that the string is of length greater than 2).

Give the code of an algorithm that applies this method.

In which cases does it seem efficient? [*Hint:* see Zhu and Takaoka [186] or Baeza-Yates [88].]

1.18 (After the window)
Same statement than the one of Exercise 1.17, but with using the letter located immediately to the right of the window (beware of the overflow at the right extremity of the text). [*Hint:* see Sunday [178].]

1.19 (Sentinel)
We come back again to the string matching problem: locating occurrences of a string x of length m in a text y of length n.

The sentinel technique can be used for searching the letter $x[m-1]$ by performing the shifts with the help of the table *last-occ*. Since the shifts can be of length m, we set $y[n\,..\,n+m-1]$ to $x[m-1]^m$. Give a code for this sentinel method.

In order to speed up the process and decrease the number of tests on letters, it is possible to chain several shifts without testing the letters of the text. For that, we back up the value of $\mathit{last\text{-}occ}[x[m-1]]$ in a variable, let say d, then we fix the value of $\mathit{last\text{-}occ}[x[m-1]]$ to 0. We can then chain shifts until one of them is of length 0. We then test the other letters of the window, signalling an occurrence when it arises, and we apply a shift of length d. Give a code for this method. [*Hint:* see Hume and Sunday [131].]

1.20 (In C)
Give an implementation in C language of the algorithm SHORT-STRINGS-SEARCH. The operators $\vee$, $\wedge$ and $\dashv$ are encoded by `|`, `&` and `<<`. Extend the implementation so that it accepts any parameter m (possibly strictly greater than the number of bits of a machine word).

Compare the obtained code to the source of the Unix command `agrep`.

1.21 (Short strings)
Describe a pattern matching algorithm for short strings in a similar way to the algorithm SHORT-STRINGS-SEARCH, but in which the binary values `0` and `1` are swapped.

1.22 (Bound)
Show that the number of positive comparisons and the number of negative comparisons performed during the operation BORDERS(x, m) are at most $m - 1$. Prove again the bound $2m - 3$ of Proposition 1.24.

1.23 (Table of prefixes)
Describe a linear time algorithm for the computation of the table *pref*, given the table *border* for the string x.

1.24 (Localisation by the borders or the prefixes)
Show that the table of borders for the string $x\$y$ can be directly used in order to locate all the occurrences of the string x in the string y, where $\$ \notin alph(xy)$.

Same question with the table of prefixes for the string xy.

1.25 (Cover)
A string u is a cover of a string x if for every position i on x, there exists a position j on x for which $0 \leq j \leq i < j + |u| \leq |x|$ and $u = x[j\,..\,j + |u| - 1]$.

Design an algorithm for the computation of the shortest cover of a string. State its complexity.

1.26 (Long border)
Let u be a non-empty border of the string $x \in A^*$.

Let $v \in A^*$ be such that $|v| < |u|$. Show that v is a border of u if and only if it is a border of x.

Show that x has another non-empty border if u satisfies the inequality $|x| < 2|u|$. Show that x has no other border satisfying the same inequality if $per(x) > |x|/4$.

1.27 (Border free)
We say that a non-empty string u is border free if $Border(u) = \varepsilon$, or, equivalently, if $per(u) = |u|$.

Let $x \in A^*$. Show that $C = \{u : u \preceq_{pref} x \text{ and } u \text{ is border free}\}$ is a suffix code (see Exercise 1.10).

Show that x uniquely factorizes into $x_k x_{k-1} \dots x_1$ according to the strings of C ($x_i \in C$ for $i = 1, 2, \dots, k$). Show that x_1 is the shortest string of C that is a suffix of x and that x_k is the longest string of C that is a prefix of x.

Design a linear time algorithm for computing the factorization.

1.28 (Maximal suffix)
We denote by $SM(\leq, u)$ the maximal suffix of $u \in A^+$ for the lexicographic order where, in this notation, $\leq$ denotes the order on the alphabet. Let $x \in A^+$.

Show that $|x| - |SM(\leq, x)| < per(x)$.

We assume that $SM(\leq, x) = x$ and we denote by $w_1, w_2, \ldots, w_k$ the borders of x in decreasing order of length (we have $k > 0$ and $w_k = \varepsilon$). Let $a_1, a_2, \ldots, a_k \in A$ and $z_1, z_2, \ldots, z_k \in A^*$ be such that

$$x = w_1 a_1 z_1 = w_2 a_2 z_2 = \cdots = w_k a_k z_k \ .$$

Show that $a_1 \leq a_2 \leq \cdots \leq a_k$.

Design a linear-time algorithm that computes the maximal suffix (for the lexicographic order) of a string $x \in A^+$. [*Hint:* use the algorithm that computes the borders of Section 1.6 or see Booth [93]; see also [3].]

1.29 (Critical factorisation)
Let $x \in A^+$. For each position i on x, we denote by

$$rep(i) = \min\{|u| : u \in A^+, A^*u \cup A^*x[0\,..\,i-1] \neq \emptyset \text{ and } uA^* \cup x[i\,..\,|x|-1]A^* \neq \emptyset\}$$

the ***local period*** of x at position i.

Let $w = SM(\leq, x)$ (SM is defined in Exercise 1.28) and assume that $|w| \leq |SM(\leq^{-1}, x)|$; show that $rep_x(|x| - |w|) = per(x)$. [*Hint:* note that the intersection of the two orderings on strings induced by $\leq$ and $\leq^{-1}$ is the prefix ordering, and use Proposition 1.4; see Crochemore and Perrin [108, 3].]