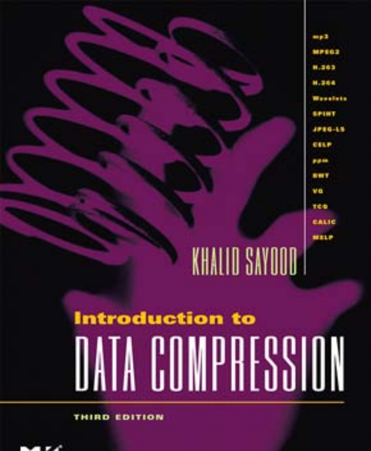


mp3
MPEG2
H.263
H.264
Wavelets
SPIHT
JPEG-LS
CELP
ppm
DWT
VQ
TCQ
CALIC
MELP

KHALID SAYOOD

Introduction to
DATA COMPRESSION

THIRD EDITION

THIRD EDITION

Introduction to Data Compression

The Morgan Kaufmann Series in Multimedia Information and Systems
Series Editor, Edward A. Fox, Virginia Polytechnic University

Introduction to Data Compression, Third Edition
Khalid Sayood

Understanding Digital Libraries, Second Edition
Michael Lesk

Bioinformatics: Managing Scientific Data
Zoe Lacroix and Terence Critchlow

How to Build a Digital Library
Ian H. Witten and David Bainbridge

Digital Watermarking
Ingemar J. Cox, Matthew L. Miller, and Jeffrey A. Bloom

Readings in Multimedia Computing and Networking
Edited by Kevin Jeffay and HongJiang Zhang

Introduction to Data Compression, Second Edition
Khalid Sayood

Multimedia Servers: Applications, Environments, and Design
Dinkar Sitaram and Asit Dan

Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition
Ian H. Witten, Alistair Moffat, and Timothy C. Bell

Digital Compression for Multimedia: Principles and Standards
Jerry D. Gibson, Toby Berger, Tom Lookabaugh, Dave Lindbergh, and Richard L. Baker

Readings in Information Retrieval
Edited by Karen Sparck Jones and Peter Willett

THIRD EDITION

Introduction to Data Compression

Khalid Sayood
University of Nebraska

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

MORGAN KAUFMANN

Senior Acquisitions Editor	Rick Adams
Publishing Services Manager	Simon Crump
Assistant Editor	Rachel Roumeliotis
Cover Design	Cate Barr
Composition	Integra Software Services Pvt. Ltd.
Copyeditor	Jessika Bella Mura
Proofreader	Jacqui Brownstein
Indexer	Northwind Editorial Sevices
Interior printer	Maple Vail Book Manufacturing Group
Cover printer	Phoenix Color

Morgan Kaufmann Publishers is an imprint of Elsevier.
500 Sansome Street, Suite 400, San Francisco, CA 94111

This book is printed on acid-free paper.

©2006 by Elsevier Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, scanning, or otherwise—without prior written permission of the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.co.uk. You may also complete your request on-line via the Elsevier homepage (http://elsevier.com) by selecting "Customer Support" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data
Sayood, Khalid.
Introduction to data compression / Khalid Sayood.—3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-12-620862-7
ISBN-10: 0-12-620862-X
1. Data compression (Telecommunication) 2. Coding theory. I. Title
TK5102.92.S39 2005
005.74′6—dc22
2005052759

ISBN 13: 978-0-12-620862-7
ISBN 10: 0-12-620862-X

For information on all Morgan Kaufmann publications,
visit our Web site at www.mkp.com or www.books.elsevier.com

Printed in the United States of America
05 06 07 08 09 5 4 3 2 1

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER BOOK AID International Sabre Foundation

To Füsun

Contents

Preface **xvii**

1 Introduction **1**
- 1.1 Compression Techniques 3
  - 1.1.1 Lossless Compression 4
  - 1.1.2 Lossy Compression 5
  - 1.1.3 Measures of Performance 5
- 1.2 Modeling and Coding 6
- 1.3 Summary 10
- 1.4 Projects and Problems 11

2 Mathematical Preliminaries for Lossless Compression **13**
- 2.1 Overview 13
- 2.2 A Brief Introduction to Information Theory 13
  - 2.2.1 Derivation of Average Information ★ 18
- 2.3 Models 23
  - 2.3.1 Physical Models 23
  - 2.3.2 Probability Models 23
  - 2.3.3 Markov Models 24
  - 2.3.4 Composite Source Model 27
- 2.4 Coding 27
  - 2.4.1 Uniquely Decodable Codes 28
  - 2.4.2 Prefix Codes 31
  - 2.4.3 The Kraft-McMillan Inequality ★ 32
- 2.5 Algorithmic Information Theory 35
- 2.6 Minimum Description Length Principle 36
- 2.7 Summary 37
- 2.8 Projects and Problems 38

3 Huffman Coding **41**
- 3.1 Overview 41
- 3.2 The Huffman Coding Algorithm 41
  - 3.2.1 Minimum Variance Huffman Codes 46
  - 3.2.2 Optimality of Huffman Codes ★ 48
  - 3.2.3 Length of Huffman Codes ★ 49
  - 3.2.4 Extended Huffman Codes ★ 51

3.3 Nonbinary Huffman Codes ★ 55
3.4 Adaptive Huffman Coding 58
  3.4.1 Update Procedure 59
  3.4.2 Encoding Procedure 62
  3.4.3 Decoding Procedure 63
3.5 Golomb Codes 65
3.6 Rice Codes 67
  3.6.1 CCSDS Recommendation for Lossless Compression 67
3.7 Tunstall Codes 69
3.8 Applications of Huffman Coding 72
  3.8.1 Lossless Image Compression 72
  3.8.2 Text Compression 74
  3.8.3 Audio Compression 75
3.9 Summary 77
3.10 Projects and Problems 77

4 Arithmetic Coding 81
4.1 Overview 81
4.2 Introduction 81
4.3 Coding a Sequence 83
  4.3.1 Generating a Tag 84
  4.3.2 Deciphering the Tag 91
4.4 Generating a Binary Code 92
  4.4.1 Uniqueness and Efficiency of the Arithmetic Code 93
  4.4.2 Algorithm Implementation 96
  4.4.3 Integer Implementation 102
4.5 Comparison of Huffman and Arithmetic Coding 109
4.6 Adaptive Arithmetic Coding 112
4.7 Applications 112
4.8 Summary 113
4.9 Projects and Problems 114

5 Dictionary Techniques 117
5.1 Overview 117
5.2 Introduction 117
5.3 Static Dictionary 118
  5.3.1 Digram Coding 119
5.4 Adaptive Dictionary 121
  5.4.1 The LZ77 Approach 121
  5.4.2 The LZ78 Approach 125
5.5 Applications 133
  5.5.1 File Compression—UNIX `compress` 133
  5.5.2 Image Compression—The Graphics Interchange Format (GIF) 133
  5.5.3 Image Compression—Portable Network Graphics (PNG) 134
  5.5.4 Compression over Modems—V.42 bis 136

5.6 Summary 138
5.7 Projects and Problems 139

6 Context-Based Compression 141
6.1 Overview 141
6.2 Introduction 141
6.3 Prediction with Partial Match (*ppm*) 143
  6.3.1 The Basic Algorithm 143
  6.3.2 The Escape Symbol 149
  6.3.3 Length of Context 150
  6.3.4 The Exclusion Principle 151
6.4 The Burrows-Wheeler Transform 152
  6.4.1 Move-to-Front Coding 156
6.5 Associative Coder of Buyanovsky (ACB) 157
6.6 Dynamic Markov Compression 158
6.7 Summary 160
6.8 Projects and Problems 161

7 Lossless Image Compression 163
7.1 Overview 163
7.2 Introduction 163
  7.2.1 The Old JPEG Standard 164
7.3 CALIC 166
7.4 JPEG-LS 170
7.5 Multiresolution Approaches 172
  7.5.1 Progressive Image Transmission 173
7.6 Facsimile Encoding 178
  7.6.1 Run-Length Coding 179
  7.6.2 CCITT Group 3 and 4—Recommendations T.4 and T.6 180
  7.6.3 JBIG 183
  7.6.4 JBIG2—T.88 189
7.7 MRC—T.44 190
7.8 Summary 193
7.9 Projects and Problems 193

8 Mathematical Preliminaries for Lossy Coding 195
8.1 Overview 195
8.2 Introduction 195
8.3 Distortion Criteria 197
  8.3.1 The Human Visual System 199
  8.3.2 Auditory Perception 200
8.4 Information Theory Revisited ★ 201
  8.4.1 Conditional Entropy 202
  8.4.2 Average Mutual Information 204
  8.4.3 Differential Entropy 205

8.5 Rate Distortion Theory ★ 208
8.6 Models 215
  8.6.1 Probability Models 216
  8.6.2 Linear System Models 218
  8.6.3 Physical Models 223
8.7 Summary 224
8.8 Projects and Problems 224

9 Scalar Quantization 227
9.1 Overview 227
9.2 Introduction 227
9.3 The Quantization Problem 228
9.4 Uniform Quantizer 233
9.5 Adaptive Quantization 244
  9.5.1 Forward Adaptive Quantization 244
  9.5.2 Backward Adaptive Quantization 246
9.6 Nonuniform Quantization 253
  9.6.1 *pdf*-Optimized Quantization 253
  9.6.2 Companded Quantization 257
9.7 Entropy-Coded Quantization 264
  9.7.1 Entropy Coding of Lloyd-Max Quantizer Outputs 265
  9.7.2 Entropy-Constrained Quantization ★ 265
  9.7.3 High-Rate Optimum Quantization ★ 266
9.8 Summary 269
9.9 Projects and Problems 270

10 Vector Quantization 273
10.1 Overview 273
10.2 Introduction 273
10.3 Advantages of Vector Quantization over Scalar Quantization 276
10.4 The Linde-Buzo-Gray Algorithm 282
  10.4.1 Initializing the LBG Algorithm 287
  10.4.2 The Empty Cell Problem 294
  10.4.3 Use of LBG for Image Compression 294
10.5 Tree-Structured Vector Quantizers 299
  10.5.1 Design of Tree-Structured Vector Quantizers 302
  10.5.2 Pruned Tree-Structured Vector Quantizers 303
10.6 Structured Vector Quantizers 303
  10.6.1 Pyramid Vector Quantization 305
  10.6.2 Polar and Spherical Vector Quantizers 306
  10.6.3 Lattice Vector Quantizers 307
10.7 Variations on the Theme 311
  10.7.1 Gain-Shape Vector Quantization 311
  10.7.2 Mean-Removed Vector Quantization 312

10.7.3 Classified Vector Quantization 313
10.7.4 Multistage Vector Quantization 313
10.7.5 Adaptive Vector Quantization 315
10.8 Trellis-Coded Quantization 316
10.9 Summary 321
10.10 Projects and Problems 322

11 Differential Encoding 325
11.1 Overview 325
11.2 Introduction 325
11.3 The Basic Algorithm 328
11.4 Prediction in DPCM 332
11.5 Adaptive DPCM 337
11.5.1 Adaptive Quantization in DPCM 338
11.5.2 Adaptive Prediction in DPCM 339
11.6 Delta Modulation 342
11.6.1 Constant Factor Adaptive Delta Modulation (CFDM) 343
11.6.2 Continuously Variable Slope Delta Modulation 345
11.7 Speech Coding 345
11.7.1 G.726 347
11.8 Image Coding 349
11.9 Summary 351
11.10 Projects and Problems 352

12 Mathematical Preliminaries for Transforms, Subbands, and Wavelets 355
12.1 Overview 355
12.2 Introduction 355
12.3 Vector Spaces 356
12.3.1 Dot or Inner Product 357
12.3.2 Vector Space 357
12.3.3 Subspace 359
12.3.4 Basis 360
12.3.5 Inner Product—Formal Definition 361
12.3.6 Orthogonal and Orthonormal Sets 361
12.4 Fourier Series 362
12.5 Fourier Transform 365
12.5.1 Parseval's Theorem 366
12.5.2 Modulation Property 366
12.5.3 Convolution Theorem 367
12.6 Linear Systems 368
12.6.1 Time Invariance 368
12.6.2 Transfer Function 368
12.6.3 Impulse Response 369
12.6.4 Filter 371

12.7 Sampling 372
  12.7.1 Ideal Sampling—Frequency Domain View 373
  12.7.2 Ideal Sampling—Time Domain View 375
12.8 Discrete Fourier Transform 376
12.9 Z-Transform 378
  12.9.1 Tabular Method 381
  12.9.2 Partial Fraction Expansion 382
  12.9.3 Long Division 386
  12.9.4 Z-Transform Properties 387
  12.9.5 Discrete Convolution 387
12.10 Summary 389
12.11 Projects and Problems 390

13 Transform Coding 391
13.1 Overview 391
13.2 Introduction 391
13.3 The Transform 396
13.4 Transforms of Interest 400
  13.4.1 Karhunen-Loéve Transform 401
  13.4.2 Discrete Cosine Transform 402
  13.4.3 Discrete Sine Transform 404
  13.4.4 Discrete Walsh-Hadamard Transform 404
13.5 Quantization and Coding of Transform Coefficients 407
13.6 Application to Image Compression—JPEG 410
  13.6.1 The Transform 410
  13.6.2 Quantization 411
  13.6.3 Coding 413
13.7 Application to Audio Compression—the MDCT 416
13.8 Summary 419
13.9 Projects and Problems 421

14 Subband Coding 423
14.1 Overview 423
14.2 Introduction 423
14.3 Filters 428
  14.3.1 Some Filters Used in Subband Coding 432
14.4 The Basic Subband Coding Algorithm 436
  14.4.1 Analysis 436
  14.4.2 Quantization and Coding 437
  14.4.3 Synthesis 437
14.5 Design of Filter Banks ★ 438
  14.5.1 Downsampling ★ 440
  14.5.2 Upsampling ★ 443
14.6 Perfect Reconstruction Using Two-Channel Filter Banks ★ 444
  14.6.1 Two-Channel PR Quadrature Mirror Filters ★ 447
  14.6.2 Power Symmetric FIR Filters ★ 449

14.7 *M*-Band QMF Filter Banks ★ 451
14.8 The Polyphase Decomposition ★ 454
14.9 Bit Allocation 459
14.10 Application to Speech Coding—G.722 461
14.11 Application to Audio Coding—MPEG Audio 462
14.12 Application to Image Compression 463
  14.12.1 Decomposing an Image 465
  14.12.2 Coding the Subbands 467
14.13 Summary 470
14.14 Projects and Problems 471

15 Wavelet-Based Compression 473
15.1 Overview 473
15.2 Introduction 473
15.3 Wavelets 476
15.4 Multiresolution Analysis and the Scaling Function 480
15.5 Implementation Using Filters 486
  15.5.1 Scaling and Wavelet Coefficients 488
  15.5.2 Families of Wavelets 491
15.6 Image Compression 494
15.7 Embedded Zerotree Coder 497
15.8 Set Partitioning in Hierarchical Trees 505
15.9 JPEG 2000 512
15.10 Summary 513
15.11 Projects and Problems 513

16 Audio Coding 515
16.1 Overview 515
16.2 Introduction 515
  16.2.1 Spectral Masking 517
  16.2.2 Temporal Masking 517
  16.2.3 Psychoacoustic Model 518
16.3 MPEG Audio Coding 519
  16.3.1 Layer I Coding 520
  16.3.2 Layer II Coding 521
  16.3.3 Layer III Coding—*mp3* 522
16.4 MPEG Advanced Audio Coding 527
  16.4.1 MPEG-2 AAC 527
  16.4.2 MPEG-4 AAC 532
16.5 Dolby AC3 (Dolby Digital) 533
  16.5.1 Bit Allocation 534
16.6 Other Standards 535
16.7 Summary 536

17 Analysis/Synthesis and Analysis by Synthesis Schemes 537

17.1 Overview 537
17.2 Introduction 537
17.3 Speech Compression 539
  17.3.1 The Channel Vocoder 539
  17.3.2 The Linear Predictive Coder (Government Standard LPC-10) 542
  17.3.3 Code Excited Linear Predicton (CELP) 549
  17.3.4 Sinusoidal Coders 552
  17.3.5 Mixed Excitation Linear Prediction (MELP) 555
17.4 Wideband Speech Compression—ITU-T G.722.2 558
17.5 Image Compression 559
  17.5.1 Fractal Compression 560
17.6 Summary 568
17.7 Projects and Problems 569

18 Video Compression 571

18.1 Overview 571
18.2 Introduction 571
18.3 Motion Compensation 573
18.4 Video Signal Representation 576
18.5 ITU-T Recommendation H.261 582
  18.5.1 Motion Compensation 583
  18.5.2 The Loop Filter 584
  18.5.3 The Transform 586
  18.5.4 Quantization and Coding 586
  18.5.5 Rate Control 588
18.6 Model-Based Coding 588
18.7 Asymmetric Applications 590
18.8 The MPEG-1 Video Standard 591
18.9 The MPEG-2 Video Standard—H.262 594
  18.9.1 The Grand Alliance HDTV Proposal 597
18.10 ITU-T Recommendation H.263 598
  18.10.1 Unrestricted Motion Vector Mode 600
  18.10.2 Syntax-Based Arithmetic Coding Mode 600
  18.10.3 Advanced Prediction Mode 600
  18.10.4 PB-frames and Improved PB-frames Mode 600
  18.10.5 Advanced Intra Coding Mode 600
  18.10.6 Deblocking Filter Mode 601
  18.10.7 Reference Picture Selection Mode 601
  18.10.8 Temporal, SNR, and Spatial Scalability Mode 601
  18.10.9 Reference Picture Resampling 601
  18.10.10 Reduced-Resolution Update Mode 602
  18.10.11 Alternative Inter VLC Mode 602
  18.10.12 Modified Quantization Mode 602
  18.10.13 Enhanced Reference Picture Selection Mode 603

18.11 ITU-T Recommendation H.264, MPEG-4 Part 10, Advanced Video Coding 603
  18.11.1 Motion-Compensated Prediction 604
  18.11.2 The Transform 605
  18.11.3 Intra Prediction 605
  18.11.4 Quantization 606
  18.11.5 Coding 608
18.12 MPEG-4 Part 2 609
18.13 Packet Video 610
18.14 ATM Networks 610
  18.14.1 Compression Issues in ATM Networks 611
  18.14.2 Compression Algorithms for Packet Video 612
18.15 Summary 613
18.16 Projects and Problems 614

A Probability and Random Processes 615
A.1 Probability 615
  A.1.1 Frequency of Occurrence 615
  A.1.2 A Measure of Belief 616
  A.1.3 The Axiomatic Approach 618
A.2 Random Variables 620
A.3 Distribution Functions 621
A.4 Expectation 623
  A.4.1 Mean 624
  A.4.2 Second Moment 625
  A.4.3 Variance 625
A.5 Types of Distribution 625
  A.5.1 Uniform Distribution 625
  A.5.2 Gaussian Distribution 626
  A.5.3 Laplacian Distribution 626
  A.5.4 Gamma Distribution 626
A.6 Stochastic Process 626
A.7 Projects and Problems 629

B A Brief Review of Matrix Concepts 631
B.1 A Matrix 631
B.2 Matrix Operations 632

C The Root Lattices 637

Bibliography 639

Index 655

Preface

Within the last decade the use of data compression has become ubiquitous. From *mp3* players whose headphones seem to adorn the ears of most young (and some not so young) people, to cell phones, to DVDs, to digital television, data compression is an integral part of almost all information technology. This incorporation of compression into more and more of our lives also points to a certain degree of maturation of the technology. This maturity is reflected in the fact that there are fewer differences between this and the previous edition of this book than there were between the second and first editions. In the second edition we had added new techniques that had been developed since the first edition of this book came out. In this edition our purpose is more to include some important topics, such as audio compression, that had not been adequately covered in the second edition. During this time the field has not entirely stood still and we have tried to include information about new developments. We have added a new chapter on audio compression (including a description of the *mp3* algorithm). We have added information on new standards such as the new video coding standard and the new facsimile standard. We have reorganized some of the material in the book, collecting together various lossless image compression techniques and standards into a single chapter, and we have updated a number of chapters, adding information that perhaps should have been there from the beginning.

All this has yet again enlarged the book. However, the intent remains the same: to provide an introduction to the art or science of data compression. There is a tutorial description of most of the popular compression techniques followed by a description of how these techniques are used for image, speech, text, audio, and video compression.

Given the pace of developments in this area, there are bound to be new ones that are not reflected in this book. In order to keep you informed of these developments, we will periodically provide updates at *http://www.mkp.com.*

Audience

If you are designing hardware or software implementations of compression algorithms, or need to interact with individuals engaged in such design, or are involved in development of multimedia applications and have some background in either electrical or computer engineering, or computer science, this book should be useful to you. We have included a large number of examples to aid in self-study. We have also included discussion of various multimedia standards. The intent here is not to provide all the details that may be required to implement a standard but to provide information that will help you follow and understand the standards documents.

Course Use

The impetus for writing this book came from the need for a self-contained book that could be used at the senior/graduate level for a course in data compression in either electrical engineering, computer engineering, or computer science departments. There are problems and project ideas after most of the chapters. A solutions manual is available from the publisher. Also at *http://sensin.unl.edu/idc/index.html* we provide links to various course homepages, which can be a valuable source of project ideas and support material.

The material in this book is too much for a one semester course. However, with judicious use of the starred sections, this book can be tailored to fit a number of compression courses that emphasize various aspects of compression. If the course emphasis is on lossless compression, the instructor could cover most of the sections in the first seven chapters. Then, to give a taste of lossy compression, the instructor could cover Sections 1–5 of Chapter 9, followed by Chapter 13 and its description of JPEG, and Chapter 18, which describes video compression approaches used in multimedia communications. If the class interest is more attuned to audio compression, then instead of Chapters 13 and 18, the instructor could cover Chapters 14 and 16. If the latter option is taken, depending on the background of the students in the class, Chapter 12 may be assigned as background reading. If the emphasis is to be on lossy compression, the instructor could cover Chapter 2, the first two sections of Chapter 3, Sections 4 and 6 of Chapter 4 (with a cursory overview of Sections 2 and 3), Chapter 8, selected parts of Chapter 9, and Chapter 10 through 15. At this point depending on the time available and the interests of the instructor and the students portions of the remaining three chapters can be covered. I have always found it useful to assign a term project in which the students can follow their own interests as a means of covering material that is not covered in class but is of interest to the student.

Approach

In this book, we cover both lossless and lossy compression techniques with applications to image, speech, text, audio, and video compression. The various lossless and lossy coding techniques are introduced with just enough theory to tie things together. The necessary theory is introduced just before we need it. Therefore, there are three *mathematical preliminaries* chapters. In each of these chapters, we present the mathematical material needed to understand and appreciate the techniques that follow.

Although this book is an introductory text, the word *introduction* may have a different meaning for different audiences. We have tried to accommodate the needs of different audiences by taking a dual-track approach. Wherever we felt there was material that could enhance the understanding of the subject being discussed but could still be skipped without seriously hindering your understanding of the technique, we marked those sections with a star (⋆). If you are primarily interested in understanding how the various techniques function, especially if you are using this book for self-study, we recommend you skip the starred sections, at least in a first reading. Readers who require a slightly more theoretical approach should use the starred sections. Except for the starred sections, we have tried to keep the mathematics to a minimum.

Learning from This Book

I have found that it is easier for me to understand things if I can see examples. Therefore, I have relied heavily on examples to explain concepts. You may find it useful to spend more time with the examples if you have difficulty with some of the concepts.

Compression is still largely an art and to gain proficiency in an art we need to get a "feel" for the process. We have included software implementations for most of the techniques discussed in this book, along with a large number of data sets. The software and data sets can be obtained from *ftp://ftp.mkp.com/pub/Sayood/*. The programs are written in C and have been tested on a number of platforms. The programs should run under most flavors of UNIX machines and, with some slight modifications, under other operating systems as well. More detailed information is contained in the README file in the *pub/Sayood* directory.

You are strongly encouraged to use and modify these programs to work with your favorite data in order to understand some of the issues involved in compression. A useful and achievable goal should be the development of your own compression package by the time you have worked through this book. This would also be a good way to learn the trade-offs involved in different approaches. We have tried to give comparisons of techniques wherever possible; however, different types of data have their own idiosyncrasies. The best way to know which scheme to use in any given situation is to try them.

Content and Organization

The organization of the chapters is as follows: We introduce the mathematical preliminaries necessary for understanding lossless compression in Chapter 2; Chapters 3 and 4 are devoted to coding algorithms, including Huffman coding, arithmetic coding, Golomb-Rice codes, and Tunstall codes. Chapters 5 and 6 describe many of the popular lossless compression schemes along with their applications. The schemes include LZW, *ppm*, BWT, and DMC, among others. In Chapter 7 we describe a number of lossless image compression algorithms and their applications in a number of international standards. The standards include the JBIG standards and various facsimile standards.

Chapter 8 is devoted to providing the mathematical preliminaries for lossy compression. Quantization is at the heart of most lossy compression schemes. Chapters 9 and 10 are devoted to the study of quantization. Chapter 9 deals with scalar quantization, and Chapter 10 deals with vector quantization. Chapter 11 deals with differential encoding techniques, in particular differential pulse code modulation (DPCM) and delta modulation. Included in this chapter is a discussion of the CCITT G.726 standard.

Chapter 12 is our third mathematical preliminaries chapter. The goal of this chapter is to provide the mathematical foundation necessary to understand some aspects of the transform, subband, and wavelet-based techniques that are described in the next three chapters. As in the case of the previous mathematical preliminaries chapters, not all material covered is necessary for everyone. We describe the JPEG standard in Chapter 13, the CCITT G.722 international standard in Chapter 14, and EZW, SPIHT, and JPEG 2000 in Chapter 15.

Chapter 16 is devoted to audio compression. We describe the various MPEG audio compression schemes in this chapter including the scheme popularly known as *mp3*.

Chapter 17 covers techniques in which the data to be compressed are analyzed, and a model for the generation of the data is transmitted to the receiver. The receiver uses this model to synthesize the data. These analysis/synthesis and analysis by synthesis schemes include linear predictive schemes used for low-rate speech coding and the fractal compression technique. We describe the federal government LPC-10 standard. Code-excited linear prediction (CELP) is a popular example of an analysis by synthesis scheme. We also discuss three CELP-based standards, the federal standard 1016, the CCITT G.728 international standard, and the relatively new wideband speech compression standard G.722.2. We have also included a discussion of the mixed excitation linear prediction (MELP) technique, which is the new federal standard for speech coding at 2.4 kbps.

Chapter 18 deals with video coding. We describe popular video coding techniques via description of various international standards, including H.261, H.264, and the various MPEG standards.

A Personal View

For me, data compression is more than a manipulation of numbers; it is the process of discovering structures that exist in the data. In the 9th century, the poet Omar Khayyam wrote

> The moving finger writes, and having writ,
> moves on; not all thy piety nor wit,
> shall lure it back to cancel half a line,
> nor all thy tears wash out a word of it.
> *(The Rubaiyat of Omar Khayyam)*

To explain these few lines would take volumes. They tap into a common human experience so that in our mind's eye, we can reconstruct what the poet was trying to convey centuries ago. To understand the words we not only need to know the language, we also need to have a model of reality that is close to that of the poet. The genius of the poet lies in identifying a model of reality that is so much a part of our humanity that centuries later and in widely diverse cultures, these few words can evoke volumes.

Data compression is much more limited in its aspirations, and it may be presumptuous to mention it in the same breath as poetry. But there is much that is similar to both endeavors. Data compression involves identifying models for the many different types of structures that exist in different types of data and then using these models, perhaps along with the perceptual framework in which these data will be used, to obtain a compact representation of the data. These structures can be in the form of patterns that we can recognize simply by plotting the data, or they might be statistical structures that require a more mathematical approach to comprehend.

In *The Long Dark Teatime of the Soul* by Douglas Adams, the protagonist finds that he can enter Valhalla (a rather shoddy one) if he tilts his head in a certain way. Appreciating the structures that exist in data sometimes require us to tilt our heads in a certain way. There are an infinite number of ways we can tilt our head and, in order not to get a pain in the neck (carrying our analogy to absurd limits), it would be nice to know some of the ways that will generally lead to a profitable result. One of the objectives of this book is to provide you with a frame of reference that can be used for further exploration. I hope this exploration will provide as much enjoyment for you as it has given to me.

Acknowledgments

It has been a lot of fun writing this book. My task has been made considerably easier and the end product considerably better because of the help I have received. Acknowledging that help is itself a pleasure.

The first edition benefitted from the careful and detailed criticism of Roy Hoffman from IBM, Glen Langdon from the University of California at Santa Cruz, Debra Lelewer from California Polytechnic State University, Eve Riskin from the University of Washington, Ibrahim Sezan from Kodak, and Peter Swaszek from the University of Rhode Island. They provided detailed comments on all or most of the first edition. Nasir Memon from Polytechnic University, Victor Ramamoorthy then at S3, Grant Davidson at Dolby Corporation, Hakan Caglar, who was then at TÜBITAK in Istanbul, and Allen Gersho from the University of California at Santa Barbara reviewed parts of the manuscript.

For the second edition Steve Tate at the University of North Texas, Sheila Horan at New Mexico State University, Edouard Lamboray at Oerlikon Contraves Group, Steven Pigeon at the University of Montreal, and Jesse Olvera at Raytheon Systems reviewed the entire manuscript. Emin Anarım of Boğaziçi University and Hakan Çağlar helped me with the development of the chapter on wavelets. Mark Fowler provided extensive comments on Chapters 12–15, correcting mistakes of both commission and omission. Tim James, Devajani Khataniar, and Lance Pérez also read and critiqued parts of the new material in the second edition. Chloeann Nelson, along with trying to stop me from splitting infinitives, also tried to make the first two editions of the book more user-friendly.

Since the appearance of the first edition, various readers have sent me their comments and critiques. I am grateful to all who sent me comments and suggestions. I am especially grateful to Roberto Lopez-Hernandez, Dirk vom Stein, Christopher A. Larrieu, Ren Yih Wu, Humberto D'Ochoa, Roderick Mills, Mark Elston, and Jeerasuda Keesorth for pointing out errors and suggesting improvements to the book. I am also grateful to the various instructors who have sent me their critiques. In particular I would like to thank Bruce Bomar from the University of Tennessee, Mark Fowler from SUNY Binghamton, Paul Amer from the University of Delaware, K.R. Rao from the University of Texas at Arlington, Ralph Wilkerson from the University of Missouri–Rolla, Adam Drozdek from Duquesne University, Ed Hong and Richard Ladner from the University of Washington, Lars Nyland from the Colorado School of Mines, Mario Kovac from the University of Zagreb, and Pierre Jouvelet from the Ecole Superieure des Mines de Paris.

Frazer Williams and Mike Hoffman, from my department at the University of Nebraska, provided reviews for the first edition of the book. Mike read the new chapters in the second and third edition in their raw form and provided me with critiques that led to major rewrites. His insights were always helpful and the book carries more of his imprint than he is perhaps aware of. It is nice to have friends of his intellectual caliber and generosity. Rob Maher at Montana State University provided me with an extensive critique of the new chapter on

audio compression pointing out errors in my thinking and gently suggesting corrections. I thank him for his expertise, his time, and his courtesy.

Rick Adams, Rachel Roumeliotis, and Simon Crump at Morgan Kaufmann had the task of actually getting the book out. This included the unenviable task of getting me to meet deadlines. Vytas Statulevicius helped me with LaTex problems that were driving me up the wall.

Most of the examples in this book were generated in a lab set up by Andy Hadenfeldt. James Nau helped me extricate myself out of numerous software puddles giving freely of his time. In my times of panic, he was always just an email or voice mail away.

I would like to thank the various "models" for the data sets that accompany this book and were used as examples. The individuals in the images are Sinan Sayood, Sena Sayood, and Elif Sevuktekin. The female voice belongs to Pat Masek.

This book reflects what I have learned over the years. I have been very fortunate in the teachers I have had. David Farden, now at North Dakota State University, introduced me to the area of digital communication. Norm Griswold at Texas A&M University introduced me to the area of data compression. Jerry Gibson, now at University of California at Santa Barbara was my Ph.D. advisor and helped me get started on my professional career. The world may not thank him for that, but I certainly do.

I have also learned a lot from my students at the University of Nebraska and Boğaziçi University. Their interest and curiosity forced me to learn and kept me in touch with the broad field that is data compression today. I learned at least as much from them as they learned from me.

Much of this learning would not have been possible but for the support I received from NASA. The late Warner Miller and Pen-Shu Yeh at the Goddard Space Flight Center and Wayne Whyte at the Lewis Research Center were a source of support and ideas. I am truly grateful for their helpful guidance, trust, and friendship.

Our two boys, Sena and Sinan, graciously forgave my evenings and weekends at work. They were tiny (witness the images) when I first started writing this book. Soon I will have to look up when talking to them. "The book" has been their (sometimes unwanted) companion through all these years. For their graciousness and for always being such perfect joys, I thank them.

Above all the person most responsible for the existence of this book is my partner and closest friend Füsun. Her support and her friendship gives me the freedom to do things I would not otherwise even consider. She centers my universe and, as with every significant endeavor that I have undertaken since I met her, this book is at least as much hers as it is mine.

1

Introduction

In the last decade we have been witnessing a transformation—some call it a revolution—in the way we communicate, and the process is still under way. This transformation includes the ever-present, ever-growing Internet; the explosive development of mobile communications; and the ever-increasing importance of video communication. Data compression is one of the enabling technologies for each of these aspects of the multimedia revolution. It would not be practical to put images, let alone audio and video, on websites if it were not for data compression algorithms. Cellular phones would not be able to provide communication with increasing clarity were it not for compression. The advent of digital TV would not be possible without compression. Data compression, which for a long time was the domain of a relatively small group of engineers and scientists, is now ubiquitous. Make a long-distance call and you are using compression. Use your modem, or your fax machine, and you will benefit from compression. Listen to music on your *mp3* player or watch a DVD and you are being entertained courtesy of compression.

So, what is data compression, and why do we need it? Most of you have heard of JPEG and MPEG, which are standards for representing images, video, and audio. Data compression algorithms are used in these standards to reduce the number of bits required to represent an image or a video sequence or music. In brief, data compression is the art or science of representing information in a compact form. We create these compact representations by identifying and using structures that exist in the data. Data can be characters in a text file, numbers that are samples of speech or image waveforms, or sequences of numbers that are generated by other processes. The reason we need data compression is that more and more of the information that we generate and use is in digital form—in the form of numbers represented by bytes of data. And the number of bytes required to represent multimedia data can be huge. For example, in order to digitally represent 1 second of video without compression (using the CCIR 601 format), we need more than 20 megabytes, or 160 megabits. If we consider the number of seconds in a movie, we can easily see why we would need compression. To represent 2 minutes of uncompressed CD-quality music (44,100 samples per second, 16 bits per sample) requires more than 84 million bits. Downloading music from a website at these rates would take a long time.

As human activity has a greater and greater impact on our environment, there is an ever-increasing need for more information about our environment, how it functions, and what we are doing to it. Various space agencies from around the world, including the European Space Agency (ESA), the National Aeronautics and Space Agency (NASA), the Canadian Space Agency (CSA), and the Japanese Space Agency (STA), are collaborating on a program to monitor global change that will generate half a terabyte of data per *day* when they are fully operational. Compare this to the 130 terabytes of data currently stored at the EROS data center in South Dakota, that is the largest archive for land mass data in the world.

Given the explosive growth of data that needs to be transmitted and stored, why not focus on developing better transmission and storage technologies? This is happening, but it is not enough. There have been significant advances that permit larger and larger volumes of information to be stored and transmitted without using compression, including CD-ROMs, optical fibers, Asymmetric Digital Subscriber Lines (ADSL), and cable modems. However, while it is true that both storage and transmission capacities are steadily increasing with new technological innovations, as a corollary to Parkinson's First Law,[1] it seems that the need for mass storage and transmission increases at least twice as fast as storage and transmission capacities improve. Then there are situations in which capacity has not increased significantly. For example, the amount of information we can transmit over the airwaves will always be limited by the characteristics of the atmosphere.

An early example of data compression is Morse code, developed by Samuel Morse in the mid-19th century. Letters sent by telegraph are encoded with dots and dashes. Morse noticed that certain letters occurred more often than others. In order to reduce the average time required to send a message, he assigned shorter sequences to letters that occur more frequently, such as *e* (·) and *a* (· —), and longer sequences to letters that occur less frequently, such as *q* (— — · —) and *j* (· — — —). This idea of using shorter codes for more frequently occurring characters is used in Huffman coding, which we will describe in Chapter 3.

Where Morse code uses the frequency of occurrence of single characters, a widely used form of Braille code, which was also developed in the mid-19th century, uses the frequency of occurrence of words to provide compression [1]. In Braille coding, 2×3 arrays of dots are used to represent text. Different letters can be represented depending on whether the dots are raised or flat. In Grade 1 Braille, each array of six dots represents a single character. However, given six dots with two positions for each dot, we can obtain 2^6, or 64, different combinations. If we use 26 of these for the different letters, we have 38 combinations left. In Grade 2 Braille, some of these leftover combinations are used to represent words that occur frequently, such as "and" and "for." One of the combinations is used as a special symbol indicating that the symbol that follows is a word and not a character, thus allowing a large number of words to be represented by two arrays of dots. These modifications, along with contractions of some of the words, result in an average reduction in space, or compression, of about 20% [1].

---

[1] Parkinson's First Law: "Work expands so as to fill the time available," in *Parkinson's Law and Other Studies in Administration*, by Cyril Northcote Parkinson, Ballantine Books, New York, 1957.

Statistical structure is being used to provide compression in these examples, but that is not the only kind of structure that exists in the data. There are many other kinds of structures existing in data of different types that can be exploited for compression. Consider speech. When we speak, the physical construction of our voice box dictates the kinds of sounds that we can produce. That is, the mechanics of speech production impose a structure on speech. Therefore, instead of transmitting the speech itself, we could send information about the conformation of the voice box, which could be used by the receiver to synthesize the speech. An adequate amount of information about the conformation of the voice box can be represented much more compactly than the numbers that are the sampled values of speech. Therefore, we get compression. This compression approach is being used currently in a number of applications, including transmission of speech over mobile radios and the synthetic voice in toys that speak. An early version of this compression approach, called the *vocoder* (*vo*ice *coder*), was developed by Homer Dudley at Bell Laboratories in 1936. The vocoder was demonstrated at the New York World's Fair in 1939, where it was a major attraction. We will revisit the vocoder and this approach to compression of speech in Chapter 17.

These are only a few of the many different types of structures that can be used to obtain compression. The structure in the data is not the only thing that can be exploited to obtain compression. We can also make use of the characteristics of the user of the data. Many times, for example, when transmitting or storing speech and images, the data are intended to be perceived by a human, and humans have limited perceptual abilities. For example, we cannot hear the very high frequency sounds that dogs can hear. If something is represented in the data that cannot be perceived by the user, is there any point in preserving that information? The answer often is "no." Therefore, we can make use of the perceptual limitations of humans to obtain compression by discarding irrelevant information. This approach is used in a number of compression schemes that we will visit in Chapters 13, 14, and 16.

Before we embark on our study of data compression techniques, let's take a general look at the area and define some of the key terms and concepts we will be using in the rest of the book.

1.1 Compression Techniques

When we speak of a compression technique or compression algorithm,[2] we are actually referring to two algorithms. There is the compression algorithm that takes an input $\mathcal{X}$ and generates a representation $\mathcal{X}_c$ that requires fewer bits, and there is a reconstruction algorithm that operates on the compressed representation $\mathcal{X}_c$ to generate the reconstruction $\mathcal{Y}$. These operations are shown schematically in Figure 1.1. We will follow convention and refer to both the compression and reconstruction algorithms together to mean the compression algorithm.

---

[2] The word *algorithm* comes from the name of an early 9th-century Arab mathematician, Al-Khwarizmi, who wrote a treatise entitled *The Compendious Book on Calculation by* al-jabr *and* al-muqabala, in which he explored (among other things) the solution of various linear and quadratic equations via rules or an "algorithm." This approach became known as the method of Al-Khwarizmi. The name was changed to *algoritni* in Latin, from which we get the word *algorithm*. The name of the treatise also gave us the word *algebra* [2].

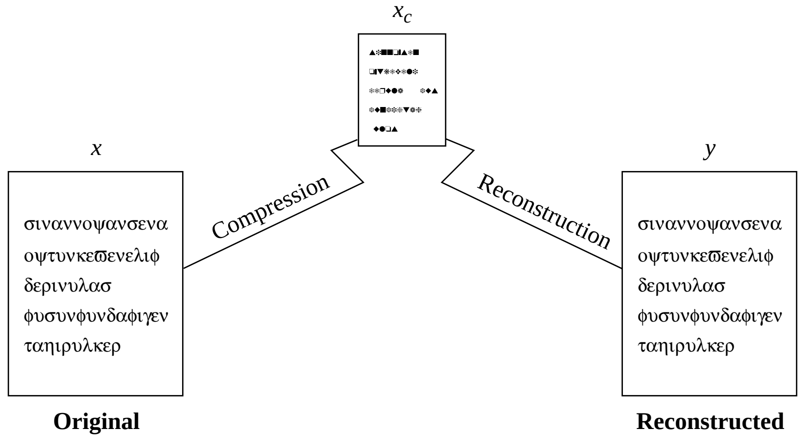


FIGURE 1.1 **Compression and reconstruction.**

Based on the requirements of reconstruction, data compression schemes can be divided into two broad classes: *lossless* compression schemes, in which $\mathcal{Y}$ is identical to $\mathcal{X}$, and *lossy* compression schemes, which generally provide much higher compression than lossless compression but allow $\mathcal{Y}$ to be different from $\mathcal{X}$.

1.1.1 Lossless Compression

Lossless compression techniques, as their name implies, involve no loss of information. If data have been losslessly compressed, the original data can be recovered exactly from the compressed data. Lossless compression is generally used for applications that cannot tolerate any difference between the original and reconstructed data.

Text compression is an important area for lossless compression. It is very important that the reconstruction is identical to the text original, as very small differences can result in statements with very different meanings. Consider the sentences "Do *not* send money" and "Do *now* send money." A similar argument holds for computer files and for certain types of data such as bank records.

If data of any kind are to be processed or "enhanced" later to yield more information, it is important that the integrity be preserved. For example, suppose we compressed a radiological image in a lossy fashion, and the difference between the reconstruction $\mathcal{Y}$ and the original $\mathcal{X}$ was visually undetectable. If this image was later enhanced, the previously undetectable differences may cause the appearance of artifacts that could seriously mislead the radiologist. Because the price for this kind of mishap may be a human life, it makes sense to be very careful about using a compression scheme that generates a reconstruction that is different from the original.

Data obtained from satellites often are processed later to obtain different numerical indicators of vegetation, deforestation, and so on. If the reconstructed data are not identical to the original data, processing may result in "enhancement" of the differences. It may not be possible to go back and obtain the same data over again. Therefore, it is not advisable to allow for any differences to appear in the compression process.

There are many situations that require compression where we want the reconstruction to be identical to the original. There are also a number of situations in which it is possible to relax this requirement in order to get more compression. In these situations we look to lossy compression techniques.

1.1.2 Lossy Compression

Lossy compression techniques involve some loss of information, and data that have been compressed using lossy techniques generally cannot be recovered or reconstructed exactly. In return for accepting this distortion in the reconstruction, we can generally obtain much higher compression ratios than is possible with lossless compression.

In many applications, this lack of exact reconstruction is not a problem. For example, when storing or transmitting speech, the exact value of each sample of speech is not necessary. Depending on the quality required of the reconstructed speech, varying amounts of loss of information about the value of each sample can be tolerated. If the quality of the reconstructed speech is to be similar to that heard on the telephone, a significant loss of information can be tolerated. However, if the reconstructed speech needs to be of the quality heard on a compact disc, the amount of information loss that can be tolerated is much lower.

Similarly, when viewing a reconstruction of a video sequence, the fact that the reconstruction is different from the original is generally not important as long as the differences do not result in annoying artifacts. Thus, video is generally compressed using lossy compression.

Once we have developed a data compression scheme, we need to be able to measure its performance. Because of the number of different areas of application, different terms have been developed to describe and measure the performance.

1.1.3 Measures of Performance

A compression algorithm can be evaluated in a number of different ways. We could measure the relative complexity of the algorithm, the memory required to implement the algorithm, how fast the algorithm performs on a given machine, the amount of compression, and how closely the reconstruction resembles the original. In this book we will mainly be concerned with the last two criteria. Let us take each one in turn.

A very logical way of measuring how well a compression algorithm compresses a given set of data is to look at the ratio of the number of bits required to represent the data before compression to the number of bits required to represent the data after compression. This ratio is called the *compression ratio*. Suppose storing an image made up of a square array of 256×256 pixels requires 65,536 bytes. The image is compressed and the compressed version requires 16,384 bytes. We would say that the compression ratio is 4:1. We can also represent the compression ratio by expressing the reduction in the amount of data required as a percentage of the size of the original data. In this particular example the compression ratio calculated in this manner would be 75%.

Another way of reporting compression performance is to provide the average number of bits required to represent a single sample. This is generally referred to as the *rate*. For example, in the case of the compressed image described above, if we assume 8 bits per byte (or pixel), the average number of bits per pixel in the compressed representation is 2. Thus, we would say that the rate is 2 bits per pixel.

In lossy compression, the reconstruction differs from the original data. Therefore, in order to determine the efficiency of a compression algorithm, we have to have some way of quantifying the difference. The difference between the original and the reconstruction is often called the *distortion*. (We will describe several measures of distortion in Chapter 8.) Lossy techniques are generally used for the compression of data that originate as analog signals, such as speech and video. In compression of speech and video, the final arbiter of quality is human. Because human responses are difficult to model mathematically, many approximate measures of distortion are used to determine the quality of the reconstructed waveforms. We will discuss this topic in more detail in Chapter 8.

Other terms that are also used when talking about differences between the reconstruction and the original are *fidelity* and *quality*. When we say that the fidelity or quality of a reconstruction is high, we mean that the difference between the reconstruction and the original is small. Whether this difference is a mathematical difference or a perceptual difference should be evident from the context.

1.2 Modeling and Coding

While reconstruction requirements may force the decision of whether a compression scheme is to be lossy or lossless, the exact compression scheme we use will depend on a number of different factors. Some of the most important factors are the characteristics of the data that need to be compressed. A compression technique that will work well for the compression of text may not work well for compressing images. Each application presents a different set of challenges.

There is a saying attributed to Bobby Knight, the basketball coach at Texas Tech University: “If the only tool you have is a hammer, you approach every problem as if it were a nail.” Our intention in this book is to provide you with a large number of tools that you can use to solve the particular data compression problem. It should be remembered that data compression, if it is a science at all, is an experimental science. The approach that works best for a particular application will depend to a large extent on the redundancies inherent in the data.

The development of data compression algorithms for a variety of data can be divided into two phases. The first phase is usually referred to as *modeling*. In this phase we try to extract information about any redundancy that exists in the data and describe the redundancy in the form of a model. The second phase is called *coding*. A description of the model and a “description” of how the data differ from the model are encoded, generally using a binary alphabet. The difference between the data and the model is often referred to as the *residual*. In the following three examples we will look at three different ways that data can be modeled. We will then use the model to obtain compression.

Example 1.2.1:

Consider the following sequence of numbers $\{x_1, x_2, x_3, \ldots\}$:

9	11	11	11	14	13	15	17	16	17	20	21

If we were to transmit or store the binary representations of these numbers, we would need to use 5 bits per sample. However, by exploiting the structure in the data, we can represent the sequence using fewer bits. If we plot these data as shown in Figure 1.2, we see that the data seem to fall on a straight line. A model for the data could therefore be a straight line given by the equation

$$\hat{x}_n = n + 8 \qquad n = 1, 2, \ldots$$

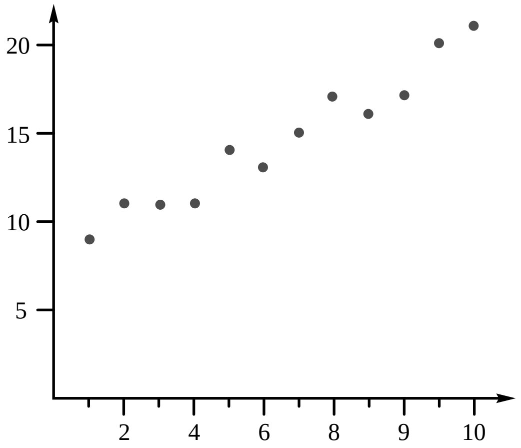


FIGURE 1. 2 **A sequence of data values.**

Thus, the structure in the data can be characterized by an equation. To make use of this structure, let's examine the difference between the data and the model. The difference (or residual) is given by the sequence

$$e_n = x_n - \hat{x}_n \; : \; 0 \; 1 \; 0 \; -1 \; 1 \; -1 \; 0 \; 1 \; -1 \; -1 \; 1 \; 1$$

The residual sequence consists of only three numbers $\{-1, 0, 1\}$. If we assign a code of 00 to -1, a code of 01 to 0, and a code of 10 to 1, we need to use 2 bits to represent each element of the residual sequence. Therefore, we can obtain compression by transmitting or storing the parameters of the model and the residual sequence. The encoding can be exact if the required compression is to be lossless, or approximate if the compression can be lossy. ◆

The type of structure or redundancy that existed in these data follows a simple law. Once we recognize this law, we can make use of the structure to *predict* the value of each element in the sequence and then encode the residual. Structure of this type is only one of many types of structure. Consider the following example.

Example 1.2.2:

Consider the following sequence of numbers:

27	28	29	28	26	27	29	28	30	32	34	36	38

The sequence is plotted in Figure 1.3.

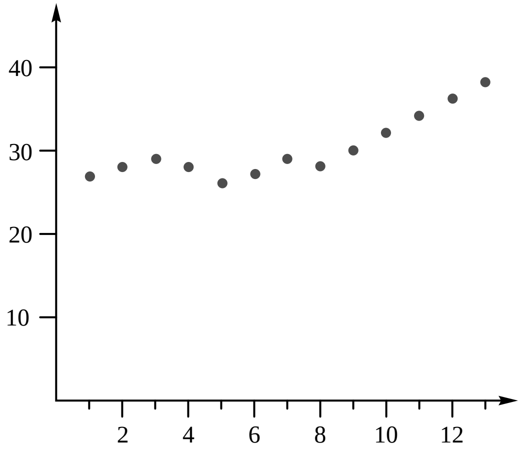


FIGURE 1. 3 **A sequence of data values.**

The sequence does not seem to follow a simple law as in the previous case. However, each value is close to the previous value. Suppose we send the first value, then in place of subsequent values we send the difference between it and the previous value. The sequence of transmitted values would be

27	1	1	−1	−2	1	2	−1	2	2	2	2	2

Like the previous example, the number of distinct values has been reduced. Fewer bits are required to represent each number and compression is achieved. The decoder adds each received value to the previous decoded value to obtain the reconstruction corresponding to the received value. Techniques that use the past values of a sequence to *predict* the current value and then encode the error in prediction, or residual, are called *predictive coding* schemes. We will discuss lossless predictive compression schemes in Chapter 7 and lossy predictive coding schemes in Chapter 11.

Assuming both encoder and decoder know the model being used, we would still have to send the value of the first element of the sequence.

◆

A very different type of redundancy is statistical in nature. Often we will encounter sources that generate some symbols more often than others. In these situations, it will be advantageous to assign binary codes of different lengths to different symbols.

Example 1.2.3:

Suppose we have the following sequence:

a␢barayaran␢barray␢bran␢bfar␢bfaar␢bfaaar␢baway

which is typical of all sequences generated by a source. Notice that the sequence is made up of eight different symbols. In order to represent eight symbols, we need to use 3 bits per symbol. Suppose instead we used the code shown in Table 1.1. Notice that we have assigned a codeword with only a single bit to the symbol that occurs most often, and correspondingly longer codewords to symbols that occur less often. If we substitute the codes for each symbol, we will use 106 bits to encode the entire sequence. As there are 41 symbols in the sequence, this works out to approximately 2.58 bits per symbol. This means we have obtained a compression ratio of 1.16:1. We will study how to use statistical redundancy of this sort in Chapters 3 and 4.

TABLE 1.1 A code with codewords of varying length.

a	1
n	001
b	01100
f	0100
n	0111
r	000
w	01101
y	0101

◆

When dealing with text, along with statistical redundancy, we also see redundancy in the form of words that repeat often. We can take advantage of this form of redundancy by constructing a list of these words and then represent them by their position in the list. This type of compression scheme is called a *dictionary* compression scheme. We will study these schemes in Chapter 5.

Often the structure or redundancy in the data becomes more evident when we look at groups of symbols. We will look at compression schemes that take advantage of this in Chapters 4 and 10.

Finally, there will be situations in which it is easier to take advantage of the structure if we decompose the data into a number of components. We can then study each component separately and use a model appropriate to that component. We will look at such schemes in Chapters 13, 14, and 15.

There are a number of different ways to characterize data. Different characterizations will lead to different compression schemes. We will study these compression schemes in the upcoming chapters, and use a number of examples that should help us understand the relationship between the characterization and the compression scheme.

With the increasing use of compression, there has also been an increasing need for standards. Standards allow products developed by different vendors to communicate. Thus, we can compress something with products from one vendor and reconstruct it using the products of a different vendor. The different international standards organizations have responded to this need, and a number of standards for various compression applications have been approved. We will discuss these standards as applications of the various compression techniques.

Finally, compression is still largely an art, and to gain proficiency in an art you need to get a feel for the process. To help, we have developed software implementations of most of the techniques discussed in this book, and also provided the data sets used for developing the examples in this book. Details on how to obtain these programs and data sets are provided in the Preface. You should use these programs on your favorite data or on the data sets provided in order to understand some of the issues involved in compression. We would also encourage you to write your own software implementations of some of these techniques, as very often the best way to understand how an algorithm works is to implement the algorithm.

1.3 Summary

In this chapter we have introduced the subject of data compression. We have provided some motivation for why we need data compression and defined some of the terminology we will need in this book. Additional terminology will be introduced as needed. We have briefly introduced the two major types of compression algorithms: lossless compression and lossy compression. Lossless compression is used for applications that require an exact reconstruction of the original data, while lossy compression is used when the user can tolerate some differences between the original and reconstructed representations of the data. An important element in the design of data compression algorithms is the modeling of the data. We have briefly looked at how modeling can help us in obtaining more compact representations of the data. We have described some of the different ways we can view the data in order to model it. The more ways we have of looking at the data, the more successful we will be in developing compression schemes that take full advantage of the structures in the data.

1.4 Projects and Problems

1. Use the compression utility on your computer to compress different files. Study the effect of the original file size and file type on the ratio of compressed file size to original file size.

2. Take a few paragraphs of text from a popular magazine and compress them by removing all words that are not essential for comprehension. For example, in the sentence "This is the dog that belongs to my friend," we can remove the words *is*, *the*, *that*, and *to* and still convey the same meaning. Let the ratio of the words removed to the total number of words in the original text be the measure of redundancy in the text. Repeat the experiment using paragraphs from a technical journal. Can you make any quantitative statements about the redundancy in the text obtained from different sources?

2

Mathematical Preliminaries for Lossless Compression

2.1 Overview

The treatment of data compression in this book is not very mathematical. (For a more mathematical treatment of some of the topics covered in this book, see [3, 4, 5, 6].) However, we do need some mathematical preliminaries to appreciate the compression techniques we will discuss. Compression schemes can be divided into two classes, lossy and lossless. Lossy compression schemes involve the loss of some information, and data that have been compressed using a lossy scheme generally cannot be recovered exactly. Lossless schemes compress the data without loss of information, and the original data can be recovered exactly from the compressed data. In this chapter, some of the ideas in information theory that provide the framework for the development of lossless data compression schemes are briefly reviewed. We will also look at some ways to model the data that lead to efficient coding schemes. We have assumed some knowledge of probability concepts (see Appendix A for a brief review of probability and random processes).

2.2 A Brief Introduction to Information Theory

Although the idea of a quantitative measure of information has been around for a while, the person who pulled everything together into what is now called information theory was Claude Elwood Shannon [7], an electrical engineer at Bell Labs. Shannon defined a quantity called *self-information*. Suppose we have an event A, which is a set of outcomes of some random experiment. If $P(A)$ is the probability that the event A will occur, then the self-information associated with A is given by

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A). \tag{2.1}$$

Note that we have not specified the base of the log function. We will discuss this in more detail later in the chapter. The use of the logarithm to obtain a measure of information was not an arbitrary choice as we shall see later in this chapter. But first let's see if the use of a logarithm in this context makes sense from an intuitive point of view. Recall that $\log(1) = 0$, and $-\log(x)$ increases as x decreases from one to zero. Therefore, if the probability of an event is low, the amount of self-information associated with it is high; if the probability of an event is high, the information associated with it is low. Even if we ignore the mathematical definition of information and simply use the definition we use in everyday language, this makes some intuitive sense. The barking of a dog during a burglary is a high-probability event and, therefore, does not contain too much information. However, if the dog did not bark during a burglary, this is a low-probability event and contains a lot of information. (Obviously, Sherlock Holmes understood information theory!)[1] Although this equivalence of the mathematical and semantic definitions of information holds true most of the time, it does not hold all of the time. For example, a totally random string of letters will contain more information (in the mathematical sense) than a well-thought-out treatise on information theory.

Another property of this mathematical definition of information that makes intuitive sense is that the information obtained from the occurrence of two independent events is the sum of the information obtained from the occurrence of the individual events. Suppose A and B are two independent events. The self-information associated with the occurrence of both event A *and* event B is, by Equation (2.1),

$$i(AB) = \log_b \frac{1}{P(AB)}.$$

As A and B are independent,

$$P(AB) = P(A)P(B)$$

and

$$\begin{aligned} i(AB) &= \log_b \frac{1}{P(A)P(B)} \\ &= \log_b \frac{1}{P(A)} + \log_b \frac{1}{P(B)} \\ &= i(A) + i(B). \end{aligned}$$

The unit of information depends on the base of the log. If we use log base 2, the unit is *bits*; if we use log base e, the unit is *nats*; and if we use log base 10, the unit is *hartleys*.

---

[1] *Silver Blaze* by Arthur Conan Doyle.

Note that to calculate the information in bits, we need to take the logarithm base 2 of the probabilities. Because this probably does not appear on your calculator, let's review logarithms briefly. Recall that

$$\log_b x = a$$

means that

$$b^a = x.$$

Therefore, if we want to take the log base 2 of x

$$\log_2 x = a \Rightarrow 2^a = x,$$

we want to find the value of a. We can take the natural log (log base e) or log base 10 of both sides (which do appear on your calculator). Then

$$\ln(2^a) = \ln x \Rightarrow a \ln 2 = \ln x$$

and

$$a = \frac{\ln x}{\ln 2}$$

Example 2.2.1:

Let H and T be the outcomes of flipping a coin. If the coin is fair, then

$$P(H) = P(T) = \tfrac{1}{2}$$

and

$$i(H) = i(T) = 1 \text{ bit}.$$

If the coin is not fair, then we would expect the information associated with each event to be different. Suppose

$$P(H) = \tfrac{1}{8}, \quad P(T) = \tfrac{7}{8}.$$

Then

$$i(H) = 3 \text{ bits}, \quad i(T) = 0.193 \text{ bits}.$$

At least mathematically, the occurrence of a head conveys much more information than the occurrence of a tail. As we shall see later, this has certain consequences for how the information conveyed by these outcomes should be encoded. ◆

If we have a set of independent events A_i, which are sets of outcomes of some experiment $\mathcal{S}$, such that

$$\bigcup A_i = S$$

where S is the sample space, then the average self-information associated with the random experiment is given by

$$H = \sum P(A_i)i(A_i) = -\sum P(A_i)\log_b P(A_i).$$

This quantity is called the *entropy* associated with the experiment. One of the many contributions of Shannon was that he showed that if the experiment is a source that puts out symbols A_i from a set $\mathcal{A}$, then the entropy is a measure of the average number of binary symbols needed to code the output of the source. Shannon showed that the best that a lossless compression scheme can do is to encode the output of a source with an average number of bits equal to the entropy of the source.

The set of symbols $\mathcal{A}$ is often called the *alphabet* for the source, and the symbols are referred to as *letters*. For a general source $\mathcal{S}$ with alphabet $\mathcal{A} = \{1, 2, \ldots, m\}$ that generates a sequence $\{X_1, X_2, \ldots\}$, the entropy is given by

$$H(\mathcal{S}) = \lim_{n\to\infty} \frac{1}{n} G_n \tag{2.2}$$

where

$$G_n = -\sum_{i_1=1}^{i_1=m}\sum_{i_2=1}^{i_2=m}\cdots\sum_{i_n=1}^{i_n=m} P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n)\log P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n)$$

and $\{X_1, X_2, \ldots, X_n\}$ is a sequence of length n from the source. We will talk more about the reason for the limit in Equation (2.2) later in the chapter. If each element in the sequence is independent and identically distributed (*iid*), then we can show that

$$G_n = -n\sum_{i_1=1}^{i_1=m} P(X_1 = i_1)\log P(X_1 = i_1) \tag{2.3}$$

and the equation for the entropy becomes

$$H(S) = -\sum P(X_1)\log P(X_1). \tag{2.4}$$

For most sources Equations (2.2) and (2.4) are not identical. If we need to distinguish between the two, we will call the quantity computed in (2.4) the *first-order entropy* of the source, while the quantity in (2.2) will be referred to as the *entropy* of the source.

In general, it is not possible to know the entropy for a physical source, so we have to estimate the entropy. The estimate of the entropy depends on our assumptions about the structure of the source sequence.

Consider the following sequence:

1 2 3 2 3 4 5 4 5 6 7 8 9 8 9 10

Assuming the frequency of occurrence of each number is reflected accurately in the number of times it appears in the sequence, we can estimate the probability of occurrence of each symbol as follows:

$$P(1) = P(6) = P(7) = P(10) = \tfrac{1}{16}$$

$$P(2) = P(3) = P(4) = P(5) = P(8) = P(9) = \tfrac{2}{16}.$$

Assuming the sequence is *iid*, the entropy for this sequence is the same as the first-order entropy as defined in (2.4). The entropy can then be calculated as

$$H = -\sum_{i=1}^{10} P(i) \log_2 P(i).$$

With our stated assumptions, the entropy for this source is 3.25 bits. This means that the best scheme we could find for coding this sequence could only code it at 3.25 bits/sample.

However, if we assume that there was sample-to-sample correlation between the samples and we remove the correlation by taking differences of neighboring sample values, we arrive at the *residual* sequence

$$1\ 1\ 1 -1\ 1\ 1\ 1\ -1\ 1\ 1\ 1\ 1\ 1\ 1\ -1\ 1\ 1\ 1$$

This sequence is constructed using only two values with probabilities $P(1) = \frac{13}{16}$ and $P(-1) = \frac{3}{16}$. The entropy in this case is 0.70 bits per symbol. Of course, knowing only this sequence would not be enough for the receiver to reconstruct the original sequence. The receiver must also know the process by which this sequence was generated from the original sequence. The process depends on our assumptions about the structure of the sequence. These assumptions are called the *model* for the sequence. In this case, the model for the sequence is

$$x_n = x_{n-1} + r_n$$

where x_n is the nth element of the original sequence and r_n is the nth element of the residual sequence. This model is called a *static* model because its parameters do not change with n. A model whose parameters change or adapt with n to the changing characteristics of the data is called an *adaptive* model.

Basically, we see that knowing something about the structure of the data can help to "reduce the entropy." We have put "reduce the entropy" in quotes because the entropy of the source is a measure of the amount of information generated by the source. As long as the information generated by the source is preserved (in whatever representation), the entropy remains the same. What we are reducing is our estimate of the entropy. The "actual" structure of the data in practice is generally unknowable, but anything we can learn about the data can help us to estimate the actual source entropy. Theoretically, as seen in Equation (2.2), we accomplish this in our definition of the entropy by picking larger and larger blocks of data to calculate the probability over, letting the size of the block go to infinity.

Consider the following contrived sequence:

$$1\ 2\ 1\ 2\ 3\ 3\ 3\ 3\ 1\ 2\ 3\ 3\ 3\ 3\ 1\ 2\ 3\ 3\ 1\ 2$$

Obviously, there is some structure to this data. However, if we look at it one symbol at a time, the structure is difficult to extract. Consider the probabilities: $P(1) = P(2) = \frac{1}{4}$, and $P(3) = \frac{1}{2}$. The entropy is 1.5 bits/symbol. This particular sequence consists of 20 symbols; therefore, the total number of bits required to represent this sequence is 30. Now let's take the same sequence and look at it in blocks of two. Obviously, there are only two symbols, 1 2, and 3 3. The probabilities are $P(1\ 2) = \frac{1}{2}$, $P(3\ 3) = \frac{1}{2}$, and the entropy is 1 bit/symbol.

As there are 10 such symbols in the sequence, we need a total of 10 bits to represent the entire sequence—a reduction of a factor of three. The theory says we can always extract the structure of the data by taking larger and larger block sizes; in practice, there are limitations to this approach. To avoid these limitations, we try to obtain an accurate model for the data and code the source with respect to the model. In Section 2.3, we describe some of the models commonly used in lossless compression algorithms. But before we do that, let's make a slight detour and see a more rigorous development of the expression for average information. While the explanation is interesting, it is not really necessary for understanding much of what we will study in this book and can be skipped.

2.2.1 Derivation of Average Information ★

We start with the properties we want in our measure of average information. We will then show that requiring these properties in the information measure leads inexorably to the particular definition of average information, or entropy, that we have provided earlier.

Given a set of independent events $A_1, A_2, \ldots, A_n$ with probability $p_i = P(A_i)$, we desire the following properties in the measure of average information H:

1. We want H to be a continuous function of the probabilities p_i. That is, a small change in p_i should only cause a small change in the average information.
2. If all events are equally likely, that is, $p_i = 1/n$ for all i, then H should be a monotonically increasing function of n. The more possible outcomes there are, the more information should be contained in the occurrence of any particular outcome.
3. Suppose we divide the possible outcomes into a number of groups. We indicate the occurrence of a particular event by first indicating the group it belongs to, then indicating which particular member of the group it is. Thus, we get some information first by knowing which group the event belongs to and then we get additional information by learning which particular event (from the events in the group) has occurred. The information associated with indicating the outcome in multiple stages should not be any different than the information associated with indicating the outcome in a single stage.

   For example, suppose we have an experiment with three outcomes A_1, A_2, and A_3, with corresponding probabilities p_1, p_2, and p_3. The average information associated with this experiment is simply a function of the probabilities:

   $$H = H(p_1, p_2, p_3).$$

   Let's group the three outcomes into two groups

   $$B_1 = \{A_1\}, \quad B_2 = \{A_2, A_3\}.$$

   The probabilities of the events B_i are given by

   $$q_1 = P(B_1) = p_1, \quad q_2 = P(B_2) = p_2 + p_3.$$

If we indicate the occurrence of an event A_i by first declaring which group the event belongs to and then declaring which event occurred, the total amount of average information would be given by

$$H = H(q_1, q_2) + q_1 H\left(\frac{p_1}{q_1}\right) + q_2 H\left(\frac{p_2}{q_2}, \frac{p_3}{q_2}\right).$$

We require that the average information computed either way be the same.

In his classic paper, Shannon showed that the only way all these conditions could be satisfied was if

$$H = -K \sum p_i \log p_i$$

where K is an arbitrary positive constant. Let's review his proof as it appears in the appendix of his paper [7].

Suppose we have an experiment with $n = k^m$ equally likely outcomes. The average information $H(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$ associated with this experiment is a function of n. In other words,

$$H\left(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right) = A(n).$$

We can indicate the occurrence of an event from k^m events by a series of m choices from k equally likely possibilities. For example, consider the case of $k = 2$ and $m = 3$. There are eight equally likely events; therefore, $H(\frac{1}{8}, \frac{1}{8}, \ldots, \frac{1}{8}) = A(8)$.

We can indicate occurrence of any particular event as shown in Figure 2.1. In this case, we have a sequence of three selections. Each selection is between two equally likely possibilities. Therefore,

$$\begin{aligned} H\left(\tfrac{1}{8}, \tfrac{1}{8}, \ldots, \tfrac{1}{8}\right) &= A(8) \\ &= H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}\left[H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2})\right. \\ &\qquad\qquad \left. + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2})\right] \\ &\quad + \tfrac{1}{2}\left[H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2})\right. \\ &\qquad\qquad \left. + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2})\right] \\ &= 3H(\tfrac{1}{2}, \tfrac{1}{2}) \\ &= 3A(2). \end{aligned} \tag{2.5}$$

In other words,

$$A(8) = 3A(2).$$

(The rather odd way of writing the left-hand side of Equation (2.5) is to show how the terms correspond to the branches of the tree shown in Figure 2.1.) We can generalize this for the case of $n = k^m$ as

$$A(n) = A(k^m) = mA(k).$$

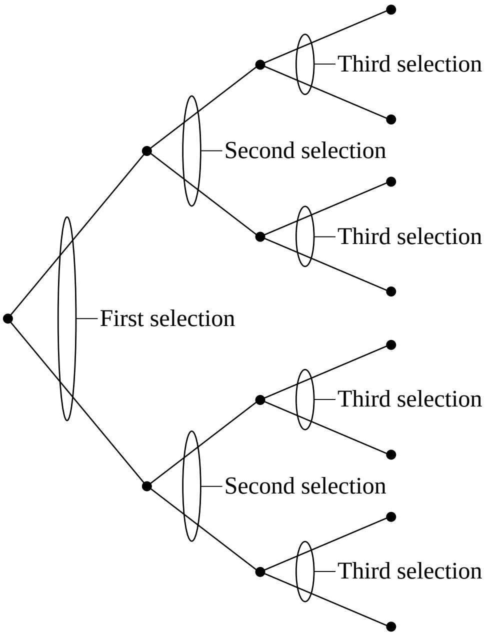

Third selection
Second selection
Third selection
First selection
Third selection
Second selection
Third selection

FIGURE 2. 1 **A possible way of identifying the occurrence of an event.**

Similarly, for j^l choices,

$$A(j^l) = lA(j).$$

We can pick l arbitrarily large (more on this later) and then choose m so that

$$k^m \leq j^l \leq k^{(m+1)}.$$

Taking logarithms of all terms, we get

$$m \log k \leq l \log j \leq (m+1) \log k.$$

Now divide through by $l \log k$ to get

$$\frac{m}{l} \leq \frac{\log j}{\log k} \leq \frac{m}{l} + \frac{1}{l}.$$

Recall that we picked l arbitrarily large. If l is arbitrarily large, then $\frac{1}{l}$ is arbitrarily small. This means that the upper and lower bounds of $\frac{\log j}{\log k}$ can be made arbitrarily close to $\frac{m}{l}$ by picking l arbitrarily large. Another way of saying this is

$$\left| \frac{m}{l} - \frac{\log j}{\log k} \right| < \epsilon$$

where ϵ can be made arbitrarily small. We will use this fact to find an expression for $A(n)$ and hence for $H(\frac{1}{n}, \ldots, \frac{1}{n})$.

To do this we use our second requirement that $H(\frac{1}{n}, \ldots, \frac{1}{n})$ be a monotonically increasing function of n. As

$$H\left(\frac{1}{n}, \ldots, \frac{1}{n}\right) = A(n),$$

this means that $A(n)$ is a monotonically increasing function of n. If

$$k^m \leq j^l \leq k^{m+1}$$

then in order to satisfy our second requirement

$$A(k^m) \leq A(j^l) \leq A(k^{m+1})$$

or

$$mA(k) \leq lA(j) \leq (m+1)A(k).$$

Dividing through by $lA(k)$, we get

$$\frac{m}{l} \leq \frac{A(j)}{A(k)} \leq \frac{m}{l} + \frac{1}{l}.$$

Using the same arguments as before, we get

$$\left| \frac{m}{l} - \frac{A(j)}{A(k)} \right| < \epsilon$$

where ϵ can be made arbitrarily small.

Now $\frac{A(j)}{A(k)}$ is at most a distance of ϵ away from $\frac{m}{l}$, and $\frac{\log j}{\log k}$ is at most a distance of ϵ away from $\frac{m}{l}$. Therefore, $\frac{A(j)}{A(k)}$ is at most a distance of 2ϵ away from $\frac{\log j}{\log k}$.

$$\left| \frac{A(j)}{A(k)} - \frac{\log j}{\log k} \right| < 2\epsilon$$

We can pick ϵ to be arbitrarily small, and j and k are arbitrary. The only way this inequality can be satisfied for arbitrarily small ϵ and arbitrary j and k is for $A(j) = K\log(j)$, where K is an arbitrary constant. In other words,

$$H = K\log(n).$$

Up to this point we have only looked at equally likely events. We now make the transition to the more general case of an experiment with outcomes that are not equally likely. We do that by considering an experiment with $\sum n_i$ equally likely outcomes that are grouped in n unequal groups of size n_i with rational probabilities (if the probabilities are not rational, we approximate them with rational probabilities and use the continuity requirement):

$$p_i = \frac{n_i}{\sum_{j=1}^{n} n_j}.$$

Given that we have $\sum n_i$ equally likely events, from the development above we have

$$H = K \log\left(\sum n_j\right). \tag{2.6}$$

If we indicate an outcome by first indicating which of the n groups it belongs to, and second indicating which member of the group it is, then by our earlier development the average information H is given by

$$\begin{aligned} H &= H(p_1, p_2, \ldots, p_n) + p_1 H\left(\frac{1}{n_1}, \ldots, \frac{1}{n_1}\right) + \cdots + p_n H\left(\frac{1}{n_n}, \ldots, \frac{1}{n_n}\right) && (2.7)\\ &= H(p_1, p_2, \ldots, p_n) + p_1 K \log n_1 + p_2 K \log n_2 + \cdots + p_n K \log n_n && (2.8)\\ &= H(p_1, p_2, \ldots, p_n) + K \sum_{i=1}^{n} p_i \log n_i. && (2.9) \end{aligned}$$

Equating the expressions in Equations (2.6) and (2.9), we obtain

$$K \log\left(\sum n_j\right) = H(p_1, p_2, \ldots, p_n) + K \sum_{i=1}^{n} p_i \log n_i$$

or

$$\begin{aligned} H(p_1, p_2, \ldots, p_n) &= K \log\left(\sum n_j\right) - K \sum_{i=1}^{n} p_i \log n_i \\ &= -K\left[\sum_{i=1}^{n} p_i \log n_i - \log\left(\sum_{j=1}^{n} n_j\right)\right] \\ &= -K\left[\sum_{i=1}^{n} p_i \log n_i - \log\left(\sum_{j=1}^{n} n_j\right) \sum_{i=1}^{n} p_i\right] && (2.10)\\ &= -K\left[\sum_{i=1}^{n} p_i \log n_i - \sum_{i=1}^{n} p_i \log\left(\sum_{j=1}^{n} n_j\right)\right] \\ &= -K \sum_{i=1}^{n} p_i \left[\log n_i - \log\left(\sum_{j=1}^{n} n_j\right)\right] \\ &= -K \sum_{i=1}^{n} p_i \log \frac{n_i}{\sum_{j=1}^{n} n_j} && (2.11)\\ &= -K \sum p_i \log p_i && (2.12) \end{aligned}$$

where, in Equation (2.10) we have used the fact that $\sum_{i=1}^{n} p_i = 1$. By convention we pick K to be 1, and we have the formula

$$H = -\sum p_i \log p_i.$$

Note that this formula is a natural outcome of the requirements we imposed in the beginning. It was not artificially forced in any way. Therein lies the beauty of information theory. Like the laws of physics, its laws are intrinsic in the nature of things. Mathematics is simply a tool to express these relationships.

2.3 Models

As we saw in Section 2.2, having a good model for the data can be useful in estimating the entropy of the source. As we will see in later chapters, good models for sources lead to more efficient compression algorithms. In general, in order to develop techniques that manipulate data using mathematical operations, we need to have a mathematical model for the data. Obviously, the better the model (i.e., the closer the model matches the aspects of reality that are of interest to us), the more likely it is that we will come up with a satisfactory technique. There are several approaches to building mathematical models.

2.3.1 Physical Models

If we know something about the physics of the data generation process, we can use that information to construct a model. For example, in speech-related applications, knowledge about the physics of speech production can be used to construct a mathematical model for the sampled speech process. Sampled speech can then be encoded using this model. We will discuss speech production models in more detail in Chapter 8.

Models for certain telemetry data can also be obtained through knowledge of the underlying process. For example, if residential electrical meter readings at hourly intervals were to be coded, knowledge about the living habits of the populace could be used to determine when electricity usage would be high and when the usage would be low. Then instead of the actual readings, the difference (residual) between the actual readings and those predicted by the model could be coded.

In general, however, the physics of data generation is simply too complicated to understand, let alone use to develop a model. Where the physics of the problem is too complicated, we can obtain a model based on empirical observation of the statistics of the data.

2.3.2 Probability Models

The simplest statistical model for the source is to assume that each letter that is generated by the source is independent of every other letter, and each occurs with the same probability. We could call this the *ignorance model*, as it would generally be useful only when we know nothing about the source. (Of course, that *really* might be true, in which case we have a rather unfortunate name for the model!) The next step up in complexity is to keep the independence assumption, but remove the equal probability assumption and assign a probability of occurrence to each letter in the alphabet. For a source that generates letters from an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_M\}$, we can have a *probability model* $\mathcal{P} = \{P(a_1), P(a_2), \ldots, P(a_M)\}$.

Given a probability model (and the independence assumption), we can compute the entropy of the source using Equation (2.4). As we will see in the following chapters using the probability model, we can also construct some very efficient codes to represent the letters in $\mathcal{A}$. Of course, these codes are only efficient if our mathematical assumptions are in accord with reality.

If the assumption of independence does not fit with our observation of the data, we can generally find better compression schemes if we discard this assumption. When we discard the independence assumption, we have to come up with a way to describe the dependence of elements of the data sequence on each other.

2.3.3 Markov Models

One of the most popular ways of representing dependence in the data is through the use of Markov models, named after the Russian mathematician Andrei Andrevich Markov (1856–1922). For models used in lossless compression, we use a specific type of Markov process called a *discrete time Markov chain*. Let $\{x_n\}$ be a sequence of observations. This sequence is said to follow a kth-order Markov model if

$$P(x_n | x_{n-1}, \ldots, x_{n-k}) = P(x_n | x_{n-1}, \ldots, x_{n-k}, \ldots). \tag{2.13}$$

In other words, knowledge of the past k symbols is equivalent to the knowledge of the entire past history of the process. The values taken on by the set $\{x_{n-1}, \ldots, x_{n-k}\}$ are called the *states* of the process. If the size of the source alphabet is l, then the number of states is l^k. The most commonly used Markov model is the first-order Markov model, for which

$$P(x_n | x_{n-1}) = P(x_n | x_{n-1}, x_{n-2}, x_{n-3}, \ldots). \tag{2.14}$$

Equations (2.13) and (2.14) indicate the existence of dependence between samples. However, they do not describe the form of the dependence. We can develop different first-order Markov models depending on our assumption about the form of the dependence between samples.

If we assumed that the dependence was introduced in a linear manner, we could view the data sequence as the output of a linear filter driven by white noise. The output of such a filter can be given by the difference equation

$$x_n = \rho x_{n-1} + \epsilon_n \tag{2.15}$$

where ϵ_n is a white noise process. This model is often used when developing coding algorithms for speech and images.

The use of the Markov model does not require the assumption of linearity. For example, consider a binary image. The image has only two types of pixels, white pixels and black pixels. We know that the appearance of a white pixel as the next observation depends, to some extent, on whether the current pixel is white or black. Therefore, we can model the pixel process as a discrete time Markov chain. Define two states S_w and S_b (S_w would correspond to the case where the current pixel is a white pixel, and S_b corresponds to the case where the current pixel is a black pixel). We define the transition probabilities $P(w/b)$ and $P(b/w)$, and the probability of being in each state $P(S_w)$ and $P(S_b)$. The Markov model can then be represented by the state diagram shown in Figure 2.2.

The entropy of a finite state process with states S_i is simply the average value of the entropy at each state:

$$H = \sum_{i=1}^{M} P(S_i) H(S_i). \tag{2.16}$$

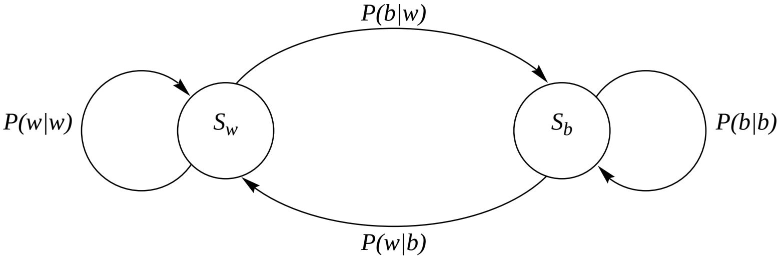


FIGURE 2. 2 **A two-state Markov model for binary images.**

For our particular example of a binary image

$$H(S_w) = -P(b/w)\log P(b/w) - P(w/w)\log P(w/w)$$

where $P(w/w) = 1 - P(b/w)$. $H(S_b)$ can be calculated in a similar manner.

Example 2.3.1: Markov model

To see the effect of modeling on the estimate of entropy, let us calculate the entropy for a binary image, first using a simple probability model and then using the finite state model described above. Let us assume the following values for the various probabilities:

$$P(S_w) = 30/31 \quad P(S_b) = 1/31$$

$$P(w|w) = 0.99 \quad P(b|w) = 0.01 \quad P(b|b) = 0.7 \quad P(w|b) = 0.3.$$

Then the entropy using a probability model and the *iid* assumption is

$$H = -0.8\log 0.8 - 0.2\log 0.2 = 0.206 \text{ bits.}$$

Now using the Markov model

$$H(S_b) = -0.3\log 0.3 - 0.7\log 0.7 = 0.881 \text{ bits}$$

and

$$H(S_w) = -0.01\log 0.01 - 0.99\log 0.99 = 0.081 \text{ bits}$$

which, using Equation (2.16), results in an entropy for the Markov model of 0.107 bits, about a half of the entropy obtained using the *iid* assumption. ◆

Markov Models in Text Compression

As expected, Markov models are particularly useful in text compression, where the probability of the next letter is heavily influenced by the preceding letters. In fact, the use of Markov models for written English appears in the original work of Shannon [7]. In current text compression literature, the *k*th-order Markov models are more widely known as *finite context models*, with the word *context* being used for what we have earlier defined as state.

Consider the word *preceding*. Suppose we have already processed *precedin* and are going to encode the next letter. If we take no account of the context and treat each letter as a surprise, the probability of the letter *g* occurring is relatively low. If we use a first-order Markov model or single-letter context (that is, we look at the probability model given *n*), we can see that the probability of *g* would increase substantially. As we increase the context size (go from *n* to *in* to *din* and so on), the probability of the alphabet becomes more and more skewed, which results in lower entropy.

Shannon used a second-order model for English text consisting of the 26 letters and one space to obtain an entropy of 3.1 bits/letter [8]. Using a model where the output symbols were words rather than letters brought down the entropy to 2.4 bits/letter. Shannon then used predictions generated by people (rather than statistical models) to estimate the upper and lower bounds on the entropy of the second order model. For the case where the subjects knew the 100 previous letters, he estimated these bounds to be 1.3 and 0.6 bits/letter, respectively.

The longer the context, the better its predictive value. However, if we were to store the probability model with respect to all contexts of a given length, the number of contexts would grow exponentially with the length of context. Furthermore, given that the source imposes some structure on its output, many of these contexts may correspond to strings that would never occur in practice. Consider a context model of order four (the context is determined by the last four symbols). If we take an alphabet size of 95, the possible number of contexts is 95^4—more than 81 million!

This problem is further exacerbated by the fact that different realizations of the source output may vary considerably in terms of repeating patterns. Therefore, context modeling in text compression schemes tends to be an adaptive strategy in which the probabilities for different symbols in the different contexts are updated as they are encountered. However, this means that we will often encounter symbols that have not been encountered before for any of the given contexts (this is known as the *zero frequency problem*). The larger the context, the more often this will happen. This problem could be resolved by sending a code to indicate that the following symbol was being encountered for the first time, followed by a prearranged code for that symbol. This would significantly increase the length of the code for the symbol on its first occurrence (in the given context). However, if this situation did not occur too often, the overhead associated with such occurrences would be small compared to the total number of bits used to encode the output of the source. Unfortunately, in context-based encoding, the zero frequency problem is encountered often enough for overhead to be a problem, especially for longer contexts. Solutions to this problem are presented by the *ppm* (prediction with partial match) algorithm and its variants (described in detail in Chapter 6).

Briefly, the *ppm* algorithms first attempt to find if the symbol to be encoded has a nonzero probability with respect to the maximum context length. If this is so, the symbol is encoded and transmitted. If not, an escape symbol is transmitted, the context size is reduced by one, and the process is repeated. This procedure is repeated until a context is found with respect to which the symbol has a nonzero probability. To guarantee that this process converges, a null context is always included with respect to which all symbols have equal probability. Initially, only the shorter contexts are likely to be used. However, as more and more of the source output is processed, the longer contexts, which offer better prediction,

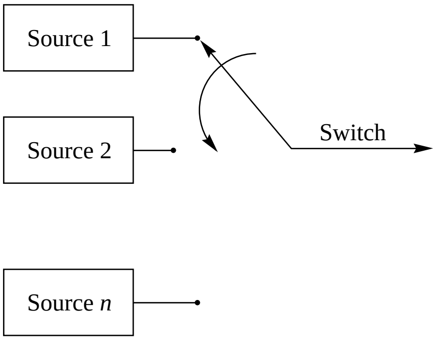


FIGURE 2.3 **A composite source.**

will be used more often. The probability of the escape symbol can be computed in a number of different ways leading to different implementations [1].

The use of Markov models in text compression is a rich and active area of research. We describe some of these approaches in Chapter 6 (for more details, see [1]).

2.3.4 Composite Source Model

In many applications, it is not easy to use a single model to describe the source. In such cases, we can define a *composite source*, which can be viewed as a combination or composition of several sources, with only one source being *active* at any given time. A composite source can be represented as a number of individual sources $\mathcal{S}_i$, each with its own model $\mathcal{M}_i$, and a switch that selects a source $\mathcal{S}_i$ with probability P_i (as shown in Figure 2.3). This is an exceptionally rich model and can be used to describe some very complicated processes. We will describe this model in more detail when we need it.

2.4 Coding

When we talk about *coding* in this chapter (and through most of this book), we mean the assignment of binary sequences to elements of an alphabet. The set of binary sequences is called a *code*, and the individual members of the set are called *codewords*. An *alphabet* is a collection of symbols called *letters*. For example, the alphabet used in writing most books consists of the 26 lowercase letters, 26 uppercase letters, and a variety of punctuation marks. In the terminology used in this book, a comma is a letter. The ASCII code for the letter *a* is 1000011, the letter *A* is coded as 1000001, and the letter "," is coded as 0011010. Notice that the ASCII code uses the same number of bits to represent each symbol. Such a code is called a *fixed-length code*. If we want to reduce the number of bits required to represent different messages, we need to use a different number of bits to represent different symbols. If we use fewer bits to represent symbols that occur more often, on the average we would use fewer bits per symbol. The average number of bits per symbol is often called the *rate* of the code. The idea of using fewer bits to represent symbols that occur more often is the same idea that is used in Morse code: the codewords for letters that occur more frequently are shorter than for letters that occur less frequently. For example, the codeword for E is •, while the codeword for Z is — — • • [9].

2.4.1 Uniquely Decodable Codes

The average length of the code is not the only important point in designing a "good" code. Consider the following example adapted from [10]. Suppose our source alphabet consists of four letters a_1, a_2, a_3, and a_4, with probabilities $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, and $P(a_3) = P(a_4) = \frac{1}{8}$. The entropy for this source is 1.75 bits/symbol. Consider the codes for this source in Table 2.1.

The average length l for for each code is given by

$$l = \sum_{i=1}^{4} P(a_i)n(a_i)$$

where $n(a_i)$ is the number of bits in the codeword for letter a_i and the average length is given in bits/symbol. Based on the average length, Code 1 appears to be the best code. However, to be useful, a code should have the ability to transfer information in an unambiguous manner. This is obviously not the case with Code 1. Both a_1 and a_2 have been assigned the codeword 0. When a 0 is received, there is no way to know whether an a_1 was transmitted or an a_2. We would like each symbol to be assigned a *unique* codeword.

At first glance Code 2 does not seem to have the problem of ambiguity; each symbol is assigned a distinct codeword. However, suppose we want to encode the sequence a_2 a_1 a_1. Using Code 2, we would encode this with the binary string 100. However, when the string 100 is received at the decoder, there are several ways in which the decoder can decode this string. The string 100 can be decoded as a_2 a_1 a_1, or as a_2 a_3. This means that once a sequence is encoded with Code 2, the original sequence cannot be recovered with certainty. In general, this is not a desirable property for a code. We would like *unique decodability* from the code; that is, any given sequence of codewords can be decoded in one, and only one, way.

We have already seen that Code 1 and Code 2 are not uniquely decodable. How about Code 3? Notice that the first three codewords all end in a 0. In fact, a 0 always denotes the termination of a codeword. The final codeword contains no 0s and is 3 bits long. Because all other codewords have fewer than three 1s and terminate in a 0, the only way we can get three 1s in a row is as a code for a_4. The decoding rule is simple. Accumulate bits until you get a 0 or until you have three 1s. There is no ambiguity in this rule, and it is reasonably

TABLE 2.1 Four different codes for a four-letter alphabet.

Letters	Probability	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average length		1.125	1.25	1.75	1.875

easy to see that this code is uniquely decodable. With Code 4 we have an even simpler condition. Each codeword starts with a 0, and the only time we see a 0 is in the beginning of a codeword. Therefore, the decoding rule is accumulate bits until you see a 0. The bit before the 0 is the last bit of the previous codeword.

There is a slight difference between Code 3 and Code 4. In the case of Code 3, the decoder knows the moment a code is complete. In Code 4, we have to wait till the beginning of the next codeword before we know that the current codeword is complete. Because of this property, Code 3 is called an *instantaneous* code. Although Code 4 is not an instantaneous code, it is almost that.

While this property of instantaneous or near-instantaneous decoding is a nice property to have, it is not a requirement for unique decodability. Consider the code shown in Table 2.2. Let's decode the string 01111111111111111. In this string, the first codeword is either 0 corresponding to a_1 or 01 corresponding to a_2. We cannot tell which one until we have decoded the whole string. Starting with the assumption that the first codeword corresponds to a_1, the next eight pairs of bits are decoded as a_3. However, after decoding eight a_3s, we are left with a single (dangling) 1 that does not correspond to any codeword. On the other hand, if we assume the first codeword corresponds to a_2, we can decode the next 16 bits as a sequence of eight a_3s, and we do not have any bits left over. The string can be uniquely decoded. In fact, Code 5, while it is certainly not instantaneous, is uniquely decodable.

We have been looking at small codes with four letters or less. Even with these, it is not immediately evident whether the code is uniquely decodable or not. In deciding whether larger codes are uniquely decodable, a systematic procedure would be useful. Actually, we should include a caveat with that last statement. Later in this chapter we will include a class of variable-length codes that are always uniquely decodable, so a test for unique decodability may not be that necessary. You might wish to skip the following discussion for now, and come back to it when you find it necessary.

Before we describe the procedure for deciding whether a code is uniquely decodable, let's take another look at our last example. We found that we had an incorrect decoding because we were left with a binary string (1) that was not a codeword. If this had not happened, we would have had two valid decodings. For example, consider the code shown in Table 2.3. Let's

TABLE 2.2 Code 5.

Letter	Codeword
a_1	0
a_2	01
a_3	11

TABLE 2.3 Code 6.

Letter	Codeword
a_1	0
a_2	01
a_3	10

encode the sequence a_1 followed by eight a_3s using this code. The coded sequence is 01010101010101010. The first bit is the codeword for a_1. However, we can also decode it as the first bit of the codeword for a_2. If we use this (incorrect) decoding, we decode the next seven pairs of bits as the codewords for a_2. After decoding seven a_2s, we are left with a single 0 that we decode as a_1. Thus, the incorrect decoding is also a valid decoding, and this code is not uniquely decodable.

A Test for Unique Decodability ★

In the previous examples, in the case of the uniquely decodable code, the binary string left over after we had gone through an incorrect decoding was not a codeword. In the case of the code that was not uniquely decodable, in the incorrect decoding what was left was a valid codeword. Based on whether the dangling suffix is a codeword or not, we get the following test [11, 12].

We start with some definitions. Suppose we have two binary codewords a and b, where a is k bits long, b is n bits long, and $k < n$. If the first k bits of b are identical to a, then a is called a *prefix* of b. The last $n - k$ bits of b are called the *dangling suffix* [11]. For example, if $a = 010$ and $b = 01011$, then a is a prefix of b and the dangling suffix is 11.

Construct a list of all the codewords. Examine all pairs of codewords to see if any codeword is a prefix of another codeword. Whenever you find such a pair, add the dangling suffix to the list unless you have added the same dangling suffix to the list in a previous iteration. Now repeat the procedure using this larger list. Continue in this fashion until one of the following two things happens:

1. You get a dangling suffix that is a codeword.
2. There are no more unique dangling suffixes.

If you get the first outcome, the code is not uniquely decodable. However, if you get the second outcome, the code is uniquely decodable.

Let's see how this procedure works with a couple of examples.

Example 2.4.1:

Consider Code 5. First list the codewords

$$\{0, 01, 11\}$$

The codeword 0 is a prefix for the codeword 01. The dangling suffix is 1. There are no other pairs for which one element of the pair is the prefix of the other. Let us augment the codeword list with the dangling suffix.

$$\{0, 01, 11, 1\}$$

Comparing the elements of this list, we find 0 is a prefix of 01 with a dangling suffix of 1. But we have already included 1 in our list. Also, 1 is a prefix of 11. This gives us a dangling suffix of 1, which is already in the list. There are no other pairs that would generate a dangling suffix, so we cannot augment the list any further. Therefore, Code 5 is uniquely decodable. ◆

Example 2.4.2:

Consider Code 6. First list the codewords

$$\{0, 01, 10\}$$

The codeword 0 is a prefix for the codeword 01. The dangling suffix is 1. There are no other pairs for which one element of the pair is the prefix of the other. Augmenting the codeword list with 1, we obtain the list

$$\{0, 01, 10, 1\}$$

In this list, 1 is a prefix for 10. The dangling suffix for this pair is 0, which is the codeword for a_1. Therefore, Code 6 is not uniquely decodable. ♦

2.4.2 Prefix Codes

The test for unique decodability requires examining the dangling suffixes initially generated by codeword pairs in which one codeword is the prefix of the other. If the dangling suffix is itself a codeword, then the code is not uniquely decodable. One type of code in which we will never face the possibility of a dangling suffix being a codeword is a code in which no codeword is a prefix of the other. In this case, the set of dangling suffixes is the null set, and we do not have to worry about finding a dangling suffix that is identical to a codeword. A code in which no codeword is a prefix to another codeword is called a *prefix code*. A simple way to check if a code is a prefix code is to draw the rooted binary tree corresponding to the code. Draw a tree that starts from a single node (the *root node*) and has a maximum of two possible branches at each node. One of these branches corresponds to a 1 and the other branch corresponds to a 0. In this book, we will adopt the convention that when we draw a tree with the root node at the top, the left branch corresponds to a 0 and the right branch corresponds to a 1. Using this convention, we can draw the binary tree for Code 2, Code 3, and Code 4 as shown in Figure 2.4.

Note that apart from the root node, the trees have two kinds of nodes—nodes that give rise to other nodes and nodes that do not. The first kind of nodes are called *internal nodes*, and the second kind are called *external nodes* or *leaves*. In a prefix code, the codewords are only associated with the external nodes. A code that is not a prefix code, such as Code 4, will have codewords associated with internal nodes. The code for any symbol can be obtained

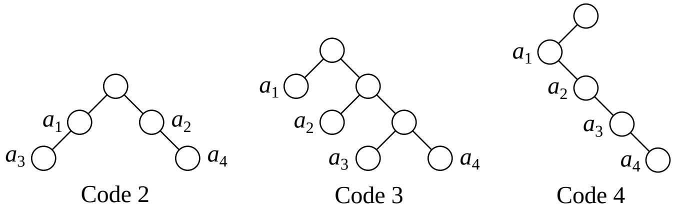


FIGURE 2.4 **Binary trees for three different codes.**

by traversing the tree from the root to the external node corresponding to that symbol. Each branch on the way contributes a bit to the codeword: a 0 for each left branch and a 1 for each right branch.

It is nice to have a class of codes, whose members are so clearly uniquely decodable. However, are we losing something if we restrict ourselves to prefix codes? Could it be that if we do not restrict ourselves to prefix codes, we can find shorter codes? Fortunately for us the answer is no. For any nonprefix uniquely decodable code, we can always find a prefix code with the same codeword lengths. We prove this in the next section.

2.4.3 The Kraft-McMillan Inequality ★

The particular result we look at in this section consists of two parts. The first part provides a necessary condition on the codeword lengths of uniquely decodable codes. The second part shows that we can always find a prefix code that satisfies this necessary condition. Therefore, if we have a uniquely decodable code that is not a prefix code, we can always find a prefix code with the same codeword lengths.

Theorem *Let $\mathcal{C}$ be a code with N codewords with lengths $l_1, l_2, \ldots, l_N$. If $\mathcal{C}$ is uniquely decodable, then*

$$K(\mathcal{C}) = \sum_{i=1}^{N} 2^{-l_i} \leq 1.$$

This inequality is known as the Kraft-McMillan inequality.

Proof The proof works by looking at the nth power of $K(\mathcal{C})$. If $K(\mathcal{C})$ is greater than one, then $K(\mathcal{C})^n$ should grow exponentially with n. If it does not grow exponentially with n, then this is proof that $\sum_{i=1}^{N} 2^{-l_i} \leq 1$.

Let n be an arbitrary integer. Then

$$\left[\sum_{i=1}^{N} 2^{-l_i}\right]^n = \left(\sum_{i_1=1}^{N} 2^{-l_{i_1}}\right)\left(\sum_{i_2=1}^{N} 2^{-l_{i_2}}\right)\cdots\left(\sum_{i_n=1}^{N} 2^{-l_{i_n}}\right) \tag{2.17}$$

$$= \sum_{i_1=1}^{N}\sum_{i_2=1}^{N}\cdots\sum_{i_n=1}^{N} 2^{-(l_{i_1}+l_{i_2}+\cdots+l_{i_n})}. \tag{2.18}$$

The exponent $l_{i_1}+l_{i_2}+\cdots+l_{i_n}$ is simply the length of n codewords from the code $\mathcal{C}$. The smallest value that this exponent can take is greater than or equal to n, which would be the case if all codewords were 1 bit long. If

$$l = \max\{l_1, l_2, \ldots, l_N\}$$

then the largest value that the exponent can take is less than or equal to nl. Therefore, we can write this summation as

$$K(\mathcal{C})^n = \sum_{k=n}^{nl} A_k 2^{-k}$$

where A_k is the number of combinations of n codewords that have a combined length of k. Let's take a look at the size of this coefficient. The number of possible distinct binary sequences of length k is 2^k. If this code is uniquely decodable, then each sequence can represent one and only one sequence of codewords. Therefore, the number of possible combinations of codewords whose combined length is k cannot be greater than 2^k. In other words,

$$A_k \leq 2^k.$$

This means that

$$K(\mathcal{C})^n = \sum_{k=n}^{nl} A_k 2^{-k} \leq \sum_{k=n}^{nl} 2^k 2^{-k} = nl - n + 1. \tag{2.19}$$

But if $K(\mathcal{C})$ is greater than one, it will grow exponentially with n, while $n(l-1)+1$ can only grow linearly. So if $K(\mathcal{C})$ is greater than one, we can always find an n large enough that the inequality (2.19) is violated. Therefore, for a uniquely decodable code $\mathcal{C}$, $K(\mathcal{C})$ is less than or equal to one. □

This part of the Kraft-McMillan inequality provides a necessary condition for uniquely decodable codes. That is, if a code is uniquely decodable, the codeword lengths have to satisfy the inequality. The second part of this result is that if we have a set of codeword lengths that satisfy the inequality, we can always find a prefix code with those codeword lengths. The proof of this assertion presented here is adapted from [6].

Theorem *Given a set of integers* $l_1, l_2, \ldots, l_N$ *that satisfy the inequality*

$$\sum_{i=1}^{N} 2^{-l_i} \leq 1$$

we can always find a prefix code with codeword lengths $l_1, l_2, \ldots, l_N$.

Proof We will prove this assertion by developing a procedure for constructing a prefix code with codeword lengths $l_1, l_2, \ldots, l_N$ that satisfy the given inequality.

Without loss of generality, we can assume that

$$l_1 \leq l_2 \leq \cdots \leq l_N.$$

Define a sequence of numbers $w_1, w_2, \ldots, w_N$ as follows:

$$\begin{aligned} w_1 &= 0 \\ w_j &= \sum_{i=1}^{j-1} 2^{l_j - l_i} \qquad j > 1. \end{aligned}$$

The binary representation of w_j for $j > 1$ would take up $\lceil \log_2(w_j + 1) \rceil$ bits. We will use this binary representation to construct a prefix code. We first note that the number of bits in the binary representation of w_j is less than or equal to l_j. This is obviously true for w_1. For $j > 1$,

$$\begin{aligned}
\log_2(w_j + 1) &= \log_2\left[\sum_{i=1}^{j-1} 2^{l_j - l_i} + 1\right] \\
&= \log_2\left[2^{l_j} \sum_{i=1}^{j-1} 2^{-l_i} + 2^{-l_j}\right] \\
&= l_j + \log_2\left[\sum_{i=1}^{j} 2^{-l_i}\right] \\
&\le l_j.
\end{aligned}$$

The last inequality results from the hypothesis of the theorem that $\sum_{i=1}^{N} 2^{-l_i} \le 1$, which implies that $\sum_{i=1}^{j} 2^{-l_i} \le 1$. As the logarithm of a number less than one is negative, $l_j + \log_2\left[\sum_{i=1}^{j} 2^{-l_i}\right]$ has to be less than l_j.

Using the binary representation of w_j, we can devise a binary code in the following manner: If $\lceil \log_2(w_j + 1) \rceil = l_j$, then the jth codeword c_j is the binary representation of w_j. If $\lceil \log_2(w_j + 1) \rceil < l_j$, then c_j is the binary representation of w_j, with $l_j - \lceil \log_2(w_j + 1) \rceil$ zeros appended to the right. This is certainly a code, but is it a prefix code? If we can show that the code $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$ is a prefix code, then we will have proved the theorem by construction.

Suppose that our claim is not true. Then for some $j < k$, c_j is a prefix of c_k. This means that the l_j most significant bits of w_k form the binary representation of w_j. Therefore if we right-shift the binary representation of w_k by $l_k - l_j$ bits, we should get the binary representation for w_j. We can write this as

$$w_j = \left\lfloor \frac{w_k}{2^{l_k - l_j}} \right\rfloor.$$

However,

$$w_k = \sum_{i=1}^{k-1} 2^{l_k - l_i}.$$

Therefore,

$$\begin{aligned}
\frac{w_k}{2^{l_k - l_j}} &= \sum_{i=0}^{k-1} 2^{l_j - l_i} \\
&= w_j + \sum_{i=j}^{k-1} 2^{l_j - l_i} \\
&= w_j + 2^0 + \sum_{i=j+1}^{k-1} 2^{l_j - l_i} \\
&\ge w_j + 1. \qquad (2.20)
\end{aligned}$$

That is, the smallest value for $\frac{w_k}{2^{l_k - l_j}}$ is $w_j + 1$. This contradicts the requirement for c_j being the prefix of c_k. Therefore, c_j cannot be the prefix for c_k. As j and k were arbitrary, this means that no codeword is a prefix of another codeword, and the code $\mathcal{C}$ is a prefix code. □

Therefore, if we have a uniquely decodable code, the codeword lengths have to satisfy the Kraft-McMillan inequality. And, given codeword lengths that satisfy the Kraft-McMillan inequality, we can always find a prefix code with those codeword lengths. Thus, by restricting ourselves to prefix codes, we are not in danger of overlooking nonprefix uniquely decodable codes that have a shorter average length.

2.5 Algorithmic Information Theory

The theory of information described in the previous sections is intuitively satisfying and has useful applications. However, when dealing with real world data, it does have some theoretical difficulties. Suppose you were given the task of developing a compression scheme for use with a specific set of documentations. We can view the entire set as a single long string. You could develop models for the data. Based on these models you could calculate probabilities using the relative frequency approach. These probabilities could then be used to obtain an estimate of the entropy and thus an estimate of the amount of compression available. All is well except for a fly in the "ointment." The string you have been given is fixed. There is nothing probabilistic about it. There is no abstract source that will generate different sets of documentation at different times. So how can we talk about the entropies without pretending that reality is somehow different from what it actually is? Unfortunately, it is not clear that we can. Our definition of entropy requires the existence of an abstract source. Our estimate of the entropy is still useful. It will give us a very good idea of how much compression we can get. So, practically speaking, information theory comes through. However, theoretically it seems there is some pretending involved. Algorithmic information theory is a different way of looking at information that has not been as useful in practice (and therefore we will not be looking at it a whole lot) but it gets around this theoretical problem. At the heart of algorithmic information theory is a measure called *Kolmogorov complexity*. This measure, while it bears the name of one person, was actually discovered independently by three people: R. Solomonoff, who was exploring machine learning; the Russian mathematician A.N. Kolmogorov; and G. Chaitin, who was in high school when he came up with this idea.

The Kolmogorov complexity $K(x)$ of a sequence x is the size of the program needed to generate x. In this size we include all inputs that might be needed by the program. We do not specify the programming language because it is always possible to translate a program in one language to a program in another language at fixed cost. If x was a sequence of all ones, a highly compressible sequence, the program would simply be a print statement in a loop. On the other extreme, if x were a random sequence with no structure then the only program that could generate it would contain the sequence itself. The size of the program, would be slightly larger than the sequence itself. Thus, there is a clear correspondence between the size of the smallest program that can generate a sequence and the amount of compression that can be obtained. Kolmogorov complexity seems to be the ideal measure to use in data compression. The problem is we do not know of any systematic way of computing or closely approximating Kolmogorov complexity. Clearly, any program that can generate a particular sequence is an upper bound for the Kolmogorov complexity of the sequence. However, we have no way of determining a lower bound. Thus, while the notion of Kolmogorov complexity is more satisfying theoretically than the notion of entropy when compressing sequences, in practice it is not yet as helpful. However, given the active interest in these ideas it is quite possible that they will result in more practical applications.

2.6 Minimum Description Length Principle

One of the more practical offshoots of Kolmogorov complexity is the minimum description length (MDL) principle. The first discoverer of Kolmogorov complexity, Ray Solomonoff, viewed the concept of a program that would generate a sequence as a way of modeling the data. Independent from Solomonoff but inspired nonetheless by the ideas of Kolmogorov complexity, Jorma Risannen in 1978 [13] developed the modeling approach commonly known as MDL.

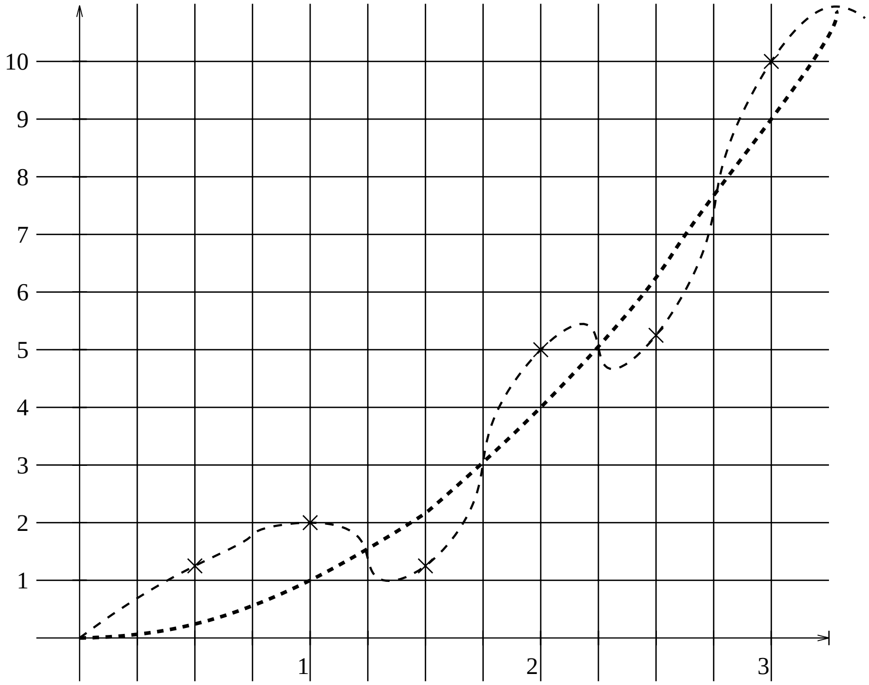


FIGURE 2. 5 **An example to illustrate the MDL principle.**

Let M_j be a model from a set of models $\mathcal{M}$ that attempt to characterize the structure in a sequence x. Let D_{M_j} be the number of bits required to describe the model M_j. For example, if the set of models $\mathcal{M}$ can be represented by a (possibly variable) number of coefficients, then the description of M_j would include the number of coefficients and the value of each coefficient. Let $R_{M_j}(x)$ be the number of bits required to represent x with respect to the model M_j. The minimum description length would be given by

$$\min_j (D_{M_j} + R_{M_j}(x))$$

Consider the example shown as Figure 2. 5, where the X's represent data values. Suppose the set of models $\mathcal{M}$ is the set of k^{th} order polynomials. We have also sketched two polynomials that could be used to model the data. Clearly, the higher-order polynomial does a much "better" job of modeling the data in the sense that the model exactly describes the data. To describe the higher order polynomial, we need to specify the value of each coefficient. The coefficients have to be exact if the polynomial is to exactly model the data requiring a large number of bits. The quadratic model, on the other hand, does not fit any of the data values. However, its description is very simple and the data values are either $+1$ or -1 away from the quadratic. So we could exactly represent the data by sending the coefficients of the quadratic (1, 0) and 1 bit per data value to indicate whether each data value is $+1$ or -1 away from the quadratic. In this case, from a compression point of view, using the worse model actually gives better compression.

2.7 Summary

In this chapter we learned some of the basic definitions of information theory. This was a rather brief visit, and we will revisit the subject in Chapter 8. However, the coverage in this chapter will be sufficient to take us through the next four chapters. The concepts introduced in this chapter allow us to estimate the number of bits we need to represent the output of a source given the probability model for the source. The process of assigning a binary representation to the output of a source is called coding. We have introduced the concepts of unique decodability and prefix codes, which we will use in the next two chapters when we describe various coding algorithms. We also looked, rather briefly, at different approaches to modeling. If we need to understand a model in more depth later in the book, we will devote more attention to it at that time. However, for the most part, the coverage of modeling in this chapter will be sufficient to understand methods described in the next four chapters.

Further Reading

1. A very readable book on information theory and its applications in a number of fields is *Symbols, Signals, and Noise—The Nature and Process of Communications*, by J.R. Pierce [14].
2. Another good introductory source for the material in this chapter is Chapter 6 of *Coding and Information Theory*, by R.W. Hamming [9].

3. Various models for text compression are described very nicely and in more detail in *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1].
4. For a more thorough and detailed account of information theory, the following books are especially recommended (the first two are my personal favorites): *Information Theory*, by R.B. Ash [15]; *Transmission of Information*, by R.M. Fano [16]; *Information Theory and Reliable Communication*, by R.G. Gallagher [11]; *Entropy and Information Theory*, by R.M. Gray [17]; *Elements of Information Theory*, by T.M. Cover and J.A. Thomas [3]; and *The Theory of Information and Coding*, by R.J. McEliece [6].
5. Kolmogorov complexity is addressed in detail in *An Introduction to Kolmogorov Complexity and Its Applications,* by M. Li and P. Vitanyi [18].
6. A very readable overview of Kolmogorov complexity in the context of lossless compression can be found in the chapter *Complexity Measures*, by S.R. Tate [19].
7. Various aspects of the minimum description length principle are discussed in *Advances in Minimum Description Length* edited by P. Grunwald, I.J. Myung, and M.A. Pitt [20]. Included in this book is a very nice introduction to the minimum description length principle by Peter Grunwald [21].

2.8 Projects and Problems

1. Suppose X is a random variable that takes on values from an M-letter alphabet. Show that $0 \leq H(X) \leq \log_2 M$.
2. Show that for the case where the elements of an observed sequence are *iid*, the entropy is equal to the first-order entropy.
3. Given an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$, find the first-order entropy in the following cases:
   (a) $P(a_1) = P(a_2) = P(a_3) = P(a_4) = \frac{1}{4}$.
   (b) $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, $P(a_3) = P(a_4) = \frac{1}{8}$.
   (c) $P(a_1) = 0.505$, $P(a_2) = \frac{1}{4}$, $P(a_3) = \frac{1}{8}$, and $P(a_4) = 0.12$.
4. Suppose we have a source with a probability model $P = \{p_0, p_1, \ldots, p_m\}$ and entropy H_P. Suppose we have another source with probability model $Q = \{q_0, q_1, \ldots, q_m\}$ and entropy H_Q, where

$$q_i = p_i \qquad i = 0, 1, \ldots, j-2, j+1, \ldots, m$$

and

$$q_j = q_{j-1} = \frac{p_j + p_{j-1}}{2}.$$

How is H_Q related to H_P (greater, equal, or less)? Prove your answer.

5. There are several image and speech files among the accompanying data sets.

   (a) Write a program to compute the first-order entropy of some of the image and speech files.

   (b) Pick one of the image files and compute its second-order entropy. Comment on the difference between the first- and second-order entropies.

   (c) Compute the entropy of the differences between neighboring pixels for the image you used in part (b). Comment on what you discover.

6. Conduct an experiment to see how well a model can describe a source.

   (a) Write a program that randomly selects letters from the 26-letter alphabet $\{a, b, \ldots, z\}$ and forms four-letter words. Form 100 such words and see how many of these words make sense.

   (b) Among the accompanying data sets is a file called `4letter.words`, which contains a list of four-letter words. Using this file, obtain a probability model for the alphabet. Now repeat part (a) generating the words using the probability model. To pick letters according to a probability model, construct the cumulative density function (*cdf*) $F_X(x)$ (see Appendix A for the definition of *cdf*). Using a uniform pseudorandom number generator to generate a value r, where $0 \leq r < 1$, pick the letter x_k if $F_X(x_k - 1) \leq r < F_X(x_k)$. Compare your results with those of part (a).

   (c) Repeat (b) using a single-letter context.

   (d) Repeat (b) using a two-letter context.

7. Determine whether the following codes are uniquely decodable:

   (a) $\{0, 01, 11, 111\}$

   (b) $\{0, 01, 110, 111\}$

   (c) $\{0, 10, 110, 111\}$

   (d) $\{1, 10, 110, 111\}$

8. Using a text file compute the probabilities of each letter p_i.

   (a) Assume that we need a codeword of length $\lceil \log_2 \frac{1}{p_i} \rceil$ to encode the letter i. Determine the number of bits needed to encode the file.

   (b) Compute the conditional probabilities $P(i/j)$ of a letter i given that the previous letter is j. Assume that we need $\lceil \log_2 \frac{1}{P(i/j)} \rceil$ to represent a letter i that follows a letter j. Determine the number of bits needed to encode the file.

3

Huffman Coding

3.1 Overview

In this chapter we describe a very popular coding algorithm called the Huffman coding algorithm. We first present a procedure for building Huffman codes when the probability model for the source is known, then a procedure for building codes when the source statistics are unknown. We also describe a few techniques for code design that are in some sense similar to the Huffman coding approach. Finally, we give some examples of using the Huffman code for image compression, audio compression, and text compression.

3.2 The Huffman Coding Algorithm

This technique was developed by David Huffman as part of a class assignment; the class was the first ever in the area of information theory and was taught by Robert Fano at MIT [22]. The codes generated using this technique or procedure are called *Huffman codes*. These codes are prefix codes and are optimum for a given model (set of probabilities).

The Huffman procedure is based on two observations regarding optimum prefix codes.

1. In an optimum code, symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently.
2. In an optimum code, the two symbols that occur least frequently will have the same length.

It is easy to see that the first observation is correct. If symbols that occur more often had codewords that were longer than the codewords for symbols that occurred less often, the average number of bits per symbol would be larger than if the conditions were reversed. Therefore, a code that assigns longer codewords to symbols that occur more frequently cannot be optimum.

To see why the second observation holds true, consider the following situation. Suppose an optimum code $\mathcal{C}$ exists in which the two codewords corresponding to the two least probable symbols do not have the same length. Suppose the longer codeword is k bits longer than the shorter codeword. Because this is a prefix code, the shorter codeword cannot be a prefix of the longer codeword. This means that even if we drop the last k bits of the longer codeword, the two codewords would still be distinct. As these codewords correspond to the least probable symbols in the alphabet, no other codeword can be longer than these codewords; therefore, there is no danger that the shortened codeword would become the prefix of some other codeword. Furthermore, by dropping these k bits we obtain a new code that has a shorter average length than $\mathcal{C}$. But this violates our initial contention that $\mathcal{C}$ is an optimal code. Therefore, for an optimal code the second observation also holds true.

The Huffman procedure is obtained by adding a simple requirement to these two observations. This requirement is that the codewords corresponding to the two lowest probability symbols differ only in the last bit. That is, if γ and δ are the two least probable symbols in an alphabet, if the codeword for γ was $\mathbf{m}*0$, the codeword for δ would be $\mathbf{m}*1$. Here $\mathbf{m}$ is a string of 1s and 0s, and $*$ denotes concatenation.

This requirement does not violate our two observations and leads to a very simple encoding procedure. We describe this procedure with the help of the following example.

Example 3.2.1: Design of a Huffman code

Let us design a Huffman code for a source that puts out letters from an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4, a_5\}$ with $P(a_1) = P(a_3) = 0.2$, $P(a_2) = 0.4$, and $P(a_4) = P(a_5) = 0.1$. The entropy for this source is 2.122 bits/symbol. To design the Huffman code, we first sort the letters in a descending probability order as shown in Table 3.1. Here $c(a_i)$ denotes the codeword for a_i.

TABLE 3.1 The initial five-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$
a_4	0.1	$c(a_4)$
a_5	0.1	$c(a_5)$

The two symbols with the lowest probability are a_4 and a_5. Therefore, we can assign their codewords as

$$c(a_4) = \alpha_1 * 0$$

$$c(a_5) = \alpha_1 * 1$$

where α_1 is a binary string, and $*$ denotes concatenation.

We now define a new alphabet A' with a four-letter alphabet a_1, a_2, a_3, a'_4, where a'_4 is composed of a_4 and a_5 and has a probability $P(a'_4) = P(a_4) + P(a_5) = 0.2$. We sort this new alphabet in descending order to obtain Table 3.2.

TABLE 3.2 The reduced four-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$
a'_4	0.2	α_1

In this alphabet, a_3 and a'_4 are the two letters at the bottom of the sorted list. We assign their codewords as

$$\begin{aligned} c(a_3) &= \alpha_2 * 0 \\ c(a'_4) &= \alpha_2 * 1 \end{aligned}$$

but $c(a'_4) = \alpha_1$. Therefore,

$$\alpha_1 = \alpha_2 * 1$$

which means that

$$\begin{aligned} c(a_4) &= \alpha_2 * 10 \\ c(a_5) &= \alpha_2 * 11. \end{aligned}$$

At this stage, we again define a new alphabet A'' that consists of three letters a_1, a_2, a'_3, where a'_3 is composed of a_3 and a'_4 and has a probability $P(a'_3) = P(a_3) + P(a'_4) = 0.4$. We sort this new alphabet in descending order to obtain Table 3.3.

TABLE 3.3 The reduced three-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a'_3	0.4	α_2
a_1	0.2	$c(a_1)$

In this case, the two least probable symbols are a_1 and a'_3. Therefore,

$$\begin{aligned} c(a'_3) &= \alpha_3 * 0 \\ c(a_1) &= \alpha_3 * 1. \end{aligned}$$

But $c(a'_3) = \alpha_2$. Therefore,

$$\alpha_2 = \alpha_3 * 0$$

which means that

$$\begin{aligned} c(a_3) &= \alpha_3 * 00 \\ c(a_4) &= \alpha_3 * 010 \\ c(a_5) &= \alpha_3 * 011. \end{aligned}$$

Again we define a new alphabet, this time with only two letters a''_3, a_2. Here a''_3 is composed of the letters a'_3 and a_1 and has probability $P(a''_3) = P(a'_3) + P(a_1) = 0.6$. We now have Table 3.4.

TABLE 3.4 The reduced two-letter alphabet.

Letter	Probability	Codeword
a''_3	0.6	α_3
a_2	0.4	$c(a_2)$

As we have only two letters, the codeword assignment is straightforward:

$$\begin{aligned} c(a''_3) &= 0 \\ c(a_2) &= 1 \end{aligned}$$

which means that $\alpha_3 = 0$, which in turn means that

$$\begin{aligned} c(a_1) &= 01 \\ c(a_3) &= 000 \\ c(a_4) &= 0010 \\ c(a_5) &= 0011 \end{aligned}$$

TABLE 3.5 Huffman code for the original five-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	1
a_1	0.2	01
a_3	0.2	000
a_4	0.1	0010
a_5	0.1	0011

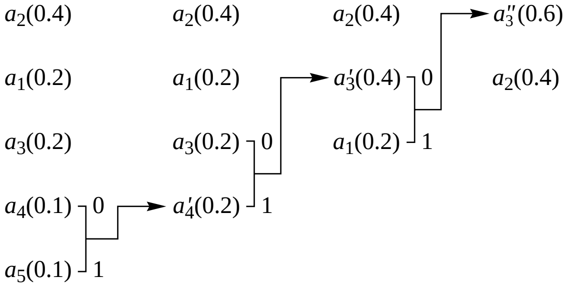


FIGURE 3.1 **The Huffman encoding procedure. The symbol probabilities are listed in parentheses.**

and the Huffman code is given by Table 3.5. The procedure can be summarized as shown in Figure 3.1. ◆

The average length for this code is

$$l = .4 \times 1 + .2 \times 2 + .2 \times 3 + .1 \times 4 + .1 \times 4 = 2.2 \text{ bits/symbol.}$$

A measure of the efficiency of this code is its *redundancy*—the difference between the entropy and the average length. In this case, the redundancy is 0.078 bits/symbol. The redundancy is zero when the probabilities are negative powers of two.

An alternative way of building a Huffman code is to use the fact that the Huffman code, by virtue of being a prefix code, can be represented as a binary tree in which the external nodes or leaves correspond to the symbols. The Huffman code for any symbol can be obtained by traversing the tree from the root node to the leaf corresponding to the symbol, adding a 0 to the codeword every time the traversal takes us over an upper branch and a 1 every time the traversal takes us over a lower branch.

We build the binary tree starting at the leaf nodes. We know that the codewords for the two symbols with smallest probabilities are identical except for the last bit. This means that the traversal from the root to the leaves corresponding to these two symbols must be the same except for the last step. This in turn means that the leaves corresponding to the two symbols with the lowest probabilities are offspring of the same node. Once we have connected the leaves corresponding to the symbols with the lowest probabilities to a single node, we treat this node as a symbol of a reduced alphabet. The probability of this symbol is the sum of the probabilities of its offspring. We can now sort the nodes corresponding to the reduced alphabet and apply the same rule to generate a parent node for the nodes corresponding to the two symbols in the reduced alphabet with lowest probabilities. Continuing in this manner, we end up with a single node, which is the root node. To obtain the code for each symbol, we traverse the tree from the root to each leaf node, assigning a 0 to the upper branch and a 1 to the lower branch. This procedure as applied to the alphabet of Example 3.2.1 is shown in Figure 3.2. Notice the similarity between Figures 3.1 and 3.2. This is not surprising, as they are a result of viewing the same procedure in two different ways.

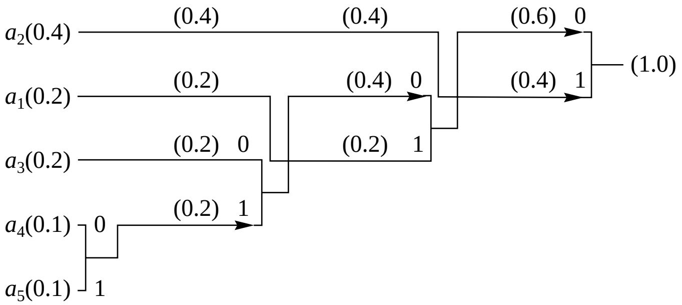


FIGURE 3.2 **Building the binary Huffman tree.**

3.2.1 Minimum Variance Huffman Codes

By performing the sorting procedure in a slightly different manner, we could have found a different Huffman code. In the first re-sort, we could place a'_4 higher in the list, as shown in Table 3.6.

Now combine a_1 and a_3 into a'_1, which has a probability of 0.4. Sorting the alphabet a_2, a'_4, a'_1 and putting a'_1 as far up the list as possible, we get Table 3.7. Finally, by combining a_2 and a'_4 and re-sorting, we get Table 3.8. If we go through the unbundling procedure, we get the codewords in Table 3.9. The procedure is summarized in Figure 3.3. The average length of the code is

$$l = .4 \times 2 + .2 \times 2 + .2 \times 2 + .1 \times 3 + .1 \times 3 = 2.2 \text{ bits/symbol.}$$

The two codes are identical in terms of their redundancy. However, the variance of the length of the codewords is significantly different. This can be clearly seen from Figure 3.4.

TABLE 3.6 **Reduced four-letter alphabet.**

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a'_4	0.2	α_1
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$

TABLE 3.7 **Reduced three-letter alphabet.**

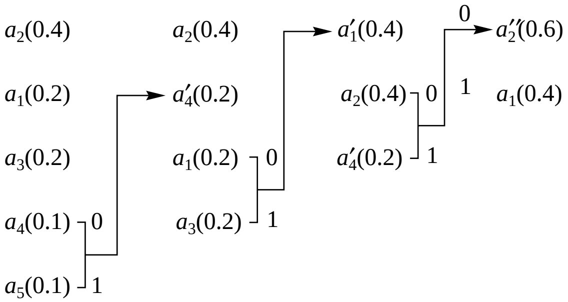
Letter	Probability	Codeword
a'_1	0.4	α_2
a_2	0.4	$c(a_2)$
a'_4	0.2	α_1

TABLE 3.8 Reduced two-letter alphabet.

Letter	Probability	Codeword
a'_2	0.6	α_3
a'_1	0.4	α_2

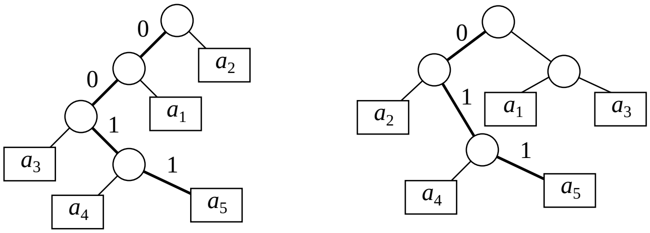
TABLE 3.9 Minimum variance Huffman code.

Letter	Probability	Codeword
a_1	0.2	10
a_2	0.4	00
a_3	0.2	11
a_4	0.1	010
a_5	0.1	011



$a_2(0.4)$
$a_1(0.2)$
$a_3(0.2)$
$a_4(0.1)$
$a_5(0.1)$
0
1
$a_2(0.4)$
$a'_4(0.2)$
$a_1(0.2)$
$a_3(0.2)$
0
1
$a'_1(0.4)$
$a_2(0.4)$
$a'_4(0.2)$
0
1
$a''_2(0.6)$
$a_1(0.4)$
0
1

FIGURE 3. 3 The minimum variance Huffman encoding procedure.



0
0
1
1
a_2
a_1
a_3
a_4
a_5
0
1
1
a_2
a_1
a_3
a_4
a_5

FIGURE 3. 4 Two Huffman trees corresponding to the same probabilities.

Remember that in many applications, although you might be using a variable-length code, the available transmission rate is generally fixed. For example, if we were going to transmit symbols from the alphabet we have been using at 10,000 symbols per second, we might ask for transmission capacity of 22,000 bits per second. This means that during each second the channel expects to receive 22,000 bits, no more and no less. As the bit generation rate will vary around 22,000 bits per second, the output of the source coder is generally fed into a buffer. The purpose of the buffer is to smooth out the variations in the bit generation rate. However, the buffer has to be of finite size, and the greater the variance in the codewords, the more difficult the buffer design problem becomes. Suppose that the source we are discussing generates a string of a_4s and a_5s for several seconds. If we are using the first code, this means that we will be generating bits at a rate of 40,000 bits per second. For each second, the buffer has to store 18,000 bits. On the other hand, if we use the second code, we would be generating 30,000 bits per second, and the buffer would have to store 8000 bits for every second this condition persisted. If we have a string of a_2s instead of a string of a_4s and a_5s, the first code would result in the generation of 10,000 bits per second. Remember that the channel will still be expecting 22,000 bits every second, so somehow we will have to make up a deficit of 12,000 bits per second. The same situation using the second code would lead to a deficit of 2000 bits per second. Thus, it seems reasonable to elect to use the second code instead of the first. To obtain the Huffman code with minimum variance, we always put the combined letter as high in the list as possible.

3.2.2 Optimality of Huffman Codes ★

The optimality of Huffman codes can be proven rather simply by first writing down the necessary conditions that an optimal code has to satisfy and then showing that satisfying these conditions necessarily leads to designing a Huffman code. The proof we present here is based on the proof shown in [16] and is obtained for the binary case (for a more general proof, see [16]).

The necessary conditions for an optimal variable-length binary code are as follows:

- **Condition 1:** Given any two letters a_j and a_k, if $P[a_j] \geq P[a_k]$, then $l_j \leq l_k$, where l_j is the number of bits in the codeword for a_j.
- **Condition 2:** The two least probable letters have codewords with the same maximum length l_m.

We have provided the justification for these two conditions in the opening sections of this chapter.

- **Condition 3:** In the tree corresponding to the optimum code, there must be two branches stemming from each intermediate node.

If there were any intermediate node with only one branch coming from that node, we could remove it without affecting the decipherability of the code while reducing its average length.

- **Condition 4:** Suppose we change an intermediate node into a leaf node by combining all the leaves descending from it into a composite word of a reduced alphabet. Then, if the original tree was optimal for the original alphabet, the reduced tree is optimal for the reduced alphabet.

If this condition were not satisfied, we could find a code with smaller average code length for the reduced alphabet and then simply expand the composite word again to get a new code tree that would have a shorter average length than our original "optimum" tree. This would contradict our statement about the optimality of the original tree.

In order to satisfy conditions 1, 2, and 3, the two least probable letters would have to be assigned codewords of maximum length l_m. Furthermore, the leaves corresponding to these letters arise from the same intermediate node. This is the same as saying that the codewords for these letters are identical except for the last bit. Consider the common prefix as the codeword for the composite letter of a reduced alphabet. Since the code for the reduced alphabet needs to be optimum for the code of the original alphabet to be optimum, we follow the same procedure again. To satisfy the necessary conditions, the procedure needs to be iterated until we have a reduced alphabet of size one. But this is exactly the Huffman procedure. Therefore, the necessary conditions above, which are all satisfied by the Huffman procedure, are also sufficient conditions.

3.2.3 Length of Huffman Codes ★

We have said that the Huffman coding procedure generates an optimum code, but we have not said what the average length of an optimum code is. The length of any code will depend on a number of things, including the size of the alphabet and the probabilities of individual letters. In this section we will show that the optimal code for a source $\mathcal{S}$, hence the Huffman code for the source $\mathcal{S}$, has an average code length $\bar{l}$ bounded below by the entropy and bounded above by the entropy plus 1 bit. In other words,

$$H(\mathcal{S}) \leq \bar{l} < H(\mathcal{S}) + 1. \tag{3.1}$$

In order for us to do this, we will need to use the Kraft-McMillan inequality introduced in Chapter 2. Recall that the first part of this result, due to McMillan, states that if we have a uniquely decodable code $\mathcal{C}$ with K codewords of length $\{l_i\}_{i=1}^{K}$, then the following inequality holds:

$$\sum_{i=1}^{K} 2^{-l_i} \leq 1. \tag{3.2}$$

Example 3.2.2:

Examining the code generated in Example 3.2.1 (Table 3.5), the lengths of the codewords are $\{1, 2, 3, 4, 4\}$. Substituting these values into the left-hand side of Equation (3.2), we get

$$2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-4} = 1$$

which satisfies the Kraft-McMillan inequality.

If we use the minimum variance code (Table 3.9), the lengths of the codewords are $\{2, 2, 2, 3, 3\}$. Substituting these values into the left-hand side of Equation (3.2), we get

$$2^{-2} + 2^{-2} + 2^{-2} + 2^{-3} + 2^{-3} = 1$$

which again satisfies the inequality. ♦

The second part of this result, due to Kraft, states that if we have a sequence of positive integers $\{l_i\}_{i=1}^{K}$, which satisfies (3.2), then there exists a uniquely decodable code whose codeword lengths are given by the sequence $\{l_i\}_{i=1}^{K}$.

Using this result, we will now show the following:

1. The average codeword length $\bar{l}$ of an optimal code for a source $\mathcal{S}$ is greater than or equal to $H(\mathcal{S})$.
2. The average codeword length $\bar{l}$ of an optimal code for a source $\mathcal{S}$ is strictly less than $H(\mathcal{S})+1$.

For a source $\mathcal{S}$ with alphabet $\mathcal{A} = \{a_1, a_2, \ldots a_K\}$, and probability model $\{P(a_1), P(a_2), \ldots, P(a_K)\}$, the average codeword length is given by

$$\bar{l} = \sum_{i=1}^{K} P(a_i) l_i.$$

Therefore, we can write the difference between the entropy of the source $H(\mathcal{S})$ and the average length as

$$\begin{aligned}
H(\mathcal{S}) - \bar{l} &= -\sum_{i=1}^{K} P(a_i) \log_2 P(a_i) - \sum_{i=1}^{K} P(a_i) l_i \\
&= \sum_{i=1}^{K} P(a_i) \left(\log_2 \left[\frac{1}{P(a_i)} \right] - l_i \right) \\
&= \sum_{i=1}^{K} P(a_i) \left(\log_2 \left[\frac{1}{P(a_i)} \right] - \log_2 [2^{l_i}] \right) \\
&= \sum_{i=1}^{K} P(a_i) \log_2 \left[\frac{2^{-l_i}}{P(a_i)} \right] \\
&\leq \log_2 \left[\sum_{i=1}^{K} 2^{-l_i} \right].
\end{aligned}$$

The last inequality is obtained using Jensen's inequality, which states that if $f(x)$ is a concave (convex cap, convex $\cap$) function, then $E[f(X)] \leq f(E[X])$. The log function is a concave function.

As the code is an optimal code $\sum_{i=1}^{K} 2^{-l_i} \leq 1$, therefore

$$H(\mathcal{S}) - \bar{l} \leq 0. \qquad (3.3)$$

We will prove the upper bound by showing that there exists a uniquely decodable code with average codeword length $H(\mathcal{S})+1$. Therefore, if we have an optimal code, this code must have an average length that is less than or equal to $H(\mathcal{S})+1$.

Given a source, alphabet, and probability model as before, define

$$l_i = \left\lceil \log_2 \frac{1}{P(a_i)} \right\rceil$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x. For example, $\lceil 3.3 \rceil = 4$ and $\lceil 5 \rceil = 5$. Therefore,

$$\lceil x \rceil = x + \epsilon \qquad \text{where } 0 \leq \epsilon < 1.$$

Therefore,

$$\log_2 \frac{1}{P(a_i)} \leq l_i < \log_2 \frac{1}{P(a_i)} + 1. \tag{3.4}$$

From the left inequality of (3.4) we can see that

$$2^{-l_i} \leq P(a_i).$$

Therefore,

$$\sum_{i=1}^{K} 2^{-l_i} \leq \sum_{i=1}^{K} P(a_i) = 1$$

and by the Kraft-McMillan inequality there exists a uniquely decodable code with codeword lengths $\{l_i\}$. The average length of this code can be upper-bounded by using the right inequality of (3.4):

$$\bar{l} = \sum_{i=1}^{K} P(a_i) l_i < \sum_{i=1}^{K} P(a_i) \left[\log_2 \frac{1}{P(a_i)} + 1 \right]$$

or

$$\bar{l} < H(\mathcal{S}) + 1.$$

We can see from the way the upper bound was derived that this is a rather loose upper bound. In fact, it can be shown that if $p_{\max}$ is the largest probability in the probability model, then for $p_{\max} \geq 0.5$, the upper bound for the Huffman code is $H(\mathcal{S}) + p_{\max}$, while for $p_{\max} < 0.5$, the upper bound is $H(\mathcal{S}) + p_{\max} + 0.086$. Obviously, this is a much tighter bound than the one we derived above. The derivation of this bound takes some time (see [23] for details).

3.2.4 Extended Huffman Codes ★

In applications where the alphabet size is large, $p_{\max}$ is generally quite small, and the amount of deviation from the entropy, especially in terms of a percentage of the rate, is quite small. However, in cases where the alphabet is small and the probability of occurrence of the different letters is skewed, the value of $p_{\max}$ can be quite large and the Huffman code can become rather inefficient when compared to the entropy.

Example 3.2.3:

Consider a source that puts out *iid* letters from the alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ with the probability model $P(a_1) = 0.8$, $P(a_2) = 0.02$, and $P(a_3) = 0.18$. The entropy for this source is 0.816 bits/symbol. A Huffman code for this source is shown in Table 3.10.

TABLE 3.10 **Huffman code for the alphabet $\mathcal{A}$.**

Letter	Codeword
a_1	0
a_2	11
a_3	10

The average length for this code is 1.2 bits/symbol. The difference between the average code length and the entropy, or the redundancy, for this code is 0.384 bits/symbol, which is 47% of the entropy. This means that to code this sequence we would need 47% more bits than the minimum required. ◆

We can sometimes reduce the coding rate by blocking more than one symbol together. To see how this can happen, consider a source S that emits a sequence of letters from an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$. Each element of the sequence is generated independently of the other elements in the sequence. The entropy for this source is given by

$$H(S) = -\sum_{i=1}^{m} P(a_i) \log_2 P(a_i).$$

We know that we can generate a Huffman code for this source with rate R such that

$$H(S) \leq R < H(S) + 1. \tag{3.5}$$

We have used the looser bound here; the same argument can be made with the tighter bound. Notice that we have used "rate R" to denote the number of bits per symbol. This is a standard convention in the data compression literature. However, in the communication literature, the word "rate" often refers to the number of bits per second.

Suppose we now encode the sequence by generating one codeword for every n symbols. As there are m^n combinations of n symbols, we will need m^n codewords in our Huffman code. We could generate this code by viewing the m^n symbols as letters of an *extended alphabet*

$$\mathcal{A}^{(n)} = \{\overbrace{a_1 a_1 \ldots a_1}^{n \text{ times}}, a_1 a_1 \ldots a_2, \ldots, a_1 a_1 \ldots a_m, a_1 a_1 \ldots a_2 a_1, \ldots, a_m a_m \ldots a_m\}$$

from a source $S^{(n)}$. Let us denote the rate for the new source as $R^{(n)}$. Then we know that

$$H(S^{(n)}) \leq R^{(n)} < H(S^{(n)}) + 1. \tag{3.6}$$

$R^{(n)}$ is the number of bits required to code n symbols. Therefore, the number of bits required per symbol, R, is given by

$$R = \frac{1}{n} R^{(n)}.$$

The number of bits per symbol can be bounded as

$$\frac{H(S^{(n)})}{n} \leq R < \frac{H(S^{(n)})}{n} + \frac{1}{n}.$$

In order to compare this to (3.5), and see the advantage we get from encoding symbols in blocks instead of one at a time, we need to express $H(S^{(n)})$ in terms of $H(S)$. This turns out to be a relatively easy (although somewhat messy) thing to do.

$$\begin{aligned}
H(S^{(n)}) &= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_1}, a_{i_2}, \dots a_{i_n}) \log[P(a_{i_1}, a_{i_2}, \dots a_{i_n})] \\
&= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_2})\dots P(a_{i_n}) \log[P(a_{i_1})P(a_{i_2})\dots P(a_{i_n})] \\
&= -\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_2})\dots P(a_{i_n}) \sum_{j=1}^{n}\log[P(a_{i_j})] \\
&= -\sum_{i_1=1}^{m} P(a_{i_1})\log[P(a_{i_1})]\left\{\sum_{i_2=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_2})\dots P(a_{i_n})\right\} \\
&\quad -\sum_{i_2=1}^{m} P(a_{i_2})\log[P(a_{i_2})]\left\{\sum_{i_1=1}^{m}\sum_{i_3=1}^{m}\dots\sum_{i_n=1}^{m} P(a_{i_1})P(a_{i_3})\dots P(a_{i_n})\right\} \\
&\ \vdots \\
&\quad -\sum_{i_n=1}^{m} P(a_{i_n})\log[P(a_{i_n})]\left\{\sum_{i_1=1}^{m}\sum_{i_2=1}^{m}\dots\sum_{i_{n-1}=1}^{m} P(a_{i_1})P(a_{i_2})\dots P(a_{i_{n-1}})\right\}
\end{aligned}$$

The $n-1$ summations in braces in each term sum to one. Therefore,

$$\begin{aligned}
H(S^{(n)}) &= -\sum_{i_1=1}^{m} P(a_{i_1})\log[P(a_{i_1})] - \sum_{i_2=1}^{m} P(a_{i_2})\log[P(a_{i_2})] - \dots - \sum_{i_n=1}^{m} P(a_{i_n})\log[P(a_{i_n})] \\
&= nH(S)
\end{aligned}$$

and we can write (3.6) as

$$H(S) \leq R \leq H(S) + \frac{1}{n}. \tag{3.7}$$

Comparing this to (3.5), we can see that by encoding the output of the source in longer blocks of symbols we are *guaranteed* a rate closer to the entropy. Note that all we are talking about here is a bound or guarantee about the rate. As we have seen in the previous chapter, there are a number of situations in which we can achieve a rate *equal* to the entropy with a block length of one!

Example 3.2.4:

For the source described in the previous example, instead of generating a codeword for every symbol, we will generate a codeword for every *two* symbols. If we look at the source sequence two at a time, the number of possible symbol pairs, or size of the extended alphabet, is $3^2 = 9$. The extended alphabet, probability model, and Huffman code for this example are shown in Table 3.11.

TABLE 3.11 **The extended alphabet and corresponding Huffman code.**

Letter	Probability	Code
a_1a_1	0.64	0
a_1a_2	0.016	10101
a_1a_3	0.144	11
a_2a_1	0.016	101000
a_2a_2	0.0004	10100101
a_2a_3	0.0036	1010011
a_3a_1	0.1440	100
a_3a_2	0.0036	10100100
a_3a_3	0.0324	1011

The average codeword length for this extended code is 1.7228 bits/symbol. However, each symbol in the extended alphabet corresponds to two symbols from the original alphabet. Therefore, in terms of the original alphabet, the average codeword length is $1.7228/2 = 0.8614$ bits/symbol. This redundancy is about 0.045 bits/symbol, which is only about 5.5% of the entropy. ♦

We see that by coding blocks of symbols together we can reduce the redundancy of Huffman codes. In the previous example, two symbols were blocked together to obtain a rate reasonably close to the entropy. Blocking two symbols together means the alphabet size goes from m to m^2, where m was the size of the initial alphabet. In this case, m was three, so the size of the extended alphabet was nine. This size is not an excessive burden for most applications. However, if the probabilities of the symbols were more unbalanced, then it would require blocking many more symbols together before the redundancy lowered to acceptable levels. As we block more and more symbols together, the size of the alphabet grows exponentially, and the Huffman coding scheme becomes impractical. Under these conditions, we need to look at techniques other than Huffman coding. One approach that is very useful in these conditions is *arithmetic coding*. We will discuss this technique in some detail in the next chapter.

3.3 Nonbinary Huffman Codes ★

The binary Huffman coding procedure can be easily extended to the nonbinary case where the code elements come from an m-ary alphabet, and m is not equal to two. Recall that we obtained the Huffman algorithm based on the observations that in an optimum binary prefix code

1. symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently, and
2. the two symbols that occur least frequently will have the same length,

and the requirement that the two symbols with the lowest probability differ only in the last position.

We can obtain a nonbinary Huffman code in almost exactly the same way. The obvious thing to do would be to modify the second observation to read: "The m symbols that occur least frequently will have the same length," and also modify the additional requirement to read "The m symbols with the lowest probability differ only in the last position."

However, we run into a small problem with this approach. Consider the design of a ternary Huffman code for a source with a six-letter alphabet. Using the rules described above, we would first combine the three letters with the lowest probability into a composite letter. This would give us a reduced alphabet with four letters. However, combining the three letters with lowest probability from this alphabet would result in a further reduced alphabet consisting of only two letters. We have three values to assign and only two letters. Instead of combining three letters at the beginning, we could have combined two letters. This would result in a reduced alphabet of size five. If we combined three letters from this alphabet, we would end up with a final reduced alphabet size of three. Finally, we could combine two letters in the second step, which would again result in a final reduced alphabet of size three. Which alternative should we choose?

Recall that the symbols with lowest probability will have the longest codeword. Furthermore, all the symbols that we combine together into a composite symbol will have codewords of the same length. This means that all letters we combine together at the very first stage will have codewords that have the same length, and these codewords will be the longest of all the codewords. This being the case, if at some stage we are allowed to combine less than m symbols, the logical place to do this would be in the very first stage.

In the general case of an m-ary code and an M-letter alphabet, how many letters should we combine in the first phase? Let m' be the number of letters that are combined in the first phase. Then m' is the number between two and m, which is equal to M modulo $(m-1)$.

Example 3.3.1:

Generate a ternary Huffman code for a source with a six-letter alphabet and a probability model $P(a_1) = P(a_3) = P(a_4) = 0.2$, $P(a_5) = 0.25$, $P(a_6) = 0.1$, and $P(a_2) = 0.05$. In this case $m = 3$, therefore m' is either 2 or 3.

$$6 \pmod 2 = 0, \qquad 2 \pmod 2 = 0, \qquad 3 \pmod 2 = 1$$

Since 6 (mod 2) = 2 (mod 2), $m' = 2$. Sorting the symbols in probability order results in Table 3.12.

TABLE 3.12 **Sorted six-letter alphabet.**

Letter	Probability	Codeword
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$
a_3	0.20	$c(a_3)$
a_4	0.20	$c(a_4)$
a_6	0.10	$c(a_6)$
a_2	0.05	$c(a_2)$

As m' is 2, we can assign the codewords of the two symbols with lowest probability as

$$c(a_6) = \alpha_1 * 0$$

$$c(a_2) = \alpha_1 * 1$$

where α_1 is a ternary string and * denotes concatenation. The reduced alphabet is shown in Table 3.13.

TABLE 3.13 **Reduced five-letter alphabet.**

Letter	Probability	Codeword
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$
a_3	0.20	$c(a_3)$
a_4	0.20	$c(a_4)$
a'_6	0.15	α_1

Now we combine the three letters with the lowest probability into a composite letter a'_3 and assign their codewords as

$$c(a_3) = \alpha_2 * 0$$

$$c(a_4) = \alpha_2 * 1$$

$$c(a'_6) = \alpha_2 * 2.$$

But $c(a'_6) = \alpha_1$. Therefore,

$$\alpha_1 = \alpha_2 * 2$$

which means that

$$c(a_6) = \alpha_2 * 20$$
$$c(a_2) = \alpha_2 * 21.$$

Sorting the reduced alphabet, we have Table 3.14. Thus, $\alpha_2 = 0$, $c(a_5) = 1$, and $c(a_1) = 2$. Substituting for α_2, we get the codeword assignments in Table 3.15.

TABLE 3.14 **Reduced three-letter alphabet.**

Letter	Probability	Codeword
a'_3	0.45	α_2
a_5	0.25	$c(a_5)$
a_1	0.20	$c(a_1)$

TABLE 3.15 **Ternary code for six-letter alphabet.**

Letter	Probability	Codeword
a_1	0.20	2
a_2	0.05	021
a_3	0.20	00
a_4	0.20	01
a_5	0.25	1
a_6	0.10	020

The tree corresponding to this code is shown in Figure 3.5. Notice that at the lowest level of the tree we have only two codewords. If we had combined three letters at the first step, and combined two letters at a later step, the lowest level would have contained three codewords and a longer average code length would result (see Problem 7).

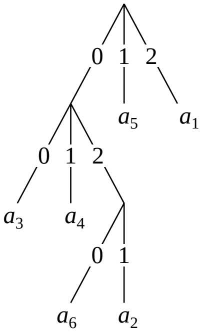


FIGURE 3.5 **Code tree for the nonbinary Huffman code.** ◆

3.4 Adaptive Huffman Coding

Huffman coding requires knowledge of the probabilities of the source sequence. If this knowledge is not available, Huffman coding becomes a two-pass procedure: the statistics are collected in the first pass, and the source is encoded in the second pass. In order to convert this algorithm into a one-pass procedure, Faller [24] and Gallagher [23] independently developed adaptive algorithms to construct the Huffman code based on the statistics of the symbols already encountered. These were later improved by Knuth [25] and Vitter [26].

Theoretically, if we wanted to encode the $(k+1)$-th symbol using the statistics of the first k symbols, we could recompute the code using the Huffman coding procedure each time a symbol is transmitted. However, this would not be a very practical approach due to the large amount of computation involved—hence, the adaptive Huffman coding procedures.

The Huffman code can be described in terms of a binary tree similar to the ones shown in Figure 3.4. The squares denote the external nodes or leaves and correspond to the symbols in the source alphabet. The codeword for a symbol can be obtained by traversing the tree from the root to the leaf corresponding to the symbol, where 0 corresponds to a left branch and 1 corresponds to a right branch. In order to describe how the adaptive Huffman code works, we add two other parameters to the binary tree: the *weight* of each leaf, which is written as a number inside the node, and a *node number*. The weight of each external node is simply the number of times the symbol corresponding to the leaf has been encountered. The weight of each internal node is the sum of the weights of its offspring. The node number y_i is a unique number assigned to each internal and external node. If we have an alphabet of size n, then the $2n-1$ internal and external nodes can be numbered as $y_1, \ldots, y_{2n-1}$ such that if x_j is the weight of node y_j, we have $x_1 \leq x_2 \leq \cdots \leq x_{2n-1}$. Furthermore, the nodes y_{2j-1} and y_{2j} are offspring of the same parent node, or siblings, for $1 \leq j < n$, and the node number for the parent node is greater than y_{2j-1} and y_{2j}. These last two characteristics are called the *sibling property*, and any tree that possesses this property is a Huffman tree [23].

In the adaptive Huffman coding procedure, neither transmitter nor receiver knows anything about the statistics of the source sequence at the start of transmission. The tree at both the transmitter and the receiver consists of a single node that corresponds to all symbols not yet transmitted (NYT) and has a weight of zero. As transmission progresses, nodes corresponding to symbols transmitted will be added to the tree, and the tree is reconfigured using an update procedure. Before the beginning of transmission, a fixed code for each symbol is agreed upon between transmitter and receiver. A simple (short) code is as follows:

If the source has an alphabet $(a_1, a_2, \ldots, a_m)$ of size m, then pick e and r such that $m = 2^e + r$ and $0 \leq r < 2^e$. The letter a_k is encoded as the $(e+1)$-bit binary representation of $k-1$, if $1 \leq k \leq 2r$; else, a_k is encoded as the e-bit binary representation of $k-r-1$. For example, suppose $m = 26$, then $e = 4$, and $r = 10$. The symbol a_1 is encoded as 00000, the symbol a_2 is encoded as 00001, and the symbol a_{22} is encoded as 1011.

When a symbol is encountered for the first time, the code for the NYT node is transmitted, followed by the fixed code for the symbol. A node for the symbol is then created, and the symbol is taken out of the NYT list.

Both transmitter and receiver start with the same tree structure. The updating procedure used by both transmitter and receiver is identical. Therefore, the encoding and decoding processes remain synchronized.

3.4.1 Update Procedure

The update procedure requires that the nodes be in a fixed order. This ordering is preserved by numbering the nodes. The largest node number is given to the root of the tree, and the smallest number is assigned to the NYT node. The numbers from the NYT node to the root of the tree are assigned in increasing order from left to right, and from lower level to upper level. The set of nodes with the same weight makes up a *block*. Figure 3.6 is a flowchart of the updating procedure.

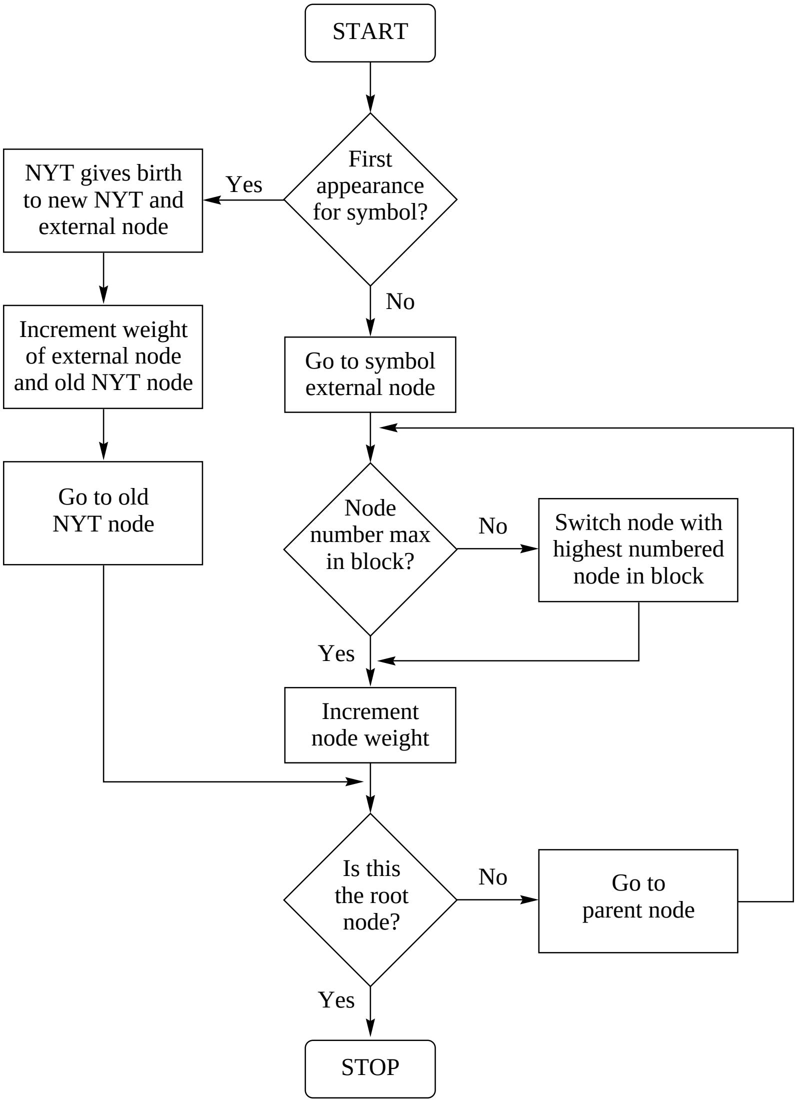

START
First appearance for symbol?
Yes
NYT gives birth to new NYT and external node
Increment weight of external node and old NYT node
Go to old NYT node
No
Go to symbol external node
Node number max in block?
No
Switch node with highest numbered node in block
Yes
Increment node weight
Is this the root node?
No
Go to parent node
Yes
STOP

FIGURE 3.6 **Update procedure for the adaptive Huffman coding algorithm.**

The function of the update procedure is to preserve the sibling property. In order that the update procedures at the transmitter and receiver both operate with the same information, the tree at the transmitter is updated after each symbol is encoded, and the tree at the receiver is updated after each symbol is decoded. The procedure operates as follows:

After a symbol has been encoded or decoded, the external node corresponding to the symbol is examined to see if it has the largest node number in its block. If the external node does not have the largest node number, it is exchanged with the node that has the largest node number in the block, as long as the node with the higher number is not the parent of the node being updated. The weight of the external node is then incremented. If we did not exchange the nodes before the weight of the node is incremented, it is very likely that the ordering required by the sibling property would be destroyed. Once we have incremented the weight of the node, we have adapted the Huffman tree at that level. We then turn our attention to the next level by examining the parent node of the node whose weight was incremented to see if it has the largest number in its block. If it does not, it is exchanged with the node with the largest number in the block. Again, an exception to this is when the node with the higher node number is the parent of the node under consideration. Once an exchange has taken place (or it has been determined that there is no need for an exchange), the weight of the parent node is incremented. We then proceed to a new parent node and the process is repeated. This process continues until the root of the tree is reached.

If the symbol to be encoded or decoded has occurred for the first time, a new external node is assigned to the symbol and a new NYT node is appended to the tree. Both the new external node and the new NYT node are offsprings of the old NYT node. We increment the weight of the new external node by one. As the old NYT node is the parent of the new external node, we increment its weight by one and then go on to update all the other nodes until we reach the root of the tree.

Example 3.4.1: Update procedure

Assume we are encoding the message [a a r d v a r k], where our alphabet consists of the 26 lowercase letters of the English alphabet.

The updating process is shown in Figure 3.7. We begin with only the NYT node. The total number of nodes in this tree will be $2 \times 26 - 1 = 51$, so we start numbering backwards from 51 with the number of the root node being 51. The first letter to be transmitted is a. As a does not yet exist in the tree, we send a binary code 00000 for a and then add a to the tree. The NYT node gives birth to a new NYT node and a terminal node corresponding to a. The weight of the terminal node will be higher than the NYT node, so we assign the number 49 to the NYT node and 50 to the terminal node corresponding to the letter a. The second letter to be transmitted is also a. This time the transmitted code is 1. The node corresponding to a has the highest number (if we do not consider its parent), so we do not need to swap nodes. The next letter to be transmitted is r. This letter does not have a corresponding node on the tree, so we send the codeword for the NYT node, which is 0 followed by the index of r, which is 10001. The NYT node gives birth to a new NYT node and an external node corresponding to r. Again, no update is required. The next letter to be transmitted is d, which is also being sent for the first time. We again send the code for

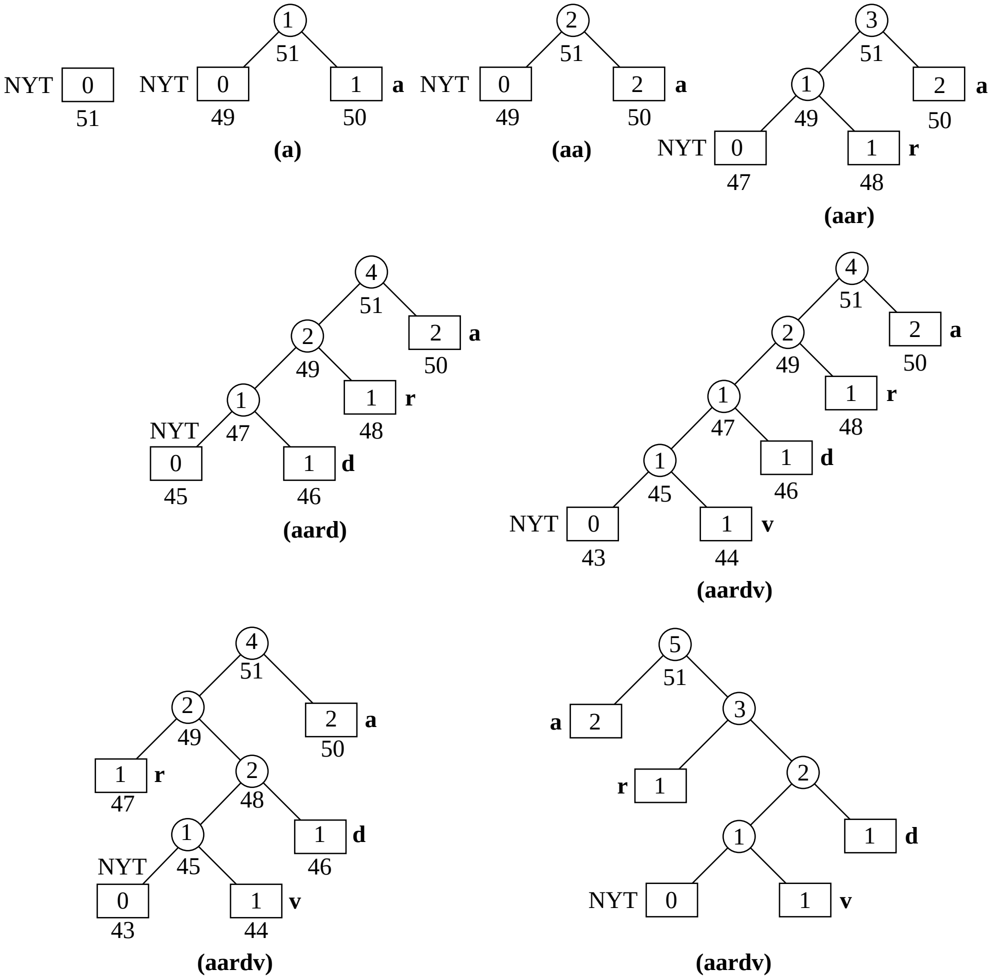

NYT
0
51
1
51
NYT
0
49
1
a
50
(a)
2
51
NYT
0
49
2
a
50
(aa)
3
51
1
49
2
a
50
NYT
0
47
1
r
48
(aar)
4
51
2
49
2
a
50
1
47
1
r
48
NYT
0
45
1
d
46
(aard)
4
51
2
49
2
a
50
1
47
1
r
48
1
45
1
d
46
NYT
0
43
1
v
44
(aardv)
4
51
2
49
2
a
50
1
r
47
2
48
1
45
1
d
46
NYT
0
43
1
v
44
(aardv)
5
51
a
2
3
r
1
2
1
1
d
NYT
0
1
v
(aardv)

FIGURE 3.7 **Adaptive Huffman tree after [a a r d v] is processed.**

the NYT node, which is now 00 followed by the index for *d*, which is 00011. The NYT node again gives birth to two new nodes. However, an update is still not required. This changes with the transmission of the next letter, *v*, which has also not yet been encountered. Nodes 43 and 44 are added to the tree, with 44 as the terminal node corresponding to *v*. We examine the grandparent node of *v* (node 47) to see if it has the largest number in its block. As it does not, we swap it with node 48, which has the largest number in its block. We then increment node 48 and move to its parent, which is node 49. In the block containing node 49, the largest number belongs to node 50. Therefore, we swap nodes 49 and 50 and then increment node 50. We then move to the parent node of node 50, which is node 51. As this is the root node, all we do is increment node 51.

◆

3.4.2 Encoding Procedure

The flowchart for the encoding procedure is shown in Figure 3.8. Initially, the tree at both the encoder and decoder consists of a single node, the NYT node. Therefore, the codeword for the very first symbol that appears is a previously agreed-upon fixed code. After the very first symbol, whenever we have to encode a symbol that is being encountered for the first time, we send the code for the NYT node, followed by the previously agreed-upon fixed code for the symbol. The code for the NYT node is obtained by traversing the Huffman tree from the root to the NYT node. This alerts the receiver to the fact that the symbol whose code follows does not as yet have a node in the Huffman tree. If a symbol to be encoded has a corresponding node in the tree, then the code for the symbol is generated by traversing the tree from the root to the external node corresponding to the symbol.

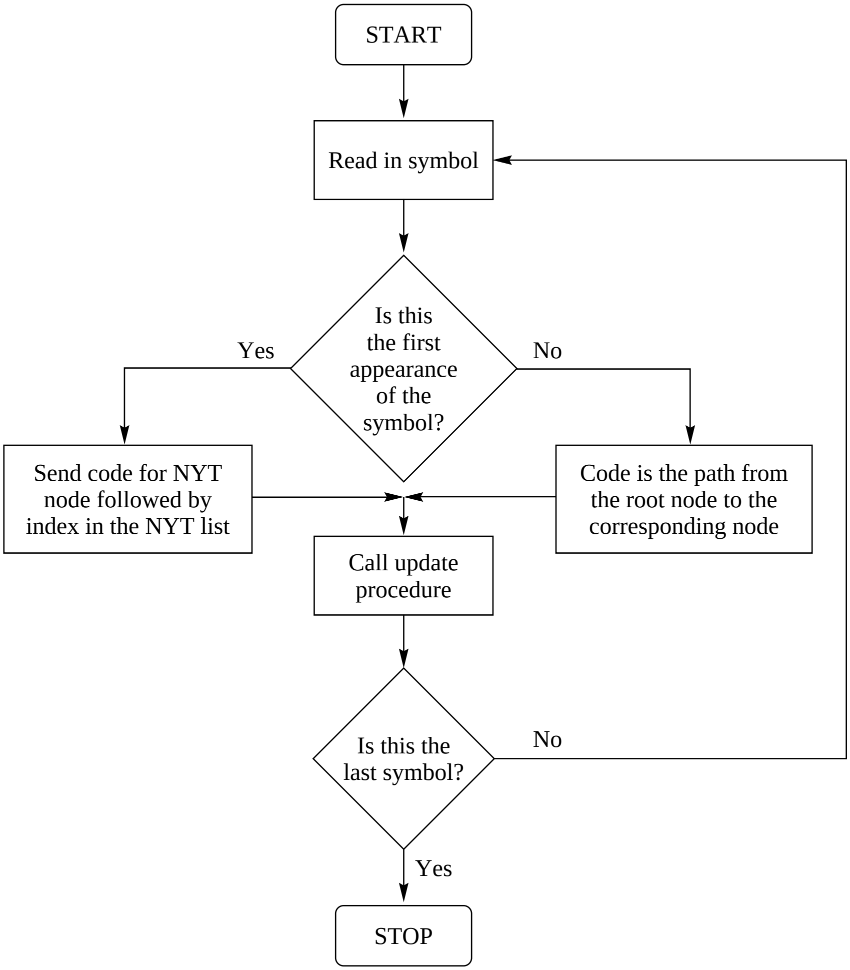

START
Read in symbol
Is this the first appearance of the symbol?
Yes
No
Send code for NYT node followed by index in the NYT list
Code is the path from the root node to the corresponding node
Call update procedure
Is this the last symbol?
No
Yes
STOP

FIGURE 3.8 Flowchart of the encoding procedure.

To see how the coding operation functions, we use the same example that was used to demonstrate the update procedure.

Example 3.4.2: Encoding procedure

In Example 3.4.1 we used an alphabet consisting of 26 letters. In order to obtain our prearranged code, we have to find m and e such that $2^e + r = 26$, where $0 \leq r < 2^e$. It is easy to see that the values of $e = 4$ and $r = 10$ satisfy this requirement.

The first symbol encoded is the letter a. As a is the first letter of the alphabet, $k = 1$. As 1 is less than 20, a is encoded as the 5-bit binary representation of $k - 1$, or 0, which is 00000. The Huffman tree is then updated as shown in the figure. The NYT node gives birth to an external node corresponding to the element a and a new NYT node. As a has occurred once, the external node corresponding to a has a weight of one. The weight of the NYT node is zero. The internal node also has a weight of one, as its weight is the sum of the weights of its offspring. The next symbol is again a. As we have an external node corresponding to symbol a, we simply traverse the tree from the root node to the external node corresponding to a in order to find the codeword. This traversal consists of a single right branch. Therefore, the Huffman code for the symbol a is 1.

After the code for a has been transmitted, the weight of the external node corresponding to a is incremented, as is the weight of its parent. The third symbol to be transmitted is r. As this is the first appearance of this symbol, we send the code for the NYT node followed by the previously arranged binary representation for r. If we traverse the tree from the root to the NYT node, we get a code of 0 for the NYT node. The letter r is the 18th letter of the alphabet; therefore, the binary representation of r is 10001. The code for the symbol r becomes 010001. The tree is again updated as shown in the figure, and the coding process continues with symbol d. Using the same procedure for d, the code for the NYT node, which is now 00, is sent, followed by the index for d, resulting in the codeword 0000011. The next symbol v is the 22nd symbol in the alphabet. As this is greater than 20, we send the code for the NYT node followed by the 4-bit binary representation of $22 - 10 - 1 = 11$. The code for the NYT node at this stage is 000, and the 4-bit binary representation of 11 is 1011; therefore, v is encoded as 0001011. The next symbol is a, for which the code is 0, and the encoding proceeds. ◆

3.4.3 Decoding Procedure

The flowchart for the decoding procedure is shown in Figure 3.9. As we read in the received binary string, we traverse the tree in a manner identical to that used in the encoding procedure. Once a leaf is encountered, the symbol corresponding to that leaf is decoded. If the leaf is the NYT node, then we check the next e bits to see if the resulting number is less than r. If it is less than r, we read in another bit to complete the code for the symbol. The index for the symbol is obtained by adding one to the decimal number corresponding to the e- or $e + 1$-bit binary string. Once the symbol has been decoded, the tree is updated and the next received bit is used to start another traversal down the tree. To see how this procedure works, let us decode the binary string generated in the previous example.

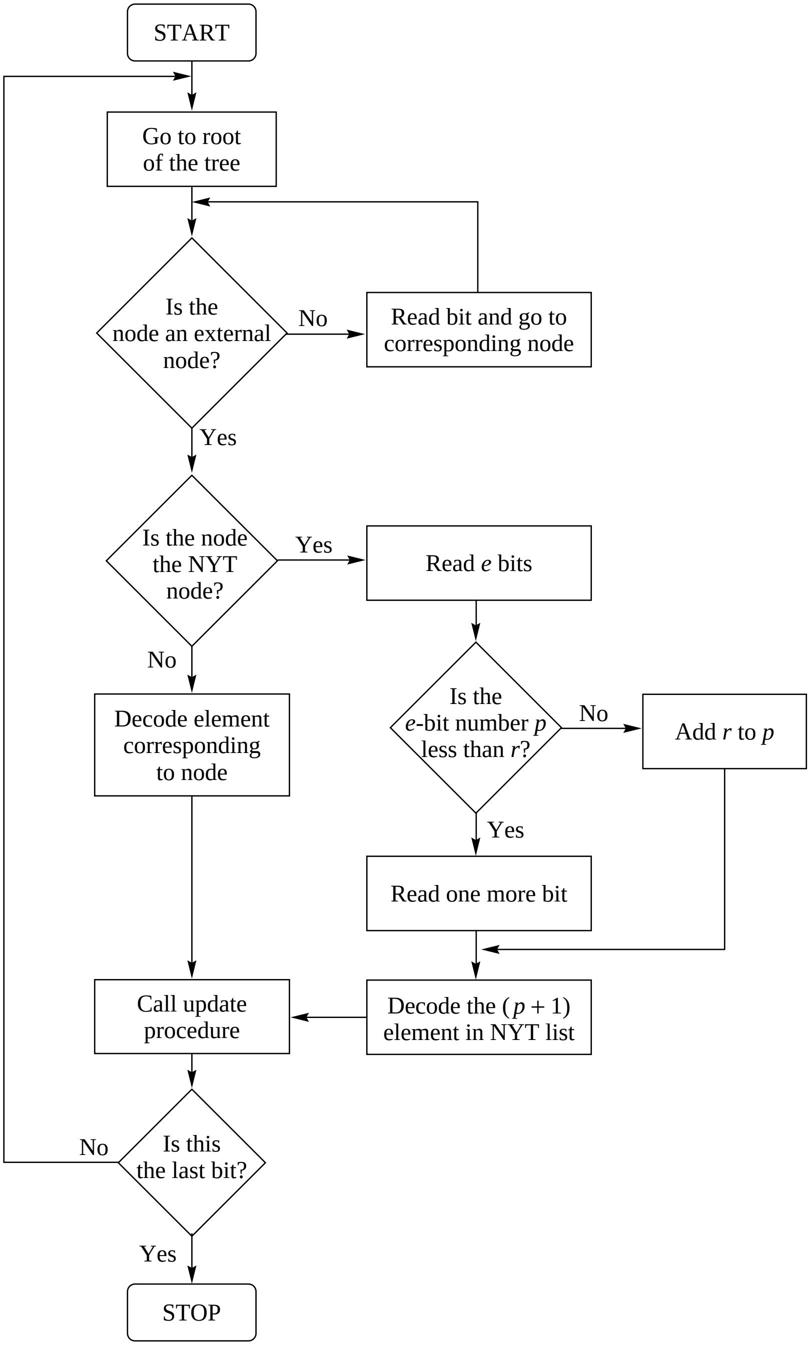

START
Go to root of the tree
Is the node an external node?
No
Read bit and go to corresponding node
Yes
Is the node the NYT node?
Yes
Read *e* bits
No
Decode element corresponding to node
Is the *e*-bit number *p* less than *r*?
No
Add *r* to *p*
Yes
Read one more bit
Call update procedure
Decode the (*p* + 1) element in NYT list
No
Is this the last bit?
Yes
STOP

FIGURE 3.9 **Flowchart of the decoding procedure.**

Example 3.4.3: Decoding procedure

The binary string generated by the encoding procedure is

$$000001010001000001100010110$$

Initially, the decoder tree consists only of the NYT node. Therefore, the first symbol to be decoded must be obtained from the NYT list. We read in the first 4 bits, 0000, as the value of e is four. The 4 bits 0000 correspond to the decimal value of 0. As this is less than the value of r, which is 10, we read in one more bit for the entire code of 00000. Adding one to the decimal value corresponding to this binary string, we get the index of the received symbol as 1. This is the index for a; therefore, the first letter is decoded as a. The tree is now updated as shown in Figure 3.7. The next bit in the string is 1. This traces a path from the root node to the external node corresponding to a. We decode the symbol a and update the tree. In this case, the update consists only of incrementing the weight of the external node corresponding to a. The next bit is a 0, which traces a path from the root to the NYT node. The next 4 bits, 1000, correspond to the decimal number 8, which is less than 10, so we read in one more bit to get the 5-bit word 10001. The decimal equivalent of this 5-bit word plus one is 18, which is the index for r. We decode the symbol r and then update the tree. The next 2 bits, 00, again trace a path to the NYT node. We read the next 4 bits, 0001. Since this corresponds to the decimal number 1, which is less than 10, we read another bit to get the 5-bit word 00011. To get the index of the received symbol in the NYT list, we add one to the decimal value of this 5-bit word. The value of the index is 4, which corresponds to the symbol d. Continuing in this fashion, we decode the sequence *aardva*. ◆

Although the Huffman coding algorithm is one of the best-known variable-length coding algorithms, there are some other lesser-known algorithms that can be very useful in certain situations. In particular, the Golomb-Rice codes and the Tunstall codes are becoming increasingly popular. We describe these codes in the following sections.

3.5 Golomb Codes

The Golomb-Rice codes belong to a family of codes designed to encode integers with the assumption that the larger an integer, the lower its probability of occurrence. The simplest code for this situation is the *unary* code. The unary code for a positive integer n is simply n 1s followed by a 0. Thus, the code for 4 is 11110, and the code for 7 is 11111110. The unary code is the same as the Huffman code for the semi-infinite alphabet $\{1, 2, 3, \ldots\}$ with probability model

$$P[k] = \frac{1}{2^k}.$$

Because the Huffman code is optimal, the unary code is also optimal for this probability model.

Although the unary code is optimal in very restricted conditions, we can see that it is certainly very simple to implement. One step higher in complexity are a number of coding schemes that split the integer into two parts, representing one part with a unary code and the other part with a different code. An example of such a code is the Golomb code. Other examples can be found in [27].

The Golomb code is described in a succinct paper [28] by Solomon Golomb, which begins “Secret Agent 00111 is back at the Casino again, playing a game of chance, while the fate of mankind hangs in the balance.” Agent 00111 requires a code to represent runs of success in a roulette game, and Golomb provides it! The Golomb code is actually a family of codes parameterized by an integer $m > 0$. In the Golomb code with parameter m, we represent an integer $n > 0$ using two numbers q and r, where

$$q = \left\lfloor \frac{n}{m} \right\rfloor$$

and

$$r = n - qm.$$

$\lfloor x \rfloor$ is the integer part of x. In other words, q is the quotient and r is the remainder when n is divided by m. The quotient q can take on values $0, 1, 2, \ldots$ and is represented by the unary code of q. The remainder r can take on the values $0, 1, 2, \ldots, m-1$. If m is a power of two, we use the $\log_2 m$-bit binary representation of r. If m is not a power of two, we could still use $\lceil \log_2 m \rceil$ bits, where $\lceil x \rceil$ is the smallest integer greater than or equal to x. We can reduce the number of bits required if we use the $\lfloor \log_2 m \rfloor$-bit binary representation of r for the first $2^{\lceil \log_2 m \rceil} - m$ values, and the $\lceil \log_2 m \rceil$-bit binary representation of $r + 2^{\lceil \log_2 m \rceil} - m$ for the rest of the values.

Example 3.5.1: Golomb code

Let’s design a Golomb code for $m = 5$. As

$$\lceil \log_2 5 \rceil = 3, \quad \text{and} \quad \lfloor \log_2 5 \rfloor = 2$$

the first $8 - 5 = 3$ values of r (that is, $r = 0$, 1, 2) will be represented by the 2-bit binary representation of r, and the next two values (that is, $r = 3$, 4) will be represented by the 3-bit representation of $r + 3$. The quotient q is always represented by the unary code for q. Thus, the codeword for 3 is 0110, and the codeword for 21 is 1111001. The codewords for $n = 0, \ldots, 15$ are shown in Table 3.16.

TABLE 3.16 Golomb code for $m = 5$.

n	q	r	Codeword	n	q	r	Codeword
0	0	0	000	8	1	3	10110
1	0	1	001	9	1	4	10111
2	0	2	010	10	2	0	11000
3	0	3	0110	11	2	1	11001
4	0	4	0111	12	2	2	11010
5	1	0	1000	13	2	3	110110
6	1	1	1001	14	2	4	110111
7	1	2	1010	15	3	0	111000

◆

It can be shown that the Golomb code is optimal for the probability model

$$P(n) = p^{n-1}q, \qquad q = 1 - p$$

when

$$m = \left\lceil -\frac{1}{\log_2 p} \right\rceil.$$

3.6 Rice Codes

The Rice code was originally developed by Robert F. Rice (he called it the Rice machine) [29, 30] and later extended by Pen-Shu Yeh and Warner Miller [31]. The Rice code can be viewed as an adaptive Golomb code. In the Rice code, a sequence of nonnegative integers (which might have been obtained from the preprocessing of other data) is divided into blocks of J integers apiece. Each block is then coded using one of several options, most of which are a form of Golomb codes. Each block is encoded with each of these options, and the option resulting in the least number of coded bits is selected. The particular option used is indicated by an identifier attached to the code for each block.

The easiest way to understand the Rice code is to examine one of its implementations. We will study the implementation of the Rice code in the recommendation for lossless compression from the Consultative Committee on Space Data Standards (CCSDS).

3.6.1 CCSDS Recommendation for Lossless Compression

As an application of the Rice algorithm, let's briefly look at the algorithm for lossless data compression recommended by CCSDS. The algorithm consists of a preprocessor (the modeling step) and a binary coder (coding step). The preprocessor removes correlation from the input and generates a sequence of nonnegative integers. This sequence has the property that smaller values are more probable than larger values. The binary coder generates a bitstream to represent the integer sequence. The binary coder is our main focus at this point.

The preprocessor functions as follows: Given a sequence $\{y_i\}$, for each y_i we generate a prediction $\hat{y}_i$. A simple way to generate a prediction would be to take the previous value of the sequence to be a prediction of the current value of the sequence:

$$\hat{y}_i = y_{i-1}.$$

We will look at more sophisticated ways of generating a prediction in Chapter 7. We then generate a sequence whose elements are the difference between y_i and its predicted value $\hat{y}_i$:

$$d_i = y_i - \hat{y}_i.$$

The d_i value will have a small magnitude when our prediction is good and a large value when it is not. Assuming an accurate modeling of the data, the former situation is more likely than the latter. Let $y_{\max}$ and $y_{\min}$ be the largest and smallest values that the sequence

$\{y_i\}$ takes on. It is reasonable to assume that the value of $\hat{y}$ will be confined to the range $[y_{\min}, y_{\max}]$. Define

$$T_i = \min\{y_{\max} - \hat{y}, \hat{y} - y_{\min}\}. \tag{3.8}$$

The sequence $\{d_i\}$ can be converted into a sequence of nonnegative integers $\{x_i\}$ using the following mapping:

$$x_i = \begin{cases} 2d_i & 0 \leq d_i \leq T_i \\ 2\,|d_i| - 1 & -T_i \leq d_i < 0 \\ T_i + |d_i| & \text{otherwise.} \end{cases} \tag{3.9}$$

The value of x_i will be small whenever the magnitude of d_i is small. Therefore, the value of x_i will be small with higher probability. The sequence $\{x_i\}$ is divided into segments with each segment being further divided into blocks of size J. It is recommended by CCSDS that J have a value of 16. Each block is then coded using one of the following options. The coded block is transmitted along with an identifier that indicates which particular option was used.

- **Fundamental sequence:** This is a unary code. A number n is represented by a sequence of n 0s followed by a 1 (or a sequence of n 1s followed by a 0).
- **Split sample options:** These options consist of a set of codes indexed by a parameter m. The code for a k-bit number n using the mth split sample option consists of the m least significant bits of k followed by a unary code representing the $k - m$ most significant bits. For example, suppose we wanted to encode the 8-bit number 23 using the third split sample option. The 8-bit representation of 23 is 00010111. The three least significant bits are 111. The remaining bits (00010) correspond to the number 2, which has a unary code 001. Therefore, the code for 23 using the third split sample option is 111011. Notice that different values of m will be preferable for different values of x_i, with higher values of m used for higher-entropy sequences.
- **Second extension option:** The second extension option is useful for sequences with low entropy—when, in general, many of the values of x_i will be zero. In the second extension option the sequence is divided into consecutive pairs of samples. Each pair is used to obtain an index γ using the following transformation:

  $$\gamma = \frac{1}{2}(x_i + x_{i+1})(x_i + x_{i+1} + 1) + x_{i+1} \tag{3.10}$$

  and the value of γ is encoded using a unary code. The value of γ is an index to a lookup table with each value of γ corresponding to a pair of values x_i, x_{i+1}.
- **Zero block option:** The zero block option is used when one or more of the blocks of x_i are zero—generally when we have long sequences of y_i that have the same value. In this case the number of zero blocks are transmitted using the code shown in Table 3.17. The ROS code is used when the last five or more blocks in a segment are all zero.

The Rice code has been used in several space applications, and variations of the Rice code have been proposed for a number of different applications.

TABLE 3.17 Code used for zero block option.

Number of All-Zero Blocks	Codeword
1	1
2	01
3	001
4	0001
5	000001
6	0000001
⋮	⋮
63	$\overbrace{000\cdots0}^{63\ 0\text{s}}1$
ROS	00001

3.7 Tunstall Codes

Most of the variable-length codes that we look at in this book encode letters from the source alphabet using codewords with varying numbers of bits: codewords with fewer bits for letters that occur more frequently and codewords with more bits for letters that occur less frequently. The Tunstall code is an important exception. In the Tunstall code, all codewords are of equal length. However, each codeword represents a different number of letters. An example of a 2-bit Tunstall code for an alphabet $\mathcal{A} = \{A, B\}$ is shown in Table 3.18. The main advantage of a Tunstall code is that errors in codewords do not propagate, unlike other variable-length codes, such as Huffman codes, in which an error in one codeword will cause a series of errors to occur.

Example 3.7.1:

Let's encode the sequence $AAABAABAABAABAAA$ using the code in Table 3.18. Starting at the left, we can see that the string AAA occurs in our codebook and has a code of 00. We then code B as 11, AAB as 01, and so on. We finally end up with coded string 001101010100.

◆

TABLE 3.18 A 2-bit Tunstall code.

Sequence	Codeword
AAA	00
AAB	01
AB	10
B	11

TABLE 3.19 **A 2-bit (non-Tunstall) code.**

Sequence	Codeword
AAA	00
ABA	01
AB	10
B	11

The design of a code that has a fixed codeword length but a variable number of symbols per codeword should satisfy the following conditions:

1. We should be able to parse a source output sequence into sequences of symbols that appear in the codebook.
2. We should maximize the average number of source symbols represented by each codeword.

In order to understand what we mean by the first condition, consider the code shown in Table 3.19. Let's encode the same sequence $AAABAABAABAABAAA$ as in the previous example using the code in Table 3.19. We first encode AAA with the code 00. We then encode B with 11. The next three symbols are AAB. However, there are no codewords corresponding to this sequence of symbols. Thus, this sequence is unencodable using this particular code—not a desirable situation.

Tunstall [32] gives a simple algorithm that fulfills these conditions. The algorithm is as follows:

Suppose we want an n-bit Tunstall code for a source that generates *iid* letters from an alphabet of size N. The number of codewords is 2^n. We start with the N letters of the source alphabet in our codebook. Remove the entry in the codebook that has the highest probability and add the N strings obtained by concatenating this letter with every letter in the alphabet (including itself). This will increase the size of the codebook from N to $N+(N-1)$. The probabilities of the new entries will be the product of the probabilities of the letters concatenated to form the new entry. Now look through the $N+(N-1)$ entries in the codebook and find the entry that has the highest probability, keeping in mind that the entry with the highest probability may be a concatenation of symbols. Each time we perform this operation we increase the size of the codebook by $N-1$. Therefore, this operation can be performed K times, where

$$N+K(N-1) \leq 2^n.$$

Example 3.7.2: Tunstall codes

Let us design a 3-bit Tunstall code for a memoryless source with the following alphabet:

$$\mathcal{A} = \{A, B, C\}$$

$$P(A) = 0.6, \quad P(B) = 0.3, \quad P(C) = 0.1$$

TABLE 3.20 **Source alphabet and associated probabilities.**

Letter	Probability
A	0.60
B	0.30
C	0.10

TABLE 3.21 **The codebook after one iteration.**

Sequence	Probability
B	0.30
C	0.10
AA	0.36
AB	0.18
AC	0.06

TABLE 3.22 **A 3-bit Tunstall code.**

Sequence	Probability
B	000
C	001
AB	010
AC	011
AAA	100
AAB	101
AAC	110

We start out with the codebook and associated probabilities shown in Table 3.20. Since the letter *A* has the highest probability, we remove it from the list and add all two-letter strings beginning with *A* as shown in Table 3.21. After one iteration we have 5 entries in our codebook. Going through one more iteration will increase the size of the codebook by 2, and we will have 7 entries, which is still less than the final codebook size. Going through another iteration after that would bring the codebook size to 10, which is greater than the maximum size of 8. Therefore, we will go through just one more iteration. Looking through the entries in Table 3.22, the entry with the highest probability is *AA*. Therefore, at the next step we remove *AA* and add all extensions of *AA* as shown in Table 3.22. The final 3-bit Tunstall code is shown in Table 3.22. ◆

3.8 Applications of Huffman Coding

In this section we describe some applications of Huffman coding. As we progress through the book, we will describe more applications, since Huffman coding is often used in conjunction with other coding techniques.

3.8.1 Lossless Image Compression

A simple application of Huffman coding to image compression would be to generate a Huffman code for the set of values that any pixel may take. For monochrome images, this set usually consists of integers from 0 to 255. Examples of such images are contained in the accompanying data sets. The four that we will use in the examples in this book are shown in Figure 3.10.

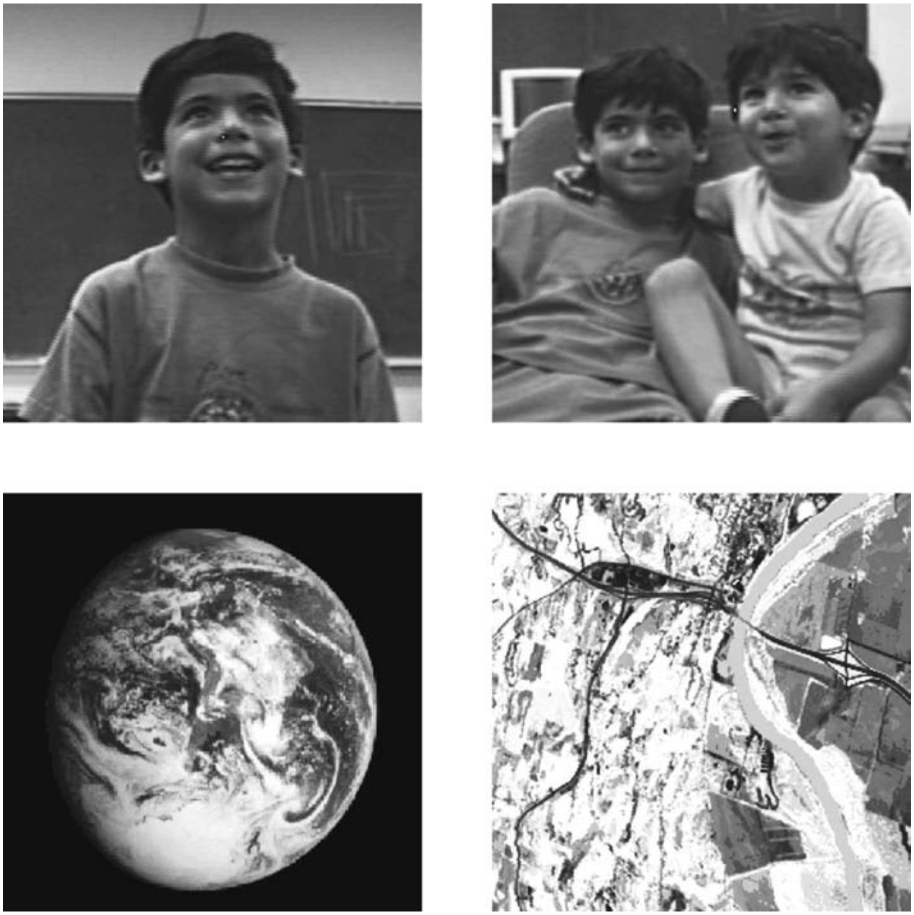


FIGURE 3. 10 **Test images.**

TABLE 3.23 Compression using Huffman codes on pixel values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	7.01	57,504	1.14
Sensin	7.49	61,430	1.07
Earth	4.94	40,534	1.62
Omaha	7.12	58,374	1.12

We will make use of one of the programs from the accompanying software (see Preface) to generate a Huffman code for each image, and then encode the image using the Huffman code. The results for the four images in Figure 3.10 are shown in Table 3.23. The Huffman code is stored along with the compressed image as the code will be required by the decoder to reconstruct the image.

The original (uncompressed) image representation uses 8 bits/pixel. The image consists of 256 rows of 256 pixels, so the uncompressed representation uses 65,536 bytes. The compression ratio is simply the ratio of the number of bytes in the uncompressed representation to the number of bytes in the compressed representation. The number of bytes in the compressed representation includes the number of bytes needed to store the Huffman code. Notice that the compression ratio is different for different images. This can cause some problems in certain applications where it is necessary to know in advance how many bytes will be needed to represent a particular data set.

The results in Table 3.23 are somewhat disappointing because we get a reduction of only about $\frac{1}{2}$ to 1 bit/pixel after compression. For some applications this reduction is acceptable. For example, if we were storing thousands of images in an archive, a reduction of 1 bit/pixel saves many megabytes in disk space. However, we can do better. Recall that when we first talked about compression, we said that the first step for any compression algorithm was to model the data so as to make use of the structure in the data. In this case, we have made absolutely no use of the structure in the data.

From a visual inspection of the test images, we can clearly see that the pixels in an image are heavily correlated with their neighbors. We could represent this structure with the crude model $\hat{x}_n = x_{n-1}$. The residual would be the difference between neighboring pixels. If we carry out this differencing operation and use the Huffman coder on the residuals, the results are as shown in Table 3.24. As we can see, using the structure in the data resulted in substantial improvement.

TABLE 3.24 Compression using Huffman codes on pixel difference values.

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	4.02	32,968	1.99
Sensin	4.70	38,541	1.70
Earth	4.13	33,880	1.93
Omaha	6.42	52,643	1.24

TABLE 3.25 **Compression using adaptive Huffman codes on pixel difference values.**

Image Name	Bits/Pixel	Total Size (bytes)	Compression Ratio
Sena	3.93	32,261	2.03
Sensin	4.63	37,896	1.73
Earth	4.82	39,504	1.66
Omaha	6.39	52,321	1.25

The results in Tables 3.23 and 3.24 were obtained using a two-pass system, in which the statistics were collected in the first pass and a Huffman table was generated. Instead of using a two-pass system, we could have used a one-pass adaptive Huffman coder. The results for this are given in Table 3.25.

Notice that there is little difference between the performance of the adaptive Huffman code and the two-pass Huffman coder. In addition, the fact that the adaptive Huffman coder can be used as an on-line or real-time coder makes the adaptive Huffman coder a more attractive option in many applications. However, the adaptive Huffman coder is more vulnerable to errors and may also be more difficult to implement. In the end, the particular application will determine which approach is more suitable.

3.8.2 Text Compression

Text compression seems natural for Huffman coding. In text, we have a discrete alphabet that, in a given class, has relatively stationary probabilities. For example, the probability model for a particular novel will not differ significantly from the probability model for another novel. Similarly, the probability model for a set of FORTRAN programs is not going to be much different than the probability model for a different set of FORTRAN programs. The probabilities in Table 3.26 are the probabilities of the 26 letters (upper- and lowercase) obtained for the U.S. Constitution and are representative of English text. The probabilities in Table 3.27 were obtained by counting the frequency of occurrences of letters in an earlier version of this chapter. While the two documents are substantially different, the two sets of probabilities are very much alike.

We encoded the earlier version of this chapter using Huffman codes that were created using the probabilities of occurrence obtained from the chapter. The file size dropped from about 70,000 bytes to about 43,000 bytes with Huffman coding.

While this reduction in file size is useful, we could have obtained better compression if we first removed the structure existing in the form of correlation between the symbols in the file. Obviously, there is a substantial amount of correlation in this text. For example, *Huf* is always followed by *fman*! Unfortunately, this correlation is not amenable to simple numerical models, as was the case for the image files. However, there are other somewhat more complex techniques that can be used to remove the correlation in text files. We will look more closely at these in Chapters 5 and 6.

TABLE 3.26 **Probabilities of occurrence of the letters in the English alphabet in the U.S. Constitution.**

Letter	Probability	Letter	Probability
A	0.057305	N	0.056035
B	0.014876	O	0.058215
C	0.025775	P	0.021034
D	0.026811	Q	0.000973
E	0.112578	R	0.048819
F	0.022875	S	0.060289
G	0.009523	T	0.078085
H	0.042915	U	0.018474
I	0.053475	V	0.009882
J	0.002031	W	0.007576
K	0.001016	X	0.002264
L	0.031403	Y	0.011702
M	0.015892	Z	0.001502

TABLE 3.27 **Probabilities of occurrence of the letters in the English alphabet in this chapter.**

Letter	Probability	Letter	Probability
A	0.049855	N	0.048039
B	0.016100	O	0.050642
C	0.025835	P	0.015007
D	0.030232	Q	0.001509
E	0.097434	R	0.040492
F	0.019754	S	0.042657
G	0.012053	T	0.061142
H	0.035723	U	0.015794
I	0.048783	V	0.004988
J	0.000394	W	0.012207
K	0.002450	X	0.003413
L	0.025835	Y	0.008466
M	0.016494	Z	0.001050

3.8.3 Audio Compression

Another class of data that is very suitable for compression is CD-quality audio data. The audio signal for each stereo channel is sampled at 44.1 kHz, and each sample is represented by 16 bits. This means that the amount of data stored on one CD is enormous. If we want to transmit this data, the amount of channel capacity required would be significant. Compression is definitely useful in this case. In Table 3.28 we show for a variety of audio material the file size, the entropy, the estimated compressed file size if a Huffman coder is used, and the resulting compression ratio.

TABLE 3.28 Huffman coding of 16-bit CD-quality audio.

File Name	Original File Size (bytes)	Entropy (bits)	Estimated Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	12.8	725,420	1.30
Cohn	402,442	13.8	349,300	1.15
Mir	884,020	13.7	759,540	1.16

The three segments used in this example represent a wide variety of audio material, from a symphonic piece by Mozart to a folk rock piece by Cohn. Even though the material is varied, Huffman coding can lead to some reduction in the capacity required to transmit this material.

Note that we have only provided the *estimated* compressed file sizes. The estimated file size in bits was obtained by multiplying the entropy by the number of samples in the file. We used this approach because the samples of 16-bit audio can take on 65,536 distinct values, and therefore the Huffman coder would require 65,536 distinct (variable-length) codewords. In most applications, a codebook of this size would not be practical. There is a way of handling large alphabets, called recursive indexing, that we will describe in Chapter 9. There is also some recent work [14] on using a Huffman tree in which leaves represent sets of symbols with the same probability. The codeword consists of a prefix that specifies the set followed by a suffix that specifies the symbol within the set. This approach can accommodate relatively large alphabets.

As with the other applications, we can obtain an increase in compression if we first remove the structure from the data. Audio data can be modeled numerically. In later chapters we will examine more sophisticated modeling approaches. For now, let us use the very simple model that was used in the image-coding example; that is, each sample has the same value as the previous sample. Using this model we obtain the difference sequence. The entropy of the difference sequence is shown in Table 3.29.

Note that there is a further reduction in the file size: the compressed file sizes are about 60% of the original files. Further reductions can be obtained by using more sophisticated models.

Many of the lossless audio compression schemes, including FLAC (Free Lossless Audio Codec), Apple's ALAC or ALE, *Shorten* [33], *Monkey's Audio*, and the proposed (as of now) MPEG-4 ALS [34] algorithms, use a linear predictive model to remove some of

TABLE 3.29 Huffman coding of differences of 16-bit CD-quality audio.

File Name	Original File Size (bytes)	Entropy of Differences (bits)	Estimated Compressed File Size (bytes)	Compression Ratio
Mozart	939,862	9.7	569,792	1.65
Cohn	402,442	10.4	261,590	1.54
Mir	884,020	10.9	602,240	1.47