Contents

1 Introduction **3**
1.1 Texts and their processing . . . 3
1.2 Text file facilities . . . 4
1.3 Dictionaries . . . 8
1.4 Data compression . . . 10
1.5 Applications of text algorithms in genetics . . . 10
1.6 Structure of this book . . . 12

Chapter 1

Introduction

1.1 Texts and their processing

One of the simplest and natural types of information representation is by means of written texts. Data to be processed often does not decompose into independent records. This type of data is characterized by the fact that it can be written down as a long sequence of characters. Such linear 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 which are quite large. For example, this book contains probably 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 where the practice and theory are very close together.

The basic textual problem is the problem called *pattern matching*. It is used to access information and probably many computers are solving in this moment 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 word “Marie-Curie-Sklodowska”. This is an example of a pattern matching problem, or string-matching. In this case, the word “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 only a single word. In many cases m is very large. In genetics the pattern can correspond to a genome that can be very long, in image processing the digitized images sent serially take millions of characters each. The string-matching problem is the basic question considered in the book, together with its variations. String-matching is also the basic subproblem in very other algorithmic problems on texts. Below there is a (not exclusive) list of basic groups of problems discussed in the book:

- variations on the string-matching problem,
- problems related to the structure 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 the string-matching and many other problems is given in the next chapter. We introduce now some of them informally in the context of applications.

1.2 Text file facilities

The UNIX system uses as a main feature text files for exchanging information. 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 be easily connected with each others, and especially through the pipelining facility. This often reduces the creation of new commands to 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 which contain an occurrence of the word "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 only 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 by specific strings. Regular expressions are also available with `sed`. But the editor contains an even more powerful notation. This allows, for instance, the action on a line of the input text containing twice same word. It can be applied to delete two consecutive occurrences of a same word in a text. This is both an instance 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 { action1 }
pattern2 { action2 }
pattern3 { action3 }
```

The basic components of this program are pattern to be find inside the lines of the current file. When a pattern is found, the corresponding action is applied to the line. So, several actions may be applied sequentially to a same line. This is an instance 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 eliminate control flow. This also frequently reduces the size of a program to 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 nice 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 language `awk` 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 few other possibilities). The output is the source in the C programming language of the specified lexical analyzer. A specification in `lex` is mainly a sequence of pattern-action statements 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])* { yylval = Install(); return(ID); }
```

The pattern specifies identifiers, that is, strings of characters starting with one letter and containing only letters and digits. The 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 practical application in the `lex` software.

`Lex` is often combined with `yacc` and `twig` for the development of parsers, translators, compilers, or simply prototypes of these. `Yacc` allows the specification of a parser by grammar rules. It produces the source of the specified parser in C language. `Twig` is a tree-manipulation language that helps to design efficient code generators. It transforms a tree-translation scheme into a code generator based on a fast top-down tree-pattern matching algorithm. The output generator also uses dynamic programming techniques for optimization purposes but the essential feature is the tree-pattern matching action for tree rewriting. It has been shown, by experimental results on few significant applications, that `twig`-generated code generators are often faster than hand-generated compilers even when the latter incorporate some tricky optimizations. One of the basic problems considered for the design of such a tool is the **tree pattern-matching** problem.

Texts such as books or programs are likely to be changed during elaboration. Even after their achievement they often support periodic upgrades. These questions are related to **text comparisons**. Sometimes also we wish to find a string, and we do not completely remember it. The search has to be performed with a non entirely specified pattern. This is an instance of the **approximate pattern matching**. Keeping track of all consecutive versions of a text sometimes does not help 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 on what are the differences, nor conversely what are the similarities between two texts. However, it can be agreed that the intersection of two texts is a longest common subtext of both. This is called in our book the **longest common subsequence** problem, so that the differences between two texts are the respective complements of the common part. The UNIX command `diff` builds on this notion. Consider the texts A and B as follows (Molière's joke, 1670):
Text A:

> Belle Marquise,
> vos beaux yeux
> me font mourir d'amour

Text B:

> D'amour mourir me font,
> Belle Marquise,
> vos beaux yeux

The command `diff A B` produces

```
0a1
> D'amour mourir me font,
3d3
< me font mourir d'amour
```

An option of the command `diff` produces a sequence of `ed` instructions to transform one text into the other text. The similarity of texts can be measured as a 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.

1.3 Dictionaries

The search for words or patterns in static texts is a quite different question than the previous pattern matching mechanism. Usual dictionaries, for instance, 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 text related to words in 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 hard problem during the development of a compiler. A contrario, English contains approximatively $100,000$ words, and even twice more if inflected forms are considered. In French inflected forms produce more than $500,000$ words. The representation of lexicons of this size makes the problem a bit harder.

A simple use of dictionaries is made by spelling checkers. The UNIX command, `spell`, reports the words of its input that are not stored in the lexicon. This rough approach does not yield a pertinent checker, but it practically helps to find typing errors. The lexicon used by `spell` contains around $70,000$ entries stored within less than 60 kilobytes of random-access memory.

Quick access to lexicons is a necessary condition to produce good parsers. The data structure useful for such accesses is called an index. In our book indexes correspond to data structures representing all factors of a given (presumably long) text. We consider problems of the construction such structures: **suffix trees**, **directed acyclic word graphs**, **factor automata**, **suffix arrays**. The tool *PAT* 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 softwares and character recognition.

An image scanner is a kind of photocopier. It is used to give a digitalised version of an image. When the image is the page of text, the natural output of the scanner must be a digital form available by a text editor. The transformation of a digitalised image of a text into a usual computer representation of the text is realized by an Optical Character Reader (OCR). Scanning a text with an OCR can be 50 times faster than retyping the text on a keyboard. OCR softwares are thus likely to become more common. But they still suffer from a high degree of imprecision. The average rate of errors in the recognition of characters is around one percent. Even if this may happen as rather small, this means that scanning a book produces around one error per line. This is to be compared with the usual very high quality of texts checked by specialized persons. Technical improvements on the hardware can help to eliminate certain kinds of errors occurring on scanned texts in printed forms. But this cannot avoid the problem of recognizing texts written by hand. In this case, isolation of characters is much harder, and ambiguous forms can be encountered at recognition stage. Reduction on the number of errors can thus only be achieved by considering the context of characters, which assumes some understanding 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, this is discussed in the book in the context of the **dictionary of basic factors (subimages)**.

No system is presently able to parse natural languages. Indeed, it is questionable whether such an effective system can exist. Words are certainly units of syntax in languages, but they are not units of meaning. This is both because words can have several meanings, and because they can be parts of compound phrases that carry a meaning only as a whole. It appears that the proportion in the lexicon (of French and several other European languages) of idiomatic sentences, of metaphoric and technical sentences whose meaning cannot be inferred from their elements, is very high. The natural consequence is that compound expressions and the so-called frozen phrases must be included in computerized dictionaries. The data base must also incorporate at least the syntactic properties of the elements. At present, no complete dictionary established according to these principles exists. This certainly represent of huge amount of data to be collected but this seems an indispensable work. Data bases for the European languages, French, English, Italian, and Spanish are presently being built. And studies on Arabic, Chinese, German, Korean and Madagascan have been performed. These question are studied, for instance, at LADL (Paris), and the reader can refer to [5] for more information on specific aspects of computational linguistics.

The theoretical approach to the representation of lexicons is 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. We have shown at LITP (Paris) that the use of automata to represent lexicons is particularly efficient. Experiments have been done on a $500,000$ word lexicon of LADL (Paris). The representation supports direct access to any word of the lexicon and takes only 300 kbytes of random-access memory.

1.4 Data compression

One of the basic problems in keeping 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 form. No loss of information is allowed. Text compression is related to **Huffman coding problem** and the **factorization problem**. This kind of compression contrasts with other kind of compression technics applied to sounds or images, where approximation is acceptable. Availability of large mass storage does not decrease the interest for compressing data. Indeed, users always use extra available space to store more data or new kind 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 networks, for instance. The success of Fax is perhaps to be credited to compression technics.

General compression methods often adapts themselves to the data. This phenomenon is central in getting high compression ratios. But, it appears in practice that methods tailored for specific data lead to the best results. We have experimented this fact on data sent by "géostationnaires" satellites. The data have been compressed to 7% 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 expertize the data.

1.5 Applications of text algorithms in genetics

Molecules of nucleic acids carry a large part 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 desoxyrybonucleic 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 is a sequence on the four-letter alphabet of nucleotides: adenine (A), guanine (G), cytosine (C), and thymine (T). RNA molecules is usually single-stranded composed of ribonucleotides: A, G, C, and uracil (U). Processes of "transcription" and "translation" lead to the production of proteins, which have also a primary structure as a string composed of 20 amino acids. In a first approach all these molecules can be viewed as texts. The discovery of powerful sequencing techniques fifteen years ago has led to a rapid accumulation of sequence data (more than 10 million nucleotides of sequences). From the collect of sequences up to their analysis a lot of algorithms on texts are implied. Moreover, only fast algorithms are often feasible because of the huge amount of data.

Collecting sequences is made through gel audioradiographs. 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 is another example of 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 on it is whether it resembles to any other sequence already stored in databanks. Before adding a new molecular sequence into an existing database one needs to know whether the sequence is already present or not. The comparison of several sequences is usually realized by writing one over another. The result is known as an alignment of the set of sequences. This is a common display in molecular biology that can gives an idea on the evolution of the sequences though the mutations, insertions and deletions 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 on sequences are related to their analysis. The aim is to discover the functions of all parts of the sequence. For instance, DNA sequences contain important regions (coding sequences) for the production of proteins. But, no good answer is presently known to find all coding sequences of a DNA sequence. Another question on 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 instance, 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 any other **regularities**.

1.6 Structure of this book

The structure of our book reflects both the relationships between applications of text algorithmic techniques and the classification of algorithms according to the measures of complexity considered. The book can be viewed as a "parade" of algorithms in which our main aim is to discuss the foundations of these algorithms and their interconnections.

Chapter 2 begins with the algorithmic and theoretical foundations: machine models, discussion on algorithmic efficiency, review of basic problems and their interaction, and some combinatorics on words.

The topic of Chapters 3 and 4 is *string matching*. The chapters deal with two famous algorithms, Knuth-Morris-Pratt and Boyer-Moore algorithms respectively. The upmost improvements and results on these algorithms are incorporated. We discuss here those string- matching algorithms which are efficient with respect to only one measure of complexity — sequential time. Hence, the chapter can be treated as a "warm-up" for other chapters devoted to the same topic. We later discuss more difficult algorithms in Chapter 13 — *time-space optimal string-matching* algorithms that work in linear time and simultaneously use only a very small additional memory. The consideration of two complexity measures together usually leads to most complicated algorithms. Chapters 3 and 4 can also be treated as a preparation to Chapters 9 and 14 where are presented advanced *parallel string-matching* algorithms, and to Chapter 12 that deals with two-dimensional pattern matching.

Chapters 5 and 6 are basic chapters of the book. They cover data structures representing succinctly all subwords of a text, namely, *suffix trees* and *subword graphs*. The considerations are typical from the point of view of data structures manipulations: trees, links, vectors, lists, etc. The prepared data structures are used later in the discussion on other problems; especially in the computation of the longest common factor of texts, factorization problems and finding squares in words. Also it is used to construct in Chapter 7 certain kind of automata related to string-matching.

Chapter 7 gives an automata theoretical approach to string-matching that leads to a natural solution of the *multi-pattern matching* problem. It covers Aho-Corasick automata for multi-pattern matching, automata for the set of factors and suffixes of a given text, and application of them.

Chapter 8 considers *regularities in texts: symmetries and repetitions.* It covers problems related to symmetries of words as well as problems related to repetitions. Surprisingly it turns out that finding a palindrome in a word in linear time is much easier than finding a square. Generally, problems related to symmetries are easier than that related to repetitions. The basic component of the chapter is a new data structure, called the dictionary of basic factors, that is introduced as a basic technique for the development of efficient, but not necessarily optimal, algorithms.

Chapter 9 deals with *parallel algorithms on texts*. Only one, very general model of parallel computations is considered. The parallel implementation and applications of the dictionary of basic factors, suffix trees and subword graphs are presented. We discuss parallel version of the Karp-Miller-Rosenberg algorithm (Chapter 8) and some computations related to trees.

Chapter 10 is about data compression. It discusses only some of the *compression techniques for texts*. This area is exceptionally broad as compression of data is the vital part of many real world systems. We focus on few important algorithms. Two of the subjects here are the Ziv-Lempel algorithm, and applications of data structures from Chapters 5 and 6 to the text factorization.

Questions related to algorithms involving the notion of *approximate patterns* are presented in Chapter 11. It starts with algorithms to compute edit distance of sequences (alignment of sequences). Two specific classical notions on approximate patterns are then considered, the first for string-matching with errors, and the second for patterns with "don't care" symbols.

Two-dimensional pattern matching, in Chapter 12, is the problem of identifying a subimage inside a larger one. The notion naturally extends that of pattern matching in texts. Basic techniques on the subject show the importance of algorithms presented in Chapter 7.

Chapter 13 presents three different *time-space optimal string-matching* algorithms. None of them have never appeared in a textbook nor even in a journal, and the third one is presented in a completely new version. This is an advanced material since the algorithms rely heavily on the combinatorics of words, mostly related to periodicities.

The discussion on *optimal parallel algorithm on texts* is the subject of Chapter 14. It includes more technical considerations than Chapter 9. Are presented there the well-known Vishkin's algorithm and a recent optimal parallel algorithm for string-matching, based on suffix versus prefix computations, designed by Kedem, Landau and Palem. The last chapter considers several miscellaneous problems like tree-pattern matching problem, or the computation shortest common superstrings, maximal suffixes, Lyndon factorizations

One can partition algorithmic problems discussed in this book into practical and theoretical problems. Certainly the 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 the form directed mainly to specialists. We hope that this book will cover a gap on algorithms on words in the book literature, and bring together many results presently dispersed in the masses of journal articles.

Selected references

1. A.V. Aho, B.W. Kernighan & P.J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, Mass., 1988.

2. A.V. Aho, R. Sethi & J. Ullman, *Compilers—Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1988, Chapter 3.

3. T.C. Bell, J.C. Cleary & I.H. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

4. M.J. Bishop & C.J. Rawlings, *Nucleic acid and protein sequence analysis: a practical approach*, IRL Press Limited, Oxford, England, 1987.

5. M. Gross, Constructing lexicon-grammars, in: (B.T.S. Atkins & A. Zampolli eds, *Computational approaches to the lexicon*, Oxford University Press, 1991).

6. G. Salton, *Automatic Text Processing*, Addison-Wesley, Reading, Mass., 1989.

7. P.D. Smith, *An introduction to text processing*, The MIT Press, Cambridge, Mass., 1990.

8. M.S. Waterman, *Mathematical methods for DNA sequences*, CRC Press, Boca Raton, Florida, 1989.