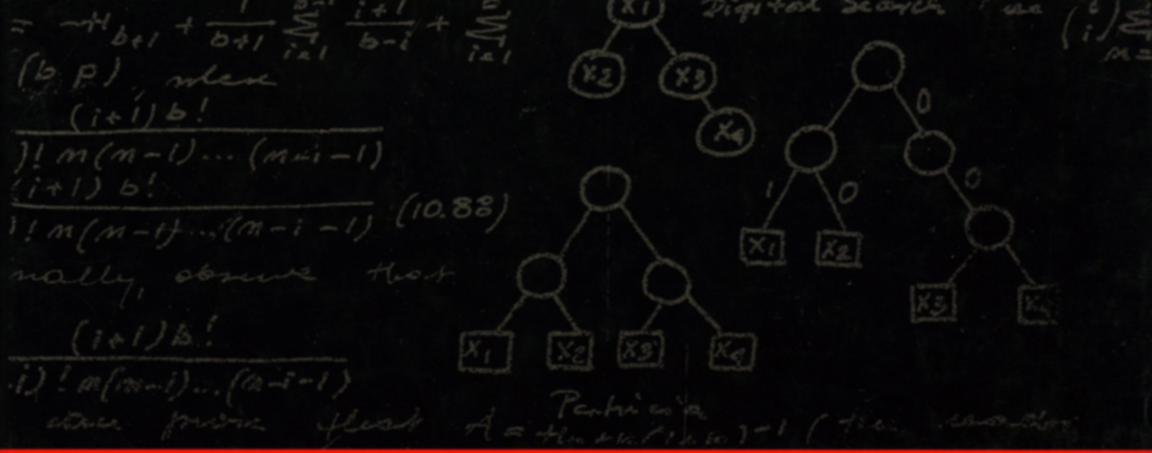


Wiley-Interscience Series in Discrete Mathematics and Optimization

AVERAGE CASE ANALYSIS OF ALGORITHMS ON SEQUENCES

Wojciech Szpankowski

Average Case Analysis of Algorithms on Sequences

Average Case Analysis of Algorithms on Sequences

WOJCIECH SZPANKOWSKI
Department of Computer Science
Purdue University

A Wiley-Interscience Publication
JOHN WILEY & SONS, INC.
New York • Chichester • Weinheim • Brisbane • Singapore • Toronto

Library of Congress Cataloging-in-Publication Data is available.
Szpankowski, Wojciech, 1952–
Average case analysis of algorithms on sequences / Wojciech Szpankowski.
p. cm. – (Wiley-Interscience series in discrete mathematics and optimization)
Includes bibliographical references and index.
ISBN 0-471-24063-X (cloth : alk. paper)
1. Computer algorithms. I. Title. II. Series.
QA76.9.A43 S87 2000
005.1—DC21 00-042253

10 9 8 7 6 5 4 3 2 1

Książke te poświęcam Moim Rodzicom,
Aleksandrze i Wincentemu,
na ich 50-cio lecie małżeństwa,
za ich nieustającą wiarę we mnie.

This page intentionally left blank

Contents

Foreword

Sequences—also known as strings or words—surface in many areas of science. Initially studied by combinatorialists in relation to problems of formal linguistics, they have proved to be of fundamental importance in a large number of computer science applications, most notably in textual data processing and data compression. Indeed, designers of large internet search engines acknowledge them to be huge algorithmic factories operating on strings. In a different sphere, properties of words are essential to the processing and statistical interpretation of biological or genetic sequences. There it is crucial to discern signal from noise and to do so in a computationally efficient manner.

As its title indicates, Szpankowski's book is dedicated to the analysis of algorithms operating on sequences. First, perhaps, a few words are in order regarding analysis of algorithms. The subject was founded by Knuth around 1963 and its aim is a precise characterization of the behaviour of algorithms that operate on large ensembles of data. A complexity measure (like execution time) is fixed and there are usually a few natural probabilistic models meant to reflect the data under consideration. The analyst's task then consists in predicting the complexity to be observed. Average-case analysis focuses on expected performance; whenever possible, finer characteristics like variances, limit distributions, or large deviations should also be quantified.

For a decade or so, it has been widely recognized that average-case analyses tend to be far more informative than worst-case ones as the latter focus on somewhat special pathological configurations. Provided the randomness model is realistic, average-case complexity estimates better reflect what is encountered in practice—hence their rôle in the design, optimization, and fine tuning of algorithms. In this context, for algorithms operating on words, properties of random sequences are crucial. Their study is the central theme of the book.

"Give a man a fish and you feed him for the day; teach him to fish and you feed him for his lifetime." Following good precepts, Szpankowski's book has a largely methodological orientation. The book teaches us in a lucid and balanced fashion the two main competing tracks for analysing properties of discrete randomness. Probabilistic methods based on inequalities and approximations include moment inequalities, limit theorems, large deviations, as well as martingales and ergodic theory, all of which nicely suited to the analysis of random discrete structures. Analytic methods fundamentally rely on exact modelling by generating functions which, once viewed as transformations of the complex plane, begin to reveal their secrets. Singularity analysis, saddle point strategies, and Mellin transforms become then instrumental. As Hadamard was fond of saying, the shortest path between two truths on the real line passes through the complex domain.

Throughout the book, the methodology is put to effective use in analysing some of the major problems concerning sequences that have an algorithmic or information-theoretic nature. In fact the book starts right away with a few easily stated questions that form recurrent themes; these are relative to digital trees, data compression, string editing, and pattern-matching. A great many more problems are thoroughly analysed in later chapters, including the celebrated leader election problem of distributed computing, fast pattern matching, basic information theory, Lempel-Ziv compression, lossy compression.

Analysis of algorithms is now a mature field. The time is ripe for books like this one which treat wide fields of applications. Szpankowski's book offers the first systematic synthesis on an especially important area—algorithms on sequences. Enjoy its mathematics! Enjoy its information theory! Enjoy its multi-faceted computational aspects!

PHILIPPE FLAJOLET

Preface

An algorithm is a systematic procedure that produces in a finite number of steps the solution to a problem. The name derives from the Latin translation *Algoritmi de numero Indorum* of the 9th-century Arab mathematician al-Khwarizmi's arithmetic treatise *Al-Khwarizmi Concerning the Hindu Art of Reckoning*. The most obvious reason for *analyzing* algorithms, and data structures associated with them, is to discover their characteristics in order to evaluate their suitability for various applications or to compare them with other algorithms for the same application. Often such analyses shed light on properties of computer programs and provide useful insights of the combinatorial behaviors of such programs. Needless to say, we are interested in *good* algorithms in order to efficiently use scarce resources such as computer space and time.

Most often algorithm designs are finalized toward the optimization of the asymptotic *worst-case* performance. Insightful, elegant, and useful constructions have been set up in this endeavor. Along these lines, however, the design of an algorithm is sometimes targeted at coping efficiently with unrealistic, even pathological inputs and the possibility is neglected that a simpler algorithm that works fast on average might perform just as well, or even better in practice. This alternative solution, also called a probabilistic approach, was an important issue three decades ago when it became clear that the prospects for showing the existence of polynomial time algorithms for NP-hard problems were very dim. This fact, and the apparently high success rate of heuristic approaches to solving certain difficult problems, led Richard Karp in 1976 to undertake a more serious investigation of probabilistic algorithms. In the last decade we have witnessed an increasing interest in the *probabilistic analysis* (also called *average-case analysis*) of algorithms, possibly due to a high success rate of randomized algorithms for computational geometry, scientific visualization, molecular biology, and information theory. Nowadays worst-case and average-case analyses coexist in a friendly symbiosis, enriching each other.

The focus of this book is on *tools* and *techniques* used in the average-case analysis of algorithms, where *average case* is understood very broadly (e.g., it includes exact and limiting distributions, large deviations, variance, and higher moments). Such methods can be roughly divided into two categories: *analytic* and *probabilistic*. The former were popularized by D. E. Knuth in his monumental three volumes, *The Art of Computer Programming*, whose prime goal was to accurately predict the performance characteristics of a wide class of algorithms. Probabilistic methods were introduced by Erdős and Rényi and popularized in the book by Erdős and Spencer, *Probabilistic Methods in Combinatorics*. In general, nicely structured problems are amenable to an analytic approach that usually gives much more precise information about the algorithm under consideration. As argued by Andrew Odlyzko: "Analytic methods are extremely powerful and when they apply, they often yield estimates of unparalleled precision." On the other hand, structurally complex algorithms are more likely to be first solved by probabilistic tools that later could be further enhanced by a more precise analytic approach.

The area of analysis of algorithms (at least, the way we understand it) was born on July 27, 1963, when D. E. Knuth wrote his "Notes on Open Addressing" about hashing tables with linear probing. Since 1963 the field has been undergoing substantial changes. We see now the emergence of combinatorial and asymptotic methods that allow the classification of data structures into broad categories that are amenable to a unified treatment. Probabilistic methods that have been so successful in the study of random graphs and hard combinatorial optimization problems play an equally important role in this field. These developments have two important consequences for the analysis of algorithms: it becomes possible to predict average behavior under more *general* probabilistic models; at the same time it becomes possible to analyze much more structurally *complex* algorithms. To achieve these goals the analysis of algorithms draws on a number of branches in mathematics: combinatorics, probability theory, graph theory, real and complex analysis, number theory and occasionally algebra, geometry, operations research, and so forth.

In this book, we choose one facet of the theory of algorithms, namely, algorithms and data structures on sequences (also called strings or words) and present a detailed exposition of the analytic and probabilistic methods that have become popular in such analyses. As stated above, the focus of the book is on techniques of analysis, but every method is illustrated by a variety of specific problems that arose from algorithms and data structures on strings. Our choice stems from the fact that there has been a resurgence of interest in algorithms on sequences and their applications in computational biology and information theory.

Our choice of methods covered here is aimed at closing the gap between analytic and probabilistic methods. There are excellent books on analytic methods such as the three volumes of D. E. Knuth and the recent book by Sedgewick and Flajolet. Probabilistic methods are discussed extensively in the books by Alon and Spencer, Coffman and Lueker, and Motwani and Raghavan. However, remarkably few books have been dedicated to both analytic and probabilistic analysis of algorithms, with possible exceptions of the books by Hofri and Mahmoud.

ABOUT THIS BOOK

This is a graduate textbook intended for graduate students in computer science, discrete mathematics, information theory, applied mathematics, applied probability, and statistics. It should also be a useful reference for researchers in these areas. In particular, I hope that those who are experts in probabilistic methods will find the analytic part of this book interesting, and vice versa.

The book consists of three parts: Part I describes a class of algorithms (with associated data structures) and formulates probabilistic and analytic models for studying them. Part II is devoted to probabilistic methods, whereas Part III presents analytic techniques.

Every chapter except the first two has a similar structure. After a general overview, we discuss the method(s) and illustrate every new concept with a simple example. In most cases we try to provide proofs. However, if the method is well explained elsewhere or the material to cover is too vast (e.g., the asymptotic techniques in Chapter 8), we then concentrate on explaining the main ideas behind the methods and provide references to rigorous proofs. At the end of each chapter there is an application section that illustrates the methods discussed in the chapter in two or three challenging research problems. Needless to say, the techniques discussed in this book were selected for inclusion on exactly one account: how useful they are to solve these application problems. Naturally, the selection of these problems is very biased, and often the problem shows up in the application section because I was involved in it. Finally, every chapter has a set of exercises. Some are routine calculations, some ask for details of derivations presented in the chapter, and others are research problems denoted as ⚠ and unsolved problems marked as ▽.

Now we discuss in some detail the contents of the book. Part I has two chapters. The first chapter is on the algorithms and data structures on words that are studied in this book: We discuss digital search trees (i.e., tries, PATRICIA tries, digital search trees, and suffix trees), data compression algorithms such as Lempel-Ziv schemes (e.g., Lempel-Ziv'77, Lempel-Ziv'78, lossy extensions of Lempel-Ziv schemes), pattern matching algorithms (e.g., Knuth-Morris-Pratt and Boyer-Moore), the shortest common superstring problem, string editing problem (e.g., the longest common subsequence), and certain combinatorial optimization problems. Chapter 2 builds probabilistic models on sequences that are used throughout the book to analyze the algorithms on strings. In particular, we discuss memoryless, Markov, and mixing sources of generating random sequences.

We also review some facts from probability theory and complex analysis. We finish this chapter with an overview on special functions (e.g., the Euler gamma function and the Riemann zeta function) that are indispensable for the analytic methods of Part III.

Part II consists of four chapters. Chapter 3 is on the probabilistic and combinatorial inclusion–exclusion principle, the basic tool of combinatorial analysis. After proving the principle, we discuss three applications, namely, the depth in a trie, order statistics, and the longest aligned matching word. Chapter 4 is devoted to the most popular probabilistic tool, that of the the first and the second moment methods. We illustrate them with a variety of examples and research problems (e.g., Markov approximation of a stationary distribution and the height analysis of digital trees). In Chapter 5 we discuss both the subadditive ergodic theorem and the large deviations. We use martingale differences to derive the very powerful Azuma's inequality (also known as the method of bounded differences). Finally, in Chapter 6 we introduce elements of information theory. In particular, we use random coding technique to prove three fundamental theorems of Shannon (i.e., the source coding theorem, the channel coding theorem and the rate distortion theorem). In the applications section we turn our attention to some recent developments in data compression based on pattern matching and the shortest common superstring problem. In particular, we show that with high probability a greedy algorithm that finds the shortest common superstring is asymptotically optimal. This is of practical importance because the problem itself is NP-hard.

Part III is on analytic methods and is composed of four chapters. Chapter 7 introduces generating functions, a fundamental and the most popular analytic tool. We discuss ordinary generating functions, exponential generating functions, and Dirichlet series. Applications range from pattern matching algorithms to the Delange formula on a digital sum. Chapter 8 is the longest in this book and arguably the most important. It presents an extensive course on complex asymptotic methods. It starts with the Euler-Maclaurin summation formula, matched asymptotics and the WKB method, continues with the singularity analysis and the saddle point method, and finishes with asymptotics of certain alternating sums. In the applications section we discuss the minimax redundancy rate for memoryless sources and the limiting distribution of the depth in digital search trees. The next two chapters continue our discussion of asymptotic methods. Chapter 9 presents the Mellin transform and its asymptotic properties. Since there are good accounts on this method (cf. [132, 149]), we made this chapter quite short. Finally, the last chapter is devoted to a relatively new asymptotic method known as depoissonization. The main thrust lies in an observation that certain problems are easier to solve when a deterministic input is replaced by a Poisson process. However, nothing is for free in this world, and after solving the problem in the Poisson domain one must translate the results back to the original problem, that is, depoissonize them. We cover here almost all known depoissonization results and illustrate them with three problems: analysis of the leader election algorithm, and the depth in generalized digital search trees for memoryless and Markovian sources. The latter analysis is among the most sophisticated in this book.

PERSONAL PERSPECTIVE

I can almost pin down the exact date when I got interested in the analysis of algorithms. It was in January 1983 in Paris during a conference on performance evaluation (at that time I was doing research in performance evaluation of multiaccess protocols and queueing networks). I came from a gloomy Poland, still under martial law, to a glamorous, bright, and joyful Paris. The conference was excellent, one of the best I have ever participated in. Among many good talks one stood out for me. It was on approximate counting, by Philippe Flajolet. The precision of the analysis and the brightness (and speed) of the speaker made a lasting impression on me. I wished I could be a disciple of this new approach to the analysis of algorithms. I learned from Philippe that he was influenced by the work of D. E. Knuth. For the first time I got a pointer to the three volumes of Knuth's book, but I could not find them anywhere in Poland.

In 1984 I left Gdańsk and moved to McGill University, Montreal. I had received a paper to review on the analysis of conflict resolution algorithms. The paper was interesting, but even more exciting was a certain recurrence that amazingly had a "simple" asymptotic solution. I verified numerically the asymptotics and the accuracy was excellent. I wanted to know why. Luckily, Luc Devroye had just returned from his sabbatical and he pointed me again to Knuth's books. I found what I needed in volumes I and III. It was an illumination! I was flabbergasted that problems of this complexity could be analyzed with such accuracy. I spent the whole summer of 1984 (re)learning complex analysis and reading Knuth's books. I started solving these recurrences using the new methods that I had been learning. I was becoming a disciple of the precise analysis of algorithms.

When I moved to Purdue University in 1985, I somehow figured out that the recurrences I was studying were also useful in data structures called *tries*. It was a natural topic for me to explore since I moved from an electrical engineering department to a computer science department. I got hooked and decided to write to Philippe Flajolet, to brag about my new discoveries. In response he sent me a ton of papers of his own, solving even more exciting problems. I was impressed. In May 1987 he also sent to Purdue his best weapon, a younger colleague whose name was Philippe Jacquet. When Philippe Jacquet visited me I was working on the analysis of the so-called interval searching algorithm, which I knew how to solve but only with numerical help. Philippe got a crucial idea on how to push it using only analytic tools. We wrote our first paper [214]. Since then we have met regularly every year producing more and longer papers (cf. Chapter 10; in particular, Section 10.5.3). We have become friends.

Finally, in 1989 I *again* rediscovered the beauty of information theory after reading the paper [452] by Wyner and Ziv. There is a story associated with it. Wyner and Ziv proved that the typical length of repeated substrings found within the first n positions of a random sequence is with high probability $\frac{1}{h}\log n$, where h is the entropy of the source (cf. Section 6.5.1). They asked if this result can be extended to almost sure convergence. My work on suffix trees (cf. Section 4.2.6) paid off since I figured out that the answer is in the negative [411] (cf. also Section 6.5.1). The crucial intuition came from the work of Boris Pittel [337], who encouraged me to study it and stood behind me in the critical time when my analysis was under attack. It turned out that Ornstein and Weiss [333] proved that the Wyner and Ziv conjecture is true. To make the story short, let me say that fortunately both results were correct since we analyzed slightly different settings (i.e., I analyzed the problem in the so-called right domain of asymptotics, and Ornstein and Weiss in the left domain). The reader may read more about it in Chapter 6. Since then I have found information theory more and more interesting. Philippe Jacquet and I have even coined the term *analytic information theory* for dealing with problems of information theory by using analytic methods, that is, those in which complex analysis plays a pivotal role.

Acknowledgments

There is a long list of colleagues and friends from whom I benefited through encouragement and critical comments. They helped me in various ways during my tenure in the analysis of algorithms. All I know about analytic techniques comes from two Philippes: Jacquet and Flajolet. And vice versa, what I do not know is, of course, their fault. On a more serious note, my direct contact with Philippe Jacquet and Philippe Flajolet has influenced my thinking and they have taught me many tricks of the trade. (I sometimes wonder if I didn't become French during numerous visits to INRIA, Rocquencourt; all except for mastering the language!) Many applications in this book are results of my work with them. Needless to say, their friendship and inspirations were invaluable to me. *Merci beaucoup mes amis.*

I have been learning probabilistic techniques from masters: David Aldous, Alan Frieze, Tomasz Łuczak, and Boris Pittel. *Thanks. Thanks. Dziękuje.* Спасибо.

I thank my numerous co-authors with whom I worked over the last 15 years: David Aldous, Marc Alzina, Izydor Apostol, Alberto Apostolico, Mikhail Atallah, Luc Devroye, Jim Fill, Philippe Flajolet, Ioannis Fudos, Yann Genin, Leonidas Georgiadis, Ananth Grama, Micha Hofri, Svante Janson, Philippe Jacquet, Peter Kirschenhofer, Chuck Knessl, Guy Louchard, Tomasz Łuczak, Hosam Mahmoud, Dan Marinescu, Evaggelia Pitoura, Helmut Prodinger, Bonita Rais, Vernon Rego, Mireille Régnier, John Sadowsky, Erkki Sutinen, Jing Tang, and Leandros Tassiulas.

I also had the privilege of talking with D. E. Knuth, whose encouragement was very important to me. His influence pervades the book.

Many colleagues have read various versions of this book. I profoundly thank Luc Devroye, Michael Drmota, Leonidas Georgiadis, Philippe Jacquet, Svante Janson, Chuck Knessl, Yiannis Kontoyiannis, Guy Louchard, John Kieffer, Gabor Lugosi, Kihong Park, Helmut Prodinger, Yuri Reznik, and Erkki Sutinen. I am particularly in debt to Leonidas Georgiadis, who read the entire book and gave me many valuable comments.

As with every large project, I am sure I did not avoid mistakes and typos. I will try to keep errata on my home page `www.cs.purdue.edu/people/spa`. Readers' help in eliminating remaining inaccuracies will be greatly appreciated. Please send comments to `spa@cs.purdue.edu`.

This book was written over the last four years. I have received invaluable help from staff and faculty of the Department of Computer Sciences at Purdue University. I am grateful to the National Science Foundation, which supported me over the last 15 years. The last three chapters were written during my sabbatical at Stanford University during fall 1999. I thank Tom Cover for being a wonderful host and his students for welcoming me into their weekly seminar.

Philippe Flajolet kindly agreed to write a Foreword to this book. I thank him from the bottom of my heart. Philippe Jacquet has been entertaining me for more than ten years with his good humor and comic sketches. He generously agreed to contribute ten comic illustrations to this book; one for each chapter. If only for these sketches, the book should be on your shelf!

Finally, I am in debt to my family for putting up with me through these years and helping me, in their own way, to continue this project. My wife, Mariola, designed the cover. I have much to thank her for, including some twenty years with me.

WOJTEK SZPANKOWSKI

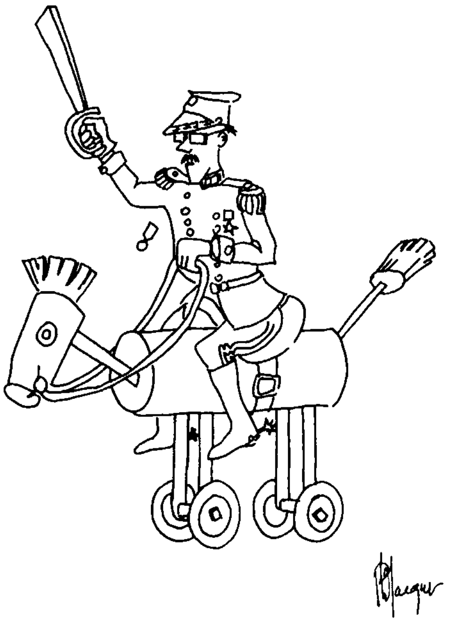
West Lafayette, Indiana, 1997–1998
Stanford, California, 1999

Part I

PROBLEMS ON WORDS

1

Data Structures and Algorithms on Words



In this book we choose one facet of the theory of algorithms, namely data structures and algorithms on sequences (strings, words) to illustrate probabilistic, combinatorial, and analytic techniques of analysis. In this chapter, we briefly describe some data structures and algorithms on words (e.g., tries, PATRICIA tries, digital search trees, pattern matching algorithms, Lempel-Ziv schemes, string editing, and shortest common superstring) that are used extensively throughout the book to illustrate the methods of analysis.

Data structures and algorithms on sequences have experienced a new wave of interest due to a number of novel applications in computer science, communications, and biology. Among others, these include dynamic hashing, partial match retrieval of multidimensional data, searching and sorting, pattern matching, conflict resolution algorithms for broadcast communications, data compression, coding, security, genes searching, DNA sequencing, genome maps, double digest problem, and so forth. To satisfy these diversified demands various data structures were proposed for these algorithms. Undoubtedly, the most popular data structures in algorithms on words are digital trees [3, 269, 305] (e.g., tries, PATRICIA, digital search trees), and in particular suffix trees [3, 17, 77, 375, 383, 411, 412]. We discuss various digital trees and introduce several parameters characterizing them that we shall study throughout the book.

The importance of digital trees stems from their abundance of applications in other problems such as data compression (Section 1.2), pattern matching (Section 1.3), and the shortest common superstring problem (Section 1.4). These problems recently became very important due to the need for an efficient storage and transmission of multimedia, and possible applications to DNA sequencing.

Graphs and directed acyclic graphs (DAG) also find several applications in problems on strings. In particular, we consider the *edit distance* problem and its variants (Section 1.5). Finally, we close this chapter with a brief discussion of certain class of optimization problems on graphs that find applications for algorithms on sequences (Section 1.6).

1.1 DIGITAL TREES

We start our discussion with a brief review of the **digital trees**. The most basic digital tree, known as a **trie** (from re*trie*val), is defined first, and then other digital trees (such as PATRICIA, digital search trees and suffix trees) are described in terms of the trie.

The primary purpose of a trie is to store a set $\mathcal{X}$ of strings (words, sequences), say $\mathcal{X} = \{X^1, \ldots, X^n\}$. Throughout the book, the terms strings, words and sequences are used interchangeably. Each string is a finite or infinite sequence of symbols taken from a finite alphabet $\mathcal{A} = \{\omega_1, \ldots, \omega_V\}$ of size $V = |\mathcal{A}|$. We use a generic notation $\mathcal{D}_n$ for all digital trees built over a set $\mathcal{X}$ of n strings. A string will be stored in a leaf of the trie. The trie over $\mathcal{X}$ is built recursively as follows: For $|\mathcal{X}| = 0$, the trie is, of course, empty. For $|\mathcal{X}| = 1$, $trie(\mathcal{X})$ is a single node. If $|\mathcal{X}| > 1$, $\mathcal{X}$ is split into V subsets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_V$ so that a string is in $\mathcal{X}_j$ if its first symbol is ω_j. The tries $trie(\mathcal{X}_1), trie(\mathcal{X}_2), \ldots, trie(\mathcal{X}_V)$ are constructed in the same way except that at the kth step, the splitting of sets is based on the kth symbol. They are then connected from their respective roots to a single node to create $trie(\mathcal{X})$. Figure 1.1 illustrates such a construction. Observe

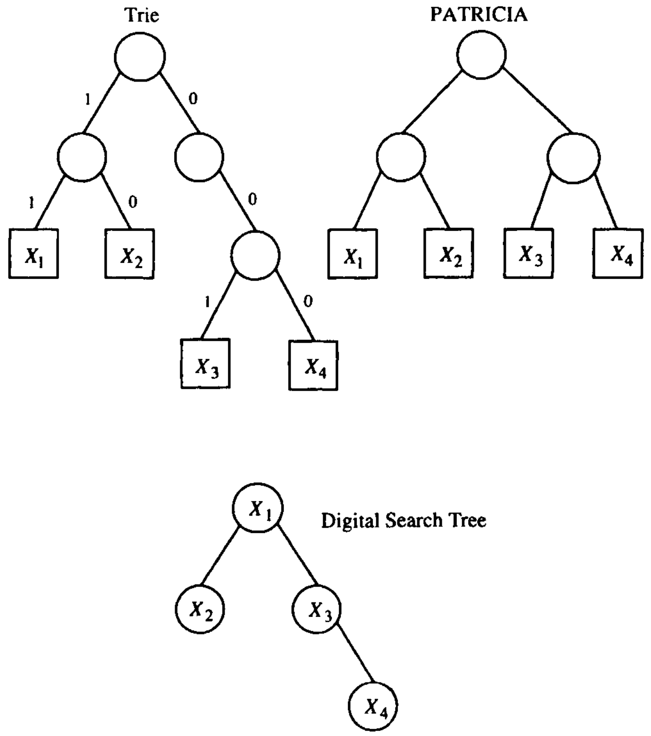


Figure 1.1. A trie, Patricia trie and a digital search tree (DST) built from the following four strings $X^1 = 11100\ldots$, $X^2 = 10111\ldots$, $X^3 = 00110\ldots$, and $X^4 = 00001\ldots$.

that all strings are stored in external nodes (shown as boxes in Figure 1.1) while internal nodes are branching nodes used to direct strings to their destinations (i.e., external nodes). When a new string is inserted, the search starts at the root and proceeds down the tree as directed by the input symbols (e.g., symbol "0" in the input string means move to the right and "1" means proceed to the left as shown in Figure 1.1).

There are many possible variations of the trie. One such variation is the **b-trie**, in which a leaf is allowed to hold as many as b strings (cf. [145, 305, 411]). The b-trie is particularly useful in algorithms for extendible hashing in which the capacity of a page or other storage unit is b. A second variation of the trie, the **PATRICIA trie** (*P*ractical *A*lgorithm *T*o *R*etrieve *I*nformation *C*oded *I*n *A*lphanumeric) eliminates the waste of space caused by nodes having only one branch. This is done by collapsing one-way branches into a single node (Figure 1.1). In a **digital search tree** (DST), shown also in Figure 1.1, strings are directly stored in nodes so that external nodes are eliminated. More precisely, the root contains the first string (however, in some applications the root is left empty). The next string occupies the right or the left child of the root depending on whether its first symbol is "0" or "1". The remaining strings are stored in available nodes which are directly attached to nodes already existing in the tree. The search for an available node follows the prefix structure of a string as in tries. That is, if the next symbol in a string is "0" we move to the right, otherwise we move to the left.

As in the case of a trie, we can consider an extension of the above digital trees by allowing them to store up to b strings in an (external) node. We denote such digital trees as $\mathcal{D}_n^{(b)}$, but we often drop the upper index when $b = 1$ is discussed. Figure 1.1 illustrates these definitions for $b = 1$.

One of the most important example of tries and PATRICIA tries are **suffix trees** and **compact suffix trees** (also called PAT). In suffix trees and compact suffix trees, the words stored in these trees are suffixes of a given string $X = x_1x_2\ldots$; that is, the word $X^j = x_jx_{j+1}x_{j+2}\ldots$ is the jth suffix of X which begins at the jth position of X. Thus a suffix tree is a trie and a compact suffix tree is a PATRICIA trie in which the words are all suffixes of a *given* string. Clearly, in this case the strings X^j for $j = 1, \ldots, n$ strongly depend on each other while in a trie the strings of $\mathcal{X}$ might be completely independent. A suffix tree is illustrated in Figure 1.2.

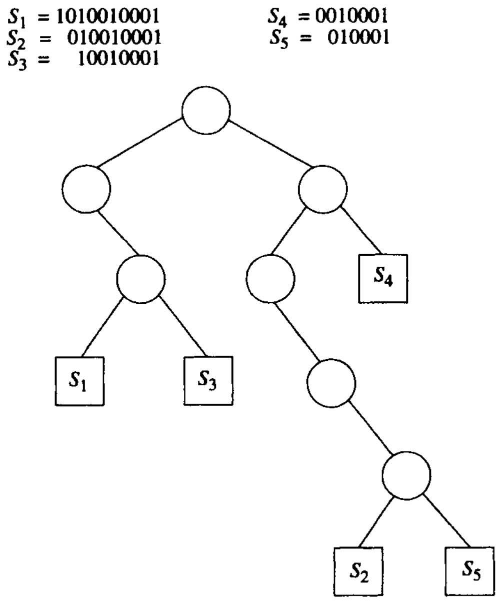


Figure 1.2. Suffix tree built from the first five suffixes $S_1, \ldots, S_5$ of $X = 0101101110\ldots$.

Certain characteristics of tries and suffix trees are of primary importance. We define them below.

Definition 1.1 (Digital Trees Parameters) *Let us consider a b-digital tree $\mathcal{D}_n^{(b)}$ built over n strings and capable of storing up to b strings in an (external) node.*

(i) *The mth depth $D_n^{(b)}(m)$ is the length of a path from the root of the digital tree to the (external) node containing the mth string.*

(ii) *The typical depth $D_n^{(b)}$ is the depth of a randomly selected string, that is,*

$$\Pr\{D_n^{(b)} \le k\} = \frac{1}{n}\sum_{m=1}^{n} \Pr\{D_n^{(b)}(m) \le k\}. \tag{1.1}$$

(iii) *The (external) path length $L_n^{(b)}$ is the sum of all depths, that is,*

$$L_n^{(b)} = \sum_{m=1}^{n} D_n^{(b)}(m). \tag{1.2}$$

(iv) *The height $H_n^{(b)}$ is the length of the longest depth, that is,*

$$H_n^{(b)} = \max_{1\le m\le n} \{D_n^{(b)}(m)\}. \tag{1.3}$$

(v) *The fill-up level $F_n^{(b)}$ is the maximal full level in the tree $\mathcal{D}_n^{(b)}$, that is, $\mathcal{D}_n^{(b)}$ is a full tree up to level $F_n^{(b)}$ but not on the level $F_n^{(b)}+1$. In other words, on levels $0 \le i \le F_n^{(b)}$ there are exactly V^i nodes (either external or internal) in the tree, but there are less than $V^{F_n^{(b)}+1}$ nodes on level $F_n^{(b)}+1$.*

(vi) *The shortest depth $s_n^{(b)}$ is the length of the shortest path from the root to an external node, that is,*

$$s_n^{(b)} = \min_{1\le m\le n} \{D_n^{(b)}(m)\}. \tag{1.4}$$

(Observe that $F_n^{(b)} \le s_n^{(b)}$.)

(vii) *The size $S_n^{(b)}$ is the number of (internal) nodes in $\mathcal{D}_n^{(b)}$.*

(viii) *The kth average profile $\bar{B}_n^{(b)}(k)$ is the average number of strings stored on level k of $\mathcal{D}_n^{(b)}$.*

These parameters can be conveniently represented in another way that reveals combinatorial relationships between strings stored in such digital trees. We start with the following definition.

Definition 1.2 **(Alignments)** *For the set of strings $\mathcal{X} = \{X^1, X^2, \dots, X^n\}$, the alignment $C_{i_1 \dots i_{b+1}}$ between $b+1$ strings $X^{i_1}, \dots, X^{i_{b+1}}$ is the length of the longest common prefix of all these $b+1$ strings.*

To illustrate this definition and show its usefulness to the analysis of digital trees, we consider the following example.

Example 1.1: *Illustration of the Self-Alignments* Let us consider the suffix tree shown in Figure 1.2. Define $C = \{C_{ij}\}_{i,j=1}^{5}$ for $b = 1$ as the self-alignment matrix which is shown explicitly below:

$$C = \begin{bmatrix} \star & 0 & 2 & 0 & 0 \\ 0 & \star & 0 & 1 & 4 \\ 2 & 0 & \star & 0 & 0 \\ 0 & 1 & 0 & \star & 1 \\ 0 & 4 & 0 & 1 & \star \end{bmatrix}.$$

Observe that we can express the parameters of Definition 1.1 in terms of the self-alignments C_{ij} as follows:

$$D_n(1) = \max_{2 \le j \le 5} \{C_{1j}\} + 1 = 3,$$

$$H_n = \max_{1 \le i < j \le n} \{C_{ij}\} + 1 = 5,$$

$$s_n = \min_{1 \le i \le n} \max_{1 \le j \le n} \{C_{ij}\} + 1 = 2$$

(since $b = 1$ we drop b from the above notations).

Certainly, similar relationships hold for tries, but not for PATRICIA tries and digital search trees. In the latter case, however, one can still express parameters of the trees in terms of the alignments matrix C. For example, the depth of the fourth string $D_4(4)$ can be expressed as follows:

$$D_4(4) = \max\{\min\{C_{41}, D_4(1)\}, \min\{C_{42}, D_4(2)\}, \min\{C_{43}, D_4(3)\}\}.$$

This is a bit too complicated to be of any help. ■

The above example suggests that there are relatively simple relationships between parameters of a trie and the alignments. Indeed, this is the case as the theorem below shows. The reader is asked to provide a formal proof in Exercise 1.

Theorem 1.3 *In a trie the following holds:*

$$D_n^{(b)}(i_{b+1}) = \max_{1 \le i_1, \dots, i_b \le n} \{C_{i_1 \dots i_{b+1}}\} + 1, \tag{1.5}$$

$$H_n^{(b)} = \max_{1 \le i_1, \dots, i_{b+1} \le n} \{C_{i_1 \dots i_{b+1}}\} + 1, \tag{1.6}$$

$$D_n^{(b)}(n+1) = \max_{1 \le i_1, \dots, i_b \le n} \{C_{i_1 \dots i_b, n+1}\} + 1, \tag{1.7}$$

$$s_n^{(b)} = \min_{1 \le i_{b+1} \le n} \{D_n^{(b)}(i_{b+1})\} = \min_{1 \le i_{b+1} \le n} \max_{1 \le i_1, \dots, i_b \le n} \{C_{i_1 \dots i_{b+1}}\} + 1 \tag{1.8}$$

for any $1 \le i_1, \dots, i_{b+1} \le n$, *where* $D_n^b(n+1) = I_n$ *is the depth of insertion.*

In passing, we should mention that the above combinatorial relationships find applications in problems not directly related to digital trees. We shall meet them again in Section 6.5.2 when analyzing the shortest common superstring problem described in Section 1.4.

The digital trees are used very extensively in this book as illustrative examples. We analyze the height of tries, PATRICIA tries, and suffix trees in Chapter 4. Recurrences arising in the analysis of digital trees are discussed in Chapter 7, while the typical depth of digital search trees is studied in Chapters 8–10.

1.2 DATA COMPRESSION: LEMPEL-ZIV ALGORITHMS

Source coding is an area of information theory (see Chapter 6) that deals with problems of optimal data compression. The most successful and best-known data compression schemes are due to Lempel and Ziv [460, 461]. Efficient implementation of these algorithms involves digital trees. We describe here some aspects of the Lempel-Ziv schemes and return to them in Chapter 6.

We start with some definitions. Consider a sequence $\{X_k\}_{k=1}^{\infty}$ taking values in a finite alphabet $\mathcal{A}$ (e.g., for English text the cardinality $|\mathcal{A}| = 26$ symbols, while for an image $|\mathcal{A}| = 256$). We write X_m^n to denote $X_m, X_{m+1} \dots X_n$. We encode X_1^n into a *binary* (compression) code $\mathscr{C}_n$, and the decoder produces the reproduction sequence $\hat{X}_1^n$ of X_1^n. More precisely, a code $\mathscr{C}_n$ is a function $\phi : \mathcal{A}^n \to \{0, 1\}^*$. On the decoding side, the decoder function $\psi : \{0, 1\}^* \to \mathcal{A}^n$ is applied to find $\hat{X}_1^n = \psi(\phi(X_1^n))$. Let $\ell(\mathscr{C}_n(X_1^n))$ be the length of the code $\mathscr{C}_n$ (in bits) representing X_1^n. Then the *bit rate* is defined as

$$r(X_1^n) = \frac{\ell(\mathscr{C}_n(X_1^n))}{n}.$$

For example, for text $r(X_1^n)$ is expressed in bits per symbol, while for image compression in bits per pixel or in short bpp.

We shall discuss below two basic Lempel-Ziv schemes, namely the so-called Lempel-Ziv'77 (LZ77) [460] and Lempel-Ziv'78 (LZ78) [461]. Both schemes are examples of **lossless** compression; that is, the decoder can recover exactly the encoded sequence. A number of interesting problems arise in lossy extensions of the Lempel-Ziv schemes. In the **lossy** data compression, discussed below, some information can be lost during the encoding.

1.2.1 Lempel-Ziv'77 Algorithm

The Lempel-Ziv algorithm partitions or parses a sequence into phrases that are similar in some sense. Depending on how such a parsing is encoded we have different versions of the algorithm. However, the basic idea is to find the longest prefix of yet uncompressed sequence that occurs in the already compressed sequence.

More specifically, let us assume that the first n symbols X_1^n are given to the encoder and the decoder. This initial string is sometimes called the "database string" or the "training sequence." Then we search for the longest prefix $X_{n+1}^{n+\ell}$ of X_{n+1}^{∞} that is repeated in X_1^n, that is,

Let l_n be the largest ℓ such that $X_{n+1}^{n+\ell=} = X_m^{m+\ell-1}$ for some prescribed range of m and ℓ.

Depending on the range of m and ℓ, we can have different versions of the LZ77 scheme. In the original LZ77 algorithm [460], m and ℓ were restricted to a window size W and "lookahead" buffer B, that is, $n-W+1 \leq m \leq n$ and $\ell \leq B$. This implementation is sometimes called the *sliding window* LZ77. In the *fixed database* (FDLZ) version [452, 453], one sets $1 \leq m \leq W$ and $m-1+\ell \leq W$; that is, the database sequence is fixed and the parser always looks for matches inside such a fixed substring X_1^W. Such a scheme is sometimes called the Wyner-Ziv scheme [452]. Finally, in the *growing database* version the only restriction is that $1 \leq m \leq n$ (i.e., the database consists of the last n symbols).

In general, the code built for LZ77 consists of the triple (m, ℓ, `char`) where `char` is the symbol $X_{m+\ell}$. Since the pointer to m needs $\log_2 n$ bits, the length ℓ could be coded in $O(\log l_n)$ bits and `char` requires $\log|\mathcal{A}|$ bits, the code length of a *phrase* is $\log_2 n + O(\log l_n) + \log_2|\mathcal{A}|$ bits. In Exercise 5 we propose a formula for the code length of the FDLZ, while in Exercise 6 the reader is asked to find the code length for all other versions of the Lempel-Ziv schemes.

The heart of all versions of the Lempel-Ziv schemes is the algorithm that finds the longest prefix of length l_n that occurs in the database string of length n. It turns out that the suffix tree discussed in Section 1.1 can be used to efficiently find such a prefix. Indeed, let us consider a sequence $X = 1010010001\ldots$, and assume X_1^4 is the database string. The suffix tree built over X_1^4 is shown in Figure 1.3. Let us now look for l_4, that is, the longest prefix of X_5^{∞} that occurs (starts) in the database X_1^4. In the growing database implementation it is X_5^8 since it is equal to

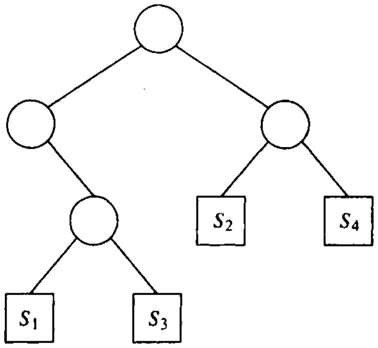


Figure 1.3. Suffix tree built from the first four suffixes of $X = 1010010001\ldots$.

X_2^5. This can be seen by inserting the fifth suffix of X into the suffix tree from Figure 1.3—which actually leads to the suffix tree shown in Figure 1.2.

1.2.2 Lempel-Ziv'78 Algorithm

The Lempel-Ziv'78 (LZ78) is a *dictionary-based* scheme that partitions a sequence into phrases (blocks) of variable sizes such that a new block is the shortest substring not seen in the past as a phrase. Every such phrase is encoded by the index of its prefix appended by a symbol, thus LZ78 code consists of pairs (`pointer, symbol`). A phrase containing only one symbol is coded with the index equal to zero.

Example 1.2: *The Lempel-Ziv'78 and Its Code* Consider the string $X_1^{14} =$ *ababbbabbaaaba* over the alphabet $\mathcal{A} = \{a, b\}$, which is parsed and coded as follows:

Phrase No:	1	2	3	4	5	6	7
Sequence:	(a)	(b)	(ab)	(bb)	(abb)	(aa)	(aba)
Code:	0a	0b	1b	2b	3b	1a	3a

Observe that we need $\lceil \log_2 7 \rceil$ bits to code a phrase, and two bits to code a symbol, so in total for 7 phrases we need 28 bits. ∎

The most time consuming part of the algorithm is finding the next phrase, that is, searching the dictionary. However, this can be speeded up by using a digital search tree to build the dictionary. For example, the string 11001010001000100 is parsed into (1)(10)(0)(101)(00)(01)(000)(100), and this process is represented in Figure 1.4 using the digital search tree structure. In this case, however, we

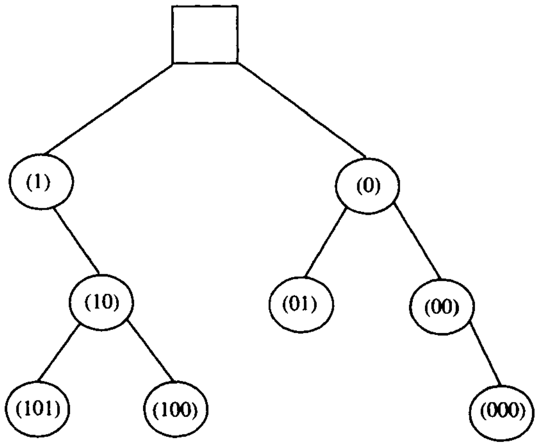


Figure 1.4. A digital tree representation of the Lempel-Ziv parsing for the string 11001010001000100.

leave the root empty (or we put an empty phrase into it). To show that the root is different from other nodes we draw it in Figure 1.4 as a square. All other phrases of the Lempel-Ziv parsing algorithm are stored in internal nodes (represented in the figure as circles). When a new phrase is created, the search starts at the root and proceeds down the tree as directed by the input symbols exactly in the same manner as in the digital tree construction (cf. Section 1.1). The search is completed when a branch is taken from an existing tree node to a new node that has not been visited before. Then the edge and the new node are added to the tree. The phrase is just a concatenation of symbols leading from the root to this node, which also stores the phrase.

We should observe differences between digital search trees discussed in Section 1.1 and the one described above. For the Lempel-Ziv scheme we consider a word of *fixed length*, say n, while before we dealt with *fixed number of strings*, say m, resulting in a digital tree consisting of exactly m nodes. Looking at Figure 1.4, we conclude that the number of nodes in the associated digital tree is equal to the number of phrases generated by the Lempel-Ziv algorithm.

1.2.3 Extensions of Lempel-Ziv Schemes

Finally, we shall discuss two extensions of Lempel-Ziv schemes, namely *generalized* Lempel-Ziv'78 and *lossy* Lempel-Ziv'77. Not only are these extensions useful from a practical point of view (cf. [11, 29, 299, 399, 361]), but they are also a source of interesting analytical problems. We return to them in Chapters 6, 9, and 10.

Generalized Lempel-Ziv'78 Let us first consider the **generalized** Lempel-Ziv'78 scheme. It is known that the original Lempel–Ziv scheme does not cope very well with sequences containing a long string of repeated symbols (i.e., the associated digital search tree is a skewed one with a long path). To somewhat remedy this situation, Louchard, Szpankowski and Tang [299] introduced a generalization of the Lempel–Ziv parsing scheme that works as follows: Fix an integer $b \geq 1$. The algorithm parses a sequence into phrases such that the next phrase is the *shortest* phrase seen in the past by *at most* $b - 1$ phrases ($b = 1$ corresponds to the original Lempel–Ziv algorithm). It turns out that such an extension of the Lempel-Ziv algorithm protects against the propagation of errors in the dictionary (cf. [399, 361]).

Example 1.3: *Generalized Lempel-Ziv'78* Consider the sequence

$$\alpha\beta\alpha\beta\beta\alpha\beta\alpha\beta\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\gamma$$

over the alphabet $\mathcal{A} = \{\alpha, \beta, \gamma\}$. For $b = 2$ it is parsed as follows:

$$(\alpha)(\beta)(\alpha)(\beta)(\beta\alpha)(\beta\alpha)(\beta\alpha\alpha)(\alpha\alpha)(\alpha\alpha)(\alpha\alpha\alpha)(\gamma)$$

that has seven *distinct* phrases and eleven phrases. The code for this new algorithm consists: (i) either of (`pointer, symbol`) when `pointer` refers to the *first* previous occurrence of the prefix of the phrase and `symbol` is the value of the last symbol of this phrase; (ii) or just (`pointer`) if the phrase has occurred previously (i.e., it is the second or the third or ... the bth occurrence of this phrase). For example, the code for the previously parsed sequence is for $b = 2$: $0\alpha0\beta122\alpha33\alpha1\alpha55\alpha0\gamma$ (e.g., the phrase (2α) occurs for the first time as a new phrase, hence (2) refers to the second distinct phrase appended by α, while code (5) represents a phrase that has its second occurrence as the fifth distinct phrase). Observe that this code is of length 47 bits since there are eleven phrases each requiring up to $\lceil \log_2 7 \rceil = 3$ bits and seven symbols need 14 additional bits (i.e., $47 = 11 \cdot 3 + 7 \cdot 2 = 47$). The original LZ78 code needs 54 bits. We saved 7 bits! But, the reader may verify that the same sequence requires only 46 bits for $b = 3$ (so only one additional bit is saved), while for $b = 4$ the bit count increases again to 52. ■

The above example suggests that $b = 3$ is (at least local) optimum for the above sequence. Can one draw similar conclusions "on average" for a typical sequence (i.e., generated randomly)? This book is intended to provide tools to analyze such problems.

As for the original Lempel-Ziv algorithm, the most time-consuming part of the construction is to generate a new phrase. An efficient way of accomplishing this

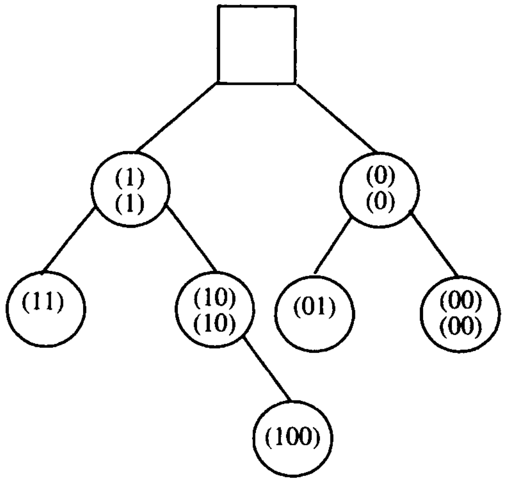


Figure 1.5. A 2-digital search tree representation of the generalized Lempel–Ziv parsing for the string 11001010001000010011.

is by means of generalized digital search trees, introduced in Section 1.1, namely b-digital search tree (b-DST). We recall that in such a digital tree one is allowed to store up to b strings in a node. In Figure 1.5 we show the 2-DST constructed from the sequence 11001010001000010011.

Lossy Extension of Lempel-Ziv'77 We now discuss another extension, namely a **lossy** Lempel-Ziv'77 scheme. In such a scheme in the process of encoding some information is lost. To control this loss, one needs a measure of fidelity $d(\cdot,\cdot)$ between two sequences. For example, the Hamming distance is defined as

$$d_n(X_1^n, \hat{X}_1^n) = \frac{1}{n}\sum_{i=1}^{n} d_1(X_i, \hat{X}_i)$$

where $d_1(X_i, \hat{X}_i) = 0$ for $X_i = \hat{X}_i$ and 1 otherwise. In the square error distortion we set $d(X_i, \hat{X}_i) = (X_i - \hat{X}_i)^2$.

Let us now fix $D > 0$. In the lossy LZ77, we consider the longest prefix of the uncompressed file that approximately (within distance D) occurs in the database sequence. More precisely, the quantity l_n defined in Section 1.2.1 becomes in this case:

Let l_n be the largest K such that a prefix of X_{n+1}^{∞} of length K is within distance D from X_i^{i-1+K} for some $1 \le i \le n - K + 1$, that is, $d(X_i^{i-1+K}, X_{n+1}^{n+K}) \le D$.

Not surprisingly, the bit rate of such a compression scheme depends on the probabilistic behavior of I_n. We shall analyze it in Chapter 6. The reader is also referred to [91, 278, 303, 403, 439].

1.3 PATTERN MATCHING

There are various kinds of patterns occurring in strings that are important to locate. These include squares, palindromes, and specific patterns. For example, in computer security one wants to know if a certain pattern (i.e., a substring, or even better a subsequence) appears (too) frequently in an audit file (text) since this may indicate an intrusion. In general, pattern matching involves a pattern H and a text string T. One is asked to determine the existence of H within T, the first occurrence of H, the number of occurrences or the location of all occurrences of H.

Two well-known pattern matching algorithms are the Knuth-Morris-Pratt (KMP) algorithm and the Boyer-Moore (BM) algorithm [3, 77]. In this section we focus on the former. The efficiency of these algorithms depends on how quickly one determines the location of the next matching attempt provided the previous attempt was unsuccessful. The key observation here is that following a mismatch at, say the kth position of the pattern, the preceding $k - 1$ symbols of the pattern and their structure give insight as to where the next matching attempt should begin. This idea is used in the KMP pattern matching algorithm and is illustrated in the following example and Figure 1.6.

Example 1.4: *The Morris–Pratt Algorithm* We now consider a simplified version of the KMP algorithm, namely that of the Morris–Pratt pattern matching algorithm. Let $H_1^6 = 011010$ and the text string $T_1^{10} = 1011011011$, as shown in

```
H   0 1 1 0 1 0
T   1 0 1 1 0 1 1 0 1 1
    (a) first attempt

H     0 1 1 0 1 0
T   1 0 1 1 0 1 1 0 1 1
    (b) second attempt

H           0 1 1 0 1 0
T   1 0 1 1 0 1 1 0 1 1
    (c) third attempt
```

Figure 1.6. Comparisons made by the Morris–Pratt pattern matching algorithm

Figure 1.6. When attempting to match P with T, we proceed from left to right, comparing each symbol. No match is made with the first symbol of each, so the pattern H is moved one position to the right. On the second attempt, the sixth symbol of H does not match the text, so this attempt is halted and the pattern H is shifted to the right. Notice that it is not fruitful to begin matching at either the third or fourth position of T since the suffix 01 of the so far matched pattern $H_1^5 = 01101$ is equal to the prefix 01 of H_1^5. Thus the next matching attempt begins at the fifth symbol of T. ■

Knowing how far to shift the pattern H is the key to both the KMP and the BM algorithms. Therefore, the pattern H is preprocessed to determine the shift. Let us assume that a mismatch occurs when comparing T_l with H_k. Then some alignment positions can be disregarded without further text-pattern comparisons. Indeed, let $1 \leq i \leq k$ be the largest integer such that $H_{k-i}^{k-1} = H_1^i$, that is, i is the longest prefix of H that is equal to the suffix of H^{k-1} of length i.Then positions $l-k+1, l-k+2, \ldots, l-i+1$ of the text do not need to be inspected and the pattern can be shifted by $k-i$ positions (as already observed in Figure 1.6). The set of such i can be known by a preprocessing of H.

There are different variants of the classic Knuth-Morris-Pratt algorithm [272] that differ by the way one uses the information obtained from the mismatching position. We formally define two variants, and provide an example. They can be described formally by assigning to them the so-called shift function S that determines by how much the pattern H can be shifted before the next comparison at $l+1$ is made. We have:

Morris-Pratt variant:

$$S = \min\{k : \ \min\{s > 0 : \ H_{1+s}^{k-1} = H_1^{k-1-s}\}\} \ ;$$

Knuth-Morris-Pratt variant:

$$S = \min\{k : \ \min\{s : \ H_{1+s}^{k-1} = H_1^{k-1-s} \text{ and } H_k \neq H_{k-s}\}\}$$

There are several parameters of pattern matching algorithms that either determine their performance or shed some light on their behaviors. For example, the efficiency of an algorithm is characterized by its complexity, defined below.

Definition 1.4

(i) *For any pattern matching algorithm that runs on a given text T and a given pattern H, let $M(l,k) = 1$ if the lth symbol T_l of the text is compared by the algorithm to the kth symbol H_k of the pattern, and $M(l,k) = 0$ otherwise.*