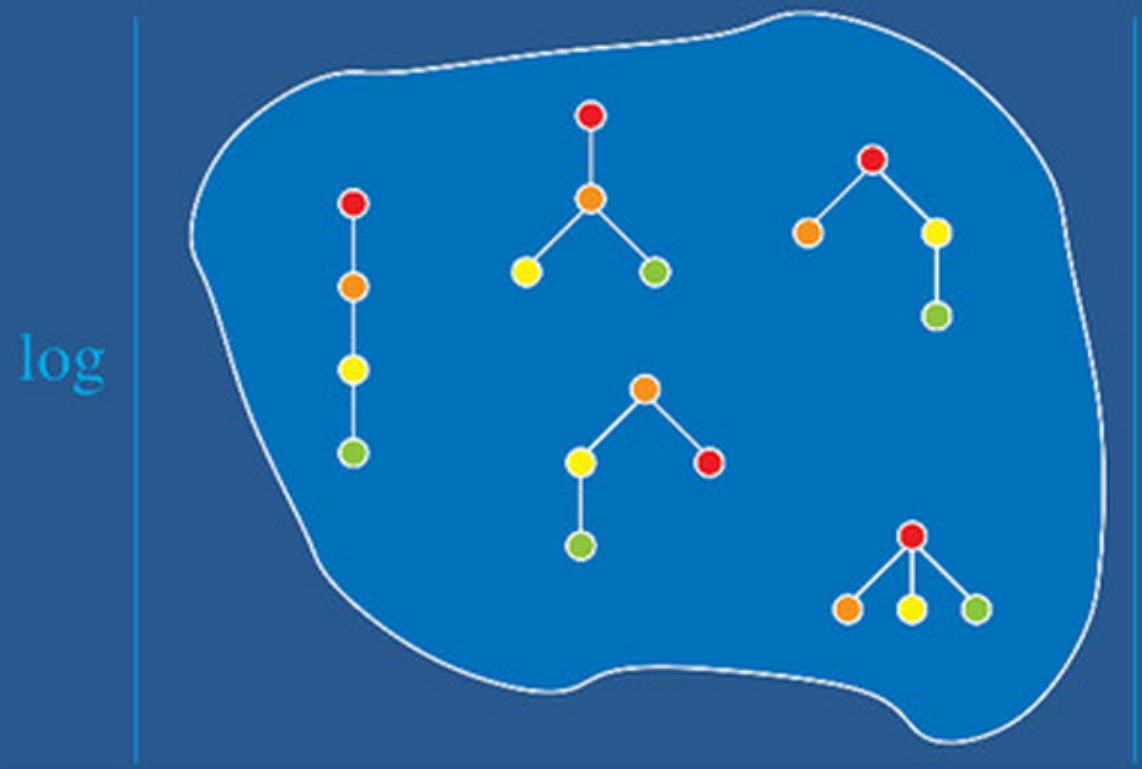
COMPACT DATA STRUCTURES

A PRACTICAL APPROACH



GONZALO NAVARRO

Compact Data Structures

A Practical Approach

Compact data structures help represent data in reduced space while allowing querying, navigating, and operating it in compressed form. They are essential tools for efficiently handling massive amounts of data by exploiting the memory hierarchy. They also reduce the resources needed in distributed deployments and make better use of the limited memory in low-end devices.

The field has developed rapidly, reaching a level of maturity that allows practitioners and researchers in application areas to benefit from the use of compact data structures. This first comprehensive book on the topic focuses on the structures that are most relevant for practical use. Readers will learn how the structures work, how to choose the right ones for their application scenario, and how to implement them. Researchers and students in the area will find in the book a definitive guide to the state of the art in compact data structures.

Gonzalo Navarro is Professor of Computer Science at the University of Chile. He has worked for 20 years on the relation between compression and data structures. He has directed or participated in numerous large projects on web research, information retrieval, compressed data structures, and bioinformatics. He is the Editor in Chief of the *ACM Journal of Experimental Algorithmics* and also a member of the editorial board of the journals *Information Retrieval* and *Information Systems*. His publications include the book *Flexible Pattern Matching in Strings* (with M. Raffinot), 20 book chapters, more than 100 journal papers and 200 conference papers; he has also chaired eight international conferences.

Compact Data Structures

A Practical Approach

Gonzalo Navarro

Department of Computer Science,
University of Chile

CAMBRIDGE
UNIVERSITY PRESS

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning, and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107152380

First published 2016

Printed in the United States of America by Sheridan Books, Inc.

A catalogue record for this publication is available from the British Library.

Library of Congress Cataloging-in-Publication Data
Names: Navarro, Gonzalo, 1969– author.
Title: Compact data structures : a practical approach / Gonzalo Navarro, Universidad de Chile.
Description: New York, NY : University of Cambridge, [2016] | Includes bibliographical references and index.
Identifiers: LCCN 2016023641 | ISBN 9781107152380 (hardback : alk. paper)
Subjects: LCSH: Data structures (Computer science) | Computer algorithms.
Classification: LCC QA76.9.D35 N38 2016 | DDC 005.7/3–dc23
LC record available at https://lccn.loc.gov/2016023641

ISBN 978-1-107-15238-0 Hardback

A Aylén, Facundo y Martina, que aún me creen.
A Betina, que aún me soporta.
A mi padre, a mi hermana, y a la memoria de mi madre.

Contents

List of Algorithms

Foreword

This is a delightful book on data structures that are both time and space efficient. Space as well as time efficiency is crucial in modern information systems. Even if we have extra space somewhere, it is unlikely to be close to the processors. The space used by most such systems is overwhelmingly for structural indexing, such as B-trees, hash tables, and various cross-references, rather than for "raw data." Indeed data, such as text, take far too much space in raw form and must be compressed. A system that keeps both data and indices in a compact form has a major advantage.

Hence the title of the book. Gonzalo Navarro uses the term "compact data structures" to describe a newly emerging research area. It has developed from two distinct but interrelated topics. The older is that of text compression, dating back to the work of Shannon, Fano, and Huffman (among others) in the late 1940s and early 1950s (although text compression as such was not their main concern). Through the last half of the 20th century, as the size of the text to be processed increased and computing platforms became more powerful, algorithmics and information theory became much more sophisticated. The goal of data compression, at least until the year 2000 or so, simply meant compressing information as well as possible and then decompressing each time it was needed. A hallmark of compact data structures is working with text in compressed form saving both decompression time and space. The newer contributing area evolved in the 1990s after the work of Jacobson and is generally referred to as "succinct data structures." The idea is to represent a combinatorial object, such as a graph, tree, or sparse bit vector, in a number of bits that differs from the information theory lower bound by only a lower order term. So, for example, a binary tree on n nodes takes only $2n + o(n)$ bits. The trick is to perform the necessary operations, e.g., find child, parent, or subtree size, in constant time.

Compact data structures take into account both "data" and "structures" and are a little more tolerant of "best effort" than one might be with exact details of information theoretic lower bounds. Here the subtitle, "A Practical Approach," comes into play. The emphasis is on methods that are reasonable to implement and appropriate for today's (and tomorrow's) data sizes, rather than on the asymptotics that one sees with the "theoretical approach."

Reading the book, I was taken with the thorough coverage of the topic and the clarity of presentation. Finding, easily, specific results was, well, easy, as suits the experienced researcher in the field. On the other hand, the careful exposition of key concepts, with elucidating examples, makes it ideal as a graduate text or for the researcher from a tangentially related area. The book covers the historical and mathematical background along with the key developments of the 1990s and early years of the current century, which form its core. Text indexing has been a major driving force for the area, and techniques for it are nicely covered. The final two chapters point to long-term challenges and recent advances. Updates to compact data structures have been a problem for as long as the topic has been studied. The treatment here is not only state of the art but will undoubtedly be a major influence on further improvements to dynamic structures, a key aspect of improving their applicability. The final chapter focuses on encodings, working with repetitive text, and issues of the memory hierarchy. The book will be a key reference and guiding light in the field for years to come.

J. Ian Munro
University of Waterloo

Acknowledgments

I am indebted to Joshimar Córdova and Simon Gog, who took the time to exhaustively read large portions of the book. They made a number of useful comments and killed many dangerous bugs. Several other students and colleagues read parts of the book and also made useful suggestions: Travis Gagie, Patricio Huepe, Roberto Konow, Susana Ladra, Veli Mäkinen, Miguel Ángel Martínez-Prieto, Ian Munro, and Alberto Ordóñez. Others, like Yakov Nekrich, Rajeev Raman, and Kunihiko Sadakane, saved me hours of searching by providing instant answers to my questions. Last but not least, Renato Cerro carefully polished my English grammar. It is most likely that some bugs remain, for which I am the only one to blame.

Ian Munro enthusiastically agreed to write the Foreword of the book. My thanks, again, to a pioneer of this beautiful area.

I would also like to thank my family for bearing with me along this two-year-long effort. It has been much more fun for me than for them.

Finally, I wish to thank the Department of Computer Science at the University of Chile for giving me the opportunity of a life dedicated to academia in a friendly and supportive environment.

CHAPTER 1

Introduction

1.1 Why Compact Data Structures?

Google's stated mission, "to organize the world's information and make it universally accessible and useful," could not better capture the immense ambition of modern society for gathering all kinds of data and putting them to use to improve our lives. We are collecting not only huge amounts of data from the physical world (astronomical, climatological, geographical, biological), but also human-generated data (voice, pictures, music, video, books, news, Web contents, emails, blogs, tweets) and society-based behavioral data (markets, shopping, traffic, clicks, Web navigation, likes, friendship networks).

Our hunger for more and more information is flooding our lives with data. Technology is improving and our ability to store data is growing fast, but the data we are collecting also grow fast – in many cases faster than our storage capacities. While our ability to store the data in secondary or perhaps tertiary storage does not yet seem to be compromised, performing the desired processing of these data in the main memory of computers is becoming more and more difficult. Since accessing a datum in main memory is about 10^5 times faster than on disk, operating in main memory is crucial for carrying out many data-processing applications.

In many cases, the problem is not so much the size of the actual data, but that of the *data structures* that must be built on the data in order to efficiently carry out the desired processing or queries. In some cases the data structures are one or two orders of magnitude larger than the data! For example, the DNA of a human genome, of about 3.3 billion bases, requires slightly less than 800 megabytes if we use only 2 bits per base (`A`, `C`, `G`, `T`), which fits in the main memory of any desktop PC. However, the suffix tree, a powerful data structure used to efficiently perform sequence analysis on the genome, requires at least 10 bytes per base, that is, more than 30 gigabytes.

The main techniques to cope with the growing size of data over recent years can be classified into three families:

Efficient secondary-memory algorithms. While accessing a random datum from disk is comparatively very slow, subsequent data are read much faster, only 100 times slower than from main memory. Therefore, algorithms that minimize the random accesses to the data can perform reasonably well on disk. Not every problem, however, admits a good disk-based solution.

Streaming algorithms. In these algorithms one goes to the extreme of allowing only one or a small number of sequential passes over the data, storing intermediate values on a comparatively small main memory. When only one pass over the data is allowed, the algorithm can handle situations in which the data cannot even be stored on disk, because they either are too large or flow too fast. In many cases streaming algorithms aim at computing approximate information from the data.

Distributed algorithms. These are parallel algorithms that work on a number of computers connected through a local-area network. Network transfer speeds are around 10 times slower than those of disks. However, some algorithms are amenable to parallelization in a way that the data can be partitioned over the processors and little transfer of data is needed.

Each of these approaches pays a price in terms of performance or accuracy, and neither one is always applicable. There are also cases where memory is limited and a large secondary memory is not at hand: routers, smartphones, smartwatches, sensors, and a large number of low-end embedded devices that are more and more frequently seen everywhere (indeed, they are the stars of the promised Internet of Things).

A topic that is strongly related to the problem of managing large volumes of data is *compression*, which seeks a way of representing data using less space. Compression builds on Information Theory, which studies the minimum space necessary to represent the data.

Most compression algorithms require decompressing all of the data from the beginning before we can access a random datum. Therefore, compression generally serves as a space-saving *archival* method: the data can be *stored* using less space but must be fully decompressed before being used again. Compression is not useful for managing more data in main memory, except if we need only to process the data sequentially.

Compact data structures aim precisely at this challenge. A compact data structure maintains the data, and the desired extra data structures over it, in a form that not only uses less space, but is able to access and query the data *in compact form*, that is, without decompressing them. Thus, a compact data structure allows us to fit and efficiently query, navigate, and manipulate much larger datasets in main memory than what would be possible if we used the data represented in plain form and classical data structures on top.

Compact data structures lie at the intersection of Data Structures and Information Theory. One looks at data representations that not only need space close to the minimum possible (as in compression) but also require that those representations allow one to efficiently carry out some operations on the data. In terms of information, data structures are fully *redundant*: they can be reconstructed from the data itself. However, they are built for efficiency reasons: once they are built from the data, data structures speed up operations significantly. When designing compact data structures, one struggles with this tradeoff: supporting the desired operations as efficiently as possible while

increasing the space as little as possible. In some lucky cases, a compact data structure reaches almost the minimum possible space to represent the data and provides a rich functionality that encompasses what is provided by a number of independent data structures. General trees and text collections are probably the two most striking success stories of compact data structures (and they have been combined to store the human genome *and* its suffix tree in less than 4 gigabytes!).

Compact data structures usually require more steps than classical data structures to complete the same operations. However, if these operations are carried out on a faster memory, the net result is a faster (and smaller) representation. This can occur at any level of the memory hierarchy; for example, a compact data structure may be faster because it fits in cache when the classical one does not. The most dramatic improvement, however, is seen when the compact data structure fits in main memory while the classical one needs to be handled on disk (even if it is a solid-state device). In some cases, such as limited-memory devices, compact data structures may be the only approach to operate on larger datasets.

The other techniques we have described can also benefit from the use of compact data structures. For example, distributed algorithms may use fewer computers to carry out the same task, as their aggregated memory is virtually enlarged. This reduces hardware, communication, and energy costs. Secondary-memory algorithms may also benefit from a virtually larger main memory by reducing the amount of disk transfers. Streaming algorithms may store more accurate estimations within the same main memory budget.

1.2 Why This Book?

The starting point of the formal study of compact data structures can be traced back to the 1988 Ph.D. thesis of Jacobson, although earlier works, in retrospect, can also be said to belong to this area. Since then, the study of these structures has fluorished, and research articles appear routinely in most conferences and journals on algorithms, compression, and databases. Various software repositories offer mature libraries implementing generic or problem-specific compact data structures. There are also indications of the increasing use of compact data structures inside the products of Google, Facebook, and others.

We believe that compact data structures have reached a level of maturity that deserves a book to introduce them. There are already established compact data structures to represent bitvectors, sequences, permutations, trees, grids, binary relations, graphs, tries, text collections, and others. Surprisingly, there are no other books on this topic as far as we know, and for many relevant structures there are no survey articles.

This book aims to introduce the reader to the fascinating algorithmic world of the compact data structures, with a strong emphasis on practicality. Most of the structures we present have been implemented and found to be reasonably easy to code and efficient in space and time. A few of the structures we present have not yet been implemented, but based on our experience we believe they will be practical as well. We have obtained the material from the large universe of published results and from our own experience, carefully choosing the results that should be most relevant to a

practitioner. Each chapter finishes with a list of selected references to guide the reader who wants to go further.

On the other hand, we do not leave aside the theory, which is essential for a solid understanding of why and how the data structures work, and thus for applying and extending them to face new challenges. We gently introduce the reader to the beauty of the algorithmics and the mathematics that are behind the study of compact data structures. Only a basic background is expected from the reader. From algorithmics, knowledge of sorting, binary search, dynamic programming, graph traversals, hashing, lists, stacks, queues, priority queues, trees, and $\mathcal{O}$-notation suffices (we will briefly review this notation later in this chapter). This material corresponds to a first course on algorithms and data structures. From mathematics, understanding of induction, basic combinatorics, probability, summations, and limits, that is, a first-year university course on algebra or discrete mathematics, is sufficient.

We expect this book to be useful for advanced undergraduate students, graduate students, researchers, and professionals interested in algorithmic topics. Hopefully you will enjoy the reading as much as I have enjoyed writing it.

1.3 Organization

The book is divided into 13 chapters. Each chapter builds on previous ones to introduce a new concept and includes a section on applications and a bibliographic discussion at the end. Applications are smaller or more specific problems where the described data structures provide useful solutions. Most can be safely skipped if the reader has no time, but we expect them to be inspiring. The bibliography contains annotated references pointing to the best sources of the material described in the chapter (which not always are the first publications), the most relevant historic landmarks in the development of the results, and open problems. This section is generally denser and can be safely skipped by readers not interested in going deeper, especially into the theoretical aspects.

Pseudocode is included for most of the procedures we describe. The pseudocode is presented in an algorithmic language, not in any specific programming language. For example, well-known variables are taken as global without notice, widely known procedures such as a binary search are not detailed, and tedious but obvious details are omitted (with notice). This lets us focus on the important aspects that we want the pseudocode to clear up; our intention is not that the pseudocode is a cut-and-paste text to get the structures running without understanding them. We refrain from making various programming-level optimizations to the pseudocode to favor clarity; any good programmer should be able to considerably speed up a verbatim implementation of the pseudocodes without altering their logic.

After this introductory chapter, Chapter 2 introduces the concepts of Information Theory and compression needed to follow the book. In particular, we introduce the concepts of worst-case, Shannon, and empirical entropy and their relations. This is the most mathematical part of the book. We also introduce Huffman codes and codes suitable for small integers.

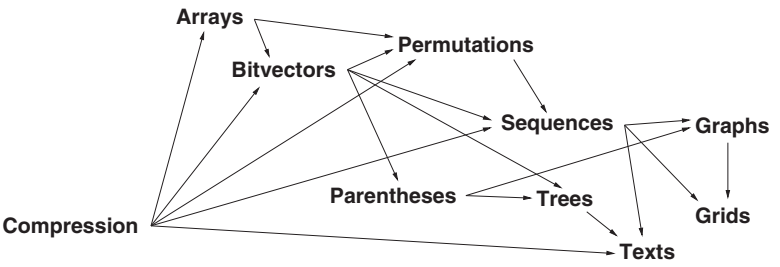


Figure 1.1. The most important dependencies among Chapters 2–11.

The subsequent chapters describe compact data structures for different problems. Each compact data structure stores some kind of data and supports a well-defined set of operations. Chapter 3 considers arrays, which support the operations of reading and writing values at arbitrary positions. Chapter 4 describes bitvectors, arrays of bits that in addition support a couple of bit-counting operations. Chapter 5 covers representations of permutations that support both the application of the permutation and its inverse as well as powers of the permutation. Chapter 6 considers sequences of symbols, which, apart from accessing the sequence, support a couple of symbol-counting operations. Chapter 7 addresses hierarchical structures described with balanced sequences of parentheses and operations to navigate them. Chapter 8 deals with the representation of general trees, which support a large number of query and navigation operations. Chapter 9 considers graph representations, both general ones and for some specific families such as planar graphs, allowing navigation toward neighbors. Chapter 10 considers discrete two-dimensional grids of points, with operations for counting and reporting points in a query rectangle. Chapter 11 shows how text collections can be represented so that pattern search queries are supported.

As said, each chapter builds upon the structures described previously, although most of them can be read independently with only a conceptual understanding of what the operations on previous structures mean. Figure 1.1 shows the most important dependencies for understanding why previous structures reach the claimed space and time performance.

These chapters are dedicated to *static* data structures, that is, those that are built once and then serve many queries. These are the most developed and generally the most efficient ones. We pay attention to construction time and, especially, construction space, ensuring that structures that take little space can also be built within little extra memory, or that the construction is disk-friendly. Structures that support updates are called *dynamic* and are considered in Chapter 12.

The book concludes in Chapter 13, which surveys some current research topics on compact data structures: encoding data structures, indexes for repetitive text collections, and data structures for secondary storage. Those areas are not general or mature enough to be included in previous chapters, yet they are very promising and will probably be the focus of much research in the upcoming years. The chapter then also serves as a guide to current research topics in this area.

Although we have done our best to make the book error-free, and have manually verified the algorithms several times, it is likely that some errors remain. A Web page with comments, updates, and corrections on the book will be maintained at http://www.dcc.uchile.cl/gnavarro/CDSbook.

1.4 Software Resources

Although this book focuses on understanding the compact data structures so that the readers can implement them by themselves, it is worth noting that there are several open-source software repositories with mature implementations, both for general and for problem-specific compact data structures. These are valuable both for practitioners that need a structure implemented efficiently, well tested, and ready to be used, and for students and researchers that wish to build further structures on top of them. In both cases, understanding why and how each structure works is essential to making the right decisions on which structure to use for which problem, how to parameterize it, and what can be expected from it.

Probably the most general, professional, exhaustive, and well tested of all these libraries is Simon Gog's *Succinct Data Structure Library (SDSL)*, available at https://github.com/simongog/sdsl-lite. It contains `C++` implementations of compact data structures for bitvectors, arrays, sequences, text indexes, trees, range minimum queries, and suffix trees, among others. The library includes tools to verify correctness and measure efficiency along with tutorials and examples.

Another generic library is Francisco Claude's *Library of Compact Data Structures (LIBCDS)*, available at https://github.com/fclaude/libcds. It contains optimized and well-tested `C++` implementations of bitvectors, sequences, permutations, and others. A tutorial on how to use the library and how it works is included.

Sebastiano Vigna's *Sux* library, available at http://sux.di.unimi.it, contains high-quality `C++` and/or `Java` implementations of various compact data structures, including bitvectors, arrays with cells of varying lengths, and (general and monotone) minimal perfect hashing. Other projects accessible from there include sophisticated tools to manage inverted indexes and Web graphs in compressed form.

Giuseppe Ottaviano's *Succinct* library provides efficient `C++` implementations of bitvectors, arrays of fixed and variable-length cells, range minimum queries, and others. It is available at https://github.com/ot/succinct.

Finally, Nicola Prezza's *Dynamic* library provides C++ implementations of various data structures supporting insertions of new elements: partial sums, bitvectors, sparse arrays, strings, and text indexes. It is available at https://github.com/nicolaprezza/DYNAMIC.

The authors of many of these libraries have explored much deeper practical aspects of the implementation, including cache efficiency, address translation, word alignments, machine instructions for long computer words, instruction pipelining, and other issues beyond the scope of this book.

Many other authors of articles on practical compact data structures for specific problems have left their implementations publicly available or are willing to share them upon request. There are too many to list here, but browsing the personal pages

of the authors, or requesting the code, is probably a fast way to obtain a good implementation.

1.5 Mathematics and Notation

This final technical section is a reminder of the mathematics behind the $\mathcal{O}$-notation, which we use to describe the time performance of algorithms and the space usage of data structures. We also introduce other notation used throughout the book.

***$\mathcal{O}$*-notation.** This notation is used to describe the asymptotic growth of functions (for example, the cost of an algorithm as a function of the size of the input) in a way that considers only sufficiently large values of the argument (hence the name "asymptotic") and ignores constant factors.

Formally, $\mathcal{O}(f(n))$ is the set of all functions $g(n)$ for which there exist constants $c > 0$ and $n_0 > 0$ such that, for all $n > n_0$, it holds $|g(n)| \leq c \cdot |f(n)|$. We say that $g(n)$ is $\mathcal{O}(f(n))$, meaning that $g(n) \in \mathcal{O}(f(n))$. Thus, for example, $3n^2 + 6n - 3$ is $\mathcal{O}\big(n^2\big)$ and also $\mathcal{O}\big(n^3\big)$, but it is not $\mathcal{O}(n \log n)$. In particular, $\mathcal{O}(1)$ is used to denote a function that is always below some constant. For example, the cost of an algorithm that, independently of the input size, performs 3 accesses to tables and terminates is $\mathcal{O}(1)$. An algorithm taking $\mathcal{O}(1)$ time is said to be constant-time.

It is also common to abuse the notation and write $g(n) = \mathcal{O}(f(n))$ to mean $g(n) \in \mathcal{O}(f(n))$, and even to write, say, $g(n) < 2n + \mathcal{O}(\log n)$, meaning that $g(n)$ is smaller than $2n$ plus a function that is $\mathcal{O}(\log n)$. Sometimes we will write, for example, $g(n) = 2n - \mathcal{O}(\log n)$, to stress that $g(n) \leq 2n$ and the function that separates $g(n)$ from $2n$ is $\mathcal{O}(\log n)$.

Several other notations are related to $\mathcal{O}$. Mostly for lower bounds, we write $g(n) \in \Omega(f(n))$, meaning that there exist constants $c > 0$ and $n_0 > 0$ such that, for all $n > n_0$, it holds $|g(n)| \geq c \cdot |f(n)|$. Alternatively, we can define $g(n) \in \Omega(f(n))$ iff $f(n) \in \mathcal{O}(g(n))$. We say that $g(n)$ is $\Theta(f(n))$ to mean that $g(n)$ is $\mathcal{O}(f(n))$ and also $\Omega(f(n))$. This means that both functions grow, asymptotically, at the same speed, except for a constant factor.

To denote functions that are asymptotically negligible compared to $f(n)$, we use $g(n) = o(f(n))$, which means that $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$. For example, saying that a data structure uses $2n + o(n)$ bits means that it uses $2n$ plus a number of bits that grows sublinearly with n, such as $2n + \mathcal{O}(n/\log n)$. The notation $o(1)$ denotes a function that tends to zero as n tends to infinity, for example, $\log\log n/\log n = o(1)$. Finally, the opposite of the $o(\cdot)$ notation is $\omega(\cdot)$, where $g(n) = \omega(f(n))$ iff $f(n) = o(g(n))$. In particular, $\omega(1)$ denotes a function that tends to infinity (no matter how slowly) when n tends to infinity. For example, $\log\log n = \omega(1)$.

When several variables are used, as in $o(n \log \sigma)$, it must be clear to which the $o(\cdot)$ notation refers. For example, $n \log\log \sigma$ is $o(n \log \sigma)$ if the variable is σ, or if the variable is n but σ grows with n (i.e., $\sigma = \omega(1)$ as a function of n). Otherwise, if we refer to n but σ is a constant, then $n \log\log \sigma$ is not $o(n \log \sigma)$.

These notations are also used on decreasing functions of n, to describe error margins. For example, we may approximate the harmonic number $H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln$

$n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \ldots$, where $\gamma \approx 0.577$ is a constant, with any of the following formulas, having a decreasing level of detail:[1]

$$\begin{aligned} H_n &= \ln n + \gamma + \frac{1}{2n} + \mathcal{O}\left(\frac{1}{n^2}\right) \\ &= \ln n + \gamma + \mathcal{O}\left(\frac{1}{n}\right) \\ &= \ln n + \mathcal{O}(1) \\ &= \mathcal{O}(\log n), \end{aligned}$$

depending on the degree of accuracy we want. We can also use $o(\cdot)$ to give less details about the error level, for example,

$$\begin{aligned} H_n &= \ln n + \gamma + \frac{1}{2n} + o\left(\frac{1}{n}\right) \\ &= \ln n + \gamma + o(1) \\ &= \ln n + o(\log n). \end{aligned}$$

We can also write the error in relative form, for example,

$$\begin{aligned} H_n &= \ln n + \gamma + \frac{1}{2n} \cdot \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right) \\ &= \ln n \cdot \left(1 + \mathcal{O}\left(\frac{1}{\log n}\right)\right) \\ &= \ln n \cdot (1 + o(1)). \end{aligned}$$

When using the notation to denote errors, the inequality $\frac{1}{1+x} = 1 - x + x^2 - \ldots = 1 - \mathcal{O}(x)$, for any $0 < x < 1$, allows us to write $\frac{1}{1+o(1)} = 1 + o(1)$, which is useful for moving error terms from the denominator to the numerator.

Logarithm. This is a very important function in Information Theory, as it is the key to describing the entropy, or amount of information, in an object. When the entropy (or information) is described in bits, the logarithm must be to the base 2. We use log to denote the logarithm to the base 2. When we use a logarithm to some other base b, we write $\log_b$. As shown, the natural logarithm is written as ln. Of course, the base of the logarithm makes no difference inside $\mathcal{O}$-formulas (unless it is in the exponent!).

The inequality $\frac{x}{1+x} \leq \ln(1+x) \leq x$ is useful in many cases, in particular in combination with the $\mathcal{O}$-notation. For example,

$$\ln(n(1 + o(1))) = \ln n + \ln(1 + o(1)) \leq \ln n + o(1).$$

It also holds

$$\ln(n(1 + o(1))) \geq \ln n + \frac{o(1)}{1 + o(1)} = \ln n + o(1).$$

[1] In the first line, we use the fact that the tail of the series converges to $\frac{c}{n^2}$, for some constant c.

Therefore, $\ln(n(1+o(1))) = \ln n + o(1)$. More generally, if $f(n) = o(1)$, and b is any constant, we can write $\log_b(n(1+f(n))) = \log_b n + \mathcal{O}(f(n))$. For example, $\log(n + \log n) = \log n + \mathcal{O}(\log n/n)$.

Model of computation. We consider realistic computers, with a computer word of w bits, where we can carry out in constant time all the basic arithmetic ($+$, $-$, $\cdot$, $/$, mod, ceilings and floors, etc.) and logic operations (bitwise *and*, *or*, *not*, *xor*, bit shifts, etc.). In modern computers w is almost always 32 or 64, but several architectures allow for larger words to be handled natively, reaching, for example, 128, 256, or 512 bits.

When connecting with theory, this essentially corresponds to the RAM model of computation, where we do not pay attention to restrictions in some branches of the RAM model that are unrealistic on modern computers (for example, some variants disallow multiplication and division). In the RAM model, it is usually assumed that the computer word has $w = \Theta(\log n)$ bits, where n is the size of the data in memory. This logarithmic model of growth of the computer word is appropriate in practice, as w has been growing approximately as the logarithm of the size of main memories. It is also reasonable to expect that we can store any memory address in a constant number of words (and in constant time).

For simplicity and practicality, we will use the assumption $w \geq \log n$, which means that with one computer word we can address any data element. While the assumption $w = \mathcal{O}(\log n)$ may also be justified (we may argue that the data should be large enough for the compact storage problem to be of interest), this is not always the case. For example, the dynamic structures (Chapter 12) may grow and shrink over time. Therefore, we will not rely on this assumption. Thus, for example, we will say that the cost of an algorithm that inspects n bits by chunks of w bits, processing each chunk in constant time, is $\mathcal{O}(n/w) = \mathcal{O}(n/\log n) = o(n)$. Instead, we will not take an $\mathcal{O}(w)$-time algorithm to be $\mathcal{O}(\log n)$.

Strings, sequences, and intervals. In most cases, our arrays start at position 1. With $[a, b]$ we denote the set $\{a, a+1, a+2, \ldots, b\}$, unless we explicitly imply it is a real interval. For example, $A[1, n]$ denotes an array of n elements $A[1], A[2], \ldots, A[n]$. A *string* is an array of elements drawn from a finite universe, called the *alphabet*. Alphabets are usually denoted $\Sigma = [1, \sigma]$, where σ is some integer, meaning that $\Sigma = \{1, 2, \ldots, \sigma\}$. The alphabet elements are called *symbols*, *characters*, or *letters*. The *length* of the string $S[1, n]$ is $|S| = n$. The set of all the strings of length n over alphabet Σ is denoted Σ^n, and the set of all the strings of any length over Σ is denoted $\Sigma^* = \cup_{n \geq 0} \Sigma^n$. Strings and sequences are basically synonyms in this book; however, substring and subsequence are different concepts. Given a string $S[1, n]$, a *substring* $S[i, j]$ is, precisely, the array $S[i], S[i+1], \ldots, S[j]$. Particular cases of substrings are *prefixes*, of the form $S[1, j]$, and *suffixes*, of the form $S[i, n]$. When $i > j$, $S[i, j]$ denotes the empty string ε, that is, the only string of length zero. A *subsequence* is more general than a substring: it can be any $S[i_1] \,.\, S[i_2] \ldots S[i_r]$ for $i_1 < i_2 < \ldots < i_r$, where we use the dot to denote concatenation of symbols (we might also simply write one symbol after the other, or mix strings and symbols in a concatenation). Sometimes we will also use $\langle a, b \rangle$ to denote the same as $[a, b]$ or write sequences as $\langle a_1, a_2, \ldots, a_n \rangle$. Finally, given a string $S[1, n]$, S^{rev} denotes the reversed string, $S[n] \,.\, S[n-1] \ldots S[2] \,.\, S[1]$.

1.6 Bibliographic Notes

Growth of information and computing power. Google's mission is stated in http://www.google.com/about/company.

There are many sources that describe the amount of information the world is gathering. For example, a 2011 study from *International Data Corporation (IDC)* found that we are generating a few zettabytes per year (a zettabyte is 2^{70}, or roughly 10^{21}, bytes), and that data are more than doubling per year, outperforming Moore's law (which governs the growth of hardware capacities).[2] A related discussion from 2013, arguing that we are much better at storing than at using all these data, can be read in *Datamation*.[3] For a shocking and graphical message, the 2012 poster of *Domo* is also telling.[4]

There are also many sources about the differences in performance between CPU, caches, main memory, and secondary storage, as well as how these have evolved over the years. In particular, we used the book of Hennessy and Patterson (2012, Chap. 1) for the rough numbers shown here.

Examples of books about the mentioned algorithmic approaches to solve the problem of data growth are, among many others, Vitter (2008) for secondary-memory algorithms, Muthukrishnan (2005) for streaming algorithms, and Roosta (1999) for distributed algorithms.

Suffix trees. The book by Gusfield (1997) provides a good introduction to suffix trees in the context of bioinformatics. Modern books pay more attention to space issues and make use of some of the compact data structures we describe here (Ohlebusch, 2013; Mäkinen *et al.*, 2015). Our size estimates for compressed suffix trees are taken from the Ph.D. thesis of Gog (2011).

Compact data structures. Despite some previous isolated results, the Ph.D. thesis of Jacobson (1988) is generally taken as the starting point of the systematic study of compact data structures. Jacobson coined the term *succinct data structure* to denote a data structure that uses $\log N + o(\log N)$ bits, where N is the total number of different objects that can be encoded. For example, succinct data structures for arrays of n bits must use $n + o(n)$ bits, since $N = 2^n$. To exclude mere data compressors, succinct data structures are sometimes required to support queries in constant time (Munro, 1996).

In this book we use the term *compact data structure*, which refers to the broader class of data structures that aim at using little space and query time. Other related terms are used in the literature (not always consistently) to refer to particular subclasses of data structures (Ferragina and Manzini, 2005; Gál and Miltersen, 2007; Fischer and Heun, 2011; Raman, 2015): *compressed* or *opportunistic* data structures are those using $\mathcal{H} + o(\log N)$ bits, where $\mathcal{H}$ is the entropy of the data under some compression model (such as the bit array representations we describe in Section 4.1.1); data structures using $\mathcal{H} + o(\mathcal{H})$ bits are sometimes called *fully compressed* (for example, the Huffman-shaped wavelet trees of Section 6.2.4 are almost fully compressed). A data structure that adds

[2] http://www.emc.com/about/news/press/2011/20110628-01.htm.
[3] http://www.datamation.com/applications/big-data-analytics-overview.html.
[4] http://www.domo.com/blog/2012/06/how-much-data-is-created-every-minute.

$o(\log N)$ bits and operates on any raw data representation that offers basic access (such as the rank and select structures for bitvectors in Sections 4.2 and 4.3) is sometimes called a *succinct index* or a *systematic* data structure. Many indexes supporting different functionalities may coexist over the same raw data, adding up to $\log N + o(\log N)$ bits. A *non-systematic* or *encoding* data structure, instead, needs to encode the data in a particular format, which may be unsuitable for another non-systematic data structure (like the wavelet trees of Section 6.2); in exchange it may use less space than the best systematic data structure (see Section 4.7 for the case of bitvectors). A data structure that uses $o(\log N)$ bits and does not need to access to the data at all is also called *non-systematic* or an *encoding*, in the sense that it does not access the raw data. Such small encodings are special, however, because they cannot possibly reproduce the original data; they answer only some types of queries on it (an example is given in Section 7.4.1; then we study encodings with more detail in Section 13.1).

The second edition of the *Encyclopedia of Algorithms* (Kao, 2016) contains good short surveys on many of the structures we discuss in the book.

Required knowledge. Good books on algorithms, which serve as a complement to follow this book, are by Cormen *et al.* (2009), Sedgewick and Wayne (2011), and Aho *et al.* (1974), among too many others to cite here. The last one (Aho *et al.*, 1974) is also a good reference for the RAM model of computation. Authoritative sources on algorithmics (yet possibly harder to read for the novice) are the monumental works of Knuth (1998) and Mehlhorn (1984). Books on algorithms generally cover analysis and $\mathcal{O}$-notation as well. Rawlins (1992) has a nice book that is more focused on analysis. The books by Graham *et al.* (1994) and by Sedgewick and Flajolet (2013) give a deeper treatment, and the handbook by Abramowitz and Stegun (1964) is an outstanding reference. Cover and Thomas (2006) offer an excellent book on Information Theory and compression fundamentals; we will cover the required concepts in Chapter 2.

Implementations. Some of the compact data structure libraries we have described have associated publications, for example, Gog's (Gog and Petri, 2014) and Ottaviano's (Grossi and Ottaviano, 2013). Another recent publication (Agarwal *et al.*, 2015) reports on *Succinct*, a distributed string store for column-oriented databases that supports updates and sophisticated string searches, achieving high performance through the use of compact data structures. No public code is reported for the latter, however.

A couple of recent articles hint at the interest inside Google for the development of compact tries (Chapter 8) for speech recognition in Android devices (Lei *et al.*, 2013) and for machine translation (Sorensen and Allauzen, 2011). A related implementation, called MARISA tries, is available at https://code.google.com/p/marisa-trie.

Facebook's *Folly* library (https://github.com/facebook/folly) now contains an implementation of Elias-Fano codes (Chapter 3).[5]

An example of the use of compressed text indexes (Chapter 11) in bioinformatic applications is the Burrows-Wheeler Aligner (BWA) software (Li and Durbin, 2010), available from http://bio-bwa.sourceforge.net.

[5] https://github.com/facebook/folly/blob/master/folly/experimental/EliasFanoCoding.h

Bibliography

Abramowitz, M. and Stegun, I. A. (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, 9th edition.

Agarwal, R., Khandelwal, A., and Stoica, I. (2015). Succinct: Enabling queries on compressed data. In *Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 337–350.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3rd edition.

Cover, T. and Thomas, J. (2006). *Elements of Information Theory*. Wiley, 2nd edition.

Ferragina, P. and Manzini, G. (2005). Indexing compressed texts. *Journal of the ACM*, **52**(4), 552–581.

Fischer, J. and Heun, V. (2011). Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, **40**(2), 465–492.

Gál, A. and Miltersen, P. B. (2007). The cell probe complexity of succinct data structures. *Theoretical Computer Science*, **379**(3), 405–417.

Gog, S. (2011). *Compressed Suffix Trees: Design, Construction, and Applications*. Ph.D. thesis, Ulm University, Germany.

Gog, S. and Petri, M. (2014). Optimized succinct data structures for massive data. *Software Practice and Experience*, **44**(11), 1287–1314.

Graham, R. L., Knuth, D. E., and Patashnik, O. (1994). *Concrete Mathematics – A Foundation for Computer Science*. Addison-Wesley, 2nd edition.

Grossi, R. and Ottaviano, G. (2013). Design of practical succinct data structures for large data collections. In *Proc. 12th International Symposium on Experimental Algorithms (SEA)*, LNCS 7933, pages 5–17.

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

Hennessy, J. L. and Patterson, D. A. (2012). *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 5th edition.

Jacobson, G. (1988). *Succinct Data Structures*. Ph.D. thesis, Carnegie Mellon University.

Kao, M.-Y., editor (2016). *Encyclopedia of Algorithms*. Springer, 2nd edition.

Knuth, D. E. (1998). *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition.

Lei, X., Senior, A., Gruenstein, A., and Sorensen, J. (2013). Accurate and compact large vocabulary speech recognition on mobile devices. In *Proc. 14th Annual Conference of the International Speech Communication Association (INTERSPEECH)*, pages 662–665.

Li, H. and Durbin, R. (2010). Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, **26**(5), 589–595.

Mäkinen, V., Belazzougui, D., Cunial, F., and Tomescu, A. I. (2015). *Genome-Scale Algorithm Design*. Cambridge University Press.

Mehlhorn, K. (1984). *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Munro, J. I. (1996). Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42.

Muthukrishnan, S. (2005). *Data Streams: Algorithms and Applications*. Now Publishers.

Ohlebusch, E. (2013). *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag.

Raman, R. (2015). Encoding data structures. In *Proc. 9th International Workshop on Algorithms and Computation (WALCOM)*, LNCS 8973, pages 1–7.

Rawlins, G. J. E. (1992). *Compared to What? An Introduction to the Analysis of Algorithms*. Computer Science Press.

Roosta, S. H. (1999). *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer.

Sedgewick, R. and Flajolet, P. (2013). *An Introduction to the Analysis of Algorithms*. Addison-Wesley-Longman, 2nd edition.

Sedgewick, R. and Wayne, K. (2011). *Algorithms*. Addison-Wesley, 4th edition.

Sorensen, J. and Allauzen, C. (2011). Unary data structures for language models. In *Proc. 12th Annual Conference of the International Speech Communication Association (INTERSPEECH)*, pages 1425–1428.

Vitter, J. S. (2008). *Algorithms and Data Structures for External Memory*. Now Publishers.

CHAPTER 2

Entropy and Coding

In this chapter we cover some minimal notions of Information Theory and Data Compression required to understand the compact data structures we present in the book. We offer further pointers at the end of the chapter.

In broad terms, the *entropy* is the minimum number of bits needed to unambiguously identify an object from a set. The entropy is then a lower bound to the space used by the compressed representation of an object. The holy grail of compressed data structures is to use essentially the space needed to identify the objects, but choosing a representation that makes it easy to answer queries on them.

2.1 Worst-Case Entropy

The most basic notion of entropy is that it is the minimum number of bits required by identifiers, called *codes*, if we assign a unique code to each element of a set $\mathcal{U}$ and all the codes are of the same length.

This is called the *worst-case* entropy of $\mathcal{U}$ and is denoted $\mathcal{H}_{\mathrm{wc}}(\mathcal{U})$. It is easy to see that

$$\mathcal{H}_{\mathrm{wc}}(\mathcal{U}) = \log |\mathcal{U}|$$

bits (recall that log is the logarithm in base 2): If we used codes of length $\ell < \mathcal{H}_{\mathrm{wc}}(\mathcal{U})$, we would have only $2^\ell < 2^{\mathcal{H}_{\mathrm{wc}}(\mathcal{U})} = 2^{\log |\mathcal{U}|} = |\mathcal{U}|$ distinct codes, which would be insufficient for giving a distinct code to each element in $\mathcal{U}$.

Therefore, if all the codes have the same length, this length must be at least $\lceil \mathcal{H}_{\mathrm{wc}}(\mathcal{U}) \rceil$ bits. If they are of different lengths, then the *longest* ones still must use at least $\lceil \mathcal{H}_{\mathrm{wc}}(\mathcal{U}) \rceil$ bits. This explains the adjective "worst-case."

For example, the worst-case entropy of all the sequences of n bits is $\log(2^n) = n$ bits, whereas the worst-case entropy of all the strings of length n over alphabet $\Sigma = [1, \sigma]$ is $\log(\sigma^n) = n \log \sigma$ bits. That is, one needs that many bits to encode *any possible* sequence of n symbols.

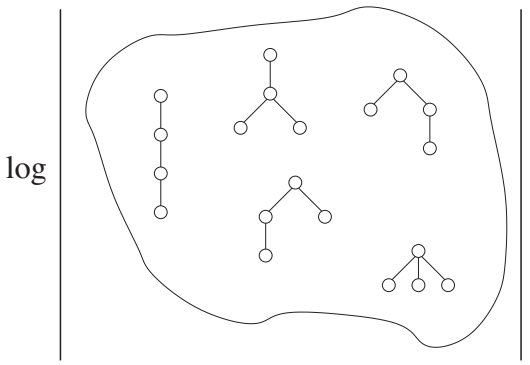


Figure 2.1. The worst-case entropy of $\mathcal{T}_4$, the general ordinal trees of 4 nodes.

As a more exciting example, consider the worst-case entropy of the set of all the general ordinal trees of n nodes, $\mathcal{T}_n$. In an ordinal tree, each node has an arbitrary number of children and distinguishes their order. It is known that the number of general ordinal trees is

$$|\mathcal{T}_n| = \frac{1}{n}\binom{2n-2}{n-1},$$

which is the $(n-1)$th Catalan number. By using Stirling's approximation to the factorial, $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\mathcal{O}\left(\frac{1}{n}\right)\right)$, we have

$$|\mathcal{T}_n| = \frac{(2n-2)!}{n!(n-1)!} = \frac{(2n-2)^{2n-2}e^n e^{n-1}}{e^{2n-2}n^n(n-1)^{n-1}\sqrt{\pi n}}\left(1+\mathcal{O}\left(\frac{1}{n}\right)\right) = \frac{4^n}{n^{3/2}}\cdot\Theta(1),$$

and therefore

$$\mathcal{H}_{\mathrm{wc}}(\mathcal{T}_n) = \log|\mathcal{T}_n| = 2n - \Theta(\log n)$$

is the minimum number of bits into which any general ordinal tree of n nodes can be encoded.

Example 2.1 *Figure 2.1 illustrates the worst-case entropy of* $\mathcal{T}_4$. *This is* $\mathcal{H}_{\mathrm{wc}}(\mathcal{T}_4) = \log\left(\frac{1}{4}\binom{6}{3}\right) = \log 5 \approx 2.322$.

Unlike the examples of bit sequences and strings, the classical representations of trees are very far from this number; actually they use at least n *pointers*. Since such pointers must distinguish among the n different nodes, by the same reasoning on the entropy they must use at least $\log n$ bits. Therefore, classical representations of trees use at least $n\log n$ bits, whereas $2n$ bits should be sufficient to represent any tree of n nodes.

Indeed, it is not very difficult to encode a general tree using $2n$ bits: traverse the tree in depth-first-order, writing a 1 upon arriving at a node, and a 0 when leaving it in the recursion. It is an easy exercise to see that one can recover the tree from the resulting stream of $2n$ 1s and 0s.

Example 2.2 *The* 5 *trees of Figure 2.1 would be encoded as* 11110000, 11010100, 11101000, 11011000, *and* 11100100. *We leave as an exercise to the reader to identify which tree corresponds to which code.*

It is much more difficult, however, to *navigate* this representation in constant time per operation; for example, going to the parent or to a child of a node. This is the whole point of compact data structures: to represent combinatorial objects within their entropy space (or close), not for the sole purpose of compressing them, but also with the aim of navigating and querying them efficiently in compressed form.

Note that in general it is not difficult to encode a set $\mathcal{U}$ within $\lceil \log |\mathcal{U}| \rceil$ bits. It is a matter of assigning to each element a distinct number of $\lceil \log |\mathcal{U}| \rceil$ bits. The problem with such a code is practicality: It might not be simple to convert the code back into the element unless one has a table of $|\mathcal{U}|$ cells that stores the mapping. This can be practical (and we will use it) when $\mathcal{U}$ is small and we have to encode many such elements, but otherwise we need to consider codes that are easier to handle.

Example 2.3 *We can encode the* $|\mathcal{T}_4| = 5$ *trees of* $\mathcal{T}_4$ *with codes of length* $\lceil \log(|\mathcal{T}_4|) \rceil = 3$, *for example,* {000, 001, 010, 011, 100}.

Note that in this example we use only 3 bits to encode the elements of $\mathcal{T}_4$, whereas in Example 2.2 we used $2n = 8$. We showed that $\mathcal{H}_{\text{wc}}(\mathcal{T}_n) = 2n - \Theta(\log n)$, so for large trees, using $2n$ bits comes close to optimal (or, in technical terms, using $2n$ bits is *asymptotically optimal*). For small trees of $n = 4$ nodes, the $\Theta(\log n)$ factor is still significant. In Chapters 7 and 8 we show that we can navigate the representation of Example 2.2 in constant time, using $o(n)$ bits in addition to the $2n$.

2.2 Shannon Entropy

In classical Information Theory, one assumes that there is an infinite source that emits elements $u \in \mathcal{U}$ with *probabilities* $\Pr(u)$. By using codes of varying lengths of $\ell(u)$ bits (shorter codes for the more probable elements), one can reduce the *average* length of the codes:

$$\overline{\ell} = \sum_{u \in \mathcal{U}} \Pr(u) \cdot \ell(u).$$

Then a natural question is how to assign codes that minimize the average code length. A fundamental result of Information Theory is that the minimum possible average code length for codes that can be univocally decoded is

$$\mathcal{H}(\Pr) = \sum_{u \in \mathcal{U}} \Pr(u) \cdot \log \frac{1}{\Pr(u)},$$

which is called the *Shannon entropy* of the *probability distribution* $\Pr : \mathcal{U} \to [0.0, 1.0]$. The measure $\mathcal{H}(\Pr)$ can be interpreted as how many bits *of information* are contained in each element emitted by the source. The more biased the probability distribution, the more "predictable," or the less "surprising," the output of the source is,

and the less information is carried by its elements. In particular, if one probability tends to 1.0 and all the others tend to 0.0, then $\mathcal{H}(\Pr)$ tends to 0.[1]

Example 2.4 *A service in the Sahara desert announces the weather condition every day, which is an element in* {sun, rain, snow}. *The Shannon entropy, or amount of surprise, from such a source is close to zero. If the service is in, say, New York, the source will be less predictable and thus contain more information, and its Shannon entropy will be higher.*

The formula of $\mathcal{H}(\Pr)$ also suggests that an optimal code for u should be $\log \frac{1}{\Pr(u)}$ bits long. Actually, it can be proved that no other choice of code lengths reaches the optimal average code length $\mathcal{H}(\Pr)$. This means that, as anticipated, more probable elements should receive shorter codes. Note that, no matter how we assign the codes to reduce the *average* code length, the length of the *longest* code is still at least $\lceil \mathcal{H}_{\text{wc}}(\mathcal{U}) \rceil$.

Example 2.5 *Assume that the* 5 *trees of* $\mathcal{T}_4$ *arise with probabilities* $\{0.6, 0.3, 0.05, 0.025, 0.025\}$. *Their Shannon entropy is then* $\mathcal{H} = 0.6 \cdot \log \frac{1}{0.6} + 0.3 \cdot \log \frac{1}{0.3} + 0.05 \cdot \log \frac{1}{0.05} + 2 \times 0.025 \cdot \log \frac{1}{0.025} \approx 1.445 < \log 5 = \mathcal{H}_{\text{wc}}(\mathcal{T}_4)$. *Thus we could encode the trees using, on average, less than* $\mathcal{H}_{\text{wc}}$ *bits per tree. For example, we could assign* 0 *to the first tree,* 10 *to the second,* 110 *to the third,* 1110 *to the fourth, and* 1111 *to the fifth. The average code length is then* $0.6 \cdot 1 + 0.3 \cdot 2 + 0.05 \cdot 3 + 2 \times 0.025 \cdot 4 = 1.550$ *bits. This is larger than* $\mathcal{H}$ *but smaller than* $\mathcal{H}_{\text{wc}}$. *On the other hand, our longest code length is* $4 \geq \lceil \mathcal{H}_{\text{wc}} \rceil$.

When all the probabilities are equal, $\Pr(u) = \frac{1}{|\mathcal{U}|}$, the Shannon entropy is maximized, reaching precisely $\log |\mathcal{U}|$, and the optimum is to use codes of about the same length for all the elements. Then the set is said to be *incompressible*. In this case, the Shannon entropy coincides with our measure of worst-case entropy, $\mathcal{H}_{\text{wc}}(\mathcal{U})$.

Example 2.6 *Assume that the* 5 *trees of* $\mathcal{T}_4$ *arise with probabilities* 0.2 *each. Then the Shannon entropy is* $\mathcal{H} = 5 \times 0.2 \cdot \log \frac{1}{0.2} = \log 5 = \mathcal{H}_{\text{wc}}$. *Still we could use codes of average length less than* $\lceil \log 5 \rceil = 3$, *for example,* {00, 01, 10, 110, 111}, *with average code length* $3 \times 0.2 \cdot 2 + 2 \times 0.2 \cdot 3 = 2.4$, *which is larger than* $\mathcal{H}_{\text{wc}} \approx 2.322$ *but smaller than* $3 = \lceil \mathcal{H}_{\text{wc}} \rceil$. *The longest code, however, is of length* $3 = \lceil \mathcal{H}_{\text{wc}} \rceil$.

2.3 Empirical Entropy

Consider the particular case $\mathcal{U} = \{0, 1\}$, that is, where our infinite source emits bits. Assume that bit 1 is emitted with probability p and bit 0 with probability $1 - p$. The Shannon entropy of this source, also called binary entropy, is then

$$\mathcal{H}(p) = p \log \frac{1}{p} + (1 - p) \log \frac{1}{1 - p}.$$

Figure 2.2 shows $\mathcal{H}(p)$ as a function of the probability p. As expected, $\mathcal{H}(0.0) = \mathcal{H}(1.0) = 0$, that is, the entropy (or information, or surprise) is zero when one element

[1] We take the usual analytical limit $0 \cdot \log \frac{1}{0} = \lim_{x \to 0} x \cdot \log \frac{1}{x} = 0$.

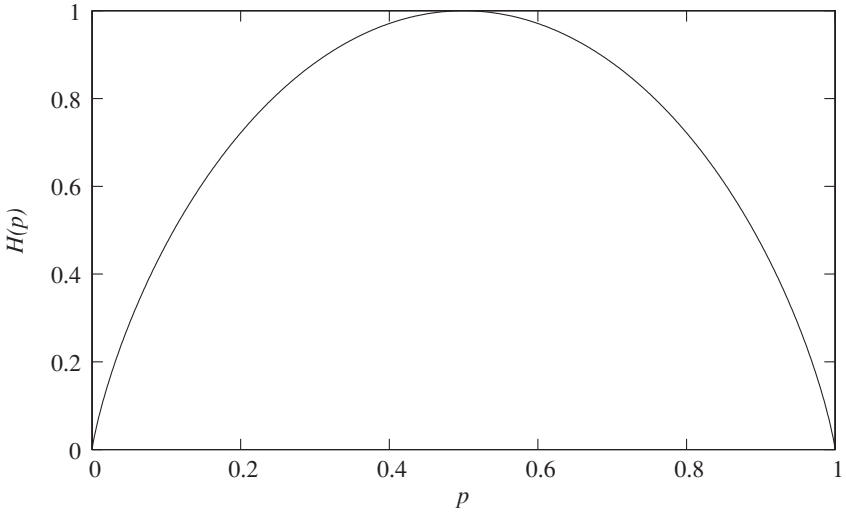


Figure 2.2. The binary entropy function $\mathcal{H}(p)$.

has probability 1.0 and the other 0.0. Also, $\mathcal{H}(0.5) = 1$, that is, the entropy reaches its maximum (in this case, 1 bit per bit) when the elements are emitted with the same probability.

The concept of Shannon entropy also applies when the elements are *sequences* of those bits emitted by the source. Assume that the source emits each bit independently of the rest (this is called a "memoryless" or "zero-order" source). If we take chunks of n bits and call them our elements, then we have $\mathcal{U} = \{0, 1\}^n$, and the Shannon entropy of the sequences is $n\mathcal{H}(p)$ (more generally, the entropy of two independent events is the sum of their entropies).

When the source emits symbols from a more general alphabet $\Sigma = [1, \sigma]$, where symbol s has probability p_s, the Shannon entropy of the source becomes

$$\mathcal{H}(\langle p_1, \ldots, p_\sigma \rangle) = \sum_{1 \le s \le \sigma} p_s \log \frac{1}{p_s}$$

bits. Again, we have $\mathcal{H} = 0$ when some p_s is 1.0 and all the rest are 0.0, and $\mathcal{H} = \log \sigma$ when $p_s = \frac{1}{\sigma}$ for all s. A sequence of n elements from Σ, belonging to $\mathcal{U} = \Sigma^n$, that is, a string of length n, has Shannon entropy $n\mathcal{H}(\langle p_1, \ldots, p_\sigma \rangle)$.

In this section we show how the Shannon entropy can be used to define an entropy notion for individual sequences without assuming they come from a certain source.

2.3.1 Bit Sequences

Assume we have a *concrete* bit sequence, $B[1, n]$, that we wish to compress somehow. Here we do not have an ideal model of a known source that emits bits, we have only B. We may have a good reason to expect that B has more 0s than 1s, however, or more 1s than 0s. Therefore, we may try to compress B using that property. That is, we will *assume* that B has been generated by a zero-order source that emits 0s and 1s. Let m be the number of 1s in B. Then it makes sense to assume that the source emits 1s with

probability $p = \frac{m}{n}$. This leads to the concept of *zero-order empirical entropy*:

$$\mathcal{H}_0(B) = \mathcal{H}\left(\frac{m}{n}\right) = \frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}.$$

The empirical entropy is thus the Shannon entropy of a source that emits 1s with the *observed* probability of the 1s in B. It can be shown that if B indeed comes from a zero-order source with probability p, then it has $m \approx p \cdot n$ 1s with very high probability.

The practical meaning of the zero-order empirical entropy is that, if a compressor attempts to compress B by using some fixed code $C(1)$ for the 1 (which can even use a fractional number of bits; see Section 2.6.4) and some fixed code $C(0)$ for the 0, then it cannot compress B to less than $n\mathcal{H}_0(B)$ bits. Otherwise we would have $m|C(1)| + (n-m)|C(0)| < n\mathcal{H}_0(B)$. Calling $p = \frac{m}{n}$, this would give $p|C(1)| + (1-p)|C(0)| < \mathcal{H}_0(B) = \mathcal{H}(p)$, which means that this code would break the lower bound of the Shannon entropy.

A Connection with Worst-Case Entropy

It is interesting that the concepts of zero-order empirical entropy and of worst-case entropy are closely related, despite their different origins. Consider the set $\mathcal{B}_{n,m}$ of all the bit sequences of length n with m 1s. Then $|\mathcal{B}_{n,m}| = \binom{n}{m}$, and its worst-case entropy is

$$\begin{aligned}\mathcal{H}_{\text{wc}}(\mathcal{B}_{n,m}) &= \log\binom{n}{m} = n\log n - m\log m - (n-m)\log(n-m) - \mathcal{O}(\log n)\\ &= n\left(\frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}\right) - \mathcal{O}(\log n)\\ &= n\mathcal{H}_0(B) - \mathcal{O}(\log n),\end{aligned}$$

where we used Stirling's approximation in the first line and expanded $n\log n = m\log n + (n-m)\log n$ in the second.

Therefore, the zero-order empirical entropy can be understood simply in terms of how many different bit sequences there are with m 1s. We saw in Section 2.1 that the worst-case entropy of bit sequences of length n is n bits. However, if $m \ll n$, the subset $\mathcal{B}_{n,m}$ of the sparse bit sequences that have only m 1s is much smaller, and thus it can be encoded with $n\mathcal{H}(\frac{m}{n})$ bits. Note that the zero-order empirical entropy is the same for any sequence in $\mathcal{B}_{n,m}$ and thus refers to a worst-case encoding of the set.

Example 2.7 *Consider $\mathcal{B}_{100,20}$, the class of bit sequences of length* 100 *with* 20 1*s. Its worst-case entropy is $\mathcal{H}_{\text{wc}}(\mathcal{B}_{100,20}) = \log\binom{100}{20} \approx 68.861$ bits. The zero-order empirical entropy of any $B \in \mathcal{B}_{100,20}$ is $\mathcal{H}_0(B) = \mathcal{H}(\frac{20}{100}) = \frac{20}{100}\log\frac{100}{20} + \frac{80}{100}\log\frac{100}{80} \approx 0.72193$, thus $n\mathcal{H}_0(B) \approx 72.193$ bits. Note that $\mathcal{H}_0(B)$ is slightly larger than $\mathcal{H}_{\text{wc}}(\mathcal{B}_{100,20})$, by an $\mathcal{O}(\log n)$ factor.*

On the other hand, consider the class of "balanced" bit sequences, which have as many 0s as 1s, $\mathcal{B}_{n,n/2}$. This set is of size $|\mathcal{B}_{n,n/2}| = \binom{n}{n/2} = \frac{2^n}{\sqrt{n}} \cdot \Theta(1)$, and therefore its worst-case entropy is $\mathcal{H}_{\text{wc}}(\mathcal{B}_{n,n/2}) = n - \Theta(\log n)$. Then this class is incompressible in terms of worst-case entropy because it is not much smaller than the whole class of

sequences of n bits. In terms of zero-order empirical entropy, it cannot be compressed to less than $n\mathcal{H}(\frac{1}{2}) = n$ bits.

Note that a balanced sequence can be perfectly compressible for other reasons. For example, the 0s might tend to appear before the 1s. The limitation above arises because we are assuming that the only source of compressibility of B is the frequency of its 1s. That is, once again, we are assuming that B is generated from a zero-order source of bits. Depending on the application, one must find out the reasons why the objects at hand might be compressible and use an according model (there are some exceptions, such as the general ordinal trees, where even the worst-case entropy is a quite good model).

Finally, sometimes we will use the following bound:

$$\begin{aligned} n\mathcal{H}_0(B) &= n\left(\frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}\right) \\ &= m\log\frac{n}{m} + (n-m)\log\left(1+\frac{m}{n-m}\right) \\ &\le m\log\frac{n}{m} + (n-m)\frac{1}{\ln 2}\,\frac{m}{n-m} = m\log\frac{n}{m} + \mathcal{O}(m), \end{aligned}$$

where in the last line we used $\log(1+x) = \frac{\ln(1+x)}{\ln 2}$ and $\ln(1+x) \le x$.

2.3.2 Sequences of Symbols

The zero-order empirical entropy of a string $S[1,n]$, where each symbol s appears n_s times in S, is also defined in terms of the Shannon entropy of its observed probabilities:

$$\mathcal{H}_0(S) = \mathcal{H}\left(\left\langle\frac{n_1}{n},\ldots,\frac{n_\sigma}{n}\right\rangle\right) = \sum_{1\le s\le\sigma}\frac{n_s}{n}\log\frac{n}{n_s}.$$

Example 2.8 *Let* $S = \texttt{abracadabra}$. *Then* $n = 11$, $n_{\texttt{a}} = 5$, $n_{\texttt{b}} = n_{\texttt{r}} = 2$, *and* $n_{\texttt{c}} = n_{\texttt{d}} = 1$. *The zero-order empirical entropy of* S *is* $\mathcal{H}_0(S) = \frac{5}{11}\cdot\log\frac{11}{5} + 2\times\frac{2}{11}\cdot\log\frac{11}{2} + 2\times\frac{1}{11}\cdot\log\frac{11}{1} \approx 2.040$. *Therefore, one could compress* S *to* $n\mathcal{H}_0(S) \approx 22.44$ *bits, less than the* $n\log\sigma = 11\cdot\log 5 \approx 25.54$ *bits of the worst-case entropy of the strings of length* $n = 11$ *and alphabet of size* $\sigma = 5$.

Once again, this explains why texts may be compressible even when their worst-case entropy (Section 2.1) is $n\log\sigma$ bits. If, for example, their symbol frequencies are far from uniform, then they can be compressed up to $n\mathcal{H}(\langle\frac{n_1}{n},\ldots,\frac{n_\sigma}{n}\rangle)$ bits, simply because there are fewer strings with those frequencies, and then fewer bits are needed to identify which string is at hand:

$$\log\binom{n}{n_1,\ldots,n_\sigma} = n\mathcal{H}\left(\left\langle\frac{n_1}{n},\ldots,\frac{n_\sigma}{n}\right\rangle\right) - \mathcal{O}(\sigma\log n).$$

A subtle point for the curious reader: The $\mathcal{O}(\sigma\log n)$ extra bits of the Shannon entropy allow us to encode *any* $S[1,n]$, obtaining $n\mathcal{H}(\langle\frac{n_1}{n},\ldots,\frac{n_\sigma}{n}\rangle)$ bits on average if the source has those frequencies. Instead, within $\log\binom{n}{n_1,\ldots,n_\sigma}$ bits we can encode *only* the sequences that have exactly those frequencies. The reason why both entropies

are so close is that long sequences emitted with a probabilistic distribution concentrate sharply around the mean.

2.4 High-Order Entropy

While the mere frequency of the symbols $\Sigma = [1, \sigma]$ can be an important source of compressibility for a sequence S, it is not the only one. For example, if one tokenizes the words in an English text, zero-order compression typically reduces the text to about 25% of its size. However, good compressors can reach less than 15% by exploiting the so-called *high-order entropy* or *kth-order entropy*. This is a measure of the information carried by a symbol given that we know the k symbols that precede it in S. Indeed, one can better guess what the next word is in an English text if one knows some words preceding it. The lower the surprise, the lower the entropy.

In terms of the Shannon entropy, sources "with memory" remember the last k symbols emitted, and the probability of each emitted symbol may depend on this memory (this is a particular case of a Markov chain). The probability of s given that $s_1 \dots s_k$ was just emitted is $\Pr(s|s_1 \dots s_k)$. Then, when $s_1 \dots s_k$ has just been emitted, the entropy of the next symbol is

$$\mathcal{H}([s_1 \dots s_k]) = \mathcal{H}(\langle \Pr(1|s_1 \dots s_k), \dots, \Pr(\sigma|s_1 \dots s_k) \rangle),$$

and the Shannon entropy of the probabilistic distribution is

$$\mathcal{H}(\Pr) = \sum_{s_1 \dots s_k} \Pr(s_1 \dots s_k)\, \mathcal{H}([s_1 \dots s_k]).$$

Here $\Pr(s_1 \dots s_k)$ is the probability of emitting $s_1 \dots s_k$, which can be computed from the conditional probabilities.

Example 2.9 *Consider again the source of weather conditions of New York, in Example 2.4. This is for sure not a memoryless source. Let us consider a first-order model for it, where the weather of a day depends (only) on that of the previous day. Assume*

$$\begin{array}{lll} \Pr(\texttt{sun}|\texttt{sun}) = 0.5, & \Pr(\texttt{rain}|\texttt{sun}) = 0.25, & \Pr(\texttt{snow}|\texttt{sun}) = 0.25, \\ \Pr(\texttt{sun}|\texttt{rain}) = 0.3, & \Pr(\texttt{rain}|\texttt{rain}) = 0.6, & \Pr(\texttt{snow}|\texttt{rain}) = 0.1, \\ \Pr(\texttt{sun}|\texttt{snow}) = 0.2, & \Pr(\texttt{rain}|\texttt{snow}) = 0.2, & \Pr(\texttt{snow}|\texttt{snow}) = 0.6. \end{array}$$

For example, since

$$\Pr(\texttt{sun}) = \Pr(\texttt{sun}|\texttt{sun})\Pr(\texttt{sun}) + \Pr(\texttt{sun}|\texttt{rain})\Pr(\texttt{rain}) + \Pr(\texttt{sun}|\texttt{snow})\Pr(\texttt{snow}),$$

and similarly for $\Pr(\texttt{rain})$ *and* $\Pr(\texttt{snow})$*, we can obtain from the system of three equations the global probabilities* $\Pr(\texttt{sun}) \approx 0.34$, $\Pr(\texttt{rain}) \approx 0.36$, $\Pr(\texttt{snow}) \approx 0.30$*. The Shannon entropy of a source with these global probabilities is* $\mathcal{H}(\langle 0.34, 0.36, 0.30 \rangle) \approx 1.581$ *bits. However, the Shannon entropy of the first-order model is lower,* $\mathcal{H}(\Pr) = \Pr(\texttt{sun}) \cdot \mathcal{H}(\langle 0.5, 0.25, 0.25 \rangle) + \Pr(\texttt{rain}) \cdot \mathcal{H}(\langle 0.3, 0.6, 0.1 \rangle) + \Pr(\texttt{snow}) \cdot \mathcal{H}(\langle 0.2, 0.2, 0.6 \rangle) \approx 1.387$.

When considering concrete sequences $S[1, n]$, we can compute the *empirical kth-order entropy* of S by considering the frequencies of the symbols depending on the

preceding k symbols:

$$\mathcal{H}_k(S) = \sum_{C=s_1\ldots s_k} \frac{|S_C|}{n} \cdot \mathcal{H}_0(S_C),$$

where S_C is a string formed by collecting the symbol that follows each occurrence of the context $C = s_1 \ldots s_k$ in S. A moment of thought shows that $\mathcal{H}_k(S)$ is equal to $\mathcal{H}(\Pr)$ if we replace $\Pr(s|s_1 \ldots s_k)$ by the relative number of times s appears after the context $s_1 \ldots s_k$ (or, which is the same, in $S_{s_1\ldots s_k}$), and $\Pr(s_1 \ldots s_k)$ by the relative frequency of the string $s_1 \ldots s_k$ in S.

Example 2.10 *Consider again the string* $S = \texttt{abracadabra}$ *of Example 2.8, and* $k = 1$. *Then* $S_{\texttt{a}} = \texttt{bcdb\$}$ *(we have added an artificial terminator* \$ *to* S *for technical convenience),* $S_{\texttt{b}} = \texttt{rr}$, $S_{\texttt{c}} = \texttt{a}$, $S_{\texttt{d}} = \texttt{a}$, *and* $S_{\texttt{r}} = \texttt{aa}$. *Then* $\mathcal{H}_0(S_{\texttt{a}}) \approx 1.922$ *and* $\mathcal{H}_0(S_{\texttt{b}}) = \mathcal{H}_0(S_{\texttt{c}}) = \mathcal{H}_0(S_{\texttt{d}}) = \mathcal{H}_0(S_{\texttt{r}}) = 0$. *The first-order empirical entropy of* S *is thus* $\mathcal{H}_1(S) = \frac{5}{11}\mathcal{H}_0(S_{\texttt{a}}) + \frac{2}{11}\mathcal{H}_0(S_{\texttt{b}}) + \frac{1}{11}\mathcal{H}_0(S_{\texttt{c}}) + \frac{1}{11}\mathcal{H}_0(S_{\texttt{d}}) + \frac{2}{11}\mathcal{H}_0(S_{\texttt{r}}) \approx 0.874$, *much less than its zero-order entropy computed in Example 2.8.*

Extending the concept of zero-order empirical entropy, $n\mathcal{H}_k(S)$ is a lower bound to the number of bits that any encoding of S can achieve if the code of each symbol can be a function of itself and the k symbols preceding it in S. As before, any compressor breaking that barrier would also be able to compress symbols coming from the corresponding kth-order source into less than its Shannon entropy.

As expected, it holds $\log\sigma \geq \mathcal{H}_0(S) \geq \mathcal{H}_1(S) \geq \ldots \geq \mathcal{H}_{k-1}(S) \geq \mathcal{H}_k(S)$ for any k. Note that, for sufficiently large k values (at most for $k = n - 1$, and usually much sooner), it holds $\mathcal{H}_k(S) = 0$ because all the contexts of length k appear only once in S. At this point, the model becomes useless as a lower bound for compressors. Even before reaching $\mathcal{H}_k(S) = 0$, compressors cannot achieve $n\mathcal{H}_k(S)$ bits in practice for very high k values, because they must store the set of σ^{k+1} probabilities, or, equivalently, the set of σ^{k+1} codes,[2] so that the decompressor can reconstruct S. In theory, it is common to assume that S can be compressed up to $n\mathcal{H}_k(S) + o(n)$ bits for any $k + 1 \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$, because in this case one can store σ^{k+1} numbers in $[1, n]$ (such as frequencies) within $\sigma^{k+1} \log n \leq n^\alpha \log n = o(n)$ bits.

2.5 Coding

The entropy tells us the minimum average code length we can achieve, and even suggests giving each symbol s a code with length $\log \frac{1}{\Pr(s)}$ to reach that optimum. To obtain kth-order entropy, we must use a different set of codes for each different previous context. However, in either case the entropy measure does not tell how to build a suitable set of codes.

An *encoding* is an injective function $\mathcal{C} : \Sigma \to \{0, 1\}^*$ that assigns a distinct sequence of bits $\mathcal{C}(s)$ to each symbol $s \in \Sigma$. Encodings are also simply called *codes*, overloading the meaning of the individual code assigned to a symbol. Since we will

[2] Similarly, adaptive compressors must record σ^{k+1} escape symbols somewhere along the compressed file.

encode a sequence of symbols of Σ by means of concatenating their codes, even injective functions may be unsuitable, because they may lead to concatenations that cannot be unambiguously decoded.

Example 2.11 *Assume our set of codes is* $\mathcal{C}(s_1) = 0$, $\mathcal{C}(s_2) = 1$, $\mathcal{C}(s_3) = 00$. *Then the concatenation* $\mathcal{C}(s_1) \cdot \mathcal{C}(s_3) = 000$, *looks exactly the same as* $\mathcal{C}(s_3) \cdot \mathcal{C}(s_1) = 000$, *and thus we cannot know which one of the two sequences was encoded.*

An encoding is said to be *unambiguous* if, given a concatenation of bits, there is no ambiguity regarding the original sequence that was encoded. Said another way, if we define a new code $\mathcal{C}'$ on sequences of symbols of Σ, $\mathcal{C}' : \Sigma^* \to \{0, 1\}^*$, so that $\mathcal{C}'(s_1 \dots s_k) = \mathcal{C}(s_1) \dots \mathcal{C}(s_k)$, then code $\mathcal{C}$ is unambiguous iff $\mathcal{C}'$ is an injective function.

Example 2.12 *Assume our set of codes is* $\mathcal{C}(s_1) = 1$, $\mathcal{C}(s_2) = 10$, $\mathcal{C}(s_3) = 00$. *This code is unambiguous: if a bit sequence starts with* 0, *then it must start with* $\mathcal{C}(s_3) = 00$ *and we can continue. Otherwise it starts with a* 1, *and we must see how many* 0*s are there up to the next* 1 *or the end of the sequence. If there are* $2z$ 0*s (for some* $z \geq 0$*), then it was* $\mathcal{C}(s_1) = 1$ *followed by* z *occurrences of* $\mathcal{C}(s_3) = 00$. *If there are* $2z + 1$ 0*s, then it was* $\mathcal{C}(s_2) = 10$ *followed by* z *occurrences of* $\mathcal{C}(s_3) = 00$.

The example shows a shortcoming of unambiguous codes: It might not be possible to decode the next symbol s in constant time or even in time proportional to $|\mathcal{C}(s)|$. It is more useful that an encoding also be *instantaneous*, that is, that we have sufficient information to determine s as soon as we read the last bit of $\mathcal{C}(s)$. Apart from the obvious efficiency advantage, instantaneous codes can be easily embedded in other streams, because they do not depend on their context to be decoded.

It can be shown that instantaneous codes are precisely the so-called *prefix-free codes* or just *prefix codes*. A code $\mathcal{C}$ is a prefix code if no code $\mathcal{C}(s)$ is a prefix of any other code $\mathcal{C}(s')$. Clearly, with a prefix code there cannot be ambiguity with respect to whether we are seeing the final bit of a code or rather the middle of a longer code. Somewhat surprisingly, unambiguous codes that are not prefix codes are useless: there is always a prefix code that is at least as good as any given unambiguous code.

Example 2.13 *We can find a prefix code where code lengths are the same as those assigned in Example 2.12:* $\mathcal{C}(s_1) = 1$, $\mathcal{C}(s_2) = 01$, $\mathcal{C}(s_3) = 00$. *Now we can determine the symbol that is encoded next in a bit sequence as soon as we read the last bit of its code.*

The reason is the so-called *Kraft-McMillan inequality*: Any unambiguous code $\mathcal{C}$ (and thus any prefix code $\mathcal{C}$), satisfies

$$\sum_{s \in \Sigma} 2^{-\ell(s)} \leq 1,$$

where $\ell(s) = |\mathcal{C}(s)|$, and on the other hand, a prefix code (and thus an unambiguous code) with lengths $\ell(s)$ always exists if the above inequality holds. Hence, given an unambiguous code $\mathcal{C}$, it satisfies the inequality, and thus there is a prefix code with the same code lengths.

Algorithm 2.1: Building a prefix code over $\Sigma = [1, \sigma]$, given the desired lengths. Assumes for simplicity that the codes fit in a computer word.

Input : $S[1, \sigma]$, with $S[i].s$ a distinct symbol and $S[i].\ell$ its desired code length.
Output: Array S (reordered) with a new field computed, $S[i].code$, an integer whose $S[i].\ell$ lowest bits are a prefix code for $S[i].s$ (reading from most to least significant bit).

```
1 Sort S[1, σ] by increasing S[i].ℓ values
2 S[1].code ← 0
3 for i ← 2 to σ do
4 |  S[i].code ← (S[i − 1].code + 1) · 2^(S[i].ℓ − S[i−1].ℓ)
```

This leads to a simple way of assigning reasonably good codes, called the *Shannon-Fano codes*. Given that the optimal code length is $\log \frac{1}{\Pr(s)}$, assign code length $\ell(s) = \lceil \log \frac{1}{\Pr(s)} \rceil$. Then it holds

$$\sum_{s\in\Sigma} 2^{-\ell(s)} = \sum_{s\in\Sigma} 2^{-\left\lceil \log \frac{1}{\Pr(s)} \right\rceil} \le \sum_{s\in\Sigma} 2^{-\log \frac{1}{\Pr(s)}} = \sum_{s\in\Sigma} \Pr(s) = 1,$$

and thus, by the Kraft-McMillan inequality, there is a prefix code with those lengths. Actually, it is not difficult to find: Process the lengths from shortest to longest, giving to each code the next available binary number of the appropriate length, where "available" means we have not yet used the number or a prefix of it. More precisely, if the sorted lengths are $\ell_1, \ldots, \ell_\sigma$, then the first code is 0^{ℓ_1} (ℓ_1 0s), and the code for s_{i+1} is obtained by summing 1 to the ℓ_i-bit number assigned to s_i, and then appending $\ell_{i+1} - \ell_i$ 0s to it. The Kraft-McMillan inequality guarantees that we will always find a free number of the desired length for the next code. Algorithm 2.1 gives the pseudocode, returning the assigned codes as integer values (for example, code 000110 is stored as the number 6, which cannot lead to confusion because we also know the code length).

A Shannon-Fano code obtains an average code length that is less than 1 bit over the Shannon entropy. This is because the average code length is

$$\overline{\ell} = \sum_{s\in\Sigma} \Pr(s) \cdot \left\lceil \log \frac{1}{\Pr(s)} \right\rceil < \sum_{s\in\Sigma} \Pr(s) \cdot \left(\log \frac{1}{\Pr(s)} + 1 \right) = \mathcal{H}(\Pr) + 1.$$

Example 2.14 *Consider the set $\mathcal{T}_4$ with the probabilities given in Example 2.5, $\{0.6, 0.3, 0.05, 0.025, 0.025\}$, which had Shannon entropy $\mathcal{H} \approx 1.445$. The Shannon-Fano code gives lengths $\lceil \log \frac{1}{0.6} \rceil = 1$ for the first tree, $\lceil \log \frac{1}{0.3} \rceil = 2$ for the second, $\lceil \log \frac{1}{0.05} \rceil = 5$ for the third, and $\lceil \log \frac{1}{0.025} \rceil = 6$ for the last two. Then we start assigning code 0 to the first tree. For the second, the first available code of length 2 is 10, obtained by summing 1 to 0 and appending $2 - 1 = 1$ 0s to it. For the third, we obtain 11000 by adding 1 to 10, getting 11, and then appending $5 - 2 = 3$ 0s. For the fourth and fifth we have codes 110010 and 110011. The average length of this code is $0.6 \cdot 1 + 0.3 \cdot 2 + 0.05 \cdot 5 + 2 \times 0.025 \cdot 6 = 1.75$ bits. This is less than $\mathcal{H} + 1$, but we found a better code in Example 2.5.*

The example shows that, although Shannon-Fano codes always spend less than 1 bit over the entropy, they are not necessarily optimal. The next section shows how to build optimal codes.

2.6 Huffman Codes

Huffman devised an algorithm that, given a probability distribution $\Pr : \Sigma \to [0.0, 1.0]$, obtains a prefix code of minimum average length. Thus, Huffman is optimal among the codes that assign an integral number of bits to each symbol. In particular, it is no worse than Shannon-Fano codes, and so the number of bits it outputs is between $n\mathcal{H}(\Pr)$ and $n(\mathcal{H}(\Pr) + 1)$, wasting less than 1 bit per symbol. Similarly, if Pr are the relative symbol frequencies in a string $S[1, n]$, then Huffman codes compress S to less than $n(\mathcal{H}_0(S) + 1)$ bits.

2.6.1 Construction

The Huffman algorithm progressively converts a set of $|\Sigma|$ nodes into a binary tree. At every moment, it maintains a set of binary trees. The leaves of the trees correspond to the symbols $s \in \Sigma$, and each tree has a *weight* equal to the sum of the probabilities of the symbols at its leaves. The algorithm starts with $|\Sigma|$ trees, each being a leaf node corresponding to a distinct symbol $s \in \Sigma$, with weight $\Pr(s)$. Then $|\Sigma| - 1$ tree merging steps are carried out, finishing with a single tree of weight 1.0 and with all the $|\Sigma|$ leaves.

The merging step always chooses two trees T_1 and T_2 of minimum weight and joins them by creating a new root, whose left and right children are T_1 and T_2, and whose weight is the sum of the weights of T_1 and T_2.

The single tree resulting from the algorithm is called the *Huffman tree*. If we interpret going left as the bit $\mathsf{0}$ and going right as the bit $\mathsf{1}$, then the path from the root to the leaf of each $s \in \Sigma$ spells out its code $\mathcal{C}(s)$. The Huffman tree minimizes the average code length, $\sum_{s\in\Sigma} \Pr(s) \cdot \ell(s)$.

Example 2.15 *Figure 2.3 illustrates the Huffman algorithm on the probabilities of Example 2.8:* $\Pr(\mathtt{a}) = \frac{5}{11}$, $\Pr(\mathtt{b}) = \Pr(\mathtt{r}) = \frac{2}{11}$, *and* $\Pr(\mathtt{c}) = \Pr(\mathtt{d}) = \frac{1}{11}$. *The final Huffman tree on the left assigns the codes* $\mathcal{C}(\mathtt{a}) = \mathsf{0}$, $\mathcal{C}(\mathtt{b}) = \mathsf{110}$, $\mathcal{C}(\mathtt{r}) = \mathsf{111}$, $\mathcal{C}(\mathtt{c}) = \mathsf{100}$, *and* $\mathcal{C}(\mathtt{d}) = \mathsf{101}$. *The average code length is* $\frac{5}{11} \cdot 1 + 2 \times \frac{2}{11} \cdot 3 + 2 \times \frac{1}{11} \cdot 3 \approx 2.091$, *very close to* $\mathcal{H}(\Pr) = \mathcal{H}_0(S) \approx 2.040$. *A Shannon-Fano code obtains a higher average code length,* ≈ 2.727 *bits.*

On the bottom right we show another valid Huffman tree that is obtained by breaking ties in another way (we leave as an exercise to the reader to determine the corresponding merging order). Its average code length is also $\frac{5}{11} \cdot 1 + \frac{2}{11} \cdot 2 + \frac{2}{11} \cdot 3 + 2 \times \frac{1}{11} \cdot 4 \approx 2.091$.

The Huffman algorithm runs in time $\mathcal{O}(|\Sigma| \log |\Sigma|)$. This can be obtained, for example, by maintaining the trees in a priority queue to extract the next two minimum weights. Algorithm 2.2 shows another way. We first sort the weights into a linked list L and then repeatedly remove the first (i.e., lightest) two trees of the list in order to

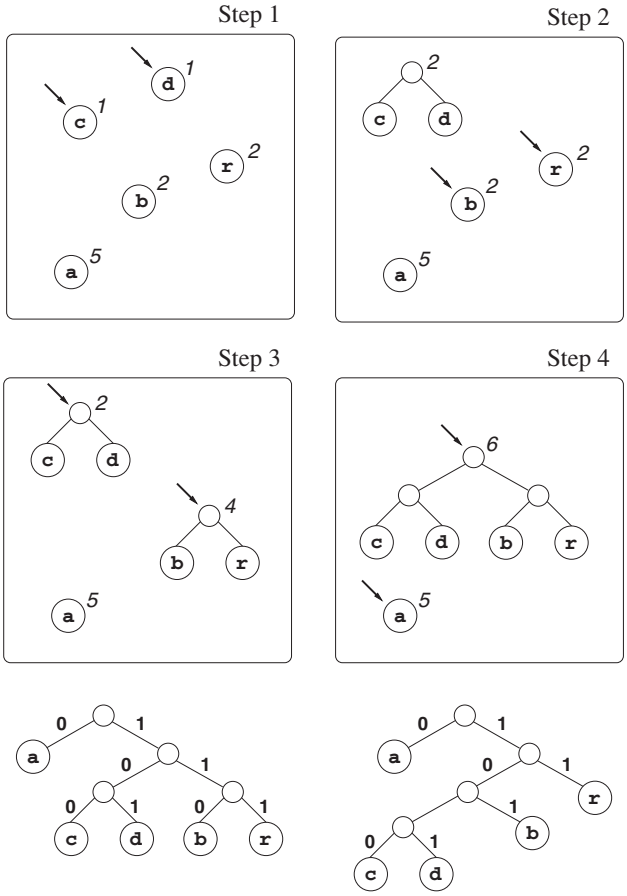


Figure 2.3. The Huffman algorithm on 5 symbols. Instead of probabilities we show their frequencies, in italics and close to the tree roots. The arrows show the trees chosen at each step. The final Huffman tree on the left shows the codes assigned. The one on the right is another valid Huffman tree obtained by choosing other trees to merge when there are ties.

merge them into a new tree. The insertion point I for this new tree always advances in L toward the higher weights (because the new merged tree is never lighter than the previous). Thus, although each search for the new insertion point I scans L linearly (lines 8–9), only one traversal of L is carried out along the whole construction. The total cost after sorting then amortizes to $\mathcal{O}(|\Sigma|)$, because L has fewer than $2|\Sigma|$ nodes.

The sorting by frequencies can be done in time $\mathcal{O}(|\Sigma| \log |\Sigma|)$ as usual, or if the weights are frequencies in a string $S[1, n]$, one can use RadixSort to obtain $\mathcal{O}(|\Sigma| + n)$ time. This can be useful when Σ is large compared to n.

2.6.2 Encoding and Decoding

Once the Huffman tree is built, we simply traverse it recursively to obtain the code for each symbol, and store the codes in an array indexed by $s \in \Sigma$. Encoding is then easily done symbol by symbol.

Algorithm 2.2: Building a Huffman tree over the alphabet $\Sigma = [1, \sigma]$. The list L can be allocated as an array $L[1, 2\sigma - 1]$.

```
1  Proc HuffmanTree(S)
       Input  : S[1, σ], with S[i].s a distinct symbol and S[i].f its frequency (or
                weight).
       Output: Huffman tree with internal node fields l (left) and r (right) and leaf
                field s (symbol).
2      Sort S[1, σ] by increasing S[i].f values
3      Create list L linked by field L.next, with its ith node
4          L.s ← S[i].s, L.f ← S[i].f, L.l ← null, L.r ← null
5      I ← L (the first list node)
6      while L.next ≠ null do
7          Create list node N.l ← L, N.r ← L.next, N.f ← N.l.f + N.r.f
8          while I.next ≠ null and I.next.f ≤ N.f do
9              I ← I.next
10         N.next ← I.next
11         I.next ← N
12         L ← L.next.next
13     return L (regarded as a tree; fields next are to be ignored)
```

Although the longest Huffman code can be of length $|\Sigma| - 1$, it turns out that its length is also limited by $\lfloor \log_\phi \frac{1}{p_{\min}} \rfloor$, where $p_{\min}$ is the minimum probability in Pr and $\phi = (1 + \sqrt{5})/2 \approx 1.618$ is the golden ratio. In particular, if the probabilities are obtained from the observed frequencies in a sequence of n symbols, then $p_{\min} \geq \frac{1}{n}$, and the maximum possible Huffman code length is $\log_\phi n$. Thus, a 32-bit word can for sure hold the code of any file of size up to $n = \phi^{32} > 2^{22}$ (4 megabytes, if we assume one byte per symbol); this rises to 16 terabytes with 64-bit words. Thus, a few computer words suffice in all the conceivable cases to manipulate Huffman codes and write them to the output in constant time (Chapter 3 offers more details on handling the bits inside computer words). Thus, encoding takes $\mathcal{O}(n)$ time. This is also true in theory, as in the RAM model of computation $\mathcal{O}(\log n)$ bits can be manipulated in $\mathcal{O}(1)$ time.

For decoding, we can use the Huffman tree. We read the successive bits of the stream and move from the root to a leaf according to the bits read. When we arrive at a leaf, we output its corresponding symbol and return to the tree root. Therefore, the total decoding time is proportional to the bit length of the compressed sequence, $\mathcal{O}(n(\mathcal{H}(\Pr) + 1))$. Given that the codes are of length $\mathcal{O}(\log n)$, it also holds that any symbol is decoded in $\mathcal{O}(\log n)$ time. We see next how to speed this up.

2.6.3 Canonical Huffman Codes

Example 2.15 shows that there are several valid (and optimal) Huffman trees: there may not only be ties in the weights when choosing the two trees to merge, but we can also exchange the left and right children of any node.

From all the possible Huffman trees, the so-called *canonical Huffman tree* is usually preferred because it enables more efficient decoding. In this tree, the leaf depths are nondecreasing when read left to right. The Huffman tree we built on the bottom left of Figure 2.3 happens to be canonical, whereas that on the bottom right is not.[3]

Building a canonical Huffman tree is simple. We use Algorithm 2.2 just to determine the Huffman code lengths, $\ell(s)$ (that is, the depth of the leaf labeled s), but ignore the actual codes produced by the algorithm. We then assign codes, from shortest to longest, choosing the next available binary number of $\ell(s)$ bits, as done in Algorithm 2.1 for the Shannon-Fano codes.

Canonical Huffman trees also allow for a compact representation using $|\Sigma| \log |\Sigma| + \mathcal{O}(\log^2 n)$ bits, using three arrays:

1. $L[1, |\Sigma|]$ stores the symbols of Σ in left-to-right leaf order.
2. $F[1, h]$, where $h = \mathcal{O}(\log n)$ is the Huffman tree height, stores $F[\ell] = i$ iff $L[i]$ is the first symbol whose code is of length ℓ. If there are no symbols of length ℓ, then $F[\ell] = F[\ell + 1]$.
3. $C[1, h]$ stores in $C[\ell]$ the Huffman code of $L[F[\ell]]$, that is, the first code of length ℓ. The code is stored in the ℓ lowest bits of the number $C[\ell]$. If there are no codes of length ℓ, then $C[\ell] = C[\ell + 1]/2$ (that is, the first code of length $\ell + 1$ without its last 0; this value will not be used for encoding, but instead to drive a binary search).

Algorithm 2.3 shows how to build this representation, by generating the Huffman tree shape with Algorithm 2.2, then collecting the symbols and depths from the Huffman tree leaves in array S, then using Algorithm 2.1 to generate the codes, and finally building the arrays L, F, and C.

We show how this structure can be used for efficient decoding. From the incoming bit sequence to decode, we take the first h bits and place them at the lowest h bits of a number N. Now we have to find the ℓ such that $C[\ell] \cdot 2^{h-\ell} \le N < C[\ell + 1] \cdot 2^{h-\ell-1}$, which is found by binary search in time $\mathcal{O}(\log h) = \mathcal{O}(\log \log n)$. We know that the next code is of length ℓ, so we discard the $h - \ell$ lowest bits of N by doing $N \leftarrow \lfloor N/2^{h-\ell} \rfloor$, and output the symbol $L[F[\ell] + N - C[\ell]]$. We then advance in the input stream by ℓ bits and are ready for the next code. Algorithm 2.4 gives the pseudocode.

Example 2.16 *Consider the canonical code* $\mathcal{C}(\mathtt{a}) = \mathsf{00}$, $\mathcal{C}(\mathtt{b}) = \mathsf{010}$, $\mathcal{C}(\mathtt{c}) = \mathsf{011}$, $\mathcal{C}(\mathtt{d}) = \mathsf{100}$, $\mathcal{C}(\mathtt{e}) = \mathsf{10100}$, $\mathcal{C}(\mathtt{f}) = \mathsf{10101}, \ldots, \mathcal{C}(\mathtt{p}) = \mathsf{11111}$. *Then we have* $L[1, 16] = \langle \mathtt{a}, \ldots, \mathtt{p} \rangle$, $h = 5$, $F[1, 5] = \langle 1, 1, 2, 5, 5 \rangle$, *and* $C[1, 5] = \langle \mathsf{0}, \mathsf{00}, \mathsf{010}, \mathsf{1010}, \mathsf{10100} \rangle = \langle 0, 0, 2, 10, 20 \rangle$. *This is illustrated in Figure 2.4.*

Now assume we have to decode $\mathsf{1000010101}$. *We read the first* $h = 5$ *bits,* $N = \mathsf{10000} = 16$. *It is between* $C[3] \cdot 2^{5-3} = 2 \cdot 2^2 = 8$ *and* $C[4] \cdot 2^{5-4} - 1 = 10 \cdot 2^1 - 1 = 19$, *therefore the length of the first code is* 3. *Thus we set* $N \leftarrow \lfloor 16/2^{5-3} \rfloor = 4$, *and the code is* $L[F[3] + 4 - C[3]] = L[4] = \mathtt{d}$. *We advance* $\ell = 3$ *positions in the input.*

[3] Some definitions assume that the consecutive codes of the same length must correspond to increasing symbols of Σ, but we do not require this.

Algorithm 2.3: Building a Canonical Huffman code representation.

Input : $S[1, \sigma]$, with $S[i].s$ a distinct symbol and $S[i].f$ its frequency (or weight).
Output: Arrays L, F, and C representing a Canonical Huffman code for S. Uses fields ℓ and *code* in the cells of S.

```
1  T ← HuffmanTree(S) (Algorithm 2.2)
2  computeLengths(S, T, 0, 1)
3  Run Algorithm 2.1 on S (it also sorts S by increasing ℓ values)
4  for i ← 1 to σ do L[i] ← S[i].s
5  h ← S[σ].ℓ
6  for l ← 1 to h do F[l] ← 0
7  for i ← σ downto 1 do
8      F[S[i].ℓ] ← i
9      C[S[i].ℓ] ← S[i].code
10 for l ← h − 1 downto 1 do
11     if F[l] = 0 then
12         F[l] ← F[l + 1]
13         C[l] ← C[l + 1]/2

14 Proc computeLengths(S, T, d, i)
       Input : Huffman subtree rooted at node T, of depth d, table S and position i.
       Output: Records the symbols S[i].s and lengths S[i].ℓ, consecutively starting
               from i, for the leaves below T. Returns the next free position i.
15     if T.l = null then (T is a leaf)
16         S[i].s ← T.s; S[i].ℓ ← d
17         i ← i + 1
18     else
19         i ← computeLengths(S, T.l, d + 1, i)
20         i ← computeLengths(S, T.r, d + 1, i)
21     return i
```

Algorithm 2.4: Reading a symbol with a Canonical Huffman code.

Input : $L[1, \sigma]$, $F[1, h]$, and $C[1, h]$ representing a Canonical Huffman code.
Output: The next symbol decoded from a bit stream.

```
1 N ← next h bits from the stream
2 Find ℓ such that C[ℓ] · 2^(h−ℓ) ≤ N < C[ℓ + 1] · 2^(h−ℓ−1) by binary search
3 N ← ⌊N/2^(h−ℓ)⌋
4 Advance ℓ bits in the input stream
5 return L[F[ℓ] + N − C[ℓ]]
```

i	$L[i]$	$code$	len	$code \cdot 2^{h-len}$	len	$F[len]$	$C[len]$	$C[len] \cdot 2^{h-len}$
1	a	00	2	00000 = 0	1	1	0	0
2	b	010	3	01000 = 8	2	1	00	0
3	c	011	3	01100 = 12	3	2	010	8
4	d	100	3	10000 = 16	4	5	1010	20
5	e	10100	5	10100 = 20	5	5	10100	20
6	f	10101	5	10101 = 21				
...	...	...	...	... ...				
16	p	11111	5	11111 = 31				

Figure 2.4. The representation of the Canonical Huffman code of Example 2.16. We show the code on the left and its representation on the right. Under column $code \cdot 2^{h-len}$ we show in gray how the codes are completed up to length h. On the right, under column $C[len]$, we show in gray the codes that are extrapolated because no code of that length exists. The last column shows the data where the binary search for decoding is carried out.

Now we read the next $h = 5$ bits, $N = 00101 = 5$. It is between $C[2] \cdot 2^{5-2} = 0$ and $C[3] \cdot 2^{5-3} - 1 = 7$, so the length of the next code is $\ell = 2$. We set $N \leftarrow \lfloor 5/2^{5-2} \rfloor = 0$, and the code is $L[F[2] + 0 - C[2]] = L[1] =$ a. We advance $\ell = 2$ positions in the input.

Finally, we read the next $h = 5$ bits, $N = 10101 = 21$. It is $\geq C[5] \cdot 2^0 = 20$, thus $\ell = 5$ and N does not change. The code is $L[F[5] + 21 - C[5]] = L[6] =$ f. We advance $\ell = 5$ positions and the input is consumed.

Though not evident in this example, the algorithm may request up to $h - 1$ nonexistent bits from the end of the stream. Those can be filled arbitrarily.

Thus, using canonical Huffman codes, we can decompress a sequence of n encoded symbols in time $\mathcal{O}(n \log \log n)$. This is usually faster than a bitwise decoding, unless the alphabet is small or the codes are very short on average.

2.6.4 Better than Huffman

As said, Huffman codes are optimal, but only if we assign an integral number of bits to each code. It is possible to do better than Huffman if one assigns a *fractional* number of bits per code. This is not as weird as it sounds: consider encoding *trits*, symbols in $\{0, 1, 2\}$. We can encode 3 trits (which have $3^3 = 27$ possible values) in 5 bits (which have 32 combinations), thus using $\frac{5}{3} \approx 1.667 > \log 3 \approx 1.585$ bits per trit. The trick is to encode several symbols together. A more principled approach is *arithmetic coding*, which uses less than 2 extra bits *for the whole sequence*, that is, it encodes the sequence in less than $n\mathcal{H}(\Pr) + 2$ bits. For compact data structures, however, arithmetic coding is less convenient, because it is not possible to access a symbol of the sequence without decoding from the beginning. In general, we will use Huffman as our gold standard, although in some cases we will use the trick of encoding several symbols together.

2.7 Variable-Length Codes for Integers

Huffman codes are the best possible among those giving the same code to the same symbol. Sometimes, however, they can be inconvenient because of the size of Σ: Even

with a canonical code, we have spent $|\Sigma| \log |\Sigma|$ bits to store L, which in some applications can be large compared to the $n(\mathcal{H}(\mathrm{Pr}) + 1)$ bits of the compressed data. For some particular cases, we can design *fixed* codes, which do not depend on the sequence to compress but work well on the typical sequences in which we are interested.

A good example is the need to compress a sequence of natural numbers when usually most of them are small. In this case, we can choose a fixed code that favors small numbers. We next show some of the most popular ones. Note that all these codes are prefix codes. From now on we assume we want to encode a natural number $x > 0$ and call $|x|$ its length in bits. If we want to encode the 0 as well (or up to some negative value), we may shift the values to encode.

Unary Codes

The unary code is convenient when x is extremely small:

$$u(x) = \mathsf{0}^{x-1}\,.\,\mathsf{1},$$

where $\mathsf{0}^{x-1}$ means $x - 1$ bits $\mathsf{0}$. The unary code of x uses $|u(x)| = x$ bits. For example, $u(1) = \mathsf{1}$, $u(2) = \mathsf{01}$, $u(3) = \mathsf{001}$, $u(4) = \mathsf{0001}$, and so on. It is also customary to use the reverse bits, $\mathsf{1}^{x-1}\,.\,\mathsf{0}$.

Gamma (γ) Codes

This code is convenient when x is small. The γ-code of x is

$$\gamma(x) = \mathsf{0}^{|x|-1}\,.\,[x]_{|x|} = u(|x|)\,.\,[x]_{|x|-1},$$

where $[x]_\ell$ stands for the ℓ least significant bits of x. Gamma codes can also be understood in terms of unary codes: We encode in unary the length $|x|$ of x and then encode the number x (without its highest bit) in binary.

For example, $\gamma(1) = \gamma(\mathsf{1}) = \mathsf{1}$, $\gamma(2) = \gamma(\mathsf{10}) = \mathsf{010}$, $\gamma(3) = \gamma(\mathsf{11}) = \mathsf{011}$, $\gamma(4) = \gamma(\mathsf{100}) = \mathsf{00100}$, and so on. It holds

$$|\gamma(x)| = 2|x| - 1 = 2\lfloor \log x \rfloor + 1 = \mathcal{O}(\log x),$$

so gamma codes become shorter than unary codes for $x \geq 6$.

To decode $\gamma(x)$, we read z $\mathsf{0}$s until finding a $\mathsf{1}$, and then read that $\mathsf{1}$ and the following z bits to form the value of x. This can be done in $\mathcal{O}(z) = \mathcal{O}(|x|) = \mathcal{O}(\log x)$ time, which can be good enough because we choose to use γ-codes when most x values are small. Still, we can do better. If we assume that we are encoding numbers x that fit in the computer word, then the first part of $\gamma(x)$, $u(|x|)$, also fits in a computer word, and thus the problem of finding z reduces to finding the highest $\mathsf{1}$ in a computer word. This is directly supported in many architectures (more details are given near the end of Section 4.5.2). If it is not, we can precompute a table that, for every nonzero chunk of b bits, tells where the highest $\mathsf{1}$ is. Then we need at most $\lceil |x|/b \rceil$ table accesses to find the highest $\mathsf{1}$. The table has 2^b entries of $\log b$ bits. For example, with $b = 16$ we require only 32 kilobytes of memory and can decode any 32-bit number in 1 or 2 accesses. Since most numbers are small when γ-codes are used, it might be even better to use $b = 8$, so the global table is tiny and will normally reside in cache.

Delta (δ) Codes

When numbers are too large for γ-codes to be efficient, we can use δ-codes:

$$\delta(x) = \gamma(|x|) \,.\, [x]_{|x|-1},$$

that is, we γ-encode the length of x and then encode x without its highest bit. For example, $\delta(1) = \delta(\mathsf{1}) = \mathsf{1}$, $\delta(2) = \delta(\mathsf{10}) = \mathsf{0100}$, $\delta(3) = \delta(\mathsf{11}) = \mathsf{0101}$, $\delta(4) = \delta(\mathsf{100}) = \mathsf{01100}$, and so on. It holds

$$\begin{aligned} |\delta(x)| &= |\gamma(|x|)| + |x| - 1 = |x| + 2||x|| - 2 = |x| + 2\lfloor \log |x| \rfloor \\ &= \log x + \mathcal{O}(\log \log x). \end{aligned}$$

The δ-codes are shorter than γ-codes for $x \geq 32$ (and of the same length for $16 \leq x \leq 31$). The δ-codes can be decoded in $\mathcal{O}(1)$ time if γ-codes can. In case a global table is used, it is very small because it must decode numbers up to length $\mathcal{O}(\log |x|) = \mathcal{O}(\log \log x)$. For example, a table of 64 entries suffices to decode any 64-bit number with one access.

Rice Codes

Rice codes choose one parameter ℓ for the whole sequence. Let $y = \lfloor x/2^{\ell} \rfloor$. The Rice code for x is then

$$Rice(x) = u(y+1) \,.\, [x]_{\ell},$$

that is, the ℓ lowest bits of x are encoded preceded by the highest bits, which are encoded in unary. Note that $[x]_{\ell}$ will use ℓ bits even if x has fewer than ℓ significant bits, padding with $\mathsf{0}$s on the left if necessary.

By appropriately choosing ℓ, Rice codes can yield better compression than γ-codes or δ-codes and are the favorite encoders for inverted indexes. Rice codes are a particular case of Golomb codes, which are more complicated and can yield slightly better compression.

Variable Byte (VByte) Codes

These codes usually require more space than the previous ones and are useful only for larger numbers. Their aim is to speed up decoding by ensuring that only byte-aligned data are read. The number x is cut into 7-bit chunks, $x = x_1 \,.\, x_2 \ldots x_k$. Each chunk is then stored in the lowest bits of a byte, whose highest bit is $\mathsf{0}$ for $x_1, \ldots, x_{k-1}$, and $\mathsf{1}$ for x_k.

To read x, we start with $y \leftarrow 0$ and read bytes $b_1, b_2, \ldots, b_k$, until we read a byte $b_k \geq 128$. Each time we read a byte $b_i < 128$, we do $y \leftarrow (y + b_i) \cdot 2^7$. When we read b_k, the final value is $x = y + b_k - 128$. In practice, numbers fit in a few bytes, and this decoding process is faster than the previous ones.

Simple-9 and PforDelta

These codes are aimed at retaining the good space performance of bitwise codes and the good time performance of bytewise codes. They encode and decode a short sequence

of numbers (not each one individually) and read whole computer words from the input. In general, they achieve excellent performance and can be easily decoded in constant time.

Simple-9 encodes as many numbers as possible in a 32-bit word. The highest 4 bits of the word indicate how many numbers are encoded. If the next 28 values to encode are 1 or 2, then we can use one bit for each. Otherwise, if the next 14 values are up to 4, then we can use 2 bits for each. If not, but the next 9 numbers are up to 8, then we can use 3 bits for each (wasting a bit), and so on. There are in total 9 possibilities (i.e., encoding the next $\lfloor 28/m \rfloor$ values using $m = 1, 2, 3, 4, 5, 7, 9, 14$, or 28 bits per value). Numbers over 28 bits cannot be encoded. The variant Simple-16 introduces more cases, combining different lengths, to use all the 16 4-bit combinations.

PforDelta, instead, encodes a fixed amount of numbers at a time (typically 128), using for all of them the number of bits needed for the largest one. A fraction of the largest numbers (usually 10%) is encoded separately, and the other 90% is used to calculate how many bits are needed per number.

Algorithm 2.5 gives the encoding procedures for most of the described codes. It assumes the code fits in an integer variable (c). To compute $|x| = \lfloor \log x \rfloor + 1$ we must find its highest 1, as explained. In vbyte, we assume that the highest bytes will be read first. In simple9, x is an array and we must encode from $x[k]$, leaving k at the next position to encode; function $code(m)$ encodes the chosen value of m using 4 bits.

2.8 Jensen's Inequality

A tool that is frequently used to analyze compression methods is Jensen's inequality. It establishes that, if $f(x)$ is a concave function (that is, $f(\frac{x+y}{2}) \geq \frac{f(x)+f(y)}{2}$ for all x and y, like the logarithm), then

$$f\left(\frac{\sum_i a_i x_i}{\sum_i a_i}\right) \geq \frac{\sum_i a_i f(x_i)}{\sum_i a_i},$$

for any values x_i and positive weights a_i. Equality is reached only if all the x_i values are equal or f is linear. In particular, if we have m values $x_1, \ldots, x_m$ and the weights are all equal, $a_i = \frac{1}{m}$, this gives

$$f\left(\frac{\sum_i x_i}{m}\right) \geq \frac{\sum_i f(x_i)}{m}.$$

Roughly said, the function of the average is larger than the average of the functions. We will use Jensen's inequality many times in the book.

Differential Encoding of Increasing Numbers

Assume we have increasing numbers $0 = y_0 < y_1 < y_2 < \ldots < y_m = n$. As a way to compress them, we store them differentially: we encode $x_1, \ldots, x_m$, where $x_i = y_i - y_{i-1}$. If m is not much smaller than n, then most x_i values will be small, and we can apply the encoding methods just seen. For example, assume we use δ-codes. The total

Algorithm 2.5: Various integer encodings.

Input : x, the number to be encoded, Rice parameter ℓ, and Simple9 position k.
Output: $\langle c, b\rangle$ so that the code consists of the b lowest bits of integer c.

```
1  Proc unary (x)
2      return ⟨1, x⟩

3  Proc gamma (x)
4      return ⟨x, 2 · |x| − 1⟩

5  Proc delta (x)
6      ⟨c′, b′⟩ ← gamma(|x|)
7      return ⟨x + 2^(|x|−1)(c′ − 1), |x| + b′ − 1⟩

8  Proc rice (x, ℓ)
9      y ← ⌊x/2^ℓ⌋
10     return ⟨(x mod 2^ℓ) + 2^ℓ, ℓ + y + 1⟩

11 Proc vbyte (x)
12     c ← 0; b ← 0
13     while x ≥ 128 do
14         c ← c + 2^b · (x mod 128)
15         b ← b + 8
16         x ← ⌊x/128⌋
17     return ⟨c + 2^b · (x + 128), b + 8⟩

18 Proc simple9 (x, k)
19     m ← |x[k]|
20     p ← 1; c ← 0
21     while (p + 1) · max(m, |x[k + p]|) ≤ 28 do
22         m ← max(m, |x[k + p]|)
23         p ← p + 1;
24     while p · (m + 1) ≤ 28 do m ← m + 1
25     while p > 0 do
26         c ← c · 2^m + x[k]
27         k ← k + 1; p ← p − 1
28     return ⟨c + 2^28 · code(m), 32⟩
```

size of the encoding will then be

$$\begin{aligned}\sum_i |\delta(x_i)| &= \sum_i |x_i| + 2\lfloor \log |x_i| \rfloor = \sum_i \lfloor \log x_i \rfloor + 1 + 2\lfloor \log(\lfloor \log x_i \rfloor + 1) \rfloor \\ &= \sum_i \log x_i + 2 \log\log x_i + \mathcal{O}(1) \\ &\leq m \log \frac{n}{m} + 2m \log\log \frac{n}{m} + \mathcal{O}(m),\end{aligned}$$

where we have applied Jensen's inequality twice in the last line, on the concave functions $\log x$ and $\log \log x$, and $a_i = 1/m$.

This also yields our first encoding for a bit sequence $B[1, n]$ with m 1s that gets close to its zero-order entropy, $n\mathcal{H}_0(B) = m \log \frac{n}{m} + \mathcal{O}(m)$ (recall the end of Section 2.3.1): We call y_i the positions of the 1s and δ-encode the gaps x_i between them. Note that Huffman coding is useless in principle to compress a bit sequence, as it needs at least one bit per bit. We will see, however, successful encodings in Chapter 4, by encoding the bits in groups.

Concatenations of Strings

Another consequence of Jensen's inequality we will use later is the following. Let S_1 and S_2 be two strings of lengths n_1 and n_2, respectively, and $S = S_1 . S_2$ be their concatenation, of length $n = n_1 + n_2$. Then $n_1\mathcal{H}_0(S_1) + n_2\mathcal{H}_0(S_2) \leq n\mathcal{H}_0(S)$. To see this, let us call $n_{s,1}$, $n_{s,2}$, and n_s the number of occurrences of symbol s in S_1, S_2, and S, respectively. Then we have

$$\begin{aligned} n_1\mathcal{H}_0(S_1) + n_2\mathcal{H}_0(S_2) &= \sum_{s\in\Sigma} n_{s,1} \log \frac{n_1}{n_{s,1}} + \sum_{s\in\Sigma} n_{s,2} \log \frac{n_2}{n_{s,2}} \\ &= \sum_{s\in\Sigma} \left(n_{s,1} \log \frac{n_1}{n_{s,1}} + n_{s,2} \log \frac{n_2}{n_{s,2}} \right) \leq \sum_{s\in\Sigma} n_s \log \frac{n}{n_s} = n\mathcal{H}_0(S). \end{aligned}$$

To obtain the inequality we have used Jensen's formula on each $s \in \Sigma$, with $i \in \{1, 2\}$, $a_i = n_{s,i}$, $x_i = \frac{n_i}{n_{s,i}}$, and $f = \log$.

2.9 Application: Positional Inverted Indexes

A (positional) inverted index is a data structure that provides fast word searches on a natural language text T. For each distinct word s of the text, the index stores the list of the positions where s appears in the text, in increasing order. Let n be the number of words in T, and n_s the number of times word s appears in T. Then the positions stored by the list of word s form a sequence $0 < p_1 < p_2 < \ldots < p_{n_s} \leq n$. If we encode the differences using δ-codes as in Section 2.8, the total space used by the list is $n_s \log \frac{n}{n_s} + 2n_s \log \log \frac{n}{n_s} + \mathcal{O}(n_s)$ bits. Summing this space over all the words s in the text, we obtain

$$\sum_s n_s \log \frac{n}{n_s} + 2n_s \log \log \frac{n}{n_s} + \mathcal{O}(n_s) \leq n\mathcal{H}_0(T) + 2n \log \mathcal{H}_0(T) + \mathcal{O}(n)$$

bits, where $\mathcal{H}_0(T) = \sum_s \frac{n_s}{n} \log \frac{n}{n_s}$ is the zero-order empirical entropy of T if regarded as a sequence of words. The intriguing part of the inequality is the second term, $2n \log \mathcal{H}_0(T)$. It comes from $\sum_s 2n_s \log \log \frac{n}{n_s}$, by applying Jensen's inequality with $f = \log$, $a_s = n_s$, and $x_s = \log \frac{n}{n_s}$.

Note that $n\mathcal{H}_0(T)$ is basically the space reached by a Huffman compression of T if we take the words as the basic symbols. As said, such a Huffman compression reduces T to about 25% of its plain representation on typical English texts. The fact that the positional inverted index can be compressed as much as T should not be surprising: the

index can be regarded as an alternative representation of T, as T can be reconstructed from the lists.

In both cases, the actual words must be stored separately, but the vocabulary is usually small compared to the text size. An empirical law known as Heaps' law states that the number of distinct words in a text of n words grows as $\mathcal{O}(n^\beta)$, where $0 < \beta < 1$ is a constant that depends on the text type (and is in practice close to 0.5).

2.10 Summary

The worst-case entropy of a set $\mathcal{U}$ is $\log |\mathcal{U}|$, the minimum number of bits needed to distinguish an element in the worst case. When each $u \in \mathcal{U}$ is assigned a probability p_u, then Information Theory establishes that the minimum average code length is $\mathcal{H} = \sum_{u \in \mathcal{U}} p_u \log \frac{1}{p_u}$. To compress specific sequences one can use their empirical entropy, which estimates p_u from the frequencies in the sequence. Optimal coding methods like Huffman reach a code length below $\mathcal{H} + 1$ bits per element. On large or infinite sets like the natural numbers one may use fixed codes that optimize the space usage for certain common distributions.

2.11 Bibliographic Notes

In this chapter we focused on *semi-static* compression. This means that first we compute the probabilities of the symbols, then we build the codes, and finally we encode the symbols, all using the same codes. Instead of these steps, *adaptive* compression gathers and updates the probabilities as it compresses the sequence, performing only one pass. Although dynamic compression is convenient in many cases, semi-static compression is more appropriate for the compact data structures we use in this book.

An excellent book on Information Theory is by Cover and Thomas (2006). It covers entropy and coding, although it does not focus on the practical aspects of efficient coding. This is well covered in many books on compression (Storer, 1988; Bell *et al.*, 1990; Witten *et al.*, 1999; Moffat and Turpin, 2002; Solomon, 2007; Salomon *et al.*, 2013). While these describe most of the topics we have covered, some parts of the chapter deserve further references.

Worst-case entropy. Computing worst-case entropies is a matter of counting the number of combinatorial objects of a certain kind. There are excellent books on this topic (Sedgewick and Flajolet, 2013; Graham *et al.*, 1994).

Shannon and empirical entropy. Modern Information Theory started with the seminal work of Shannon (1948), presented more in depth in the book by Shannon and Weaver (1949). Gagie (2006) gives more insights on the limits of the kth-order empirical entropy measure.

Huffman codes. Huffman (1952) found the well-known algorithm to build an optimal prefix code. Schwartz and Kallick (1964) introduced canonical Huffman codes. Katona