Appendix A
Information Theory

This appendix serves as a brief introduction to information theory, the foundation of many techniques used in data compression. The two most important terms covered here are entropy and redundancy (see also Section 2.3 for an alternative discussion of these concepts).

A.1 Information Theory Concepts

We intuitively know what information is. We constantly receive and send information in the form of text, sound, and images. We also feel that information is an elusive nonmathematical quantity that cannot be precisely defined, captured, or measured. The standard dictionary definitions of information are (1) knowledge derived from study, experience, or instruction; (2) knowledge of a specific event or situation; intelligence; (3) a collection of facts or data; (4) the act of informing or the condition of being informed; communication of knowledge.

Imagine a person who does not know what information is. Would those definitions make it clear to them? Unlikely.

The importance of information theory is that it quantifies information. It shows how to measure information, so that we can answer the question “how much information is included in this piece of data?” with a precise number! Quantifying information is based on the observation that the information content of a message is equivalent to the amount of surprise in the message. If I tell you something that you already know (for example, “you and I work here”), I haven’t given you any information. If I tell you something new (for example, “we both received a raise”), I have given you some information. If I tell you something that really surprises you (for example, “only I received a raise”), I have given you more information, regardless of the number of words I have used, and of how you feel about my information.

D. Salomon, G. Motta, Handbook of Data Compression, 5th ed.
DOI 10.1007/978-1-84882-903-9_BM2, © Springer-Verlag London Limited 2010
We start with a simple, familiar event that’s easy to analyze, namely the toss of a coin. There are two results, so the result of any toss is initially uncertain. We have to actually throw the coin in order to resolve the uncertainty. The result is heads or tails, which can also be expressed as a yes or no, or as a 0 or 1; a bit.

A single bit resolves the uncertainty in the toss of a coin. What makes this example important is the fact that it can easily be generalized. Many real-life problems can be resolved, and their solutions expressed, by means of several bits. The principle of doing so is to find the minimum number of yes/no questions that must be answered in order to arrive at the result. Since the answer to a yes/no question can be expressed with one bit, the number of questions will equal the number of bits it takes to express the information contained in the result.

A slightly more complex example is a deck of 64 playing cards. For simplicity let’s ignore their traditional names and numbers and simply number them 1 to 64. Consider the event of person A drawing one card and person B having to guess what it was. The guess is a number between 1 and 64. What is the minimum number of yes/no questions that are needed to guess the card? Those who are familiar with the technique of binary search know the answer. Using this technique, B should divide the interval 1–64 in two, and should start by asking “is the result between 1 and 32?” If the answer is no, then the result is in the interval 33 to 64. This interval is then divided by two and B’s next question should be “is the result between 33 and 48?” This process continues until the interval selected by B reduces to a single number.

It does not take much to see that exactly six questions are necessary to get at the result. This is because 6 is the number of times 64 can be divided in half. Mathematically, this is equivalent to writing $6 = \log_2 64$. This is why the logarithm is the mathematical function that quantifies information.

Another approach to the same problem is to ask the question; Given a nonnegative integer $N$, how many digits does it take to express it? The answer, of course, depends on $N$. The greater $N$, the more digits are needed. The first 100 nonnegative integers (0 to 99) can be expressed by two decimal digits. The first 1000 such integers can be expressed by three digits. Again it does not take long to see the connection. The number of digits required to represent $N$ equals approximately $\log N$. The base of the logarithm is the same as the base of the digits. For decimal digits, use base 10; for binary digits (bits), use base 2. If we agree that the number of digits it takes to express $N$ is proportional to the information content of $N$, then again the logarithm is the function that gives us a measure of the information.

⋄ Exercise A.1: What is the precise size, in bits, of the binary integer $i$?

Here is another approach to quantifying information. We are familiar with the ten decimal digits. We know that the value of a digit in a number depends on its position. Thus, the value of the digit 4 in the number 14708 is $4 \times 10^3$, or 4000, since it is in position 3 (positions are numbered from right to left, starting from 0). We are also familiar with the two binary digits (bits) 0 and 1. The value of a bit in a binary number similarly depends on its position, except that powers of 2 are used. Mathematically, there is nothing special about 2 or 10. We can use the number 3 as the basis of our arithmetic. This would require the three digits, 0, 1, and 2 (we might call them trits). A trit $t$ at position $i$ would have a value of $t \times 3^i$. 
A.1 Information Theory Concepts

⋄ Exercise A.2: Actually, there is something special about 10. We use base-10 numbers because we have ten fingers. There is also something special about the use of 2 as the basis for a number system. What is it?

Given a decimal (base 10) or a ternary (base 3) number with \( k \) digits, a natural question is; how much information is included in this \( k \)-digit number? We answer this by determining the number of bits it takes to express the given number. Assuming that the answer is \( x \), then \( 10^k - 1 = 2^x - 1 \). This is because \( 10^k - 1 \) is the largest \( k \)-digit decimal number and \( 2^x - 1 \) is the largest \( x \)-bit binary number. Solving the equation above for \( x \) as the unknown is easily done using logarithms and yields

\[
x = k \log_{10} \frac{10}{2}.
\]

We can use any base for the logarithm, as long as we use the same base for \( \log_{10} \) and \( \log_{2} \). Selecting base 2 simplifies the result, which becomes \( x = k \log_{2} 10 \approx 3.32k \). This shows that the information included in one decimal digit equals that contained in about 3.32 bits. In general, given numbers in base \( n \), we can write \( x = k \log_{2} n \), which expresses the fact that the information included in one base-\( n \) digit equals that included in \( \log_{2} n \) bits.

⋄ Exercise A.3: How many bits does it take to express the information included in one trit?

We now turn to a transmitter, a piece of hardware that can transmit data over a communications line (a channel). In practice, such a transmitter sends binary data (a modem is a good example). However, in order to obtain general results, we assume that the data is a string made up of occurrences of the \( n \) symbols \( a_1 \) through \( a_n \). Such a set is an \( n \)-symbol alphabet. Since there are \( n \) symbols, we can think of each as a base-\( n \) digit, which means that it is equivalent to \( \log_{2} n \) bits. As far as the hardware is concerned, this means that it must be able to transmit at \( n \) discrete levels.

If the transmitter takes \( 1/s \) time units to transmit a single symbol, then the speed of the transmission is \( s \) symbols per time unit. A common example is \( s = 28800 \) baud (baud is the term for “bits per second”), which translates to \( 1/s \approx 34.7 \) \( \mu \)sec (where the Greek letter \( \mu \) stands for “micro” and \( 1 \mu \)sec = \( 10^{-6} \) sec). In one time unit, the transmitter can send \( s \) symbols, which as far as information content is concerned, is equivalent to \( s \log_{2} n \) bits. We denote by \( H = s \log_{2} n \) the amount of information, measured in bits, transmitted in each time unit.

The next step is to express \( H \) in terms of the probabilities of occurrence of the \( n \) symbols. We assume that symbol \( a_i \) occurs in the data with probability \( P_i \). The sum of the probabilities equals, of course, unity: \( P_1 + P_2 + \cdots + P_n = 1 \). In the special case where all \( n \) probabilities are equal, \( P_1 = P_2 = \cdots = P_n = P \), we get \( 1 = \sum P_i = nP \), implying that \( P = 1/n \), and resulting in \( H = s \log_{2} n = s \log_{2}(1/P) = -s \log_{2} P \). In general, the probabilities are different, and we want to express \( H \) in terms of all of them. Since symbol \( a_i \) occurs a fraction \( P_i \) of the time in the data, it occurs on the average \( sP_i \) times each time unit, so its contribution to \( H \) is \( -sP_i \log_{2} P_i \). The sum of the contributions of all \( n \) symbols to \( H \) is therefore \( H = -s \sum_{i=1}^{n} P_i \log_{2} P_i \).
As a reminder, $H$ is the amount of information, in bits, sent by the transmitter in one time unit. The amount of information contained in one base-$n$ symbol is therefore $H/n$ (because it takes time $1/s$ to transmit one symbol), or $-\sum_i^n P_i \log_2 P_i$. This quantity is called the entropy of the data being transmitted. In analogy we can define the entropy of a single symbol $a_i$ as $-P_i \log_2 P_i$. This is the smallest number of bits needed, on average, to represent the symbol.

(Information theory was developed, in the late 1940s, by Claude Shannon, of Bell Labs, and he chose the term entropy because this term is used in thermodynamics to indicate the amount of disorder in a physical system.)

Since I think it is better to take the names of such quantities as these, which are important for science, from the ancient languages, so that they can be introduced without change into all the modern languages, I propose to name the magnitude $S$ the entropy of the body, from the Greek word "trope" for "transformation." I have intentionally formed the word "entropy" so as to be as similar as possible to the word "energy" since both these quantities which are to be known by these names are so nearly related to each other in their physical significance that a certain similarity in their names seemed to me advantageous.

—Rudolph Clausius, 1865 (translated by Hans C. von Baeyer)

The entropy of data depends on the individual probabilities $P_i$ and it is at its maximum (see Exercise A.4) when all $n$ probabilities are equal. This fact is used to define the redundancy $R$ in the data. It is defined as the difference between a symbol set’s largest possible entropy and its actual entropy. Thus

$$R = \left[ -\sum_1^n P \log_2 P \right] - \left[ -\sum_1^n P_i \log_2 P_i \right] = \log_2 n + \sum_1^n P_i \log_2 P_i.$$ 

Thus, the test for fully compressed data (no redundancy) is $\log_2 n + \sum_1^n P_i \log_2 P_i = 0$.

◊ Exercise A.4: Analyze the entropy of a two-symbol set.

Given a string of characters, the probability of a character can be determined by counting the frequency of the character and dividing by the length of the string. Once the probabilities of all the characters are known, the entropy of the entire string can be calculated. With current availability of powerful mathematical software, it is easy to calculate the entropy of a given string. The Mathematica code

```mathematica
Frequencies[list_] := Map[{Count[list, #], #} &, Union[list]];
Entropy[list_] := -Plus @@ N[# Log[2, #]] & @
  (First[Transpose[Frequencies[list]]]/Length[list]);
```
A.1 Information Theory Concepts

Characters["swiss miss"]

Entropy[%]

does that and shows that, for example, the entropy of the string \textit{swiss miss} is 1.96096.

The main theorem proved by Shannon says essentially that a message of \( n \) symbols can, on average, be compressed down to \( nH \) bits, but not further. It also says that almost optimal compressors (called \textit{entropy encoders}) exist, but does not show how to construct them. Arithmetic coding (Section 5.9) is an example of an entropy encoder, as are also the dictionary-based algorithms of Chapter 6 (but the latter require huge quantities of data to perform at the entropy level).

---

You have two chances—
One of getting the germ
And one of not.
And if you get the germ
You have two chances—
One of getting the disease
And one of not.
And if you get the disease
You have two chances—
One of dying
And one of not.
And if you die—
Well, you still have two chances.

—Unknown

---

Appendix A.1.1 Algorithmic Information Content

Consider the following three sequences:

\[ S_1 = 100100100100100100100100100100100100\ldots, \]
\[ S_2 = 01011011011010110110101101101101101\ldots, \]
\[ S_3 = 01110010011110010000101100000011101111\ldots. \]

The first sequence, \( S_1 \), is just a repetition of the simple pattern 100. \( S_2 \) is less regular. It can be described as a 01, followed by \( r_1 \) repetitions of 011, followed by another 01, followed by \( r_2 \) repetitions of 011, etc., where \( r_1 = 3 \), \( r_2 = 2 \), \( r_3 = 4 \), and the other \( r_i \)'s are not shown. \( S_3 \) is more difficult to describe, since it does not seem to have any apparent regularity; it looks random. Notice that the meaning of the ellipsis is clear in the case of \( S_1 \) (just repeat the pattern 100), less clear in \( S_2 \) (what are the other \( r_i \)'s?), and completely unknown in \( S_3 \) (is it random?).

We now assume that these sequences are very long (say, 999,999 bits each), and each continues “in the same way.” How can we define the complexity of such a binary sequence, at least qualitatively? One way to do so, called the Kolmogorov-Chaitin
complexity (KCC), is to define the complexity of a binary string $S$ as the length, in bits, of the shortest computer program that, when executed, generates $S$ (display it, print it, or write it on file). This definition is also called the \textit{algorithmic information content} of string $S$.

A computer program $P_1$ to generate string $S_1$ could just loop 333,333 times and print 100 in each iteration. Alternatively, the program could loop 111,111 times and print 100100100 in each iteration. Such a program is very short (especially when compared with the length of the sequence it generates), concurring with our intuitive feeling that $S_1$ has low complexity.

A program $P_2$ to generate string $S_2$ should know the values of all the $r_i$’s. They could either be built in or input by the user at run time. The program initializes a variable $i$ to 1. It then prints “01”, loops $r_i$ times printing “011” in each iteration, increments $i$ by 1, and repeats this behavior until 999,999 bits have been printed. Such a program is longer than $P_1$, thereby reflecting our intuitive feel that $S_2$ is more complex than $S_1$.

A program $P_3$ to generate $S_3$ should (assuming that we cannot express this string in any regular way) simply print all 999,999 bits of the sequence. Such a program is as long as the sequence itself, implying that the KCC of $S_3$ is as large as $S_3$.

Using this definition of complexity, Gregory Chaitin showed (see [Chaitin 77] or [Chaitin 97]) that most binary strings of length $n$ are random; their complexities are close to $n$. However, the “interesting” (or “practical”) binary strings, those that are used in practice to represent text, images, and sound, and are compressed all the time, are similar to $S_2$. They are not random. They exhibit some regularity, which makes it possible to compress them. Very regular strings, such as $S_1$, are rare in practice.

Algorithmic information content is a measure of the amount of information included in a message. It is related to the KCC and is different from the way information is measured in information theory. Shannon’s information theory defines the amount of information in a string by considering the amount of \textit{surprise} this information contains when revealed. Algorithmic information content, on the other hand, measures information that has already been revealed. An example may serve to illustrate this difference. Imagine two persons $A$ (well-read, sophisticated, and knowledgeable) and $B$ (inexperienced and naive), reading the same story. There are few surprises in the story for $A$. He has already read many similar stories and can predict the development of the plot, the behavior of the characters, and even the end. The opposite is true for $B$. As he reads, he is surprised by the (to him) unexpected twists and turns that the story takes and by the (to him) unpredictable behavior of the characters. The question is; How much information does the story really contain?

Shannon’s information theory tells us that the story contains less information for $A$ than for $B$, since it contains fewer surprises for $A$ than for $B$. Recall that $A$’s mind already has memories of similar plots and characters. As they read more and more, however, both $A$ and $B$ become more and more familiar with the plot and characters, and therefore less and less surprised (although at different rates). Thus, they get less and less (Shannon’s type of) information. At the same time, as more of the story is revealed to them, their minds’ complexities increase (again at different rates). Thus, they get more algorithmic information content. The sum of Shannon’s information and KCC is therefore constant (or close to constant).
This example suggests a way to measure the information content of the story in an absolute way, regardless of the particular reader. It is the sum of Shannon’s information and the KCC. This measure has been proposed by the physicist Wojciech Zurek [Zurek 89], who termed it “physical entropy.”

Information’s pretty thin stuff unless mixed with experience.
—Clarence Day The Crow’s Nest
Answers to Exercises

A bird does not sing because he has an answer, he sings because he has a song.
—Chinese Proverb

Intro.1: abstemious, abstentious, adventitious, annelidous, arsenious, arterious, facetious, sacrilegious.

Intro.2: When a software house has a popular product they tend to come up with new versions. A user can update an old version to a new one, and the update usually comes as a compressed file on a floppy disk. Over time the updates get bigger and, at a certain point, an update may not fit on a single floppy. This is why good compression is important in the case of software updates. The time it takes to compress and decompress the update is unimportant since these operations are typically done just once. Recently, software makers have taken to providing updates over the Internet, but even in such cases it is important to have small files because of the download times involved.

1.1: (1) ask a question, (2) absolutely necessary, (3) advance warning, (4) boiling hot, (5) climb up, (6) close scrutiny, (7) exactly the same, (8) free gift, (9) hot water heater, (10) my personal opinion, (11) newborn baby, (12) postponed until later, (13) unexpected surprise, (14) unsolved mysteries.

1.2: A reasonable way to use them is to code the five most-common strings in the text. Because irreversible text compression is a special-purpose method, the user may know what strings are common in any particular text to be compressed. The user may specify five such strings to the encoder, and they should also be written at the start of the output stream, for the decoder’s use.

1.3: 6,8,0,1,3,1,4,1,3,4,1,3,1,4,1,3,1,2,2,2,2,6,1,1. The first two are the bitmap resolution (6×8). If each number occupies a byte on the output stream, then its size is 25 bytes, compared to a bitmap size of only 6×8 bits = 6 bytes. The method does not work for small images.
1.4: RLE of images is based on the idea that adjacent pixels tend to be identical. The last pixel of a row, however, has no reason to be identical to the first pixel of the next row.

1.5: Each of the first four rows yields the eight runs 1, 1, 1, 2, 1, 1, 1, eol. Rows 6 and 8 yield the four runs 0, 7, 1, eol each. Rows 5 and 7 yield the two runs 8, eol each. The total number of runs (including the eol’s) is thus 44.

When compressing by columns, columns 1, 3, and 6 yield the five runs 5, 1, 1, 1, eol each. Columns 2, 4, 5, and 7 yield the six runs 0, 5, 1, 1, eol each. Column 8 gives 4, 4, eol, so the total number of runs is 42. This image is thus “balanced” with respect to rows and columns.

1.6: The result is five groups as follows:

\[
\begin{align*}
W_1 & \text{ to } W_2: 00000, 11111, \\
W_3 & \text{ to } W_{10}: 00001, 00011, 00111, 01111, 11110, 11100, 10000, \\
W_{11} & \text{ to } W_{22}: 00010, 00100, 01000, 00110, 01100, 01110, \\
& \text{ 11101, 11011, 10111, 11001, 10011, 10001}, \\
W_{23} & \text{ to } W_{30}: 01011, 10110, 01101, 11010, 10100, 01001, 10010, 00101, \\
W_{31} & \text{ to } W_{32}: 01010, 10101.
\end{align*}
\]

1.7: The seven codes are

\[
0000, 1111, 0001, 1110, 0000, 0011, 1111,
\]

forming a string with six runs. Applying the rule of complementing yields the sequence

\[
0000, 1111, 1110, 0000, 0011, 0000,
\]

with seven runs. The rule of complementing does not always reduce the number of runs.

1.8: As “11 22 90 00 00 33 44”. The 00 following the 90 indicates no run, and the following 00 is interpreted as a regular character.

1.9: The six characters “123ABC” have ASCII codes 31, 32, 33, 41, 42, and 43. Translating these hexadecimal numbers to binary produces “00110001 00110010 00110011 01000001 01000010 01000011”.

The next step is to divide this string of 48 bits into 6-bit blocks. They are 001100=12, 010011=19, 001000=8, 110011=51, 010000=16, 010100=20, 001001=9, and 000011=3.

The character at position 12 in the BinHex table is “-” (position numbering starts at zero). The one at position 19 is “6”. The final result is the string “-6$c38*$”.

1.10: Exercise A.1 shows that the binary code of the integer \(i\) is \(1 + \lfloor \log_2 i \rfloor\) bits long. We add \(\lfloor \log_2 i \rfloor\) zeros, bringing the total size to \(1 + 2 \lfloor \log_2 i \rfloor\) bits.
1.11: Table Ans.1 summarizes the results. In (a), the first string is encoded with $k = 1$. In (b) it is encoded with $k = 2$. Columns (c) and (d) are the encodings of the second string with $k = 1$ and $k = 2$, respectively. The averages of the four columns are 3.4375, 3.25, 3.56, and 3.6875; very similar! The move-ahead-$k$ method used with small values of $k$ does not favor strings satisfying the concentration property.

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a abcdmnop 0</td>
<td>a abcdmnop 0</td>
<td>a abcdmnop 0</td>
<td>a abcdmnop 0</td>
</tr>
<tr>
<td>b</td>
<td>b abcdmnop 1</td>
<td>b abcdmnop 1</td>
<td>b abcdmnop 1</td>
<td>b abcdmnop 1</td>
</tr>
<tr>
<td>c</td>
<td>c bacdmnop 2</td>
<td>c bacdmnop 2</td>
<td>c bacdmnop 2</td>
<td>c bacdmnop 2</td>
</tr>
<tr>
<td>d</td>
<td>d cbadmnop 3</td>
<td>d cbadmnop 3</td>
<td>d cbadmnop 3</td>
<td>d cbadmnop 3</td>
</tr>
<tr>
<td>d</td>
<td>d cbdamnop 2</td>
<td>d cbdamnop 1</td>
<td>m bedamnop 4</td>
<td>m edbamnop 4</td>
</tr>
<tr>
<td>c</td>
<td>c bdcmnop 2</td>
<td>c debcmnop 1</td>
<td>n bedcmnop 5</td>
<td>n edcmnop 5</td>
</tr>
<tr>
<td>b</td>
<td>b bcdmnop 0</td>
<td>b edcmnop 2</td>
<td>o bcdmenop 6</td>
<td>o edcmnop 6</td>
</tr>
<tr>
<td>a</td>
<td>a bedcmnop 3</td>
<td>a bedcmnop 3</td>
<td>p bedcmnop 7</td>
<td>p edcmnop 7</td>
</tr>
<tr>
<td>m</td>
<td>m bcadmnop 4</td>
<td>m badcmnop 4</td>
<td>a bcdmopna 7</td>
<td>a cdmnopab 7</td>
</tr>
<tr>
<td>n</td>
<td>n bcamdnop 5</td>
<td>n bamcdnp 5</td>
<td>b bcdmopab 0</td>
<td>b cdmnopab 7</td>
</tr>
<tr>
<td>o</td>
<td>o bcamnoepd 6</td>
<td>o b analep 6</td>
<td>c edcmnop 1</td>
<td>c edcmnop 0</td>
</tr>
<tr>
<td>p</td>
<td>p bcamn epd 7</td>
<td>p bamedcepd 7</td>
<td>d edcmnop 2</td>
<td>d edcmnop 1</td>
</tr>
<tr>
<td>p</td>
<td>p bcamnopd 6</td>
<td>p bamedcnopd 5</td>
<td>m edcmnop 3</td>
<td>m edcmnop 2</td>
</tr>
<tr>
<td>o</td>
<td>o bcamnepcd 6</td>
<td>o bamedcnpd 5</td>
<td>n edcmnop 4</td>
<td>n edcmnop 3</td>
</tr>
<tr>
<td>n</td>
<td>n bcamnpod 4</td>
<td>n bamedcnped 5</td>
<td>o edcmnop 5</td>
<td>o edcmnop 4</td>
</tr>
<tr>
<td>m</td>
<td>m bcamn poed 4</td>
<td>m bamedcnpod 2</td>
<td>p edcmnop 7</td>
<td>p edcmnop 7</td>
</tr>
<tr>
<td>bcamnpod</td>
<td>m banopced</td>
<td>cdnopoab</td>
<td>nnocdecpba</td>
<td></td>
</tr>
</tbody>
</table>

Table Ans.1: Encoding With Move-Ahead-$k$.

1.12: Table Ans.2 summarizes the decoding steps. Notice how similar it is to Table 1.16, indicating that move-to-front is a symmetric data compression method.

<table>
<thead>
<tr>
<th>Code input</th>
<th>A (before adding)</th>
<th>A (after adding)</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0the</td>
<td>()</td>
<td>(the)</td>
<td>the</td>
</tr>
<tr>
<td>1boy</td>
<td>(the)</td>
<td>(the, boy)</td>
<td>boy</td>
</tr>
<tr>
<td>2on</td>
<td>(boy, the)</td>
<td>(boy, the, on)</td>
<td>on</td>
</tr>
<tr>
<td>3my</td>
<td>(on, boy, the)</td>
<td>(on, boy, the, my)</td>
<td>my</td>
</tr>
<tr>
<td>4right</td>
<td>(my, on, boy, the)</td>
<td>(my, on, boy, the, right)</td>
<td>right</td>
</tr>
<tr>
<td>5is</td>
<td>(right, my, on, boy, the)</td>
<td>(right, my, on, boy, the, is)</td>
<td>is</td>
</tr>
<tr>
<td>5</td>
<td>(is, right, my, on, boy, the)</td>
<td>(is, right, my, on, boy, the)</td>
<td>the</td>
</tr>
<tr>
<td>2</td>
<td>(the, is, right, my, on, boy)</td>
<td>(the, is, right, my, on, boy)</td>
<td>right</td>
</tr>
<tr>
<td>5</td>
<td>(right, the, is, my, on, boy)</td>
<td>(right, the, is, my, on, boy)</td>
<td>boy</td>
</tr>
</tbody>
</table>

Table Ans.2: Decoding Multiple-Letter Words.
Answers to Exercises

<table>
<thead>
<tr>
<th>Prob.</th>
<th>Steps</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 0.25</td>
<td>1 1</td>
<td>:11</td>
</tr>
<tr>
<td>2. 0.20</td>
<td>1 0</td>
<td>:101</td>
</tr>
<tr>
<td>3. 0.15</td>
<td>1 0</td>
<td>:100</td>
</tr>
<tr>
<td>4. 0.15</td>
<td>0 1</td>
<td>:01</td>
</tr>
<tr>
<td>5. 0.10</td>
<td>0 0 1</td>
<td>:001</td>
</tr>
<tr>
<td>6. 0.10</td>
<td>0 0 0 0</td>
<td>:0001</td>
</tr>
<tr>
<td>7. 0.05</td>
<td>0 0 0 0</td>
<td>:0000</td>
</tr>
</tbody>
</table>

Table Ans.3: Shannon-Fano Example.

5.1: Subsequent splits can be done in different ways, but Table Ans.3 shows one way of assigning Shannon-Fano codes to the 7 symbols. The average size in this case is $0.25 \times 2 + 0.20 \times 3 + 0.15 \times 3 + 0.15 \times 2 + 0.10 \times 3 + 0.10 \times 4 + 0.05 \times 4 = 2.75$ bits/symbols.

5.2: The entropy is $-2(0.25 \times \log_2 0.25) - 4(0.125 \times \log_2 0.125) = 2.5$.

5.3: Figure Ans.4a,b,c shows the three trees. The codes sizes for the trees are

- $(5 + 5 + 5 + 5 \cdot 2 + 3 \cdot 3 + 3 \cdot 5 + 3 \cdot 5 + 12)/30 = 76/30$,
- $(5 + 5 + 4 + 4 \cdot 2 + 4 \cdot 3 + 3 \cdot 5 + 3 \cdot 5 + 12)/30 = 76/30$,
- $(6 + 6 + 5 + 4 \cdot 2 + 3 \cdot 3 + 3 \cdot 5 + 3 \cdot 5 + 12)/30 = 76/30$.

![Figure Ans.4: Three Huffman Trees for Eight Symbols.](image)

5.4: After adding symbols A, B, C, D, E, F, and G to the tree, we were left with the three symbols ABEF (with probability 10/30), CDG (with probability 8/30), and H (with probability 12/30). The two symbols with lowest probabilities were ABEF and
Answers to Exercises

CDG, so they had to be merged. Instead, symbols CDG and H were merged, creating a non-Huffman tree.

5.5: The second row of Table Ans.6 (due to Guy Blelloch) shows a symbol whose Huffman code is three bits long, but for which $\lceil -\log_2 0.3 \rceil = \lceil 1.737 \rceil = 2$.

<table>
<thead>
<tr>
<th>$P_i$</th>
<th>Code</th>
<th>$-\log_2 P_i$</th>
<th>$\lceil -\log_2 P_i \rceil$</th>
</tr>
</thead>
<tbody>
<tr>
<td>.01</td>
<td>000</td>
<td>6.644</td>
<td>7</td>
</tr>
<tr>
<td>*.30</td>
<td>001</td>
<td>1.737</td>
<td>2</td>
</tr>
<tr>
<td>.34</td>
<td>01</td>
<td>1.556</td>
<td>2</td>
</tr>
<tr>
<td>.35</td>
<td>1</td>
<td>1.515</td>
<td>2</td>
</tr>
</tbody>
</table>

Table Ans.6: A Huffman Code Example.

5.6: The explanation is simple. Imagine a large alphabet where all the symbols have (about) the same probability. Since the alphabet is large, that probability will be small, resulting in long codes. Imagine the other extreme case, where certain symbols have high probabilities (and, therefore, short codes). Since the probabilities have to add up to 1, the rest of the symbols will have low probabilities (and, therefore, long codes). We therefore see that the size of a code depends on the probability, but is indirectly affected by the size of the alphabet.

5.7: Figure Ans.7 shows Huffman codes for 5, 6, 7, and 8 symbols with equal probabilities. In the case where $n$ is a power of 2, the codes are simply the fixed-sized ones. In other cases the codes are very close to fixed-length. This shows that symbols with equal probabilities do not benefit from variable-length codes. (This is another way of saying that random text cannot be compressed.) Table Ans.8 shows the codes, their average sizes and variances.
Figure Ans.7: Huffman Codes for Equal Probabilities.

Table Ans.8: Huffman Codes for 5–8 Symbols.
5.8: It increases exponentially from $2^s$ to $2^{s+n} = 2^s \times 2^n$.

5.9: The binary value of 127 is 01111111 and that of 128 is 10000000. Half the pixels in each bitplane will therefore be 0 and the other half, 1. In the worst case, each bitplane will be a checkerboard, i.e., will have many runs of size one. In such a case, each run requires a 1-bit code, leading to one codebit per pixel per bitplane, or eight codebits per pixel for the entire image, resulting in no compression at all. In comparison, a Huffman code for such an image requires just two codes (since there are just two pixel values) and they can be one bit each. This leads to one codebit per pixel, or a compression factor of eight.

5.10: The two trees are shown in Figure 5.16c,d. The average code size for the binary Huffman tree is

$$1 \times 0.49 + 2 \times 0.25 + 5 \times 0.02 + 5 \times 0.03 + 5 \times 0.04 + 5 \times 0.04 + 3 \times 0.12 = 2 \text{ bits/symbol},$$

and that of the ternary tree is

$$1 \times 0.26 + 3 \times 0.02 + 3 \times 0.03 + 3 \times 0.04 + 2 \times 0.04 + 2 \times 0.12 + 1 \times 0.49 = 1.34 \text{ trits/symbol}.$$ 

5.11: Figure Ans.9 shows how the loop continues until the heap shrinks to just one node that is the single pointer 2. This indicates that the total frequency (which happens to be 100 in our example) is stored in $A[2]$. All other frequencies have been replaced by pointers. Figure Ans.10a shows the heaps generated during the loop.

5.12: The final result of the loop is

$$\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\end{array}$$

from which it is easy to figure out the code lengths of all seven symbols. To find the length of the code of symbol 14, e.g., we follow the pointers 7, 5, 3, 2 from $A[14]$ to the root. Four steps are necessary, so the code length is 4.

5.13: The code lengths for the seven symbols are 2, 2, 3, 4, 3, and 4 bits. This can also be verified from the Huffman code-tree of Figure Ans.10b. A set of codes derived from this tree is shown in the following table:

<table>
<thead>
<tr>
<th>Count:</th>
<th>25 20 13 17 9 11 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code:</td>
<td>01 11 101 000 0011 100 0010</td>
</tr>
<tr>
<td>Length:</td>
<td>2 2 3 3 4 3 4</td>
</tr>
</tbody>
</table>

5.14: A symbol with high frequency of occurrence should be assigned a shorter code. Therefore, it has to appear high in the tree. The requirement that at each level the frequencies be sorted from left to right is artificial. In principle, it is not necessary, but it simplifies the process of updating the tree.
Answers to Exercises

1214

1 [7 11 6 8 9] 24 14 25 20 6 17 7 6 7

1 [11 9 8 6] 2 41 42 52 0 6 17 7 6 7

1 [17+14] 2 41 42 52 0 6 17 7 6 7

1 [1 1986] 17+14 24 14 25 20 6 17 7 6 7

1 5986 3 1 2 4 5 25 20 6 17 7 6 7

1 9685 3 1 2 4 5 25 20 6 17 7 6 7

1 685 20+24 31 24 5 25 20 6 17 7 6 7

1 [6 8 5] 4 4 3 1 45 25 46 5 7 6 7

1 485 4 4 3 1 45 25 46 5 7 6 7

1 854 4 4 3 1 45 25 46 5 7 6 7

1 54 25+31 44 31 4 5 25 20 6 17 7 6 7

1 54 25+31 44 31 4 5 25 20 6 17 7 6 7

1 [6 8 5] 20+24 31 24 5 25 20 6 17 7 6 7

1 [4 8 5] 44 31 4 5 25 4 6 5 7 6 7

1 [8 5 4] 44 31 4 5 25 4 6 5 7 6 7

1 [5 4] 44 31 4 5 25 4 6 5 7 6 7

1 [5 4] 44 31 4 5 25 4 6 5 7 6 7

1 [6 8 5] 20+24 31 24 5 25 20 6 17 7 6 7

1 [4 8 5] 44 31 4 5 25 4 6 5 7 6 7

1 [8 5 4] 44 31 4 5 25 4 6 5 7 6 7

1 [5 4] 44 31 4 5 25 4 6 5 7 6 7

1 [6 8 5] 20+24 31 24 5 25 20 6 17 7 6 7

1 [4 8 5] 44 31 4 5 25 4 6 5 7 6 7

1 [8 5 4] 44 31 4 5 25 4 6 5 7 6 7

1 [5 4] 44 31 4 5 25 4 6 5 7 6 7

1 [3 4] 56 44 3 4 5 3 4 5 3 4 6 5 7 6 7

1 [3 4] 56 44 3 4 5 3 4 5 3 4 6 5 7 6 7

1 [3 4] 56 44 3 4 5 3 4 5 3 4 6 5 7 6 7

1 [3 4] 56 44 3 4 5 3 4 5 3 4 6 5 7 6 7

1 [3 4] 56 44 3 4 5 3 4 5 3 4 6 5 7 6 7

1 [3 4] 56 44 3 4 5 3 4 5 3 4 6 5 7 6 7

1 [2] 100 2 2 3 4 5 3 4 6 5 7 6 7

Figure Ans.9: Sifting the Heap.
Figure Ans.10: (a) Heaps. (b) Huffman Code-Tree.
5.15: Figure Ans.11 shows the initial tree and how it is updated in the 11 steps (a) through (k). Notice how the esc symbol gets assigned different codes all the time, and how the different symbols move about in the tree and change their codes. Code 10, e.g., is the code of symbol “i” in steps (f) and (i), but is the code of “s” in steps (e) and (j). The code of a blank space is 011 in step (h), but 00 in step (k).

The final output is: “s0i00r100␣1010000d011101000”. A total of $5 \times 8 + 22 = 62$ bits. The compression ratio is thus $62/88 \approx 0.7$.

5.16: A simple calculation shows that the average size of a token in Table 5.25 is about nine bits. In stage 2, each 8-bit byte will be replaced, on average, by a 9-bit token, resulting in an expansion factor of $9/8 = 1.125$ or 12.5%.

5.17: The decompressor will interpret the input data as 111110 0110 11000 0..., which is the string XRP....

5.18: Because a typical fax machine scans lines that are about 8.2 inches wide ($\approx 208$ mm), so a blank scan line produces 1,664 consecutive white pels.

5.19: These codes are needed for cases such as example 4, where the run length is 64, 128, or any length for which a make-up code has been assigned.

5.20: There may be fax machines (now or in the future) built for wider paper, so the Group 3 code was designed to accommodate them.

5.21: Each scan line starts with a white pel, so when the decoder inputs the next code it knows whether it is for a run of white or black pels. This is why the codes of Table 5.31 have to satisfy the prefix property in each column but not between the columns.

5.22: Imagine a scan line where all runs have length one (strictly alternating pels). It’s easy to see that this case results in expansion. The code of a run length of one white pel is 000111, and that of one black pel is 010. Two consecutive pels of different colors are thus coded into 9 bits. Since the uncoded data requires just two bits (01 or 10), the compression ratio is $9/2 = 4.5$ (the compressed stream is 4.5 times longer than the uncompressed one; a large expansion).

5.23: Figure Ans.12 shows the modes and the actual code generated from the two lines.
\[ \text{esc } s_1 \]

(b). Input: i. Output: 0’i’.
\[ \text{esc } i_1 s_1 \]

(c). Input: r. Output: 00’r’.
\[ \text{esc } r_1 i_1 s_1 2 \]

(d). Input: \( \uparrow \). Output: 100’\( \uparrow \)’.
\[ \text{esc } \uparrow_1 r_1 i_1 s_1 3 \rightarrow \text{esc } \uparrow_1 r_1 s_1 i_1 22 \]

Figure Ans.11: Exercise 5.15. Part I.
\[ \text{esc}_{\omega_1} 1 \ r_1 \ s_2 \ i_1 \ 2 \ 3 \rightarrow \ \text{esc}_{\omega_1} 1 \ r_1 \ i_1 \ s_2 \ 2 \ 3 \]

\[ \text{esc}_{\omega_1} 1 \ r_1 \ i_2 \ s_2 \ 2 \ 4 \]

(g). Input: d. Output: 000’d’.
\[ \text{esc} \ d_1 \ 1 \ \omega_1 \ 2 \ r_1 \ i_2 \ s_2 \ 3 \ 4 \rightarrow \ \text{esc} \ d_1 \ 1 \ \omega_1 \ r_1 \ i_2 \ s_2 \ 3 \ 4 \]

Figure Ans.11: Exercise 5.15. Part II.
(h). Input: \( \uparrow \). Output: 011.
\[
\text{esc } d_1 \downarrow_2 r_1 \triangleright_3 i_2 \downarrow_2 s_2 44 \rightarrow \\
\text{esc } d_1 \downarrow_1 r_1 \downarrow_2 \triangleright_2 i_2 \downarrow_2 s_2 44
\]

\[
\text{esc } d_1 \downarrow_1 \uparrow_1 r_1 \downarrow_2 s_2 45 \rightarrow \\
\text{esc } d_1 \downarrow_1 r_1 \downarrow_2 \triangleright_2 s_2 i_3 45
\]
\[ \text{esc } d_1 r_1 \uparrow 2 s_3 i_3 4 6 \]

(k). Input: \( \uparrow 3 \). Output: 00.
\[ \text{esc } d_1 r_1 \uparrow 3 \downarrow 2 s_3 i_3 5 6 \rightarrow \text{esc } d_1 r_1 \uparrow 3 s_3 i_3 5 6 \]

Figure Ans.11: Exercise 5.15. Part IV.
5.24: Table Ans.13 shows the steps of encoding the string $a_2a_2a_2a_2$. Because of the high probability of $a_2$ the low and high variables start at very different values and approach each other slowly.

\[
\begin{align*}
a_2 & : 0.0 + (1.0 - 0.0) \times 0.023162 = 0.023162 \\
& : 0.0 + (1.0 - 0.0) \times 0.998162 = 0.998162 \\
a_2 & : 0.023162 + .975 \times 0.023162 = 0.04574495 \\
& : 0.023162 + .975 \times 0.998162 = 0.99636995 \\
a_2 & : 0.04574495 + 0.950625 \times 0.023162 = 0.0677632625 \\
& : 0.04574495 + 0.950625 \times 0.998162 = 0.99462270125 \\
a_2 & : 0.0677632625 + 0.926859375 \times 0.023162 = 0.08923124309375 \\
& : 0.0677632625 + 0.926859375 \times 0.998162 = 0.99291913371875 \\
\end{align*}
\]

Table Ans.13: Encoding the String $a_2a_2a_2a_2$.

5.25: It can be written either as 0.1000... or 0.0111....

5.26: In practice, the eof symbol has to be included in the original table of frequencies and probabilities. This symbol is the last to be encoded, and the decoder stops when it detects an eof.

5.27: The encoding steps are simple (see first example on page 265). We start with the interval $[0, 1)$. The first symbol $a_2$ reduces the interval to $[0.4, 0.9)$. The second one, to $[0.6, 0.85)$, the third one to $[0.7, 0.825)$ and the eof symbol, to $[0.8125, 0.8250)$. The approximate binary values of the last interval are 0.1101000000 and 0.1101001100, so we select the 7-bit number 1101000 as our code.

The probability of $a_2a_2a_2$eof is $(0.5)^3 \times 0.1 = 0.0125$, but since $-\log_2 0.0125 \approx 6.322$ it follows that the practical minimum code size is 7 bits.

5.28: Perhaps the simplest way to do this is to compute a set of Huffman codes for the symbols, using their probabilities. This converts each symbol to a binary string, so the input stream can be encoded by the QM-coder. After the compressed stream is decoded by the QM-decoder, an extra step is needed, to convert the resulting binary strings back to the original symbols.

5.29: The results are shown in Tables Ans.14 and Ans.15. When all symbols are LPS, the output $C$ always points at the bottom $A(1 - Qe)$ of the upper (LPS) subinterval. When the symbols are MPS, the output always points at the bottom of the lower (MPS) subinterval, i.e., 0.

5.30: If the current input bit is an LPS, $A$ is shrunk to $Qe$, which is always 0.5 or less, so $A$ always has to be renormalized in such a case.

5.31: The results are shown in Tables Ans.16 and Ans.17 (compare with the answer to exercise 5.29).
### Answers to Exercises

<table>
<thead>
<tr>
<th>Symbol</th>
<th>CA</th>
<th>Initially</th>
<th>$C$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1 (LPS)</td>
<td>0 + 1(1 − 0.5) = 0.5</td>
<td>1 × 0.5 = 0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2 (LPS)</td>
<td>0.5 + 0.5(1 − 0.5) = 0.75</td>
<td>0.5 × 0.5 = 0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3 (LPS)</td>
<td>0.75 + 0.25(1 − 0.5) = 0.875</td>
<td>0.25 × 0.5 = 0.125</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4 (LPS)</td>
<td>0.875 + 0.125(1 − 0.5) = 0.9375</td>
<td>0.125 × 0.5 = 0.0625</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table Ans.14: Encoding Four Symbols With $Q_e = 0.5$.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>CA</th>
<th>Initially</th>
<th>$C$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1 (MPS)</td>
<td>0 + 1(1 − 0.1) = 0.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2 (MPS)</td>
<td>0.9 × (1 − 0.1) = 0.81</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3 (MPS)</td>
<td>0.81 × (1 − 0.1) = 0.729</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4 (MPS)</td>
<td>0.729 × (1 − 0.1) = 0.6561</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table Ans.15: Encoding Four Symbols With $Q_e = 0.1$.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>CA</th>
<th>Initially</th>
<th>$C$</th>
<th>$A$</th>
<th>Renor. A</th>
<th>Renor. C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1 (LPS)</td>
<td>0 + 1 − 0.5 = 0.5</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2 (LPS)</td>
<td>1 + 1 − 0.5 = 1.5</td>
<td>0.5</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3 (LPS)</td>
<td>3 + 1 − 0.5 = 3.5</td>
<td>0.5</td>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4 (LPS)</td>
<td>7 + 1 − 0.5 = 6.5</td>
<td>0.5</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table Ans.16: Renormalization Added to Table Ans.14.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>CA</th>
<th>Initially</th>
<th>$C$</th>
<th>$A$</th>
<th>Renor. A</th>
<th>Renor. C</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1 (MPS)</td>
<td>0 + 1 − 0.1 = 0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2 (MPS)</td>
<td>0.9 − 0.1 = 0.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3 (MPS)</td>
<td>0.8 − 0.1 = 0.7</td>
<td>1.4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s4 (MPS)</td>
<td>1.4 − 0.1 = 1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table Ans.17: Renormalization Added to Table Ans.15.
5.32: The four decoding steps are as follows:

**Step 1:** \( C = 0.981, A = 1 \), the dividing line is \( A(1 - Qe) = 1(1 - 0.1) = 0.9 \), so the LPS and MPS subintervals are \([0, 0.9)\) and \([0.9, 1)\). Since \( C \) points to the upper subinterval, an LPS is decoded. The new \( C \) is \( 0.981 - 1(1 - 0.1) = 0.81 \) and the new \( A \) is \( 1 \times 0.1 = 0.1 \).

**Step 2:** \( C = 0.81, A = 0.1 \), the dividing line is \( A(1 - Qe) = 0.1(1 - 0.1) = 0.09 \), so the LPS and MPS subintervals are \([0, 0.09)\) and \([0.09, 0.1)\), and an MPS is decoded. \( C \) is unchanged and the new \( A \) is \( 0.09 \times 0.1 = 0.009 \).

**Step 3:** \( C = 0.081, A = 0.09 \), the dividing line is \( A(1 - Qe) = 0.09(1 - 0.1) = 0.0081 \), so the LPS and MPS subintervals are \([0, 0.0081)\) and \([0.0081, 0.09)\), and an LPS is decoded. The new \( C \) is \( 0.081 - 0.09(1 - 0.1) = 0 \) and the new \( A \) is \( 0.09 \times 0.1 = 0.009 \).

**Step 4:** \( C = 0, A = 0.009 \), the dividing line is \( A(1 - Qe) = 0.009(1 - 0.1) = 0.00081 \), so the LPS and MPS subintervals are \([0, 0.00081)\) and \([0.00081, 0.009)\), and an MPS is decoded. \( C \) is unchanged and the new \( A \) is \( 0.009(1 - 0.1) = 0.00081 \).

5.33: In practice, an encoder may encode texts other than English, such as a foreign language or the source code of a computer program. Acronyms, such as QED and abbreviations, such as qwerty, are also good examples. Even in English texts there are many examples of a \( q \) not followed by a \( u \), such as in words transliterated from other languages, most commonly Arabic and Chinese. Examples are \( suq \) (market) and \( qadi \) (a Moslem judge). See [PQ-wiki 09] for more examples.

5.34: The number of order-2 and order-3 contexts for an alphabet of size \( 2^8 = 256 \) is \( 256^2 = 65,536 \) and \( 256^3 = 16,777,216 \), respectively. The former is manageable, whereas the latter is perhaps too big for a practical implementation, unless a sophisticated data structure is used or unless the encoder gets rid of older data from time to time.

For a small alphabet, larger values of \( N \) can be used. For a 16-symbol alphabet there are \( 16^4 = 65,536 \) order-4 contexts and \( 16^6 = 16,777,216 \) order-6 contexts.

5.35: A practical example of a 16-symbol alphabet is a color or grayscale image with 4-bit pixels. Each symbol is a pixel, and there are 16 different symbols.

5.36: An object file generated by a compiler or an assembler normally has several distinct parts including the machine instructions, symbol table, relocation bits, and constants. Such parts may have different bit distributions.

5.37: The alphabet has to be extended, in such a case, to include one more symbol. If the original alphabet consisted of all the possible 256 8-bit bytes, it should be extended to 9-bit symbols, and should include 257 values.

5.38: Table Ans.18 shows the groups generated in both cases and makes it clear why these particular probabilities were assigned.

5.39: The \( d \) is added to the order-0 contexts with frequency 1. The escape frequency should be incremented from 5 to 6, bringing the total frequencies from 19 up to 21. The probability assigned to the new \( d \) is therefore \( 1/21 \), and that assigned to the escape is \( 6/21 \). All other probabilities are reduced from \( x/19 \) to \( x/21 \).
Table Ans.18: Stable vs. Variable Data.

5.40: The new $d$ would require switching from order 2 to order 0, sending two escapes that take 1 and 1.32 bits. The $d$ is now found in order-0 with probability $1/21$, so it is encoded in 4.39 bits. The total number of bits required to encode the second $d$ is therefore $1 + 1.32 + 4.39 = 6.71$, still greater than 5.

5.41: The first three cases don’t change. They still code a symbol with 1, 1.32, and 6.57 bits, which is less than the 8 bits required for a 256-symbol alphabet without compression. Case 4 is different since the $d$ is now encoded with a probability of $1/256$, producing 8 instead of 4.8 bits. The total number of bits required to encode the $d$ in case 4 is now $1 + 1.32 + 1.93 + 8 = 12.25$.

5.42: The final trie is shown in Figure Ans.19.
5.43: This probability is, of course

$$1 - P_e(b_{t+1} = 1|b_t) = 1 - \frac{b + 1/2}{a + b + 1} = \frac{a + 1/2}{a + b + 1}.$$ 

5.44: For the first string the single bit has a suffix of 00, so the probability of leaf 00 is $P_e(1, 0) = 1/2$. This is equal to the probability of string 0 without any suffix. For the second string each of the two zero bits has suffix 00, so the probability of leaf 00 is $P_e(2, 0) = 3/8 = 0.375$. This is greater than the probability 0.25 of string 00 without any suffix. Similarly, the probabilities of the remaining three strings are $P_e(3, 0) = 5/8 \approx 0.625$, $P_e(4, 0) = 35/128 \approx 0.273$, and $P_e(5, 0) = 63/256 \approx 0.246$. As the strings get longer, their probabilities get smaller but they are greater than the probabilities without the suffix. Having a suffix of 00 thus increases the probability of having strings of zeros following it.

5.45: The four trees are shown in Figure Ans.20a–d. The weighted probability that the next bit will be a zero given that three zeros have just been generated is 0.5. The weighted probability to have two consecutive zeros given the suffix 000 is 0.375, higher than the 0.25 without the suffix.

6.1: The size of the output stream is $N[48 - 28P] = N[48 - 25.2] = 22.8N$. The size of the input stream is, as before, $40N$. The compression factor is therefore $40/22.8 \approx 1.75$.

6.2: The list has up to 256 entries, each consisting of a byte and a count. A byte occupies eight bits, but the counts depend on the size and content of the file being compressed. If the file has high redundancy, a few bytes may have large counts, close to the length of the file, while other bytes may have very low counts. On the other hand, if the file is close to random, then each distinct byte has approximately the same count.

Thus, the first step in organizing the list is to reserve enough space for each “count” field to accommodate the maximum possible count. We denote the length of the file by $L$ and find the positive integer $k$ that satisfies $2^{k-1} < L \leq 2^k$. Thus, $L$ is a $k$-bit number. If $k$ is not already a multiple of 8, we increase it to the next multiple of 8. We now denote $k = 8m$, and allocate $m$ bytes to each “count” field.

Once the file has been input and processed and the list has been sorted, we examine the largest count. It may be large and may occupy all $m$ bytes, or it may be smaller. Assuming that the largest count occupies $n$ bytes (where $n \leq m$), we can store each of the other counts in $n$ bytes.

When the list is written on the compressed file as the dictionary, its length $s$ is first written in one byte. $s$ is the number of distinct bytes in the original file. This is followed by $n$, followed by $s$ groups, each with one of the distinct data bytes followed by an $n$-byte count. Notice that the value $n$ should be fairly small and should fit in a single byte. If $n$ does not fit in a single byte, then it is greater than 255, implying that the largest count does not fit in 255 bytes, implying in turn a file whose length $L$ is greater than $2^{255} \approx 10^{76}$ bytes.

An alternative is to start with $s$, followed by $n$, followed by the $s$ distinct data bytes, followed by the $n \times s$ bytes of counts. The last part could also be in compressed
form, because only a few largest counts will occupy all $n$ bytes. Most counts may be small and occupy just one or two bytes, which implies that many of the $n \times s$ count bytes will be zero, resulting in high redundancy and therefore good compression.

6.3: The straightforward answer is The decoder doesn’t know but it does not need to know. The decoder simply reads tokens and uses each offset to locate a string of text without having to know whether the string was a first or a last match.

6.4: The next step matches the space and encodes the string `i.e.`

$$\text{sir|sid|eastman|easily,} \quad \Rightarrow \quad (4,1,e)$$

$$\text{sir|sid|e|eastman|easily, te} \quad \Rightarrow \quad (0,0,a)$$

and the next one matches nothing and encodes the `a`.

6.5: The first two characters `CA` at positions 17–18 are a repeat of the `CA` at positions 9–10, so they will be encoded as a string of length 2 at offset $18 - 10 = 8$. The next two
characters AC at positions 19–20 are a repeat of the string at positions 8–9, so they will be encoded as a string of length 2 at offset 20 − 9 = 11.

6.6: The decoder interprets the first 1 of the end marker as the start of a token. The second 1 is interpreted as the prefix of a 7-bit offset. The next 7 bits are 0, and they identify the end marker as such, since a “normal” offset cannot be zero.

6.7: This is straightforward. The remaining steps are shown in Table Ans.21

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Token</th>
<th>Dictionary</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 u</td>
<td>t</td>
<td>21 _</td>
<td>si</td>
</tr>
<tr>
<td>16 e</td>
<td>0, e</td>
<td>22 c</td>
<td>(0, c)</td>
</tr>
<tr>
<td>17 as</td>
<td>8, s</td>
<td>23 k</td>
<td>(0, k)</td>
</tr>
<tr>
<td>18 es</td>
<td>16,s</td>
<td>24 _</td>
<td>se</td>
</tr>
<tr>
<td>19 _</td>
<td>s</td>
<td>25 al</td>
<td>(8, l)</td>
</tr>
<tr>
<td>20 ea</td>
<td>4, a</td>
<td>26 s(eof)</td>
<td>(1, (eof))</td>
</tr>
</tbody>
</table>

Table Ans.21: Next 12 Encoding Steps in the LZ78 Example.

6.8: Table Ans.22 shows the last three steps.

<table>
<thead>
<tr>
<th>p_src</th>
<th>3 chars</th>
<th>Hash</th>
<th>Output</th>
<th>Binary output</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 h t</td>
<td>7</td>
<td>any→11</td>
<td>h</td>
<td>01101000</td>
</tr>
<tr>
<td>12 _</td>
<td>th</td>
<td>5</td>
<td>5→12</td>
<td>4,7</td>
</tr>
<tr>
<td>16 ws</td>
<td></td>
<td></td>
<td></td>
<td>01110111</td>
</tr>
</tbody>
</table>

Table Ans.22: Last Steps of Encoding that thatch thaws.

The final compressed stream consists of 1 control word followed by 11 items (9 literals and 2 copy items)
0000010010000000|01110100|01101000|01100001|01110100|00100000|0000|0011 |00000001|01100001|01110000|0000|0011|01110011|01110011.

6.9: An example is a compression utility for a personal computer that maintains all the files (or groups of files) on the hard disk in compressed form, to save space. Such a utility should be transparent to the user; it should automatically decompress a file every time it is opened and automatically compress it when it is being closed. In order to be transparent, such a utility should be fast, with compression ratio being only a secondary feature.

6.10: Table Ans.23 summarizes the steps. The output emitted by the encoder is 97 (a), 108 (l), 102 (f), 32 ( ), 101 (e), 97 (a), 116 (t), 115 (s), 32 ( ), 256 (al), 102 (f), 265 (alf), 97 (a), and the following new entries are added to the dictionary (256: al), (257: lf), (258: f ), (259: se), (260: ea), (261: at), (262: ts), (263: s ), (264: a), (265: alf), (266: fa), (267: alfa).
6.11: The encoder inputs the first a into I, searches and finds a in the dictionary. It inputs the next a but finds that Ix, which is now aa, is not in the dictionary. The encoder thus adds string aa to the dictionary as entry 256 and outputs the token 97 (a). Variable I is initialized to the second a. The third a is input, so Ix is the string aa, which is now in the dictionary. I becomes this string, and the fourth a is input. Ix is now aaa which is not in the dictionary. The encoder thus adds string aaa to the dictionary as entry 257 and outputs 256 (aa). I is initialized to the fourth a. Continuing this process is straightforward.

The result is that strings aa, aaa, aaaa,... are added to the dictionary as entries 256, 257, 258,..., and the output is

97 (a), 256 (aa), 257 (aaa), 258 (aaaa),...

The output consists of pointers pointing to longer and longer strings of as. The first k pointers thus point at strings whose total length is $1 + 2 + \cdots + k = (k + k^2)/2$.

Assuming an input stream that consists of one million as, we can find the size of the compressed output stream by solving the quadratic equation $(k + k^2)/2 = 1000000$ for the unknown $k$. The solution is $k \approx 1414$. The original, 8-million bit input is thus compressed into 1414 pointers, each at least 9-bit (and in practice, probably 16-bit) long. The compression factor is thus either $8M/(1414 \times 9) \approx 628.6$ or $8M/(1414 \times 16) \approx 353.6$.

This is an impressive result but such input streams are rare (notice that this particular input can best be compressed by generating an output stream containing just “1000000 a”, and without using LZW).

6.12: We simply follow the decoding steps described in the text. The results are:
1. Input 97. This is in the dictionary so set I=a and output a. String ax needs to be saved in the dictionary but x is still unknown.
2. Input 108. This is in the dictionary so set $J=1$ and output 1. Save $a_1$ in entry 256. Set $I=1$.
3. Input 102. This is in the dictionary so set $J=f$ and output $f$. Save $1f$ in entry 257. Set $I=f$.
4. Input 32. This is in the dictionary so set $J=\_\_\_$ and output $\_\_\_$. Save $\_\_\_1$ in entry 258. Set $I=\_\_\_\_\_$.
5. Input 101. This is in the dictionary so set $J=e$ and output $e$. Save $\_\_\_e$ in entry 259. Set $I=e$.
6. Input 97. This is in the dictionary so set $J=a$ and output $a$. Save $a\_\_\_$ in entry 260. Set $I=a$.
7. Input 116. This is in the dictionary so set $J=t$ and output $t$. Save $at$ in entry 261. Set $I=t$.
8. Input 115. This is in the dictionary so set $J=s$ and output $s$. Save $ts$ in entry 262. Set $I=t$.
9. Input 32. This is in the dictionary so set $J=\_\_\_$ and output $\_\_\_$. Save $\_\_\_s$ in entry 263. Set $I=\_\_\_\_\_$.
10. Input 256. This is in the dictionary so set $J=a1$ and output $a1$. Save $\_\_\_a$ in entry 264. Set $I=a1$.
11. Input 102. This is in the dictionary so set $J=f$ and output $f$. Save $alf$ in entry 265. Set $I=f$.
12. Input 265. This has just been saved in the dictionary so set $J=alf$ and output $alf$. Save $fa$ in dictionary entry 266. Set $I=alf$.
13. Input 97. This is in the dictionary so set $J=a$ and output $a$. Save $alfa$ in entry 267 (even though it will never be used). Set $I=a$.

6.13: We assume that the dictionary is initialized to just the two entries (1: a) and (2: b). The encoder outputs

1 (a), 2 (b), 3 (ab), 5(ab), 4(ba), 7 (bab), 6 (abab), 9 (ababa), 8 (baba), ...

and adds the new entries (3: ab), (4: ba), (5: ab), (6: ab), (7: bab), (8: baba), (9: ababa), (10: ababab), (11: babab),... to the dictionary. This regular behavior can be analyzed and the $k$th output pointer and dictionary entry predicted, but the effort is probably not worth it.

6.14: The answer to exercise 6.11 shows the relation between the size of the compressed file and the size of the largest dictionary string for the “worst case” situation (input that creates the longest strings). For a 1 Mbyte input stream, there will be 1,414 strings in the dictionary, the largest of which is 1,414 symbols long.

6.15: This is straightforward (Table Ans.24) but not very efficient since only one two-symbol dictionary phrase is used.

6.16: Table Ans.25 shows all the steps. In spite of the short input, the result is quite good (13 codes to compress 18-symbols) because the input contains concentrations of as and bs.
Answers to Exercises

6.17: 1. The encoder starts by shifting the first two symbols xy to the search buffer, outputting them as literals and initializing all locations of the index table to the null pointer.

2. The current symbol is a (the first a) and the context is xy. It is hashed to, say, 5, but location 5 of the index table contains a null pointer, so P is null. Location 5 is set to point to the first a, which is then output as a literal. The data in the encoder's buffer is shifted to the left.

3. The current symbol is the second a and the context is ya. It is hashed to, say, 1, but location 1 of the index table contains a null pointer, so P is null. Location 1 is set to point to the second a, which is then output as a literal. The data in the encoder's buffer is shifted to the left.

4. The current symbol is the third a and the context is aa. It is hashed to, say, 2, but
location 2 of the index table contains a null pointer, so $P$ is null. Location 2 is set to
point to the third $a$, which is then output as a literal. The data in the encoder’s buffer
is shifted to the left.

5. The current symbol is the fourth $a$ and the context is $aa$. We know from step 4 that
it is hashed to 2, and location 2 of the index table points to the third $a$. Location 2 is
set to point to the fourth $a$, and the encoder tries to match the string starting with the
third $a$ to the string starting with the fourth $a$. Assuming that the look-ahead buffer is
full of $a$s, the match length $L$ will be the size of that buffer. The encoded value of $L$
will be written to the compressed stream, and the data in the buffer shifted $L$ positions
to the left.

6. If the original input stream is long, more $a$’s will be shifted into the look-ahead buffer,
and this step will also result in a match of length $L$. If only $n$ as remain in the input
stream, they will be matched, and the encoded value of $n$ output.

The compressed stream will consist of the three literals $x$, $y$, and $a$, followed by
(perhaps several values of) $L$, and possibly ending with a smaller value.

6.18: $T$ percent of the compressed stream is made up of literals, some appearing
consecutively (and thus getting the flag “1” for two literals, half a bit per literal) and
others with a match length following them (and thus getting the flag “01”, one bit for
the literal). We assume that two thirds of the literals appear consecutively and one third
are followed by match lengths. The total number of flag bits created for literals is thus

$$\frac{2}{3}T \times 0.5 + \frac{1}{3}T \times 1.$$ 

A similar argument for the match lengths yields

$$\frac{2}{3}(1 - T) \times 2 + \frac{1}{3}(1 - T) \times 1$$

for the total number of the flag bits. We now write the equation

$$\frac{2}{3}T \times 0.5 + \frac{1}{3}T \times 1 + \frac{2}{3}(1 - T) \times 2 + \frac{1}{3}(1 - T) \times 1 = 1,$$

which is solved to yield $T = 2/3$. This means that if two thirds of the items in the
compressed stream are literals, there would be 1 flag bit per item on the average. More
literals would result in fewer flag bits.

6.19: The first three 1’s indicate six literals. The following 01 indicates a literal ($b$)
followed by a match length (of 3). The 10 is the code of match length 3, and the last 1
indicates two more literals ($x$ and $y$).

7.1: An image with no redundancy is not always random. The definition of redundancy
(Section A.1) tells us that an image where each color appears with the same frequency
has no redundancy (statistically) yet it is not necessarily random and may even be
interesting and/or useful.
7.2: Figure Ans.26 shows two $32 \times 32$ matrices. The first one, $a$, with random (and therefore decorrelated) values and the second one, $b$, is its inverse (and therefore with correlated values). Their covariance matrices are also shown and it is obvious that matrix $\text{cov}(a)$ is close to diagonal, whereas matrix $\text{cov}(b)$ is far from diagonal. The Matlab code for this figure is also listed.

```
a = rand(32); b = inv(a);
figure(1), imagesc(a), colormap(gray); axis square
figure(2), imagesc(b), colormap(gray); axis square
figure(3), imagesc(cov(a)), colormap(gray); axis square
figure(4), imagesc(cov(b)), colormap(gray); axis square
```

Figure Ans.26: Covariance Matrices of Correlated and Decorrelated Values.
The results are shown in Table Ans.27 together with the Matlab code used to calculate it.

One feature is the regular way in which each of the five code bits alternates periodically between 0 and 1. It is easy to write a program that will set all five bits to 0, will flip the rightmost bit after two codes have been calculated, and will flip any of the other four code bits in the middle of the period of its immediate neighbor on the right. Another feature is the fact that the second half of the table is a mirror image of the first half, but with the most significant bit set to one. After the first half of the table has been computed, using any method, this symmetry can be used to quickly calculate the second half.

Figure Ans.28 is an angular code wheel representation of the 4-bit and 6-bit RGC codes (part a) and the 4-bit and 6-bit binary codes (part b). The individual bitplanes are shown as rings, with the most significant bits as the innermost ring. It is easy to see that the maximum angular frequency of the RGC is half that of the binary code and that the first and last codes also differ by just one bit.

If pixel values are in the range [0, 255], a difference \((P_i - Q_i)\) can be at most 255. The worst case is where all the differences are 255. It is easy to see that such a case yields an RMSE of 255.

The code of Figure 7.16 yields the coordinates of the rotated points

\[
(7.071, 0), \ (9.19, 0.7071), \ (17.9, 0.78), \ (33.9, 1.41), \ (43.13, -2.12)
\]

(notice how all the \(y\) coordinates are small numbers) and shows that the cross-correlation drops from 1729.72 before the rotation to \(-23.0846\) after it. A significant reduction!

Figure Ans.29 shows the 64 basis images and the Matlab code to calculate and display them. Each basis image is an 8 \(\times\) 8 matrix.
7.9: \( A_4 \) is the \( 4 \times 4 \) matrix

\[
A_4 = \begin{pmatrix}
  h_0(0/4) & h_0(1/4) & h_0(2/4) & h_0(3/4) \\
  h_1(0/4) & h_1(1/4) & h_1(2/4) & h_1(3/4) \\
  h_2(0/4) & h_2(1/4) & h_2(2/4) & h_2(3/4) \\
  h_3(0/4) & h_3(1/4) & h_3(2/4) & h_3(3/4)
\end{pmatrix} = \frac{1}{\sqrt{4}} \begin{pmatrix}
  1 & 1 & 1 & 1 \\
  \sqrt{2} & -\sqrt{2} & 0 & 0 \\
  0 & 0 & \sqrt{2} & -\sqrt{2} \\
  -\sqrt{2} & \sqrt{2} & 0 & 0
\end{pmatrix}.
\]

Similarly, \( A_8 \) is the matrix

\[
A_8 = \frac{1}{\sqrt{8}} \begin{pmatrix}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
  \sqrt{2} & -\sqrt{2} & -\sqrt{2} & -\sqrt{2} & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & \sqrt{2} & \sqrt{2} & -\sqrt{2} & -\sqrt{2} \\
  2 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & -2 & -2 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 2 & -2 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 2 & -2
\end{pmatrix}.
\]

7.10: The average of vector \( w^{(i)} \) is zero, so Equation (7.13) yields

\[
(W \cdot W^T)_{jj} = \sum_{i=1}^{k} w_j^{(i)} w_j^{(i)} = \sum_{i=1}^{k} (w_j^{(i)} - 0)^2 = \sum_{i=1}^{k} (c_j^{(i)} - 0)^2 = k \text{ Variance}(c_j^{(i)}).
\]
M=3; N=2^M; H=[1 1; 1 -1]/sqrt(2);
for m=1:(M-1) % recursion
    H=[H H; H -H]/sqrt(2);
end

A=H';
map=[1 5 7 3 4 8 6 2]; % 1:N
for n=1:N, B(:,n)=A(:,map(n)); end;
A=B;
sc=1/(max(abs(A(:))).^2); % scale factor
for row=1:N
    for col=1:N
        BI=A(:,row)*A(:,col).'; % tensor product
        subplot(N,N,(row-1)*N+col)
        oe=round(BI*sc); % results in -1, +1
        imagesc(oe), colormap([1 1 1; .5 .5 .5; 0 0 0])
        drawnow
    end
end

Figure Ans.29: The 8×8 WHT Basis Images and Matlab Code.
7.11: The Mathematica code of Figure 7.22 produces the eight coefficients 140, −71, 0, −7, 0, −2, 0, and 0. We now quantize this set coarsely by clearing the last two nonzero weights −7 and −2. When the IDCT is applied to the sequence 140, −71, 0, 0, 0, 0, 0, 0, it produces 15, 20, 30, 43, 56, 69, 79, and 84. These are not identical to the original values, but the maximum difference is only 4; an excellent result considering that only two of the eight DCT coefficients are nonzero.

7.12: The eight values in the top row are very similar (the differences between them are either 2 or 3). Each of the other rows is obtained as a right-circular shift of the preceding row.

7.13: It is obvious that such a block can be represented as a linear combination of the patterns in the leftmost column of Figure 7.40. The actual calculation yields the eight weights 4, 0.72, 0, 0.85, 0, 1.27, 0, and 3.62 for the patterns of this column. The other 56 weights are zero or very close to zero.

7.14: The arguments of the cosine functions used by the DCT are of the form \((2x + 1)i\pi/16\), where \(i\) and \(x\) are integers in the range \([0, 7]\). Such an argument can be written in the form \(n\pi/16\), where \(n\) is an integer in the range \([0, 15 \times 7]\). Since the cosine function is periodic, it satisfies \(\cos(32\pi/16) = \cos(0\pi/16)\), \(\cos(33\pi/16) = \cos(\pi/16)\), and so on. As a result, only the 32 values \(\cos(n\pi/16)\) for \(n = 0, 1, 2, \ldots, 31\) are needed. We are indebted to V. Saravanan for pointing out this feature of the DCT.

7.15: Figure 7.54 shows the results (that resemble Figure 7.40) and the Matlab code. Notice that the same code can also be used to calculate and display the DCT basis images.

7.16: First figure out the zigzag path manually, then record it in an array \(\texttt{zz}\) of structures, where each structure contains a pair of coordinates for the path as shown, e.g., in Figure Ans.30.

\[
\begin{array}{cccccccc}
(0,0) & (0,1) & (1,0) & (2,0) & (1,1) & (0,2) & (0,3) & (1,2) \\
(2,1) & (3,0) & (4,0) & (3,1) & (2,2) & (1,3) & (0,4) & (0,5) \\
(1,4) & (2,3) & (3,2) & (4,1) & (5,0) & (6,0) & (5,1) & (4,2) \\
(3,3) & (2,4) & (1,5) & (0,6) & (0,7) & (1,6) & (2,5) & (3,4) \\
(4,3) & (5,2) & (6,1) & (7,0) & (7,1) & (6,2) & (5,3) & (4,4) \\
(3,5) & (2,6) & (1,7) & (2,7) & (3,6) & (4,5) & (5,4) & (6,3) \\
(7,2) & (7,3) & (6,4) & (5,5) & (4,6) & (3,7) & (4,7) & (5,6) \\
(6,5) & (7,4) & (7,5) & (6,6) & (5,7) & (6,7) & (7,6) & (7,7) \\
\end{array}
\]

Figure Ans.30: Coordinates for the Zigzag Path.

If the two components of a structure are \(\texttt{zz.r}\) and \(\texttt{zz.c}\), then the zigzag traversal can be done by a loop of the form:

```plaintext
for (i=0; i<64; i++){
    row:=zz[i].r; col:=zz[i].c
    ...data_unit[row][col]...
}
```
7.17: The third DC difference, 5, is located in row 3 column 5, so it is encoded as 1110\text|101.

7.18: Thirteen consecutive zeros precede this coefficient, so \( Z = 13 \). The coefficient itself is found in Table 7.65 in row 1, column 0, so \( R = 1 \) and \( C = 0 \). Assuming that the Huffman code in position \((R, Z) = (1, 13)\) of Table 7.68 is 1110101, the final code emitted for 1 is 1110101\text|0.

7.19: This is shown by multiplying the largest four \( n \)-bit number, \( 11 \ldots 1 \) by 4, which is easily done by shifting it 2 positions to the left. The result is the \( n + 2 \)-bit number \( 11 \ldots 100 \).

7.20: They make for a more natural progressive growth of the image. They make it easier for a person watching the image develop on the screen to decide if and when to stop the decoding process and accept or discard the image.

7.21: The only specification that depends on the particular bits assigned to the two colors is Equation (7.26). All the other parts of JBIG are independent of the bit assignment.

7.22: For the 16-bit template of Figure 7.101a the relative coordinates are

\[
A_1 = (3, -1), \quad A_2 = (-3, -1), \quad A_3 = (2, -2), \quad A_4 = (-2, -2).
\]

For the 13-bit template of Figure 7.101b the relative coordinates of \( A_1 \) are (3, -1). For the 10-bit templates of Figure 7.101c,d the relative coordinates of \( A_1 \) are (2, -1).

7.23: Transposing S and T produces better compression in cases where the text runs vertically.

7.24: It may simply be too long. When compressing text, each symbol is normally 1-byte long (two bytes in Unicode). However, images with 24-bit pixels are very common, and a 16-pixel block in such an image is 48-bytes long.

7.25: If the encoder uses a \((2, 1, k)\) general unary code, then the value of \( k \) should also be included in the header.

7.26: Going back to step 1 we have the same points participate in the partition for each codebook entry (this happens because our points are concentrated in four distinct regions, but in general a partition \( P^{(k)}_i \) may consist of different image blocks in each iteration \( k \)). The distortions calculated in step 2 are summarized in Table Ans.32. The average distortion \( D^{(1)}_i \) is

\[
D^{(1)} = \frac{(277 + 277 + 277 + 277 + 50 + 50 + 200 + 117 + 37 + 117 + 162 + 117)}{12} = 163.17,
\]

much smaller than the original 603.33. If step 3 indicates no convergence, more iterations should follow (Exercise 7.27), reducing the average distortion and improving the values of the four codebook entries.
I: \[(46 - 32)^2 + (41 - 32)^2 = 277, \quad (46 - 60)^2 + (41 - 32)^2 = 277, \quad (46 - 32)^2 + (41 - 50)^2 = 277, \quad (46 - 60)^2 + (41 - 50)^2 = 277, \]

II: \[(65 - 60)^2 + (145 - 150)^2 = 50, \quad (65 - 70)^2 + (145 - 140)^2 = 50, \]

III: \[(210 - 200)^2 + (200 - 210)^2 = 200, \]

IV: \[(206 - 200)^2 + (41 - 32)^2 = 117, \quad (206 - 200)^2 + (41 - 40)^2 = 37, \quad (206 - 200)^2 + (41 - 50)^2 = 117, \quad (206 - 215)^2 + (41 - 50)^2 = 162, \quad (206 - 215)^2 + (41 - 35)^2 = 117. \]

table Ans.32: Twelve Distortions for $k = 1$. 

Figure Ans.31: Twelve Points and Four Codebook Entries $C_i^{(1)}$. 

Table Ans.32: Twelve Distortions for $k = 1$. 

7.27: Each new codebook entry $C_i^{(k)}$ is calculated, in step 4 of iteration $k$, as the average of the block images comprising partition $P_i^{(k-1)}$. In our example the image blocks (points) are concentrated in four separate regions, so the partitions calculated for iteration $k = 1$ are the same as those for $k = 0$. Another iteration, for $k = 2$, will therefore compute the same partitions in its step 1 yielding, in step 3, an average distortion $D^{(2)}$ that equals $D^{(1)}$. Step 3 will therefore indicate convergence.

7.28: Monitor the compression ratio and delete the dictionary and start afresh each time compression performance drops below a certain threshold.

7.29: Step 4: Point $(2, 0)$ is popped out of the GPP. The pixel value at this position is 7. The best match for this point is with the dictionary entry containing 7. The encoder outputs the pointer 7. The match does not have any concave corners, so we push the point on the right of the matched block, $(2, 1)$, and the point below it, $(3, 0)$, into the GPP. The GPP now contains points $(2, 1), (3, 0), (0, 2), \text{ and } (1, 1)$. The dictionary is updated by appending to it (at location 18) the block $47$.

Step 5: Point $(1, 1)$ is popped out of the GPP. The pixel value at this position is 5. The best match for this point is with the dictionary entry containing 5. The encoder outputs the pointer 5. The match does not have any concave corners, so we push the point to the right of the matched block, $(1, 2)$, and the point below it, $(2, 1)$, into the GPP. The GPP contains points $(1, 2), (2, 1), (3, 0), \text{ and } (0, 2)$. The dictionary is updated by appending to it (at locations 19, 20) the two blocks $25$ and $45$.

7.30: The mean and standard deviation are $\bar{p} = 115$ and $\sigma = 77.93$, respectively. The counts become $n^+ = n^- = 8$, and Equations (7.34) are solved to yield $p^+ = 193$ and $p^- = 37$. The original block is compressed to the 16 bits

$$
\begin{pmatrix}
1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0
\end{pmatrix},
$$

and the two 8-bit values 37 and 193.

7.31: Table Ans.33 summarizes the results. Notice how a 1-pixel with a context of 00 is assigned high probability after being seen 3 times.

<table>
<thead>
<tr>
<th>#</th>
<th>Pixel</th>
<th>Context</th>
<th>Counts</th>
<th>Probability</th>
<th>New counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>10=2</td>
<td>1,1</td>
<td>1/2</td>
<td>2,1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>00=0</td>
<td>1,3</td>
<td>3/4</td>
<td>1,4</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>11=3</td>
<td>1,1</td>
<td>1/2</td>
<td>2,1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>10=2</td>
<td>2,1</td>
<td>1/3</td>
<td>2,2</td>
</tr>
</tbody>
</table>

Table Ans.33: Counts and Probabilities for Next Four Pixels.
7.32: Such a thing is possible for the encoder but not for the decoder. A compression method using “future” pixels in the context is useless because its output would be impossible to decompress.

7.33: The model used by FELICS to predict the current pixel is a second-order Markov model. In such a model the value of the current data item depends on just two of its past neighbors, not necessarily the two immediate ones.

7.34: The two previously seen neighbors of P=8 are A=1 and B=11. P is thus in the central region, where all codes start with a zero, and L=1, H=11. The computations are straightforward:

\[ k = \lfloor \log_2(11 - 1 + 1) \rfloor = 3, \quad a = 2^{3+1} - 11 = 5, \quad b = 2(11 - 2^3) = 6. \]

Table Ans.34 lists the five 3-bit codes and six 4-bit codes for the central region. The code for 8 is thus 0|111.

The two previously seen neighbors of P=7 are A=2 and B=5. P is thus in the right outer region, where all codes start with 11, and L=2, H=7. We are looking for the code of 7 - 5 = 2. Choosing \( m = 1 \) yields, from Table 7.135, the code 11|01.

The two previously seen neighbors of P=0 are A=3 and B=5. P is thus in the left outer region, where all codes start with 10, and L=3, H=5. We are looking for the code of 3 - 0 = 3. Choosing \( m = 1 \) yields, from Table 7.135, the code 10|100.

<table>
<thead>
<tr>
<th>Pixel</th>
<th>Region code</th>
<th>Pixel code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>011</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0001</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0101</td>
</tr>
</tbody>
</table>

Table Ans.34: The Codes for a Central Region.

7.35: Because the decoder has to resolve ties in the same way as the encoder.

7.36: The weights have to add up to 1 because this results in a weighted sum whose value is in the same range as the values of the pixels. If pixel values are, for example, in the range \([0, 15]\) and the weights add up to 2, a prediction may result in values of up to 30.
7.37: Each of the three weights 0.0039, −0.0351, and 0.3164 is used twice. The sum of the weights is therefore 0.5704, and the result of dividing each weight by this sum is 0.0068, −0.0615, and 0.5547. It is easy to verify that the sum of the renormalized weights 2(0.0068 − 0.0615 + 0.5547) equals 1.

7.38: An archive of static images is an example where this approach is practical. NASA has a large archive of images taken by various satellites. They should be kept highly compressed, but they never change, so each image has to be compressed only once. A slow encoder is therefore acceptable but a fast decoder is certainly handy. Another example is an art collection. Many museums have already scanned and digitized their collections of paintings, and those are also static.

7.39: Such a polynomial depends on three coefficients b, c, and d that can be considered three-dimensional points, and any three points are on the same plane.

7.40: This is straightforward

\[
P(2/3) = (0, -9)(2/3)^3 + (-4.5, 13.5)(2/3)^2 + (4.5, -3.5)(2/3) = (0, -8/3) + (-2, 6) + (3, -7/3) = (1, 1) = P_3.
\]

7.41: We use the relations \(\sin 30^\circ = \cos 60^\circ = 0.5\) and the approximation \(\cos 30^\circ = \sin 60^\circ \approx 0.866\). The four points are \(P_1 = (1, 0)\), \(P_2 = (\cos 30^\circ, \sin 30^\circ) = (0.866, 0.5)\), \(P_3 = (0.5, 0.866)\), and \(P_4 = (0, 1)\). The relation \(A = N \cdot P\) becomes

\[
\begin{pmatrix}
a \\
b \\
c \\
d
\end{pmatrix} = A = N \cdot P = 
\begin{pmatrix}
-4.5 & 13.5 & -13.5 & 4.5 \\
9.0 & -22.5 & 18 & -4.5 \\
-5.5 & 9.0 & -4.5 & 1.0 \\
1.0 & 0 & 0 & 0
\end{pmatrix} 
\begin{pmatrix}
1, 0 \\
0.866, 0.5 \\
0.5, 0.866 \\
0, 1
\end{pmatrix}
\]

and the solutions are

\[
\begin{aligned}
a &= -4.5(1, 0) + 13.5(0.866, 0.5) - 13.5(0.5, 0.866) + 4.5(0, 1) = (0.441, -0.441), \\
b &= 19(1, 0) - 22.5(0.866, 0.5) + 18(0.5, 0.866) - 4.5(0, 1) = (-1.485, -0.162), \\
c &= -5.5(1, 0) + 9(0.866, 0.5) - 4.5(0.5, 0.866) + 1(0, 1) = (0.044, 1.603), \\
d &= 1(1, 0) - 0(0.866, 0.5) + 0(0.5, 0.866) - 0(0, 1) = (1, 0).
\end{aligned}
\]

Thus, the PC is \(P(t) = (0.441, -0.441)t^3 + (-1.485, -0.162)t^2 + (0.044, 1.603)t + (1, 0)\). The midpoint is \(P(0.5) \approx (.7058, .7058)\), only 0.2% away from the midpoint of the arc, which is at \((\cos 45^\circ \sin 45^\circ) \approx (.7071, .7071)\).

7.42: The new equations are easy enough to set up. Using Mathematica, they are also easy to solve. The following code

\[
\text{Solve[\{d==p1,}
\]
\(a^3 + b^3 + c^3 + d^3 = p_2,\)
\(a^3 + b^3 + c^3 + d^3 = p_3,\)
\(a + b + c + d = p_4,\) \(\{a, b, c, d\};\)
\(\text{ExpandAll}[\text{Simplify}[\%]]\)

(where \(a\) and \(b\) stand for \(\alpha\) and \(\beta\), respectively) produces the (messy) solutions

\[a = -\frac{P_1}{\alpha\beta} + \frac{P_2}{-\alpha^2 + \alpha^3 + \alpha^2 \beta - \alpha^2 \beta^2 + \alpha \beta^3 + 1 - \alpha - \beta + \alpha \beta},\]
\[b = P_1 \left(-\alpha + \alpha^3 + \beta - \alpha^3 \beta - \beta^3 + \alpha \beta^3\right) / \gamma + P_2 \left(-\beta + \beta^3\right) / \gamma + P_3 (\alpha - \alpha^3) / \gamma + P_4 (\alpha^3 \beta - \alpha \beta^3) / \gamma,\]
\[c = -P_1 \left(1 + \frac{1}{\alpha} + \frac{1}{\beta}\right) + \frac{\beta P_2}{-\alpha^2 + \alpha^3 + \alpha \beta^3 + \alpha^2 \beta^2 + \alpha \beta^3 + 1 - \alpha - \beta + \alpha \beta},\]
\[d = P_1,\]

where \(\gamma = (-1 + \alpha)\alpha(-1 + \beta)\beta(-\alpha + \beta)\).

From here, the basis matrix immediately follows

\[
\begin{pmatrix}
\frac{-1}{\alpha\beta} & \frac{1}{-\alpha^2 + \alpha^3 \beta - \alpha \beta^3 + \alpha^2 \beta^2 + \alpha \beta^3} & \frac{1}{\alpha \beta - \beta^2 - \alpha \beta^3 + \beta^3} & \frac{1}{1 - \alpha - \beta + \alpha \beta} \\
\frac{-\alpha + \alpha^3 + \beta - \alpha^3 \beta - \beta^3 + \alpha \beta^3}{\gamma} & \frac{-\beta + \beta^3}{\gamma} & \frac{\alpha - \alpha^3}{\gamma} & \frac{\alpha^3 \beta - \alpha \beta^3}{\gamma} \\
-\left(1 + \frac{1}{\alpha} + \frac{1}{\beta}\right) & \frac{\beta}{-\alpha^2 + \alpha^3 \beta - \alpha \beta^3 + \alpha^2 \beta^2 + \alpha \beta^3} & \frac{\alpha}{\alpha \beta - \beta^2 - \alpha \beta^3 + \beta^3} & \frac{\alpha^3 \beta - \alpha \beta^3}{\gamma} \\
1 & 0 & 0 & \frac{1}{1 - \alpha - \beta + \alpha \beta}
\end{pmatrix}.
\]

A direct check, again using Mathematica, for \(\alpha = 1/3\) and \(\beta = 2/3\), reduces this matrix to matrix \(N\) of Equation (7.45).

**7.43:** The missing points will have to be estimated by interpolation or extrapolation from the known points before our method can be applied. Obviously, the fewer points are known, the worse the final interpolation. Note that 16 points are necessary, because a bicubic polynomial has 16 coefficients.

**7.44:** Figure Ans.35a shows a diamond-shaped grid of 16 equally-spaced points. The eight points with negative weights are shown in black. Figure Ans.35b shows a cut (labeled xx) through four points in this surface. The cut is a curve that passes through pour data points. It is easy to see that when the two exterior (black) points are raised, the center of the curve (and, as a result, the center of the surface) gets lowered. It is now clear that points with negative weights push the center of the surface in a direction opposite that of the points.

Figure Ans.35c is a more detailed example that also shows why the four corner points should have positive weights. It shows a simple symmetric surface patch that
Clear[Nh,p,pnts,U,W];
p00={0,0,0}; p10={1,0,1}; p20={2,0,1}; p30={3,0,0};
p01={0,1,1}; p11={1,1,2}; p21={2,1,2}; p31={3,1,1};
p02={0,2,1}; p12={1,2,2}; p22={2,2,2}; p32={3,2,1};
p03={0,3,0}; p13={1,3,1}; p23={2,3,1}; p33={3,3,0};
Nh={{-4.5,13.5,-13.5,4.5},{9,-22.5,18,-4.5},
{-5.5,9,-4.5,1},{1,0,0,0}};
pnts={{p33,p32,p31,p30},{p23,p22,p21,p20},
{p13,p12,p11,p10},{p03,p02,p01,p00}};
U[u_]:={u^3,u^2,u,1}; W[w_]:={w^3,w^2,w,1};
(* prt [i] extracts component i from the 3rd dimen of P *)
prt[i_]:=pnts[[Range[1,4],Range[1,4],i]];
p[u_,w_]:={U[u].Nh.prt[1].Transpose[Nh].W[w],
U[u].Nh.prt[2].Transpose[Nh].W[w],
U[u].Nh.prt[3].Transpose[Nh].W[w];
g1=ParametricPlot3D[p[u,w], 
{u,0,1},{w,0,1},Compiled->False, DisplayFunction->Identity];
g2=Graphics3D[{AbsolutePointSize[2],
Table[Point[pnts[[i,j]]],{i,1,4},{j,1,4}]}];
Show[g1,g2, ViewPoint->{-2.876, -1.365, 1.718}]
interpolates the 16 points

\[ P_{00} = (0, 0, 0), \quad P_{10} = (1, 0, 1), \quad P_{20} = (2, 0, 1), \quad P_{30} = (3, 0, 0), \]
\[ P_{01} = (0, 1, 1), \quad P_{11} = (1, 1, 2), \quad P_{21} = (2, 1, 2), \quad P_{31} = (3, 1, 1), \]
\[ P_{02} = (0, 2, 1), \quad P_{12} = (1, 2, 2), \quad P_{22} = (2, 2, 2), \quad P_{32} = (3, 2, 1), \]
\[ P_{03} = (0, 3, 0), \quad P_{13} = (1, 3, 1), \quad P_{23} = (2, 3, 1), \quad P_{33} = (3, 3, 0). \]

We first raise the eight boundary points from \( z = 1 \) to \( z = 1.5 \). Figure Ans.35d shows how the center point \( P(0.5, 0.5) \) gets lowered from \((1.5, 1.5, 2.25)\) to \((1.5, 1.5, 2.10938)\). We next return those points to their original positions and instead raise the four corner points from \( z = 0 \) to \( z = 1 \). Figure Ans.35e shows how this raises the center point from \((1.5, 1.5, 2.25)\) to \((1.5, 1.5, 2.26563)\).

### 7.45: The decoder knows this pixel since it knows the value of average \( \mu[i - 1, j] = 0.5(I[2i - 2, 2j] + I[2i - 1, 2j + 1]) \) and since it has already decoded pixel \( I[2i - 2, 2j] \)

### 7.46: The decoder knows how to do this because when the decoder inputs the 5, it knows that the difference between \( p \) (the pixel being decoded) and the reference pixel starts at position 6 (counting from the left). Since bit 6 of the reference pixel is 0, that of \( p \) must be 1.

### 7.47: Yes, but compression would suffer. One way to apply this method is to separate each byte into two 4-bit pixels and encode each pixel separately. This approach is bad since the prefix and suffix of a 4-bit pixel may often consist of more than four bits. Another approach is to ignore the fact that a byte contains two pixels, and use the method as originally described. This may still compress the image, but is not very efficient, as the following example illustrates.

Example: The two bytes 1100|1101 and 1110|1111 represent four pixels, each differing from its immediate neighbor by its least-significant bit. The four pixels therefore have similar colors (or grayscale). Comparing consecutive pixels results in prefixes of 3 or 2, but comparing the two bytes produces the prefix 2.

### 7.48: The weights add up to 1 because this produces a value \( X \) in the same range as \( A, B, \) and \( C \). If the weights were, for instance, 1, 100, and 1, \( X \) would have much bigger values than any of the three pixels.

### 7.49: The four vectors are

\[
\mathbf{a} = (90, 95, 100, 80, 90, 85),
\]
\[
\mathbf{b}^{(1)} = (100, 90, 95, 102, 80, 90),
\]
\[
\mathbf{b}^{(2)} = (101, 128, 108, 100, 90, 95),
\]
\[
\mathbf{b}^{(3)} = (128, 108, 110, 90, 95, 100),
\]

and the code of Figure Ans.36 produces the solutions \( w_1 = 0.1051, \ w_2 = 0.3974, \) and \( w_3 = 0.3690. \) Their total is 0.8715, compared with the original solutions, which added
up to 0.9061. The point is that the numbers involved in the equations (the elements of the four vectors) are not independent (for example, pixel 80 appears in $a$ and in $b^{(1)}$) except for the last element (85 or 91) of $a$ and the first element 101 of $b^{(2)}$, which are independent. Changing these two elements affects the solutions, which is why the solutions do not always add up to unity. However, compressing nine pixels produces solutions whose total is closer to one than in the case of six pixels. Compressing an entire image, with many thousands of pixels, produces solutions whose sum is very close to 1.

$$a=\{90.,95,100,80,90,85\};$$
$$b1=\{100,90,95,100,80,90\};$$
$$b2=\{100,128,108,100,90,95\};$$
$$b3=\{128,108,110,90,95,100\};$$
Solve[{b1.(a-w1 b1-w2 b2-w3 b3)==0,
b2.(a-w1 b1-w2 b2-w3 b3)==0,
b3.(a-w1 b1-w2 b2-w3 b3)==0},{w1,w2,w3}]

Figure Ans.36: Solving for Three Weights.

7.50: Figure Ans.37a,b,c shows the results, with all $H_i$ values shown in small type. Most $H_i$ values are zero because the pixels of the original image are so highly correlated. The $H_i$ values along the edges are very different because of the simple edge rule used. The result is that the $H_i$ values are highly decorrelated and have low entropy. Thus, they are candidates for entropy coding.

7.51: There are 16 values. The value 0 appears nine times, and each of the other seven values appears once. The entropy is therefore

$$-\sum p_i \log_2 p_i = -\frac{9}{16} \log_2 \left(\frac{9}{16}\right) - \frac{7}{16} \log_2 \left(\frac{1}{16}\right) \approx 2.2169.$$
Not very small, because seven of the 16 values have the same probability. In practice, values of an $H_i$ difference band tend to be small, are both positive and negative, and are concentrated around zero, so their entropy is small.

**7.52:** Because the decoder needs to know how the encoder estimated $X$ for each $H_i$ difference value. If the encoder uses one of three methods for prediction, it has to precede each difference value in the compressed stream with a code that tells the decoder which method was used. Such a code can have variable size (for example, 0, 10, 11) but even adding just one or two bits to each prediction reduces compression performance significantly, because each $H_i$ value needs to be predicted, and the number of these values is close to the size of the image.

**7.53:** The binary tree is shown in Figure Ans.38. From this tree, it is easy to see that the progressive image file is 3 6|5 7|7 7 10 5.

![Figure Ans.38: A Binary Tree for an 8-Pixel Image.](image1)

**7.54:** They are shown in Figure Ans.39.

![Figure Ans.39: The 15 6-Tuples With Two White Pixels.](image2)

**7.55:** No. An image with little or no correlation between the pixels will not compress with quadrisection, even though the size of the last matrix is always small. Even without knowing the details of quadrisection we can confidently state that such an image will produce a sequence of matrices $M_j$ with few or no identical rows. In the extreme case, where the rows of any $M_j$ are all distinct, each $M_j$ will have four times the number of rows of its predecessor. This will create indicator vectors $I_j$ that get longer and longer, thereby increasing the size of the compressed stream and reducing the overall compression performance.
7.56: Matrix $M_5$ is just the concatenation of the 12 distinct rows of $M_4$

$$M_5^T = (0000|0001|1111|0011|1010|1101|1000|0111|1110|0101|1011|0010).$$

7.57: $M_4$ has four columns, so it can have at most 16 distinct rows, implying that $M_5$ can have at most $4 \times 16 = 64$ elements.

7.58: The decoder has to read the entire compressed stream, save it in memory, and start the decoding with $L_5$. Grouping the eight elements of $L_5$ yields the four distinct elements 01, 11, 00, and 10 of $L_4$, so $I_4$ can now be used to reconstruct $L_4$. The four zeros of $I_4$ correspond to the four distinct elements of $L_4$, and the remaining 10 elements of $L_4$ can be constructed from them. Once $L_4$ has been constructed, its eight elements are grouped to form the four distinct elements of $L_3$. These elements are 0111, 0010, 0110, 1111, 0101, and 1010, and they correspond to the seven zeros of $I_3$. Once $L_3$ has been constructed, its eight elements are grouped to form the four distinct elements of $L_2$. Those four elements are the entire $L_2$ since $I_2$ is all zero. Reconstructing $L_1$ and $L_0$ is now trivial.

7.59: The two halves of $L_0$ are distinct, so $L_1$ consists of the two elements

$$L_1 = (0101010101010101, 1010101010101010),$$

and the first indicator vector is $I_1 = (0, 0)$. The two elements of $L_1$ are distinct, so $L_2$ has the four elements

$$L_2 = (01010101, 01010101, 10101010, 10101010),$$

and the second indicator vector is $I_2 = (0, 1, 0, 2)$. Two elements of $L_2$ are distinct, so $L_3$ has the four elements $L_3 = (0101, 0101, 1010, 1010)$, and the third indicator vector is $I_3 = (0, 1, 0, 2)$. Again two elements of $L_3$ are distinct, so $L_4$ has the four elements $L_4 = (01, 01, 10, 10)$, and the fourth indicator vector is $I_4 = (0, 1, 0, 2)$. Only two elements of $L_4$ are distinct, so $L_5$ has the four elements $L_5 = (0, 1, 1, 0)$.

The output thus consists of $k = 5$, the value 2 (indicating that $I_2$ is the first nonzero vector) $I_2$, $I_3$, and $I_4$ (encoded), followed by $L_5 = (0, 1, 1, 0)$.

7.60: Using a Hilbert curve produces the 21 runs 5, 1, 2, 1, 2, 7, 3, 1, 2, 1, 5, 1, 2, 2, 11, 7, 2, 1, 1, 1, 6. RLE produces the 27 runs 0, 1, 7, col, 2, 1, 5, col, 5, 1, 2, col, 0, 3, 2, 3, col, 0, 3, 2, 3, col, 0, 3, 2, 3, col, 4, 1, 3, col, 3, 1, 4, col.

7.61: A straight line segment from $a$ to $b$ is an example of a one-dimensional curve that passes through every point in the interval $a, b$.

7.62: The key is to realize that $P_0$ is a single point, and $P_1$ is constructed by connecting nine copies of $P_0$ with straight segments. Similarly, $P_2$ consists of nine copies of $P_1$, in different orientations, connected by segments (the dashed segments in Figure Ans.40).
7.63: Written in binary, the coordinates are (1101, 0110). We iterate four times, each time taking 1 bit from the x coordinate and 1 bit from the y coordinate to form an (x, y) pair. The pairs are 10, 11, 01, 10. The first one yields [from Table 7.185(1)] 01. The second pair yields [also from Table 7.185(1)] 10. The third pair [from Table 7.185(1)] 11, and the last pair [from Table 7.185(4)] 01. Thus, the result is 01|10|11|01 = 109.

7.64: Table Ans.41 shows that this traversal is based on the sequence 2114.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table Ans.41: The Four Orientations of $H_2$.

7.65: This is straightforward

$$(00, 01, 11, 10) \rightarrow (000, 001, 011, 010)(100, 101, 111, 110)$$
$$\quad \rightarrow (000, 001, 011, 010)(110, 111, 101, 100)$$
$$\quad \rightarrow (000, 001, 011, 010, 110, 111, 101, 100).$$

7.66: The gray area of Figure 7.186c is identified by the string 2011.

7.67: This particular numbering makes it easy to convert between the number of a subsquare and its image coordinates. (We assume that the origin is located at the bottom-left corner of the image and that image coordinates vary from 0 to 1.) As an example, translating the digits of the number 1032 to binary results in (01)(00)(11)(10). The first bits of these groups constitute the x coordinate of the subsquare, and the second bits constitute the y coordinate. Thus, the image coordinates of subsquare 1032 are $x = .0011_2 = 3/16$ and $y = .1010_2 = 5/8$, as can be directly verified from Figure 7.186c.
7.68: This is shown in Figure Ans.42.

7.69: This image is described by the function

\[
f(x, y) = \begin{cases} 
  x + y, & \text{if } x + y \leq 1, \\
  0, & \text{if } x + y > 1.
\end{cases}
\]

7.70: The graph has five states, so each transition matrix is of size $5 \times 5$. Direct computation from the graph yields

\[
W_0 = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0.5 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & -0.5 & 0 & 0 & 1.5 \\
0 & -0.25 & 0 & 0 & 1
\end{pmatrix},
W_3 = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 1.5 \\
0 & 1 & 0 & 0 & 1
\end{pmatrix},
\]

\[
W_1 = W_2 = \begin{pmatrix}
0 & 0 & 1 & 0 & 0 \\
0 & 0.25 & 0 & 0 & 0.5 \\
0 & 0 & 0 & 1.5 & 0 \\
0 & 0 & -0.5 & 1.5 & 0 \\
0 & -0.375 & 0 & 0 & 1.25
\end{pmatrix}.
\]

The final distribution is the five-component vector

\[
F = (0.25, 0.5, 0.375, 0.4125, 0.75)^T.
\]

7.71: One way to specify the center is to construct string $033\ldots3$. This yields

\[
\psi_i(03\ldots3) = (W_0 \cdot W_3 \cdots W_3 \cdot F)_i = \begin{pmatrix} 0.5 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0.5 & 0.5 \\ 0 & 1 \end{pmatrix} \cdots \begin{pmatrix} 0.5 & 0.5 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0.5 \\ 1 \end{pmatrix}_i
= \begin{pmatrix} 0.5 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0.5 \\ 1 \end{pmatrix}_i = \begin{pmatrix} 0.5 \\ 1 \end{pmatrix}_i.
\]
dim=256;
for i=1:dim
  for j=1:dim
    m(i,j)=(i+j-2)/(2*dim-2);
  end
end
m

Figure Ans.43: Matlab Code for a Matrix \( m_{i,j} = (i + j)/2 \).

**7.72:** Figure Ans.43 shows Matlab code to compute a matrix such as those of Figure 7.190.

**7.73:** A direct examination of the graph yields the \( \psi_i \) values

\[
\psi_i(0) = (W_0 \cdot F)_i = (0.5, 0.25, 0.75, 0.875, 0.625)^T,
\]

\[
\psi_i(01) = (W_0 \cdot W_1 \cdot F)_i = (0.5, 0.25, 0.75, 0.875, 0.625)^T,
\]

\[
\psi_i(1) = (W_1 \cdot F)_i = (0.375, 0.5, 0.61875, 0.43125, 0.75)^T,
\]

\[
\psi_i(00) = (W_0 \cdot W_0 \cdot F)_i = (0.25, 0.125, 0.625, 0.8125, 0.5625)^T,
\]

\[
\psi_i(03) = (W_0 \cdot W_3 \cdot F)_i = (0.75, 0.375, 0.625, 0.5625, 0.4375)^T,
\]

\[
\psi_i(3) = (W_3 \cdot F)_i = (0, 0.75, 0, 0, 0.625)^T,
\]

and the \( f \) values

\[
f(0) = I \cdot \psi(0) = 0.5, \quad f(01) = I \cdot \psi(01) = 0.5, \quad f(1) = I \cdot \psi(1) = 0.375,
\]

\[
f(00) = I \cdot \psi(00) = 0.25, \quad f(03) = I \cdot \psi(03) = 0.75, \quad f(3) = I \cdot \psi(3) = 0.
\]

**7.74:** Figure Ans.44a,b shows the six states and all 21 edges. We use the notation \( i(q,t)j \) for the edge with quadrant number \( q \) and transformation \( t \) from state \( i \) to state \( j \). This GFA is more complex than pervious ones since the original image is less self-similar.

**7.75:** The transformation can be written \( (x, y) \to (x, -x + y) \), so \( (1, 0) \to (1, -1) \), \( (3, 0) \to (3, -3) \), \( (1, 1) \to (1, 0) \) and \( (3, 1) \to (3, -2) \). Thus, the original rectangle is transformed into a parallelogram.

**7.76:** The explanation is that the two sets of transformations produce the same Sierpiński triangle but at different sizes and orientations.

**7.77:** All three transformations shrink an image to half its original size. In addition, \( w_2 \) and \( w_3 \) place two copies of the shrunken image at relative displacements of \( (0, 1/2) \) and \( (1/2, 0) \), as shown in Figure Ans.45. The result is the familiar Sierpiński gasket but in a different orientation.
Figure Ans.44: A GFA for Exercise 7.74.
7.78: There are $32 \times 32 = 1,024$ ranges and $(256 - 15) \times (256 - 15) = 58,081$ domains. Thus, the total number of steps is $1,024 \times 58,081 \times 8 = 475,799,552$, still a large number. PIFS is therefore computationally intensive.

7.79: Suppose that the image has $G$ levels of gray. A good measure of data loss is the difference between the value of an average decompressed pixel and its correct value, expressed in number of gray levels. For large values of $G$ (hundreds of gray levels) an average difference of $\log_2 G$ gray levels (or fewer) is considered satisfactory.

8.1: A written page is such an example. A person can place marks on a page and read them later as text, mathematical expressions, and drawings. This is a two-dimensional representation of the information on the page. The page can later be scanned by, e.g., a fax machine, and its contents transmitted as a one-dimensional stream of bits that constitute a different representation of the same information.

8.2: Figure Ans.46 shows $f(t)$ and three shifted copies of the wavelet, for $a = 1$ and $b = 2, 4, \text{ and } 6$. The inner product $W(a,b)$ is plotted below each copy of the wavelet. It is easy to see how the inner products are affected by the increasing frequency.

The table of Figure Ans.47 lists 15 values of $W(a,b)$, for $a = 1, 2, \text{ and } 3$ and for $b = 2$ through 6. The density plot of the figure, where the bright parts correspond to large values, shows those values graphically. For each value of $a$, the CWT yields values that drop with $b$, reflecting the fact that the frequency of $f(t)$ increases with $t$. The five values of $W(1,b)$ are small and very similar, while the five values of $W(3,b)$ are larger.
Figure Ans.46: An Inner Product for $a = 1$ and $b = 2, 4, 6$.

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b = 2$</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.032512</td>
<td>0.000299</td>
<td>1.10923 $\times 10^{-6}$</td>
<td>2.73032 $\times 10^{-9}$</td>
<td>8.33866 $\times 10^{-11}$</td>
</tr>
<tr>
<td>2</td>
<td>0.510418</td>
<td>0.212575</td>
<td>0.0481292</td>
<td>0.00626348</td>
<td>0.00048097</td>
</tr>
<tr>
<td>3</td>
<td>0.743313</td>
<td>0.629473</td>
<td>0.380634</td>
<td>0.173591</td>
<td>0.064264</td>
</tr>
</tbody>
</table>

Figure Ans.47: Fifteen Values and a Density Plot of $W(a, b)$. 
and differ more. This shows how scaling the wavelet up makes the CWT more sensitive to frequency changes in \( f(t) \).

**8.3:** Figure 8.11c shows these wavelets.

**8.4:** Figure Ans.48a shows a simple, \( 8 \times 8 \) image with one diagonal line above the main diagonal. Figure Ans.48b,c shows the first two steps in its pyramid decomposition. It is obvious that the transform coefficients in the bottom-right subband (HH) indicate a diagonal artifact located above the main diagonal. It is also easy to see that subband LL is a low-resolution version of the original image.

![Figure Ans.48: The Subband Decomposition of a Diagonal Line.](image)

**8.5:** The average can easily be calculated. It turns out to be 131.375, which is exactly \( 1/8 \) of 1051. The reason the top-left transform coefficient is eight times the average is that the Matlab code that did the calculations uses \( \sqrt{2} \) instead of 2 (see function `individ(n)` in Figure 8.22).

**8.6:** Figure Ans.49a–c shows the results of reconstruction from 3277, 1639, and 820 coefficients, respectively. Despite the heavy loss of wavelet coefficients, only a very small loss of image quality is noticeable. The number of wavelet coefficients is, of course, the same as the image resolution \( 128 \times 128 = 16,384 \). Using 820 out of 16,384 coefficients corresponds to discarding 95% of the smallest of the transform coefficients (notice, however, that some of the coefficients were originally zero, so the actual loss may amount to less than 95%).

**8.7:** The Matlab code of Figure Ans.50 calculates \( W \) as the product of the three matrices \( A_1, A_2, \) and \( A_3 \) and computes the \( 8 \times 8 \) matrix of transform coefficients. Notice that the top-left value 131.375 is the average of all the 64 image pixels.

**8.8:** The vector \( x = (\ldots, 1, -1, 1, -1, 1, \ldots) \) of alternating values is transformed by the lowpass filter \( H_0 \) to a vector of all zeros.
Figure Ans.49: Three Lossy Reconstructions of the 128 × 128 Lena Image.
clear
a1=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0; 0 0 0 0 1/2 1/2 0 0; 1/2 -1/2 0 0 0 0 0 0; 0 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
% a1*[255; 224; 192; 159; 127; 95; 63; 32];
a2=[1/2 1/2 0 0 0 0 0 0; 0 0 1/2 1/2 0 0 0 0; 1/2 -1/2 0 0 0 0 0 0; 0 0 0 0 1 0 0 0; 0 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1];
a3=[1/2 1/2 0 0 0 0 0 0; 1/2 -1/2 0 0 0 0 0 0; 0 0 1 0 0 0 0 0; 0 0 0 1 0 0 0 0; 0 0 0 0 1 0 0 0; 0 0 0 0 0 0 1];
w=a3*a2*a1;
dim=8; fid=fopen('8x8','r');
img=fread(fid,[dim,dim])'; fclose(fid);
w*img*w' % Result of the transform

Figure Ans.50: Code and Results for the Calculation of Matrix W and Transform W·I·WT.

8.9: For these filters, rules 1 and 2 imply
\[ h_0^2(0) + h_0^2(1) + h_0^2(2) + h_0^2(3) + h_0^2(4) + h_0^2(5) + h_0^2(6) + h_0^2(7) = 1, \]
\[ h_0(0)h_0(2) + h_0(1)h_0(3) + h_0(2)h_0(4) + h_0(3)h_0(5) + h_0(4)h_0(6) + h_0(5)h_0(7) = 0, \]
\[ h_0(0)h_0(4) + h_0(1)h_0(5) + h_0(2)h_0(6) + h_0(3)h_0(7) = 0, \]
\[ h_0(0)h_0(6) + h_0(1)h_0(7) = 0, \]
and rules 3–5 yield
\[ f_0 = (h_0(7), h_0(6), h_0(5), h_0(4), h_0(3), h_0(2), h_0(1), h_0(0)), \]
\[ h_1 = (-h_0(7), h_0(6), -h_0(5), h_0(4), -h_0(3), h_0(2), -h_0(1), h_0(0)), \]
\[ f_1 = (h_0(0), -h_0(1), h_0(2), -h_0(3), h_0(4), -h_0(5), h_0(6), -h_0(7)). \]
The eight coefficients are listed in Table 8.35 (this is the Daubechies D8 filter).

8.10: Figure Ans.51 lists the Matlab code of the inverse wavelet transform function iwt1(wc,coarse,filter) and a test.
function dat=iwt1(wc,coarse,filter)
% Inverse Discrete Wavelet Transform
    dat=wc(1:2^coarse);
    n=length(wc); j=log2(n);
    for i=coarse:j-1
        dat=ILoPass(dat,filter)+ ...
            IHiPass(wc((2^(i)+1):(2^(i+1))),filter);
    end

function f=ILoPass(dt,filter)
    f=iconv(filter,AltrntZro(dt));

function f=IHiPass(dt,filter)
    f=aconv(mirror(filter),rshift(AltrntZro(dt)));

function sgn=mirror(filt)
% return filter coefficients with alternating signs
    sgn=-((-1).^(1:length(filt))).*filt;

function f=AltrntZro(dt)
% returns a vector of length 2*n with zeros
% placed between consecutive values
    n =length(dt)*2; f =zeros(1,n);
    f(1:2:(n-1))=dt;

Figure Ans.51: Code for the 1D Inverse Discrete Wavelet Transform.

A simple test of iwt1 is

n=16; t=(1:n)./n;
    dat=sin(2*pi*t)
    filt=[0.4830 0.8365 0.2241 -0.1294];
    wc=fwt1(dat,1,filt)
    rec=iwt1(wc,1,filt)

8.11: Figure Ans.52 shows the result of blurring the “lena” image. Parts (a) and (b) show the logarithmic multiresolution tree and the subband structure, respectively. Part (c) shows the results of the quantization. The transform coefficients of subbands 5–7 have been divided by two, and all the coefficients of subbands 8–13 have been cleared. We can say that the blurred image of part (d) has been reconstructed from the coefficients of subbands 1–4 (1/64th of the total number of transform coefficients) and half of the coefficients of subbands 5–7 (half of 3/64, or 3/128). On average, the image has been reconstructed from 5/128 ≈ 0.039 or 3.9% of the transform coefficients. Notice that the Daubechies D8 filter was used in the calculations. Readers are encouraged to use this code and experiment with the performance of other filters.

8.12: They are written in the form a-=b/2; b+=a;.
Figure Ans.52: Blurring as a Result of Coarse Quantization.

Code for Figure Ans.52.

```matlab
clear, colormap(gray);
filename='lena128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
filt=[0.23037,0.71484,0.63088,-0.02798, ... 
-0.18703,0.03084,0.03288,-0.01059];
fwim=fwt2(img,3,filt);
figure(1), imagesc(fwim), axis square
fwim(1:16,17:32)=fwim(1:16,17:32)/2;
fwim(1:16,33:128)=0;
fwim(17:32,1:32)=fwim(17:32,1:32)/2;
fwim(17:32,33:128)=0;
fwim(33:128,:)=0;
figure(2), colormap(gray), imagesc(fwim)
rec=iwt2(fwim,3,filt);
figure(3), colormap(gray), imagesc(rec)
```
8.13: We sum Equation (8.13) over all the values of \( l \) to get

\[
\sum_{l=0}^{2^{i-1}-1} s_{j-l,l} = \sum_{l=0}^{2^{i-1}-1} \left( s_{j,2l} + d_{j-1,l}/2 \right) = \frac{1}{2} \sum_{l=0}^{2^{i-1}-1} \left( s_{j,2l} + s_{j,2l+1} \right) = \frac{1}{2} \sum_{l=0}^{2^{i-1}-1} s_{j,l}. \quad \text{(Ans.1)}
\]

Therefore, the average of set \( s_{j-1} \) equals

\[
\frac{1}{2^{j-1}} \sum_{l=0}^{2^{j-1}-1} s_{j-1,l} = \frac{1}{2^{j-1}} \sum_{l=0}^{2^{j-1}-1} s_{j,l} = \frac{1}{2^{j-1}} \sum_{l=0}^{2^{j-1}-1} s_{j,l}
\]

the average of set \( s_j \).

8.14: The code of Figure Ans.53 produces the expression

\[
0.0117P_1 - 0.0977P_2 + 0.5859P_3 + 0.5859P_4 - 0.0977P_5 + 0.0117P_6.
\]

Clear[p,a,b,c,d,e,f];
p[t_]:=a t^5+b t^4+c t^3+d t^2+e t+f;
sol=ExpandAll[Simplify[%]];
Simplify[p[0.5] /.sol]

Figure Ans.53: Code for a Degree-5 Interpolating Polynomial.

8.15: The Matlab code of Figure Ans.54 does that and produces the transformed integer vector \( y = (111, -1, 84, 0, 120, 25, 84, 3) \). The inverse transform generates vector \( z \) that is identical to the original data \( x \). Notice how the detail coefficients are much smaller than the weighted averages. Notice also that Matlab arrays are indexed from 1, whereas the discussion in the text assumes arrays indexed from 0. This causes the difference in index values in Figure Ans.54.

8.16: For the case \( M_C = 3 \), the first six images \( g_0 \) through \( g_5 \) will have dimensions

\[
(3 \cdot 2^5 + 1 \times 4 \cdot 2^5 + 1) = 97 \times 129, \quad 49 \times 65, \quad 25 \times 33, \quad 13 \times 17, \text{ and } 7 \times 9.
\]

8.17: In the sorting pass of the third iteration the encoder transmits the number \( l = 3 \) (the number of coefficients \( c_{i,j} \) in our example that satisfy \( 2^{12} \leq |c_{i,j}| < 2^{13} \)), followed by the three pairs of coordinates \((3,3), (4,2), \) and \((4,1)\) and by the signs of the three coefficients. In the refinement step it transmits the six bits \( cdefgh \). These are the 13th most significant bits of the coefficients transmitted in all the previous iterations.
clear; 
N=8; k=N/2; 
x=[112,97,85,99,114,120,77,80]; 
\% Forward IWT into y 
for i=0:k-2, 
y(2*i+2)=x(2*i+2)-floor((x(2*i+1)+x(2*i+3))/2); 
end; 
y(N)=x(N)-x(N-1); 
y(1)=x(1)+floor(y(2)/2); 
for i=1:k-1, 
y(2*i+1)=x(2*i+1)+floor((y(2*i)+y(2*i+2))/4); 
end; 
\% Inverse IWT into z 
z(1)=y(1)-floor(y(2)/2); 
for i=1:k-1, 
z(2*i+1)=y(2*i+1)-floor((y(2*i)+y(2*i+2))/4); 
end; 
for i=0:k-2, 
z(2*i+2)=y(2*i+2)+floor((z(2*i+1)+x(2*i+3))/2); 
end; 
z(N)=y(N)+z(N-1); 

Figure Ans.54: Matlab Code for Forward and Inverse IWT.

The information received so far enables the decoder to further improve the 16 approximate coefficients. The first nine become 

\[ c_{2,3} = s1ac0...0, \quad c_{3,4} = s1bd0...0, \quad c_{3,2} = s01e00...0, \]
\[ c_{4,4} = s01f00...0, \quad c_{1,2} = s01g00...0, \quad c_{3,1} = s01h00...0, \]
\[ c_{3,3} = s0010...0, \quad c_{4,2} = s0010...0, \quad c_{4,1} = s0010...0, \]

and the remaining seven are not changed.

8.18: The simple equation \(10 \times 2^{20} \times 8 = (500x) \times (500x) \times 8\) is solved to yield \(x^2 = 40\) square inches. If the card is square, it is approximately 6.32 inches on a side. Such a card has 10 rolled impressions (about 1.5 \times 1.5 each), two plain impressions of the thumbs (about 0.875 \times 1.875 each), and simultaneous impressions of both hands (about 3.125 \times 1.875 each). All the dimensions are in inches.

8.19: The bit of 10 is encoded, as usual, in pass 2. The bit of 1 is encoded in pass 1 since this coefficient is still insignificant but has significant neighbors. This bit is 1, so coefficient 1 becomes significant (a fact that is not used later). Also, this bit is the first 1 of this coefficient, so the sign bit of the coefficient is encoded following this bit. The bits of coefficients 3 and -7 are encoded in pass 2 since these coefficients are significant.

9.1: It is easy to calculate that \(525 \times 4/3 = 700\) pixels.
9.2: The vertical height of the picture on the author’s 27 in. television set is 16 in.,
which translates to a viewing distance of $7.12 \times 16 = 114$ in. or about 9.5 feet. It is easy
to see that individual scan lines are visible at any distance shorter than about 6 feet.

9.3: Three common examples are: (1) Surveillance camera, (2) an old, silent movie
being restored and converted from film to video, and (3) a video presentation taken
underwater.

9.4: The golden ratio $\phi \approx 1.618$ has traditionally been considered the aspect ratio that
is most pleasing to the eye. This suggests that 1.77 is the better aspect ratio.

9.5: Imagine a camera panning from left to right. New objects will enter the field of
view from the right all the time. A block on the right side of the frame may therefore
contain objects that did not exist in the previous frame.

9.6: Since $(4,4)$ is at the center of the “+”, the value of $s$ is halved, to 2. The next step
searches the four blocks labeled 4, centered on $(4,4)$. Assuming that the best match is
at $(6,4)$, the two blocks labeled 5 are searched. Assuming that $(6,4)$ is the best match,
$s$ is halved to 1, and the eight blocks labeled 6 are searched. The diagram shows that
the best match is finally found at location $(7,4)$.

9.7: This figure consists of $18 \times 18$ macroblocks, and each macroblock constitutes six
$8 \times 8$ blocks of samples. The total number of samples is therefore $18 \times 18 \times 6 \times 64 = 124,416$.

9.8: The size category of zero is 0, so code 100 is emitted, followed by zero bits. The
size category of 4 is 3, so code 110 is first emitted, followed by the three least-significant
bits of 4, which are 100.

9.9: The zigzag sequence is

$$118, 2, 0, -2, 0, \ldots, 0, -1, 0, \ldots$$

The run-level pairs are $(0,2)$, $(1,-2)$, and $(13,-1)$, so the final codes are (notice the
sign bits following the run-level codes)

$$01000|0010|010101|001000001|10,$$

(without the vertical bars).

9.10: There are no nonzero coefficients, no run-level codes, just the 2-bit EOB code.
However, in nonintra coding, such a block is encoded in a special way.

10.1: An average book may have 60 characters per line, 45 lines per page, and 400
pages. This comes to $60 \times 45 \times 400 = 1,080,000$ characters, requiring one byte of storage
each.
10.2: The period of a wave is its speed divided by its frequency. For sound we get
\[
\frac{34380 \text{ cm/s}}{22000 \text{ Hz}} = 1.562 \text{ cm}, \quad \frac{34380}{20} = 1719 \text{ cm}.
\]

10.3: The (base-10) logarithm of \(x\) is a number \(y\) such that \(10^y = x\). The number 2 is the logarithm of 100 since \(10^2 = 100\). Similarly, 0.3 is the logarithm of 2 since \(10^{0.3} = 2\). Also, The base-\(b\) logarithm of \(x\) is a number \(y\) such that \(b^y = x\) (for any real \(b > 1\)).

10.4: Each doubling of the sound intensity increases the dB level by 3. Therefore, the difference of 9 dB \((3 + 3 + 3)\) between A and B corresponds to three doublings of the sound intensity. Thus, source B is \(2 \cdot 2 \cdot 2 = 8\) times louder than source A.

10.5: Each 0 would result in silence and each sample of 1, in the same tone. The result would be a nonuniform buzz. Such sounds were common on early personal computers.

10.6: Such an experiment should be repeated with several persons, preferably of different ages. The person should be placed in a sound insulated chamber, and a pure tone of frequency \(f\) should be played. The amplitude of the tone should be gradually increased from zero until the person can just barely hear it. If this happens at a decibel value \(d\), point \((d, f)\) should be plotted. This should be repeated for many frequencies until a graph similar to Figure 10.5a is obtained.

10.7: We first select identical items. If all \(s(t - i)\) equal \(s\), Equation (10.7) yields the same \(s\). Next, we select values on a straight line. Given the four values \(a, a + 2, a + 4,\) and \(a + 6\), Equation (10.7) yields \(a + 8\), the next linear value. Finally, we select values roughly equally-spaced on a circle. The \(y\) coordinates of points on the first quadrant of a circle can be computed by \(y = \sqrt{r^2 - x^2}\). We select the four points with \(x\) coordinates 0, 0.08\(r\), 0.16\(r\), and 0.24\(r\), compute their \(y\) coordinates for \(r = 10\), and substitute them in Equation (10.7). The result is 9.96926, compared to the actual \(y\) coordinate for \(x = 0.32r\) which is \(\sqrt{r^2 - (0.32r)^2} = 9.47418\), a difference of about 5%. The code that did the computations is shown in Figure Ans.55.

```plaintext
(* Points on a circle. Used in exercise to check
  4th-order prediction in FLAC *)
r = 10;
ci[x_] := Sqrt[100 - x^2];
ci[0.32r]
4ci[0] - 6ci[0.08r] + 4ci[0.16r] - ci[0.24r]
```

Figure Ans.55: Code for Checking 4th-Order Prediction.
10.8: Imagine that the sound being compressed contains one second of a pure tone (just one frequency). This second will be digitized to 44,100 consecutive samples per channel. The samples indicate amplitudes, so they don’t have to be the same. However, after filtering, only one subband (although in practice perhaps two subbands) will have nonzero signals. All the other subbands correspond to different frequencies, so they will have signals that are either zero or very close to zero.

10.9: Assuming that a noise level $P_1$ translates to $x$ decibels

$$20 \log \left( \frac{P_1}{P_2} \right) = x \text{ dB SPL},$$

results in the relation

$$20 \log \left( \sqrt{2} \frac{P_1}{P_2} \right) = 20 \left[ \log_{10} \sqrt{2} + \log \left( \frac{P_1}{P_2} \right) \right] = 20(0.1 + x/20) = x + 2.$$

Thus, increasing the sound level by a factor of $\sqrt{2}$ increases the decibel level by 2 dB SPL.

10.10: For a sampling rate of 44,100 samples/sec, the calculations are similar. The decoder has to decode $44,100/384 \approx 114.84$ frames per second. Thus, each frame has to be decoded in approximately 8.7 ms. In order to output 114.84 frames in 64,000 bits, each frame must have $B_f = 557$ bits available to encode it. Thus, the number of slots per frame is $557/32 \approx 17.41$. Thus, the last (18th) slot is not full and has to padded.

10.11: Table 10.58 shows that the scale factor is 111 and the select information is 2. The third rule in Table 10.59 shows that a scfsi of 2 means that only one scale factor was coded, occupying just six bits in the compressed output. The decoder assigns these six bits as the values of all three scale factors.

10.12: Typical layer II parameters are (1) a sampling rate of 48,000 samples/sec, (2) a bitrate of 64,000 bits/sec, and (3) 1,152 quantized signals per frame. The decoder has to decode $48,000/1152 = 41.66$ frames per second. Thus, each frame has to be decoded in 24 ms. In order to output 41.66 frames in 64,000 bits, each frame must have $B_f = 1,536$ bits available to encode it.

10.13: A program to play .mp3 files is an MPEG layer III decoder, not an encoder. Decoding is much simpler since it does not use a psychoacoustic model, nor does it have to anticipate preechoes and maintain the bit reservoir.

11.1: Because the original string $S$ can be reconstructed from $L$ but not from $F$.

11.2: A direct application of Equation (11.1) eight more times produces:


The original string *swiss miss* is indeed reproduced in S from right to left.

11.3: Figure Ans.56 shows the rotations of S and the sorted matrix. The last column, L of Ans.56b happens to be identical to S, so S=L=sssssssssh. Since A=(s,h), a move-to-front compression of L yields C = (1,0,0,0,0,0,0,0,1). Since C contains just the two values 0 and 1, they can serve as their own Huffman codes, so the final result is 1000000001, 1 bit per character!

```
sssssssssh  hssssssss
ssssssshhs  shssssss
sssssshsss  sshssssss
ssssshhsss  sshssssss
ssssshssss  sshssssss
ssssshssss  sshssssss
ssshssssss  sshssssss
ssshssssss  sshssssss
ssshssssss  sshssssss
ssshssssss  sshssssss
shssssssss  shssssss
hsssssssss  ssssssssh
```

(a) (b)

Figure Ans.56: Permutations of “sssssssssh”.

11.4: The encoder starts at T[0], which contains 5. The first element of L is thus the last symbol of permutation 5. This permutation starts at position 5 of S, so its last element is in position 4. The encoder thus has to go through symbols S[T[i-1]] for i = 0, ..., n-1, where the notation i-1 should be interpreted cyclically (i.e., 0-1 should be n-1). As each symbol S[T[i-1]] is found, it is compressed using move-to-front. The value of I is the position where T contains 0. In our example, T[8]=0, so I=8.

11.5: The first element of a triplet is the distance between two dictionary entries, the one best matching the content and the one best matching the context. In this case there is no content match, no distance, so any number could serve as the first element, 0 being the best (smallest) choice.

11.6: because the three lines are sorted in ascending order. The bottom two lines of Table 11.13c are not in sorted order. This is why the zz...z part of string S must be preceded and followed by complementary bits.

11.7: The encoder places S between two entries of the sorted associative list and writes the (encoded) index of the entry above or below S on the compressed stream. The fewer the number of entries, the smaller this index, and the better the compression.
11.8: Context 5 is compared to the three remaining contexts 6, 7, and 8, and it is most similar to context 6 (they share a suffix of “b”). Context 6 is compared to 7 and 8 and, since they don’t share any suffix, context 7, the shorter of the two, is selected. The remaining context 8 is, of course, the last one in the ranking. The final context ranking is

\[1 \rightarrow 3 \rightarrow 4 \rightarrow 0 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8.\]

11.9: Equation (11.3) shows that the third “a” is assigned rank 1 and the “b” and “a” following it are assigned ranks 2 and 3, respectively.

11.10: Table Ans.57 shows the sorted contexts. Equation (Ans.2) shows the context ranking at each step.

\[\begin{array}{cccc}
0, & 0 \rightarrow 2, & 1 \rightarrow 3 \rightarrow 0, \\
u & u & b & l & b & u \\
0 \rightarrow 2 \rightarrow 3 \rightarrow 4, & 2 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 0, \\
u & l & a & b & l & a & d & b & u \\
3 \rightarrow 5 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 0, \\
i & a & l & b & d & u \\
\end{array}\]

\text{(Ans.2)}

The final output is “u 2 b 3 l 4 a 5 d 6 i 6.” Notice that each of the distinct input symbols appears once in this output in raw format.

11.11: All \(n_1\) bits of string \(L_1\) need be written on the output stream. This already shows that there is going to be no compression. String \(L_2\) consists of \(n_1/k\) 1’s, so all of it has to be written on the output stream. String \(L_3\) similarly consists of \(n_1/k^2\) 1’s, and so on. Thus, the size of the output stream is

\[n_1 + \frac{n_1}{k} + \frac{n_1}{k^2} + \frac{n_1}{k^3} + \cdots + \frac{n_1}{k^m} = n_1 \frac{k^{m+1} - 1}{k^m(k - 1)},\]
for some value of $m$. The limit of this expression, when $m \to \infty$, is $n_1k/(k-1)$. For $k = 2$ this equals $2n_1$. For larger values of $k$ this limit is always between $n_1$ and $2n_1$.

For the curious reader, here is how the sum above is computed. Given the series

$$S = \sum_{i=0}^{m} \frac{1}{k^i} = 1 + \frac{1}{k} + \frac{1}{k^2} + \cdots + \frac{1}{k^{m-1}} + \frac{1}{k^m},$$

we multiply both sides by $1/k$

$$\frac{S}{k} = \frac{1}{k} + \frac{1}{k^2} + \cdots + \frac{1}{k^{m-1}} + \frac{1}{k^m} = S + \frac{1}{k^{m+1}} - 1,$$

and subtract

$$\frac{S}{k}(k-1) = \frac{k^{m+1} - 1}{k^{m+1}} \to S = \frac{k^{m+1} - 1}{k^m(k-1)}.$$

11.12: The input stream consists of:
1. A run of three zero groups, coded as $10|1$ since 3 is in second position in class 2.
2. The nonzero group 0100, coded as 111100.
3. Another run of three zero groups, again coded as $10|1$.
4. The nonzero group 1000, coded as 01100.
5. A run of four zero groups, coded as $R_4 = 1001$.
6. 0010, coded as 111110.
7. A run of two zero groups, coded as $R_2 = 101$.

The output is thus the 31-bit string 101111100101011000100011110100.

11.13: The input stream consists of:
1. A run of three zero groups, coded as $R_2R_1$ or $101|11$.
2. The nonzero group 0100, coded as 00100.
3. Another run of three zero groups, again coded as $101|11$.
4. The nonzero group 1000, coded as 01000.
5. A run of four zero groups, coded as $R_4 = 1001$.
6. 0010, coded as 00010.
7. A run of two zero groups, coded as $R_2 = 101$.

The output is thus the 32-bit string 10111001001011100010001010101.

11.14: The input stream consists of:
1. A run of three zero groups, coded as $F_3$ or 1001.
2. The nonzero group 0100, coded as 00100.
3. Another run of three zero groups, again coded as 1001.
4. The nonzero group 1000, coded as 01000.
5. A run of four zero groups, coded as $F_3F_1 = 1001|11$.
6. 0010, coded as 00010.
7. A run of two zero groups, coded as $F_2 = 101$.

The output is thus the 32-bit string 100100100100100010011100010101.
11.15: Yes, if they are located in different quadrants or subquadrants. Pixels 123 and 301, for example, are adjacent in Figure 7.172 but have different prefixes.

11.16: No, because all prefixes have the same probability of occurrence. In our example the prefixes are four bits long and all 16 possible prefixes have the same probability because a pixel may be located anywhere in the image. A Huffman code constructed for 16 equally-probable symbols has an average size of four bits per symbol, so nothing would be gained. The same is true for suffixes.

11.17: This is possible, but it places severe limitations on the size of the string. In order to rearrange a one-dimensional string into a four-dimensional cube, the string size should be $2^{4n}$. If the string size happens to be $2^{4n} + 1$, it has to be extended to $2^{4(n+1)}$, which increases its size by a factor of 16. It is possible to rearrange the string into a rectangular box, not just a cube, but then its size will have to be of the form $2^{n_1}2^{n_2}2^{n_3}2^{n_4}$ where the four $n_i$'s are integers.

11.18: The LZW algorithm, which starts with the entire alphabet stored at the beginning of its dictionary, is an example of such a method. However, an adaptive version of LZW can be designed to compress words instead of individual characters.

11.19: A better choice for the coordinates may be relative values (or offsets). Each $(x, y)$ pair may specify the position of a character relative to its predecessor. This results in smaller numbers for the coordinates, and smaller numbers are easier to compress.

11.20: There may be such letters in other, “exotic” alphabets, but a more common example is a rectangular box enclosing text. The four rules that constitute such a box should be considered a mark, but the text characters inside the box should be identified as separate marks.

11.21: This guarantees that the two probabilities will add up to 1.

11.22: Figure Ans.58 shows how state $A$ feeds into the new state $D'$ which, in turn, feeds into states $E$ and $F$. Notice how states $B$ and $C$ haven’t changed. Since the new state $D'$ is identical to $D$, it is possible to feed $A$ into either $D$ or $D'$ (cloning can be done in two different but identical ways). The original counts of state $D$ should now be divided between $D$ and $D'$ in proportion to the counts of the transitions $A \rightarrow D$ and $B, C \rightarrow D$.

11.23: Figure Ans.59 shows the new state 6 after the operation $1, 1 \rightarrow 6$. Its 1-output is identical to that of state 1, and its 0-output is a copy of the 0-output of state 3.

11.24: A precise answer requires many experiments with various data files. A little thinking, though, shows that the larger $k$, the better the initial model that is created when the old one is discarded. Larger values of $k$ thus minimize the loss of compression. However, very large values may produce an initial model that is already large and cannot grow much. The best value for $k$ is therefore one that produces an initial model large enough to provide information about recent correlations in the data, but small enough so it has room to grow before it too has to be discarded.
11.25: The number of marked points can be written \(8(1 + 2 + 3 + 5 + 8 + 13) = 256\) and the numbers in parentheses are the Fibonacci numbers.

11.26: The conditional probability \(P(D_i | D_i)\) is very small. A segment pointing in direction \(D_i\) can be preceded by another segment pointing in the same direction only if the original curve is straight or very close to straight for more than 26 coordinate units (half the width of grid \(S_{13}\)).

11.27: We observe that a point has two coordinates. If each coordinate occupies eight bits, then the use of Fibonacci numbers reduces the 16-bit coordinates to an 8-bit number, a compression ratio of 0.5. The use of Huffman codes can typically reduce this 8-bit number to (on average) a 4-bit code, and the use of the Markov model can perhaps cut this by another bit. The result is an estimated compression ratio of \(3/16 = 0.1875\). If each coordinate is a 16-bit number, then this ratio improves to \(3/32 = 0.09375\).

11.28: The resulting, shorter grammar is shown in Figure Ans.60. It is one rule and one symbol shorter.

<table>
<thead>
<tr>
<th>Input</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S \rightarrow \text{abcdbcabcdbc} )</td>
<td>(S \rightarrow \text{CC} )</td>
</tr>
<tr>
<td>(A \rightarrow \text{bc} )</td>
<td>(A \rightarrow \text{bc} )</td>
</tr>
<tr>
<td>(C \rightarrow \text{aAdA} )</td>
<td>(C \rightarrow \text{aAdA} )</td>
</tr>
</tbody>
</table>

Figure Ans.60: Improving the Grammar of Figure 11.42.
11.29: Because generating rule $C$ has made rule $B$ underused (i.e., used just once).

11.30: Rule $S$ consists of two copies of rule $A$. The first time rule $A$ is encountered, its contents $aBdB$ are sent. This involves sending rule $B$ twice. The first time rule $B$ is sent, its contents $bc$ are sent (and the decoder does not know that the string $bc$ it is receiving is the contents of a rule). The second time rule $B$ is sent, the pair $(1, 2)$ is sent (offset 1, count 2). The decoder identifies the pair and uses it to set up the rule $1 \rightarrow bc$. Sending the first copy of rule $A$ therefore amounts to sending $abcd(1, 2)$. The second copy of rule $A$ is sent as the pair $(0, 4)$ since $A$ starts at offset 0 in $S$ and its length is 4. The decoder identifies this pair and uses it to set up the rule $2 \rightarrow a[1][1]$. The final result is therefore $abcd(1, 2)(0, 4)$.

11.31: In each of these cases, the encoder removes one edge from the boundary and inserts two new edges. There is a net gain of one edge.

11.32: They create triangles $(18, 2, 3)$ and $(18, 3, 4)$, and reduce the boundary to the sequence of vertices

$$(4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18).$$

A.1: It is $1 + \lfloor \log_2 i \rfloor$ as can be seen by simple experimenting.

A.2: The integer 2 is the smallest integer that can serve as the basis for a number system.

A.3: Replacing 10 by 3 we get $x = k \log_2 3 \approx 1.58k$. A trit is therefore worth about 1.58 bits.

A.4: We assume an alphabet with two symbols $a_1$ and $a_2$, with probabilities $P_1$ and $P_2$, respectively. Since $P_1 + P_2 = 1$, the entropy of the alphabet is $-P_1 \log_2 P_1 - (1 - P_1) \log_2 (1 - P_1)$. Table 2.2 shows the entropies for certain values of the probabilities. When $P_1 = P_2$, at least 1 bit is required to encode each symbol, reflecting the fact that the entropy is at its maximum, the redundancy is zero, and the data cannot be compressed. However, when the probabilities are very different, the minimum number of bits required per symbol drops significantly. We may not be able to develop a compression method using 0.08 bits per symbol but we know that when $P_1 = 99\%$, this is the theoretical minimum.

A writer is someone who can make a riddle out of an answer.

Karl Kraus
Bibliography

All URLs have been checked and updated as of early July 2009. Any broken links reported to the authors will be added to the errata list in the book’s Web site.

The main event in the life of the data compression community is the annual data compression conference (DCC, see Joining the Data Compression Community) whose proceedings are published by the IEEE. The editors have traditionally been James Andrew Storer and Martin Cohn. Instead of listing many references that differ only by year, we start this bibliography with a generic reference to the DCC, where “XX” is the last two digits of the conference year.


7z (2006) is [http://www.7-zip.org/sdk.html](http://www.7-zip.org/sdk.html).


BSDiff (2005) is http://www.daemonology.net/bsdiff/bsdiff-4.3.tar.gz


Bibliography


Buyanovsky, George (1994) “Associative Coding” (in Russian), *Monitor*, Moscow, #8, 10–19, August. (Hard copies of the Russian source and English translation are available from the authors of this book. Send requests to the authors’ email address found in the Preface.)

Buyanovsky, George (2002) Private communications (buyanovsky@home.com).


CREW 2000 was [www.crc.ricoh.com/CREW/](http://www.crc.ricoh.com/CREW/) but it no longer exists.


Fibonacci (1999) is file Fibonacci.html in [http://www-groups.dcs.st-and.ac.uk/~history/References/](http://www-groups.dcs.st-and.ac.uk/~history/References/).


H.264PaperIR (2006) is http://www.vcodex.com/h264.html (papers by Iain Richard-
son).


H.264Standards (2006) is ftp://standards.polycom.com/JVT_Site/ (the H.264 stan-
dards repository).


h2g2 (2006) is http://www.bbc.co.uk/dna/h2g2/A406973.


hydrogenaudio (2006) is www.hydrogenaudio.org/forums/.


JBIG (2003) is \url{http://www.jpeg.org/jbig/index.html}.

JBIG2 (2003) is \url{http://www.jpeg.org/jbig/jbigpt2.html}.

JBIG2 (2006) is \url{http://jbig2.com/}.


KGB (2008) is \url{http://kgbarchiver.net/}.


LBE (2007) is http://in.geocities.com/iamthebiggestone/how_lbe_works.htm


LZX (2009) is http://xavprods.free.fr/lzx/.


In computer network engineering, a Request for Comments (RFC) is a memorandum published by the Internet Engineering Task Force (IETF) describing methods, behaviors, research, or innovations applicable to the working of the Internet and Internet-connected systems.

Through the Internet Society, engineers and computer scientists may publish discourse in the form of an RFC, either for peer review or simply to convey new concepts, information, or (occasionally) engineering humor. The IETF adopts some of the proposals published as RFCs as Internet standards.

The acronym RFC also stands for Radio frequency choke, Remote function call, Request for change, and Regenerative fuel cell.

—From http://en.wikipedia.org/


Samet, Hanan (1990b) *The Design and Analysis of Spatial Data Structures*, Reading, MA, Addison-Wesley.


seul.org (2006) is http://f-cpu.seul.org/whygee/phasing-in_codes/PhasingInCodes.nb


Softsound (2003) was http://www.softsound.com/Shorten.html but try also URL http://mi.eng.cam.ac.uk/reports/ajr/TR156/tr156.html.


Bibliography


Yet the guide is fragmentary, incomplete, and in no sense a bibliography. Its emphases vary according to my own indifferences and ignorance as well as according to my own sympathies and knowledge.

J. Frank Dobie, *Guide to Life and Literature of the Southwest* (1943)
Glossary

7-Zip. A file archiver with high compression ratio. The brainchild of Igor Pavlov, this free software for Windows is based on the LZMA algorithm. Both LZMA and 7z were designed to provide high compression, fast decompression, and low memory requirements for decompression. (See also LZMA.)

AAC. A complex and efficient audio compression method. AAC is an extension of and the successor to mp3. Like mp3, AAC is a time/frequency (T/F) codec that employs a psychoacoustic model to determine how the normal threshold of the ear varies in the presence of masking sounds. Once the perturbed threshold is known, the original audio samples are converted to frequency coefficients which are quantized (thereby providing lossy compression) and then Huffman encoded (providing additional, lossless, compression).

AC-3. A perceptual audio coded designed by Dolby Laboratories to support several audio channels.

ACB. A very efficient text compression method by G. Buyanovsky (Section 11.3). It uses a dictionary with unbounded contexts and contents to select the context that best matches the search buffer and the content that best matches the look-ahead buffer.

Adaptive Compression. A compression method that modifies its operations and/or its parameters according to new data read from the input stream. Examples are the adaptive Huffman method of Section 5.3 and the dictionary-based methods of Chapter 6. (See also Semiadaptive Compression, Locally Adaptive Compression.)

Affine Transformations. Two-dimensional or three-dimensional geometric transformations, such as scaling, reflection, rotation, and translation, that preserve parallel lines (Section 7.39.1).

Alphabet. The set of all possible symbols in the input stream. In text compression, the alphabet is normally the set of 128 ASCII codes. In image compression it is the set of values a pixel can take (2, 16, 256, or anything else). (See also Symbol.)
**ALPC.** ALPC (*adaptive linear prediction and classification*), is a lossless image compression algorithm based on a linear predictor whose coefficients are computed for each pixel individually in a way that can be mimicked by the decoder.

**ALS.** MPEG-4 Audio Lossless Coding (ALS) is the latest addition to the family of MPEG-4 audio codecs. ALS can handle integer and floating-point audio samples and is based on a combination of linear prediction (both short-term and long-term), multichannel coding, and efficient encoding of audio residues by means of Rice codes and block codes.

**Antidictionary.** (See DCA.)

**ARC.** A compression/archival/cataloging program written by Robert A. Freed in the mid 1980s (Section 6.24). It offers good compression and the ability to combine several files into an archive. (See also Archive, ARJ.)

**Archive.** A set of one or more files combined into one file (Section 6.24). The individual members of an archive may be compressed. An archive provides a convenient way of transferring or storing groups of related files. (See also ARC, ARJ.)

**Arithmetic Coding.** A statistical compression method (Section 5.9) that assigns one (normally long) code to the entire input stream, instead of assigning codes to the individual symbols. The method reads the input stream symbol by symbol and appends more bits to the code each time a symbol is input and processed. Arithmetic coding is slow, but it compresses at or close to the entropy, even when the symbol probabilities are skewed. (See also Model of Compression, Statistical Methods, QM Coder.)

**ARJ.** A free compression/archiving utility for MS/DOS (Section 6.24), written by Robert K. Jung to compete with ARC and the various PK utilities. (See also Archive, ARC.)

**ASCII Code.** The standard character code on all modern computers (although Unicode is becoming a competitor). ASCII stands for American Standard Code for Information Interchange. It is a (1+7)-bit code, with one parity bit and seven data bits per symbol. As a result, 128 symbols can be coded. They include the uppercase and lowercase letters, the ten digits, some punctuation marks, and control characters. (See also Unicode.)

**Bark.** Unit of critical band rate. Named after Heinrich Georg Barkhausen and used in audio applications. The Bark scale is a nonlinear mapping of the frequency scale over the audio range, a mapping that matches the frequency selectivity of the human ear.

**BASC.** A compromise between the standard binary ($\beta$) code and the Elias gamma codes. (See also RBUC.)

**Bayesian Statistics.** (See Conditional Probability.)

**Bi-level Image.** An image whose pixels have two different colors. The colors are normally referred to as black and white, “foreground” and “background,” or 1 and 0. (See also Bitplane.)

**Benchmarks.** (See Compression Benchmarks.)
**BinHex.** A file format for reliable file transfers, designed by Yves Lempereur for use on the Macintosh computer (Section 1.4.3).

**Bintrees.** A method, somewhat similar to quadtrees, for partitioning an image into nonoverlapping parts. The image is (horizontally) divided into two halves, each half is divided (vertically) into smaller halves, and the process continues recursively, alternating between horizontal and vertical splits. The result is a binary tree where any uniform part of the image becomes a leaf. (See also Prefix Compression, Quadtrees.)

**Bitplane.** Each pixel in a digital image is represented by several bits. The set of all the $k$th bits of all the pixels in the image is the $k$th bitplane of the image. A bi-level image, for example, consists of one bitplane. (See also Bi-level Image.)

**Bitrate.** In general, the term “bitrate” refers to both bpb and bpc. However, in audio compression, this term is used to indicate the rate at which the compressed stream is read by the decoder. This rate depends on where the stream comes from (such as disk, communications channel, memory). If the bitrate of an MPEG audio file is, e.g., 128 Kbps, then the encoder will convert each second of audio into 128 K bits of compressed data, and the decoder will convert each group of 128 K bits of compressed data into one second of sound. Lower bitrates mean smaller file sizes. However, as the bitrate decreases, the encoder must compress more audio data into fewer bits, eventually resulting in a noticeable loss of audio quality. For CD-quality audio, experience indicates that the best bitrates are in the range of 112 Kbps to 160 Kbps. (See also Bits/Char.)

**Bits/Char.** Bits per character (bpc). A measure of the performance in text compression. Also a measure of entropy. (See also Bitrate, Entropy.)

**Bits/Symbol.** Bits per symbol. A general measure of compression performance.

**Block Coding.** A general term for image compression methods that work by breaking the image into small blocks of pixels, and encoding each block separately. JPEG (Section 7.10) is a good example, because it processes blocks of $8 \times 8$ pixels.

**Block Decomposition.** A method for lossless compression of discrete-tone images. The method works by searching for, and locating, identical blocks of pixels. A copy $B$ of a block $A$ is compressed by preparing the height, width, and location (image coordinates) of $A$, and compressing those four numbers by means of Huffman codes. (See also Discrete-Tone Image.)

**Block Matching.** A lossless image compression method based on the LZ77 sliding window method originally developed for text compression. (See also LZ Methods.)

**Block Truncation Coding.** BTC is a lossy image compression method that quantizes pixels in an image while preserving the first two or three statistical moments. (See also Vector Quantization.)

**BMP.** BMP (Section 1.4.4) is a palette-based graphics file format for images with 1, 2, 4, 8, 16, 24, or 32 bitplanes. It uses a simple form of RLE to compress images with 4 or 8 bitplanes.

**BOCU-1.** A simple algorithm for Unicode compression (Section 11.12.1).
**BSDiff.** A file differencing algorithm created by Colin Percival. The algorithm addresses the problem of differential file compression of executable code while maintaining a platform-independent approach. BSDiff combines matching with mismatches and entropy coding of the differences with `bzip2`. BSDiff’s decoder is called bspatch. (See also Exediff, File differencing, UNIX diff, VCDIFF, and Zdelta.)

**Burrows-Wheeler Method.** This method (Section 11.1) prepares a string of data for later compression. The compression itself is done with the move-to-front method (Section 1.5), perhaps in combination with RLE. The BW method converts a string $S$ to another string $L$ that satisfies two conditions:

1. Any region of $L$ will tend to have a concentration of just a few symbols.
2. It is possible to reconstruct the original string $S$ from $L$ (a little more data may be needed for the reconstruction, in addition to $L$, but not much).

**CALIC.** A context-based, lossless image compression method (Section 7.28) whose two main features are (1) the use of three passes in order to achieve symmetric contexts and (2) context quantization, to significantly reduce the number of possible contexts without degrading compression.

**CCITT.** The International Telegraph and Telephone Consultative Committee (Comité Consultatif International Télégraphique et Téléphonique), the old name of the ITU, the International Telecommunications Union. The ITU is a United Nations organization responsible for developing and recommending standards for data communications (not just compression). (See also ITU.)

**Cell Encoding.** An image compression method where the entire bitmap is divided into cells of, say, $8 \times 8$ pixels each and is scanned cell by cell. The first cell is stored in entry 0 of a table and is encoded (i.e., written on the compressed file) as the pointer 0. Each subsequent cell is searched in the table. If found, its index in the table becomes its code and it is written on the compressed file. Otherwise, it is added to the table. In the case of an image made of just straight segments, it can be shown that the table size is just 108 entries.

**CIE.** CIE is an abbreviation for Commission Internationale de l’Éclairage (International Committee on Illumination). This is the main international organization devoted to light and color. It is responsible for developing standards and definitions in this area. (See Luminance.)

**Circular Queue.** A basic data structure (Section 6.3.1) that moves data along an array in circular fashion, updating two pointers to point to the start and end of the data in the array.

**Codec.** A term used to refer to both encoder and decoder.

**Codes.** A code is a symbol that stands for another symbol. In computer and telecommunications applications, codes are virtually always binary numbers. The ASCII code is the defacto standard, although the new Unicode is used on several new computers and the older EBCDIC is still used on some old IBM computers. (See also ASCII, Unicode.)
Composite and Difference Values. A progressive image method that separates the image into layers using the method of bintrees. Early layers consist of a few large, low-resolution blocks, followed by later layers with smaller, higher-resolution blocks. The main principle is to transform a pair of pixels into two values, a composite and a differentiator. (See also Bintrees, Progressive Image Compression.)

Compress. In the large UNIX world, compress is commonly used to compress data. This utility uses LZW with a growing dictionary. It starts with a small dictionary of just 512 entries and doubles its size each time it fills up, until it reaches 64K bytes (Section 6.14). (See also LZW.)

Compression Benchmarks. In order to prove its value, an algorithm has to be implemented and tested. Thus, every researcher, programmer, and developer compares a new algorithm to older, well-established and known methods, and draws conclusions about its performance. Such comparisons are known as benchmarks.

Compression Factor. The inverse of compression ratio. It is defined as
\[
\text{compression factor} = \frac{\text{size of the input stream}}{\text{size of the output stream}}.
\]
Values greater than 1 indicate compression, and values less than 1 imply expansion. (See also Compression Ratio.)

Compression Gain. This measure is defined as
\[
100 \log_e \frac{\text{reference size}}{\text{compressed size}},
\]
where the reference size is either the size of the input stream or the size of the compressed stream produced by some standard lossless compression method.

Compression Ratio. One of several measures that are commonly used to express the efficiency of a compression method. It is the ratio
\[
\text{compression ratio} = \frac{\text{size of the output stream}}{\text{size of the input stream}}.
\]
A value of 0.6 indicates that the data occupies 60% of its original size after compression. Values greater than 1 mean an output stream bigger than the input stream (negative compression).

Sometimes the quantity $100 \times (1 - \text{compression ratio})$ is used to express the quality of compression. A value of 60 means that the output stream occupies 40% of its original size (or that the compression has resulted in a savings of 60%). (See also Compression Factor.)

Conditional Image RLE. A compression method for grayscale images with \(n\) shades of gray. The method starts by assigning an \(n\)-bit code to each pixel depending on its near neighbors. It then concatenates the \(n\)-bit codes into a long string, and calculates run lengths. The run lengths are encoded by prefix codes. (See also RLE, Relative Encoding.)
Conditional Probability. We tend to think of probability as something that is built into an experiment. A true die, for example, has probability of 1/6 of falling on any side, and we tend to consider this an intrinsic feature of the die. Conditional probability is a different way of looking at probability. It says that knowledge affects probability. The main task of this field is to calculate the probability of an event $A$ given that another event, $B$, is known to have occurred. This is the conditional probability of $A$ (more precisely, the probability of $A$ conditioned on $B$), and it is denoted by $P(A|B)$. The field of conditional probability is sometimes called Bayesian statistics, since it was first developed by the Reverend Thomas Bayes, who came up with the basic formula of conditional probability.

Context. The $N$ symbols preceding the next symbol. A context-based model uses context to assign probabilities to symbols.

Context-Free Grammars. A formal language uses a small number of symbols (called terminal symbols) from which valid sequences can be constructed. Any valid sequence is finite, the number of valid sequences is normally unlimited, and the sequences are constructed according to certain rules (sometimes called production rules). The rules can be used to construct valid sequences and also to determine whether a given sequence is valid. A production rule consists of a nonterminal symbol on the left and a string of terminal and nonterminal symbols on the right. The nonterminal symbol on the left becomes the name of the string on the right. The set of production rules constitutes the grammar of the formal language. If the production rules do not depend on the context of a symbol, the grammar is context-free. There are also context-sensitive grammars. The sequitur method of Section 11.10 is based on context-free grammars.

Context-Tree Weighting. A method for the compression of bitstrings. It can be applied to text and images, but they have to be carefully converted to bitstrings. The method constructs a context tree where bits input in the immediate past (context) are used to estimate the probability of the current bit. The current bit and its estimated probability are then sent to an arithmetic encoder, and the tree is updated to include the current bit in the context. (See also KT Probability Estimator.)

Continuous-Tone Image. A digital image with a large number of colors, such that adjacent image areas with colors that differ by just one unit appear to the eye as having continuously varying colors. An example is an image with 256 grayscale values. When adjacent pixels in such an image have consecutive gray levels, they appear to the eye as a continuous variation of the gray level. (See also Bi-level image, Discrete-Tone Image, Grayscale Image.)

Continuous Wavelet Transform. An important modern method for analyzing the time and frequency contents of a function $f(t)$ by means of a wavelet. The wavelet is itself a function (which has to satisfy certain conditions), and the transform is done by multiplying the wavelet and $f(t)$ and computing the integral of the product. The wavelet is then translated, and the process is repeated. When done, the wavelet is scaled, and the entire process is carried out again in order to analyze $f(t)$ at a different scale. (See also Discrete Wavelet Transform, Lifting Scheme, Multiresolution Decomposition, Taps.)
**Convolution.** A way to describe the output of a linear, shift-invariant system by means of its input.

**Correlation.** A statistical measure of the linear relation between two paired variables. The values of $R$ range from $-1$ (perfect negative relation), to 0 (no relation), to $+1$ (perfect positive relation).

**CRC.** CRC stands for *Cyclical Redundancy Check* (or *Cyclical Redundancy Code*). It is a rule that shows how to obtain vertical check bits from all the bits of a data stream (Section 6.32). The idea is to generate a code that depends on all the bits of the data stream, and use it to detect errors (bad bits) when the data is transmitted (or when it is stored and retrieved).

**CRT.** A CRT (cathode ray tube) is a glass tube with a familiar shape. In the back it has an electron gun (the cathode) that emits a stream of electrons. Its front surface is positively charged, so it attracts the electrons (which have a negative electric charge). The front is coated with a phosphor compound that converts the kinetic energy of the electrons hitting it to light. The flash of light lasts only a fraction of a second, so in order to get a constant display, the picture has to be refreshed several times a second.

**Data Compression Conference.** A scientific meeting of researchers and developers in the area of data compression. The DCC takes place every year in Snowbird, Utah, USA. The time is normally the second half of March and the conference lasts three days.

**Data Structure.** A set of data items used by a program and stored in memory such that certain operations (for example, finding, adding, modifying, and deleting items) can be performed on the data items fast and easily. The most common data structures are the array, stack, queue, linked list, tree, graph, and hash table. (See also Circular Queue.)

**DCA.** DCA stands for data compression with antidictionaries. An antidictionary contains bitstrings that do not appear in the input. With the help of an antidictionary, both encoder and decoder can predict with certainty the values of certain bits, which can then be eliminated from the output, thereby causing compression. (See also Antidictionary.)

**Decibel.** A logarithmic measure that can be used to measure any quantity that takes values over a very wide range. A common example is sound intensity. The intensity (amplitude) of sound can vary over a range of 11–12 orders of magnitude. Instead of using a linear measure, where numbers as small as 1 and as large as $10^{11}$ would be needed, a logarithmic scale is used, where the range of values is $[0, 11]$.

**Decoder.** A decompression program (or algorithm).

**Deflate.** A popular lossless compression algorithm (Section 6.25) used by Zip and gzip. Deflate employs a variant of LZ77 combined with static Huffman coding. It uses a 32-Kb-long sliding dictionary and a look-ahead buffer of 258 bytes. When a string is not found in the dictionary, its first symbol is emitted as a literal byte. (See also Gzip, Zip.)

**Dictionary-Based Compression.** Compression methods (Chapter 6) that save pieces of the data in a “dictionary” data structure. If a string of new data is identical to a piece that is already saved in the dictionary, a pointer to that piece is output to the compressed stream. (See also LZ Methods.)
Differential Image Compression. A lossless image compression method where each pixel \( p \) is compared to a reference pixel, which is one of its immediate neighbors, and is then encoded in two parts: a prefix, which is the number of most significant bits of \( p \) that are identical to those of the reference pixel, and a suffix, which is (almost all) the remaining least significant bits of \( p \). (See also DPCM.)

Digital Video. A form of video in which the original image is generated, in the camera, in the form of pixels. (See also High-Definition Television.)

Digram. A pair of consecutive symbols.

Discrete Cosine Transform. A variant of the discrete Fourier transform (DFT) that produces just real numbers. The DCT (Sections 7.8, 7.10.2, and 11.15.2) transforms a set of numbers by combining \( n \) numbers to become an \( n \)-dimensional point and rotating it in \( n \)-dimensions such that the first coordinate becomes dominant. The DCT and its inverse, the IDCT, are used in JPEG (Section 7.10) to compress an image with acceptable loss, by isolating the high-frequency components of an image, so that they can later be quantized. (See also Fourier Transform, Transform.)

Discrete-Tone Image. A discrete-tone image may be bi-level, grayscale, or color. Such images are (with some exceptions) artificial, having been obtained by scanning a document, or capturing a computer screen. The pixel colors of such an image do not vary continuously or smoothly, but have a small set of values, such that adjacent pixels may differ much in intensity or color. Figure 7.59 is an example of such an image. (See also Block Decomposition, Continuous-Tone Image.)

Discrete Wavelet Transform. The discrete version of the continuous wavelet transform. A wavelet is represented by means of several filter coefficients, and the transform is carried out by matrix multiplication (or a simpler version thereof) instead of by calculating an integral. (See also Continuous Wavelet Transform, Multiresolution Decomposition.)

DjVu. Certain images combine the properties of all three image types (bi-level, discrete-tone, and continuous-tone). An important example of such an image is a scanned document containing text, line drawings, and regions with continuous-tone pictures, such as paintings or photographs. DjVu (pronounced “déjà vu”) is designed for high compression and fast decompression of such documents.

It starts by decomposing the document into three components: mask, foreground, and background. The background component contains the pixels that constitute the pictures and the paper background. The mask contains the text and the lines in bi-level form (i.e., one bit per pixel). The foreground contains the color of the mask pixels. The background is a continuous-tone image and can be compressed at the low resolution of 100 dpi. The foreground normally contains large uniform areas and is also compressed as a continuous-tone image at the same low resolution. The mask is left at 300 dpi but can be efficiently compressed, since it is bi-level. The background and foreground are compressed with a wavelet-based method called IW44, while the mask is compressed with JB2, a version of JBIG2 (Section 7.15) developed at AT&T.
DPCM. DPCM compression is a member of the family of differential encoding compression methods, which itself is a generalization of the simple concept of relative encoding (Section 1.3.1). It is based on the fact that neighboring pixels in an image (and also adjacent samples in digitized sound) are correlated. (See also Differential Image Compression, Relative Encoding.)

**Embedded Coding.** This feature is defined as follows: Imagine that an image encoder is applied twice to the same image, with different amounts of loss. It produces two files, a large one of size $M$ and a small one of size $m$. If the encoder uses embedded coding, the smaller file is identical to the first $m$ bits of the larger file. The following example aptly illustrates the meaning of this definition. Suppose that three users wait for you to send them a certain compressed image, but they need different image qualities. The first one needs the quality contained in a 10 Kb file. The image qualities required by the second and third users are contained in files of sizes 20 Kb and 50 Kb, respectively. Most lossy image compression methods would have to compress the same image three times, at different qualities, to generate three files with the right sizes. An embedded encoder, on the other hand, produces one file, and then three chunks—of lengths 10 Kb, 20 Kb, and 50 Kb, all starting at the beginning of the file—can be sent to the three users, satisfying their needs. (See also SPIHT, EZW.)

**Encoder.** A compression program (or algorithm).

**Entropy.** The entropy of a single symbol $a_i$ is defined (in Section A.1) as $-P_i \log_2 P_i$, where $P_i$ is the probability of occurrence of $a_i$ in the data. The entropy of $a_i$ is the smallest number of bits needed, on average, to represent symbol $a_i$. Claude Shannon, the creator of information theory, coined the term *entropy* in 1948, because this term is used in thermodynamics to indicate the amount of disorder in a physical system. (See also Entropy Encoding, Information Theory.)

**Entropy Encoding.** A lossless compression method where data can be compressed such that the average number of bits/symbol approaches the entropy of the input symbols. (See also Entropy.)

**Error-Correcting Codes.** The opposite of data compression, these codes detect and correct errors in digital data by increasing the redundancy of the data. They use check bits or parity bits, and are sometimes designed with the help of generating polynomials.

**EXE Compressor.** A compression program for compressing EXE files on the PC. Such a compressed file can be decompressed and executed with one command. The original EXE compressor is LZEXE, by Fabrice Bellard (Section 6.29).

**Exediff.** A differential file compression algorithm created by Brenda Baker, Udi Manber, and Robert Muth for the differential compression of executable code. Exediff is an iterative algorithm that uses a lossy transform to reduce the effect of the secondary changes in executable code. Two operations called *pre-matching* and *value recovery* are iterated until the size of the patch converges to a minimum. Exediff’s decoder is called exepatch. (See also BSDiff, File differencing, UNIX diff, VCDIFF, and Zdelta.)
**EZW.** A progressive, embedded image coding method based on the zerotree data structure. It has largely been superseded by the more efficient SPIHT method. (See also SPIHT, Progressive Image Compression, Embedded Coding.)

**Facsimile Compression.** Transferring a typical page between two fax machines can take up to 10–11 minutes without compression. This is why the ITU has developed several standards for compression of facsimile data. The current standards (Section 5.7) are T4 and T6, also called Group 3 and Group 4, respectively. (See also ITU.)

**FELICS.** A Fast, Efficient, Lossless Image Compression method designed for grayscale images that competes with the lossless mode of JPEG. The principle is to code each pixel with a variable-length code based on the values of two of its previously seen neighbor pixels. Both the unary code and the Golomb code are used. There is also a progressive version of FELICS (Section 7.24). (See also Progressive FELICS.)

**FHM Curve Compression.** A method for compressing curves. The acronym FHM stands for Fibonacci, Huffman, and Markov. (See also Fibonacci Numbers.)

**Fibonacci Numbers.** A sequence of numbers defined by

\[ F_1 = 1, \quad F_2 = 1, \quad F_i = F_{i-1} + F_{i-2}, \quad i = 3, 4, \ldots \]

The first few numbers in the sequence are 1, 1, 2, 3, 5, 8, 13, and 21. These numbers have many applications in mathematics and in various sciences. They are also found in nature, and are related to the golden ratio. (See also FHM Curve Compression.)

**File Differencing.** A compression method that locates and compresses the differences between two slightly different data sets. The decoder, that has access to one of the two data sets, can use the differences and reconstruct the other. Applications of this compression technique include software distribution and updates (or patching), revision control systems, compression of backup files, archival of multiple versions of data. (See also VCDIFF.) (See also BSdiff, Exediff, UNIX diff, VCDIFF, and Zdelta.)

**FLAC.** An acronym for free lossless audio compression, FLAC is an audio compression method, somewhat resembling Shorten, that is based on prediction of audio samples and encoding of the prediction residues with Rice codes. (See also Rice codes.)

**Fourier Transform.** A mathematical transformation that produces the frequency components of a function (Section 8.1). The Fourier transform shows how a periodic function can be written as the sum of sines and cosines, thereby showing explicitly the frequencies “hidden” in the original representation of the function. (See also Discrete Cosine Transform, Transform.)

**Gaussian Distribution.** (See Normal Distribution.)

**GFA.** A compression method originally developed for bi-level images that can also be used for color images. GFA uses the fact that most images of interest have a certain amount of self-similarity (i.e., parts of the image are similar, up to size, orientation, or brightness, to the entire image or to other parts). GFA partitions the image into sub-squares using a quadtree, and expresses relations between parts of the image in a graph.
The graph is similar to graphs used to describe finite-state automata. The method is lossy, because parts of a real image may be very (although not completely) similar to other parts. (See also Quadtrees, Resolution Independent Compression, WFA.)

**GIF.** An acronym that stands for Graphics Interchange Format. This format (Section 6.21) was developed by CompuServe Information Services in 1987 as an efficient, compressed graphics file format that allows for images to be sent between computers. The original version of GIF is known as GIF 87a. The current standard is GIF 89a. (See also Patents.)

**Golomb Code.** The Golomb codes consist of an infinite set of parametrized prefix codes. They are the best ones for the compression of data items that are distributed geometrically. (See also Unary Code.)

**Gray Codes.** These are binary codes for the integers, where the codes of consecutive integers differ by one bit only. Such codes are used when a grayscale image is separated into bitplanes, each a bi-level image. (See also Grayscale Image.)

**Grayscale Image.** A continuous-tone image with shades of a single color. (See also Continuous-Tone Image.)

**Growth Geometry Coding.** A method for progressive lossless compression of bi-level images. The method selects some seed pixels and applies geometric rules to grow each seed pixel into a pattern of pixels. (See also Progressive Image Compression.)

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

**Gzip.** Popular software that implements the Deflate algorithm (Section 6.25) that uses a variation of LZ77 combined with static Huffman coding. It uses a 32 Kb-long sliding dictionary, and a look-ahead buffer of 258 bytes. When a string is not found in the dictionary, it is emitted as a sequence of literal bytes. (See also Zip.)

**H.261.** In late 1984, the CCITT (currently the ITU-T) organized an expert group to develop a standard for visual telephony for ISDN services. The idea was to send images and sound between special terminals, so that users could talk and see each other. This type of application requires sending large amounts of data, so compression became an important consideration. The group eventually came up with a number of standards, known as the H series (for video) and the G series (for audio) recommendations, all operating at speeds of $p \times 64$ Kbit/sec for $p = 1, 2, \ldots, 30$. These standards are known today under the umbrella name of $p \times 64$.

**H.264.** A sophisticated method for the compression of video. This method is a successor of H.261, H.262, and H.263. It has been approved in 2003 and employs the main building blocks of its predecessors, but with many additions and improvements.

**HD Photo.** A compression standard (algorithm and file format) for continuous-tone images. HD Photo was developed from Windows Media Photo, a Microsoft compression algorithm. HD Photo follows the basic steps of JPEG, but employs an integer transform instead of the DCT. (See also JPEG, JPEG XR.)
Halftoning. A method for the display of gray scales in a bi-level image. By placing groups of black and white pixels in carefully designed patterns, it is possible to create the effect of a gray area. The trade-off of halftoning is loss of resolution. (See also Bi-level Image, Dithering.)

Hamming Codes. A type of error-correcting code for 1-bit errors, where it is easy to generate the required parity bits.

Hierarchical Progressive Image Compression. An image compression method (or an optional part of such a method) where the encoder writes the compressed image in layers of increasing resolution. The decoder decompresses the lowest-resolution layer first, displays this crude image, and continues with higher-resolution layers. Each layer in the compressed stream uses data from the preceding layer. (See also Progressive Image Compression.)

High-Definition Television. A general name for several standards that are currently replacing traditional television. HDTV uses digital video, high-resolution images, and aspect ratios different from the traditional 3:4. (See also Digital Video.)

Huffman Coding. A popular method for data compression (Section 5.2). It assigns a set of “best” variable-length codes to a set of symbols based on their probabilities. It serves as the basis for several popular programs used on personal computers. Some of them use just the Huffman method, while others use it as one step in a multistep compression process. The Huffman method is somewhat similar to the Shannon-Fano method. It generally produces better codes, and like the Shannon-Fano method, it produces best code when the probabilities of the symbols are negative powers of 2. The main difference between the two methods is that Shannon-Fano constructs its codes top to bottom (from the leftmost to the rightmost bits), while Huffman constructs a code tree from the bottom up (builds the codes from right to left). (See also Shannon-Fano Coding, Statistical Methods.)

Hutter prize. The Hutter prize (Section 5.13 tries to encourage the development of new methods for text compression.

Hyperspectral data. A set of data items (called pixels) arranged in rows and columns where each pixel is a vector. An example is an image where each pixel consists of the radiation reflected from the ground in many frequencies. We can think of such data as several image planes (called bands) stacked vertically. Hyperspectral data is normally large and is an ideal candidate for compression. Any compression method for this type of data should take advantage of the correlation between bands as well as correlations between pixels in the same band.

Information Theory. A mathematical theory that quantifies information. It shows how to measure information, so that one can answer the question; How much information is included in a given piece of data? with a precise number! Information theory is the creation, in 1948, of Claude Shannon of Bell labs. (See also Entropy.)

Interpolating Polynomials. Given two numbers $a$ and $b$ we know that $m = 0.5a + 0.5b$ is their average, since it is located midway between $a$ and $b$. We say that the average is an interpolation of the two numbers. Similarly, the weighted sum $0.1a +$
0.9b represents an interpolated value located 10% away from b and 90% away from a. Extending this concept to points (in two or three dimensions) is done by means of interpolating polynomials. Given a set of points, we start by fitting a parametric polynomial $P(t)$ or $P(u, w)$ through them. Once the polynomial is known, it can be used to calculate interpolated points by computing $P(0.5)$, $P(0.1)$, or other values.

**Interpolative Coding.** An algorithm that assigns dynamic variable-length codes to a strictly monotonically increasing sequence of integers (Section 3.28).

**ISO.** The International Standards Organization. This is one of the organizations responsible for developing standards. Among other things it is responsible (together with the ITU) for the JPEG and MPEG compression standards. (See also ITU, CCITT, MPEG.)

**Iterated Function Systems (IFS).** An image compressed by IFS is uniquely defined by a few affine transformations (Section 7.39.1). The only rule is that the scale factors of these transformations must be less than 1 (shrinking). The image is saved in the output stream by writing the sets of six numbers that define each transformation. (See also Affine Transformations, Resolution Independent Compression.)

**ITU.** The International Telecommunications Union, the new name of the CCITT, is a United Nations organization responsible for developing and recommending standards for data communications (not just compression). (See also CCITT.)

**JBIG.** A special-purpose compression method (Section 7.14) developed specifically for progressive compression of bi-level images. The name JBIG stands for Joint Bi-Level Image Processing Group. This is a group of experts from several international organizations, formed in 1988 to recommend such a standard. JBIG uses multiple arithmetic coding and a resolution-reduction technique to achieve its goals. (See also Bi-level Image, JBIG2.)

**JBIG2.** A recent international standard for the compression of bi-level images. It is intended to replace the original JBIG. Its main features are

1. Large increases in compression performance (typically 3–5 times better than Group 4/MMR, and 2–4 times better than JBIG).
2. Special compression methods for text, halftones, and other bi-level image parts.
3. Lossy and lossless compression modes.
4. Two modes of progressive compression. Mode 1 is quality-progressive compression, where the decoded image progresses from low to high quality. Mode 2 is content-progressive coding, where important image parts (such as text) are decoded first, followed by less important parts (such as halftone patterns).
6. Flexible format, designed for easy embedding in other image file formats, such as TIFF.
7. Fast decompression. In some coding modes, images can be decompressed at over 250 million pixels/second in software.

(See also Bi-level Image, JBIG.)
**JFIF.** The full name of this method (Section 7.10.7) is JPEG File Interchange Format. It is a graphics file format that makes it possible to exchange JPEG-compressed images between different computers. The main features of JFIF are the use of the YCbCr triple-component color space for color images (only one component for grayscale images) and the use of a *marker* to specify features missing from JPEG, such as image resolution, aspect ratio, and features that are application-specific.

**JPEG.** A sophisticated lossy compression method (Section 7.10) for color or grayscale still images (not movies). It works best on continuous-tone images, where adjacent pixels have similar colors. One advantage of JPEG is the use of many parameters, allowing the user to adjust the amount of data loss (and thereby also the compression ratio) over a very wide range. There are two main modes: lossy (also called baseline) and lossless (which typically yields a 2:1 compression ratio). Most implementations support just the lossy mode. This mode includes progressive and hierarchical coding.

The main idea behind JPEG is that an image exists for people to look at, so when the image is compressed, it is acceptable to lose image features to which the human eye is not sensitive.

The name JPEG is an acronym that stands for Joint Photographic Experts Group. This was a joint effort by the CCITT and the ISO that started in June 1987. The JPEG standard has proved successful and has become widely used for image presentation, especially in Web pages. (See also JPEG-LS, MPEG.)

**JPEG-LS.** The lossless mode of JPEG is inefficient and often is not even implemented. As a result, the ISO decided to develop a new standard for the lossless (or near-lossless) compression of continuous-tone images. The result became popularly known as JPEG-LS. This method is not simply an extension or a modification of JPEG. It is a new method, designed to be simple and fast. It does not employ the DCT, does not use arithmetic coding, and applies quantization in a limited way, and only in its near-lossless option. JPEG-LS examines several of the previously-seen neighbors of the current pixel, uses them as the *context* of the pixel, employs the context to predict the pixel and to select a probability distribution out of several such distributions, and uses that distribution to encode the prediction error with a special Golomb code. There is also a run mode, where the length of a run of identical pixels is encoded. (See also Golomb Code, JPEG.)

**JPEG XR.** See HD Photo.

As for my mother, perhaps the Ambassador had not the type of mind towards which she felt herself most attracted. I should add that his conversation furnished so exhaustive a glossary of the superannuated forms of speech peculiar to a certain profession, class and period.

—Marcel Proust, *Within a Budding Grove* (1913–1927)

**Kraft-MacMillan Inequality.** A relation (Section 2.5) that says something about unambiguous variable-length codes. Its first part states: Given an unambiguous variable-length
code, with \( n \) codes of lengths \( L_i \), then
\[
\sum_{i=1}^{n} 2^{-L_i} \leq 1.
\]

[This is Equation (2.3).] The second part states the opposite, namely, given a set of \( n \)
positive integers \( (L_1, L_2, \ldots, L_n) \) that satisfy Equation (2.3), there exists an unambiguous
variable-length code such that \( L_i \) are the sizes of its individual codes. Together,
both parts state that a code is unambiguous if and only if it satisfies relation (2.3).

**KT Probability Estimator.** A method to estimate the probability of a bitstring containing \( a \) zeros and \( b \) ones. It is due to Krichevsky and Trofimov. (See also Context-Tree Weighting.)

**Laplace Distribution.** A probability distribution similar to the normal (Gaussian) distribution, but narrower and sharply peaked. The general Laplace distribution with variance \( V \) and mean \( m \) is given by
\[
L(V, x) = \frac{1}{\sqrt{2V}} \exp \left( -\frac{2}{V} |x - m| \right).
\]

Experience seems to suggest that the values of the residues computed by many image compression algorithms are Laplace distributed, which is why this distribution is employed by those compression methods, most notably MLP. (See also Normal Distribution.)

**Laplacian Pyramid.** A progressive image compression technique where the original image is transformed to a set of difference images that can later be decompressed and displayed as a small, blurred image that becomes increasingly sharper. (See also Progressive Image Compression.)

**LHArc.** This method (Section 6.24) is by Haruyasu Yoshizaki. Its predecessor is LHA, designed jointly by Haruyasu Yoshizaki and Haruhiko Okumura. These methods are based on adaptive Huffman coding with features drawn from LZSS.

**Lifting Scheme.** A method for computing the discrete wavelet transform in place, so no extra memory is required. (See also Discrete Wavelet Transform.)

**Locally Adaptive Compression.** A compression method that adapts itself to local conditions in the input stream, and varies this adaptation as it moves from area to area in the input. An example is the move-to-front method of Section 1.5. (See also Adaptive Compression, Semiadaptive Compression.)

**Lossless Compression.** A compression method where the output of the decoder is identical to the original data compressed by the encoder. (See also Lossy Compression.)

**Lossy Compression.** A compression method where the output of the decoder is different from the original data compressed by the encoder, but is nevertheless acceptable to a user. Such methods are common in image and audio compression, but not in text compression, where the loss of even one character may result in wrong, ambiguous, or incomprehensible text. (See also Lossless Compression, Subsampling.)
LPVQ. An acronym for Locally Optimal Partitioned Vector Quantization. LPVQ is a quantization algorithm proposed by Giovanni Motta, Francesco Rizzo, and James Storer [Motta et al. 06] for the lossless and near-lossless compression of hyperspectral data. Spectral signatures are first partitioned in sub-vectors on unequal length and independently quantized. Then, indices are entropy coded by exploiting both spectral and spatial correlation. The residual error is also entropy coded, with the probabilities conditioned by the quantization indices. The locally optimal partitioning of the spectral signatures is decided at design time, during the training of the quantizer.

Luminance. This quantity is defined by the CIE (Section 7.10.1) as radiant power weighted by a spectral sensitivity function that is characteristic of vision. (See also CIE.)

LZ Methods. All dictionary-based compression methods are based on the work of J. Ziv and A. Lempel, published in 1977 and 1978. Today, these are called LZ77 and LZ78 methods, respectively. Their ideas have been a source of inspiration to many researchers, who generalized, improved, and combined them with RLE and statistical methods to form many commonly used adaptive compression methods, for text, images, and audio. (See also Block Matching, Dictionary-Based Compression, Sliding-Window Compression.)

LZAP. The LZAP method (Section 6.16) is an LZW variant based on the following idea: Instead of just concatenating the last two phrases and placing the result in the dictionary, place all prefixes of the concatenation in the dictionary. The suffix AP stands for All Prefixes.

LZARI. An improvement on LZSS, developed in 1988 by Haruhiko Okumura. (See also LZSS.)

LZB. LZB is the result of evaluating and comparing several data structures and variable-length codes with an eye to improving the performance of LZSS.

LZC. See Compress, LZT.

LZFG. This is the name of several related methods (Section 6.10) that are hybrids of LZ77 and LZ78. They were developed by Edward Fiala and Daniel Greene. All these methods are based on the following scheme. The encoder produces a compressed file with tokens and literals (raw ASCII codes) intermixed. There are two types of tokens, a literal and a copy. A literal token indicates that a string of literals follow, a copy token points to a string previously seen in the data. (See also LZ Methods, Patents.)

LZJ. LZJ (Section 6.17) is an interesting LZ variant. It stores in its dictionary, which can be viewed either as a multiway tree or as a forest, every phrase found in the input. If a phrase if found n times in the input, only one copy is stored in the dictionary.

LZMA. LZMA (Lempel-Ziv-Markov chain-Algorithm) is one of the many LZ77 variants. Developed by Igor Pavlov, this algorithm, which is used in his popular 7z software, is based on a large search buffer, a hash function that generates indexes, somewhat similar to LZRW4, and two search methods. The fast method uses a hash-array of lists of indexes and the normal method uses a hash-array of binary decision trees. (See also 7-Zip.)
**LZMW.** A variant of LZW, the LZMW method (Section 6.15) works as follows: Instead of adding I plus one character of the next phrase to the dictionary, add I plus the entire next phrase to the dictionary. (See also LZW.)

**LZP.** An LZ77 variant developed by C. Bloom (Section 6.19). It is based on the principle of context prediction that says “if a certain string \( abcde \) has appeared in the input stream in the past and was followed by \( fg \ldots \), then when \( abcde \) appears again in the input stream, there is a good chance that it will be followed by the same \( fg \ldots \)” (See also Context.)

**LZPP.** LZPP is a modern, sophisticated algorithm that extends LZSS in several directions. LZPP identifies several sources of redundancy in the various quantities generated and manipulated by LZSS and exploits these sources to obtain better overall compression. (See also LZSS.)

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

**LZSS.** This version of LZ77 (Section 6.4) was developed by Storer and Szymanski in 1982 [Storer 82]. It improves on the basic LZ77 in three ways: (1) it holds the look-ahead buffer in a circular queue, (2) it implements the search buffer (the dictionary) in a binary search tree, and (3) it creates tokens with two fields instead of three. (See also LZ Methods, LZARI, LZPP, SLH.)

**LZT.** LZT is an extension of UNIX compress/LZC. The major innovation of LZT is the way it handles a full dictionary. (See also Compress.)

**LZW.** This is a popular variant (Section 6.13) of LZ78, developed by Terry Welch in 1984. Its main feature is eliminating the second field of a token. An LZW token consists of just a pointer to the dictionary. As a result, such a token always encodes a string of more than one symbol. (See also Patents.)

**LZW-L.** A syllable-based variant of LZW. (See also Syllable-Based Compression.)

**LZX.** LZX is an LZ77 variant for the compression of cabinet files (Section 6.8).

**LZY.** LZY (Section 6.18) is an LZW variant that adds one dictionary string per input character and increments strings by one character at a time.

**MLP.** A progressive compression method for grayscale images. An image is compressed in levels. A pixel is predicted by a symmetric pattern of its neighbors from preceding levels, and the prediction error is arithmetically encoded. The Laplace distribution is used to estimate the probability of the error. (See also Laplace Distribution, Progressive FELICS.)

**MLP Audio.** The new lossless compression standard approved for DVD-A (audio) is called MLP. It is the topic of Section 10.7.

**MNP5, MNP7.** These have been developed by Microcom, Inc., a maker of modems, for use in its modems. MNP5 (Section 5.4) is a two-stage process that starts with run-length encoding, followed by adaptive frequency encoding. MNP7 (Section 5.5) combines run-length encoding with a two-dimensional variant of adaptive Huffman.
Model of Compression. A model is a method to “predict” (to assign probabilities to) the data to be compressed. This concept is important in statistical data compression. When a statistical method is used, a model for the data has to be constructed before compression can begin. A simple model can be built by reading the entire input stream, counting the number of times each symbol appears (its frequency of occurrence), and computing the probability of occurrence of each symbol. The data stream is then input again, symbol by symbol, and is compressed using the information in the probability model. (See also Statistical Methods, Statistical Model.)

One feature of arithmetic coding is that it is easy to separate the statistical model (the table with frequencies and probabilities) from the encoding and decoding operations. It is easy to encode, for example, the first half of a data stream using one model, and the second half using another model.

Monkey’s audio. Monkey’s audio is a fast, efficient, free, lossless audio compression algorithm and implementation that offers error detection, tagging, and external support.

Move-to-Front Coding. The basic idea behind this method (Section 1.5) is to maintain the alphabet $A$ of symbols as a list where frequently occurring symbols are located near the front. A symbol $s$ is encoded as the number of symbols that precede it in this list. After symbol $s$ is encoded, it is moved to the front of list $A$.

MPEG. This acronym stands for Moving Pictures Experts Group. The MPEG standard consists of several methods for the compression of video, including the compression of digital images and digital sound, as well as synchronization of the two. There currently are several MPEG standards. MPEG-1 is intended for intermediate data rates, on the order of 1.5 Mbit/sec. MPEG-2 is intended for high data rates of at least 10 Mbit/sec. MPEG-3 was intended for HDTV compression but was found to be redundant and was merged with MPEG-2. MPEG-4 is intended for very low data rates of less than 64 Kbit/sec. The ITU-T, has been involved in the design of both MPEG-2 and MPEG-4. A working group of the ISO is still at work on MPEG. (See also ISO, JPEG.)

Multiresolution Decomposition. This method groups all the discrete wavelet transform coefficients for a given scale, displays their superposition, and repeats for all scales. (See also Continuous Wavelet Transform, Discrete Wavelet Transform.)

Multiresolution Image. A compressed image that may be decompressed at any resolution. (See also Resolution Independent Compression, Iterated Function Systems, WFA.)

Normal Distribution. A probability distribution with the well-known bell shape. It is found in many places in both theoretical models and real-life situations. The normal distribution with mean $m$ and standard deviation $s$ is defined by

$$f(x) = \frac{1}{s\sqrt{2\pi}} \exp \left\{ -\frac{1}{2} \left( \frac{x - m}{s} \right)^2 \right\}.$$ 

PAQ. An open-source, high-performance compression algorithm and free software (Section 5.15) that features sophisticated prediction combined with adaptive arithmetic encoding.
**Patents.** A mathematical algorithm can be patented if it is intimately associated with software or firmware implementing it. Several compression methods, most notably LZW, have been patented (Section 6.34), creating difficulties for software developers who work with GIF, UNIX *compress*, or any other system that uses LZW. (See also GIF, LZW, Compress.)

**Pel.** The smallest unit of a facsimile image; a dot. (See also Pixel.)

**phased-in codes.** Phased-in codes (Section 2.9) are a minor extension of fixed-length codes and may contribute a little to the compression of a set of consecutive integers by changing the representation of the integers from fixed $n$ bits to either $n$ or $n - 1$ bits.

**Phrase.** A piece of data placed in a dictionary to be used in compressing future data. The concept of phrase is central in dictionary-based data compression methods since the success of such a method depends a lot on how it selects phrases to be saved in its dictionary. (See also Dictionary-Based Compression, LZ Methods.)

**Pixel.** The smallest unit of a digital image; a dot. (See also Pel.)

**PKZip.** A compression program for MS/DOS (Section 6.24) written by Phil Katz who has founded the PKWare company which also markets the PKunzip, PKlite, and PKArc software (http://www.pkware.com).

**PNG.** An image file format (Section 6.27) that includes lossless compression with Deflate and pixel prediction. PNG is free and it supports several image types and number of bitplanes, as well as sophisticated transparency.

**Portable Document Format (PDF).** A standard developed by Adobe in 1991–1992 that allows arbitrary documents to be created, edited, transferred between different computer platforms, and printed. PDF compresses the data in the document (text and images) by means of LZW, Flate (a variant of Deflate), run-length encoding, JPEG, JBIG2, and JPEG 2000.

**PPM.** A compression method that assigns probabilities to symbols based on the context (long or short) in which they appear. (See also Prediction, PPPM.)

**PPPM.** A lossless compression method for grayscale (and color) images that assigns probabilities to symbols based on the Laplace distribution, like MLP. Different contexts of a pixel are examined, and their statistics used to select the mean and variance for a particular Laplace distribution. (See also Laplace Distribution, Prediction, PPM, MLP.)

**Prediction.** Assigning probabilities to symbols. (See also PPM.)

**Prefix Compression.** A variant of quadtrees, designed for bi-level images with text or diagrams, where the number of black pixels is relatively small. Each pixel in a $2^n \times 2^n$ image is assigned an $n$-digit, or $2n$-bit, number based on the concept of quadtrees. Numbers of adjacent pixels tend to have the same prefix (most-significant bits), so the common prefix and different suffixes of a group of pixels are compressed separately. (See also Quadtrees.)
Prefix Property. One of the principles of variable-length codes. It states: Once a certain bit pattern has been assigned as the code of a symbol, no other codes should start with that pattern (the pattern cannot be the prefix of any other code). Once the string 1, for example, is assigned as the code of $a_1$, no other codes should start with 1 (i.e., they all have to start with 0). Once $01$, for example, is assigned as the code of $a_2$, no other codes can start with $01$ (they all should start with $00$). (See also Variable-Length Codes, Statistical Methods.)

Progressive FELICS. A progressive version of FELICS where pixels are encoded in levels. Each level doubles the number of pixels encoded. To decide what pixels are included in a certain level, the preceding level can conceptually be rotated $45^\circ$ and scaled by $\sqrt{2}$ in both dimensions. (See also FELICS, MLP, Progressive Image Compression.)

Progressive Image Compression. An image compression method where the compressed stream consists of “layers,” where each layer contains more detail of the image. The decoder can very quickly display the entire image in a low-quality format, and then improve the display quality as more and more layers are being read and decompressed. A user watching the decompressed image develop on the screen can normally recognize most of the image features after only 5–10% of it has been decompressed. Improving image quality over time can be done by (1) sharpening it, (2) adding colors, or (3) increasing its resolution. (See also Progressive FELICS, Hierarchical Progressive Image Compression, MLP, JBIG.)

Psychoacoustic Model. A mathematical model of the sound masking properties of the human auditory (ear brain) system.

QIC-122 Compression. An LZ77 variant that has been developed by the QIC organization for text compression on 1/4-inch data cartridge tape drives.

QM Coder. This is the arithmetic coder of JPEG and JBIG. It is designed for simplicity and speed, so it is limited to input symbols that are single bits and it employs an approximation instead of exact multiplication. It also uses fixed-precision integer arithmetic, so it has to resort to renormalization of the probability interval from time to time, in order for the approximation to remain close to the true multiplication. (See also Arithmetic Coding.)

Quadrisection. This is a relative of the quadtree method. It assumes that the original image is a $2^k \times 2^k$ square matrix $M_0$, and it constructs matrices $M_1, M_2, \ldots, M_{k+1}$ with fewer and fewer columns. These matrices naturally have more and more rows, and quadrisection achieves compression by searching for and removing duplicate rows. Two closely related variants of quadrisection are bisection and octasection (See also Quadtrees.)

Quadtrees. This is a data compression method for bitmap images. A quadtree (Section 7.34) is a tree where each leaf corresponds to a uniform part of the image (a quadrant, subquadrant, or a single pixel) and each interior node has exactly four children. (See also Bintrees, Prefix Compression, Quadrisection.)

Quaternary. A base-4 digit. It can be 0, 1, 2, or 3.
RAR. RAR An LZ77 variant designed and developed by Eugene Roshal. RAR is extremely popular with Windows users and is available for a variety of platforms. In addition to excellent compression and good encoding speed, RAR offers options such as error-correcting codes and encryption. (See also Rarissimo.)

Rarissimo. A file utility that’s always used in conjunction with RAR. It is designed to periodically check certain source folders, automatically compress and decompress files found there, and then move those files to designated target folders. (See also RAR.)

RBUC. A compromise between the standard binary (β) code and the Elias gamma codes. (See also BASC.)

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

Relative Encoding. A variant of RLE, sometimes called differencing (Section 1.3.1). It is used in cases where the data to be compressed consists of a string of numbers that don’t differ by much, or in cases where it consists of strings that are similar to each other. The principle of relative encoding is to send the first data item $a_1$ followed by the differences $a_{i+1} - a_i$. (See also DPCM, RLE.)

Reliability. Variable-length codes and other codes are vulnerable to errors. In cases where reliable storage and transmission of codes are important, the codes can be made more reliable by adding check bits, parity bits, or CRC (Section 5.6). Notice that reliability is, in a sense, the opposite of data compression, because it is achieved by increasing redundancy. (See also CRC.)

Resolution Independent Compression. An image compression method that does not depend on the resolution of the specific image being compressed. The image can be decompressed at any resolution. (See also Multiresolution Images, Iterated Function Systems, WFA.)

Rice Codes. A special case of the Golomb code. (See also Golomb Codes.)

RLE. RLE stands for run-length encoding. This is a general name for methods that compress data by replacing a run of identical symbols with a single code, or token, that encodes the symbol and the length of the run. RLE sometimes serves as one step in a multistep statistical or dictionary-based method. (See also Relative Encoding, Conditional Image RLE.)

Scalar Quantization. The dictionary definition of the term “quantization” is “to restrict a variable quantity to discrete values rather than to a continuous set of values.” If the data to be compressed is in the form of large numbers, quantization is used to convert them to small numbers. This results in (lossy) compression. If the data to be compressed is analog (e.g., a voltage that changes with time), quantization is used to digitize it into small numbers. This aspect of quantization is used by several audio compression methods. (See also Vector Quantization.)

SCSU. A compression algorithm designed specifically for compressing text files in Unicode (Section 11.12).
SemiAdaptive Compression. A compression method that uses a two-pass algorithm, where the first pass reads the input stream to collect statistics on the data to be compressed, and the second pass performs the actual compression. The statistics (model) are included in the compressed stream. (See also Adaptive Compression, Locally Adaptive Compression.)

Semistructured Text. Such text is defined as data that is human readable and also suitable for machine processing. A common example is HTML. The sequitur method of Section 11.10 performs especially well on such text.

Shannon-Fano Coding. An early algorithm for finding a minimum-length variable-length code given the probabilities of all the symbols in the data (Section 5.1). This method was later superseded by the Huffman method. (See also Statistical Methods, Huffman Coding.)

Shorten. A simple compression algorithm for waveform data in general and for speech in particular (Section 10.9). Shorten employs linear prediction to compute residues (of audio samples) which it encodes by means of Rice codes. (See also Rice codes.)

Simple Image. A simple image is one that uses a small fraction of the possible grayscale values or colors available to it. A common example is a bi-level image where each pixel is represented by eight bits. Such an image uses just two colors out of a palette of 256 possible colors. Another example is a grayscale image scanned from a bi-level image. Most pixels will be black or white, but some pixels may have other shades of gray. A cartoon is also an example of a simple image (especially a cheap cartoon, where just a few colors are used). A typical cartoon consists of uniform areas, so it may use a small number of colors out of a potentially large palette. The EIDAC method of Section 7.16 is especially designed for simple images.

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

Sliding Window Compression. The LZ77 method (Section 6.3) uses part of the previously seen input stream as the dictionary. The encoder maintains a window to the input stream, and shifts the input in that window from right to left as strings of symbols are being encoded. The method is therefore based on a sliding window. (See also LZ Methods.)

Space-Filling Curves. A space-filling curve (Section 7.36) is a function $P(t)$ that goes through every point in a given two-dimensional region, normally the unit square, as $t$ varies from 0 to 1. Such curves are defined recursively and are used in image compression.

Sparse Strings. Regardless of what the input data represents—text, binary, images, or anything else—we can think of the input stream as a string of bits. If most of the bits are zeros, the string is sparse. Sparse strings can be compressed very efficiently by specially designed methods (Section 11.5).

Spatial Prediction. An image compression method that is a combination of JPEG and fractal-based image compression.
**Glossary**

**SPIHT.** A progressive image encoding method that efficiently encodes the image after it has been transformed by any wavelet filter. SPIHT is embedded, progressive, and has a natural lossy option. It is also simple to implement, fast, and produces excellent results for all types of images. (See also EZW, Progressive Image Compression, Embedded Coding, Discrete Wavelet Transform.)

**Statistical Methods.** These methods (Chapter 5) work by assigning variable-length codes to symbols in the data, with the shorter codes assigned to symbols or groups of symbols that appear more often in the data (have a higher probability of occurrence). (See also Variable-Length Codes, Prefix Property, Shannon-Fano Coding, Huffman Coding, and Arithmetic Coding.)

**Statistical Model.** (See Model of Compression.)

**String Compression.** In general, compression methods based on strings of symbols can be more efficient than methods that compress individual symbols (Section 6.1).

**Stuffit.** Stuffit (Section 11.16) is compression software for the Macintosh platform, developed in 1987. The methods and algorithms it employs are proprietary, but some information exists in various patents.

**Subsampling.** Subsampling is, possibly, the simplest way to compress an image. One approach to subsampling is simply to ignore some of the pixels. The encoder may, for example, ignore every other row and every other column of the image, and write the remaining pixels (which constitute 25% of the image) on the compressed stream. The decoder inputs the compressed data and uses each pixel to generate four identical pixels of the reconstructed image. This, of course, involves the loss of much image detail and is rarely acceptable. (See also Lossy Compression.)

**SVC.** SVC (scalable video coding) is an extension to H.264 that supports temporal, spatial, and quality scalable video coding, while retaining a base layer that is still backward compatible with the original H.264/AVC standard.

**Syllable-Based Compression.** An approach to compression where the basic data symbols are syllables, a syntactic unit between letters and words. (See also LZWL.)

**Symbol.** The smallest unit of the data to be compressed. A symbol is often a byte but may also be a bit, a trit \{0, 1, 2\}, or anything else. (See also Alphabet.)

**Symbol Ranking.** A context-based method (Section 11.2) where the context C of the current symbol S (the N symbols preceding S) is used to prepare a list of symbols that are likely to follow C. The list is arranged from most likely to least likely. The position of S in this list (position numbering starts from 0) is then written by the encoder, after being suitably encoded, on the output stream.

**Taps.** Wavelet filter coefficients. (See also Continuous Wavelet Transform, Discrete Wavelet Transform.)

**TAR.** The standard UNIX archiver. The name TAR stands for Tape ARchive. It groups a number of files into one file without compression. After being compressed by the UNIX compress program, a TAR file gets an extension name of tar.z.
**Textual Image Compression.** A compression method for hard copy documents containing printed or typed (but not handwritten) text. The text can be in many fonts and may consist of musical notes, hieroglyphs, or any symbols. Pattern recognition techniques are used to recognize text characters that are identical or at least similar. One copy of each group of identical characters is kept in a library. Any leftover material is considered residue. The method uses different compression techniques for the symbols and the residue. It includes a lossy option where the residue is ignored.

**Time/frequency (T/F) codec.** An audio codec that employs a psychoacoustic model to determine how the normal threshold of the ear varies (in both time and frequency) in the presence of masking sounds.

**Token.** A unit of data written on the compressed stream by some compression algorithms. A token consists of several fields that may have either fixed or variable sizes.

**Transform.** An image can be compressed by transforming its pixels (which are correlated) to a representation where they are decorrelated. Compression is achieved if the new values are smaller, on average, than the original ones. Lossy compression can be achieved by quantizing the transformed values. The decoder inputs the transformed values from the compressed stream and reconstructs the (precise or approximate) original data by applying the opposite transform. (See also Discrete Cosine Transform, Fourier Transform, Continuous Wavelet Transform, Discrete Wavelet Transform.)

**Triangle Mesh.** Polygonal surfaces are very popular in computer graphics. Such a surface consists of flat polygons, mostly triangles, so there is a need for special methods to compress a triangle mesh. One such a method is edgebreaker (Section 11.11).

**Trit.** A ternary (base 3) digit. It can be 0, 1, or 2.

**Tunstall codes.** Tunstall codes are a variation on variable-length codes. They are fixed-size codes, each encoding a variable-length string of data symbols.

**Unary Code.** A way to generate variable-length codes of the integers in one step. The unary code of the nonnegative integer \( n \) is defined (Section 3.1) as \( n – 1 \) 1’s followed by a single 0 (Table 3.1). There is also a general unary code. (See also Golomb Code.)

**Unicode.** A new international standard code, the Unicode, has been proposed, and is being developed by the international Unicode organization ([www.unicode.org](http://www.unicode.org)). Unicode uses 16-bit codes for its characters, so it provides for \( 2^{16} = 64K = 65,536 \) codes. (Notice that doubling the size of a code much more than doubles the number of possible codes. In fact, it squares the number of codes.) Unicode includes all the ASCII codes in addition to codes for characters in foreign languages (including complete sets of Korean, Japanese, and Chinese characters) and many mathematical and other symbols. Currently, about 39,000 out of the 65,536 possible codes have been assigned, so there is room for adding more symbols in the future.

The Microsoft Windows NT operating system has adopted Unicode, as have also AT&T Plan 9 and Lucent Inferno. (See also ASCII, Codes.)
UNIX diff. A file differencing algorithm that uses APPEND, DELETE and CHANGE to encode the differences between two text files. diff generates an output that is human-readable or, optionally, it can generate batch commands for a text editor like ed. (See also BSdiff, Exediff, File differencing, VCDIFF, and Zdelta.)

V.42bis Protocol. This is a standard, published by the ITU-T (page 248) for use in fast modems. It is based on the older V.32bis protocol and is supposed to be used for fast transmission rates, up to 57.6K baud. The standard contains specifications for data compression and error correction, but only the former is discussed, in Section 6.23.

V.42bis specifies two modes: a transparent mode, where no compression is used, and a compressed mode using an LZW variant. The former is used for data streams that don't compress well, and may even cause expansion. A good example is an already compressed file. Such a file looks like random data, it does not have any repetitive patterns, and trying to compress it with LZW will fill up the dictionary with short, two-symbol, phrases.

Variable-Length Codes. These are used by statistical methods. Many codes of this type satisfy the prefix property (Section 2.1) and should be assigned to symbols based on the probability distribution of the symbols. (See also Prefix Property, Statistical Methods.)

VC-1. VC-1 (Section 9.11) is a hybrid video codec, developed by SMPTE in 2006, that is based on the two chief principles of video compression, a transform and motion compensation. This codec is intended for use in a wide variety of video applications and promises to perform well at a wide range of bitrates from 10 Kbps to about 135 Mbps.

VCDIFF. A method for compressing the differences between two files. (See also BSdiff, Exediff, File differencing, UNIX diff, and Zdelta.)

Vector Quantization. This is a generalization of the scalar quantization method. It is used for both image and audio compression. In practice, vector quantization is commonly used to compress data that has been digitized from an analog source, such as sampled sound and scanned images (drawings or photographs). Such data is called digitally sampled analog data (DSAD). (See also Scalar Quantization.)

Video Compression. Video compression is based on two principles. The first is the spatial redundancy that exists in each video frame. The second is the fact that very often, a video frame is very similar to its immediate neighbors. This is called temporal redundancy. A typical technique for video compression should therefore start by encoding the first frame using an image compression method. It should then encode each successive frame by identifying the differences between the frame and its predecessor, and encoding these differences.

Voronoi Diagrams. Imagine a petri dish ready for growing bacteria. Four bacteria of different types are simultaneously placed in it at different points and immediately start multiplying. We assume that their colonies grow at the same rate. Initially, each colony consists of a growing circle around one of the starting points. After a while some of them meet and stop growing in the meeting area due to lack of food. The final result is that the entire dish gets divided into four areas, one around each of the four starting points, such that all the points within area i are closer to starting point i than to any other start point. Such areas are called Voronoi regions or Dirichlet Tessellations.
WavPack. WavPack is an open, multiplatform audio compression algorithm and software that supports three compression modes, lossless, high-quality lossy, and a unique hybrid mode. WavPack handles integer audio samples up to 32-bits wide and also 32-bit IEEE floating-point audio samples. It employs an original entropy encoder that assigns variable-length Golomb codes to the residuals and also has a recursive Golomb coding mode for cases where the distribution of the residuals is not geometric.

WFA. This method uses the fact that most images of interest have a certain amount of self-similarity (i.e., parts of the image are similar, up to size or brightness, to the entire image or to other parts). It partitions the image into subsquares using a quadtree, and uses a recursive inference algorithm to express relations between parts of the image in a graph. The graph is similar to graphs used to describe finite-state automata. The method is lossy, since parts of a real image may be very similar to other parts. WFA is a very efficient method for compression of grayscale and color images. (See also GFA, Quadtrees, Resolution-Independent Compression.)

WSQ. An efficient lossy compression method specifically developed for compressing fingerprint images. The method involves a wavelet transform of the image, followed by scalar quantization of the wavelet coefficients, and by RLE and Huffman coding of the results. (See also Discrete Wavelet Transform.)

XMill. Section 6.28 is a short description of XMill, a special-purpose compressor for XML files.

Zdelta. A file differencing algorithm developed by Dimitre Trendafilov, Nasir Memon and Torsten Suel. Zdelta adapts the compression library zlib to the problem of differential file compression. zdelta represents the target file by combining copies from both the reference and the already compressed target file. A Huffman encoder is used to further compress this representation. (See also BSdiff, Exediff, File differencing, UNIX diff, and VCDIFF.)

Zero-Probability Problem. When samples of data are read and analyzed in order to generate a statistical model of the data, certain contexts may not appear, leaving entries with zero counts and thus zero probability in the frequency table. Any compression method requires that such entries be somehow assigned nonzero probabilities.

Zip. Popular software that implements the Deflate algorithm (Section 6.25) that uses a variant of LZ77 combined with static Huffman coding. It uses a 32-Kb-long sliding dictionary and a look-ahead buffer of 258 bytes. When a string is not found in the dictionary, its first symbol is emitted as a literal byte. (See also Deflate, Gzip.)

The expression of a man’s face is commonly a help to his thoughts, or glossary on his speech.

—Charles Dickens, *Life and Adventures of Nicholas Nickleby* (1839)
Joining the Data Compression Community

Those interested in a personal touch can join the “DC community” and communicate with researchers and developers in this growing area in person by attending the Data Compression Conference (DCC). It has taken place, mostly in late March, every year since 1991, in Snowbird, Utah, USA, and it lasts three days. Detailed information about the conference, including the organizers and the geographical location, can be found at http://www.cs.brandeis.edu/~dcc/.

In addition to invited presentations and technical sessions, there is a poster session and “Midday Talks” on issues of current interest.

The poster session is the central event of the DCC. Each presenter places a description of recent work (including text, diagrams, photographs, and charts) on a 4-foot-wide by 3-foot-high poster. They then discuss the work with anyone interested, in a relaxed atmosphere, with refreshments served. The Capocelli prize is awarded annually for the best student-authored DCC paper. This is in memory of Renato M. Capocelli.

The program committee reads like a who’s who of data compression, but the two central figures are James Andrew Storer and Michael W. Marcellin.

The conference proceedings are published by the IEEE Computer Society and are distributed prior to the conference; an attractive feature. A complete bibliography (in bibLaTeX format) of papers published in past DCCs can be found at http://liinwww.ira.uka.de/bibliography/Misc/dcc.html.

What greater thing is there for two human souls than to feel that they are joined... to strengthen each other... to be one with each other in silent unspeakable memories.

—George Eliot
Index

The index caters to those who have already read the book and want to locate a familiar item, as well as to those new to the book who are looking for a particular topic. We have included any terms that may occur to a reader interested in any of the topics discussed in the book (even topics that are just mentioned in passing). As a result, even a quick glancing over the index gives the reader an idea of the terms and topics included in the book. Notice that the index items “data compression” and “image compression” have only general subitems such as “logical,” “lossless,” and “bi-level.” No specific compression methods are listed as subitems.

we have attempted to make the index items as complete as possible, including middle names and dates. Any errors and omissions brought to my attention are welcome. They will be added to the errata list and will be included in any future editions.

1-ending codes, 171–172
2-pass compression, 10, 234, 390, 532, 624, 1112, 1122, 1324
3R, see recursive range reduction
7 (as a lucky number), 1057
7z, xiv, 411–415, 1192, 1318
7-Zip, xiv, 411–415, 1303, 1318
8 1/2 (movie), 298
90° rotation, 713

A
A-law companding, 971–976, 987
AAC, see advanced audio coding
AAC-LD, see advanced audio coding
Abish, Walter (1931–), 293
Abousleman, Glen P., 1188
AbS (speech compression), 991
AC coefficient (of a transform), 471, 481, 484
AC-3 compression, see Dolby AC-3
ACB, 290, 1087, 1098–1104, 1303
Acrobat reader (software), 1192
ad hoc text compression, 28–31
Adair, Gilbert (1944–), 293
Adams, Douglas (1952–2001), 398, 509
adaptive (dynamic) Huffman algorithm, 61
adaptive arithmetic coding, 276–279, 453, 829, 1147
adaptive compression, 10, 16
adaptive differential pulse code modulation (ADPCM), 646, 977–979
adaptive frequency encoding, 240, 1319
adaptive Golomb image compression, 633–634
adaptive Huffman coding, 10, 47, 234–240, 245, 393, 399, 658, 1303, 1317, 1319
and video compression, 874
modification of, 1127
adaptive linear prediction and classification (ALPC), x, 547–549, 1304
adaptive wavelet packet (wavelet image decomposition), 796
ADC (analog-to-digital converter), 958
Addison, Joseph (adviser, 1672–1719), 1
additive codes, 59, 157–159
Adler, Mark (1959–), 400
Adobe Acrobat, see portable document format
Adobe Acrobat Capture (software), 1129
Adobe Inc. (and LZW patent), 437
ADPCM, see adaptive differential pulse code modulation
ADPCM audio compression, 977–979, 987
IMA, 646, 977–979
advanced audio coding (AAC), xv, 1055–1081, 1303
and Apple Computer, xv, 1062
low delay, 1077–1079
advanced encryption standard, see also Rijndael
128-bit keys, 395
256-bit keys, 411
advanced television systems committee, 864
advanced video coding (AVC), see H.264
AES, see audio engineering society
affine transformations, 711–714
attractor, 716, 718
affix codes, 196
age (of color), 651
Aladdin systems (stuffit), 1191
Aldus Corp. (and LZW patent), 437
Alexandre, Arsène (1859–1937), 444
algorithmic encoder, 11
algorithmic information content, 1204
Allume systems (stuffit), 1192
ALPC, see adaptive linear prediction and classification
alphabet (definition of), 16, 64, 1303
alphabetic redundancy, 3
Alphabetical Africa (book), 293
ALS, see audio lossless coding
Ambrose, Stephen Edward (1936–2002), 113
America Online (and LZW patent), 437
Amis, Martin (1949–), 246
analog data, 49, 1323
analog video, 855–862
analog-to-digital converter, see ADC
Anderson, Karen, 640
anomalies (in an image), 742
ANSI (American National Standards Institute), 247
anti-Fibonacci numbers, 134, 146
antidictionary methods, x, 430–434, 1309
Apostolico, Alberto, 147
apple audio codec (mismomer), see advanced audio coding (AAC)
Aquinas, Thomas (saint), 194
ARC, 399, 1304
archive (archival software), 399, 1304
ASCII, 43, 339, 434, 1304, 1306, 1326
history of, 60
ASCII85 (binary to text encoding), 1168
Ashland, Matt, xvi
Ashland, Matthew T. (1979–), 1017
Ashmut, Ziya (1960–), 1089
Asimov, Isaac (1920–1992), 641
aspect ratio, 857–860, 864–866
definition of, 844, 858
of HDTV, 844, 864–866
of television, 844, 858
associative coding Buyanovsky, see ACB
asymmetric compression, 11, 330, 336, 588, 650, 907, 1111, 1155
FLAC, 999
in audio, 953
LZJ, 380
ATC (speech compression), 987
atypical image, 580
audio compression, 10, 953–1085
μ-law, 971–976
A-law, 971–976
ADPCM, 646, 977–979
and dictionaries, 953
asymmetric, 953
companding, 967–968
Dolby AC-3, 1082–1085, 1303
DPCM, 644
FLAC, xiv, xvi, 996–1007, 1021, 1312
frequency masking, 964–965, 1032
lossy vs. lossless debate, 1019
LZ, 1318
MLP, 13, 979–984, 1319
monkey’s audio, xv, 1017–1018, 1320
MPEG-1, 11, 1030–1055, 1057
MPEG-2, xv, 1055–1081
silence, 967
temporal masking, 964, 966, 1032
WavPack, xiv, 1007–1017, 1328
audio engineering society (AES), 1031
audio lossless coding (ALS), xv, 1018–1030, 1077, 1304
audio, digital, 958–961
Austen, Jane (1775–1817), 245
author’s email address, xvi
authors’ email address, xi, 23
automaton, see finite-state machines

B
background pixel (white), 557, 563, 1304
Baeyer, Hans Christian von (1938–), 31, 91, 1202
Baker, Brenda, 1176
Baker, Russell (1925–), 62
balanced binary tree, 230, 276
bandpass function, 736
Bao, Derong, xi
Bark (unit of critical band rate), 965, 1304
Barkhausen, Heinrich Georg (1881–1956), 965, 1304
and critical bands, 965
Barnesley, Michael Fielding (1946–), 711
barycentric functions, 630, 806
barycentric weights, 644
BASC, see binary adaptive sequential coding
basis matrix, 630
Baudot code, 28, 29, 463
Baudot, Jean Maurice Emile (1845–1903), 28, 463
Bayes, Thomas (1702–1761), 1308
Bayesian statistics, 287, 1308
Beckett, Samuel (Barclay) (1906–1989), 81
bel (old logarithmic unit), 955
bell curve, see Gaussian distribution
Bell, Quentin (1910–1996), xvi
Bellard, Fabrice, 423, 1311
benchmarks of compression, viii, 16–18, 1307
Bentley, Jon Louis, 128
Berbínzana, Manuel Lorenzo de Lizarraz u, 294
Berger, Toby, 171
Bergmans, Werner (maximum compression benchmark), 17
Bernoulli, Nicolaus I (1687–1759), 151
Bernstein, Dan, 383
beta code, 90, 95, 101
BGMC, see block Gilbert-Moore codes
bi-level dquant (differential quantization), 931
bi-level image, 217, 446, 453, 464, 555, 558, 748, 1304, 1313–1315
bi-level image compression (extended to grayscale), 456, 558, 647
biased Elias Gamma code, 168, 995
Bible, index of (concordance), 1110
bicubic interpolation, 445–446, 631, 939–940
bicubic polynomial, 631
bicubic surface, 631
algebraic representation of, 631
geometric representation of, 632
bidirectional codes, 59, 193–204
big endian (byte order), 1002
bigram (two consecutive bytes), 424
bijection, 96
definition of, 58
bilinear interpolation, 445, 938–939
bilinear prediction, 656
bilinear surface, 445, 938, 939
binary adaptive sequential coding (BASC), ix, 109–110, 1304
binary search, 277, 279, 1200
tree, 339, 340, 348, 413–415, 610, 1132, 1319
binary tree, 230
balanced, 230, 276
complete, 230, 276
traversal, 172
binary tree predictive coding (BTPC), 652–658
BinHex, 43, 1305
binHex4, 43–44, 1208
binomial coefficient, 74
bintrees, 654, 661–668, 707, 1305, 1307
progressive, 662–668
biorthogonal filters, 769
biprefix codes, see affix codes
bisection, 680–682, 1322
bit budget (definition of), 12
bitmap, 36
bitplane, 1305
and Gray code, 458
and pixel correlation, 458
and redundancy, 457
definition of, 446
separation of, 456, 558, 647
bitrate (definition of), 12, 1305
bits/char (bpc), 12, 1305
bits/symbol, 1305
bitstream (definition of), 9
Blelloch, Guy, 217, 1211
blending functions, 629
blind spot (in the eye), 527
block codes (fixed-length), 61
block coding, 1305
block decomposition, 455, 647–651, 831, 1305
block differencing (in video compression), 871
block Gilbert-Moore codes (BGMC), xv, 1018, 1029
block matching (image compression), 579–582, 1305
block mode, 11, 1089
block sorting, 290, 1089
block truncation coding, 603–609, 1305
block-to-variable codes, 58
blocking artifacts in JPEG, 522, 840, 929
Bloom, Charles R., 1319
LZP, 384–391
PPMZ, 309
BMP file compression, 44–45, 1305
and stuffit, 1194
BNF (Backus Naur Form), 351, 1145
Bock, Johan de, 317
UCLC benchmark, 17
BOCU-1 (Unicode compression), 1088, 1166, 1305
Bohr, Niels David (1885–1962), 621
Boldi, Paolo, 122
Boldi–Vigna code, 58, 122–124
Boutell, Thomas (PNG developer), 416
Bowery, Jim (the Hutter prize), 291
bpp (bit per pixel), 13
bpc (bits per character), 12, 1305
BPE (byte pair encoding), 424–427
Braille code, 25–26
Braille, Louis (1809–1852), 25, 57
Brandenburg, Karlheinz (mp3 developer, 1954–), 1081
break even point in LZ, 348
Brislaw, Christopher M., 280, 835
Broukhis, Leonid A., 1098
Calgary challenge, 14, 15
browsers (Web), see Web browsers
Bryant, David (WavPack), xi, xiv, xv, 1007
BSDiff, 1178–1180, 1306
bspatch, 1178–1180, 1306
BTC, see block truncation coding
BTPC (binary tree predictive coding), 652–658
Buffoni, Fabio, 317
Burmann, Gottlob (1737–1805), 294
Burrows, Michael (BWT developer), 1089
Burrows-Wheeler method, 11, 112, 290, 411, 1087, 1089–1094, 1105, 1306
and ACB, 1100
stuffit, 1192, 1193
Busch, Stephan (1979–), 18
Buyanovsky, George (Georgii)
Mechislavovich, 1098, 1104, 1303
BWT, see Burrows-Wheeler method
byte coding, 92

C
C1 prefix code, 113
C2 prefix code, 113
C3 prefix code, 114
C4 prefix code, 114
identical to omega code, 114
CABAC (context-based arithmetic coding), 913
cabinet (Microsoft media format), 352
Calgary challenge, 14–15
Calgary Corpus, 13, 309, 312, 327, 344, 345, 347, 517, 1147
CALIC, 584, 636–640, 1306
and EIDAC, 578
canonical Huffman codes, 228–232, 404
Canterbury Corpus, 14, 517, 1126
Capocelli prize, 1329
Capocelli, Renato Maria (1940–1992), 139, 149, 171, 1329
Capon model for bi-level images, 633
Carr, James, 1085
Carroll, Lewis (1832–1898), 20, 364, 530
Cartesian product, 631
cartoon-like image, 447, 710
cascaded compression, 10
case flattening, 27
Castaneda, Carlos (Cesar Arana, 1925–1998), 351
Castle, William (Schloss, 1914–1977), 343
causal wavelet filters, 772
CAVLC (context-adaptive variable-length code), 532, 913, 914, 920
CCITT, 249, 435, 521, 1306, 1315, 1316
CDC display code, 28
cell encoding (image compression), 729–730, 1306
CELP (speech compression), 907, 991
centered phased-in codes, 84, 175
Chaitin, Gregory J. (1947–), 58, 1204
channel coding, 64, 177
check bits, 178–179
Chekhov, Anton Pavlovich (1860–1904), 506
chirp (test signal), 748
chm, see compiled html
Chomsky, Avram Noam (1928–), 1145
Christie Mallowan, Dame Agatha Mary Clarissa (Miller 1890–1976)), 240
chromaticity diagram, 523
chrominance, 760
CIE, 523, 1306
color diagram, 523
circular queue, 337, 389, 423, 1306
clairvoyant context, 1132
Clancy, Thomas Leo (1947–), 839
Clarke, Arthur Charles (1917–2008), 176, 1078
Clausius, Rudolph Julius Emmanuel (1822–1888), 1202
Cleary, John G., 292
cloning (in DMC), 1137
Coalson, Josh, xiv, xvi, 996, 1007
code overflow (in adaptive Huffman), 238
codec, 9, 1306
codes
ASCII, 1306
Baudot, 28, 463
biased Elias Gamma, 995
CDC, 28
definition of, 60, 1306
EBCDIC, 1306
Elias delta, 102–105
Elias Gamma, 47, 101–102, 107, 995
Elias omega, 105–107
ending with 1, 171–172
error-correcting, 1311
Golomb, 614, 617, 914, 994, 1110
Hamming distance, 180–182
interpolative, ix, 172–176, 1315
parity bits, 179–180
phased-in binary, 235
pod, xiv, 995
prefix, 42, 349, 393, 453, 1131
and video compression, 874
Rice, 52, 313, 617, 994, 997, 1001–1002, 1011–1012, 1186, 1304, 1312, 1323, 1324
subexponential, 617, 994
Tunstall, 1326
unary, 359, 390
Unicode, 1306
variable-length, 220, 234, 239, 241, 245, 247, 250, 264, 329, 331, 393, 1211, 1323
unambiguous, 69, 1316
coefficients of filters, 775–776
Cohn, Martin, 1271
and LZ78 patent, 439
collating sequence, 339
color
age of, 651
blindness, 526, 527
cool, 528
in our mind, 527
not an attribute, 526
space, 526, 527
warm, 528
color images (and grayscale compression), 38, 463, 620, 637, 641, 647, 723, 750, 760
color space, 523
comma codes, 62
commandments of data compression, 21
Commission Internationale de l’Eclairage, see CIE
compact disc (and audio compression), 1055
compact support
  definition of, 781
  of a function, 750
compaction, 26
companding, 9, 967
  ALS, 1023
    audio compression, 967–968
compiled html (chm), 353
complete binary tree, 230, 276
complete codes, 63, 147
complex methods (diminishing returns), 6, 360, 376, 554
complexity (Kolmogorov-Chaitin), 1204
composite values for progressive images, 662–666, 1307
compressed stream (definition of), 16
compression benchmarks, viii, 16–18, 1307
compression challenge (Calgary), 14–15
compression factor, 12, 764, 1307
compression gain, 13, 1307
compression performance measures, 12–13
compression ratio, 12, 1307
  in UNIX, 375
    known in advance, 28, 49, 448, 466, 589, 605, 829, 846, 870, 968, 971, 978, 1031
compression speed, 13
compressor (definition of), 9
Compuserve Information Services, 394, 437, 1313
computer arithmetic, 522
concordance, 1110
  definition of, 194
conditional exchange (in QM-coder), 284–287
conditional image RLE, 40–42, 1307
conditional probability, 1308
context
  clairvoyant, 1132
  definition of, 1308
  from other bitplanes, 578
  inter, 578
  intra, 578
order-1 in LZPP, 347
symmetric, 611, 621, 636, 638, 848
two-part, 578
context modeling, 292
  adaptive, 293
  order-N, 294
  static, 292
context-adaptive variable-length codes, see CAVLC
context-based arithmetic coding, see CABAC
context-based image compression, 609–612, 636–640
context-free grammars, 1088, 1145, 1308
context-tree weighting, 320–327, 348, 1308
  for images, 456, 646–647, 1184
contextual redundancy, 3
continuous wavelet transform (CWT), 528, 743–749, 1308
continuous-tone image, 446, 447, 517, 652, 748, 796, 1308, 1313, 1316
convolution, 767, 1309
  and filter banks, 768
cool colors, 528
Cormack, Gordon V., 1134
correlation, 1309
correlations, 452
  between phrases, 440
  between pixels, 217, 467, 611
  between planes in hyperspectral data, 1183
  between quantities, 451
  between states, 1137–1139
  between words, 1124
Costello, Adam M., 417
covariance, 452, 479
  and decorrelation, 479
CRC, 434–436, 1309
  CRC-32, 347
  in CELP, 991
  in MPEG audio, 1038, 1044, 1053
  in PNG, 417
crew (image compression), 827
  in JPEG 2000, 843
Crichton, John Michael (1942–2008), 439
Crick, Francis Harry Compton (1916–2004), 301
cross correlation of points, 468
CRT, 856–860, 1309
cryptography, 177
CS-CELP (speech compression), 991
CTW, see context-tree weighting
Culik, Karel II, 695, 696, 701
cumulative frequencies, 271, 276
curiosities of data compression, viii, 19–20
cycles
Hilbert, 683–687
Peano, 684, 694
Sierpiński, 683, 688
space-filling, 687–694
CWT, see continuous wavelet transform
cycles per byte (CPB, compression speed), 13
cyclical redundancy check, see CRC

D
DAC (digital-to-analog converter), 958
Darwin, Charles Robert (1809–1882), 27
data compression
a special case of file differencing, 11
adaptive, 10
anagram of, 384
and irrelevancy, 448
and redundancy, 3, 448, 1136
as the opposite of reliability, 25
asymmetric, 11, 330, 336, 380, 588, 650,
907, 953, 1111, 1155
audio, 953–1085
benchmarks, viii, 16–18, 1307
block mode, 11, 1089
block sorting, 1089
change of paradigm, 314, 319
commandments of, 21
compaction, 26
conference, 1271, 1309, 1329
curiosities, viii, 19–20
definition of, 2
dictionary-based methods, 9, 290,
329–441, 450
diminishing returns, 6, 21, 360, 376, 554,
823
general law, 3, 439
golden age of, 319, 344
history in Japan, 22, 399
hyperspectral, 1180–1191
image compression, 443–730
intuitive methods, 25–31, 466
irreversible, 26–28
joining the community, 1271, 1329
logical, 12, 329
lossless, 10, 28, 1317
lossy, 10, 1317
model, 15, 553, 1320
nonadaptive, 9
offline dictionary-based methods, 424–429
optimal, 11, 349
packing, 28
patents, 410, 437–439, 1321
performance measures, 12–13
physical, 12
pointers, 21
progressive, 557
reasons for, 2
semiadaptive, 10, 234, 1324
small numbers, 46, 531, 535, 554, 641,
647, 651, 652, 1092, 1267
statistical methods, 9, 211–327, 330,
449–450
streaming mode, 11, 1089
syllables, x, 1125–1127, 1325
symmetric, 10, 330, 351, 522, 835
two-pass, 10, 234, 265, 390, 532, 624,
1112, 1122, 1324
universal, 11, 349
vector quantization, 466, 550
video, 869–952
who is who, 1329
data compression (source coding), 61
data reliability, 184–193
data structures, 21, 238, 239, 337, 340, 356,
369, 370, 1306, 1309
arrays, 1309
graphs, 1309
hashing, 1309
lists, 1309
queues, 337, 1309
stacks, 1309
trees, 1309
Day, Clarence Shepard (1874–1935), 1205
DC coefficient (of a transform), 471, 481,
484, 485, 514, 523, 530–532, 535
DCC, see data compression, conference
DCT, see discrete cosine transform
De Santis, Alfredo, 171
De Vries, Peter (1910–1993), v
decibel (dB), 464, 955, 1070, 1309
decimation (in filter banks), 768, 1035
decoder, 1309
   definition of, 9
deterministic, 11
   recursive, 585
decoding, 60
decompressor (definition of), 9
decorrelated pixels, 451, 454, 467, 475, 479,
   514, 515
decorrelated values (and covariance), 452
definition of data compression, 2
Deflate, 337, 399–411, 1168, 1309, 1313,
   1328
   and RAR, xiv, 396
delta code (Elias), 102–105, 122
   and search trees, 130
deque (data structure), 195
dequeu, see deque
Destasio, Berto, 317
   EmilCont benchmark, 18
deterministic decoder, 11
Deutsch, Peter, 405, 406
DFT, see discrete Fourier transform
Dickens, Charles (1812–1870), 610, 1328
dictionary (in adaptive VQ), 598
dictionary-based compression, 1126–1127
dictionary-based methods, 9, 290, 329–441,
   450, 1309, see also antidictionary
   methods
   and sequitur, 1149
   and sound, 966
   compared to audio compression, 953
dictionaryless, 391–394
   offline, 424–429
two-dimensional, x, 582–587, 1313
   unification with statistical methods,
   439–441
DIET, 423
diff, 1170, 1327
difference values for progressive images,
   662–666, 1307
differencing, 35, 374, 535, 646, 1323
   file, xv, 1088, 1169–1180, 1312
   in BOCU, 1166
   in video compression, 870
differentiable functions, 789
differential encoding, 36, 641, 1311
differential image compression, 640–641,
   1310
differential pulse code modulation, 36,
   641–646, 1311
   adaptive, 646
   and sound, 641, 1311
differential quantization (dquant), 931
digital audio, 958–961
digital image, see image
digital silence, 1001
digital television (DTV), 866
digital video, 863–864, 1310
digital video interactive (DVI), 867
digital-to-analog converter, see DAC
digitally sampled analog data, 588, 1327
digram, 4, 35, 292, 293, 1146, 1310
   and redundancy, 3
   encoding, 35, 1146
   frequencies, 246
discrete cosine transform, 475, 480–514, 522,
   528–529, 758, 913, 918, 1310
   and speech compression, 987
   in H.261, 907
   in MPEG, 883–891
   integer, 943–949
   modified, 1035
   mp3, 1049
   three dimensional, 480, 1186–1188
discrete Fourier transform, 528
   in MPEG audio, 1032
discrete sine transform, 513–515
discrete wavelet transform (DWT), 528,
   777–789, 1310
discrete-tone image, 447, 517, 519, 520, 647,
   652, 710, 748, 1305, 1310
   distortion measures
   in vector quantization, 589
   in video compression, 872–873
distributions
   energy of, 468
   flat, 966
   Gaussian, 1312
   geometric, 543, 1011
   Laplace, 454, 619, 622–626, 635, 636, 647,
   960–961, 984, 994, 995, 1002, 1317,
   1319, 1321
normal, 666, 1320
Poission, 301, 302
dithering, 564, 668
DjVu document compression, 831–834, 1310
DMC, see dynamic Markov coding
Dobie, J. Frank (1888–1964), 1301
Dolby AC-3, xv, 1082–1085, 1303
Dolby Digital, see Dolby AC-3
Dolby, Ray Milton (1933–), 1082
Dolby, Thomas (Thomas Morgan Robertson, 1958–), 1085
downsampling, 521, 562
Doyle, Arthur Conan (1859–1930), 326
DPCM, see differential pulse code modulation
dquant (differential quantization), 931
drawing
and JBIG, 557
and sparse strings, 1110
DSAD, see digitally sampled analog data
DST, see discrete sine transform
dual tree coding, 74, 219–220
Dumpty, Humpty, 1125
duodecimal (number system), 142
Durbin J., 1006
DWT, see discrete wavelet transform
dynamic compression, see adaptive compression
dynamic dictionary, 329
GIF, 394
dynamic Markov coding, 1088, 1134–1142
dynamically-variable-flag-length (DVFL), 139
Dyson, George Bernard (1953–), 1134

E
ear (human), 961–966
Eastman, Willard L. (and LZ78 patent), 439
EBCDIC, 339, 1306
EBCOT (in JPEG 2000), 842, 843
edgebreaker, 1088, 1150–1161, 1326
Edison, Thomas Alva (1847–1931), 855, 858
EIDAC, simple image compression, 577–579, 1324
Eigen, Manfred (1927–), 265
eigenvalues of a matrix, 480
Einstein, Albert (1879–1955)
and $E = mc^2$, 31
El-Sakka, Mahmoud R., xi, 587
Elias codes, 58, 101–107
Elias delta code, 102–105
Elias Gamma code, 101–102, 107, 995
in WavPack, 1012, 1013
Elias gamma code, 47
Elias omega code, 105–107
Elias, Peter (1923–2001), 90, 101, 102, 264
Eliot, George (1819–1880), 1329
Ellsworth, Annie (and Morse telegraph), 55
Elton, John, 715
email address of author, xvi
email address of authors, xi, 23
embedded coding in image compression, 815–816, 1311
embedded coding using zerotrees (EZW), 827–831, 1312
Emilcont (a PAQ6 derivative), 319
encoder, 1311
algorithmic, 11
definition of, 9
encoding, 60
energy
concentration of, 471, 475, 478, 480, 514, 515
of a distribution, 468
of a function, 743–745
English text, 3, 15, 330
frequencies of letters, 4
entropy, 63–68, 70, 212, 213, 217, 264, 275, 558, 1210
definition of, 1202, 1311
of an image, 454, 652
physical, 1205
entropy encoder, 774
definition of, 1203
dictionary-based, 329
error metrics in image compression, 463–465
error-control codes, 177–183
error-correcting (VLEC) codes, 204–208
error-correcting codes, 177–183, 1311
in RAR, 395–396
error-detecting codes, 177–183, 434
ESARTUNILOC (letter probabilities), 4
ESC, see extended synchronizing codeword escape code
in adaptive Huffman, 234
in Macwrite, 31
in pattern substitution, 35
in PPM, 297
in RLE, 31
in textual image, 1131
in word-based compression, 1122
ETAOINSHRDLU (letter probabilities), 4
Euler’s equation, 1155
Euler, Leonhard (1707–1783), 151, 152
even functions, 514
Even, Shimon (1935–2004), 111
Even–Rodeh code, 58, 111
a special case of Stout $R$ code, 120
LZR, 338
exclusion
in ACB, 1103
in LZPP, 346
in PPM, 300–301
in symbol ranking, 1095
exclusive OR (XOR), 200
EXE compression, 423
EXE compressors, 423, 1175–1178, 1311
exediff, 1175–1178
exepatch, 1175–1178
exhaustive search VQ, 594–596
exponential Golomb codes, 98, 164, 198–200
extended synchronizing codeword (ESC), 192–193
eye
and brightness sensitivity, 453
and perception of edges, 449
and spatial integration, 859, 881
frequency response of, 886
resolution of, 527
EZW, see embedded coding using zerotrees

F
FABD, see block decomposition
Fabre, Jean Henri (1823–1915), 515
facsimile compression, 248–254, 448, 453, 570, 1129, 1312
1D, 249
2D, 253
group 3, 249
factor of compression, 12, 764, 1307
Fano, Robert Mario (1917–), 211, 212
fast PPM, 312–313
FBI fingerprint compression standard, 834–840, 1328
feasible codes, 171–172
as synchronous codes, 189
FELICS, 612–614, 1312
and phased-in codes, 84
progressive, 615–617, 619, 620
Fenwick, Peter M., 112, 152, 154, 155, 1089
Feynman, Richard Phillips (1918–1988), 959
FHM compression, 1088, 1142–1144, 1312
Fiala, Edward R., 6, 358, 439, 1318
Fibonacci code, 59, 143–146
generalized, 147–150
Fibonacci numbers, 142, 1312
and FHM compression, 1142–1144, 1268
and height of Huffman trees, 227
and sparse strings, 1115–1116
and taboo codes, 131, 134
Fibonacci, Leonardo Pisano (1170–1250), 142, 151
file differencing, xv, 11, 1088, 1169–1180, 1306, 1312, 1327, 1328
BSDiff, 1178–1180
bspatch, 1178–1180
exediff, 1175–1178
exepatch, 1175–1178
VCDIFF, 1171–1173, 1327
zdelta, 1173–1175
filter banks, 767–776
biorthogonal, 769
decimation, 768, 1035
deriving filter coefficients, 775–776
orthogonal, 769
filters
causal, 772
deriving coefficients, 775–776
finite impulse response, 768, 777
taps, 772
fingerprint compression, 789, 834–840, 1328
finite automata methods, 695–711
finite impulse response filters (FIR), 768, 777
finite-state machines, 218, 695, 1088
and compression, 633, 1134–1135
Fisher, Yuval, 711
fixed-length codes, 3
fixed-to-variable codes, 72
FLAC, see free lossless audio compression
flag codes
Wang, 59, 135–137
Yamamoto, 59, 137–141
Index  

Flate (a variant of Deflate), 1168
Forbes, Jonathan (LZX), 352
forbidden words, see antidictionary
Ford, Paul, 1123
foreground pixel (black), 557, 563, 1304
Fourier series, 736
Fourier transform, 467, 483, 528, 732–741, 758, 770, 772, 1310, 1312, 1326
and image compression, 740–741
and the uncertainty principle, 737–740
frequency domain, 734–736
Fourier, Jean Baptiste Joseph (1768–1830), 731, 735, 1312
Fowler, Thomas (1777–1843), 117
fractal image compression, 711–725, 1315
fractals (as nondifferentiable functions), 789
fractional pixels, 938–940
Fraenkel, Aviezri Siegmund (1929–), 145, 147
Frank, Amalie J., 555
free distance (in VLEC codes), 183–184
free lossless audio compression (FLAC), xiv, xvi, 961, 996–1007, 1021, 1312
Freed, Robert A., 399, 1304
Freeman, George H., 219
French (letter frequencies), 4
frequencies
  cumulative, 271, 276
  of digrams, 246
  of pixels, 445, 472, 558
  of symbols, 15, 220, 234–236, 239, 241, 242, 264–266, 558, 1320
  in LZ77, 336
frequency domain, 734–736, 964
frequency masking, 964–965, 1032
frequency of eof, 268
fricative sounds, 986
front compression, 30
full (wavelet image decomposition), 795
functions
  bandpass, 736
  barycentric, 630, 806
  blending, 629
  compact support, 750, 781
differentiable, 789
energy of, 743–745
even, 514
frequency domain, 734–736
nondifferentiable, 789
nowhere differentiable, 789
odd, 514
parity, 514
plotting of (a wavelet application), 779–781
square integrable, 743
support of, 750
fundamental frequency (of a function), 734

G

Gadsby (book), 293
Gailly, Jean-Loup, 400
gain of compression, 13, 1307
Gallager, Robert Gray (1931–), 165, 214, 232
gamma code (Elias), 47, 101–102, 107, 122, 168, 342
and Goldbach G1, 153
and punctured codes, 112
and search trees, 129
as a start-step-stop code, 98
biased, 168
its length, 155
gapless source, 188
gasket (Sierpinski), 716
Gauss, Karl Friedrich (1777–1855), 27
Gaussian distribution, 1312, 1317, 1320
general unary codes, 97–100
generalized exponential Golomb codes, 199
generalized Fibonacci codes, 147–150
generalized Fibonacci numbers, 134
generalized finite automata, 707–711, 1312
generating polynomials, 1311
CRC, 436, 1038
CRC-32, 436
geometric compression, 1088
geometric distribution, 161, 543, 1011
geometric sequence, 161
geometric series, 161
GFA, see generalized finite automata
Ghengis Khan (≈1165–1207), 55
GIF, 394–395, 1313
and DjVu, 832
and image compression, 450
and LZW patent, 437–439, 1321
and stuffit, 1194
and web browsers, 395
compared to FABD, 647
Gilbert, Jeffrey M., 648
Gilchrist, Jeff (ACT benchmark), 18
Givens rotations, 499–508
Givens, James Wallace Jr. (1910–1993), 508
Goethe, Johann Wolfgang von (1743–1832), 119
Goldbach codes, 59, 142, 151–157
and unary codes, 152
Goldbach conjecture, 151–153, 157
Goldbach, Christian (1690–1764), 151, 152
golden age of data compression, 319, 344
golden ratio, 142, 143, 1261, 1312
Golomb code, 59, 160–166, 170, 197, 614, 617, 994, 1110, 1312, 1313, 1323
and JPEG-LS, 164, 541, 543–544, 1316
and RLE, 165, 166
exponential, 198–200
H.264, 914, 920
WavPack, 1011
Golomb, Solomon Wolf (1932–), 166
Golomb–Rice code, see Rice codes
Gordon, Charles, 68
Gouraud shading (for polygonal surfaces), 1150
Graham, Heather, 967
grammar-based compression, 424
grammars, context-free, 1088, 1145, 1308
graphical image, 447
graphical user interface, see GUI
graphs (data structure), 1309
Gray code, see reflected Gray code
Gray, Frank, 463
grayscale image, 446, 453, 464, 1313, 1316, 1319, 1321
unused byte values, 425
grayscale image compression (extended to color images), 38, 463, 620, 637, 641, 647, 723, 750, 760
Greene, Daniel H., 6, 358, 439, 1318
group 3 fax compression, 249, 1312
PDF, 1168
group 4 fax compression, 249, 1312
PDF, 1168
growth geometry coding, 555–557, 1313
GS-2D-LZ, x, 582–587, 1313
GUI, 447
Guidon, Yann, xiv, xvi, 51, 54
Gzip, 375, 399, 400, 438, 1309, 1313
H
H.261 video compression, 907–909, 1313
DCT in, 907
H.263, 868
H.264 video compression, xiv, 532, 868, 910–926, 1313
compared with others, 933–935
scalable video coding, x, 922–926, 1325
Haar transform, 475, 477–478, 510, 749–767
Haar, Alfred (1885–1933), 749
Hafner, Ulrich, 707
haiku (poetic form), 1126
halftoning, 559, 562–564, 1314
and fax compression, 251
Hamming codes, 1314
Hamming distance, 178, 180–182
Hamming, Richard Wesley (1915–1998), 180
Hardy, Thomas (1840–1928), 279
harmonics (in audio), 1025–1026
Haro, Fernando Jacinto de Zurita y, 294
Hartley (information unit), 64
hashing, 370, 372, 384, 582, 610, 651, 1132, 1309
HD photo, x, 539–541, 1313
HDTV, 844, 944, 1082
and MPEG-3, 880, 1057, 1320
aspect ratio of, 844, 864–866
resolution of, 864–866
standards used in, 864–866, 1314
heap (data structure), 230
hearing
properties of, 961–966
range of, 741
Heisenberg uncertainty principle, 737
Heisenberg, Werner Karl (1901–1976), 737
Herd, Bernd (LZH), 335, 343
Herrera, Alonso Alcala y (1599–1682), 294
hertz (Hz), 732, 954
hiding data (how to), viii, 18–19
hierarchical coding (in progressive compression), 550, 811–814
hierarchical image compression, 523, 535, 1314
high-definition television, see HDTV
Hilbert curve, 661, 683–688
and vector quantization, 684–687
traversing of, 691–694
Index

Hilbert, David (1862–1943), 687
history of data compression in Japan, 22, 399
homeomorphism, 1151
homogeneous coordinates, 714
Horspool, R. Nigel, 1134
Hotelling transform, see Karhunen-Loève transform
Householder, Alston Scott (1904–1993), 508
how to hide data, viii, 18–19
HTML (as semistructured text), 1149, 1324
Huffman algorithm, 61, 68, 161
combined with Tunstall, 219–220
Huffman codes, 58, 73, 96
and Rice codes, 167
resynchronizing, 190–193
Huffman coding, 214–233, 250, 251, 253, 264, 292, 331, 343, 423, 448, 1091, 1221, 1309, 1313, 1314, 1324, 1325, 1328
adaptive, 10, 47, 234–240, 393, 399, 658, 1127, 1303, 1317
and video compression, 874
word-based, 1122
and Deflate, 400
and FABD, 651
and move-to-front, 46, 47
and MPEG, 886
and reliability, 247
and sound, 966
and sparse strings, 1113, 1114
and wavelets, 752, 760
canonical, 228–232, 404
code size, 223–224
dead?, 232–233
decoding, 220–223
for images, 218
in image compression, 640
in JPEG, 522, 530, 840
in LzP, 390
in WSQ, 835
not unique, 215
number of codes, 224–225
semiaadaptive, 234
synchronized?, 233
ternary, 225–227
2-symbol alphabet, 217
unique tree, 404
variance, 216
Huffman, David Albert (1925–1999), 214
HuffSyllable (HS), 1127
human hearing, 741, 961–966
human visual system, 463, 526–528, 605, 607
human voice (range of), 961
Humpty Dumpty, see Dumpty, Humpty
Hutter prize for text compression, ix, 290–291, 319, 1314
Hutter, Marcus (the Hutter prize), 291
hybrid speech codecs, 986, 991
hyperspectral data, 1180, 1314
hyperspectral data compression, 1088, 1180–1191
hyperthreading, 412
I
ICE, 399
IDCT, see inverse discrete cosine transform
IEC, 248, 880
IFS compression, 20, 711–725, 1315
PIFS, 720
IGS, see improved grayscale quantization
IIID, see memoryless source
IMA, see interactive multimedia association
IMA ADPCM compression, 646, 977–979
image, 443
atypical, 580
bi-level, 217, 446, 464, 555, 558, 748, 1304, 1313–1315
bitplane, 1305
cartoon-like, 447, 710
continuous-tone, 446, 447, 517, 652, 748, 796, 1308, 1316
definition of, 443
discrete tone, 519
discrete-tone, 447, 517, 520, 647, 652, 710, 748, 1305, 1310
graphical, 447
greyscale, 446, 453, 464, 1313, 1316, 1319, 1321
interlaced, 38
reconstruction, 740
resolution of, 443
simple, 577–579, 1324
synthetic, 447
types of, 446–447
image compression, 9, 10, 31, 40, 443–730
bi-level, 557
bi-level (extended to grayscale), 456, 558, 647
dictionary-based methods, 450
differential, 1310
error metrics, 463–465
intuitive methods, 466
lossy, 448
principle of, 36, 451, 453–457, 579, 609, 658, 683
and RGC, 456
progressive, 456, 549–557, 1322
growth geometry coding, 555–557
median, 554
reasons for, 447
RLE, 448
self-similarity, 456, 1312, 1328
statistical methods, 449–450
subsampling, 466
image frequencies, 445, 472
image transforms, 467–515, 684, 685, 755–758, 1326
image wavelet decompositions, 791–798
images (standard), 517–519
improper rotations, 495
improved grayscale quantization (IGS), 449
Indeo, see digital video interactive (DVI)
negativity (Kraft–MacMillan), 1316
and Huffman codes, 217
information theory, 63–68, 332, 1199–1205, 1314
algorithmic, 58
and increased redundancy, 178
basic theorem, 179
inorder traversal of a binary tree, 172
instantaneous codes, 62
integer orthogonal transforms, 481, 943–949
integer wavelet transform (IWT), 809–811
Interactive Multimedia Association (IMA), 978
interlacing scan lines, 38, 865, 930
International Committee on Illumination, see CIE
interpolating polynomials, 621, 626–633, 805–808, 813, 1314
degree-5, 807
Lagrange, 993
interpolation of pixels, 445–446, 938–940
interpolative coding, ix, 110, 172–176, 1315
interval inversion (in QM-coder), 284–285
intuitive compression, 25
intuitive methods for image compression, 466
inverse discrete cosine transform, 480–514, 528–529, 1236
in MPEG, 885–902
mismatch, 886, 902
modified, 1035
inverse discrete sine transform, 513–515
inverse modified DCT, 1035
inverse Walsh-Hadamard transform, 475–477
inverted file, 1110
inverted index, 108, 175
irrelevancy (and lossy compression), 448
irreversible data compression, 27–28
irreversible text compression, 26
Ismael, G. Mohamed, 333
ISO, 247, 521, 880, 1315, 1316, 1320
JBIG2, 567
recommendation CD 14495, 541, 1316
standard 15444, JPEG2000, 843
iterated function systems, 711–725, 1315
ITU, 248, 1306, 1312, 1315
ITU-R, 525
recommendation BT.601, 525, 862
ITU-T, 248, 558
and fax training documents, 250, 517, 580
and MPEG, 880, 1320
JBIG2, 567
recommendation H.261, 907–909, 1313
recommendation H.264, 910, 1313
recommendation T.4, 249, 251
recommendation T.6, 249, 251, 1312
recommendation T.82, 558
V.42bis, 398, 1327
IWT, see integer wavelet transform

J
JBIG, 557–567, 831, 1315, 1322
and EIDAC, 578
and FABD, 647
probability model, 559
JBIG2, 280, 567–577, 1168, 1315
and DjVu, 832, 1310
Jessel, George (1824–1883), 100
JFIF, 538–539, 1316
joining the DC community, 1271, 1329
Index

Joyce, James Aloysius Augustine (1882–1941), 23, 27
Joyce, William Brooke (1906–1946), xi
JPEG, 41, 280, 287, 448, 471, 475, 520–538, 541, 577, 646, 831, 886, 887, 1187, 1305, 1315, 1316, 1322
and DjVu, 832
and progressive image compression, 550
and sparse strings, 1110
and spatial prediction, 725
and WSQ, 840
blocking artifacts, 522, 840, 929
compared to H.264, 922
lossless mode, 535
similar to MPEG, 883
zigzag order, 118
JPEG 2000, 280, 525, 732, 840–853, 1168, 1194
JPEG XR, x, 539–541, 1316
JPEG-LS, 535, 541–547, 843, 1184, 1316
and Golomb code, 164
and stuffit, 1194
Jung, Robert K., 399, 1304

K
Karhunen-Loève transform, 475, 478–480, 512, 758, see also Hotelling transform
Kari, Jarkko J. (1964–), 695
Katz, Philip Walter (1962–2000), 399, 400, 408, 410, 1321
KGB (a PAQ6 derivative), 319
King, Stephen Edwin (writer, 1947–), 1167
Klaasen, Donna, 1101
Klein, Shmuel Tomi, 145
KLT, see Karhunen-Loève transform
Knuth, Donald Ervin (1938–), xiii, 21, 76, 107, 259, (Colophon)
Kolmogorov-Chaitin complexity, 1203
Kraft–McMillan inequality, 63, 69–71, 1316
and Huffman codes, 217
Kraus, Karl (1874–1936), 1269
Kronecker delta function, 509
Kronecker, Leopold (1823–1891), 510
KT boundary (and dinosaur extinction), 321
KT probability estimate, 321, 1317
Kublai Khan (1215–1294), 55
Kulawiec, Rich, 176
Kurtz, Mary, 441
KWIC (key word In context), 194–195

L
L Systems, 1145
Hilbert curve, 684
La Disparition (novel), 293
Lagrange interpolation, 993, 1004
Lang, Andrew (1844–1912), 229
Langdon, Glen G., 440, 609
Laplace distribution, 168, 454, 619, 622–626, 635, 636, 647, 994, 995, 1317, 1319, 1321
in audio MLP, 984
in FLAC, 1002
in image MLP, 619
of differences, 960–961
Laplacian pyramid, 732, 792, 811–814, 1317
Lau, Daniel Leo, 693
Lau, Raymond (stuffit originator, 1971–), xi, 1191
lazy wavelet transform, 799
LBE, see location based encoding
LBG algorithm for vector quantization, 549, 589–594, 598, 685, 1191
Leibniz, Gottfried Wilhelm (1646–1716), 27, 151
Lempel, Abraham (1936–), 51, 331, 1318
LZ78 patent, 439
Lempereur, Yves, 43, 1305
Lena (image), 468, 517–520, 569, 764, 1255, see also Soderberg
blurred, 1257
Les Revenentes (novelette), 293
Levenshtein, Vladimir Iosifovich (1935–), 110
Levenstein, Aaron (1911–1986), 327
Levinson, Norman (1912–1975), 1006
Levinson-Durbin algorithm, 1001, 1006, 1021, 1022, (Colophon)
lexicographic order, 339, 1090
LHA, 399, 1317
LHARc, 399, 1317
Liebchen, Tilman (1971–), 1019
LIFO, 374
lifting scheme, 798–808, 1317
Ligtenberg, Adriaan (1955–), 504
Linder, Doug, 195
Index

line (as a space-filling curve), 1247
line (wavelet filling decomposition), 792
linear prediction
  4th order, 1004–1005
  ADPCM, 977
  ALS, 1020–1022
  FLAC, 1001
  MLP audio, 983
  shorten, 992
linear predictive coding (LPC), xv, 1001, 1005–1007, 1018, 1304
  hyperspectral data, 1184–1186
linear systems, 1309
lipogram, 293
list (data structure), 1309
little endian (byte order), 412
  in Wave format, 969
Littlewood, John Edensor (1885–1977), 96
LLM DCT algorithm, 504–506
location based encoding (LBE), 117–119
lockstep, 234, 281, 369
LOCO-I (predecessor of JPEG-LS), 541
Loeffler, Christoph, 504
log-star function, 128
logarithm
  as the information function, 65, 1200
  used in metrics, 13, 464
logarithmic ramp representations, 58
logarithmic tree (in wavelet decomposition), 770
logarithmic-ramp code, see omega code (Elias)
logical compression, 12, 329
lossless compression, 10, 28, 1317
lossy compression, 10, 1317
Lovato, Darryl (stuffit developer 1966–), xi, 1198
LPC, see linear predictive coding
LPC (speech compression), 988–991
Luhn, Hans Peter (1896–1964), 194
luminance component of color, 452, 455, 521–526, 529, 531, 760, 881, 935
  use in PSNR, 464
LZ compression, see dictionary-based methods
LZ1, see LZ77
LZ2, see LZ78
LZ76, 338
LZ77, 334–337, 361, 365, 579, 1099, 1101, 1149, 1305, 1309, 1313, 1318, 1319, 1324, 1328
  and Deflate, 400
  and LZR4, 364
  and repetition times, 349
  deficiencies, 347
LZ8, 347, 354–357, 365, 1101, 1318, 1319
  patented, 439
LZAP, 378, 1318
LZARI, xiv, 343, 1318
LZB, ix, 341–342, 1318
LZC, 362, 375
LZEXE, 423, 1311
LZFG, 358–360, 362, 1318
  patented, 360, 439
LZH, 335
  confusion with SLH, 343
LZJ, x, 380–382, 1318
LZJ’, 382
LZMA, xiv, xvi, 280, 411–415, 1318
LZMW, 377–378, 1319
LZP, 384–391, 441, 581, 1095, 1096, 1319
LZPP, ix, 344–347, 1319
LZR, ix, 338, 1319
LZRW1, 361–364
LZRW4, 364, 413
LZSS, xiv, 339–347, 399, 1317–1319
  used in RAR, xiv, 396
LZT, ix, 376–377, 1319
LZW, 365–375, 1030, 1267, 1319, 1319, 1327
  decoding, 366–369
  patented, 365, 437–439, 1321
  UNIX, 375
  word-based, 1123
LZW algorithm (enhanced by recursive phased-in codes, 89
LZWL, 1126–1127, 1319
LZX, 352–354, 1319
LZY, 383–384, 1319
M
m4a, see advanced audio coding
m4v, see advanced audio coding
Mackay, Alan Lindsay (1926–), 11
MacWrite (data compression in), 31
Mahler’s third symphony, 1056
Mahoney, Matthew V. (1955–), xi
  and PAQ, 314, 316–318
Index

Hutter prize, 291
Mallat, Stephane (and multiresolution decomposition), 790
Malvar, Henrique “Rico” (1957–), 910
Manber, Udi, 1176
mandril (image), 517, 796
Marcellin, Michael W. (1959–), 1329
Markov model, 41, 245, 290, 456, 559, 1143, 1240, 1268
masked Lempel-Ziv tool (a variant of LZW), 1030
Mathematica player (software), 1192
Matisse, Henri (1869–1954), 1169
Matlab software, properties of, 468, 764
matrices
   eigenvalues, 480
   norm of, 945
   QR decomposition, 495, 507–508
   sequence of, 476
Matsumoto, Teddy, 423
Maugham, William Somerset (1874–1965), 725
MDCT, see discrete cosine transform, modified
mean (in statistics), 162
mean absolute difference, 684, 872
mean absolute error, 872
mean square difference, 873
mean square error (MSE), 13, 463, 584, 816
measures of compression efficiency, 12–13
measures of distortion, 589
median
   definition of, 554
   in statistics, 162
memoryless source, 50, 320, 322, 325
   definition of, 2
meridian lossless packing, see MLP (audio)
mesh compression, edgebreaker, 1088, 1150–1161
mesopic vision, 527
Metcalfe, Robert Melancton (1946–), 952
metric, definition of, 589, 722
Mexican hat wavelet, 743, 746, 748, 749
Meyer, Carl D., 508
Meyer, Yves (and multiresolution decomposition), 790
Microcom, Inc., 32, 240, 1319
Microsoft windows media video (WMV), see VC-1
midriser quantization, 974
midtread quantization, 974
   in MPEG audio, 1041
minimal binary code, 123
   and phased-in codes, 84
mirroring, see lockstep
Mizner, Wilson (1876–1933), 23
MLP, 454, 611, 619–633, 635, 636, 961, 1317, 1319, 1321
MLP (audio), 13, 619, 979–984, 1319
MMR coding, 253, 567, 570, 573
   in JBIG2, 567
MNG, see multiple-image network format
MNP class 5, 32, 240–245, 1319
MNP class 7, 245–246, 1319
model
   adaptive, 293, 553
   context-based, 292
   finite-state machine, 1134
   in MLP, 626
Markov, 41, 245, 290, 456, 559, 1143, 1240, 1268
   of DMC, 1136, 1138
   of JBIG, 559
   of MLP, 621
   of PPM, 279
   of probability, 265, 290
   order-N, 294
   probability, 276, 323, 558, 559
   static, 292
   zero-probability problem, 293
modern, 7, 32, 235, 240, 248, 398, 1201, 1319, 1327
Moffat, Alistair, 609
Molnár, László, 423
Monk, Ian, 293
monkey’s audio, xv, xvi, 1017–1018, 1320
monochromatic image, see bi-level image
Montesquieu, (Charles de Secondat, 1689–1755), 1166
Morlet wavelet, 743, 749
Morse code, 25, 56–57, 61
non-UD, 63
Morse, Samuel Finley Breese (1791–1872), 1, 55, 61
Moschytz, George S. (LLM method), 504
Motil, John Michael (1938–), 224
motion compensation (in video compression), 871–880, 927, 930
motion vectors (in video compression), 871, 913
Motta, Giovanni (1965–), xv, 1088, 1189, 1318
move-to-front method, 10, 45–49, 1089, 1091, 1092, 1306, 1317, 1320
and wavelets, 752, 760
inverse of, 1093
Mozart, Joannes Chrysostomus Wolfgangus Theophilus (1756–1791), xvi
mp3, 1030–1055
and stuffit, 1195
and Tom’s Diner, 1081
compared to AAC, 1060–1062
mother of, see Vega, Susan
.mp3 audio files, 1030, 1054, 1263
and shorten, 992
mp4, see advanced audio coding
MPEG, 525, 858, 874, 1315, 1320
D picture, 892
DCT in, 883–891
IDCT in, 885–902
quantization in, 884–891
similar to JPEG, 883
MPEG-1 audio, 11, 475, 1030–1055, 1057
MPEG-1 video, 868, 880–902
MPEG-2, 868, 880, 903, 904
compared with others, 933–935
MPEG-2 audio, xv, 1055–1081
MPEG-3, 868, 1057
MPEG-4, 868, 902–907, 1057
AAC, 1076–1079
audio codecs, 1077
audio lossless coding (ALS), xv, 1018–1030, 1077, 1304
extensions to AAC, 1076–1079
MPEG-7, 1057
MQ coder, 280, 567, 842, 843, 848
MSE, see mean square error
MSZIP (deflate), 352
μ-law companding, 971–976, 987
multipass compression, 424
multiple-image network format (MNG), 420
multiresolution decomposition, 790, 1320
multiresolution image, 697, 1320
multiresolution tree (in wavelet decomposition), 770
multiring chain coding, 1142
Murray, James, 54
musical notation, 567, 742
Muth, Robert, 1176

N
n-step Fibonacci numbers, 134, 142, 146
N-trees, 668–674
Nakajima, Hideki, xvii
Nakamura Murashima offline dictionary method, 428–429
Nakamura, Hirofumi, 429
nanometer (definition of), 525
nat (information unit), 64
negate and exchange rule, 714
Nelson, Mark (1958–), 22
never-self-synchronizing codes, see affix codes
Newton, Isaac (1642–1727), 27, 74, 274
nibble code, 92
compared to zeta code, 123
nonadaptive compression, 9
nondifferentiable functions, 789
nonstandard (wavelet image decomposition), 792
nonstationary data, 295, 316
norm of a matrix, 945
normal distribution, 666, 1312, 1320
NSCT (never the same color twice), 864
NTSC (television standard), 857, 858, 864, 893, 1082
number bases, 141–142
primes, 152
numerical history (mists of), 633
Nyquist rate, 741, 781, 958
Nyquist, Harry (1889–1976), 741
Nyquist-Shannon sampling theorem, 444, 958

O
Oberhumer, Markus Franz Xaver Johannes, 423
O’Brien, Katherine, 151
Ochi, Hiroshi, 137
OCR (optical character recognition), 1128
octasection, 682–683, 1322
octave, 770
Index

in wavelet analysis, 770, 793
octree (in prefix compression), 1120
octrees, 669
odd functions, 514
offline dictionary-based methods, 424–429
Ogawa, Yoko (1962–), 1127
Ogg Squish, 997
Ohm’s law, 956
Okumura, Haruhiko (LZARI), xiv, 343, 399, 1317, 1318
omega code (Elias), 105–107, 125
and search trees, 130
and Stout $R$ code, 120
identical to code $C_4$, 114
its length, 155
optimal compression method, 11, 349
orthogonal
filters, 769
projection, 645
transform, 467, 472–515, 755–758, 944
orthonormal matrix, 468, 490, 492, 495, 767, 778
Osnach, Serge (PAQ2), 317

P

P-code (pattern code), 145, 147
packing, 28
PAL (television standard), 857–859, 862, 893, 935, 1082
Pandit, Vinayaka D., 706
PAQ, ix, 17, 314–319, 1193, 1320
and GS-2D-LZ, 582, 586
PAQ7, 318
PAQAR, 318
parametric cubic polynomial (PC), 628
parcor (in MPEG-4 ALS), 1022
parity, 434
of functions, 514
vertical, 435
parity bits, 179–180
parrots (image), 457
parse tree, 73
parsimony (principle of), 31
partial correlation coefficients, see parcor
Pascal triangle, 74–79
Pascal, Blaise (1623–1662), 3, 74, 160
PA$\nu$Da, 318
and the Calgary challenge, 15
patents of algorithms, 410, 437–439, 1321
pattern substitution, 35
Pavlov, Igor (7z and LZMA creator), xiv, xvi, 411, 415, 1303, 1318
PCT (photo core transform), 540
PDF, see portable document format
PDF (Adobe’s portable document format) and DjVu, 832
peak signal to noise ratio (PSNR), 13, 463–465, 933
Peano curve, 688
traversing, 694
used by mistake, 684
Peazip (a PAQ8 derivative), 319
pel, see also pixels
aspect ratio, 858, 893
difference classification (PDC), 873
fax compression, 249, 443
peppers (image), 517
perceptive compression, 10
Percival, Colin (BSDiff creator), 1178–1180, 1306
Perec, Georges (1936–1982), 293
permutation, 1089
petri dish, 1327
phased-in binary codes, 235, 376
phased-in codes, ix, 58, 81–89, 123, 376, 382, 1321
and interpolative coding, 173
and LZB, 342
centered, 84, 175
recursive, 58, 89–91
reverse centered, 84
suffix, 84–85
phonetic alphabet, 181
Phong shading (for polygonal surfaces), 1150
photo core transform (PCT), 540
photopic vision, 526, 527
phrase, 1321
in LZW, 365, 1127
physical compression, 12
physical entropy, 1205
Picasso, Pablo Ruiz (1881–1973), 498
PIFS (IFS compression), 720
Pigeon, Patrick Steven, 89, 90, 99, 100, 131, 133
pixels, 36, 444–446, 558, see also pel
and pointillism, 444
background, 557, 563, 1304
correlated, 467
decorrelated, 451, 454, 467, 475, 479, 514, 515
definition of, 443, 1321
foreground, 557, 563, 1304
highly correlated, 451
interpolation, 938–940
not a small square, x, 444
origins of term, 444
PK font compression, ix, 258–264
PKArc, 399, 1321
PKLite, 399, 423, 1321
PKunzip, 399, 1321
PKWare, 399, 1321
PKzip, 399, 1321
Planck constant, 739
plosive sounds, 986
plotting (of functions), 779–781
PNG, see portable network graphics
pod code, xiv, 168, 995
Poe, Edgar Allan (1809–1849), 9
pointillism, 444
points (cross correlation of), 468
Poisson distribution, 301, 302
polygonal surfaces compression,
edgebreaker, 1088, 1150–1161
polynomial
bicubic, 631
definition of, 436, 628
parametric cubic, 628
parametric representation, 628
polynomials (interpolating), 621, 626–633, 805–808, 813, 1314
degree-5, 807
Porta, Giambattista della (1535–1615), 1
portable document format (PDF), xv, 1088, 1167–1169, 1193, 1321
portable network graphics (PNG), 416–420, 438, 1321
and stuffit, 1195
PostScript (and LZW patent), 437
Poutanen, Tomi (LZX), 352
Powell, Anthony Dymoke (1905–2000), 752
power law distribution, 122
power law distribution of probabilities, 104
PPM, 51, 290, 292–312, 346, 635, 1094, 1103, 1131, 1142, 1149, 1321
and PAQ, 314
exclusion, 300–301
trie, 302
vine pointers, 303
word-based, 1123–1125
PPM (fast), 312–313
PPM*, 307–309
PPMA, 301
PPMB, 301, 636
PPMC, 298, 301
PPMD (in RAR), 397
PPMxH (by Dmitry Shkarin), 411
PPMP, 301
PPMX, 301
PPMZ, 309–312
and LZPP, 347
PPPM, 635–636, 1321
prediction, 1321
nth order, 644, 993, 994, 1021
AAC, 1074–1076
ADPCM, 977, 978
BTPC, 653, 655, 656
CALIC, 637
CELP, 991
definition of, 292
deterministic, 558, 564, 566
FELICS, 1240
image compression, 454
JPEG-LS, 523, 535, 541–543
long-term, 1026
LZP, 384
LZRW4, 364
MLP, 619, 621
MLP audio, 983
monkey’s audio, 1018
Paeth, 420
PAQ, 314
PNG, 416, 418
PPM, 297
PPM*, 307
PPMZ, 309
PPPM, 635
probability, 290
progressive, 1025
video, 869, 874
preechomies
in AAC, 1073
in MPEG audio, 1051–1055
prefix codes, 42, 58, 62–94, 349, 393, 453
and video compression, 874
prefix compression
images, 674–676, 1321
sparse strings, 1116–1120
prefix property, 58, 61, 63, 247, 613, 1322, 1327
prime numbers (as a number system), 152
probability
conditional, 1308
model, 15, 265, 276, 290, 323, 553, 558, 559, 1320
adaptive, 293
Prodigy (and LZW patent), 437
product vector quantization, 597–598
progressive compression, 557, 559–567
progressive FELICS, 615–617, 619, 620, 1322
progressive image compression, 456, 549–557, 1322
growth geometry coding, 555–557
lossy option, 549, 620
median, 554
MLP, 454
SNR, 550, 852
progressive prediction, 1025
properties of speech, 985–986
Proust, Marcel Valentin Louis George
Eugene (1871–1922), 1316, (Colophon)
Prowse, David (Darth Vader, 1935–), 235
PSNR, see peak signal to noise ratio
psychoacoustic model (in MPEG audio),
1030, 1032–1033, 1035, 1040, 1044, 1049,
1051, 1054, 1263, 1322
psychoacoustics, 10, 961–966
pulse code modulation (PCM), 960
punctured Elias codes, 58, 112–113
punting (definition of), 648
PWCM (PAQ weighted context mixing), 318, 319
Pylak, Pawel, xi
pyramid (Laplacian), 732, 792, 811–814
pyramid (wavelet image decomposition), 753, 793
pyramid coding (in progressive compression), 550, 652, 654, 656, 792, 811–814
Q
Qi, Honggang, xi
QIC, 248, 350
QIC-122, 350–351, 1322
QM coder, 280–288, 313, 522, 535, 1322
QMF, see quadrature mirror filters
QR matrix decomposition, 495, 507–508
QTCQ, see quadtree classified trellis coded quantized
quadrant numbering (in a quadtree), 659, 696
quadrature mirror filters, 778
quadrisect, 676–683, 1322
quadtree classified trellis coded quantized wavelet image compression (QTCQ), 825–826
quadtrees, 453, 658–676, 683, 695, 1305, 1312, 1321, 1322, 1328
and Hilbert curves, 684
and IFS, 722
and quadrisection, 676, 678
prefix compression, 674–676, 1117, 1321
quadrant numbering, 659, 696
spatial orientation trees, 826
quantization
block truncation coding, 603–609, 1305
definition of, 49
image transform, 455, 467, 1326
in H.261, 908
in JPEG, 529–530
in MPEG, 884–891
midriser, 974
midtread, 974, 1041
scalar, 49–51, 448–449, 466, 835, 1323
in SPIHT, 818
vector, 466, 550, 588–603, 710, 1327
adaptive, 598–603
quantization noise, 643
quantized source, 188
Quantum (dictionary-based encoder), 352
quaternary (base-4 numbering), 660, 1322
Quayle, James Danforth (Dan, 1947–), 866
queue (data structure), 337, 1306, 1309
quincunx (wavelet image decomposition), 792
quindecimal (base-15), 92
Index

R

random data, 7, 217, 398, 1211, 1327
random variable (definition of), 16
range encoding, 279–280, 1018
  in LZMA, 412
  in LZPP, 344
Rao, Ashok, 706
RAR, xiv, 395–397, 1192, 1323
Rarissimo, 397, 1323
raster (origin of word), 857
raster order scan, 42, 453, 454, 467, 549,
  579, 615, 621, 635, 637, 638, 648–650,
  876, 883, 895
rate–distortion theory, 332
ratio of compression, 12, 1307
Ratushnyak, Alexander
  and PAQAR, 318
Hutter prize, 291, 319
  the Calgary challenge, 15, 18
RBUC, see recursive bottom-up coding
reasons for data compression, 2
recursive bottom-up coding (RBUC), ix,
  107–109, 1323
recursive compression, 7
recursive decoder, 585
recursive pairing (re-pair), 427–428
recursive phased-in codes, 58, 89–91
recursive range reduction (3R), xiv, xvi,
  51–54, 1323
redundancy, 64, 104, 213, 247
  alphabetic, 3
  and data compression, 3, 396, 448, 1136
  and reliability, 247
  and robust codes, 178
  contextual, 3
  definition of, 66–67, 1202, 1231
  direction of highest, 792
  spatial, 451, 869, 1327
  temporal, 869, 1327
redundancy feedback (rf) coding, 85–89
  Reed-Solomon error-correcting code, 395
  reflected Gray code, 454, 456–463, 558, 647,
  694, 1313
  Hilbert curve, 684
reflection, 712
reflection coefficients, see parcor
refresh rate (of movies and TV), 856–858,
  865, 866, 881, 894
relative encoding, 35–36, 374, 641, 870,
  1311, 1323
  in JPEG, 523
reliability, 247
  and Huffman codes, 247
  as the opposite of compression, 25
  in RAR, 395
renormalization (in the QM-coder),
  282–288, 1221
repetition finder, 391–394
repetition times, 348–350
representation (definition of), 57
residual vector quantization, 597
resolution
  of HDTV, 864–866
  of images (defined), 443
  of television, 857–860
  of the eye, 527
resynchronizing Huffman code, 190–193
reverse centered phased-in codes, 84
reversible codes, see bidirectional codes
Reynolds, Paul, 331
Reznik, Yuriy, xi
RF coding, see redundancy feedback coding
RGB
  color space, 523, 935
  reasons for using, 526
RGC, see reflected Gray code
RHC, see resynchronizing Huffman code
Ribera, Francisco Navarrete y (1600–?), 294
Rice codes, 52, 59, 166–170, 617, 1304, 1312,
  1323, 1324
  as bidirectional codes, 196–198
  as start-step-stop codes, 98
  fast PPM, 313
  FLAC, 997, 1001–1002
  in hyperspectral data, 1186
  not used in WavPack, 1011–1012
  Shorten, 994, 995
  subexponential code, 170, 617
Rice, Robert F. (Rice codes developer,
  1944–), 163, 166, 994
Richardson, Iain, 910
Riding the Bullet (novel), 1167
Riemann zeta function, 123
RIFF (Resource Interchange File Format),
  969
Rijndael, see advanced encryption standard
Rizzo, Francesco, 1189, 1318
Index

RK, see WinRK
RK (commercial compression software), 347
RLE, see run-length encoding, 31–54, 250, 453, 1323
and BW method, 1089, 1091, 1306
and sound, 966, 967
and wavelets, 752, 760
BinHex4, 43–44
BMP image files, 44–45, 1305
image compression, 36–40
in JPEG, 522
QIC-122, 350–351, 1322
RMSE, see root mean square error
Robinson, Tony (Shorten developer), 169
robust codes, 177–209
Rodeh, Michael (1949–), 111
Rokicki, Tomas Gerhard Paul, 259
Roman numerals, 732
root mean square error (RMSE), 464
Roshal, Alexander Lazarevitch, 395
Roshal, Eugene Lazarevitch (1972–), xiv, xvi, 395, 396, 1323
Rota, Gian Carlo (1932–1999), 159
rotation, 713
90°, 713
matrix of, 495, 500
rotations
Givens, 499–508
improper, 495
in three dimensions, 512–513
roulette game (and geometric distribution), 160
run-length encoding, 9, 31–54, 161, 217, 448, 1319
and EOB, 530
and Golomb codes, see also RLE
BTC, 606
FLAC, 1001
in images, 453
MNP5, 240
RVLC, see bidirectional codes, reversible codes
Ryan, Abram Joseph (1839–1886), 753, 760, 767
Sagaran, Vijayakumar, 503, 1236
Sayood, Khalid, 640
SBC (speech compression), 987
scalar quantization, 49–51, 448–449, 466, 1323
in SPIHT, 818
in WSQ, 835
scaling, 711
Schalkwijk’s variable-to-block code, 74–79
Schindler, Michael, 1089
Schmidt, Jason, 317
scotopic vision, 526, 527
Scott, Charles Prestwich (1846–1932), 880
Scott, David A., 317
SCSU (Unicode compression), 1088, 1161–1166, 1323
SECAM (television standard), 857, 862, 893
secure codes (cryptography), 177
self compression, 433
self-delimiting codes, 58, 92–94
self-similarity in images, 456, 695, 702, 1312, 1328
Sellman, Jane, 105
semaphore code, 145
semiadaptive compression, 10, 234, 1324
semiadaptive Huffman coding, 234
semistructured text, 1088, 1149–1150, 1324
sequence (definition of), 476
sequitur, 35, 1088, 1145–1150, 1308, 1324
and dictionary-based methods, 1149
set partitioning in hierarchical trees (SPIHT), 732, 815–826, 1312, 1325
and CREW, 827
Seurat, Georges-Pierre (1859–1891), 444
seven deadly sins, 1058
SHA-256 hash algorithm, 411
shadow mask, 858, 859
Shakespeare, William (1564–1616), 194, 1198
Shanahan, Murray, 313
Shannon, Claude Elwood (1916–2001), 64, 66, 178, 211, 1094, 1202, 1311, 1314
Shannon-Fano method, 211–214, 1304, 1324, 1325
shearing, 712
shift invariance, 1309
Shkarin, Dmitry, 397, 411
shorten (speech compression), 169, 992–997, 1002, 1184, 1312, 1324
sibling property, 236
Sierpiński
curve, 683, 688–691
gasket, 716–719, 1250, 1252
triangle, 716, 718, 1250
and the Pascal triangle, 76
Sierpiński, Wacław (1882–1969), 683, 688, 716, 718
sieve of Eratosthenes, 157
sign-magnitude (representation of integers), 849
signal to noise ratio (SNR), 464
signal to quantization noise ratio (SQNR), 465
silence compression, 967
simple images, EIDAC, 577–579, 1324
simple-9 code, 93–94
sins (seven deadly), 1058
skewed probabilities, 267
Skibiński, Przemysław (PAQDa developer, 1978–), 318
SLH, ix, 342–343, 1324
sliding window compression, 334–348, 579, 1149, 1305, 1324
repetition times, 348–350
SLIM algorithm, 18
small numbers (easy to compress), 46, 531, 535, 554, 641, 647, 651, 652, 752, 1092, 1267
Smith Micro Software (stuffit), 1192
SMPTE (society of motion picture and television engineers), 927
SNR, see signal to noise ratio
SNR progressive image compression, 550, 841, 852
Society of Motion Picture and Television Engineers (SMPTE), 927
Soderberg, Lena (of image fame, 1951–), 520
solid archive, see RAR
sort-based context similarity, 1087, 1105–1110
sound
fricative, 986
plosive, 986
properties of, 954–957
sampling, 958–961
unvoiced, 986
voiced, 985
source (of data), 16, 64
gapless, 188
quantized, 188
source coding, 64, 177
formal name of data compression, 2
source speech codecs, 986, 988–991
SourceForge.net, 997
SP theory (simplicity and power), 8
SP-code (synchronizable pattern code), 147
space-filling curves, 453, 683–694, 1324
Hilbert, 684–688
Peano, 688
Sierpiński, 688–691
sparse strings, 28, 146, 1087, 1110–1120, 1324
prefix compression, 1116–1120
sparseness ratio, 12, 764
spatial orientation trees, 820–821, 826, 827
spatial prediction, x, 725–728, 1324
spatial redundancy, 451, 869, 1327
in hyperspectral data, 1182
spectral dimension (in hyperspectral data), 1182
spectral selection (in JPEG), 523
speech (properties of), 985–986
speech compression, 781, 954, 984–996
μ-law, 987
A-law, 987
AbS, 991
ADPCM, 987
ATC, 987
CELP, 991
CS-CELP, 991
DPCM, 987
hybrid codecs, 986, 991
LPC, 988–991
SBC, 987
shorten, 992–997, 1312, 1324
source codecs, 986, 988–991
owncers, 986, 988
waveform codecs, 986–987
Sperry Corporation
LZ78 patent, 439
LZW patent, 437
SPIHT, see set partitioning in hierarchical trees
SQNR, see signal to quantization noise ratio
square integrable functions, 743
Squish, 997
stack (data structure), 374, 1309
Stafford, David (quantum dictionary compression), 352
standard (wavelet image decomposition), 753, 792, 793
standard test images, 517–519
standards (organizations for), 247–248
standards of television, 760, 857–862, 1082
start-step-stop codes, ix, 58, 91, 97–99
and exponential Golomb codes, 198
start/stop codes, 58, 99–100
static dictionary, 329, 330, 357, 375
statistical distributions, see distributions statistical methods, 9, 211–327, 330, 449–450, 1325
context modeling, 292
unification with dictionary methods, 439–441
statistical model, 265, 292, 329, 558, 559, 1320
steganography (data hiding), 350, see also how to hide data
Stirring formula, 77
stone-age binary (unary code), 96
Storer, James Andrew, 339, 1189, 1271, 1318, 1319, 1329
Stout codes, 58, 119–121
and search trees, 130
Stout, Quentin Fielden, 119
stream (compressed), 16
streaming mode, 11, 1089
string compression, 331–332, 1325
structured VQ methods, 594–598
stuffit (commercial software), x, 7, 1088, 1191–1198, 1325
subband (minimum size of), 792
subband transform, 467, 755–758, 767
subexponential code, 166, 170, 617, 994
subsampling, 466, 1325
in video compression, 870
successive approximation (in JPEG), 523
suffix codes, 196
ambiguous term, 59
phased-in codes, 84–85
support (of a function), 750
surprise (as an information measure), 1199, 1204
SVC, see H.264 video compression
SWT, see symmetric discrete wavelet transform
syllable-based data compression, x, 1125–1127, 1325
symbol ranking, 290, 1087, 1094–1098, 1103, 1105, 1325
symmetric (wavelet image decomposition), 791, 836
symmetric codes, 202–204
symmetric compression, 10, 330, 351, 522, 835
symmetric context (of a pixel), 611, 621, 636, 638, 848
symmetric discrete wavelet transform, 835
synchronous codes, 59, 149, 184–193
synthetic image, 447
Szymanski, Thomas G., 339, 1319
T
T/F codec, see time/frequency (T/F) codec
taboo codes, 59, 131–135
as bidirectional codes, 195
block-based, 131–133
unconstrained, 133–135
taps (wavelet filter coefficients), 772, 785, 786, 792, 827, 836, 1325
TAR (Unix tape archive), 1325
Tarantino, Quentin Jerome (1963–), 135
Tartaglia, Niccolò (1499–1557), 27
Taylor, Malcolm, 318
television
aspect ratio of, 857–860
resolution of, 857–860
scan line interlacing, 865
standards used in, 760, 857–862
temporal masking, 964, 966, 1032
temporal redundancy, 869, 1327
tera (= 2^40), 834
ternary comma code, 58, 116–117
text
case flattening, 27
English, 3, 4, 15, 330
files, 10
natural language, 246
random, 7, 217, 1211
semistructured, 1088, 1149–1150, 1324
text compression, 9, 15, 31
Hutter prize, ix, 290–291, 319, 1314
LZ, 1318
QIC-122, 350–351, 1322
RLE, 31–35
symbol ranking, 1094–1098, 1105
textual image compression, 1087,
1128–1134, 1326
textual substitution, 599
Thomas, Lewis (1913–1993), 184
Thompson, Kenneth (1943–), 71
Thomson, William (Lord Kelvin
1824–1907), 741
TIFF
and JGIB2, 1315
and LZW patent, 437, 438
and stuffit, 1194
time/frequency (T/F) codec, 1060, 1303,
1326
title of this book, vii
Tjalkens–Willems variable-to-block code,
79–81
Toeplitz, Otto (1881–1940), 1006, 1007
token (definition of), 1326
tokens
dictionary methods, 329
in block matching, 579, 580, 582
in LZ77, 335, 336, 400
in LZ78, 354, 355
in LZFG, 358
in LZSS, 339
in LZW, 365
in MNP5, 241, 242
in prefix compression, 675, 676
in QIC-122, 350
Toole, John Kennedy (1937–1969), 49
training (in data compression), 41, 250, 293,
432, 517, 580, 588, 590, 599, 638, 639,
685, 886, 1104, 1127, 1143
transforms, 9
AC coefficient, 471, 481, 484
DC coefficient, 471, 481, 484, 485, 514,
523, 530–532, 535
definition of, 732
discrete cosine, 475, 480–514, 522,
528–529, 758, 913, 918, 1049, 1310
3D, 480, 1186–1188
discrete Fourier, 528
discrete sine, 513–515
Fourier, 467, 732–741, 770, 772
Haar, 475, 477–478, 510, 749–767
Hotelling, see Karhunen-Loève transform
images, 467–515, 684, 685, 755–758, 1326
integer, 481, 943–949
inverse discrete cosine, 480–514, 528–529,
1236
inverse discrete sine, 513–515
inverse Walsh-Hadamard, 475–477
Karhunen-Loève, 475, 478–480, 512, 758
orthogonal, 467, 472–515, 755–758, 944
subband, 467, 755–758, 767
Walsh-Hadamard, 475–477, 540, 758, 918,
919, 1233
translation, 714
tree
adaptive Huffman, 234–236, 1127
binary search, 339, 340, 348, 1319
balanced, 339, 341
skewed, 339, 341
data structure, 1309
Huffman, 214, 215, 220, 234–236, 1314
height of, 227–228
unique, 404
Huffman (decoding), 238
Huffman (overflow), 238
Huffman (rebuilt), 238
tree-structured vector quantization, 596–597
trends (in an image), 742
triangle (Sierpiński), 716, 718, 1250
triangle mesh compression, edgebreaker,
1088, 1150–1161, 1326
trie
definition of, 302, 357
LZW, 369
Patricia, 415
trigram, 292, 293, 346
and redundancy, 3
trit (ternary digit), 64, 116, 141, 226, 694,
1027, 1200, 1201, 1213, 1325, 1326
Trudeau, Joseph Philippe Pierre Yves Elliott (1919–2000), 133
Truta, Cosmin, xi, xv, 420
TSVQ, see tree-structured vector quantization
Tunstall code, 58, 72–74, 195, 1326
combined with Huffman, 219–220
Turing test (and the Hutter prize), 291
Twain, Mark (1835–1910), 32
two-pass compression, 10, 234, 265, 390, 431, 532, 624, 1112, 1122, 1324

U
UD, see uniquely decodable codes
UDA (a PAQ8 derivative), 319
Udupa, Raghavendra, 706
Ulam, Stanislaw Marcin (1909–1984), 159–160
Ulysses (novel), 27
unary code, 58, 92, 96, 170, 390, 614, 617, 676, 1312, 1326
a special case of Golomb code, 162
general, 97–100, 359, 581, 1237, 1326, see also stone-age binary
ideal symbol probabilities, 114
uncertainty principle, 737–740, 749
and MDCT, 1050
Unicode, 60, 339, 1237, 1306, 1326
Unicode compression, 1088, 1161–1166, 1323
unification of statistical and dictionary methods, 439–441
uniform (wavelet image decomposition), 795
uniquely decodable (UD) codes, 57, 58, 69
not prefix codes, 68, 145, 149
Unisys (and LZW patent), 437, 438
universal codes, 68–69
universal compression method, 11, 349
univocalic, 293
UNIX
compact, 234
compress (LZC), ix, 357, 362, 375–376, 437, 438, 1193, 1307, 1319, 1321
Gzip, 375, 438
unvoiced sounds, 986
UP/S code, see unary prefix/suffix codes
UPX (exe compressor), 423

V
V.32, 235
V.32bis, 398, 1327
V.42bis, 7, 398, 1327
Vail, Alfred (1807–1859), 56, 61
Valenta, Vladimir, 695
Vanryper, William, 54
variable-length codes, viii, 3, 57–209, 211, 220, 234, 239, 241, 245, 329, 331, 1211, 1322, 1325, 1327
and reliability, 247, 1323
and sparse strings, 1112–1116
in fax compression, 250
unambiguous, 69, 1316
variable-length error-correcting (VLEC) codes, 204–208
variable-to-block codes, 58, 71–81
variable-to-fixed codes, 72
variance, 452
and MLP, 622–626
as energy, 468
definition of, 624
of differences, 641
of Huffman codes, 216
VC-1 (Video Codec), 868, 927–952, 1327
compared with others, 933–935
transform, 943–949
VCDIFF (file differencing), 1171–1173, 1327
vector quantization, 466, 550, 588–603, 710, 1327
AAC, 1077
adaptive, 598–603
exhaustive search, 594–596
hyperspectral data, 1188–1191
LBG algorithm, 589–594, 598, 685, 1191
locally optimal partitioned vector quantization, 1188–1191
product, 597–598
quantization noise, 643
residual, 597
structured methods, 594–598
tree-structured, 596–597
vector spaces, 509–512, 645
Vega, Suzanne (1959, mother of the mp3), 1081
video
analog, 855–862
digital, 863–864, 1310
high definition, 864–866
video compression, 869–952, 1327
  block differencing, 871
  differencing, 870
distortion measures, 872–873
  H.261, 907–909, 1313
  H.264, xiv, 532, 910–926, 1313
  historical overview, 867–868
inter frame, 869
intra frame, 869
motion compensation, 871–880, 927, 930
motion vectors, 871, 913
MPEG-1, 874, 880–902, 1320
MPEG-1 audio, 1030–1055
subsampling, 870
VC-1, 927–952
Vigna, Sebastiano, 122
vine pointers, 303
vision (human), 526–528, 1181
VLC, see variable-length codes
VLEC, see variable-length error-correcting codes
vmail (email with video), 863
vocoders speech codecs, 986, 988
voiced sounds, 985
Voronoi diagrams, 594, 1327

W
  Wagner's Ring Cycle, 1056
  Walsh-Hadamard transform, 473, 475–477, 540, 758, 918, 919, 1233
  Wang's flag code, 59, 135–137
  Wang, Muzhong, 135
  warm colors, 528
  Warnock, John (1940–), 1167
  WAVE audio format, xiv, 969–971
  wave particle duality, 740
  waveform speech codecs, 986–987
  wavelet image decomposition
    adaptive wavelet packet, 796
    full, 795
    Laplacian pyramid, 792
    line, 792
    nonstandard, 792
    pyramid, 753, 793
    quincunx, 792
    standard, 753, 792, 793
    symmetric, 791, 836
    uniform, 795
  wavelet packet transform, 795
  wavelet scalar quantization (WSQ), 732, 834–840
  wavelets, 454, 456, 475, 741–853
    Beylkin, 781
    Coifman, 781
    continuous transform, 528, 743–749, 1308
    Daubechies, 781–786
    D4, 769, 776, 778
    D8, 1256, 1257
    discrete transform, 528, 777–789, 1310
    biorthogonal, 769
    decimation, 768
    deriving filter coefficients, 775–776
    orthogonal, 769
    fingerprint compression, 789, 834–840, 1328
    Haar, 749–767, 781
    image decompositions, 791–798
    integer transform, 809–811
    lazy transform, 799
    lifting scheme, 798–808, 1317
    Mexican hat, 743, 746, 748, 749
    Morlet, 743, 749
    multiresolution decomposition, 790, 1320
    origin of name, 743
    quadrature mirror filters, 778
    symmetric, 781
    used for plotting functions, 779–781
    Vaidyanathan, 781
  wavelets scalar quantization (WSQ), 732, 834–840
  WavPack audio compression, xiv, 1007–1017, 1328
  web browsers
    and FABD, 650
    and GIF, 395, 437
    and PDF, 1167
    and PNG, 416
    and XML, 421
    DjVu, 831
  Web site of this book, xi, xvi
  webgraph (compression of), 122
  Weierstrass, Karl Theodor Wilhelm (1815–1897), 789
  weighted finite automata, 662, 695–707, 1328
  Weisstein, Eric W. (1969–), 684
Welch, Terry A. (1939–1988), 331, 365, 437
Welch, Terry A. (?–1985), 1319
WFA, see weighted finite automata
Wheeler, David John (BWT developer, 1927–2004), 1089
Wheeler, John Archibald (1911–2008), 65
Wheeler, Wayne, xi, xiii
Whitman, Walt (1819–1892), 422
Whittle, Robin, 168, 995
WHT, see Walsh-Hadamard transform
Wilde, Erik, 539
Willems, Frans M. J. (1954–), 348
Williams, Ross N., 361, 364, 439
Wilson, Sloan (1920–2003), 295
WinRAR, xiv, 395–397
and LZPP, 347
WinRK, 318
and PWCM, 319
WinUDA (a PAQ6 derivative), 319
Wirth, Niklaus Emil (1934–), 687
Wister, Owen (1860–1938), 382
Witten, Ian Hugh, 292
WMV, see VC-1
Wolf, Stephan, xiv, xvi, 673
woodcuts
- unusual pixel distribution, 634
word-aligned packing, 92–94
word-based compression, 1121–1125
Wordsworth, William (1770–1850), 194
Wright, Ernest Vincent (1872–1939), 293, 294
WSQ, see wavelet scalar quantization
www (web), 122, 437, 521, 984, 1316
X
Xerox Techbridge (OCR software), 1129
XML compression, XMill, 421–422, 1328
XOR, see exclusive OR
xylography (carving a woodcut), 635
Y
Yamamoto’s flag code, 59, 137–141
Yamamoto’s recursive code, 58, 125–128
Yamamoto, Hirohiko, 125, 127, 137
Yao, Andrew Chi Chih (1946–), 128
YCbCr color space, 452, 503, 525, 526, 538, 844, 862
Yeung, Raymond W., 171
YIQ color model, 707, 760
Yokoo, Hidetoshi, 391, 394, 1110
Yoshizaki, Haruyasu, 399, 1317
YPbPr color space, 503
YUV color space, 935
Z
zdelta, 1173–1175, 1328
Zeckendorf’s theorem, 142, 151
Zeilberger, Doron (1950–), 157
Zelazny, Roger (1937–1995), 406
zero-probability problem, 293, 296, 553, 611, 634, 886, 1136, 1328
in LZPP, 347
zero-redundancy estimate (in CTW), 327
zeta (ζ) codes, 58, 122–124
zeta (ζ) function (Riemann), 123
zigzag sequence, 453
in H.261, 907
in H.264, 919
in HD photo, 540
in JPEG, 530, 1236
in MPEG, 888, 890, 902, 1261
in RLE, 39
in spatial prediction, 725
in VC-1, 950
three-dimensional, 1187–1188
Zip (compression software), 400, 1193, 1309, 1328
and stuffit, 1195
Zipf, George Kingsley (1902–1950), 122
Ziv, Jacob (1931–), 51, 331, 1318
LZ78 patent, 439
Zurek, Wojciech Hubert (1951–), 1205
Indexing requires decision making of a far higher
order than computers are yet capable of.
Colophon

This volume is an extension of *Data Compression: The Complete Reference*, whose first edition appeared in 1996. The book was designed by the authors and was typeset with the TeX typesetting system developed by D. Knuth. The text and tables were done with Textures and TeXshop on a Macintosh computer. The figures were drawn in Adobe Illustrator. Figures that required calculations were computed either in Mathematica or Matlab, but even those were “polished” in Adobe Illustrator. The following facts illustrate the amount of work that went into the book:

- The book (including the auxiliary material located in the book’s Web site) contains about 523,000 words, consisting of about 3,081,000 characters (big, even by the standards of Marcel Proust). However, the size of the auxiliary material collected in the author’s computer and on his shelves while working on the book is about 10 times bigger than the entire book. This material includes articles and source codes available on the Internet, as well as many pages of information collected from various sources.

- The text is typeset mainly in font cmr10, but about 30 other fonts were used.

- The raw index file has about 5150 items.

- There are about 1300 cross references in the book.

You can’t just start a new project in Visual Studio/Delphi/whatever, then add in an ADPCM encoder, the best psychoacoustic model, some DSP stuff, Levinson Durbin, subband decomposition, MDCT, a Blum-Blum-Shub random number generator, a wavelet-based brownian movement simulator and a Feistel network cipher using a cryptographic hash of the Matrix series, and expect it to blow everything out of the water, now can you?

Anonymous, found in [hydrogenaudio 06]