Handbook of Data Compression

Fifth Edition

David Salomon
Giovanni Motta

With Contributions by David Bryant

Handbook of Data Compression

Fifth Edition

Previous editions published under the title
“Data Compression: The Complete Reference”

Springer

Prof. David Salomon (emeritus)
Computer Science Dept.
California State University, Northridge
Northridge, CA 91330-8281
USA
dsalomon@csun.edu

Dr. Giovanni Motta
Personal Systems Group, Mobility Solutions
Hewlett-Packard Corp.
10955 Tantau Ave.
Cupertino, California 95014-0770
gim@ieee.org

ISBN 978-1-84882-902-2 e-ISBN 978-1-84882-903-9
DOI 10.1007/10.1007/978-1-84882-903-9
Springer London Dordrecht Heidelberg New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

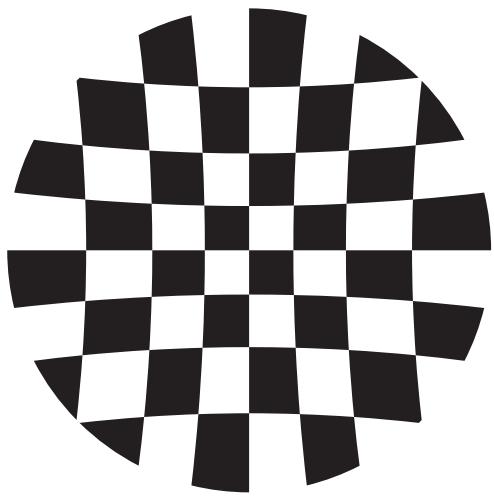
Library of Congress Control Number: 2009936315

Cover design: eStudio Calamar S.L.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To users of data compression everywhere



I love being a writer. What I can't stand is the paperwork.
—Peter De Vries

Preface to the New Handbook

Gentle Reader. The thick, heavy volume you are holding in your hands was intended to be the fifth edition of *Data Compression: The Complete Reference*. Instead, its title indicates that this is a handbook of data compression. What makes a book a handbook? What is the difference between a textbook and a handbook? It turns out that "handbook" is one of the many terms that elude precise definition. The many definitions found in dictionaries and reference books vary widely and do more to confuse than to illuminate the reader. Here are a few examples:

- A concise reference book providing specific information about a subject or location (but this book is not concise).
- A type of reference work that is intended to provide ready reference (but every reference work should provide ready reference).
- A pocket reference is intended to be carried at all times (but this book requires big pockets as well as deep ones).
- A small reference book; a manual (definitely does not apply to this book).
- General information source which provides quick reference for a given subject area. Handbooks are generally subject-specific (true for this book).

Confusing; but we will use the last of these definitions. The aim of this book is to provide a quick reference for the subject of data compression. Judging by the size of the book, the "reference" is certainly there, but what about "quick?" We believe that the following features make this book a quick reference:

- The detailed index which constitutes 3% of the book.
- The glossary. Most of the terms, concepts, and techniques discussed throughout the book appear also, albeit briefly, in the glossary.

- The particular organization of the book. Data is compressed by removing redundancies in its original representation, and these redundancies depend on the type of data. Text, images, video, and audio all have different types of redundancies and are best compressed by different algorithms which in turn are based on different approaches. Thus, the book is organized by different data types, with individual chapters devoted to image, video, and audio compression techniques. Some approaches to compression, however, are general and work well on many different types of data, which is why the book also has chapters on variable-length codes, statistical methods, dictionary-based methods, and wavelet methods.

The main body of this volume contains 11 chapters and one appendix, all organized in the following categories, basic methods of compression, variable-length codes, statistical methods, dictionary-based methods, methods for image compression, wavelet methods, video compression, audio compression, and other methods that do not conveniently fit into any of the above categories. The appendix discusses concepts of information theory, the theory that provides the foundation of the entire field of data compression.

In addition to its use as a quick reference, this book can be used as a starting point to learn more about approaches to and techniques of data compression as well as specific algorithms and their implementations and applications. The broad coverage makes the book as complete as practically possible. The extensive bibliography will be very helpful to those looking for more information on a specific topic. The liberal use of illustrations and tables of data helps to clarify the text.

This book is aimed at readers who have general knowledge of computer applications, binary data, and files and want to understand how different types of data can be compressed. The book is not for dummies, nor is it a guide to implementors. Someone who wants to implement a compression algorithm A should have coding experience and should rely on the original publication by the creator of A.

In spite of the growing popularity of Internet searching, which often locates quantities of information of questionable quality, we feel that there is still a need for a concise, reliable reference source spanning the full range of the important field of data compression.

New to the Handbook

The following is a list of the new material in this book (material not included in past editions of *Data Compression: The Complete Reference*).

- The topic of compression benchmarks has been added to the Introduction.
- The paragraphs titled "How to Hide Data" in the Introduction show how data compression can be utilized to quickly and efficiently hide data in plain sight in our computers.
- Several paragraphs on compression curiosities have also been added to the Introduction.
- The new Section 1.1.2 shows why irreversible compression may be useful in certain situations.
- Chapters 2 through 4 discuss the all-important topic of variable-length codes. These chapters discuss basic, advanced, and robust variable-length codes. Many types of VL

codes are known, they are used by many compression algorithms, have different properties, and are based on different principles. The most-important types of VL codes are prefix codes and codes that include their own length.

- Section 2.9 on phased-in codes was wrong and has been completely rewritten.

- An example of the start-step-stop code $(2, 2, \infty)$ has been added to Section 3.2.

- Section 3.5 is a description of two interesting variable-length codes dubbed recursive bottom-up coding (RBUC) and binary adaptive sequential coding (BASC). These codes represent compromises between the standard binary (β) code and the Elias gamma codes.

- Section 3.28 discusses the original method of interpolative coding whereby dynamic variable-length codes are assigned to a strictly monotonically increasing sequence of integers.

- Section 5.8 is devoted to the compression of PK (packed) fonts. These are older bitmaps fonts that were developed as part of the huge TEX project. The compression algorithm is not especially efficient, but it provides a rare example of run-length encoding (RLE) without the use of Huffman codes.

- Section 5.13 is about the Hutter prize for text compression.

- PAQ (Section 5.15) is an open-source, high-performance compression algorithm and free software that features sophisticated prediction combined with adaptive arithmetic encoding. This free algorithm is especially interesting because of the great interest it has generated and because of the many versions, subversions, and derivatives that have been spun off it.

- Section 6.3.2 discusses LZR, a variant of the basic LZ77 method, where the lengths of both the search and look-ahead buffers are unbounded.

- Section 6.4.1 is a description of LZB, an extension of LZSS. It is the result of evaluating and comparing several data structures and variable-length codes with an eye to improving the performance of LZSS.

- SLH, the topic of Section 6.4.2, is another variant of LZSS. It is a two-pass algorithm where the first pass employs a hash table to locate the best match and to count frequencies, and the second pass encodes the offsets and the raw symbols with Huffman codes prepared from the frequencies counted by the first pass.

- Most LZ algorithms were developed during the 1980s, but LZPP, the topic of Section 6.5, is an exception. LZPP is a modern, sophisticated algorithm that extends LZSS in several directions and has been inspired by research done and experience gained by many workers in the 1990s. LZPP identifies several sources of redundancy in the various quantities generated and manipulated by LZSS and exploits these sources to obtain better overall compression.

- Section 6.14.1 is devoted to LZT, an extension of UNIX compress/LZC. The major innovation of LZT is the way it handles a full dictionary.

- LZJ (Section 6.17) is an interesting LZ variant. It stores in its dictionary, which can be viewed either as a multiway tree or as a forest, *every* phrase found in the input. If a phrase is found n times in the input, only one copy is stored in the dictionary. Such behavior tends to fill the dictionary up very quickly, so LZJ limits the length of phrases to a preset parameter h.

- The interesting, original concept of antidictionary is the topic of Section 6.31. A dictionary-based encoder maintains a list of bits and pieces of the data and employs this list to compress the data. An antidictionary method, on the other hand, maintains a list of strings that do not appear in the data. This generates negative knowledge that allows the encoder to predict with certainty the values of many bits and thus to drop those bits from the output, thereby achieving compression.

- The important term "pixel" is discussed in Section 7.1, where the reader will discover that a pixel is not a small square, as is commonly assumed, but a mathematical point.

- Section 7.10.8 discusses the new HD photo (also known as JPEG XR) compression method for continuous-tone still images.

- ALPC (*adaptive linear prediction and classification*), is a lossless image compression algorithm described in Section 7.12. ALPC is based on a linear predictor whose coefficients are computed for each pixel individually in a way that can be mimiced by the decoder.

- Grayscale Two-Dimensional Lempel-Ziv Encoding (GS-2D-LZ, Section 7.18) is an innovative dictionary-based method for the lossless compression of grayscale images.

- Section 7.19 has been partially rewritten.

- Section 7.40 is devoted to spatial prediction, a combination of JPEG and fractal-based image compression.

- A short historical overview of video compression is provided in Section 9.4.

- The all-important H.264/AVC video compression standard has been extended to allow for a compressed stream that supports temporal, spatial, and quality scalable video coding, while retaining a base layer that is still backward compatible with the original H.264/AVC. This extension is the topic of Section 9.10.

- The complex and promising VC-1 video codec is the topic of the new, long Section 9.11.

- The new Section 11.6.4 treats the topic of syllable-based compression, an approach to compression where the basic data symbols are syllables, a syntactic form between characters and words.

- The commercial compression software known as stuffit has been around since 1987. The methods and algorithms it employs are proprietary, but some information exists in various patents. The new Section 11.16 is an attempt to describe what is publicly known about this software and how it works.

- There is now a short appendix that presents and explains the basic concepts and terms of information theory.

We would like to acknowledge the help, encouragement, and cooperation provided by Yuriy Reznik, Matt Mahoney, Mahmoud El-Sakka, Pawel Pylak, Darryl Lovato, Raymond Lau, Cosmin Truţa, Derong Bao, and Honggang Qi. They sent information, reviewed certain sections, made useful comments and suggestions, and corrected numerous errors.

A special mention goes to David Bryant who wrote Section 10.11.

Springer Verlag has created the Springer Handbook series on important scientific and technical subjects, and there can be no doubt that data compression should be included in this category. We are therefore indebted to our editor, Wayne Wheeler, for proposing this project and providing the encouragement and motivation to see it through.

The book's Web site is located at `www.DavidSalomon.name`. Our email addresses are `dsalomon@csun.edu` and `gim@ieee.org` and readers are encouraged to message us with questions, comments, and error corrections.

Those interested in data compression in general should consult the short section titled "Joining the Data Compression Community," at the end of the book, as well as the following resources:

- `http://compression.ca/`,
- `http://www-isl.stanford.edu/~gray/iii.html`,
- `http://www.hn.is.uec.ac.jp/~arimura/compression_links.html`, and
- `http://datacompression.info/`.

(URLs are notoriously short lived, so search the Internet.)

David Salomon Giovanni Motta

The preface is usually that part of a
book which can most safely be omitted.
—William Joyce, *Twilight Over England* (1940)

Preface to the Fourth Edition

(This is the Preface to the 4th edition of *Data Compression: The Complete Reference*, the predecessor of this volume.) I was pleasantly surprised when in November 2005 a message arrived from Wayne Wheeler, the new computer science editor of Springer Verlag, notifying me that he intends to qualify this book as a Springer major reference work (MRW), thereby releasing past restrictions on page counts, freeing me from the constraint of having to compress my style, and making it possible to include important and interesting data compression methods that were either ignored or mentioned in passing in previous editions.

> These fascicles will represent my best attempt to write a comprehensive account, but computer science has grown to the point where I cannot hope to be an authority on all the material covered in these books. Therefore I'll need feedback from readers in order to prepare the official volumes later.
>
> I try to learn certain areas of computer science exhaustively; then I try to digest that knowledge into a form that is accessible to people who don't have time for such study.
>
> —Donald E. Knuth, `http://www-cs-faculty.stanford.edu/~knuth/` (2006)

Naturally, all the errors discovered by me and by readers in the third edition have been corrected. Many thanks to all those who bothered to send error corrections, questions, and comments. I also went over the entire book and made numerous additions, corrections, and improvements. In addition, the following new topics have been included in this edition:

- Tunstall codes (Section 2.6). The advantage of variable-size codes is well known to readers of this book, but these codes also have a downside; they are difficult to work with. The encoder has to accumulate and append several such codes in a short buffer, wait until n bytes of the buffer are full of code bits (where n must be at least 1), write the n bytes on the output, shift the buffer n bytes, and keep track of the location of the last bit placed in the buffer. The decoder has to go through the reverse process.

The idea of Tunstall codes is to construct a set of fixed-size codes, each encoding a variable-size string of input symbols. As an aside, the "pod" code (Table 10.29) is also a new addition.

- Recursive range reduction (3R) (Section 1.7) is a simple coding algorithm due to Yann Guidon that offers decent compression, is easy to program, and its performance is independent of the amount of data to be compressed.

- LZARI, by Haruhiko Okumura (Section 6.4.3), is an improvement of LZSS.

- RAR (Section 6.22). The popular RAR software is the creation of Eugene Roshal. RAR has two compression modes, general and special. The general mode employs an LZSS-based algorithm similar to ZIP Deflate. The size of the sliding dictionary in RAR can be varied from 64 Kb to 4 Mb (with a 4 Mb default value) and the minimum match length is 2. Literals, offsets, and match lengths are compressed further by a Huffman coder. An important feature of RAR is an error-control code that increases the reliability of RAR archives while being transmitted or stored.

- 7-z and LZMA (Section 6.26). LZMA is the main (as well as the default) algorithm used in the popular `7z` (or `7-Zip`) compression software [7z 06]. Both `7z` and LZMA are the creations of Igor Pavlov. The software runs on Windows and is free. Both LZMA and `7z` were designed to provide high compression, fast decompression, and low memory requirements for decompression.

- Stephan Wolf made a contribution to Section 7.34.4.

- H.264 (Section 9.9). H.264 is an advanced video codec developed by the ISO and the ITU as a replacement for the existing video compression standards H.261, H.262, and H.263. H.264 has the main components of its predecessors, but they have been extended and improved. The only new component in H.264 is a (wavelet based) filter, developed specifically to reduce artifacts caused by the fact that individual macroblocks are compressed separately.

- Section 10.4 is devoted to the `WAVE` audio format. `WAVE` (or simply Wave) is the native file format employed by the Windows opearting system for storing digital audio data.

- FLAC (Section 10.10). FLAC (free lossless audio compression) is the brainchild of Josh Coalson who developed it in 1999 based on ideas from Shorten. FLAC was especially designed for audio compression, and it also supports streaming and archival of audio data. Coalson started the FLAC project on the well-known sourceforge Web site [sourceforge.flac 06] by releasing his reference implementation. Since then many developers have contributed to improving the reference implementation and writing alternative implementations. The FLAC project, administered and coordinated by Josh Coalson, maintains the software and provides a reference codec and input plugins for several popular audio players.

- WavPack (Section 10.11, written by David Bryant). WavPack [WavPack 06] is a completely open, multiplatform audio compression algorithm and software that supports three compression modes, lossless, high-quality lossy, and a unique hybrid compression

mode. It handles integer audio samples up to 32 bits wide and also 32-bit IEEE floating-point data [IEEE754 85]. The input stream is partitioned by WavPack into blocks that can be either mono or stereo and are generally 0.5 seconds long (but the length is actually flexible). Blocks may be combined in sequence by the encoder to handle multichannel audio streams. All audio sampling rates are supported by WavPack in all its modes.

- Monkey's audio (Section 10.12). Monkey's audio is a fast, efficient, free, lossless audio compression algorithm and implementation that offers error detection, tagging, and external support.

- MPEG-4 ALS (Section 10.13). MPEG-4 Audio Lossless Coding (ALS) is the latest addition to the family of MPEG-4 audio codecs. ALS can input floating-point audio samples and is based on a combination of linear prediction (both short-term and long-term), multichannel coding, and efficient encoding of audio residues by means of Rice codes and block codes (the latter are also known as block Gilbert-Moore codes, or BGMC [Gilbert and Moore 59] and [Reznik 04]). Because of this organization, ALS is not restricted to the encoding of audio signals and can efficiently and losslessly compress other types of fixed-size, correlated signals, such as medical (ECG and EEG) and seismic data.

- AAC (Section 10.15). AAC (advanced audio coding) is an extension of the three layers of MPEG-1 and MPEG-2, which is why it is often called `mp4`. It started as part of the MPEG-2 project and was later augmented and extended as part of MPEG-4. Apple Computer has adopted AAC in 2003 for use in its well-known iPod, which is why many believe (wrongly) that the acronym AAC stands for apple audio coder.

- Dolby AC-3 (Section 10.16). AC-3, also known as Dolby Digital, stands for Dolby's third-generation audio coder. AC-3 is a perceptual audio codec based on the same principles as the three MPEG-1/2 layers and AAC. The new section included in this edition concentrates on the special features of AC-3 and what distinguishes it from other perceptual codecs.

- Portable Document Format (PDF, Section 11.13). PDF is a popular standard for creating, editing, and printing documents that are independent of any computing platform. Such a document may include text and images (graphics and photos), and its components are compressed by well-known compression algorithms.

- Section 11.14 (written by Giovanni Motta) covers a little-known but important aspect of data compression, namely how to compress the differences between two files.

- Hyperspectral data compression (Section 11.15, partly written by Giovanni Motta) is a relatively new and growing field. Hyperspectral data is a set of data items (called pixels) arranged in rows and columns where each pixel is a vector. A home digital camera focuses visible light on a sensor to create an image. In contrast, a camera mounted on a spy satellite (or a satellite searching for minerals and other resources) collects and measures radiation of many wavelegths. The intensity of each wavelength is converted into a number, and the numbers collected from one point on the ground form a vector that becomes a pixel of the hyperspectral data.

Another pleasant change is the great help I received from Giovanni Motta, David Bryant, and Cosmin Truţa. Each proposed topics for this edition, went over some of

the new material, and came up with constructive criticism. In addition, David wrote Section 10.11 and Giovanni wrote Section 11.14 and part of Section 11.15.

I would like to thank the following individuals for information about certain topics and for clearing up certain points. Igor Pavlov for help with `7z` and LZMA, Stephan Wolf for his contribution, Matt Ashland for help with Monkey's audio, Yann Guidon for his help with recursive range reduction (3R), Josh Coalson for help with FLAC, and Eugene Roshal for help with RAR.

> In the first volume of this biography I expressed my gratitude to those individuals and corporate bodies without whose aid or encouragement it would not have been undertaken at all; and to those others whose help in one way or another advanced its progress. With the completion of this volume my obligations are further extended. I should like to express or repeat my thanks to the following for the help that they have given and the premissions they have granted.
>
> Christabel Lady Aberconway; Lord Annan; Dr Igor Anrep; ...
>
> —Quentin Bell, *Virginia Woolf: A Biography* (1972)

Currently, the book's Web site is part of the author's Web site, which is located at `http://www.ecs.csun.edu/~dsalomon/`. Domain `DavidSalomon.name` has been reserved and will always point to any future location of the Web site. The author's email address is `dsalomon@csun.edu`, but email sent to ⟨*anyname*⟩`@DavidSalomon.name` will be forwarded to the author.

Those interested in data compression in general should consult the short section titled "Joining the Data Compression Community," at the end of the book, as well as the following resources:

- `http://compression.ca/`,
- `http://www-isl.stanford.edu/~gray/iii.html`,
- `http://www.hn.is.uec.ac.jp/~arimura/compression_links.html`, and
- `http://datacompression.info/`.

(URLs are notoriously short lived, so search the Internet).

> People err who think my art comes easily to me.
>
> —Wolfgang Amadeus Mozart

Lakeside, California David Salomon

Contents

Content comes first. . . yet excellent design can catch people's eyes and impress the contents on their memory.

—Hideki Nakajima

Introduction

Giambattista della Porta, a Renaissance scientist sometimes known as the professor of secrets, was the author in 1558 of *Magia Naturalis* (Natural Magic), a book in which he discusses many subjects, including demonology, magnetism, and the camera obscura [della Porta 58]. The book became tremendously popular in the 16th century and went into more than 50 editions, in several languages beside Latin. The book mentions an imaginary device that has since become known as the "sympathetic telegraph." This device was to have consisted of two circular boxes, similar to compasses, each with a magnetic needle. Each box was to be labeled with the 26 letters, instead of the usual directions, and the main point was that the two needles were supposed to be magnetized by the *same lodestone*. Porta assumed that this would somehow coordinate the needles such that when a letter was dialed in one box, the needle in the other box would swing to point to the same letter.

Needless to say, such a device does not work (this, after all, was about 300 years before Samuel Morse), but in 1711 a worried wife wrote to the *Spectator*, a London periodical, asking for advice on how to bear the long absences of her beloved husband. The adviser, Joseph Addison, offered some practical ideas, then mentioned Porta's device, adding that a pair of such boxes might enable her and her husband to communicate with each other even when they "were guarded by spies and watches, or separated by castles and adventures." Mr. Addison then added that, in addition to the 26 letters, the sympathetic telegraph dials should contain, when used by lovers, "several entire words which always have a place in passionate epistles." The message "I love you," for example, would, in such a case, require sending just three symbols instead of ten.

> A woman seldom asks advice before she has bought her wedding clothes.
>
> —Joseph Addison

This advice is an early example of *text compression* achieved by using short codes for common messages and longer codes for other messages. Even more importantly, this shows how the concept of data compression comes naturally to people who are interested in communications. We seem to be preprogrammed with the idea of sending as little data as possible in order to save time.

Data compression is the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, the bitstream, or the compressed stream) that has a smaller size. A stream can be a file, a buffer in memory, or individual bits sent on a communications channel.

The decades of the 1980s and 1990s saw an exponential decrease in the cost of digital storage. There seems to be no need to compress data when it can be stored inexpensively in its raw format, yet the same two decades have also experienced rapid progress in the development and applications of data compression techniques and algorithms. The following paragraphs try to explain this apparent paradox.

- Many like to accumulate data and hate to throw anything away. No matter how big a storage device one has, sooner or later it is going to overflow. Data compression is useful because it delays this inevitability.

- As storage devices get bigger and cheaper, it becomes possible to create, store, and transmit larger and larger data files. In the old days of computing, most files were text or executable programs and were therefore small. No one tried to create and process other types of data simply because there was no room in the computer. In the 1970s, with the advent of semiconductor memories and floppy disks, still images, which require bigger files, became popular. These were followed by audio and video files, which require even bigger files.

- We hate to wait for data transfers. When sitting at the computer, waiting for a Web page to come in or for a file to download, we naturally feel that anything longer than a few seconds is a long time to wait. Compressing data before it is transmitted is therefore a natural solution.

- CPU speeds and storage capacities have increased dramatically in the last two decades, but the speed of mechanical components (and therefore the speed of disk input/output) has increased by a much smaller factor. Thus, it makes sense to store data in compressed form, even if plenty of storage space is still available on a disk drive. Compare the following scenarios: (1) A large program resides on a disk. It is read into memory and is executed. (2) The same program is stored on the disk in compressed form. It is read into memory, decompressed, and executed. It may come as a surprise to learn that the latter case is faster in spite of the extra CPU work involved in decompressing the program. This is because of the huge disparity between the speeds of the CPU and the mechanical components of the disk drive.

- A similar situation exists with regard to digital communications. Speeds of communications channels, both wired and wireless, are increasing steadily but not dramatically. It therefore makes sense to compress data sent on telephone lines between fax machines, data sent between cellular telephones, and data (such as web pages and television signals) sent to and from satellites.

The field of data compression is often called *source coding*. We imagine that the input symbols (such as bits, ASCII codes, bytes, audio samples, or pixel values) are emitted by a certain information source and have to be coded before being sent to their destination. The source can be *memoryless*, or it can have memory. In the former case, each symbol is independent of its predecessors. In the latter case, each symbol depends

on some of its predecessors and, perhaps, also on its successors, so they are correlated. A memoryless source is also termed "independent and identically distributed" or IIID.

Data compression has come of age in the last 20 years. Both the quantity and the quality of the body of literature in this field provide ample proof of this. However, the need for compressing data has been felt in the past, even before the advent of computers, as the following quotation suggests:

> I have made this letter longer than usual because I lack the time to make it shorter.
>
> —Blaise Pascal

There are many known methods for data compression. They are based on different ideas, are suitable for different types of data, and produce different results, but they are all based on the same principle, namely they compress data by removing *redundancy* from the original data in the source file. Any nonrandom data has some structure, and this structure can be exploited to achieve a smaller representation of the data, a representation where no structure is discernible. The terms *redundancy* and *structure* are used in the professional literature, as well as *smoothness*, *coherence*, and *correlation*; they all refer to the same thing. Thus, redundancy is a key concept in any discussion of data compression.

◇ **Exercise Intro.1:** (Fun) Find English words that contain all five vowels "aeiou" in their original order.

In typical English text, for example, the letter `E` appears very often, while `Z` is rare (Tables Intro.1 and Intro.2). This is called *alphabetic redundancy*, and it suggests assigning variable-length codes to the letters, with `E` getting the shortest code and `Z` getting the longest code. Another type of redundancy, *contextual redundancy*, is illustrated by the fact that the letter `Q` is almost always followed by the letter `U` (i.e., that in plain English certain digrams and trigrams are more common than others). Redundancy in images is illustrated by the fact that in a nonrandom image, adjacent pixels tend to have similar colors.

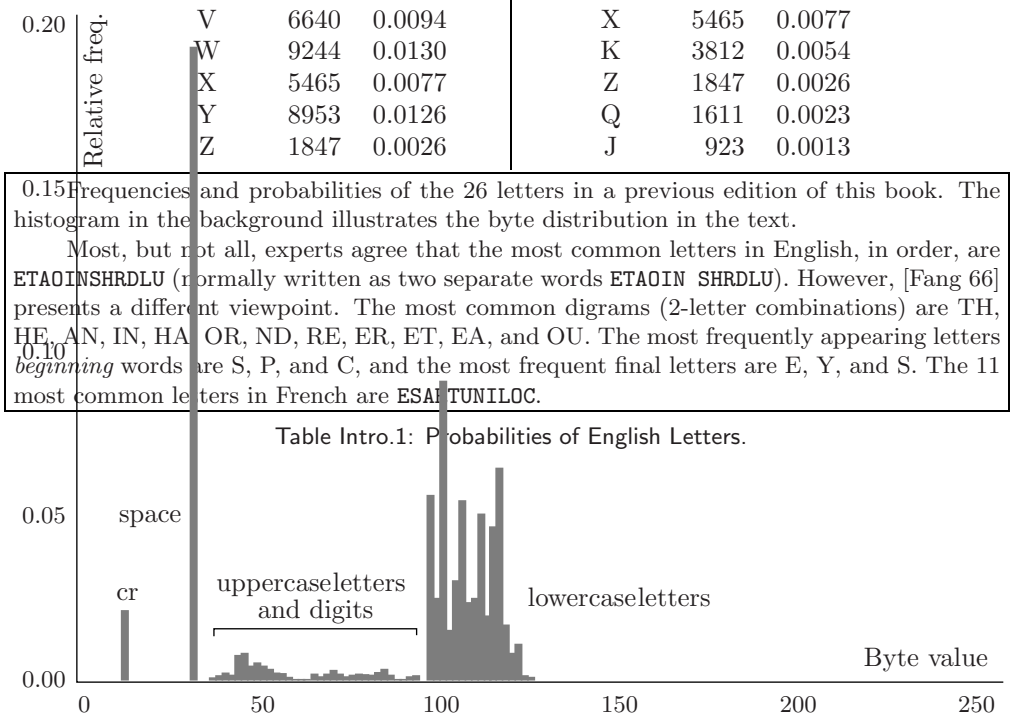
Section A.1 discusses the theory of information and presents a rigorous definition of redundancy. However, even without a precise definition for this term, it is intuitively clear that a variable-length code has less redundancy than a fixed-length code (or no redundancy at all). Fixed-length codes make it easier to work with text, so they are useful, but they are redundant.

The idea of compression by reducing redundancy suggests the *general law* of data compression, which is to "assign short codes to common events (symbols or phrases) and long codes to rare events." There are many ways to implement this law, and an analysis of any compression method shows that, deep inside, it works by obeying the general law.

Compressing data is done by changing its representation from inefficient (i.e., long) to efficient (short). Compression is therefore possible only because data is normally represented in the computer in a format that is longer than absolutely necessary. The reason that inefficient (long) data representations are used all the time is that they make it easier to process the data, and data processing is more common and more important than data compression. The ASCII code for characters is a good example of a data

Letter	Freq.	Prob.	Letter	Freq.	Prob.
A	51060	0.0721	E	86744	0.1224
B	17023	0.0240	T	64364	0.0908
C	27937	0.0394	I	55187	0.0779
D	26336	0.0372	S	51576	0.0728
E	86744	0.1224	A	51060	0.0721
F	19302	0.0272	O	48277	0.0681
G	12640	0.0178	N	45212	0.0638
H	31853	0.0449	R	45204	0.0638
I	55187	0.0779	H	31853	0.0449
J	923	0.0013	L	30201	0.0426
K	3812	0.0054	C	27937	0.0394
L	30201	0.0426	D	26336	0.0372
M	20002	0.0282	P	20572	0.0290
N	45212	0.0638	M	20002	0.0282
O	48277	0.0681	F	19302	0.0272
P	20572	0.0290	B	17023	0.0240
Q	1611	0.0023	U	16687	0.0235
R	45204	0.0638	G	12640	0.0178
S	51576	0.0728	W	9244	0.0130
T	64364	0.0908	Y	8953	0.0126
U	16687	0.0235	V	6640	0.0094
V	6640	0.0094	X	5465	0.0077
W	9244	0.0130	K	3812	0.0054
X	5465	0.0077	Z	1847	0.0026
Y	8953	0.0126	Q	1611	0.0023
Z	1847	0.0026	J	923	0.0013

Table Intro.1: Probabilities of English Letters.



Frequencies and probabilities of the 26 letters in a previous edition of this book. The histogram in the background illustrates the byte distribution in the text.

Most, but not all, experts agree that the most common letters in English, in order, are `ETAOINSHRDLU` (normally written as two separate words `ETAOIN SHRDLU`). However, [Fang 66] presents a different viewpoint. The most common digrams (2-letter combinations) are TH, HE, AN, IN, HA, OR, ND, RE, ER, ET, EA, and OU. The most frequently appearing letters *beginning* words are S, P, and C, and the most frequent final letters are E, Y, and S. The 11 most common letters in French are `ESARTUNILOC`.

Char.	Freq.	Prob.	Char.	Freq.	Prob.	Char.	Freq.	Prob.
e	85537	0.099293	x	5238	0.006080	F	1192	0.001384
t	60636	0.070387	|	4328	0.005024	H	993	0.001153
i	53012	0.061537	-	4029	0.004677	B	974	0.001131
s	49705	0.057698	)	3936	0.004569	W	971	0.001127
a	49008	0.056889	(	3894	0.004520	+	923	0.001071
o	47874	0.055573	T	3728	0.004328	!	895	0.001039
n	44527	0.051688	k	3637	0.004222	#	856	0.000994
r	44387	0.051525	3	2907	0.003374	D	836	0.000970
h	30860	0.035823	4	2582	0.002997	R	817	0.000948
l	28710	0.033327	5	2501	0.002903	M	805	0.000934
c	26041	0.030229	6	2190	0.002542	;	761	0.000883
d	25500	0.029601	I	2175	0.002525	/	698	0.000810
m	19197	0.022284	^	2143	0.002488	N	685	0.000795
\	19140	0.022218	:	2132	0.002475	G	566	0.000657
p	19055	0.022119	A	2052	0.002382	j	508	0.000590
f	18110	0.021022	9	1953	0.002267	@	460	0.000534
u	16463	0.019111	[	1921	0.002230	Z	417	0.000484
b	16049	0.018630	C	1896	0.002201	J	415	0.000482
.	12864	0.014933	]	1881	0.002183	O	403	0.000468
1	12335	0.014319	’	1876	0.002178	V	261	0.000303
g	12074	0.014016	S	1871	0.002172	X	227	0.000264
0	10866	0.012613	_	1808	0.002099	U	224	0.000260
,	9919	0.011514	7	1780	0.002066	?	177	0.000205
&	8969	0.010411	8	1717	0.001993	K	175	0.000203
y	8796	0.010211	‘	1577	0.001831	%	160	0.000186
w	8273	0.009603	=	1566	0.001818	Y	157	0.000182
$	7659	0.008891	P	1517	0.001761	Q	141	0.000164
}	6676	0.007750	L	1491	0.001731	>	137	0.000159
{	6676	0.007750	q	1470	0.001706	*	120	0.000139
v	6379	0.007405	z	1430	0.001660	<	99	0.000115
2	5671	0.006583	E	1207	0.001401	”	8	0.000009

Frequencies and probabilities of the 93 most-common characters in a prepublication previous edition of this book, containing 861,462 characters. See Figure Intro.3 for the Mathematica code.

Table Intro.2: Frequencies and Probabilities of Characters.

representation that is longer than absolutely necessary. It uses 7-bit codes because fixed-size codes are easy to work with. A variable-size code, however, would be more efficient, since certain characters are used more than others and so could be assigned shorter codes.

In a world where data is always represented by its shortest possible format, there would therefore be no way to compress data. Instead of writing books on data compression, authors in such a world would write books on how to determine the shortest format for different types of data.

```
fpc = OpenRead["test.txt"];
g = 0; ar = Table[{i, 0}, {i, 256}];
While[0 == 0,
 g = Read[fpc, Byte];
 (* Skip space, newline & backslash *)
 If[g==10||g==32||g==92, Continue[]];
 If[g==EndOfFile, Break[]];
 ar[[g, 2]]++] (* increment counter *)
Close[fpc];
ar = Sort[ar, #1[[2]] > #2[[2]] &];
tot = Sum[
ar[[i,2]], {i,256}] (* total chars input *)
Table[{FromCharacterCode[ar[[i,1]]],ar[[i,2]],ar[[i,2]]/N[tot,4]},
 {i,93}] (* char code, freq., percentage *)
TableForm[%]
```

Figure Intro.3: Code for Table Intro.2.

A Word to the Wise ...

The main aim of the field of data compression is, of course, to develop methods for better and faster compression. However, one of the main dilemmas of the *art* of data compression is when to stop looking for better compression. Experience shows that fine-tuning an algorithm to squeeze out the last remaining bits of redundancy from the data gives diminishing returns. Modifying an algorithm to improve compression by 1% may increase the run time by 10% and the complexity of the program by more than that. A good way out of this dilemma was taken by Fiala and Greene (Section 6.10). After developing their main algorithms A1 and A2, they modified them to produce less compression at a higher speed, resulting in algorithms B1 and B2. They then modified A1 and A2 again, but in the opposite direction, sacrificing speed to get slightly better compression.

The principle of compressing by removing redundancy also answers the following question: Why is it that an already compressed file cannot be compressed further? The

answer, of course, is that such a file has little or no redundancy, so there is nothing to remove. An example of such a file is random text. In such text, each letter occurs with equal probability, so assigning them fixed-size codes does not add any redundancy. When such a file is compressed, there is no redundancy to remove. (Another answer is that if it were possible to compress an already compressed file, then successive compressions would reduce the size of the file until it becomes a single byte, or even a single bit. This, of course, is ridiculous since a single byte cannot contain the information present in an arbitrarily large file.)

In spite of the arguments above and the proof below, claims of recursive compression appear from time to time in the Internet. These are either checked and proved wrong or disappear silently. Reference [Barf 08], is a joke intended to amuse (and temporarily confuse) readers. A careful examination of this "claim" shows that any gain achieved by recursive compression of the Barf software is offset (perhaps more than offset) by the long name of the output file generated. The reader should also consult page 1132 for an interesting twist on the topic of compressing random data.

Definition of barf (verb): to vomit; purge; cast; sick; chuck; honk; throw up.

Since random data has been mentioned, let's say a few more words about it. Normally, it is rare to have a file with random data, but there is at least one good example—an already compressed file. Someone owning a compressed file normally knows that it is already compressed and would not attempt to compress it further, but there may be exceptions and one of them is data transmission by modems. Modern modems include hardware to automatically compress the data they send, and if that data is already compressed, there will not be further compression. There may even be expansion. This is why a modem should monitor the compression ratio "on the fly," and if it is low, it should stop compressing and should send the rest of the data uncompressed. The V.42bis protocol (Section 6.23) is a good example of this technique.

Before we prove the impossibility of recursive compression, here is an interesting twist on this concept. Several algorithms, such as JPEG, LZW, and MPEG, have long become de facto standards and are commonly used in web sites and in our computers. The field of data compression, however, is rapidly advancing and new, sophisticated methods are continually being developed. Thus, it is possible to take a compressed file, say JPEG, decompress it, and recompress it with a more efficient method. On the outside, this would look like recursive compression and may become a marketing tool for new, commercial compression software. The Stuffit software for the Macintosh platform (Section 11.16) does just that. It promises to compress already-compressed files and in many cases, it does!

The following simple argument illustrates the essence of the statement "Data compression is achieved by reducing or removing redundancy in the data." The argument shows that most data files cannot be compressed, no matter what compression method is used. This seems strange at first because we compress our data files all the time. The point is that most files cannot be compressed because they are random or close to random and therefore have no redundancy. The (relatively) few files that can be compressed are the ones that we *want* to compress; they are the files we use all the time. They have redundancy, are nonrandom, and are therefore useful and interesting.

Here is the argument. Given two different files A and B that are compressed to files C and D, respectively, it is clear that C and D must be different. If they were identical, there would be no way to decompress them and get back file A or file B.

Suppose that a file of size n bits is given and we want to compress it efficiently. Any compression method that can compress this file to, say, 10 bits would be welcome. Even compressing it to 11 bits or 12 bits would be great. We therefore (somewhat arbitrarily) assume that compressing such a file to half its size or better is considered good compression. There are 2^n n-bit files and they would have to be compressed into 2^n different files of sizes less than or equal to $n/2$. However, the total number of these files is

$$N = 1 + 2 + 4 + \cdots + 2^{n/2} = 2^{1+n/2} - 1 \approx 2^{1+n/2},$$

so only N of the 2^n original files have a chance of being compressed efficiently. The problem is that N is much smaller than 2^n. Here are two examples of the ratio between these two numbers.

For $n = 100$ (files with just 100 bits), the total number of files is 2^{100} and the number of files that can be compressed efficiently is 2^{51}. The ratio of these numbers is the ridiculously small fraction $2^{-49} \approx 1.78\times10^{-15}$.

For $n = 1000$ (files with just 1000 bits, about 125 bytes), the total number of files is 2^{1000} and the number of files that can be compressed efficiently is 2^{501}. The ratio of these numbers is the incredibly small fraction $2^{-499} \approx 9.82\times10^{-91}$.

Most files of interest are at least some thousands of bytes long. For such files, the percentage of files that can be efficiently compressed is so small that it cannot be computed with floating-point numbers even on a supercomputer (the result comes out as zero).

The 50% figure used here is arbitrary, but even increasing it to 90% isn't going to make a significant difference. Here is why. Assuming that a file of n bits is given and that $0.9n$ is an integer, the number of files of sizes up to $0.9n$ is

$$2^0 + 2^1 + \cdots + 2^{0.9n} = 2^{1+0.9n} - 1 \approx 2^{1+0.9n}.$$

For $n = 100$, there are 2^{100} files and $2^{1+90} = 2^{91}$ of them can be compressed well. The ratio of these numbers is $2^{91}/2^{100} = 2^{-9} \approx 0.00195$. For $n = 1000$, the corresponding fraction is $2^{901}/2^{1000} = 2^{-99} \approx 1.578\times10^{-30}$. These are still extremely small fractions.

It is therefore clear that no compression method can hope to compress all files or even a significant percentage of them. In order to compress a data file, the compression algorithm has to examine the data, find redundancies in it, and try to remove them. The redundancies in data depend on the type of data (text, images, audio, etc.), which is why a new compression method has to be developed for each specific type of data and it performs best on this type. There is no such thing as a universal, efficient data compression algorithm.

Data compression has become so important that some researchers (see, for example, [Wolff 99]) have proposed the SP theory (for "simplicity" and "power"), which suggests that all computing is compression! Specifically, it says: Data compression may be interpreted as a process of removing unnecessary complexity (redundancy) in information, and thereby maximizing simplicity while preserving as much as possible of its nonredundant descriptive power. SP theory is based on the following conjectures:

- All kinds of computing and formal reasoning may usefully be understood as information compression by pattern matching, unification, and search.

- The process of finding redundancy and removing it may always be understood at a fundamental level as a process of searching for patterns that match each other, and merging or unifying repeated instances of any pattern to make one.

This book discusses many compression methods, some suitable for text and others for graphical data (still images or video) or for audio. Most methods are classified into four categories: run length encoding (RLE), statistical methods, dictionary-based (sometimes called LZ) methods, and transforms. Chapters 1 and 11 describe methods based on other principles.

Before delving into the details, we discuss important data compression terms.

- A *compressor* or *encoder* is a program that compresses the raw data in the input stream and creates an output stream with compressed (low-redundancy) data. A *decompressor* or *decoder* converts in the opposite direction. Note that the term *encoding* is very general and has several meanings, but since we discuss only data compression, we use the name *encoder* to mean data compressor. The term *codec* is often used to describe both the encoder and the decoder. Similarly, the term *companding* is short for "compressing/expanding."

- The term *stream* is used throughout this book instead of *file*. *Stream* is a more general term because the compressed data may be transmitted directly to the decoder, instead of being written to a file and saved. Also, the data to be compressed may be downloaded from a network instead of being input from a file.

- For the original input stream, we use the terms *unencoded*, *raw*, or *original* data. The contents of the final, compressed, stream are considered the *encoded* or *compressed* data. The term *bitstream* is also used in the literature to indicate the compressed stream.

The Gold Bug

Here, then, we have, in the very beginning, the groundwork for something more than a mere guess. The general use which may be made of the table is obvious—but, in this particular cipher, we shall only very partially require its aid. As our predominant character is 8, we will commence by assuming it as the "e" of the natural alphabet. To verify the supposition, let us observe if the 8 be seen often in couples—for "e" is doubled with great frequency in English—in such words, for example, as "meet," "fleet," "speed," "seen," "been," "agree," etc. In the present instance we see it doubled no less than five times, although the cryptograph is brief.

—Edgar Allan Poe

- A *nonadaptive* compression method is rigid and does not modify its operations, its parameters, or its tables in response to the particular data being compressed. Such a method is best used to compress data that is all of a single type. Examples are

the Group 3 and Group 4 methods for facsimile compression (Section 5.7). They are specifically designed for facsimile compression and would do a poor job compressing any other data. In contrast, an *adaptive* method examines the raw data and modifies its operations and/or its parameters accordingly. An example is the adaptive Huffman method of Section 5.3. Some compression methods use a 2-pass algorithm, where the first pass reads the input stream to collect statistics on the data to be compressed, and the second pass does the actual compressing using parameters determined by the first pass. Such a method may be called *semiadaptive*. A data compression method can also be *locally adaptive*, meaning it adapts itself to local conditions in the input stream and varies this adaptation as it moves from area to area in the input. An example is the move-to-front method (Section 1.5).

- *Lossy/lossless compression:* Certain compression methods are lossy. They achieve better compression at the price of losing some information. When the compressed stream is decompressed, the result is not identical to the original data stream. Such a method makes sense especially in compressing images, video, or audio. If the loss of data is small, we may not be able to tell the difference. In contrast, text files, especially files containing computer programs, may become worthless if even one bit gets modified. Such files should be compressed only by a lossless compression method. [Two points should be mentioned regarding text files: (1) If a text file contains the source code of a program, consecutive blank spaces can often be replaced by a single space. (2) When the output of a word processor is saved in a text file, the file may contain information about the different fonts used in the text. Such information may be discarded if the user is interested in saving just the text.]

- *Cascaded compression:* The difference between lossless and lossy codecs can be illuminated by considering a cascade of compressions. Imagine a data file A that has been compressed by an encoder X, resulting in a compressed file B. It is possible, although pointless, to pass B through another encoder Y, to produce a third compressed file C. The point is that if methods X and Y are lossless, then decoding C by Y will produce an exact B, which when decoded by X will yield the original file A. However, if any of the compression algorithms is lossy, then decoding C by Y may produce a file B' different from B. Passing B' through X may produce something very different from A and may also result in an error, because X may not be able to read B'.

- *Perceptive compression:* A lossy encoder must take advantage of the special type of data being compressed. It should delete only data whose absence would not be detected by our senses. Such an encoder must therefore employ algorithms based on our understanding of psychoacoustic and psychovisual perception, so it is often referred to as a perceptive encoder. Such an encoder can be made to operate at a constant compression ratio, where for each x bits of raw data, it outputs y bits of compressed data. This is convenient in cases where the compressed stream has to be transmitted at a constant rate. The trade-off is a variable subjective quality. Parts of the original data that are difficult to compress may, after decompression, look (or sound) bad. Such parts may require more than y bits of output for x bits of input.

- *Symmetrical compression* is the case where the compressor and decompressor employ basically the same algorithm but work in "opposite" directions. Such a method

makes sense for general work, where the same number of files are compressed as are decompressed. In an asymmetric compression method, either the compressor or the decompressor may have to work significantly harder. Such methods have their uses and are not necessarily bad. A compression method where the compressor executes a slow, complex algorithm and the decompressor is simple is a natural choice when files are compressed into an archive, where they will be decompressed and used very often. The opposite case is useful in environments where files are updated all the time and backups are made. There is a small chance that a backup file will be used, so the decompressor isn't used very often.

> Like the ski resort full of girls hunting for husbands and husbands hunting for girls, the situation is not as symmetrical as it might seem.
>
> —Alan Lindsay Mackay, lecture, Birckbeck College, 1964

◇ **Exercise Intro.2:** Give an example of a compressed file where good compression is important but the speed of both compressor and decompressor isn't important.

▪ Many modern compression methods are asymmetric. Often, the formal description (the standard) of such a method specifies the decoder and the format of the compressed stream, but does not discuss the operation of the encoder. Any encoder that generates a correct compressed stream is considered *compliant*, as is also any decoder that can read and decode such a stream. The advantage of such a description is that anyone is free to develop and implement new, sophisticated algorithms for the encoder. The implementor need not even publish the details of the encoder and may consider it proprietary. If a compliant encoder is demonstrably better than competing encoders, it may become a commercial success. In such a scheme, the encoder is considered *algorithmic*, while the decoder, which is normally much simpler, is termed *deterministic*. A good example of this approach is the MPEG-1 audio compression method (Section 10.14).

▪ A data compression method is called *universal* if the compressor and decompressor do not know the statistics of the input stream. A universal method is *optimal* if the compressor can produce compression ratios that asymptotically approach the entropy of the input stream for long inputs.

▪ The term *file differencing* refers to any method that locates and compresses the differences between two files. Imagine a file A with two copies that are kept by two users. When a copy is updated by one user, it should be sent to the other user, to keep the two copies identical. Instead of sending a copy of A, which may be big, a much smaller file containing just the differences, in compressed format, can be sent and used at the receiving end to update the copy of A. Section 11.14.2 discusses some of the details and shows why compression can be considered a special case of file differencing. Note that the term *differencing* is used in Section 1.3.1 to describe an entirely different compression method.

▪ Most compression methods operate in the *streaming mode*, where the codec inputs a byte or several bytes, processes them, and continues until an end-of-file is sensed. Some methods, such as Burrows-Wheeler transform (Section 11.1), work in the *block mode*, where the input stream is read block by block and each block is encoded separately. The

block size should be a user-controlled parameter, since its size may significantly affect the performance of the method.

- Most compression methods are *physical*. They look only at the bits in the input stream and ignore the meaning of the data items in the input (e.g., the data items may be words, pixels, or audio samples). Such a method translates one bitstream into another, shorter bitstream. The only way to make sense of the output stream (to decode it) is by knowing how it was encoded. Some compression methods are *logical*. They look at individual data items in the source stream and replace common items with short codes. A logical method is normally special purpose and can operate successfully on certain types of data only. The pattern substitution method described on page 35 is an example of a logical compression method.

- *Compression performance*: Several measures are commonly used to express the performance of a compression method.

1. The *compression ratio* is defined as

$$\text{Compression ratio} = \frac{\text{size of the output stream}}{\text{size of the input stream}}.$$

A value of 0.6 means that the data occupies 60% of its original size after compression. Values greater than 1 imply an output stream bigger than the input stream (negative compression). The compression ratio can also be called bpb (bit per bit), since it equals the number of bits in the compressed stream needed, on average, to compress one bit in the input stream. In modern, efficient text compression methods, it makes sense to talk about bpc (bits per character)—the number of bits it takes, on average, to compress one character in the input stream.

Two more terms should be mentioned in connection with the compression ratio. The term *bitrate* (or "bit rate") is a general term for bpb and bpc. Thus, the main goal of data compression is to represent any given data at low bit rates. The term *bit budget* refers to the functions of the individual bits in the compressed stream. Imagine a compressed stream where 90% of the bits are variable-size codes of certain symbols, and the remaining 10% are used to encode certain tables. The bit budget for the tables is 10%.

2. The inverse of the compression ratio is called the *compression factor*:

$$\text{Compression factor} = \frac{\text{size of the input stream}}{\text{size of the output stream}}.$$

In this case, values greater than 1 indicate compression and values less than 1 imply expansion. This measure seems natural to many people, since the bigger the factor, the better the compression. This measure is distantly related to the sparseness ratio, a performance measure discussed in Section 8.6.2.

3. The expression $100 \times (1 - \text{compression ratio})$ is also a reasonable measure of compression performance. A value of 60 means that the output stream occupies 40% of its original size (or that the compression has resulted in savings of 60%).

4. In image compression, the quantity bpp (bits per pixel) is commonly used. It equals the number of bits needed, on average, to compress one pixel of the image. This quantity should always be compared with the bpp before compression.
5. The *compression gain* is defined as

$$100 \log_e \frac{\text{reference size}}{\text{compressed size}},$$

where the reference size is either the size of the input stream or the size of the compressed stream produced by some standard lossless compression method. For small numbers x, it is true that $\log_e(1+x) \approx x$, so a small change in a small compression gain is very similar to the same change in the compression ratio. Because of the use of the logarithm, two compression gains can be compared simply by subtracting them. The unit of the compression gain is called *percent log ratio* and is denoted by $\frac{\%}{\%}$.
6. The speed of compression can be measured in cycles per byte (CPB). This is the average number of machine cycles it takes to compress one byte. This measure is important when compression is done by special hardware.
7. Other quantities, such as mean square error (MSE) and peak signal to noise ratio (PSNR), are used to measure the distortion caused by lossy compression of images and movies. Section 7.4.2 provides information on those.
8. Relative compression is used to measure the compression gain in lossless audio compression methods, such as MLP (Section 10.7). This expresses the quality of compression by the number of bits each audio sample is reduced.

Name	Size	Description	Type
bib	111,261	A bibliography in UNIX *refer* format	Text
book1	768,771	Text of T. Hardy's *Far From the Madding Crowd*	Text
book2	610,856	Ian Witten's *Principles of Computer Speech*	Text
geo	102,400	Geological seismic data	Data
news	377,109	A Usenet news file	Text
obj1	21,504	VAX object program	Obj
obj2	246,814	Macintosh object code	Obj
paper1	53,161	A technical paper in *troff* format	Text
paper2	82,199	Same	Text
pic	513,216	Fax image (a bitmap)	Image
progc	39,611	A source program in C	Source
progl	71,646	A source program in LISP	Source
progp	49,379	A source program in Pascal	Source
trans	93,695	Document teaching how to use a terminal	Text

Table Intro.4: The Calgary Corpus.

■ The *Calgary Corpus* is a set of 18 files traditionally used to test data compression algorithms and implementations. They include text, image, and object files, for a total

of more than 3.2 million bytes (Table Intro.4). The corpus can be downloaded from [Calgary 06].

■ The *Canterbury Corpus* (Table Intro.5) is another collection of files introduced in 1997 to provide an alternative to the Calgary corpus for evaluating lossless compression methods. The following concerns led to the new corpus:

1. The Calgary corpus has been used by many researchers to develop, test, and compare many compression methods, and there is a chance that new methods would unintentionally be fine-tuned to that corpus. They may do well on the Calgary corpus documents but poorly on other documents.
2. The Calgary corpus was collected in 1987 and is getting old. "Typical" documents change over a period of decades (e.g., html documents did not exist in 1987), and any body of documents used for evaluation purposes should be examined from time to time.
3. The Calgary corpus is more or less an arbitrary collection of documents, whereas a good corpus for algorithm evaluation should be selected carefully.

The Canterbury corpus started with about 800 candidate documents, all in the public domain. They were divided into 11 classes, representing different types of documents. A representative "average" document was selected from each class by compressing every file in the class using different methods and selecting the file whose compression was closest to the average (as determined by statistical regression). The corpus is summarized in Table Intro.5 and can be obtained from [Canterbury 06].

Description	File name	Size (bytes)
English text (*Alice in Wonderland*)	alice29.txt	152,089
Fax images	ptt5	513,216
C source code	fields.c	11,150
Spreadsheet files	kennedy.xls	1,029,744
SPARC executables	sum	38,666
Technical document	lcet10.txt	426,754
English poetry ("Paradise Lost")	plrabn12.txt	481,861
HTML document	cp.html	24,603
LISP source code	grammar.lsp	3,721
GNU manual pages	xargs.1	4,227
English play (*As You Like It*)	asyoulik.txt	125,179
Complete genome of the *E. coli* bacterium	E.Coli	4,638,690
The King James version of the Bible	bible.txt	4,047,392
The CIA World Fact Book	world192.txt	2,473,400

Table Intro.5: The Canterbury Corpus.

The last three files constitute the beginning of a random collection of larger files. More files are likely to be added to it.

■ The Calgary challenge [Calgary challenge 08], is a contest to compress the Calgary corpus. It was started in 1996 by Leonid Broukhis and initially attracted a number

of contestants. In 2005, Alexander Ratushnyak achieved the current record of 596,314 bytes, using a variant of PAsQDa with a tiny dictionary of about 200 words.

Currently (late 2008), this challenge seems to have been preempted by the bigger prizes offered by the Hutter prize, and has featured no activity since 2005.

Here is part of the original challenge as it appeared in [Calgary challenge 08].

> I, Leonid A. Broukhis, will pay the amount of $(759,881.00 - X)/333$ US dollars (but not exceeding \$1001, and no less than \$10.01 "ten dollars and one cent") to the first person who sends me an archive of length X bytes, containing an executable and possibly other files, where the said executable file, run repeatedly with arguments being the names of other files contained in the original archive file one at a time (or without arguments if no other files are present) on a computer with no permanent storage or communication devices accessible to the running process(es) produces 14 new files, so that a 1-to-1 relationship of bytewise identity may be established between those new files and the files in the original Calgary corpus. (In other words, "solid" mode, as well as shared dictionaries/models and other tune-ups specific for the Calgary Corpus are allowed.)
>
> I will also pay the amount of $(777,777.00 - Y)/333$ US dollars (but not exceeding \$1001, and no less than \$0.01 "zero dollars and one cent") to the first person who sends me an archive of length Y bytes, containing an executable and exactly 14 files, where the said executable file, run with standard input taken directly (so that the `stdin` is seekable) from one of the 14 other files and the standard output directed to a new file, writes data to standard output so that the data being output matches one of the files in the original Calgary corpus and a 1-to-1 relationship may be established between the files being given as standard input and the files in the original Calgary corpus that the standard output matches. Moreover, after verifying the above requirements, an arbitrary file of size between 500 KB and 1 MB will be sent to the author of the decompressor to be compressed and sent back. The decompressor must handle that file correctly, and the compression ratio achieved on that file must be not worse that within 10% of the ratio achieved by gzip with default settings. (In other words, the compressor must be, roughly speaking, "general purpose.")

■ The *probability model*. This concept is important in statistical data compression methods. In such a method, a model for the data has to be constructed before compression can begin. A typical model may be built by reading the entire input stream, counting the number of times each symbol appears (its frequency of occurrence), and computing the probability of occurrence of each symbol. The data stream is then input again, symbol by symbol, and is compressed using the information in the probability model. A typical model is shown in Table 5.42, page 266.

Reading the entire input stream twice is slow, which is why practical compression methods use estimates, or adapt themselves to the data as it is being input and compressed. It is easy to scan large quantities of, say, English text and calculate the frequencies and probabilities of every character. This information can later serve as an approximate model for English text and can be used by text compression methods to compress any English text. It is also possible to start by assigning equal probabilities to

all the symbols in an alphabet, then reading symbols and compressing them, and, while doing that, also counting frequencies and changing the model as compression progresses. This is the principle behind the various *adaptive compression methods*.

- Source. A source of data items can be a file stored on a disk, a file that is input from outside the computer, text input from a keyboard, or a program that generates data symbols to be compressed or processed in some way. In a memoryless source, the probability of occurrence of a data symbol does not depend on its context. The term i.i.d. (independent and identically distributed) refers to a set of sources that have the same probability distribution and are mutually independent.

- Alphabet. This is the set of symbols that an application has to deal with. An alphabet may consist of the 128 ASCII codes, the 256 8-bit bytes, the two bits, or any other set of symbols.

- Random variable. This is a function that maps the results of random experiments to numbers. For example, selecting many people and measuring their heights is a random variable. The number of occurrences of each height can be used to compute the probability of that height, so we can talk about the probability distribution of the random variable (the set of probabilities of the heights). A special important case is a discrete random variable. The set of all values that such a variable can assume is finite or countably infinite.

- Compressed stream (or encoded stream). A compressor (or encoder) compresses data and generates a compressed stream. This is often a file that is written on a disk or is stored in memory. Sometimes, however, the compressed stream is a string of bits that are transmitted over a communications line.

[End of data compression terms.]

The concept of *data reliability and integrity* (page 247) is in some sense the opposite of data compression. Nevertheless, the two concepts are often related since any good data compression program should generate reliable code and so should be able to use error-detecting and error-correcting codes.

Compression benchmarks

Research in data compression, as in many other areas of computer science, concentrates on finding new algorithms and improving existing ones. In order to prove its value, however, an algorithm has to be implemented and tested. Thus, every researcher, programmer, and developer compares a new algorithm to older, well-established and known methods, and draws conclusions about its performance.

In addition to these tests, workers in the field of compression continually conduct extensive benchmarks, where many algorithms are run on the same set of data files and the results are compared and analyzed. This short section describes a few independent compression benchmarks.

Perhaps the most-important fact about these benchmarks is that they generally restrict themselves to compression ratios. Thus, a winner in such a benchmark may not be the best choice for general, everyday use, because it may be slow, may require large memory space, and may be expensive or protected by patents. Benchmarks for

compression speed are rare, because it is difficult to accurately measure the run time of an executable program (i.e., a program whose source code is unavailable). Another drawback of benchmarking is that their data files are generally publicly known, so anyone interested in record breaking for its own sake may tune an existing algorithm to the particular data files used by a benchmark and in this way achieve the smallest (but nevertheless meaningless) compression ratio.

From the Dictionary

Benchmark: A standard by which something can be measured or judged; "his painting sets the benchmark of quality."

In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.

The term benchmark originates from the chiseled horizontal marks that surveyors made in stone structures, into which an angle-iron could be placed to form a "bench" for a leveling rod, thus ensuring that a leveling rod could be accurately repositioned in the same place in future.

The following independent benchmarks compare the performance (compression ratios but generally not speeds, which are difficult to measure) of many algorithms and their implementations. Surprisingly, the results often indicate that the winner comes from the family of context-mixing compression algorithms. Such an algorithm employs several models of the data to predict the next data symbol, and then combines the predictions in some way to end up with a probability for the next symbol. The symbol and its computed probability are then sent to an adaptive arithmetic coder, to be encoded. Included in this family of lossless algorithms are the many versions and derivatives of PAQ (Section 5.15) as well as many other, less well-known methods.

- The Maximum Compression Benchmark, managed by Werner Bergmans. This suite of tests (described in [Bergmans 08]) was started in 2003 and is still frequently updated. The goal is to discover the best compression ratios for several different types of data, such as text, images, and executable code. Every algorithm included in the tests is first tuned by setting any switches and parameters to the values that yield best performance. The owner of this benchmark prefers command line (console) compression programs over GUI ones. At the time of writing (late 2008), more than 150 programs have been tested on several large collections of test files. Most of the top-ranked algorithms are of the context mixing type. Special mention goes to PAsQDa 4.1b and WinRK 2.0.6/pwcm. The latest update to this benchmark reads as follows:

  > 28-September-2008: Added PAQ8P, 7-Zip 4.60b, FreeARC 0.50a (June 23 2008), Tornado 0.4a, M1 0.1a, BZP 0.3, NanoZIP 0.04a, Blizzard 0.24b and WinRAR 3.80b5 (MFC to do for WinRAR and 7-Zip). PAQ8P manages to squeeze out an additional 12 KB from the BMP file, further increasing the gap to the number 2 in the SFC benchmark; newcomer NanoZIP takes 6th place in SFC!. In the MFC benchmark PAQ8 now takes a huge lead over WinRK 3.0.3, but WinRK 3.1.2 is on the todo list to be tested. To be continued....

- Johan de Bock started the UCLC (ultimate command-line compressors) benchmark

project [UCLC 08]. A wide variety of tests are performed to compare the latest state-of-the-art command line compressors. The only feature being compared is the compression ratio; run-time and memory requirements are ignored. More than 100 programs have been tested over the years, with WinRK and various versions of PAQ declared the best performers (except for audio and grayscale images, where the records were achieved by specialized algorithms).

- The EmilCont benchmark [Emilcont 08] is managed by Berto Destasio. At the time of writing, the latest update of this site dates back to March 2007. EmilCont tests hundreds of algorithms on a confidential set of data files that include text, images, audio, and executable code. As usual, WinRK and PAQ variants are among the record holders, followed by SLIM.

> An important feature of these benchmarks is that the test data will not be released to avoid the unfortunate practice of compression writers tuning their programs to the benchmarks.
>
> —From [Emilcont 08]

- The Archive Comparison Test (ACT), maintained by Jeff Gilchrist [Gilchrist 08], is a set of benchmarks designed to demonstrate the state of the art in lossless data compression. It contains benchmarks on various types of data for compression speed, decompression speed, and compression ratio.

  The site lists the results of comparing 162 DOS/Windows programs, eight Macintosh programs, and 11 JPEG compressors. However, the tests are old. Many were performed in or before 2002 (except the JPEG tests, which took place in 2007).

- In [Ratushnyak 08], Alexander Ratushnyak reports the results of hundreds of speed tests done in 2001.

- Site [squeezemark 09] is devoted to benchmarks of lossless codecs. It is kept up to date by its owner, Stephan Busch, and has many pictures of compression pioneers.

How to Hide Data

Here is an unforeseen, unexpected application of data compression. For a long time I have been thinking about how to best hide a sensitive data file in a computer, while still having it ready for use at a short notice. Here is what we came up with. Given a data file A, consider the following steps:

1. Compress A. The result is a file B that is small and also seems random. This has two advantages (1) the remaining steps encrypt and hide small files and (2) the next step encrypts a random file, thereby making it difficult to break the encryption simply by checking every key.

2. Encrypt B with a secret key to obtain file C. A would-be codebreaker may attempt to decrypt C by writing a program that loops and tries every key, but here is the rub. Each time a key is tried, someone (or something) has to check the result. If the result looks meaningful, it may be the decrypted file B, but if the result seems random, the loop should continue. At the end of the loop; frustration.

3. Hide C inside a cover file D to obtain a large file E. Use one of the many steganographic methods for this (notice that many such methods depend on secret keys). One reference for steganography is [Salomon 03], but today there may be better texts.

4. Hide E in plain sight in your computer by changing its name and placing it in a large folder together with hundreds of other, unfamiliar files. A good idea may be to change the file name to `msLibPort.dll` (or something similar that includes MS and other familiar-looking terms) and place it in one of the many large folders created and used exclusively by Windows or any other operating system. If files in this folder are visible, do not make your file invisible. Anyone looking inside this folder will see hundreds of unfamiliar files and will have no reason to suspect `msLibPort.dll`. Even if this happens, an opponent would have a hard time guessing the three steps above (unless he has read these paragraphs) and the keys used. If file E is large (perhaps more than a few Gbytes), it should be segmented into several smaller files and each hidden in plain sight as described above. This step is important because there are utilities that identify large files and they may attract unwanted attention to your large E.

For those who require even greater privacy, here are a few more ideas. (1) A password can be made strong by including in it special characters such §, ¶, †, and ‡. These can be typed with the help of special modifier keys found on most keyboards. (2) Add a step between steps 1 and 2 where file B is recompressed by any compression method. This will not decrease the size of B but will defeat anyone trying to decompress B into meaningful data simply by trying many decompression algorithms. (3) Add a step between steps 1 and 2 where file B is partitioned into segments and random data inserted between the segments. (4) Instead of inserting random data segments, swap segments to create a permutation of the segments. The permutation may be determined by the password used in step 2.

> Until now, the US government's default position has been: If you can't keep data secret, at least hide it on one of 24,000 federal Websites, preferably in an incompatible or obsolete format.
>
> —*Wired*, July 2009.

Compression Curiosities

People are curious and they also like curiosities (not the same thing). It is easy to find curious facts, behavior, and objects in many areas of science and technology. In the early days of computing, for example, programmers were interested in programs that print themselves. Imagine a program in a given programming language. The source code of the program is printed on paper and can easily be read. When the program is executed, it prints its own source code. Curious! The few compression curiosities that appear here are from [curiosities 08].

■ We often hear the following phrase "I want it all and I want it now!" When it comes to compressing data, we all want the best compressor. So, how much can a file possibly be compressed? Lossless methods routinely compress files to less than half their size and can go down to compression ratios of about 1/8 or smaller. Lossy algorithms do much better. Is it possible to compress a file by a factor of 10,000? Now that would be a curiosity.

The Iterated Function Systems method (IFS, Section 7.39) can compress certain files by factors of thousands. However, such files must be especially prepared (see four examples in Section 7.39.3). Given an arbitrary data file, there is no guarantee that IFS will compress it well.

The 115-byte RAR-compressed file (see Section 6.22 for RAR) found at [curio.test 08] swells, when decompressed, to 4,884,863 bytes, a compression factor of 42,477! The hexadecimal listing of this (obviously contrived) file is

```
526172211A07003BD07308000D00000000000000CB5C74C0802800300000007F894A0002F4EE5
A3DA39B29351D3508002000000074657374 2E747874A1182FD22D77B8617EF782D70000000000
00000000000000000000000000000093DE30369CFB76A800BF8867F6A9FFD4C43D7B00400700
```

■ Even if we do not obtain good compression, we certainly don't expect a compression program to *increase* the size of a file, but this is precisely what happens sometimes (even often). Expansion is the bane of compression and it is easy to choose the wrong data file or the wrong compressor for a given file and so end up with an expanded file (and high blood pressure into the bargain). If we try to compress a text file with a program designed specifically for compressing images or audio, often the result is expansion, even considerable expansion. Trying to compress random data very often results in expansion.

> The Ring was cut from Sauron's hand by Isildur at the slopes of Mount Doom, and he in turn lost it in the River Anduin just before he was killed in an Orc ambush. Since it indirectly caused Isildur's death by slipping from his finger, it was known in Gondorian lore as Isildur's Bane.
>
> —From Wikipedia

■ Another contrived file is the 99,058-byte `strange.rar`, located at [curio.strange 08]. When decompressed with RAR, the resulting file, `Compression Test.gba`, is 1,048,576-bytes long, indicating a healthy compression factor of 10.59. However, when `Compression Test.gba` is compressed in turn with zip, there is virtually no compression. Two high-performance compression algorithms yield very different results when processing the same data. Curious behavior!

■ Can a compression algorithm compress a file to itself? Obviously, the trivial algorithm that does nothing, achieves precisely that. Thus, we better ask, given a well-known, nontrivial compression algorithm, is it possible to find (or construct) a file that the algorithm will compress to itself? The surprising answer is yes. File `selfgz.gz` that can be found at `http://www.maximumcompression.com/selfgz.gz` yields, when compressed by gzip, itself!

> "Curiouser and curiouser!" Cried Alice (she was so much surprised, that for the moment she quite forgot how to speak good English).
>
> —Lewis Carroll, *Alice's Adventures in Wonderland* (1865)

The Ten Commandments of Compression

1. Redundancy is your enemy, eliminate it.

2. Entropy is your goal, strive to achieve it.

3. Read the literature before you try to publish/implement your new, clever compression algorithm. Others may have been there before you.

4. There is no universal compression method that can compress any file to just a few bytes. Thus, refrain from making incredible claims. They will come back to haunt you.

5. The G-d of compression prefers free and open source codecs.

6. If you decide to patent your algorithm, make sure anyone can understand your patent application. Others might want to improve on it. Talking about patents, recall the following warning about them (from D. Knuth) "there are better ways to earn a living than to prevent other people from making use of one's contributions to computer science."

7. Have you discovered a new, universal, simple, fast, and efficient compression algorithm? Please don't ask others (especially these authors) to evaluate it for you for free.

8. The well-known saying (also from Knuth) "beware of bugs in the above code; I have only proved it correct, not tried it," applies also (perhaps even mostly) to compression methods. Implement and test your method thoroughly before you brag about it.

9. Don't try to optimize your algorithm/code to squeeze the last few bits out of the output in order to win a prize. Instead, identify the point where you start getting diminishing returns and stop there.

10. This is your own, private commandment. Grab a pencil, write it here, and obey it.

Why this book? Most drivers know little or nothing about the operation of the engine or transmission in their cars. Few know how cellular telephones, microwave ovens, or combination locks work. Why not let scientists develop and implement compression methods and have us use them without worrying about the details? The answer, naturally, is curiosity. Many drivers try to tinker with their car out of curiosity. Many weekend sailors love to mess about with boats even on weekdays, and many children spend hours taking apart a machine, a device, or a toy in an attempt to understand its operation. If you are curious about data compression, this book is for you.

The typical reader of this book should have a basic knowledge of computer science; should know something about programming and data structures; feel comfortable with terms such as *bit*, *mega*, *ASCII*, *file*, *I/O*, and *binary search*; and should be curious. The necessary mathematical background is minimal and is limited to logarithms, matrices, polynomials, differentiation/integration, and the concept of probability. This book is not intended to be a guide to software implementors and has few programs.

The following URLs have useful links and pointers to the many data compression resources available on the Internet and elsewhere:

`http://www.hn.is.uec.ac.jp/~arimura/compression_links.html`,

http://cise.edu.mie-u.ac.jp/~okumura/compression.html, http://compression-links.info/, http://compression.ca/ (mostly comparisons), http://datacompression.info/. This URL has a wealth of information on data compression, including tutorials, links, and lists of books. The site is owned by Mark Nelson. http://directory.google.com/Top/Computers/Algorithms/Compression/ is also a growing, up-to-date, site.

Reference [Okumura 98] discusses the history of data compression in Japan.

Data Compression Resources

A vast number of resources on data compression are available. Any Internet search under "data compression," "lossless data compression," "image compression," "audio compression," and similar topics returns at least tens of thousands of results. Traditional (printed) resources range from general texts and texts on specific aspects or particular methods, to survey articles in magazines, to technical reports and research papers in scientific journals. Following is a short list of (mostly general) books, sorted by date of publication.

Khalid Sayood, *Introduction to Data Compression*, Morgan Kaufmann, 3rd edition (2005).
Ida Mengyi Pu, *Fundamental Data Compression*, Butterworth-Heinemann (2005).
Darrel Hankerson, *Introduction to Information Theory and Data Compression*, Chapman & Hall (CRC), 2nd edition (2003).
Peter Symes, *Digital Video Compression*, McGraw-Hill/TAB Electronics (2003).
Charles Poynton, *Digital Video and HDTV Algorithms and Interfaces*, Morgan Kaufmann (2003).
Iain E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia*, John Wiley and Sons (2003).
Khalid Sayood, *Lossless Compression Handbook*, Academic Press (2002).
Touradj Ebrahimi and Fernando Pereira, *The MPEG-4 Book*, Prentice Hall (2002).
Adam Drozdek, *Elements of Data Compression*, Course Technology (2001).
David Taubman and Michael Marcellin, (eds), *JPEG2000: Image Compression Fundamentals, Standards and Practice*, Springer Verlag (2001).
Kamisetty R. Rao, *The Transform and Data Compression Handbook*, CRC (2000).
Ian H. Witten, Alistair Moffat, and Timothy C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann, 2nd edition (1999).
Peter Wayner, *Compression Algorithms for Real Programmers*, Morgan Kaufmann (1999).
John Miano, *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*, ACM Press and Addison-Wesley Professional (1999).
Mark Nelson and Jean-Loup Gailly, *The Data Compression Book*, M&T Books, 2nd edition (1995).
William B. Pennebaker and Joan L. Mitchell, *JPEG: Still Image Data Compression Standard*, Springer Verlag (1992).
Timothy C. Bell, John G. Cleary, and Ian H. Witten, *Text Compression*, Prentice Hall (1990).
James A. Storer, *Data Compression: Methods and Theory*, Computer Science Press.
John Woods, ed., *Subband Coding*, Kluwer Academic Press (1990).

Notation

- The symbol "␣" is used to indicate a blank space in places where spaces may lead to ambiguity.
- The acronyms MSB and LSB refer to most-significant-bit and least-significant-bit, respectively.
- The notation 1^i0^j indicates a bit string of i consecutive 1's followed by j zeros.

Some readers called into question the title of the predecessors of this book. What does it mean for a work of this kind to be complete, and how complete is this book? Here is our opinion on the matter. We like to believe that if the entire field of data compression were (heaven forbid) to disappear, a substantial part of it could be reconstructed from this work. Naturally, we don't compare ourselves to James Joyce, but his works provide us with a similar example. He liked to claim that if the Dublin of his time were to be destroyed, it could be reconstructed from his works.

Readers who would like to get an idea of the effort it took to write this book should consult the Colophon.

The authors welcome any comments, suggestions, and corrections. They should be sent to `dsalomon@csun.edu` or `gim@ieee.org`.

The days just prior to marriage are like a
snappy introduction to a tedious book.
—Wilson Mizner