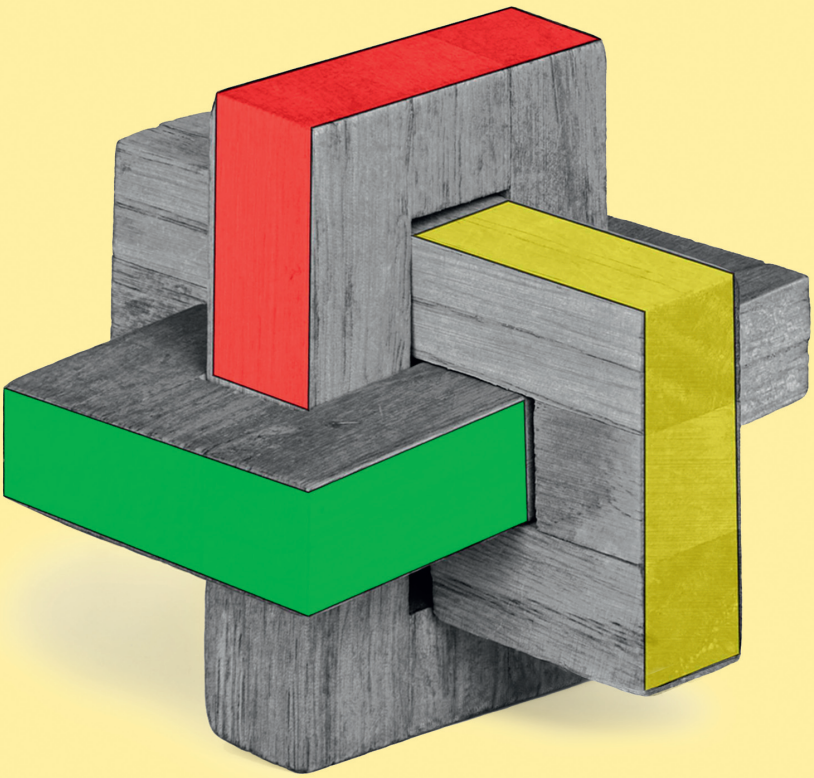
125 Problems in Text Algorithms

Maxime Crochemore
Thierry Lecroq
Wojciech Rytter



Preface

This book is about Algorithms on Texts, also called Algorithmic Stringology. Text (word, string, sequence) is one of the main unstructured data types and the subject is of vital importance in Computer science.

The subject is versatile because it is a basic requirement in many sciences, especially in Computer science and engineering. The treatment of unstructured data is a very lively area and demands efficient methods due both to their presence in highly repetitive instructions of operating systems and to the vast amount of data that needs to be analysed on digital networks and equipments. The latter is clear for Information Technology companies that manage massive data in their data centres but also holds for most scientific areas beyond Computer science.

The book presents a collection of the most interesting representative problems in Stringology. They are introduced in a short and pleasant way and open doors to more advanced topics. They were extracted from hundreds of serious scientific publications, some of which are more than hundred years old and some are very fresh and up to date. Most of the problems are related to applications while others are more abstract. The core part of most of them is an ingenious short algorithmic solution except for a few introductory combinatorial problems.

This is not just yet another monograph on the subject but a series of problems (puzzles and exercises). It is a complement to books dedicated to the subject in which topics are introduced in a more academic and comprehensive way. Nevertheless most concepts in the field are included in the book, which fills a missing gap and is very expected and needed, especially for students and teachers, as the first problem-solving textbook of the domain.

The organisation of the book consists of seven chapters.

The very basics of stringology is a preliminary chapter introducing the terminology, basic concepts and tools for the next chapters and that reflects six main streams in the area.

Combinatorial puzzles is about Combinatorics on words, an important topic since many algorithms are based on combinatorial properties of their input.

Pattern matching deals with the most classical subject, text searching and string matching.

Efficient data structures is about data structures for text indexing. They are used as fundamental tools in a large amount of algorithms, like special arrays and trees associated with texts.

Regularities in words concerns regularities that occur in texts, in particular repetitions and symmetries, that have a strong influence on the efficiency of algorithms.

Text compression is devoted to several methods of the practically im-

portant area of conservative text compression.

Miscellaneous contains various problems that do not fit in earlier chapters but certainly deserve presentation.

Problems listed in the book have been accumulated and developed over several years of teaching on string algorithms in our own different institutions in France, Poland, UK and USA. They have been taught mostly to Master's students and are given with solutions as well as with references for further readings. The content also profits from the experience authors gained in writing previous textbooks.

Anyone teaching graduate courses on data structures and algorithms can select whatever they like from our book for their students. However the overall book is not elementary and is intended as a reference for researchers, PhD and Master students, as well as for academics teaching courses on algorithms even if they are not directly related to text algorithms. It should be viewed as a companion to standard textbooks on the domain. The self-contained presentation of problems provides a rapid access to their understanding and to their solutions without requiring a deep background on the subject.

The book is useful for specialised courses on text algorithms, as well as for more general courses on algorithms and data structures. It introduces all required concepts and notions to solve problems but some prerequisites in bachelor or sophomore-level academic courses on algorithms, data structures and discrete mathematics certainly help grasping the material more easily.

November 2019

M. Crochemore, T Lecroq, W. Rytter
London, Paris, Rouen, Warsaw

1 The very basics of stringology ***1***

2 Combinatorial puzzles ***15***
1 Stringologic proof of Fermat's little theorem *17*
2 Simple case of codicity testing *18*
3 Magic squares and Thue-Morse word *19*
4 Oldenburger-Kolakoski sequence *21*
5 Square-free game *23*
6 Fibonacci words and Fibonacci numeration system *25*
7 Wythoff's game and Fibonacci word *27*
8 Distinct periodic words *29*
9 A relative of Thue-Morse word *31*
10 Thue-Morse words and sums of powers *32*
11 Conjugates and rotations of words *33*
12 Conjugate palindromes *35*
13 Many words with many palindromes *37*
14 Short superword of permutations *39*
15 Short supersequence of permutations *41*
16 Skolem words *43*
17 Langford words *46*
18 From Lyndon words to de Bruijn words *48*

3 Pattern matching ***51***
19 Border table *53*
20 Shortest covers *55*
21 Short borders *57*
22 Prefix table *59*
23 Border table to Maximal suffix *61*
24 Periodicity test *63*
25 Strict borders *65*
26 Delay of sequential string matching *67*
27 Sparse matching automaton *69*
28 Comparison-effective string matching *71*
29 Strict border table of Fibonacci word *73*
30 Words with singleton variables *75*
31 Order-preserving patterns *77*
32 Parameterised matching *79*
33 Good-suffix table *81*
34 Worst case of Boyer-Moore algorithm *83*
35 Turbo-BM algorithm *85*
36 String-matching with don't cares *87*
37 Cyclic equivalence *88*
38 Simple maximal suffix computation *91*
39 Self-maximal words *93*
40 Maximal suffix and its period *95*
41 Critical position of a word *97*
42 Periods of Lyndon word prefixes *99*
43 Searching Zimin words *101*
44 Searching irregular 2D-patterns *103*

4 Efficient data structures *105*
45 List algorithm for shortest cover *106*
46 Computing longest common prefixes *107*
47 Suffix array to Suffix tree *109*
48 Linear Suffix trie *112*
49 Ternary search trie *115*
50 Longest common factor of two words *117*
51 Subsequence automaton *119*
52 Codicity test *121*
53 LPF table *123*
54 Sorting suffixes of Thue-Morse words *126*
55 Bare Suffix tree *129*
56 Comparing suffixes of a Fibonacci word *131*
57 Avoidability of binary words *133*
58 Avoiding a set of words *135*
59 Minimal unique factors *137*
60 Minimal absent words *139*
61 Greedy superstring *142*
62 Shortest common superstring of short words *145*
63 Counting factors by length *147*
64 Counting factors covering a position *149*
65 Longest common-parity factors *150*
66 Word square-freeness with DBF *151*
67 Generic words of factor equations *153*
68 Searching an infinite word *155*
69 Perfect words *157*
70 Dense binary words *161*
71 Factor oracle *164*

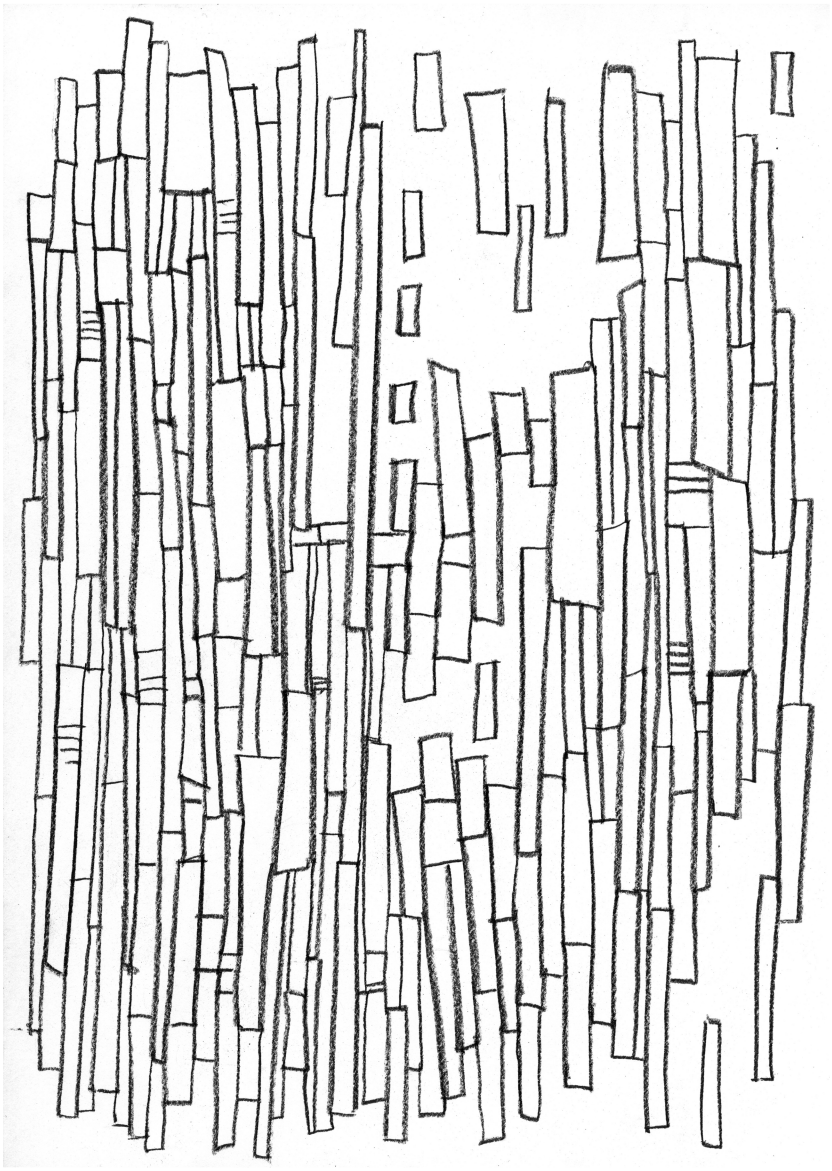
5 Regularities in words *169*
72 Three square prefixes *170*
73 Tight bounds on occurrences of powers *172*
74 Computing runs on general alphabets *174*
75 Testing overlaps in a binary word *176*
76 Overlap-free game *178*
77 Anchored squares *180*
78 Almost square-free words *183*
79 Binary words with few squares *185*
80 Building long square-free words *187*
81 Testing morphism square-freeness *189*
82 Number of square factors in labelled trees *191*
83 Counting squares in combs in linear time *193*
84 Cubic runs *195*
85 Short square and local period *197*
86 The number of runs *199*
87 Computing runs on ordered alphabets *201*
88 Periodicity and factor complexity *205*
89 Periodicity of morphic words *206*
90 Simple anti-powers *208*
91 Palindromic concatenation of palindromes *210*

92 Palindrome trees *211*
93 Unavoidable patterns *213*

6 Text compression *215*
94 BW transform of Thue-Morse words *217*
95 BW transform of balanced words *219*
96 In-place BW transform *223*
97 Lempel-Ziv factorisation *225*
98 Lempel-Ziv-Welch decoding *227*
99 Cost of Huffman code *230*
100 Length-limited Huffman coding *233*
101 On-line Huffman coding *238*
102 Run-length encoding *241*
103 A compact Factor automaton *245*
104 Compressed matching in Fibonacci word *248*
105 Prediction by partial matching *250*
106 Compressing Suffix arrays *253*
107 Compression ratio of greedy superstrings *255*

7 Miscellaneous *259*
108 Binary Pascal words *261*
109 Self-reproducing words *263*
110 Weights of factors *265*
111 Letter-occurrence differences *267*
112 Factoring with border-free prefixes *269*
113 Primitivity test for unary extensions *271*
114 Partially commutative alphabets *273*
115 Greatest fixed-density necklace *275*
116 Period-equivalent binary words *277*
117 On-line generation of de Bruijn words *280*
118 Recursive generation of de Bruijn words *283*
119 Word equations with given lengths of variables *285*
120 Diverse factors over a 3-letter alphabet *287*
121 Longest increasing subsequence *289*
122 Unavoidable sets via Lyndon words *291*
123 Synchronising words *293*
124 Safe-opening words *295*
125 Superwords of shortened permutations *299*

1 The very basics of stringology



In this chapter we introduce basic notation and definitions on words, and sketch several constructions used in text algorithms.

Texts are central in "word processing" systems, which provide facilities for the manipulation of texts. Such systems usually process objects that are quite large. Text algorithms occur in many areas of science and information processing. Many text editors and programming languages have facilities for processing texts. In molecular biology for example, text algorithms arise in the analysis of biological molecular sequences.

Words

An ***alphabet*** is a non-empty set whose elements are called ***letters*** or symbols. We typically use alphabets $\mathbf{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots\}$, $\mathbf{B} = \{\mathtt{0}, \mathtt{1}\}$ and natural numbers. A ***word*** (*mot*, in French) or ***string*** on an alphabet A is a sequence of elements of A.

The zero letter sequence is called the ***empty word*** and is denoted by ε. The set of all finite words on an alphabet A is denoted by A^*, and $A^+ = A^* \setminus \{\varepsilon\}$.

The ***length*** of a word x, length of the sequence, is denoted by $|x|$. We denote by $x[i]$, for $i = 0, 1, \ldots, |x| - 1$, the letter at ***position*** or ***index*** i on a non-empty word x. Then $x = x[0]x[1]\cdots x[|x| - 1]$ also denoted by $x[0\,..\,|x| - 1]$. The set of letters that occur in the word x is denoted by $alph(x)$. For the example $x = \mathtt{abaaab}$ we have $|x| = 6$ and $alph(x) = \{\mathtt{a}, \mathtt{b}\}$.

The ***product*** or ***concatenation*** of two words x and y is the word composed of the letters of x followed by the letters of y. It is denoted by xy, or by $x \cdot y$ to emphasise the decomposition of the resulting word. The neutral element for the product is ε and we denote respectively by zy^{-1} and $x^{-1}z$ the words x and y when $z = xy$.

A ***conjugate***, ***rotation*** or ***cyclic shift*** of a word x is any word y that factorises into vu where $uv = x$. This makes sense because the product of words is obviously non commutative. For example, the set of conjugates of $\mathtt{abba}$, its conjugacy class because conjugacy is an equivalence relation, is $\{\mathtt{aabb}, \mathtt{abba}, \mathtt{baab}, \mathtt{bbaa}\}$ and that of $\mathtt{abab}$ is $\{\mathtt{abab}, \mathtt{baba}\}$.

A word x is a ***factor*** (sometimes called ***substring***) of a word y if $y = uxv$ for two words u and v. When $u = \varepsilon$, x is a ***prefix*** of y, and when $v = \varepsilon$, x is a ***suffix*** of y. Sets $Fact(x)$, $Pref(x)$ and $Suff(x)$ denote the sets of factors, prefixes and suffixes of x respectively

When x is a non-empty factor of $y = y[0\,..\,n - 1]$ it is of the form $y[i\,..\,i + |x| - 1]$ for some i. An ***occurrence*** of x in y is an interval $[i\,..\,i + |x| - 1]$ for which $x = y[i\,..\,i + |x| - 1]$. We say that i is the ***starting position*** (or left position) on y of this occurrence, and that

$i + |x| - 1$ is its ***ending position*** (or right position). An occurrence of x in y can also be defined as a triple (u, x, v) such that $y = uxv$. Then the starting position of the occurrence is $|u|$. For example, the starting and ending positions of $x = \texttt{aba}$ on $y = \texttt{babaababa}$ are:

i	0	1	2	3	4	5	6	7	8
$y[i]$	b	a	b	a	a	b	a	b	a
starting positions		1			4		6		
ending positions				3			6		8

For words x and y, $|y|_x$ denotes the number of occurrences of x in y. Then for instance $|y| = \Sigma\{|y|_a : a \in alph(y)\}$.

The word x is a ***subsequence*** or ***subword*** of y if the latter decomposes into $w_0x[0]w_1x[1]\dots x[|x|-1]w_{|x|}$ for words w_0, w_1, …, $w_{|x|}$.

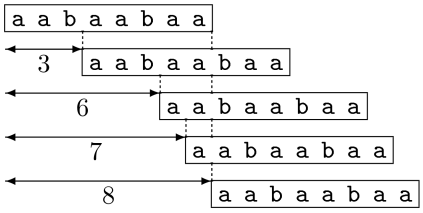
A factor or a subsequence x of a word y is said to be ***proper*** if $x \neq y$.

Periodicity

Let x be a non-empty word. An integer p, $0 < p \leq |x|$, is called a ***period*** of x if $x[i] = x[i+p]$ for $i = 0, 1, \dots, |x| - p - 1$. Note that the length of a word is a period of this word, so every non-empty word has at least one period. ***The period*** of x, denoted by $per(x)$, is its smallest period. For example, 3, 6, 7, and 8 are periods of the word aabaabaa, and $per(\texttt{aabaabaa}) = 3$. Note that if p is a period of x, its multiples not larger than $|x|$ are also periods of x.

Here is a series of properties equivalent to the definition of a period p of x. First, x can be factorised uniquely as $(uv)^k u$, where u and v are words, v is non-empty, k is a positive integer and $p = |uv|$. Second, x is a prefix of ux for a word u of length p. Third, x is a factor of u^k, where u is a word of length p and k a positive integer. Fourth, x can be factorised as $uw = wv$ for three words u, v and w verifying $p = |u| = |v|$.

The last point leads to the notion of border. A ***border*** of x is a proper factor of x that is both a prefix and a suffix of x. ***The border*** of x, denoted by $Border(x)$, is its longest border. Thus, ε, a, aa, and aabaa are the borders of aabaabaa and $Border(\texttt{aabaabaa}) = \texttt{aabaa}$.



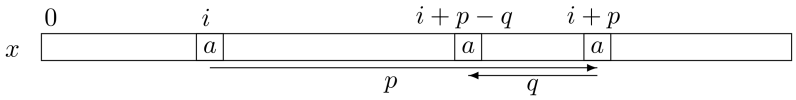
Borders and periods of x are in one-to-one correspondence due to the fourth point above: a period p of x is associated with the border $x[p\,..\,|x|-1]$.

Note that, when defined, the border of a border of x is also a border

of x. Then $\langle Border(x), Border^2(x), \ldots, Border^k(x) = \varepsilon\rangle$ is the list of all borders of x. The (non-empty) word x is said to be ***border-free*** if its only border is the empty word or equivalently if its only period is $|x|$.

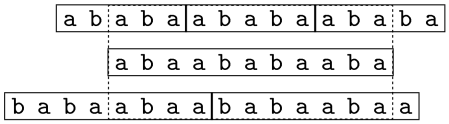
Lemma 1 (Periodicity lemma)
If p and q are periods of a word x and satisfy $p+q-\gcd(p,q) \leq |x|$ then $\gcd(p,q)$ is also a period of x.

The proof of the lemma may be found in textbooks (see Notes). The Weak Periodicity lemma refers to a variant of the lemma in which the condition is strengthen to $p+q \leq |x|$. Its proof comes readily as follows.



The conclusion obviously holds when $p = q$. Else, w.l.o.g. assume $p > q$ and let us show first that $p - q$ is a period of x. Indeed, let i be a position on x for which $i + p < |x|$. Then $x[i] = x[i + p] = x[i + p - q]$ because p and q are periods. And if $i + p \geq |x|$ the condition implies $i - q \geq 0$. Then $x[i] = x[i - q] = x[i + p - q]$ as before. Thus $p - q$ is a period of x. Iterating the reasoning or using a recurrence as for Euclid's algorithm, we conclude that $\gcd(p, q)$ is a period of x.

To illustrate the Periodicity lemma, let us consider a word x that admits 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. Thus, the condition of the lemma cannot hold, that is, $|x| < 5 + 8 - \gcd(5, 8) = 12$.



The extreme situation is displayed in the picture and shows (when generalised) that the condition required on periods in the statement of the Periodicity lemma cannot be weakened.

Regularities

The powers of a word x are defined by $x^0 = \varepsilon$ and $x^i = x^{i-1}x$ for a positive integer i. The kth ***power*** of x is x^k. It is a ***square*** if k is a positive even integer and a ***cube*** if k is a positive multiple of 3.

The next lemma states a first consequence of the Periodicity lemma.

Lemma 2
For words x and y, $xy = yx$ if and only if x and y are (integer) powers of the same word. The same conclusion holds when there exist two positive integers k and ℓ for which $x^k = y^\ell$.

The proofs of the two parts of the lemma are essentially the same (in fact the conclusion derives from a more general statement on codes). For example, if $xy = yx$, both x and y are borders of the word, then both $|x|$ and $|y|$ are periods of it and $\gcd(|x|, |y|)$ as well by the Periodicity lemma. Since $\gcd(|x|, |y|)$ divides also $|xy|$, the conclusion follows. The converse implication is straightforward.

The non-empty word x is said to be ***primitive*** if it is not the power of any other word. That is to say x is primitive if $x = u^k$, for a word u and a positive integer k, implies $k = 1$ and then $u = x$. For example, `abaab` is primitive, while ε and `bababa` $= (\texttt{ba})^3$ are not.

It follows from Lemma 2 that a non-empty word has exactly one primitive word it is a power of. When $x = u^k$ and u is primitive, u is called the ***primitive root*** of x and k is its ***exponent***, denoted by $\exp(x)$. More generally, the exponent of x is the quantity $\exp(x) = |x|/per(x)$, which is not necessarily an integer, and the word is said to be ***periodic*** if its exponent is at least 2.

Note the number of conjugates of a word, the size of its ***conjugacy class***, is the length of its (primitive) root.

Another consequence of the Periodicity lemma follows.

Lemma 3 (Primitivity lemma, Synchronisation lemma)
A non-empty word x is primitive if and only if it is a factor of its square only as a prefix and as a suffix, or equivalently if and only if $per(x^2) = |x|$.

The picture illustrates the result of the lemma. The word `abbaba` is primitive and there are only two occurrences of it in its square. While `ababab` is not primitive and has 4 occurrences in its square.

The notion of ***run*** or ***maximal periodicity*** encompasses several types of regularities occurring in words. A run in the word x is a maximal occurrence of a periodic factor. To say it more formally, it is an interval $[i\,..\,j]$ of positions on x for which $\exp(x[i\,..\,j]) \geq 2$ and both $x[i-1\,..\,j]$ and $x[i\,..\,j+1]$ have periods larger than that of $x[i\,..\,j]$ when they exist. In this situation, since the occurrence is identified by i and j, we also say abusively that $x[i\,..\,j]$ is a run.

Another type of regularity consists in the appearance of reverse factors or of palindromes in words. The ***reverse*** or ***mirror image*** of the word x is the word $x^{\mathrm{R}} = x[|x|-1]x[|x|-2]\cdots x[0]$. Associated with this operation is the notion of ***palindrome***: a word x for which $x^{\mathrm{R}} = x$.

For example, `noon` and `testset` are English palindromes. The first is an even palindrome of the form uu^{R} while the second is an odd palindrome of the form uau^{R} with a letter a. Letter a can be replaced by a short word leading to the notion of gapped palindromes useful when

related to folding operations like those occurring in sequences of biological molecules. As another example, integers whose decimal expansion is an even palindrome are multiple of 11, like $1661 = 11 \times 151$ or $175571 = 11 \times 15961$.

Ordering

Some algorithms benefit from the existence of an ordering on the alphabet, denoted by $\leq$. The ordering induces the ***lexicographic ordering*** or ***alphabetic ordering*** on words as follows. It is denoted by $\leq$ like the alphabet ordering. For $x, y \in A^*$, $x \leq y$ if and only if, either x is a prefix of y or x and y can be decomposed as $x = uav$ and $y = ubw$ for words u, v and w, letters a and b, with $a < b$. Thus, $\mathtt{ababb} < \mathtt{abba} < \mathtt{abbaab}$ when considering $\mathtt{a} < \mathtt{b}$ and more generally the natural ordering on the alphabet $\mathbf{A}$.

We say that x is ***strongly less*** than y, denoted by $x \ll y$, when $x \leq y$ but x is not a prefix of y. Note that $x \ll y$ implies $xu \ll yv$ for any words u and v.

Concepts of ***Lyndon words*** and of ***necklaces*** are built from the lexicographic ordering.

A Lyndon word x is a primitive word that is the smallest among its conjugates. Equivalently but not entirely obvious, x is smaller than all its proper non-empty suffixes, and as such is also called a ***self-minimal word***. As a consequence, x is border-free. It is known that any non-empty word w factorises uniquely into $x_0 x_1 \cdots x_k$ where x_is are Lyndon words and $x_0 \geq x_1 \geq \cdots \geq x_k$. For example, the word $\mathtt{aababaabaaba}$ factorises as $\mathtt{aabab} \cdot \mathtt{aab} \cdot \mathtt{aab} \cdot \mathtt{a}$ where $\mathtt{aabab}$, $\mathtt{aab}$ and $\mathtt{a}$ are Lyndon words.

A necklace or ***minimal word*** is a word that is the smallest in its conjugacy class. It is an (integer) power of a Lyndon word. A Lyndon word is a necklace but, for example, the word $\mathtt{aabaab} = \mathtt{aab}^2$ is a necklace without being a Lyndon word.

Remarkable words

Besides Lyndon words, three sets of words have remarkable properties and are often used in examples. They are Thue-Morse words, Fibonacci words and de Bruijn words. The first two are prefixes of (one-way) infinite words. Formally an ***infinite word*** on the alphabet A is a mapping from natural numbers to A. Their set is denoted by A^∞.

The notion of (monoid) ***morphism*** is central to define some infinite sets of words or an associate infinite word. A morphism from A^* to itself (or another free monoid) is a mapping $h : A^* \mapsto A^*$ satisfying $h(uv) = h(u)h(v)$ for all words u and v. Consequently, a morphism is entirely defined by the images $h(a)$ of letters $a \in A$.

The ***Thue-Morse word*** is produced by iterating the ***Thue-Morse morphism*** μ from $\{\mathtt{a},\mathtt{b}\}^*$ to itself defined by

$$\begin{cases} \mu(\mathtt{a}) = \mathtt{ab}, \\ \mu(\mathtt{b}) = \mathtt{ba}. \end{cases}$$

Iterating the morphism from letter a gives the list of Thue-Morse words $\mu^k(\mathtt{a})$, $k \geq 0$, that starts with:

$$\begin{array}{lcl} \tau_0 = \mu^0(\mathtt{a}) & = & \mathtt{a} \\ \tau_1 = \mu^1(\mathtt{a}) & = & \mathtt{ab} \\ \tau_2 = \mu^2(\mathtt{a}) & = & \mathtt{abba} \\ \tau_3 = \mu^3(\mathtt{a}) & = & \mathtt{abbabaab} \\ \tau_4 = \mu^4(\mathtt{a}) & = & \mathtt{abbabaabbaababba} \\ \tau_5 = \mu^5(\mathtt{a}) & = & \mathtt{abbabaabbaababbabaababbaabbabaab} \end{array}$$

and eventually produces its infinite associate:

$$\mathbf{t} = \lim_{k\to\infty} \mu^k(\mathtt{a}) = \mathtt{abbabaabbaababbabaababbaabbabaab}\cdots.$$

An equivalent definition of Thue-Morse words is provided by the following recurrence:

$$\begin{cases} \tau_0 = \mathtt{a}, \\ \tau_{k+1} = \tau_k\overline{\tau_k}, \quad \text{for } k \geq 0. \end{cases}$$

where the bar morphism is defined by $\overline{\mathtt{a}} = \mathtt{b}$ and $\overline{\mathtt{b}} = \mathtt{a}$. Note the length of the kth Thue-Morse word is $|\tau_k| = 2^k$.

A direct definition of $\mathbf{t}$ is as follows: the letter $\mathbf{t}[n]$ is b if the number of occurrences of digit 1 in the binary representation of n is odd, and is a otherwise.

The infinite Thue-Morse word is known to contain no overlap (factor of the form $auaua$ for a letter a and a word u), that is, no factor of exponent larger than 2. It is said to be ***overlap-free***.

The ***Fibonacci word*** is similarly produced by iterating a morphism, the ***Fibonacci morphism*** ϕ from $\{\mathtt{a},\mathtt{b}\}^*$ to itself defined by

$$\begin{cases} \phi(\mathtt{a}) = \mathtt{ab}, \\ \phi(\mathtt{b}) = \mathtt{a}. \end{cases}$$

Iterating the morphism from letter a gives the list of Fibonacci words $\phi^k(\mathtt{a})$, $k \geq 0$, that starts with:

$$\begin{array}{lcl} fib_0 = \phi^0(\mathtt{a}) & = & \mathtt{a} \\ fib_1 = \phi^1(\mathtt{a}) & = & \mathtt{ab} \\ fib_2 = \phi^2(\mathtt{a}) & = & \mathtt{aba} \\ fib_3 = \phi^3(\mathtt{a}) & = & \mathtt{abaab} \\ fib_4 = \phi^4(\mathtt{a}) & = & \mathtt{abaababa} \\ fib_5 = \phi^5(\mathtt{a}) & = & \mathtt{abaababaabaab} \\ fib_6 = \phi^6(\mathtt{a}) & = & \mathtt{abaababaabaababaababa} \end{array}$$

and eventually its infinite associate:

$$\mathbf{f} = \lim_{k\to\infty} \phi^k(\mathtt{a}) = \mathtt{abaababaabaababaababaabaababaabaab}\cdots.$$

An equivalent definition of Fibonacci words comes from the recurrence relation:

$$\begin{cases} fib_0 = \mathtt{a}, \\ fib_1 = \mathtt{ab}, \\ fib_{k+1} = fib_k fib_{k-1}, \quad \text{for } k \geq 1. \end{cases}$$

The sequence of lengths of these words is the sequence of Fibonacci numbers, that is, $|fib_k| = F_{k+2}$. Recall that ***Fibonacci numbers*** are defined by the recurrence:

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_{k+1} = F_k + F_{k-1}, \quad \text{for } k \geq 1. \end{cases}$$

Among many properties they satisfy:

- $\gcd(F_n, F_{n-1}) = 1$, for $n \geq 2$,
- F_n is the nearest integer of $\Phi^n/\sqrt{5}$, where $\Phi = \frac{1}{2}(1+\sqrt{5}) = 1.61803\cdots$ is the ***golden ratio***.

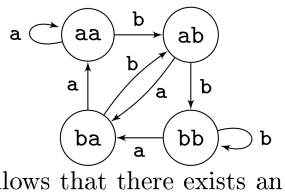
The interest in Fibonacci words comes from the combinatorial properties they satisfy and the large number of repeats they contain. However the infinite Fibonacci word contains no factor of exponent larger than $\Phi^2 + 1 = 3.61803\cdots$.

De Bruijn words are defined here on the alphabet $A = \{\mathtt{a}, \mathtt{b}\}$ and are parameterised by a positive integer k. A word $x \in A^+$ is a de Bruijn word of order k if each word of A^k occurs exactly once in x. A first example: `ab` and `ba` are the only two de Bruijn words of order 1. A second example: the word `aaababbbaa` is a de Bruijn word of order 3 since its eight factors of length 3 are the eight words of A^3, that is, `aaa`, `aab`, `aba`, `abb`, `baa`, `bab`, `bba`, and `bbb`.

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

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

The picture displays the automaton for de Bruijn words of order 3. Note that exactly two arcs exit each of the states, one labelled by `a`, the other by `b`; and that exactly two arcs enter each of the states, both labelled by the same letter. The graph associated with the automaton thus satisfies the Euler condition: every vertex has an even degree. It follows that there exists an



Eulerian circuit in the graph. Its label is a ***circular de Bruijn word***. Appending to it its prefix of length $k-1$ gives an ordinary de Bruijn word.

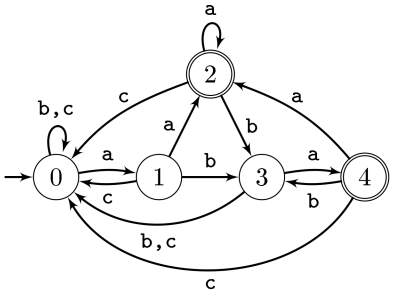
It can also be verified that the number of de Bruijn words of order k is exponential in k.

De Bruijn words can be defined on larger alphabets and are often used as examples of limit cases because they contain all the factors of a given length.

Automata

A finite ***automaton*** M on the finite alphabet A is composed of a finite set Q of ***states***, of an ***initial*** state q_0, of a set $T \subseteq Q$ of ***terminal*** states and of a set $F \subseteq Q \times A \times Q$ of ***labelled edges*** or ***arcs*** corresponding to state ***transitions***. We denote the automaton M by the quadruplet (Q, q_0, T, F) or sometimes by just (Q, F) when for example q_0 is implicit and $T = Q$. We say of an arc (p, a, q) that it leaves state p and enters state q; state p is the ***source*** of the arc, letter a its ***label***, and state q its ***target***. A graphic representation of an automaton is displayed below.

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 ***labelled successor*** of 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 word $a_0 a_1 \ldots a_{n-1}$, its ***origin*** the state p_0 and its ***end*** the state p'_{n-1}. 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 word is ***recognised*** or ***accepted*** by the automaton if it is the label of a successful path. The language composed of the words recognised by the automaton M is denoted by $Lang(M)$.

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 $q \in Q$ for which $(p, a, q) \in$

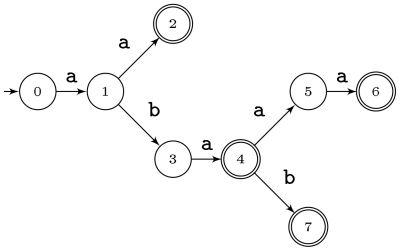
F. In such a case, it is natural to consider the ***transition function*** $\delta\colon Q \times A \to Q$ of the automaton defined for every arc $(p, a, q) \in F$ by $\delta(p, a) = q$ and undefined elsewhere. The function δ merely extends to words.

It is known that any language accepted by an automaton is also accepted by a deterministic automaton and that there is a unique (up to state naming) minimal deterministic automaton accepting it.

Trie

A ***trie*** $\mathcal{T}$ on the alphabet A, kind of digital tree, is an automaton whose paths from the initial state, the root, do not converge. A trie is used mostly to represent finite sets of words. If no word of the set is a prefix of another word of the set, words are associated with the leaves of the trie.

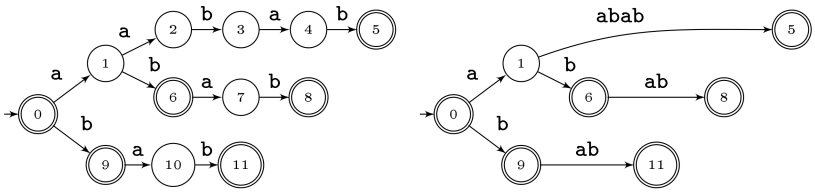
Below is the trie $\mathcal{T}(\{\texttt{aa}, \texttt{aba}, \texttt{abaaa}, \texttt{abab}\})$. States correspond to prefixes of words in the set. For example, state 3 corresponds to the prefix of length 2 of both `abaaa` and `abab`. Terminal states (doubly-circled) 2, 4, 6 and 7 correspond to the words in the set.



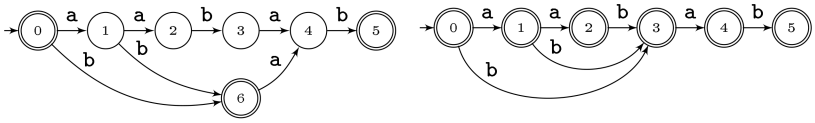
Suffix structures

Suffix structures that store the suffixes of a word are important data structures used to produce efficient indexes. Tries can be used as such but their size can be quadratic. One solution to cope with that is to compact the trie, resulting in the Suffix tree of the word. It consists in eliminating non-terminal nodes with only one outgoing edge and in labelling arcs by factors of the word accordingly. Eliminated nodes are sometimes called implicit nodes of the Suffix tree and remaining nodes called explicit nodes.

Below are the trie $\mathcal{T}(\mathit{Suff}(\texttt{aabab}))$ of suffixes of `aabab` (on the left) and its ***Suffix tree*** $\mathcal{ST}(\texttt{aabab})$ (on the right). To get a complete linear-size structure, each factor of the word that labels an arc needs to be represented by a pair of integers like (position, length).



A second solution to reduce the size of the Suffix trie is to minimise it, which means to consider the minimal deterministic automaton accepting the suffixes of the word, its ***Suffix automaton***. Below (left) is $\mathcal{S}(\texttt{aabab})$ the Suffix automaton of `aabab`.



It is known that $\mathcal{S}(x)$ possesses less than $2|x|$ states and less than $3|x|$ arcs, for a total size $O(|x|)$, i.e. linear in $|x|$. The Factor automaton $\mathcal{F}(x)$ of the word, minimal deterministic automaton accepting its factors, can even be smaller because all its states are terminal. Above (right) is the Factor automaton of `aabab` in which state 6 of $\mathcal{S}(\texttt{aabab})$ is merged with state 3.

Suffix array

The ***Suffix array*** of a word is also used to produce indexes but proceeds differently than with trees or automata. It consists primarily in sorting the suffixes of the word to allow binary search for its factors. To get actually efficient searches another feature is considered: the longest common prefixes of successive suffixes in the sorted list.

The information is stored in two arrays SA and LCP. The array SA is the inverse of the array Rank that gives the rank of each suffix attached at its starting position.

Below are the tables associated with the example word `aababa`. Its sorted list of suffixes is `a`, `aababa`, `aba`, `ababa`, `ba` and `baba` whose starting positions are 5, 0, 3, 1, 4 and 2. This latter list is stored in SA indexed by suffix ranks.

i	0	1	2	3	4	5
$x[i]$	a	a	b	a	b	a
Rank$[i]$	1	3	5	2	4	0

r	0	1	2	3	4	5	6	7	8	9	10	11	12
SA$[r]$	5	0	3	1	4	2							
LCP$[r]$	0	1	1	3	0	2	0	0	1	0	0	0	0

The table LCP essentially contains ***longest common prefixes*** stored

as maximal lengths of common prefixes between successive suffixes:

$$\mathrm{LCP}[r] = |lcp(x[\mathrm{SA}[r-1]\,..\,|x|-1], x[\mathrm{SA}[r]\,..\,|x|-1])|,$$

where *lcp* denotes the longest common prefix between two words. This gives $\mathrm{LCP}[0\,..\,6]$ for the example. Next values in $\mathrm{LCP}[7\,..\,12]$ correspond to the same information for suffixes starting at positions d and f when the pair (d, f) appears in the binary search. Formally, for such a pair, the value is stored at position $|x| + 1 + \lfloor (d+f)/2 \rfloor$. For example, in the above LCP array the value 1 corresponding to the pair $(0, 2)$, maximal length of prefixes between $x[5\,..\,5]$ and $x[3\,..\,5]$, is stored at position 8.

The table Rank is used in applications of the Suffix array mainly other than searching.

Compression

The most powerful ***compression*** methods for general texts are based either on the Ziv-Lempel factorisation of words or on easier techniques on top of the Burrows-Wheeler transform of words. We give a glimpse of both.

When processing a word on-line, the goal of ***Ziv-Lempel compression scheme*** is to capture information that has been met before. The associated factorisation of a word x is $u_0 u_1 \cdots u_k$ where u_i is the longest prefix of $u_i \cdots u_k$ that appears before this occurrence in x. When it is empty, the first letter of $u_i \cdots u_k$, which does not occur in $u_0 \cdots u_{i-1}$, is chosen. The factor u_i is sometimes called abusively the ***longest previous factor*** at position $|u_0 \cdots u_{i-1}|$ on x.

For example, the factorisation of the word `abaababababaaababb` is: `a` · `b` · `a` · `aba` · `baba` · `aabab` · `b`.

There are several variations to define the factors of the decomposition, here are a few of them. The factor u_i may include the letter immediately following the occurrence of the longest previous factor at position $|u_0 \cdots u_{i-1}|$, which amounts to extend a factor occurring before. Previous occurrences of factors may be chosen among the factors $u_0, \ldots, u_{i-1}$ or among all the factors of $u_0 \cdots u_{i-1}$ (to avoid an overlap between occurrences) or among all factors occurring before. This results in a large variety of text compression software based on the method.

When designing word algorithms the factorisation is also used to reduce some on-line processing by storing what has already been done on previous occurrences of factors.

The ***Burrows-Wheeler transform*** of a word x is a reversible mapping that transforms $x \in A^k$ into $\mathrm{BW}(x) \in A^k$. The effect is mostly to group together letters having the same context in x. The encoding proceeds as follows. Let us consider the sorted list of rotations (conjugates) of x. Then $\mathrm{BW}(x)$ is the word composed of the last letters of sorted rotations, referred to as the last column of the corresponding table.

For the example word `banana`, rotations are listed below on the left and their sorted list on the right. Then BW(`banana`) = `nnbaaa`.

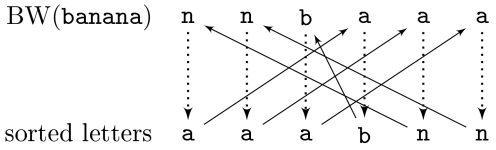
0	b	a	n	a	n	a	5	a	b	a	n	a	n
1	a	n	a	n	a	b	3	a	n	a	b	a	n
2	n	a	n	a	b	a	1	a	n	a	n	a	b
3	a	n	a	b	a	n	0	b	a	n	a	n	a
4	n	a	b	a	n	a	4	n	a	b	a	n	a
5	a	b	a	n	a	n	2	n	a	n	a	b	a

Two conjugate words have the same image by the mapping. Choosing the Lyndon word as a representative of the class of a primitive word, the mapping becomes bijective. To recover the original word x other than a Lyndon word, it is sufficient to keep the position on BW(x) of the first letter of x.

The main property of the transformation is that occurrences of a given letter are in the same relative order in BW(x) and in the sorted list of all letters. This is used to decode BW(x).

To do it on `nnbaaa` from the above example, we first sort the letters getting the word `aaabnn`. Knowing that the first letter of the initial word appears at position 2 on `nnbaaa` we can start the decoding: the first letter is `b` followed by letter `a` at the same position 2 on `aaabnn`. This is the third occurrence of `a` in `aaabnn` corresponding to its third occurrence in `nnbaaa`, which is followed by `n`, and so on.

The decoding process is similar to following the cycle in the graph below from the correct letter. Starting from a different letter produces a conjugate of the initial word.



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 referred to 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, word,

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 functions are local to them unless otherwise mentioned.
- The evaluation of boolean expressions is performed from left to right in a lazy way.
- Instructions of the form $(m_1, m_2, \ldots) \leftarrow (exp_1, exp_2, \ldots)$ abbreviate the sequence of assignments $m_1 \leftarrow exp_1$, $m_2 \leftarrow exp_2$, ...

Algorithm TRIE below is an example of how algorithms are written. It produces the trie of a dictionary X, finite set of words X. It successively considers each word of X during the **for** loop of lines 2–10 and inserts them into the structure letter by letter during execution of the **for** loop of lines 4–9. When the latter loop is over, the last considered state t, ending the path from the initial state and labelled by the current word, is set as terminal at line 10.

```
TRIE(X finite set of words)
 1  M ← NEW-AUTOMATON()
 2  for each string x ∈ X do
 3       t ← initial(M)
 4       for each letter a of x, sequentially do
 5            p ← TARGET(t, a)
 6            if p = NIL then
 7                 p ← NEW-STATE()
 8                 Succ[t] ← Succ[t] ∪ {(a, p)}
 9            t ← p
10       terminal[t] ← TRUE
11  return M
```

Notes

Basic elements on words introduced in this section follow their presentation in [74]. They can be found in other textbooks on text algorithms, like those by Crochemore and Rytter [96], by Gusfield [134], by Crochemore and Rytter [98] and by Smyth [228]. The notions are also introduced in some textbooks dealing with the wider topics of combinatorics on words, like those by Lothaire [175, 176, 177], or in the tutorial by Berstel and Karhumäki [34].