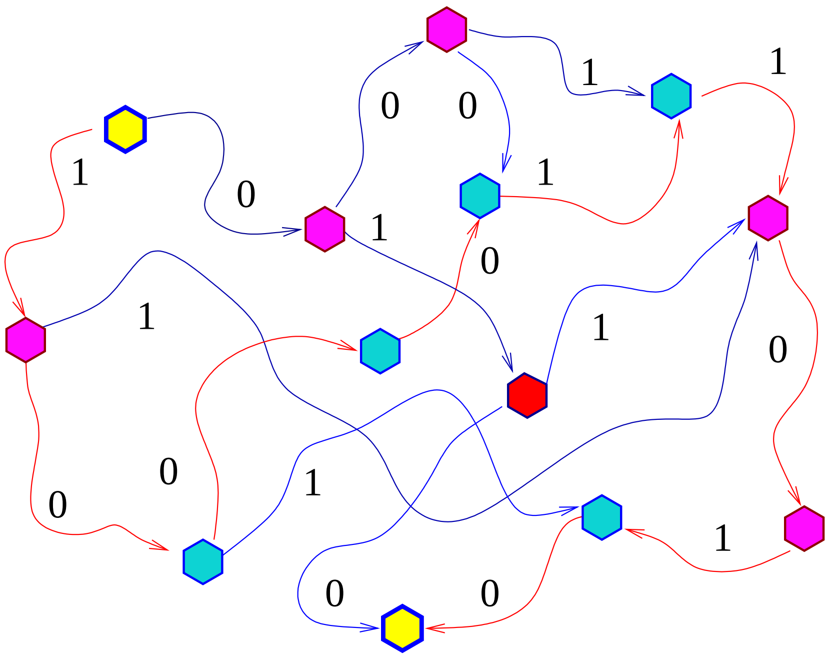
JEWELS OF STRINGOLOGY

Maxime Crochemore, Wojciech Rytter



World Scientific

Preface

The term *stringology* is a popular nickname for *string algorithms* as well as for *text algorithms*. Usually *text* and *string* have the same meaning. More formally, a text is a sequence of symbols. Text is one of the basic data types to carry information. This book is a collection of the most beautiful and at the same time very classical algorithms on strings. The selection has been done by the authors, and is rather personal, among so many famous algorithms that were natural candidates to be included and that belong to a field that has become now fairly popular.

One can partition algorithmic problems discussed in this book into practical and theoretical problems. Certainly string matching and data compression are in the first class, while most problems related to symmetries and repetitions are in the second. However, we believe that all the problems are interesting from an algorithmic point of view and enable the reader to appreciate the importance of combinatorics on words.

In most textbooks on algorithms and data structures the presentation of efficient algorithms on words is quite short as compared to issues in graph theory, sorting, searching, and some other areas. At the same time, there are many presentations of interesting algorithms on words accessible only in journals and in a form directed mainly at specialists. There are still not many books on text algorithms, especially the books which are oriented toward undergraduate and graduate students. In the book the difficult parts are indicated by a star, so the basic text becomes painless for undergraduate students. We hope that this book will cover a gap on algorithms on words in book literature for the broader audience, and bring together the many results presently dispersed in the masses of journal articles.

March 2002

M. Crochemore, W. Rytter

Contents

Chapter 1

Stringology

One of the simplest and natural types of information representation is by means of written texts. This type of data is characterized by the fact that it can be written down as a long sequence of characters. Such linear a sequence is called a *text*. The texts are central in "word processing" systems, which provide facilities for the manipulation of texts. Such systems usually process objects that are quite large. For example, this book probably contains more than a million characters. Text algorithms occur in many areas of science and information processing. Many text editors and programming languages have facilities for processing texts. In biology, text algorithms arise in the study of molecular sequences. The complexity of text algorithms is also one of the central and most studied problems in theoretical computer science. It could be said that it is the domain in which practice and theory are very close to each other.

The basic textual problem in stringology is called *pattern matching*. It is used to access information and, no doubt, at this moment many computers are solving this problem as a frequently used operation in some application system. Pattern matching is comparable in this sense to sorting, or to basic arithmetic operations.

Consider the problem of a reader of the French dictionary "Grand Larousse," who wants all entries related to the name "Marie-Curie-Sklodowska." This is an example of a pattern matching problem, or string matching. In this case, the name "Marie-Curie-Sklodowska" is the pattern. Generally we may want to find a string called a *pattern* of length m inside a text of length n, where n is greater than m. The pattern can be described in a more complex way to denote a set of strings and not just a single word. In many cases n is very large. In genetics the pattern can correspond to a gene that can be very long; in image processing, digitized images sent serially contain millions of characters each. The string-matching problem is the basic question considered in this book, together with its variations. String matching is also the basic subproblem in other algorithmic problems on texts. Following is a (not exclusive) list of basic groups of problems discussed in this book:

- variations on the string-matching problem
- problem related to the structures of the segments of a text
- data compression
- approximation problems
- finding regularities
- extensions to two-dimensional images
- extensions to trees
- optimal time-space implementations
- optimal parallel implementations.

The formal definition of string matching and many other problems is given in the next chapter. We now introduce some of them informally in the context of applications.

1.1 Text file facilities

The UNIX system uses text files for exchanging information as a main feature. The user can get information from the files and transform them through different existing commands. The tools often behave as filters that read their input once and produce the output simultaneously. These tools can easily be connected with each other, particularly through the pipelining facility. This often reduces the creation of new commands to a few lines of already existing commands.

One of these useful commands is *grep*, acronym of “general regular expression print.” An example of the format of *grep* is

```
grep Marie-Curie-Sklodowska Grand-Larousse
```

provided “Grand-Larousse” is a file on your computer. The output of this command is the list of lines from the file that contains an occurrence of the name “Marie-Curie-Sklodowska.” This is an instance of the **string-matching** problem. Another example with a more complex pattern can be

```
grep '^Chapter [0-9]' Book
```

to list the titles of a book assuming titles begin with “Chapter” followed by a number. In this case the pattern denotes a set of strings (even potentially infinite), and not simply one string. The notation to specify patterns is known as *regular expressions.* This is an instance of the **regular-expression-matching** problem.

The indispensable complement of *grep* is *sed* (stream editor). It is designed to transform its input. It can replace patterns of the input with specific strings. Regular expressions are also available with *sed.* But the editor contains an even more powerful notation. This allows, for example, the action on a line of the input text containing the same word twice. It can be applied to delete two consecutive occurrences of a same word in a text. This is simultaneously an example of the **repetition-finding** problem, **pattern-matching** problem and, more generally, the problem of finding **regularities in strings.**

The very helpful matching device based on regular expressions is omnipresent in the UNIX system. It can be used inside text editors such as *ed* and *vi*, and generally in almost all UNIX commands. The above tools, *grep* and *sed*, are based on this mechanism. There is even a programming language based on pattern-matching actions. It is the *awk* language, where the name *awk* comes from the initials of the authors, Aho, Weinberger, and Kernighan. A simple *awk* program is a sequence of pattern-action statements:

```
pattern1  {action 1}
pattern2  {action 2}
pattern3  {action 3}
```

The basic components of this program are patterns to be found inside the lines of the current file. When a pattern is found, the corresponding action is applied to the line. Therefore, several actions may be applied sequentially to a same line. This is an example of the **multi-pattern matching** problem. The language *awk* is meant for converting data from one form to another form, counting things, adding up numbers, and extracting information for reports. It contains an implicit input loop, and the pattern-action paradigm often eliminates control flow. This also frequently reduces the size of a program to a few statements. For instance, the following *awk* program prints the number of lines of the input that contain the word “`abracadabra`”:

```
abracadabra  {count++}
END          {print count}
```

The pattern “`END`” matches the end of input file, so that the result is printed after the input has been entirely processed. The language contains attractive features that strengthen the simplicity of the pattern-matching mechanism, such as default initialization for variables, implicit declarations, and associative arrays providing arbitrary kinds of subscripts. All this makes *awk* a convenient tool for rapid prototyping. The *awk* language can be considered as a generalization of another UNIX tool, *lex*, aimed at producing lexical analyzers. The input of a *lex* program is the specification of a lexical analyzer by means of regular expressions (and a few other possibilities). The output is the source of the specified lexical analyzer in the C programming language. A specification in *lex* is mainly a sequence of pattern-action statement as in *awk*. Actions are pieces of C code to be inserted in the lexical analyzer. At run time, these pieces of code execute the action corresponding to the associated pattern, when found. The following line is a typical statement of a *lex* program:

```
    [A-Za-z]+([A-Za-z0-9])*  { yyval = Install(); return(ID);}
```

The pattern specifies identifiers, that is, strings of characters starting with one letter and containing only letters and digits. This action leads the generated lexical analyzer to store the identifier and to return the string type "ID" to the calling parser. It is another instance of the regular expression-matching problem. The question of constructing **pattern-matching automata** is an important component having a practical application in the *lex* software.

Texts such as books or programs are likely to be changed during elaboration. Even after their completion they often support periodic upgrades. These questions are related to **text comparisons.** Sometimes we also wish to find a string, and do not completely remember it. The search has to be performed with an entirely non-specified pattern. This is an instance of the **approximate pattern matching.** Keeping track of all consecutive versions of a text may not be helpful because the text can be very long and changes may be hard to find. The reasonable way to control the process is to have an easy access to differences between the various versions. There is no universal notion as to what the differences are, or conversely, what the similarities are, between two texts. However, it can be agreed that the intersection of the two texts is the longest common subtext of both. In our book this is called the **longest common subsequence** problem, so that the differences between the two texts are the respective complements of the common part. The UNIX command *diff* builds on this notion. An option of the command *diff* produces a sequence of *ed* instructions to transform one text into the other. The similarity of texts can be measured as the minimal number of edit operations to transform one text into the other. The computation of such a measure is an instance of the **edit distance** problem.

1.2 Dictionaries

The search of words or patterns in static texts is quite a different question than the previous pattern-matching mechanism. Dictionaries, for example, are organized in order to speed up the access to entries. Another example of the same question is given by indexes. Technical books often contain an index of chosen terms that gives pointers to parts of the text related to words in the index. The algorithms involved in the creation of an index form a specific group. The use of dictionaries or lexicons is often related to natural language processing. Lexicons of programming languages are small, and their representation is not a difficult problem during the development of a compiler. To the contrary, English contains approximately 100,000 words, and even twice that if inflected forms are considered. In French, inflected forms produce more than 700,000 words. The representation of lexicons of this size makes the problem a bit more challenging.

A simple use of dictionaries is illustrated by spelling checkers. The UNIX command, *spell*, reports the words in its input that are not stored in the lexicon. This rough approach does not yield a pertinent checker, but, practically, it helps to find typing errors. The lexicon used by *spell* contains approximately 70,000 entries stored within less than 60 kilobytes of random-access memory. Quick access to lexicons is a necessary condition for producing good parsers. The data structure useful for such access is called an index. In our book indexes correspond to data structures representing all factors of a given (presumably long) text. We consider problems related to the construction of such structures: **suffix trees, directed acyclic word graphs, factor automata, suffix arrays**. The *PAT* tool developed at the NOED Center (Waterloo, Canada) is an implementation of one of these structures tailored to work on large texts. There are several applications that effectively require some understanding of phrases in natural languages, such as data retrieval systems, interactive software, and character recognition.

An image scanner is a kind of photocopier. It is used to give a digitized version of an image. When the image is a page of text, the natural output of the scanner must be in a digital form available to a text editor. The transformation of a digitized image of a text into a usual computer representation of the text is realized by an Optical Character Recognition (OCR). Scanning a text with an OCR can be 50 times faster than retyping the text on a keyboard. Thus, OCR softwares are likely to become more common. But they still suffer from a high degree of imprecision. The average rate of error in the recognition of characters is approximately one percent. Even if this may happen to be rather small, this means that scanning a book produces approximately one error per line. This is compared with the usually very high quality of texts checked by specialists. Technical improvements on the hardware can help eliminate certain kinds of errors occurring on scanned texts in printed forms. But this cannot alleviate the problem associated with recognizing texts in printed forms. Reduction of the number of errors can thus only be achieved by considering the context of the characters, which assumes some understanding of the structure of the text. Image processing is related to the problem of **two-dimensional pattern matching**. Another related problem is the data structure for all subimages, which is discussed in this book in the context of the **dictionary of basic factors.**

The theoretical approach to the representation of *lexicons* is either by means of trees or finite state automata. It appears that both approaches are equally efficient. This shows the practical importance of the **automata theoretic approach** to text problems. At LITP (Paris) and IGM (Marne-la-Vallée) we have shown that the use of automata to represent lexicons is particularly efficient. Experiments have been done on a 700,000 word lexicon of LADL (Paris). The representation supports direct access to any word of the lexicon and takes only 300 kilobytes of random-access memory.

1.3 Data compression

One of the basic problems in storing a large amount of textual information is the **text compression** problem. Text compression means reducing the representation of a text. It is assumed that the original text can be recovered from its compressed from. No loss of information is allowed. Text compression is related to the **Huffman coding problem** and the **factorization problem.** This kind of compression contrast with other kinds of compression techniques applied to sounds or images, in which approximation is acceptable. Availability of large mass storage does not decrease the interest for compressing data. Indeed, users always take advantage of extra available space to store more data or new kinds of data. Moreover, the question remains important for storing data on secondary storage devices. Examples of implementations of dictionaries reported above show that **data compression** is important in several domains related to natural language analysis. Text compression is also useful for telecommunications. It actually reduces the time to transmit documents via telephone network, for example. The success of Facsimile is perhaps to be credited to compression techniques.

General compression methods often adapt themselves to the data. This phenomenon is central in achieving high compression ratios. However, it appears, in practice, that methods tailored for specific data lead to the best results. We have experimented with this fact on data sent by geostationary satellites. The data have been compressed to seven percent of their original size without any loss of information.

The compression is very successful if there are redundancies and regularities in the information message. The analysis of data is related to the problem of **detecting regularities in texts**. Efficient algorithms are particularly useful to expertise the data.

1.4 Applications of text algorithms in genetics

Molecules of nucleic acids carry a large segment of information about the fundamental determinants of life, and, in particular, about the reproduction of cells. There are two types of nucleic acids known as desoxyribonucleic acid (DNA) and ribonucleic acid (RNA). DNA is usually found as double-stranded molecules. In vivo, the molecule is folded up like a ball of string. The skeleton of a DNA molecule is a sequence on the four-letter alphabet of nucleotides: adenine (A), guanine (G), cytosine (C), and thymine (T). RNA molecules are usually single-stranded molecules composed of ribonucleotides: A, G, C, and uracil (U).

Processus of "transcription" and "translation" lead to the production of proteins, which also have a string composed of 20 amino acids as a primary structure. In a first approach all these molecules can be viewed as texts. The discovery twenty years ago of powerful sequencing techniques has led to a rapid accumulation of sequence data. From the collection of sequences up to their analysis many algorithms on texts are implied. Moreover, only fast algorithms are often feasible because of the huge amount of data involved.

Collecting sequences can be accomplished through audioradiography gels. The automatic transcription of these gels into sequences is a typical **two dimensional pattern-matching** problem in two dimensions. The reconstruction of a whole sequence from small segments, used for instance in the shotgun sequencing method, is another example of a problem that occurs during this step. This problem is called the **shortest common superstring problem**: construction of the shortest text containing several given smaller texts.

Once a new sequence is obtained, the first important question to ask is whether it resembles any other sequence already stored in data banks. Before adding a new molecular sequence into an existing data base one needs to know whether or not the sequence is already present. The comparison of several sequences is usually realized by writing one over another. The result is know as an alignment of the set of nucleotides. Alignment of two sequences is the **edit distance problem**: compute the minimal number of edit operations to transform one string into another. It is realized by algorithms based on dynamic programming techniques similar to the one used by the UNIX command *diff*.

The problem of the **longest common subsequence** is a variation of the alignment of sequences. A tool, called *agrep*, developed at the University of Arizona, is devoted to these questions, related to **approximate string matching**.

Further questions about molecular sequences are related to their analysis. The aim is to discover the functions of all parts of the sequence. For example, DNA sequences contain important regions (coding sequences) for the production of proteins inside the cell. However, no good answer is presently known for finding all coding sequences of a DNA sequence. Another question about sequences is the reconstruction of their three-dimensional structure. It seems that a part of the information resides in the sequence itself. This is because, during the folding process of DNA, for example, nucleotides match pairwise (A with T, and C with G). This produces approximate palindromic symmetries (as TTAGCGGCTAA). Involved in all these questions are **approximate searches** for specific patterns, for **repetitions**, for **palindromes**, or other **regularities**.

1.5 Efficiency of algorithms

Efficient algorithms can be classified according to what is meant by efficiency. There exist different notions of efficiency depending on the complexity measure involved. Several such measures are discussed in this book: sequential time, memory space, parallel time, and number of processors.

This book deals with "feasible" problems. We can define them as problems having efficient algorithms, or as solvable in time bounded by a small-degree polynomial. In the case of sequential computations we are interested in lowering the degree of the polynomial corresponding to time complexity. The most efficient algorithms usually solve a problem in linear-time complexity. We are also interested in space complexity. Optimal space complexity often means a constant number of (small integer) registers in addition to input data. Therefore, we say that an algorithm is time-space optimal if it works simultaneously in linear time and in constant extra space. These are the most advanced sequential algorithms, and also the most interesting, both from a practical and theoretical point of view.

In the case of parallel computations we are generally interested in the parallel time $T(n)$ as well as in the number of processors $P(n)$ required for the executions of the parallel algorithm on data of size n. The total number of elementary operations performed by the parallel algorithm is not greater than the product $T(n)P(n)$.

Efficient parallel algorithms are those that operate in no more than polylogarithmic (a polynomial of logs of input size) time with a polynomial number of processors. The class of problems solvable by such algorithms is denoted by NC and hence we call the related algorithms NC-algorithms. An NC-algorithm is optimal if the total number of operations $T(n)P(n)$ is linear. Another possible definition is that this number is essentially the same as the time complexity of the best known sequential algorithm solving the given problem. However, we adopt the first option here because algorithms on strings usually have a time complexity which is at least linear.

Precisely evaluating the complexity of an algorithm according to some measure is often difficult, and, moreover, it is unlikely to be of much use. The "big O" notation clarifies what the important terms of a complexity expression are. It estimates the asymptotic order of the complexity of an algorithm and helps compare algorithms between each others. Recall that if f and g are two functions from and to integers, then we say that $f = O(g)$ if $f(n) < C.g(n)$ when $n > N$, for some constants C and N. We write $f = \Theta(g)$ when the functions f and g are of the same order, which means that both equalities $f = O(g)$ and $g = O(f)$ hold.

Comparing functions through their asymptotic orders leads to these kinds of inequalities: $O(n^{0.7}) < O(n) < O(n \log n)$, or $O(n^{\log n}) < O(\log n^n) < O(n!)$.

Within sequential models of machines one can distinguish further types of computations: off-line, on-line and real-time. These computations are also related to efficiency. It is understood that real-time computations are more efficient than general on-line, and that on-line computations are more efficient than off-line. Each algorithm is an off-line algorithm: "off-line" conceptually means that the whole input data can be put into the memory before the actual computation starts. We are not interested then in the intermediate results computed by the algorithm, but only in the final result (though this final result can be a sequence or a vector). The time complexity is measured by the total time from the moment the computation starts (with all input data previously memorized) up to the final termination. In contrast, an on-line algorithm is like a sequential transducer. The portions of the input data are "swallowed" by the algorithm step after step, and after each step an intermediate result is expected (related to the input data read so far). It then reads the next portion of the input, and so on. In on-line algorithms the input can be treated as an infinite stream of data, consequently we are not interested mainly in the termination of the algorithm for all such data. The main interest for us is the total time $T(n)$ for which we have to wait to get the n-th first outputs. The time $T(n)$ is measured starting at the beginning of the whole computation (activation of the transducer). Suppose that the input data is a sequence and that after reading the n-th symbol we want to print "1" if the text read to this moment contains a given pattern as a suffix, otherwise we print "0". Hence we have two streams of data: the stream of input symbols and an output stream of answers "1" or "0". The main feature of the on-line algorithm is that it has to give an output value before reading the next input symbol. The real-time computations are those on-line algorithms that are in a certain sense optimal; the elapsing time between reading two consecutive input symbols (the time spent for computing only the last output value) should be bounded by a constant. Most linear on-line algorithms are in fact real-time algorithms.

We are primarily interested in off-line computations in which the worst-case running time is linear, but on-line and real-time computations, as well as average complexities are also discussed in this book.

1.6 Some notation and formal definitions

Let A be an input *alphabet*–a finite set of symbols. Elements of A are called the *letters*, the *characters*, or the *symbols*. Typical examples of alphabets are: the set of all ordinary letters, the set of binary digits, or the set of 256 8-bit ASCII symbols. Texts (also called words or strings) over A are finite sequences of elements of A. The length (size) of a text is the number of its elements (with repetitions). Therefore, the length of *aba* is 3. The length of a word x is denoted by $|x|$. The input data for our problems will be words, and the size n of the input problem will usually be the length of the input word. In some situations, n will denote the maximum length or the total length of several words if the input of the problem consists of several words.

The i-th element of the word x is denoted by $x[i]$ and i is its *position* on x. We denote by $x[i\,.\,.j]$ the factor $x[i]x[i+1]\ldots x[j]$ of x. If $i > j$, by convention, the word $x[i\,.\,.j]$ is the empty word (the sequence of length zero), which is denoted by ε.

We say that the word x of length m is a factor (also called a subword) of the word y if $x = y[i+1\,.\,.i+n]$ for some integer i. We also say that x *occurs in* y *at position* i, or that the position i is a *match* for x in y.

We define the notion of *subsequence* (sometimes called a subword). The word x is a subsequence of y if x can be obtained from y by removing zero or more (not necessarily adjacent) letters from it. Likewise, x is a subsequence of y if $x = y[i_1]y[i_2]\ldots y[i_m]$, where $i_1, i_2, \ldots, i_m$ is an increasing sequence of indices on y.

Next we define formally the basic problem covered in this book. We often consider two texts *pat* (the pattern) and *text* of respective lengths m and n.

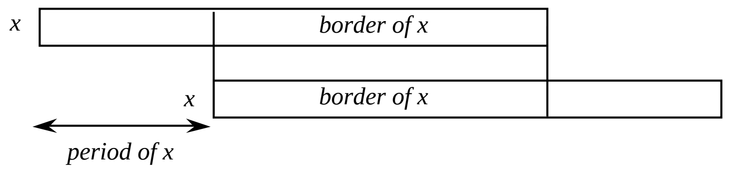


Figure 1.1: Duality between periods and borders of texts.

String matching (the basic problem). Given texts *pat* and *text*, verify if *pat* occurs in *text*. This is a decision problem: the output is a Boolean value. It is usually assumed that $m \leq n$. Therefore, the size of the problem is n. A slightly advanced version entails searching for all occurrences of *pat* in *text*, that is, computing the set of positions of *pat* in *text*. Let us denote this set by $MATCH(pat, text)$. In most cases an algorithm computing $MATCH(pat, text)$ is a trivial modification of a decision algorithm, this is the reason why we sometimes present only decision algorithms for string matching.

Instead of just one pattern, one can consider a finite set of patterns and ask if a given text contains a pattern from the set. The size of the problem is now the total length of all patterns plus the length of the text.

1.7 Some simple combinatorics of strings

The main theoretical tools in string-matching algorithms are related to mathematical properties of periodicities in strings. We define the notion of period of a word, which is central in almost all strings matching algorithms. A *period* of a word x is an integer p, $0 < p \leq |x|$, such that

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

for all $i \in \{1, \ldots, |x| - p\}$. When there is no ambiguity, we also say that the word $x[1..p]$ is a period of x. This is the usual definition of a period for a function defined on integers, as x can be viewed. Note that the length of a word is always a period of it, so that any word has at least one period. We denote by $period(x)$ the smallest period of x. We additionally say that x is *periodic* if $period(x) \leq |x|/2$.

The notion of *border* of a text is a dual notion to that of period, see Figure 1.1. A border of x is any word that is simultaneously a prefix and a suffix of x. Observe that x and the empty string ε are borders of x.

Let us denote by $Border(x)$ the longest nontrivial border (not the whole

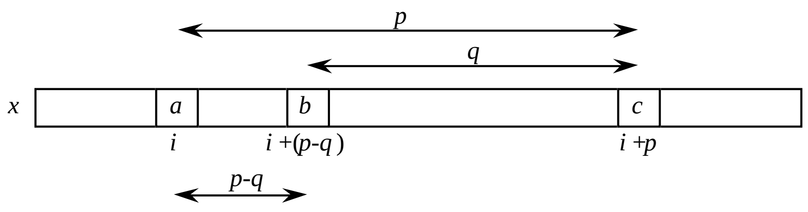


Figure 1.2: Quantity $p - q$ is also a period because letters a and b are both equal to letter c.

word) of x. Note that

$$(|x| - |Border(x)|, |x| - |Border^2(x)|, \ldots, |x| - |Border^k(x)|)$$

is the sequence of all periods of x in increasing order (k is the smallest integer for which $Border^k(x)$ is the empty word).

Example. The periods of *aabaaabaa* (of length 9) are 4, 7, 8 and 9. Its corresponding proper borders are *aabaa*, *aa*, *a*, ε.

Periodicity Lemma

Let x be a non-empty word and p be an integer such that $0 < p \leq |x|$. Then each of the following conditions equally defines p as a period of x:

1. x is a factor of some word y^k with $|y| = p$ and $k > 0$,
2. x may be written $(uv)^k$ with $|uv| = p$, v a non-empty word, and $k > 0$,
3. for some words y, z and w, $x = yw = wz$ and $|y| = |z| = p$.

Lemma 1.1 [Periodicity Lemma] *Let p and q be two periods of the word x. If $p + q < |x|$, then* $\gcd(p, q)$ *is also a period of x.*

Proof. The conclusion trivially holds if $p = q$. Assume now that $p > q$. First we show that the condition $p + q < |x|$ implies that $p - q$ is a period of x. Let $x = x[1]x[2]\ldots x[n]$ ($x[i]$'s are letters). Given $x[i]$ the i-th letter of x, the condition implies that either $i - q \geq 1$ or $i + p \leq n$. In the first case, q and p being periods of x, $x[i] = x[i-q] = x[i-q+p]$. In the second case, for the same reason, $x[i] = x[i+p] = x[i+p-q]$. Thus $p - q$ is a period of x. This situation is shown in Figure 1.2. The rest of the proof, left to the reader, is by induction on the integer $\max(p, q)$, after noting that $\gcd(p, q)$ equals $\gcd(p - q, q)$. $\square$

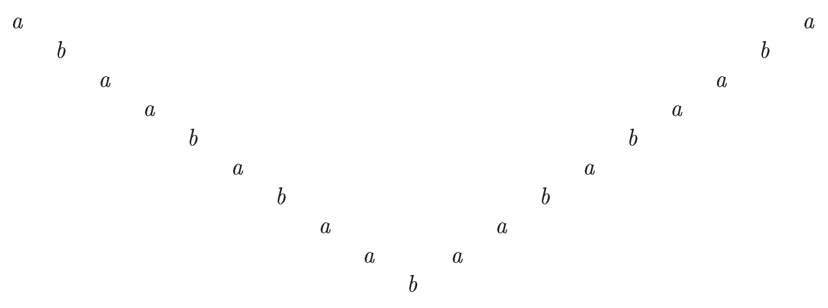


Figure 1.3: After cutting off its last two letters, Fib_8 is a symmetric word, a *palindrome.* This is not accidental.

There is a stronger version of the periodicity lemma for which we omit the proof.

Lemma 1.2 [Strong Periodicity Lemma] *If p and q are two periods of a word x such that $p + q - \gcd(p,q) \leq |x|$, then $\gcd(p,q)$ is also a period of x.*

An interesting family: Fibonacci words

Fibonacci words form an interesting family of words (from the point of view of periodicities). In sone sense, the inequality that appears in Strong Periodicity Lemma is optimal. The example supporting this claim is given by the *Fibonacci words* with the last two letters deleted.

Let Fib_n be the n-th Fibonacci word ($n \geq 0$). It is defined by

$$Fib_0 = \varepsilon,\ Fib_1 = b,\ Fib_2 = a, \text{ and } Fib_n = Fib_{n-1}Fib_{n-2}, \text{ for } n > 2.$$

Fibonacci words satisfy a large number of interesting properties related to periods and repetitions. Note that Fibonacci words (except the first two words of the sequence) are prefixes of their successors. Indeed, there is an even stronger property: the square of any Fibonacci word of high enough rank is a prefix of its succeeding Fibonacci words. Among other properties of Fibonacci words, it must be noted that they have no factor in the form u^4 (u non empty word) and they are almost symmetric, see Figure 1.3. Therefore, Fibonacci words contain a large number of periodicities, but none with an exponent higher than 3.

The lengths of Fibonacci words are the well-known Fibonacci numbers, $f_0 = 0$, $f_1 = 1$, $f_2 = 1$, $f_3 = 2$, $f_4 = 3$, ... The first Fibonacci words of the sequence ($Fib_n, n > 2$) are

$$\begin{array}{ll}
Fib_3 = ab, & |Fib_3| = 2,\\
Fib_4 = aba, & |Fib_4| = 3,\\
Fib_5 = abaab, & |Fib_5| = 5,\\
Fib_6 = abaababa, & |Fib_6| = 8,\\
Fib_7 = abaababaabaab, & |Fib_7| = 13,\\
Fib_8 = abaababaabaababaababa, & |Fib_8| = 21,\\
Fib_9 = abaababaabaababaababaabaababaabaab, & |Fib_9| = 34.
\end{array}$$

1.8 Some other interesting strings

Fibonacci words of rank greater than 1 can be treated as prefixes of a single infinite Fibonacci string Fib_∞. Similarly we can define the words of Thue-Morse $T(n)$ as prefixes of a single infinite word T_∞. Assume we count positions on this word starting from 0. Denote by $g(k)$ the number of "1" in the binary representation of the number k. Then

$$T_\infty(k) = \begin{cases} a & \text{if } g(k) \text{ is even,} \\ b & \text{otherwise.} \end{cases}$$

The Thue-Morse words T_n are the prefixes of T_∞ of length 2^n. We list several of them below.

$$\begin{array}{l}
T_1 = ab,\\
T_2 = abba,\\
T_3 = abbabaab,\\
T_4 = abbabaabbaababba,\\
T_9 = abbabaabbaababbabaababbaabbabaab.
\end{array}$$

These words have the remarkable property of being *overlap-free*, which means that there is no nonempty word x that occurs in them at two positions which distance is smaller than $|x|$. However these words are mostly known for the following *square-free* property: they contain no nonempty word in the form xx (nor, indeed, in the form $axaxa$, $a \in A$).

Let us define the following invertible encoding:

$$\beta(a) = a,\ \beta(b) = ab,\ \text{and}\ \beta(c) = abb.$$

Lemma 1.3 *For each integer n the word $\beta^{-1}(T_n)$ is square free.*

The lemma says in particular that there are infinitely many "square-free" words. Let T'_∞ be the word over the alphabet $\{0, 1, 2\}$ which symbols are the number of occurrences of letter "b" between two consecutive occurrences of letter "a" in T_∞. Then such an infinite word is also "square-free". We have

$$T'_\infty = 2\ 1\ 0\ 2\ 0\ 1\ 2\ \ldots$$

Other interesting words are sequences of moves in the *Hanoi towers* game. There are six possible moves depending from which stack to which other stack an disk is moved. If we have n disks then the optimal sequence consists of $2^n - 1$ moves and forms a word H_n. The interesting property of these words is that all of them are "square-free".

Yet another family of words that has a strange relation to numbers $g(k)$ is given by the binary words P_n, where P_n is the n-th row of the Pascal triangle modulo 2. In other words:

$$P_n(i) = \binom{n}{i} \mod 2.$$

We list below some of these words.

$$\begin{array}{lllllll} P_0 & = & 1 \\ P_1 & = & 1 & 1 \\ P_2 & = & 1 & 0 & 1 \\ P_3 & = & 1 & 1 & 1 & 1 \\ P_4 & = & 1 & 0 & 0 & 0 & 1 \\ P_5 & = & 1 & 1 & 0 & 0 & 1 & 1 \end{array}$$

The word P_n has the following remarkable property: the number of "1" in P_n equals $2^{g(n)}$.

Let us consider the infinite string W which symbols are digits and which results from concatenating all consecutive natural numbers written in decimal. Hence,

$$W = 0123456789101112131415161718192021222324252627282930313 2\ldots$$

Denote by W_n the prefix of W of size n. For a word x, let us denote by $occ_n(x)$ the number of occurrences of x in W_n. The words W_n have the following interesting property: for every two nonempty words x and y of a same length

$$\lim_{n\to\infty} \frac{occ_n(x)}{occ_n(y)} = 1.$$

This means, in a certain sense, that the sequence W is quite *random*.

An interesting property of strings is how many factors of a given length k they contain. Assume the alphabet is $\{a,\ b\}$. For a given k we have 2^k different words of length k. A natural question is:

what is the minimal length $\gamma(k)$ of a word containing each subword of length k.

Obviously $\gamma(k) \geq 2^k + k - 1$, since any shorter word has less than 2^k factors. It happens that $\gamma(k) = 2^k + k - 1$. The corresponding words are called de Bruijn words. In these strings each word of length k occurs exactly once. For a given k there are exponentially many de Bruijn words. For example for $k = 1$ we can take ab, for $k = 2$ we take $aabba$ or $abaab$ and for $k = 3$ we can take de Bruijn word $aaababbbaa$.

There is an interesting relation of de Bruijn words to Euler cycles in special graphs G_k. The nodes of G_k are all words of length $k - 1$ and for any word $x = a_1 a_2 \ldots a_{k-1}$ of length $k - 1$ we have two directed edges

$$a_1 a_2 \ldots a_{k-1} \xrightarrow{a} a_2 \ldots a_{k-1} \cdot a, \qquad a_1 a_2 \ldots a_{k-1} \xrightarrow{b} a_2 \ldots a_{k-1} \cdot b$$

The graph has a directed Euler cycle (containing each edge exactly once). Let $a_1 a_2 \ldots a_N$ be the sequence of labels of edges in a Euler cycle. Observe that $N = 2^k$. As de Bruijn word we can take the word:

$$a_1 a_2 \ldots a_N a_1 a_2 \ldots a_{k-1}.$$

1.9 Cyclic shifts and primitive words

A cyclic shift of x is any word vu, when x can be written in the form uv. Let us consider how many different cyclic shifts a word can have.

Example. Consider the cyclic shifts of the word $abaaaba$ of length 7. There are exactly 7 different cyclic shifts of $abaaaba$, the 8-th shift goes back to the initial word.

a	b	a	a	a	b	a							
	b	a	a	a	b	a	a						
		a	a	a	b	a	a	b					
			a	a	b	a	a	b	a				
				a	b	a	a	b	a	a			
					b	a	a	b	a	a	a		
						a	a	b	a	a	a	b	
							a	b	a	a	a	b	a

A word w is a said to be *primitive* if it is not of the form $w = v^k$, for a natural number $k \geq 2$. As a consequence of the *periodicity lemma* we show the following fact.

Lemma 1.4 *Assume the word x is primitive. Then x has exactly $|x|$ different* *cyclic shifts. In other words:*

$$|\{vu \ : \ u \text{ and } v \text{ words such that } x = uv \text{ and } u \neq \varepsilon\}| \ = \ |x|.$$

Proof. Assume x of length p has two cyclic shifts that are equal. Hence $x = uv = u'v'$, and $vu = v'u'$, where $u \neq u'$.

Assume without loss of generality that $|u'| < |u|$. Then $u = u'\alpha$, $v' = \alpha \cdot v$ and $vu'\alpha = \alpha v u'$. Hence the text $\alpha \cdot v \cdot u' \cdot \alpha$ has borders $\alpha \cdot v \cdot u'$ and α. Consequently, the text $\alpha \cdot v \cdot u' \cdot \alpha$ has two periods of size $r = |\alpha|$ and $p = |vu'\alpha|$. At the same time $r + p = |\alpha \cdot v \cdot u' \cdot \alpha|$.

The periodicity lemma implies that the text has period $\gcd(r, p)$. Since $r < p$ this shows that p is divisible by the length of the smaller period. This implies that x is a power of a smaller word, which contradicts the assumption. Consequently x cannot have two identical cyclic shifts. □

We show a simple number-theoretic application of primitive words and cyclic shifts. In 1640 the great French number theorist Pierre de Fermat stated the following theorem.

Theorem 1.1 [Fermat's Simple Theorem] *If p is a prime number and n is any natural number then p divides $n^p - n$.*

Proof. Define the equivalence relation $\equiv$ on words by $x \equiv y$ if x is a cyclic shift of y. A word is said to be unary if it is in a form a^p, for a letter a. Take the set S of all non-unary words of length p over the alphabet $\{1, 2, \ldots, n\}$. All these words are primitive since their length is a prime number and they are non-unary. According to Lemma 1.4 each equivalence class has exactly p elements. The cardinality of S is $n^p - n$ and S can be partitioned into disjoint subsets of the same cardinality p. Hence the cardinality of S is divisible by p, consequently $n^p - n$ also is. This completes the proof. □

Bibliographic notes

Complementary notions, problems and algorithms in stringology may be found in the books by Crochemore and Rytter [CR 94], by Stephen [St 94], by Gusfield [Gu 97], by Crochemore, Hancart and Lecroq [CHL 01], and in the collective book edited by Apostolico and Galil [AG 97].