23
Compression Techniques

In the last few decades, the digital computer has risen to become an integral part of our lives. It influences every aspect of our existence from commerce to kitchens and from entertainment to education. A large part of that influence is graphic. Digital images, whether computer generated or produced by a camera, are all around us. Images may be important, influential, entertaining, and profitable, but they are also large, which is why compressing images has become an important topic of research and why image compression constitutes an entire part (and not just a single chapter) of this book.

The first half of this chapter discusses the concept of redundancy and the principle of data compression. This is followed by the basic features and types of digital images, the main approaches to image compression, and a description of several basic image compression methods. The second half of the chapter is devoted to variable-length codes, a family of codes widely used in the compression of images and of data in general.

23.1 Redundancy in Data

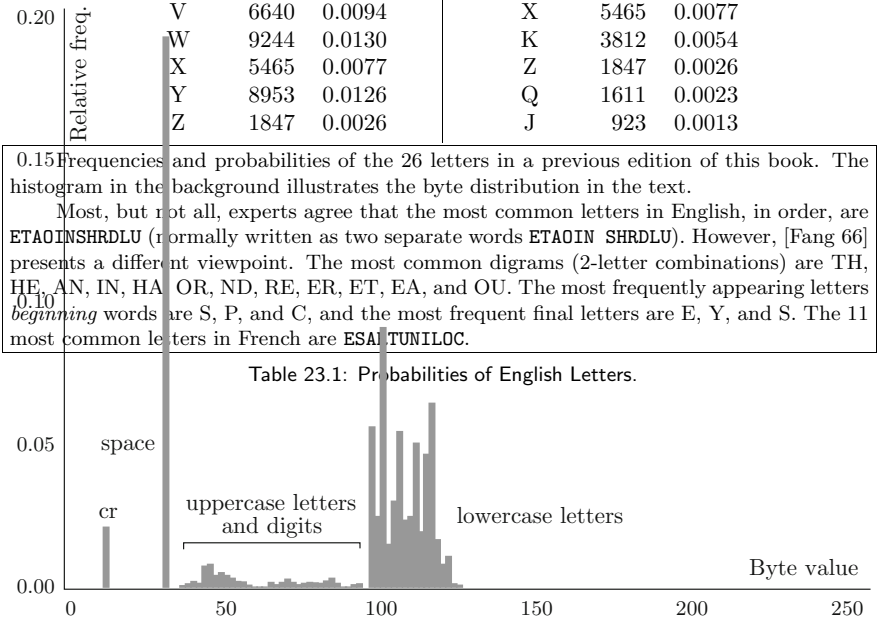
There are many known methods for the compression of images and for data compression in general. They are based on different ideas, are suitable for different types of data, and produce different results, but they are all based on the same principle; they compress data by removing *redundancy* from the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

To illustrate the meaning of the term "redundancy" we start with text. In typical English text, the letter `E` appears very often, while `Z` is rare (Tables 23.1 and 23.2). This

D. Salomon, *The Computer Graphics Manual*, Texts in Computer Science,
DOI 10.1007/978-0-85729-886-7_23, © Springer-Verlag London Limited 2011

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

Table 23.1: Probabilities of English Letters.



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

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

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

Frequencies and probabilities of the 93 most-common characters in an older book by this author, containing 861,462 characters. See Figure 23.3 for the *Mathematica* code.

Table 23.2: Frequencies and Probabilities of Characters.

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

Figure 23.3: Code for Table 23.2.

alphabetic redundancy suggests a simple way to compress text. Assign variable-length codes to the letters, with `E` being assigned the shortest code and `Z` receiving the longest code. Another type of redundancy, *contextual redundancy*, is illustrated by the fact that the letter `Q` is almost always followed by the letter `U` (i.e., that in plain English certain digrams and trigrams are more common than others).

◇ **Exercise 23.1:** (Fun.) Find English words that contain all five vowels "aeiou" in their original order.

In this book we are interested in images, and redundancy in images stems from the well-known fact that in a nonrandom image adjacent pixels tend to have similar colors. This important fact is the principle of image compression (Page 1035) and forms the basis of all the image compression methods.

The theory of information, developed in 1948 by Claude Shannon, discusses redundancy and offers a rigorous definition of this term. However, even without a precise definition, it is intuitively clear that a variable-length code has less redundancy than a fixed-length code (and may have no redundancy at all). Fixed-length codes make it easier to work with text and pixels, which is why they are useful, but they contribute to data redundancy.

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

Compressing data is done by changing its representation from inefficient (i.e., long) to efficient (short). Compression is therefore possible only because data is normally represented in the computer in a format that is longer than absolutely necessary. The reason that inefficient (long) data representations are used all the time is that they make

it easier to process the data, and data processing is more common and more important than data compression. The ASCII code for characters is a good example of a data representation that is longer than absolutely necessary. It uses 7-bit codes because fixed-size codes are easy to work with. A variable-size code, however, would be more efficient, since certain characters are used more than others and so could be assigned shorter codes.

In a world where data is always represented by its shortest possible format, there would therefore be no way to compress data.

23.2 Image Types

For the purpose of image compression it is useful to distinguish the following types of images: (For information on pixels and their history, see the lively references [Lyon 09] and [Smith 09].)

1. A *bi-level* (or monochromatic) image. This is an image where the pixels can have one of two values, normally referred to as black and white (but also as foreground and background). Each pixel in such an image is represented by one bit, making this the simplest type of image.
2. A *grayscale* image. A pixel in such an image can have one of the n values 0 through $n-1$, indicating one of 2^n shades of gray (or shades of some other color). The value of n is normally compatible with a byte size; i.e., it is 4, 8, 12, 16, 24, or some other convenient multiple of 4 or of 8. The set of the most-significant bits of all the pixels is the most-significant bitplane. Thus, a grayscale image has n bitplanes.
3. A *continuous-tone* image. This type of image can have many similar colors (or grayscales). When adjacent pixels differ by just one unit, it is hard or even impossible for the eye to distinguish their colors. As a result, such an image may contain areas with colors that seem to vary continuously as the eye moves along the area. A pixel in such an image is represented by either a single large number (in the case of many grayscales) or three components (in the case of a color image). A continuous-tone image is normally a natural image (natural as opposed to artificial) and is obtained by taking a photograph with a digital camera, or by scanning a photograph or a painting. Figures 24.40 through 24.43 are typical examples of continuous-tone images. A general survey of lossless compression of this type of images is [Carpentieri et al. 00].
4. A *discrete-tone* image (also called a graphical image or a synthetic image). This is normally an artificial image. It may have a few colors or many colors, but it does not have the noise and blurring of a natural image. Examples are an artificial object or machine, a page of text, a chart, a cartoon, or the contents of a computer screen. (Not every artificial image is discrete-tone. A computer-generated image that's meant to look natural is a continuous-tone image in spite of its being artificially generated.) Artificial objects, text, and line drawings have sharp, well-defined edges, and are therefore highly contrasted from the rest of the image (the background). Adjacent pixels in a discrete-tone image often are either identical or vary significantly in value. Such an image does not compress well with lossy methods, because the loss of just a few pixels may render a letter illegible, or change a familiar pattern to an unrecognizable one. Compression methods for continuous-tone images often do not handle sharp edges very well, so special

methods are needed for efficient compression of these images. Notice that a discrete-tone image may be highly redundant, since the same character or pattern may appear many times in the image. Figure 24.44 is a typical example of a discrete-tone image.
5. A *cartoon-like* image. This is a color image that consists of uniform areas. Each area has a uniform color but adjacent areas may have very different colors. This feature may be exploited to obtain excellent compression.

> Whether an image is treated as discrete or continuous is usually dictated by the depth of the data. However, it is possible to force an image to be continuous even if it would fit in the discrete category. (From www.genaware.com)

It is intuitively clear that each type of image may feature redundancy, but they are redundant in different ways. This is why any given compression method may not perform well for all images, and why different methods are needed to compress the different image types. There are compression methods for bi-level images, for continuous-tone images, and for discrete-tone images. There are also methods that try to break an image up into continuous-tone and discrete-tone parts, and compress each separately.

23.3 Redundancy in Images

Modern computers employ graphics extensively. Window-based operating systems display the disk's file directory graphically. The progress of many system operations, such as downloading a file, may also be displayed graphically. Many applications provide a graphical user interface (GUI), which makes it easier to use the program and to interpret displayed results. Computer graphics is used in many areas in everyday life to convert many types of complex information to images. Thus, images are important, but they tend to be big! Modern hardware can display many colors, which is why it is common to have a pixel represented internally as a 24-bit number, where the percentages of red, green, and blue occupy eight bits each. Such a 24-bit pixel can specify one of $2^{24} \approx 16.78$ million colors. As a result, an image at a resolution of 512×512 that consists of such pixels occupies 786,432 bytes. At a resolution of 1024×1024 it becomes four times as big, requiring 3,145,728 bytes. Videos are also commonly used in computers, making for even bigger images. This is why image compression is so important. An important feature of image compression is that it can be lossy. An image, after all, exists for people to look at, so, when it is compressed, it is acceptable to lose image features to which the eye is not sensitive. This is one of the main ideas behind the many lossy image compression methods described in the data compression literature.

In general, information can be compressed if it is redundant. It has already been mentioned that data compression amounts to reducing or removing redundancy in the data. With lossy compression, however, we have a new concept, namely compressing by removing *irrelevancy*. An image can be lossy-compressed by removing irrelevant information even if the original image does not have any redundancy.

◇ **Exercise 23.2:** It would seem that an image with no redundancy is always random (and therefore uninteresting). It that so?

The idea of losing image information becomes more palatable when we consider how digital images are created. Here are three examples: (1) A real-life image may be

scanned from a photograph or a painting and digitized (converted to pixels). (2) An image may be recorded by a digital camera that creates pixels and stores them directly in memory. (3) An image may be painted on the screen by means of a paint program. In all these cases, some information is lost when the image is digitized. The fact that the viewer is willing to accept this loss suggests that further loss of information might be tolerable if done properly.

(Digitizing an image involves two steps: *sampling* and *quantization*. Sampling an image is the process of dividing the two-dimensional original image into small regions: pixels. Quantization is the process of assigning an integer value to each pixel. Notice that digitizing sound involves the same two steps, with the difference that sound is one-dimensional.)

Here is a simple process that can determine qualitatively the amount of data loss in a compressed image. Given an image A, (1) compress it to B, (2) decompress B to C, and (3) subtract $D = C - A$. If A was compressed without any loss and decompressed properly, then C should be identical to A and image D should be uniformly white. The more data was lost in the compression, the farther will D be from uniformly white.

How should an image be compressed? The common compression programs that are used in practice and described in books are based on a few techniques such as run-length encoding (RLE), scalar quantization, statistical methods, and dictionary-based methods. None of these techniques is very satisfactory for color or grayscale images (although they may be used in combination with other methods). Here is why:

Reference [Salomon 09] shows how run-length encoding (RLE) can be used for (lossless or lossy) compression of an image. This is simple, and it is used by certain parts of JPEG, especially by its lossless mode. In general, however, the transform employed by JPEG (Section 24.5.1) produces much better compression than does RLE alone. Facsimile compression employs RLE combined with Huffman coding and obtains good results, but only for bi-level images.

Scalar quantization is discussed in [Salomon 09] and other texts. It can be used to compress images, but its performance is mediocre. Imagine an image with 8-bit pixels. It can be compressed with scalar quantization by cutting off the four least-significant bits of each pixel. This yields a compression ratio of 0.5, not very impressive, and at the same time reduces the number of colors (or grayscales) from 256 to just 16. Such a reduction not only degrades the overall quality of the reconstructed (decompressed) image, but may also create bands of different colors, a noticeable and annoying effect that's illustrated here.

Imagine a row of 12 pixels with similar colors, ranging from 202 to 215. In binary notation these values are

11010111 11010110 11010101 11010011 11010010 11010001 11001111 11001110 11001101 11001100 11001011 11001010.

Quantization will result in the 12 4-bit values

1101 1101 1101 1101 1101 1101 1100 1100 1100 1100 1100 1100,

which will reconstruct the 12 pixels

11010000 11010000 11010000 11010000 11010000 11010000 11000000 11000000 11000000 11000000 11000000 11000000.

The first six pixels of the row now have the value $11010000_2 = 208$, while the next six pixels are $11000000_2 = 192$. If neighboring rows have similar pixels, the first six columns will form a band, distinctly different from the band formed by the next six columns. This banding, or contouring, effect is very noticeable to the eye, since our eyes are sensitive

to edges and breaks in an image.

One way to eliminate this effect is called *improved grayscale* (IGS) *quantization*. It works by adding to each pixel a random number generated from the four rightmost bits of previous pixels. Section 23.4.1 shows that the least-significant bits of a pixel are fairly random, so IGS works by adding to each pixel randomness that depends on the neighborhood of the pixel.

The method maintains an 8-bit variable, denoted by `rsm`, that's initially set to zero. For each 8-bit pixel P to be quantized (except the first one), the IGS method does the following:

1. Set `rsm` to the sum of the eight bits of P and the four rightmost bits of `rsm`. However, if P has the form $1111xxxx$, set `rsm` to P.
2. Write the four leftmost bits of `rsm` on the compressed stream. This is the compressed value of P. IGS is thus not exactly a quantization method, but a variation of scalar quantization.

The first pixel is quantized in the usual way, by dropping its four rightmost bits. Table 23.4 illustrates the operation of IGS.

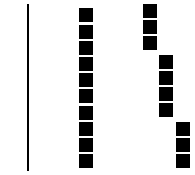
Pixel	Value	`rsm`	Compressed value
1	1010 0110	0000 0000	1010
2	1101 0010	1101 0010	1101
3	1011 0101	1011 0111	1011
4	1001 1100	1010 0011	1010
5	1111 0100	1111 0100	1111
6	1011 0011	1011 0111	1011

Table 23.4: Illustrating the IGS Method.

Vector quantization (Section 23.5.2) can be used more successfully to compress images. It is discussed in detail in [Salomon 09].

Statistical methods work best when the symbols being compressed have different probabilities. An input stream where all symbols have the same probability will not compress, even though it may not be random. It turns out that in a continuous-tone color or grayscale image, the different colors or shades of gray may often have roughly the same probabilities. This is why statistical methods are not a good choice for compressing such images, and why new approaches are needed. Images with color discontinuities, where adjacent pixels have widely different colors, compress better with statistical methods, but it is not easy to predict, just by looking at an image, whether it has enough color discontinuities.

Dictionary-based compression methods also tend to be unsuccessful in dealing with continuous-tone images. Such an image typically contains adjacent pixels with similar colors, but does not contain repeating patterns. Even an image that contains repeated patterns such as vertical lines may lose them when digitized. A vertical line in the original image may become slightly crooked when the image is digitized (Figure 23.5), so the pixels in a scan row may end up having slightly different colors from those in



An ideal vertical rule is shown in (a). In (b), the rule is assumed to be perfectly digitized into ten pixels, stacked vertically. However, if the image is placed in the scanner slightly crooked, the scanning may be imperfect, and the resulting pixels might look as in (c).

Figure 23.5: Perfect and Imperfect Digitizing.

neighboring rows, resulting in a dictionary with short strings. (This problem may also affect curved edges.)

Another problem with dictionary compression of images is that such methods scan the image row by row, and therefore may miss vertical correlations between pixels. An example is the two simple images of Figure 23.6a,b. Saving both in GIF89, a dictionary-based graphics file format, has resulted in file sizes of 1,053 and 1,527 bytes, respectively, on the author's computer.

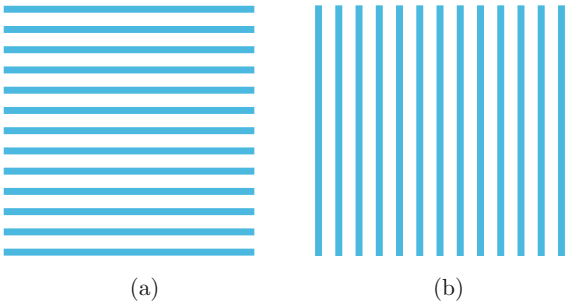


Figure 23.6: Dictionary Compression of Parallel Lines.

Traditional methods are therefore unsatisfactory for image compression, so this chapter discusses novel approaches. They are all different, but they remove redundancy from an image by employing the following principle:

The principle of image compression. If we select a pixel in an image at random, there is a good chance that its neighbors (especially its immediate neighbors) will have the same color or very similar colors.

Image compression is therefore based on the fact that neighboring pixels are *highly correlated*. This correlation is also called *spatial redundancy*.

Here is a simple example that illustrates what can be done with correlated pixels. The following sequence of values gives the intensities of 24 adjacent pixels in a row of a

continuous-tone image:

12, 17, 14, 19, 21, 26, 23, 29, 41, 38, 31, 44, 46, 57, 53, 50, 60, 58, 55, 54, 52, 51, 56, 60.

Only two of the 24 pixels are identical. Their average value is 40.3. Subtracting pairs of adjacent pixels results in the sequence

12, 5, −3, 5, 2, 4, −3, 6, 11, −3, −7, 13, 4, 11, −4, −3, 10, −2, −3, 1, −2, −1, 5, 4.

The two sequences are illustrated in Figure 23.7.

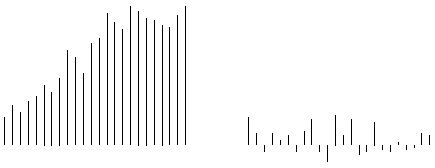


Figure 23.7: Values and Differences of 24 Adjacent Pixels.

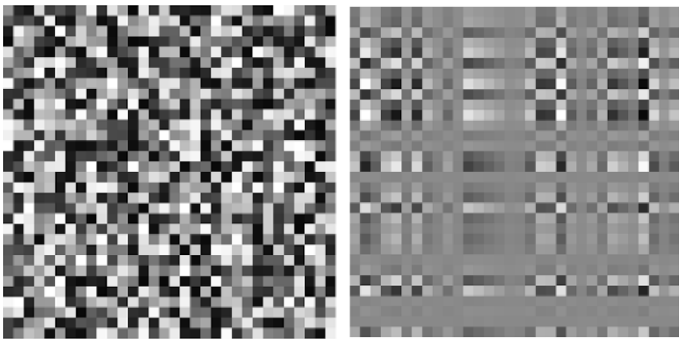
The sequence of difference values has three properties that illustrate its compression potential: (1) The difference values are smaller than the original pixel values. Their average is 2.58. (2) They repeat. There are just 15 distinct difference values, so in principle they can be coded by four bits each. (3) They are *decorrelated*. Adjacent difference values tend to be different. This can be seen by subtracting them, which results in the sequence of 24 second differences

12, −7, −8, 8, −3, 2, −7, 9, 5, −14, −4, 20, −11, 7, −15, 1, 13, −12, −1, 4, −3, 1, 6, 1.

They are larger than the differences themselves.

Figure 23.8 provides another illustration of the meaning of the term "correlated quantities." A 32×32 matrix A is constructed of random numbers, and its elements are displayed in part (a) as shaded squares. The random nature of the elements is clear. The matrix is then inverted and stored in B, which is shown in part (b). This time, there seems to be more structure to the 32×32 squares. A direct calculation using Equation (23.1) shows that the cross-correlation between the top two rows of A is 0.0412, whereas the cross-correlation between the top two rows of B is −0.9831 (these numbers change each time the code is run, because different random numbers are generated). The elements of B are correlated since each depends on *all* the elements of A

$$R = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{[n\sum x_i^2 - (\sum x_i)^2][n\sum y_i^2 - (\sum y_i)^2]}}. \tag{23.1}$$



(a) (b)

```
rm=RandomReal[1, {32,32}];
Graphics[Raster[rm]]
irm=Inverse[rm];
Graphics[Raster[irm,Automatic, {Min[irm],Max[irm]}]]
```

Figure 23.8: Maps of (a) a Random Matrix and (b) its Inverse.

◇ **Exercise 23.3:** Use mathematical software to illustrate the covariance matrices of (1) a matrix with correlated values and (2) a matrix with decorrelated values.

Once the concept of correlated quantities is grasped, we start looking for a correlation test. Given a matrix M, a statistical test is needed to determine whether its elements are correlated or not. The test is based on the statistical concept of covariance. If the elements of M are decorrelated (i.e., independent), then the covariance of any two different rows and any two different columns of M will be zero (the covariance of a row or of a column with itself is always 1). As a result, the covariance matrix of M (whether covariance of rows or of columns) will be diagonal. If the covariance matrix of M is not diagonal, then the elements of M are correlated. The statistical concepts of variance, covariance, and correlation are discussed in any text on statistics.

The principle of image compression has another aspect. We know from experience that the *brightnesses* of neighboring pixels are also correlated. Two adjacent pixels may have different colors. One may be mostly red, and the other may be mostly green. Yet if the red component of the first is bright, the green component of its neighbor will, in most cases, also be bright. This property can be exploited by converting pixel representations from RGB to three other components, one of which is the brightness, and the other two represent color. One such format (or *color space*) is YCbCr, where Y (the "luminance" component) represents the brightness of a pixel, and Cb and Cr define its color. This format is discussed in Section 21.12, but its advantage is easy to understand. The eye is sensitive to small changes in brightness but not to small changes in color. Thus, losing information in the Cb and Cr components compresses the image while introducing distortions to which the eye is not sensitive. Losing information in the Y component, on the other hand, is very noticeable to the eye.

23.4 Approaches to Image Compression

An image compression method is normally designed for a specific type of image, and this section lists various approaches to compressing images of different types. Only the general principles are discussed here; specific methods are described in the many texts on data and image compression.

Approach 1: This approach is appropriate for bi-level images. A pixel in such an image is represented by one bit. Applying the principle of image compression to a bi-level image therefore means that the immediate neighbors of a pixel P tend to be *identical* to P. Thus, it makes sense to use run-length encoding (RLE) to compress such an image. A compression method for such an image may scan it in raster order (row by row) and compute the lengths of runs of black and white pixels. The lengths are encoded by variable-length codes and are written on the compressed stream. An example of such a method is facsimile compression [Salomon 09].

It should be stressed that this is just an approach to bi-level image compression. The details of specific methods vary. For instance, a method may scan the image column by column or in zigzag (Figure 23.9), it may convert the image to a quadtree (Section 18.7), or it may scan it region by region using a space-filling curve.

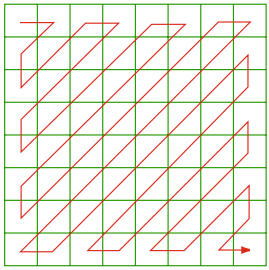


Figure 23.9: A Zigzag Sequence.

Approach 2: Also for bi-level images. The principle of image compression tells us that the neighbors of a pixel tend to be similar to the pixel. We can extend this principle and conclude that if the current pixel has color c (where c is either black or white), then pixels of the same color seen in the past (and also those that will be found in the future) tend to have the same immediate neighbors.

This approach looks at n of the near neighbors of the current pixel and considers them an n-bit number. This number is the *context* of the pixel. In principle there can be 2^n contexts, but because of image redundancy we expect them to be distributed in a nonuniform way. Some contexts should be common while others will be rare.

The encoder counts the number of times each context has already been found for a pixel of color c, and assigns probabilities to the contexts accordingly. If the current pixel has color c and its context has probability p, the encoder can use adaptive arithmetic coding to encode the pixel with that probability. This approach is used by JBIG [Salomon 09].

Next, we turn to grayscale images. A pixel in such an image is represented by n bits and can have one of 2^n values. Applying the principle of image compression to a grayscale image implies that the immediate neighbors of a pixel P tend to be similar to P, but are not necessarily identical. Thus, RLE should not be used to compress such an image. Instead, two approaches are discussed.

Approach 3: Separate the grayscale image into n bi-level images and compress each with run-length encoding (RLE) and prefix codes. The principle of image compression seems to imply intuitively that two adjacent pixels that are similar in the grayscale image will be identical in most of the n bi-level images. This, however, is not true, as the following example makes clear. Imagine a grayscale image with $n = 4$ (i.e., 4-bit pixels, or 16 shades of gray). The image can be separated into four bi-level images. If two adjacent pixels in the original grayscale image have values 0000 and 0001, then they are similar. They are also identical in three of the four bi-level images. However, two adjacent pixels with values 0111 and 1000 are also similar in the grayscale image (their values are 7 and 8, respectively) but differ in all four bi-level images.

This problem occurs because the binary codes of adjacent integers may differ by several bits. The binary codes of 0 and 1 differ by one bit, those of 1 and 2 differ by two bits, and those of 7 and 8 differ by four bits. The solution is to design special binary codes such that the codes of any consecutive integers i and $i + 1$ will differ by one bit only. An example of such a code is the *reflected Gray codes* of Section 23.4.1.

Approach 4: Use the *context* of a pixel to *predict* its value. The context of a pixel is the values of some of its neighbors. We can examine some neighbors of a pixel P, compute an average A of their values, and predict that P will have the value A. The principle of image compression tells us that our prediction will be correct in most cases, almost correct in many cases, and completely wrong in a few cases. We can say that the predicted value of pixel P represents the redundant information in P. We now calculate the difference

$$\Delta \stackrel{\text{def}}{=} P - A,$$

and assign variable-length codes to the different values of Δ such that small values (which we expect to be common) are assigned short codes and large values (which are expected to be rare) are assigned long codes. If P can have the values 0 through $m - 1$, then values of Δ are in the range $[-(m-1), +(m-1)]$, and the number of codes needed is $2(m - 1) + 1$ or $2m - 1$.

Experiments with a large number of images suggest that the values of Δ tend to be distributed according to the Laplace distribution (Figure 23.10). A compression method can, therefore, use this distribution to assign a probability to each value of Δ, and use arithmetic coding to encode the Δ values very efficiently. This is the principle of the MLP method [Salomon 09].

The context of a pixel may consist of just one or two of its immediate neighbors. However, better results may be obtained when several neighbor pixels are included in the context. The average A in such a case should be weighted, with near neighbors assigned higher weights. Another important consideration is the decoder. In order for it to decode the image, it should be able to compute the context of every pixel. This means that the context should employ only pixels that have already been encoded. If the image is scanned in raster order, the context should include only pixels located above the current pixel or on the same row and to its left.

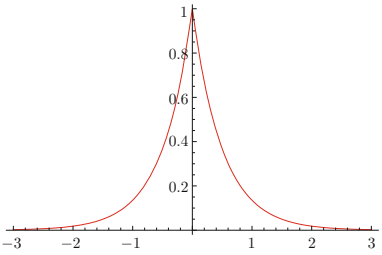


Figure 23.10: Laplace distribution.

Approach 5: Transform the values of the pixels and encode the transformed values. The concept of a transform, as well as the most important transforms used in image compression, are discussed in Chapter 24. Chapter 25 is devoted to the wavelet transform. Recall that compression is achieved by reducing or removing redundancy. The redundancy of an image is caused by the correlation between pixels, so transforming the pixels to a representation where they are decorrelated eliminates the redundancy. It is also possible to think of a transform in terms of the entropy of the image. In a highly correlated image, the pixels tend to have equiprobable values, which results in maximum entropy. If the transformed pixels are decorrelated, certain pixel values become common, thereby having large probabilities, while others are rare. This results in small entropy. Quantizing the transformed values can produce efficient lossy image compression. We want the transformed values to be independent because coding independent values makes it simpler to construct a statistical model.

We now turn to color images. A pixel in such an image consists of three color components, such as red, green, and blue. Most color images are either continuous-tone or discrete-tone.

Approach 6: The principle of this approach is to separate a continuous-tone color image into three grayscale images and compress each of the three separately, using approaches 3, 4, or 5.

For a continuous-tone image, the principle of image compression implies that adjacent pixels have similar, although perhaps not identical, colors. However, similar colors do not mean similar pixel values. Consider, for example, 12-bit pixel values where each color component is expressed in four bits. Thus, the 12 bits 1000|0100|0000 represent a pixel whose color is a mixture of eight units of red (about 50%, since the maximum is 15 units), four units of green (about 25%), and no blue. Now imagine two adjacent pixels with values 0011|0101|0011 and 0010|0101|0011. They have similar colors, since only their red components differ, and only by one unit. However, when considered as 12-bit numbers, the two numbers 001101010011 and 001001010011 are very different, since they differ in one of their most significant bits.

An important feature of this approach is to use a luminance chrominance color representation instead of the more common RGB. The concepts of luminance and chrominance are discussed in Section 21.12. The advantage of the luminance chrominance color representation is that the eye is sensitive to small changes in luminance but not in

chrominance. This allows the loss of considerable data in the chrominance components, while making it possible to decode the image without a significant visible loss of quality.
Approach 7: A different approach is needed for discrete-tone images. Recall that such an image contains uniform regions, and a region may appear several times in the image. A good example is a screen dump. Such an image consists of text and icons. Each character of text and each icon is a region, and any region may appear several times in the image. A possible way to compress such an image is to scan it, identify regions, and find repeating regions. If a region B is identical to an already found region A, then B can be compressed by writing a pointer to A on the compressed stream. The block decomposition method (FABD, [Salomon 09]) is an example of how this approach can be implemented.
Approach 8: Partition the image into parts (overlapping or not) and compress it by processing the parts one by one. Suppose that the next unprocessed image part is part number 15. Try to match it with parts 1–14 that have already been processed. If part 15 can be expressed, for example, as a combination of parts 5 (scaled) and 11 (rotated), then only the few numbers that specify the combination need be saved, and part 15 can be discarded. If part 15 cannot be expressed as a combination of already-processed parts, it is declared processed and is saved in raw format.

This approach is the basis of the various *fractal* methods for image compression. It applies the principle of image compression to image parts instead of to individual pixels. Applied in this way, the principle tells us that "interesting" images (i.e., those that we keep and try to compress) have a certain amount of *self similarity*. Parts of the image are identical or similar to the entire image or to other parts.

Image compression methods are not limited to these basic approaches. Texts on data compression discuss methods that use the concepts of context trees, Markov models, and wavelets, among others. In addition, the concept of progressive image compression [Salomon 09] should be mentioned, since it adds another dimension to the field of image compression.

23.4.1 Gray Codes

An image compression method that has been developed specifically for a certain type of image can sometimes be used for other types. Any method for compressing bi-level images, for example, can be used to compress grayscale images by separating the bitplanes and compressing each individually, as if it were a bi-level image. Imagine, for example, an image with 16 grayscale values. Each pixel is specified by four bits, so the image can be separated into four bi-level images. The trouble with this approach is that it violates the general principle of image compression. Imagine two adjacent 4-bit pixels with values $7 = 0111_2$ and $8 = 1000_2$. These pixels have close values, but when separated into four bitplanes, the resulting 1-bit pixels are different in every bitplane! This is because the binary representations of the consecutive integers 7 and 8 differ in all four bit positions. In order to apply any bi-level compression method to grayscale images, a binary representation of the integers is needed where consecutive integers have codes differing by one bit only. Such a representation exists and is called *reflected Gray code* (RGC). This code is easy to generate with the following recursive construction:

Start with the two 1-bit codes $(0, 1)$. Construct two sets of 2-bit codes by duplicating $(0, 1)$ and appending, either on the left or on the right, first a zero, then a one, to the

original set. The result is $(00, 01)$ and $(10, 11)$. We now reverse (reflect) the second set, and concatenate the two. The result is the 2-bit RGC $(00, 01, 11, 10)$; a binary code of the integers 0 through 3 where consecutive codes differ by exactly one bit. Applying the rule again produces the two sets $(000, 001, 011, 010)$ and $(110, 111, 101, 100)$, which are concatenated to form the 3-bit RGC. Note that the first and last codes of any RGC also differ by one bit. Here are the first three steps for computing the 4-bit RGC:

$$\begin{gathered}\text{Add a zero } (0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100),\\ \text{Add a one } (1000, 1001, 1011, 1010, 1110, 1111, 1101, 1100),\\ \text{reflect } (1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000).\end{gathered}$$

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	0**1000**	10010	**10000**	00011	1**1000**	10001
0000**1**	00100	0100**1**	10110	1000**1**	00111	1100**1**	10101
000**10**	01100	010**10**	11110	100**10**	01111	110**10**	11101
0001**1**	01000	0101**1**	11010	1001**1**	01011	1101**1**	11001
00**100**	11000	01**100**	01010	10**100**	11011	11**100**	01001
0010**1**	11100	0110**1**	01110	1010**1**	11111	1110**1**	01101
001**10**	10100	011**10**	00110	101**10**	10111	111**10**	00101
0011**1**	10000	0111**1**	00010	1011**1**	10011	1111**1**	00001

```
function b=rgc(a,i)
[r,c]=size(a);
b=[zeros(r,1),a; ones(r,1),flipud(a)];
if i>1, b=rgc(b,i-1); end;
```

Table 23.11: First 32 Binary and Reflected Gray Codes.

Table 23.11 shows how individual bits change when moving through the binary codes of the first 32 integers. The 5-bit binary codes of these integers are listed in the odd-numbered columns of the table, with the bits of integer i that differ from those of $i-1$ shown in boldface. It is easy to see that the least-significant bit (bit b_0) changes all the time, bit b_1 changes for every other number, and, in general, bit b_k changes every k integers. The even-numbered columns list one of the several possible reflected Gray codes for these integers. A recursive Matlab function to compute RGC is also listed.

◇ **Exercise 23.4:** It is also possible to generate the reflected Gray code of an integer n with the following nonrecursive rule: xor n with a copy of itself that's logically shifted one position to the right. In the C programming language this is denoted by `n^(n>>1)`. Use this expression to construct a table similar to Table 23.11.

The conclusion is that the most-significant bitplanes of an image obey the principle of image compression more than the least-significant ones. When adjacent pixels

```
clear;
filename='parrots128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
mask=1; % between 1 and 8


nimg=bitget(img,mask);
imagesc(nimg), colormap(gray)
```

Binary code

```
clear;
filename='parrots128'; dim=128;
fid=fopen(filename,'r');
img=fread(fid,[dim,dim])';
mask=1 % between 1 and 8
a=bitshift(img,-1);
b=bitxor(img,a);
nimg=bitget(b,mask);
imagesc(nimg), colormap(gray)
```

Gray code

Figure 23.12: Matlab Code to Separate Image Bitplanes.

have values that differ by one unit (such as p and $p+1$), chances are that the least-significant bits are different and the most-significant ones are identical. Any image compression method that compresses bitplanes individually should therefore treat the least-significant bitplanes differently from the most-significant ones, or should use RGC instead of the binary code to represent pixels. Figures 23.14, 23.15, and 23.16 (prepared by the Matlab code of Figure 23.12) show the eight bitplanes of the well-known parrots image in both the binary code (the left column) and in RGC (the right column). The bitplanes are numbered 8 (the leftmost or most-significant bits) through 1 (the rightmost or least-significant bits). It is obvious that the least-significant bitplane doesn't show any correlations between the pixels; it is random or very close to random in both binary and RGC. Bitplanes 2 through 5, however, exhibit better pixel correlation in the Gray code. Bitplanes 6 through 8 look different in Gray code and binary, but seem to be highly correlated in either representation.

43210	Gray	43210	Gray	43210	Gray	43210	Gray
00000	00000	0**1000**	01100	**10000**	11000	1**1000**	10100
0000**1**	00001	0100**1**	01101	1000**1**	11001	1100**1**	10101
000**10**	00011	010**10**	01111	100**10**	11011	110**10**	10111
0001**1**	00010	0101**1**	01110	1001**1**	11010	1101**1**	10110
00**100**	00110	01**100**	01010	10**100**	11110	11**100**	10010
0010**1**	00111	0110**1**	01011	1010**1**	11111	1110**1**	10011
0011**10**	00101	0111**10**	01001	1011**10**	11101	1111**10**	10001
0011**1**	00100	0111**1**	01000	1011**1**	11100	1111**1**	10000

```
a=linspace(0,31,32); b=bitshift(a,-1);
b=bitxor(a,b); dec2bin(b)
```

Table 23.13: First 32 Binary and Gray Codes.

Figure 23.17 is a graphic representation of two versions of the first 32 reflected Gray codes. Part (b) shows the codes of Table 23.11, and part (c) shows the codes of

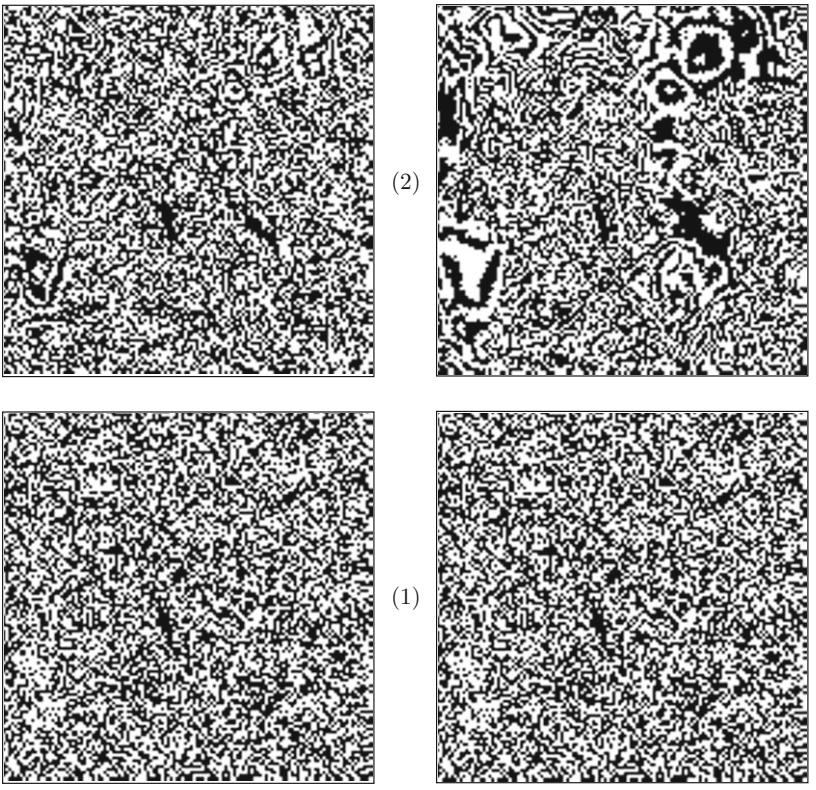


Figure 23.14: Bitplanes 1 and 2 of the Parrots Image.

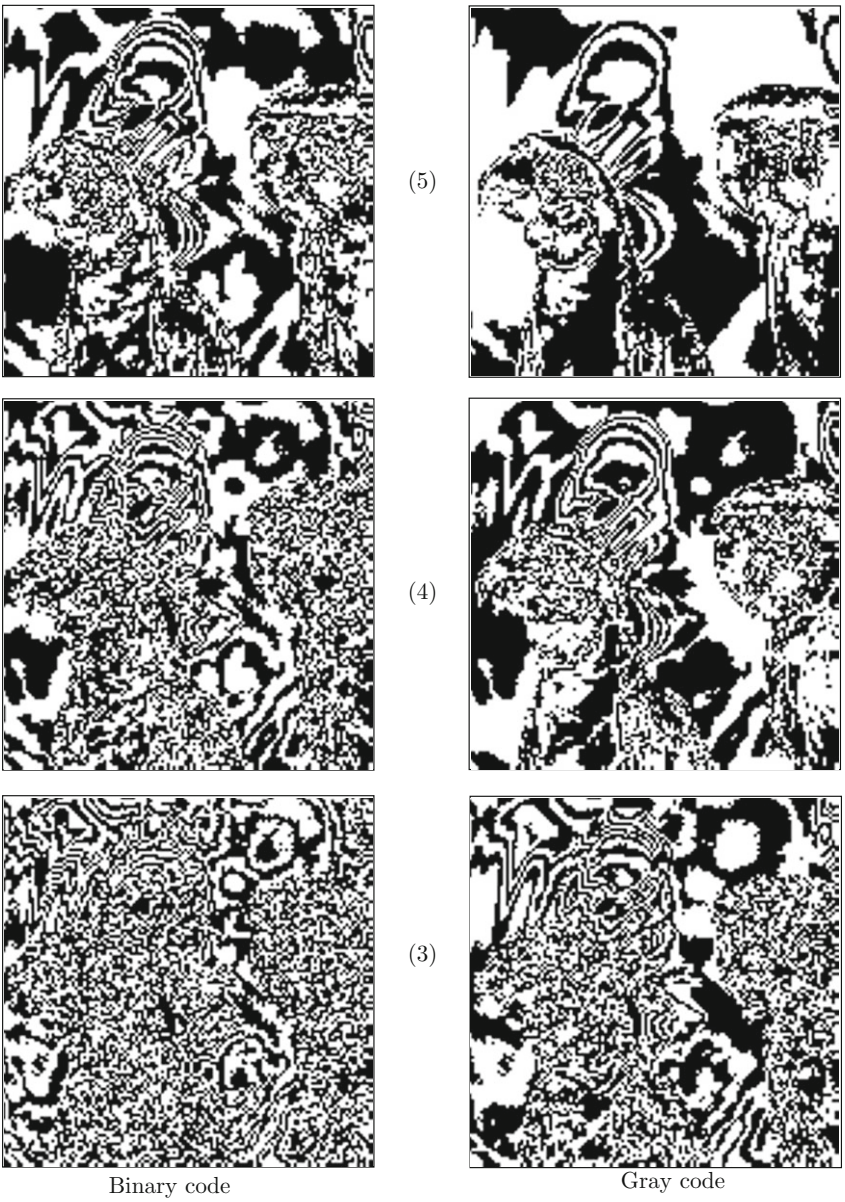


Figure 23.15: Bitplanes 3, 4, and 5 of the Parrots Image.

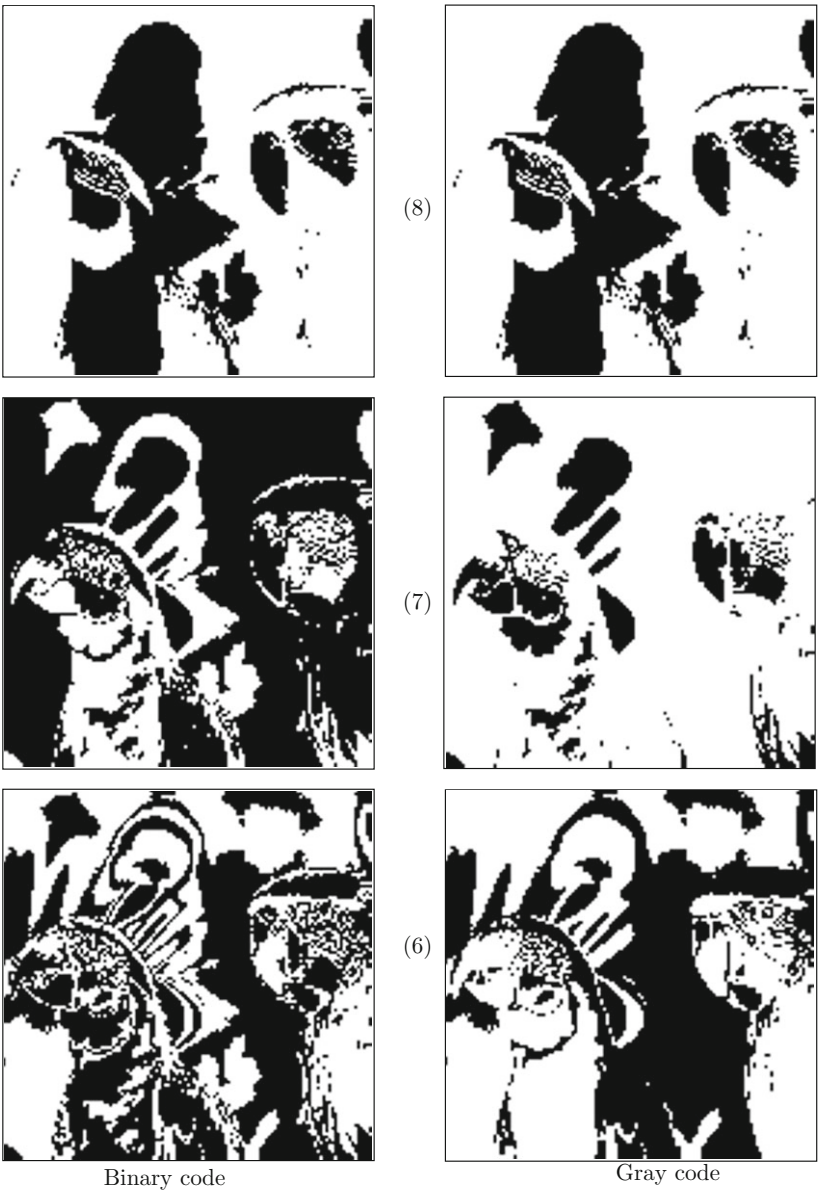


Figure 23.16: Bitplanes 6, 7, and 8 of the Parrots Image.

Table 23.13. Even though both are Gray codes, they differ in the way the bits in each bitplane alternate between 0 and 1. In part (b), the bits of the most-significant bitplane alternate four times between 0 and 1. Those of the second most-significant bitplane alternate eight times between 0 and 1, and the bits of the remaining three bitplanes alternate 16, two, and one times between 0 and 1. When the bitplanes are separated, the middle bitplane features the smallest correlation between the pixels, since the Gray codes of adjacent integers tend to have different bits in this bitplane. The Gray codes shown in Figure 23.17c, on the other hand, alternate more and more between 0 and 1 as we move from the most significant bitplanes to the least-significant ones. The least significant bitplanes of this version feature less and less correlation between the pixels and therefore tend to be random. For comparison, Figure 23.17a shows the binary code. It is obvious that bits in this code alternate more often between 0 and 1.

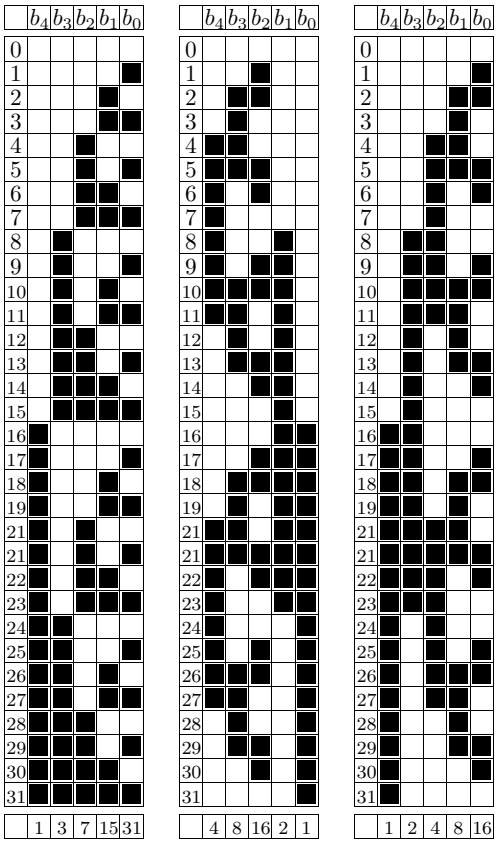
◇ **Exercise 23.5:** Even a cursory look at the Gray codes of Figure 23.17c shows that they exhibit some regularity. Examine these codes carefully and identify two features that may be used to compute the codes.

◇ **Exercise 23.6:** Figure 23.17 is a graphic representation of the binary codes and reflected Gray codes. Find a similar graphic representation of the same codes that illustrates the fact that the first and last codes also differ by one bit.

Color images provide another example of using the same compression method across image types. Any compression method for grayscale images can be used to compress color images. In a color image, each pixel is represented by three color components (such as RGB). Imagine a color image where each color component is represented by one byte. A pixel is represented by three bytes, or 24 bits, but these bits should not be considered a single number. The two pixels 118|206|12 and 117|206|12 differ by just one unit in the first component, so they have very similar colors. Considered as 24-bit numbers, however, these pixels are very different, since they differ in one of their most-significant bits. Any compression method that treats these pixels as 24-bit numbers would consider these pixels very different, and its performance would suffer as a result. A compression method for grayscale images can be applied to compressing color images, but the color image should first be separated into three color components, and each component compressed individually as a grayscale image.

23.4.2 Error Metrics

Developers and implementers of lossy image compression methods need a standard metric to measure the quality of reconstructed images compared with the original ones. The better a reconstructed image resembles the original one, the bigger should be the value produced by this metric. Such a metric should also produce a dimensionless number, and that number should not be very sensitive to small variations in the reconstructed image. A common measure used for this purpose is the *peak signal to noise ratio* (PSNR). It is familiar to workers in the field, it is also simple to calculate, but it has only a limited, approximate relationship with the perceived errors noticed by the human visual system. This is why higher PSNR values imply closer resemblance between the reconstructed and the original images, but they do not provide a guarantee that viewers will like the reconstructed image.



(a) (b) (c)

Table 23.17: First 32 Binary and Reflected Gray Codes.

> The binary Gray code is fun,
> For in it strange things can be done.
> Fifteen, as you know,
> Is one, oh, oh, oh,
> And ten is one, one, one, one.
>
> —Anonymous.

History of Gray Codes

Gray codes are named after Frank Gray, who patented their use for shaft encoders in 1953 [Gray 53]. However, the work was performed much earlier, the patent being applied for in 1947. Gray was a researcher at Bell Telephone Laboratories. During the 1930s and 1940s he was awarded numerous patents for work related to television. According to [Heath 72] the code was first, in fact, used by J. M. E. Baudot for telegraphy in the 1870s, though it is only since the advent of computers that the code has become widely known.

The Baudot code uses five bits per symbol. It can represent $32 \times 2 - 2 = 62$ characters (each code can have two meanings, the meaning being indicated by the `LS` and `FS` codes). It became popular and, by 1950, was designated the International Telegraph Code No. 1. It was used by many first- and second-generation computers.

The August 1972 issue of *Scientific American* contains two articles of interest, one on the origin of binary codes [Heath 72], and another [Gardner 72] on some entertaining aspects of the Gray codes.

Denoting the pixels of the original image by P_i and the pixels of the reconstructed image by Q_i (where $1 \le i \le n$), we first define the *mean square error* (MSE) between the two images as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(P_i - Q_i)^2. \tag{23.2}$$

It is the average of the square of the errors (pixel differences) of the two images. The *root mean square error* (RMSE) is defined as the square root of the MSE, and the PSNR is defined as

$$\text{PSNR} = 20\log_{10}\frac{\max_i |P_i|}{\text{RMSE}}, \tag{23.3}$$

The absolute value is normally not needed, since pixel values are rarely negative. For a bi-level image, the numerator is 1. For a grayscale image with eight bits per pixel, the numerator is 255. For color images, only the luminance component is used.

Greater resemblance between the images implies smaller RMSE and, as a result, larger PSNR. The PSNR is dimensionless, since the units of both numerator and denominator are pixel values. However, because of the use of the logarithm, we say that the PSNR is expressed in *decibels* (dB). The use of the logarithm also implies less sensitivity to changes in the RMSE. For example, dividing the RMSE by 10 multiplies the PSNR by 2. Notice that the PSNR has no absolute meaning. It is meaningless to say that a PSNR of, say, 25 is good. PSNR values are used only to compare the performance of different lossy compression methods or the effects of different parametric values on the performance of an algorithm. The MPEG committee, for example, uses an informal threshold of PSNR = 0.5 dB to decide whether to incorporate a coding optimization, since they believe that an improvement of that magnitude would be visible to the eye.

Typical PSNR values range between 20 and 40. Assuming pixel values in the range $[0, 255]$, an RMSE of 25.5 results in a PSNR of 20, and an RMSE of 2.55 results in a PSNR of 40. An RMSE of zero (i.e., identical images) results in an infinite (or, more precisely, undefined) PSNR. An RMSE of 255 results in a PSNR of zero, and RMSE values greater than 255 yield negative PSNRs.

◇ **Exercise 23.7:** If the maximum pixel value is 255, can the RMSE values be greater than 255?

Some authors define the PSNR as

$$\text{PSNR} = 10\log_{10}\frac{\max_i |P_i|^2}{\text{MSE}}.$$

In order for the two formulations to produce the same result, the logarithm is multiplied in this case by 10 instead of by 20, since $\log_{10} A^2 = 2\log_{10} A$. Either definition is useful, because only relative PSNR values are used in practice. However, the use of two different factors is confusing.

A related measure is *signal to noise ratio* (SNR). This is defined as

$$\text{SNR} = 20\log_{10}\frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n} P_i^2}}{\text{RMSE}}.$$

The numerator is the root mean square of the original image.

Figure 23.18 is a Matlab function to compute the PSNR of two images. A typical call is `PSNR(A,B)`, where `A` and `B` are image files. They must have the same resolution and have pixel values in the range $[0, 1]$.

```
function PSNR(A,B)
if A==B
 error('Images are identical; PSNR is undefined')
end
max2_A=max(max(A)); max2_B=max(max(B));
min2_A=min(min(A)); min2_B=min(min(B));
if max2_A>1 | max2_B>1 | min2_A<0 | min2_B<0
   error('pixels must be in [0,1]')
end
differ=A-B;
decib=20*log10(1/(sqrt(mean(mean(differ.^2)))));
disp(sprintf('PSNR = +%5.2f dB',decib))
```

Figure 23.18: A Matlab Function to Compute PSNR.

Another relative of the PSNR is the *signal to quantization noise ratio* (SQNR) This is a measure of the effect of quantization on signal quality. It is defined as

$$\text{SQNR} = 10\log_{10}\frac{\text{signal power}}{\text{quantization error}},$$

where the quantization error is the difference between the quantized signal and the original signal.

Another approach to the comparison of an original and a reconstructed image is to generate the difference image and judge it visually. Intuitively, the difference image is

$D_i = P_i - Q_i$, but such an image is hard to judge visually because its pixel values D_i tend to be small numbers. If a pixel value of zero represents white, such a difference image would be almost invisible. In the opposite case, where pixel values of zero represent black, such a difference would be too dark to judge. Better results are obtained by calculating

$$D_i = a(P_i - Q_i) + b,$$

where a is a magnification parameter (typically a small number such as 2) and b is half the maximum value of a pixel (typically 128). Parameter a serves to magnify small differences, while b shifts the difference image from extreme white (or extreme black) to a more comfortable gray.

23.5 Intuitive Methods

It is easy to come up with simple, intuitive methods for compressing images. They are generally inefficient but they illustrate how easy it is to come up with image compression algorithms.

23.5.1 Subsampling

Subsampling is perhaps the simplest way to compress an image. One approach to subsampling is simply to delete 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. Notice that the compression ratio is known in advance.

A slight improvement is obtained when the encoder calculates the average of each block of four pixels and writes this average on the compressed stream. No pixel is totally deleted, but the method is still primitive, because a good lossy image compression method should lose only data to which the eye is not sensitive.

Better results (but worse compression) are obtained when the color representation of the image is changed from the original (normally RGB) to luminance and chrominance. The encoder subsamples the two chrominance components of a pixel but not its luminance component. Assuming that each component uses the same number of bits, the two chrominance components use 2/3 of the image size. Subsampling them reduces this to 25% of 2/3, or 1/6. The size of the compressed image is therefore 1/3 (for the uncompressed luminance component), plus 1/6 (for the two chrominance components) or 1/2 of the original size.

23.5.2 Quantization

Scalar quantization has been mentioned in Section 23.3. This is an intuitive, lossy method where the information lost is not necessarily the least important. Vector quantization can obtain better results, and an intuitive version of it is described here.

The image is partitioned into equal-size blocks (called *vectors*) of pixels, and the encoder has a list (called a *codebook*) of blocks of the same size. Each image block B

is compared to all the blocks of the codebook and is matched with the "closest" one. If B is matched with codebook block C, then the encoder writes a pointer to C on the compressed stream. If the pointer is smaller than the block size, compression is achieved. Figure 23.19 shows an example.

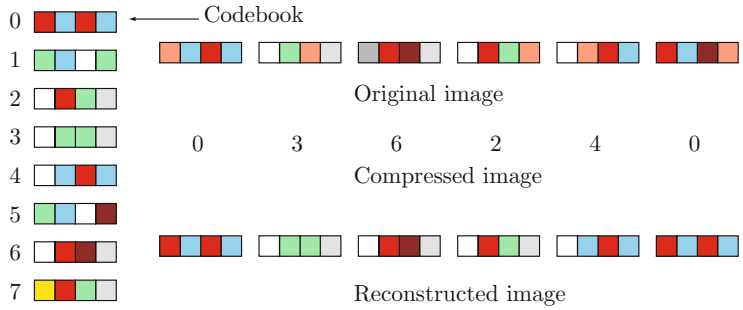


Figure 23.19: Intuitive Vector Quantization.

The details of selecting and maintaining the codebook and of matching blocks are discussed in [Salomon 09]. Notice that vector quantization is a method where the compression ratio is known in advance.

23.6 Variable-Length Codes

The remainder of this chapter is devoted to variable-length codes, a type of code that constitutes the basis of many image compression algorithms. Even methods that are based on other approaches, such as transforms or fractals, employ these codes as one of several steps in the algorithm. The following sections describe a few of the most-important variable-length codes used in image compression.

23.7 Codes, Fixed- and Variable-Length

A code is a symbol that stands for another symbol. At first, this idea seems pointless. Given a symbol S, what is the use of replacing it with another symbol Y? However, it is easy to find many important examples of the use of codes. Here are a few.

- Any language and any system of writing are codes. They provide us with symbols Y that we use in order to express our thoughts S.
- Acronyms and abbreviations can be considered codes. Thus, the string IBM is a symbol that stands for the much longer symbol "International Business Machines" and the well-known French university École Supérieure D'électricité is known to many simply as Supélec.

- Cryptography is the art and science of obfuscating messages. Before the age of computers, a message was typically a string of letters and was encrypted by replacing each letter with another letter or with a number. In the computer age, a message is a binary string (a bitstring) in a computer, and it is encrypted by replacing it with another bitstring, normally of the same length.

- Error control. Messages, even secret ones, are often transmitted over communications channels and may become damaged, corrupted, or garbled on their way from transmitter to receiver. We often experience low-quality, garbled telephone conversations. Even experienced pharmacists often find it difficult to read and understand a handwritten prescription. Computer data stored on magnetic disks may become corrupted because of exposure to magnetic fields or extreme temperatures. Music and videos recorded on optical discs (CDs and DVDs) may become unreadable because of scratches. In all these cases, it helps to augment the original data with error-control codes. Such codes—formally titled channel codes, but informally known as error-detecting or error-correcting codes—employ redundancy to detect and even correct, certain types of errors.

- ASCII and Unicode. These are character codes that make it possible to store characters of text as bitstrings in a computer. The ASCII code, which dates back to the 1960s [ascii-wiki 09], assigns 7-bit codes to 128 characters including 26 letters (upper- and lowercase), the 10 digits, certain punctuation marks, and several control characters. The Unicode project assigns 16-bit codes to many characters, and has a provision for even longer codes. The long codes make it possible to store and manipulate many thousands of characters, taken from many languages and alphabets (such as Greek, Cyrillic, Hebrew, Arabic, and Indic), and including punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, and dingbats.

The last example illustrates the use of codes in the field of computers and computations. Mathematically, a code is a mapping. Given an alphabet of symbols, a code maps individual symbols or strings of symbols to codewords, where a codeword is a string of bits, a bitstring. The process of mapping a symbol to a codeword is termed encoding and the reverse process is known as decoding.

Codes can have a fixed or variable length, and can be static or adaptive (dynamic). A static code is constructed once and never changes. ASCII and Unicode are examples of such codes. A static code can also have variable length, where short codewords are assigned to the commonly-occurring symbols. A variable-length, static code is normally designed based on the probabilities of the individual symbols. Each type of data has different probabilities and may benefit from a different code. The Huffman method (Section 23.13) is an example of an excellent variable-length, static code that can be constructed once the probabilities of all the symbols in the alphabet are known. In general, static codes that are also variable length can match well the lengths of individual codewords to the probabilities of the symbols. Notice that the code table must normally be included in the compressed file, because the decoder does not know the symbols' probabilities (the model of the data) and so has no way to construct the codewords independently.

A dynamic code varies over time, as more and more data is read and processed and more is known about the probabilities of the individual symbols. The dynamic (adaptive) Huffman algorithm [Salomon 09] is a method that employs such a code.

Fixed-length codes are known as block codes. They are easy to implement in software. It is easy to replace an original symbol with a fixed-length code, and it is equally easy to start with a string of such codes and break it up into individual codes that are then replaced by the original symbols.

There are cases where variable-length codes (VLCs) have obvious advantages. As their name implies, VLCs are codes that have different lengths. They are also known as variable-size codes. A set of such codes consists of short and long codewords. The following is a short list of important applications where such codes are commonly used.

- Data compression (or source coding). Given an alphabet of symbols where certain symbols occur often in messages, while other symbols are rare, it is possible to compress messages by assigning short codes to the common symbols and long codes to the rare symbols. This is an important application of variable-length codes.

- The Morse code for telegraphy, originated in the 1830s by Samuel Morse and Alfred Vail, exploits the same idea. It assigns short codes to commonly-occurring letters (the code of `E` is a dot and the code of `T` is a dash) and long codes to rare letters and punctuation marks (`--.-` to `Q`, `--..` to `Z`, and `--..--` to the comma).

- Processor design. Part of the architecture of any computer is an instruction set and a processor that fetches instructions from memory and executes them. It is easy to handle fixed-length instructions, but modern computers normally have instructions of different sizes. It is possible to reduce the overall size of programs by designing the instruction set such that commonly-used instructions are short. This also reduces the processor's power consumption and physical size and is especially important in embedded processors, such as processors designed for digital signal processing (DSP) or for mobile communication devices.

- Country calling codes. ITU-T recommendation E.164 is an international standard that assigns variable-length calling codes to many countries such that countries with many telephones are assigned short codes and countries with fewer telephones are assigned long codes. These codes also obey the prefix property (Section 23.8) which means that once a calling code C has been assigned, no other calling code will start with C.

- The International Standard Book Number (ISBN) is a unique number assigned to a book, to simplify inventory tracking by publishers and bookstores. The ISBN numbers are assigned according to an international standard known as ISO 2108 (1970). One component of an ISBN is a country code, that can be between one and five digits long. This code also obeys the prefix property. Once C has been assigned as a country code, no other country code will start with C.

- VCR Plus+ (also known as G-Code, VideoPlus+, and ShowView) is a prefix, variable-length code for programming video recorders. A unique number, a VCR Plus+, is computed for each television program by a proprietary algorithm from the date, time, and channel of the program. The number is published in television listings in newspapers and on the Internet. To record a program on a VCR, the number is located by the user and is typed into the video recorder. This programs the recorder to record the correct channel at the right time. This system was developed by Gemstar-TV Guide International [Gemstar 06].

> I gave up on new poetry myself thirty years ago, when most of it began to read like coded messages passing between lonely aliens on a hostile world.
>
> —Russell Baker.

23.8 Prefix Codes

Encoding a string of symbols a_i with VLCs is easy. No clever methods or algorithms are needed. The software reads the original symbols a_i one by one and replaces each a_i with its binary, variable-length code c_i. The codes are concatenated to form one (normally long) bitstring. The encoder either includes a table with all the pairs (a_i, c_i) or it executes a procedure to compute code c_i from the bits of symbol a_i.

Decoding is slightly more complex, because of the different lengths of the codes. When the decoder reads the individual bits of VLCs from a bitstring, it has to know either how long each code is or where each code ends. This is why a set of variable-length codes has to be carefully selected and why the decoder has to be taught about the codes. The decoder either has to have a table of all the valid codes, or it has to be told how to identify valid codes.

We start with a simple example. Given the set of four codes $a_1 = 0$, $a_2 = 01$, $a_3 = 011$, and $a_4 = 111$ we easily encode the message $a_2a_3a_3a_1a_2a_4$ as the bitstring 01|011|011|0|01|111. This string can be decoded unambiguously, but not easily. When the decoder inputs a 0, it knows that the next symbol is either a_1, a_2, or a_3, but the decoder has to input more bits to find out how many 1's follow the 0 before it can identify the next symbol. Similarly, given the bitstring $011\ldots111$, the decoder has to read the entire string and count the number of consecutive 1's before it finds out how many 1's follow the single 0 at the beginning. The codes could be $0|111\ldots$, $01|111\ldots$, or $011|111\ldots$. We say that such codes are not instantaneous.

In contrast, the following set of VLCs $a_1 = 0$, $a_2 = 10$, $a_3 = 110$, and $a_4 = 111$ is similar and is also instantaneous. Given a bitstring that consists of these codes, the decoder reads consecutive 1's until it has read three 1's (an a_4) or until it has read another 0. Depending on how many 1's precede the 0 (zero, one, or two 1's), the decoder knows whether the next symbol is a_1, a_2, or a_3. The 0 acts as a separator, which is why instantaneous codes are also known as comma codes. The rules that drive the decoder can be considered a finite automaton or a decision tree.

The next example is similar. We examine the set of VLCs $a_1 = 0$, $a_2 = 10$, $a_3 = 101$, and $a_4 = 111$. Only the code of a_3 is different, but a little experimenting shows that this set of VLCs is bad because it is not uniquely decodable (UD). Given the bitstring $0101111\ldots$, it can be decoded either as $a_1a_3a_4\ldots$ or $a_1a_2a_4\ldots$.

This observation is crucial because it points the way to the construction of large sets of VLCs. The set of codes above is bad because 10, the code of a_2, is also the prefix of the code of a_3. When the decoder reads $10\ldots$, it often cannot tell whether this is the code of a_2 or the start of the code of a_3.

Thus, a useful, practical set of VLCs has to be instantaneous and has to satisfy the following *prefix property*. Once a code c is assigned to a symbol, no other code should start with the bit pattern c. Prefix codes are also referred to as prefix-free codes, prefix condition codes, or instantaneous codes.

The following results can be proved: (1) A code is instantaneous if and only if it is a prefix code. (2) The set of UD codes is larger than the set of instantaneous codes (i.e., there are UD codes that are not instantaneous). (3) There is an instantaneous variable-length code with codeword lengths L_i if and only if there is a UD code with these codeword lengths.

The last of these results indicates that we cannot reduce the average word length of a variable-length code by using a UD code rather than an instantaneous code. Thus, there is no loss of compression performance if we restrict our selection of codes to instantaneous codes.

A UD code that consists of r codewords of lengths l_i must satisfy the Kraft inequality [Salomon 09], but this inequality does not require a prefix code. Thus, if a code satisfies the Kraft inequality it is UD, but if it is also a prefix code, then it is instantaneous. This feature of a UD code being also instantaneous, comes for free, because there is no need to add bits to the code and make it longer.

A prefix code (a set of codewords that satisfy the prefix property) is UD. Such a code is also *complete* if adding any codeword to it turns it into a non-UD code. A complete code is the largest UD code, but it also has a downside; it is less robust. If even a single bit is accidentally modified or deleted (or if a bit is somehow added) during storage or transmission, the decoder will lose synchronization and the rest of the transmission will be decoded incorrectly (see the discussion of robust codes in [Salomon 07]).

While discussing UD and non-UD codes, it is interesting to observe that the Morse code is non-UD (because, for example, the code of `I` is "." and the code of `H` is "...."), so Morse had to make it UD by requiring accurate relative timing.

23.9 VLCs for Integers

Following Elias, it is customary to denote the standard binary representation of the integer n by $\beta(n)$. This representation can be considered a code (the beta code), but it does not satisfy the prefix property (because, for example, $2 = 10_2$ is the prefix of $4 = 100_2$). The beta code has another disadvantage. Given a set of integers between 0 and n, we can represent each in $1+\lfloor\log_2 n\rfloor$ bits, a fixed-length representation. However, if the number of integers in the set is not known in advance (or if the largest integer is unknown), a fixed-length representation cannot be used and the natural solution is to assign variable-length codes to the integers. Any variable-length codes for integers should satisfy the following requirements:

1. Given an integer n, its codeword should be as short as possible and should be constructed from the magnitude, length, and bit pattern of n, without the need for any table lookups or other mappings.

2. Given a bitstream of variable-length codes, it should be easy to decode the next codeword and obtain an integer n even if n hasn't been seen before.

We will see that in many VLCs for integers, part of the binary representation of the integer is included in the codeword, and the rest of the codeword is side information indicating the length or precision of the encoded integer.

Several codes for the integers are described in the next few sections. Some can code only nonnegative integers and others can code only positive integers. A VLC for

positive integers can be extended to encode nonnegative integers by incrementing the integer before it is encoded and decrementing the result produced by decoding. A VLC for arbitrary integers can be obtained by a bijection, a mapping of the form

$$\begin{array}{cccccccccccc} 0 & -1 & 1 & -2 & 2 & -3 & 3 & -4 & 4 & -5 & 5 & \cdots \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & \cdots \end{array}$$

A function is bijective if it is one-to-one and onto.

Perhaps the simplest variable-length code for integers is the well-known unary code. The unary code of the positive integer n is constructed as $n-1$ bits of 1 followed by a single 0, or alternatively as $n-1$ zeros followed by a single 1 (the three left columns of Table 23.20). The length of the unary code for the integer n is therefore n bits. The two rightmost columns of Table 23.20 show how the unary code can be extended to encode the nonnegative integers (which makes the codes one bit longer). The unary code is simple to construct and is useful in many applications, but it is not universal. Stone-age people indicated the integer n by marking n adjacent vertical bars on a stone, which is why the unary code is sometimes known as a stone-age binary and each of its n or $(n-1)$ 1's (or n or $(n-1)$ zeros) is termed a stone-age bit.

Stone Age Binary?

n	Code	Reverse	Alt. code	Alt. reverse
0	–	–	0	1
1	0	1	10	01
2	10	01	110	001
3	110	001	1110	0001
4	1110	0001	11110	00001
5	11110	00001	111110	000001

Table 23.20: Some Unary Codes.

It is easy to see that the unary code satisfies the prefix property, so it is instantaneous and can be used as a variable-length code. Since its length L satisfies $L = n$, we get $2^{-L} = 2^{-n}$, so it makes sense to use this code in cases where the input data consists of integers n with exponential probabilities $P(n) \approx 2^{-n}$. Given data that lends itself to the use of the unary code (i.e., a set of integers that satisfy $P(n) \approx 2^{-n}$), we can assign unary codes to the integers and these codes will be as good as the Huffman codes with the advantage that the unary codes are trivial to encode and decode. In general, the unary code is used as part of other, more sophisticated, variable-length codes.

Example: Table 23.21 lists the integers 1 through 6 with probabilities $P(n) = 2^{-n}$, except that $P(6) = 2^{-5} \approx 2^{-6}$. The table lists the unary codes and Huffman codes for the six integers, and it is obvious that these codes have the same lengths (except the code of 6, because this symbol does not have the correct probability).

Every positive number was one of Ramanujan's personal friends.
—J. E. Littlewood.

n	Prob.	Unary	Huffman
1	2^{-1}	0	0
2	2^{-2}	10	10
3	2^{-3}	110	110
4	2^{-4}	1110	1110
5	2^{-5}	11110	11110
6	2^{-5}	111110	11111

Table 23.21: Six Unary and Huffman Codes.

23.10 Start-Step-Stop Codes

The unary code is ideal for compressing data that consists of integers n with probabilities $P(n) \approx 2^{-n}$. If the data to be compressed consists of integers with different probabilities, it may benefit from one of the general unary codes (also known as start-step-stop codes). Such a code, proposed by [Fiala and Greene 89], depends on a triplet (start, step, stop) of nonnegative integer parameters. A set of such codes is constructed subset by subset as follows:

1. Set $n = 0$.
2. Set $a = \text{start} + n \times \text{step}$.
3. Construct the subset of codes that start with n leading 1's, followed by a single intercalary bit (separator) of 0, followed by a combination of a bits. There are 2^a such codes.
4. Increment n by step. If $n <$ stop, go to step 2. If $n >$ stop, issue an error and stop. If $n =$ stop, repeat steps 2 and 3 but without the single intercalary 0 bit of step 3, and stop.

This construction makes it obvious that the three parameters have to be selected such that start $+ n \times$ step will reach "stop" for some nonnegative n. The number of codes for a given triplet is normally finite and depends on the choice of parameters. It is given by

$$\frac{2^{\text{stop}+\text{step}} - 2^{\text{start}}}{2^{\text{step}} - 1}.$$

Notice that this expression increases exponentially with parameter "stop," so large sets of these codes can be generated even with small values of the three parameters. Notice that the case step $= 0$ results in a zero denominator and thus in an infinite set of codes.

Tables 23.22 and 23.23 show the 680 codes of (3,2,9) and the 2044 codes of (2,1,10). Table 23.24 lists the number of codes of each of the general unary codes $(2, 1, k)$ for $k = 2, 3, \ldots, 11$. This table was calculated by the *Mathematica* command `Table[2^(k+1)-4,{k,2,11}]`.

Examples:

1. The triplet $(n, 1, n)$ defines the standard (beta) n-bit binary codes, as can be verified by direct construction. The number of such codes is easily seen to be

$$\frac{2^{n+1} - 2^n}{2^1 - 1} = 2^n.$$

n	$a = 3 + n \cdot 2$	nth codeword	Number of codewords	Range of integers
0	3	$0xxx$	$2^3 = 8$	0–7
1	5	$10xxxx$	$2^5 = 32$	8–39
2	7	$110xxxxxx$	$2^7 = 128$	40–167
3	9	$111xxxxxxxx$	$2^9 = 512$	168–679
		Total	680	

Table 23.22: The General Unary Code (3,2,9).

n	$a = 2 + n \cdot 1$	nth codeword	Number of codewords	Range of integers
0	2	$0xx$	4	0–3
1	3	$10xxx$	8	4–11
2	4	$110xxxx$	16	12–27
3	5	$1110xxxxx$	32	28–59
	...	...		...
8	10	$\underbrace{11...1}_{8}\underbrace{xx...x}_{10}$	1024	1020–2043
		Total	2044	

Table 23.23: The General Unary Code (2,1,10).

k :	2	3	4	5	6	7	8	9	10	11
$(2,1,k)$:	4	12	28	60	124	252	508	1020	2044	4092

Table 23.24: Number of General Unary Codes $(2,1,k)$ for $k = 2, 3, \ldots, 11$.

2. The triplet $(0, 0, \infty)$ defines the codes 0, 10, 110, 1110,... which are the unary codes but assigned to the integers 0, 1, 2,... instead of 1, 2, 3,... .

3. The triplet $(0, 1, \infty)$ generates a variance of the Elias gamma code.

4. The triplet (k, k, ∞) generates another variance of the Elias gamma code.

5. The triplet $(k, 0, \infty)$ generates the Rice code [Salomon 09] with parameter k.

6. The triplet $(s, 1, \infty)$ generates the exponential Golomb codes [Salomon 09].

7. The triplet $(1, 1, 30)$ produces $(2^{30} - 2^1)/(2^1 - 1) \approx$ a billion codes.

8. Table 23.25 shows the general unary code for (10,2,14). There are only three code lengths since "start" and "stop" are so close, but there are many codes because "start" is large.

The start-step-stop codes are very general. Often, a person gets an idea for a variable-length code only to find out that it is a special case of a start-step-stop code. Here is a typical example. We can design a variable-length code where each code consists of triplets. The first two bits of a triplet go through the four possible values 00, 01, 10, and 11, and the rightmost bit acts as a stop bit. If it is 1, we stop reading the code, otherwise we read another triplet of bits. Table 23.26 lists the first 22 codes.

To see why this is a start-step-stop code, we rewrite it by placing the stop bits on

n	$a = 10 + n \cdot 2$	nth codeword	Number of codewords	Range of integers
0	10	$0\underbrace{x...x}_{10}$	$2^{10} = 1K$	0–1023
1	12	$10\underbrace{xx...x}_{12}$	$2^{12} = 4K$	1024–5119
2	14	$11\underbrace{xx...xx}_{14}$	$2^{14} = 16K$	5120–21503
		Total	21504	

Table 23.25: The General Unary Code (10,2,14).

n	Codes	n	Codes
0	001	11	010 111
1	011	12	100 001
2	101	13	100 011
3	111	14	100 101
4	000 001	15	100 111
5	000 011	16	110 001
6	000 101	17	110 011
7	000 111	18	110 101
8	010 001	19	110 111
9	010 011	20	000 000 001
10	010 101	21	000 000 011

Table 23.26: Triplet-Based Codes.

n	Codes	n	Codes
0	1 00	11	01 01 11
1	1 01	12	01 10 00
2	1 10	13	01 10 01
3	1 11	14	01 10 10
4	01 00 00	15	01 10 11
5	01 00 01	16	01 11 00
6	01 00 10	17	01 11 01
7	01 00 11	18	01 11 10
8	01 01 00	19	01 11 11
9	01 01 01	20	001 00 00 00
10	01 01 10	21	001 00 00 01

Table 23.27: Same Codes Rearranged.

the left end, followed by pairs of code bits, as listed in Table 23.27. Examining these 22 codes shows that the code for the integer n starts with a unary code of j zeros followed by a single 1, followed by $j + 1$ pairs of code bits. There are four codes of length 3 (a 1-bit unary followed by a pair of code bits, corresponding to $j = 0$), 16 6-bit codes (a 2-bit unary followed by two pairs of code bits, corresponding to $j = 1$), 64 9-bit codes (corresponding to $j = 2$), and so on. A little tinkering shows that this is the start-step-stop code $(2, 2, \infty)$.

23.11 Start/Stop Codes

The start-step-stop codes are flexible. By carefully adjusting the values of the three parameters it is possible to construct sets of codes of many different lengths. However, the lengths of these codes are restricted to the values $n + 1 + \text{start} + n \times \text{step}$ (except for the last subset where the codes have lengths $\text{stop} + \text{start} + \text{stop} \times \text{step}$). The start/stop codes of this section were conceived by Steven Pigeon and are described in [Pigeon 01a,b], where it is shown that they are universal. A set of these codes is fully specified by an array of nonnegative integer parameters $(m_0, m_1, \ldots, m_t)$ and is constructed in subsets, similar to the start-step-stop codes, in the following steps:

1. Set $i = 0$ and $a = m_0$.

2. Construct the subset of codes that start with i leading 1's, followed by a single separator of 0, followed by a combination of a bits. There are 2^a such codes.

3. Increment i by 1 and set $a \leftarrow a + m_i$.

4. If $i < t$, go to step 2. Otherwise ($i = t$), repeat step 2 but without the single 0 intercalary, and stop.

Thus, the parameter array $(0, 3, 1, 2)$ results in the set of codes listed in Table 23.28.

i	a	Codeword	# of codes	Length
0	2	$0xx$	4	3
1	5	$10xxxx$	32	7
2	6	$110xxxxx$	64	9
3	8	$111xxxxxxx$	256	11

Table 23.28: The Start/Stop Code (2,3,1,2).

The maximum code length is $t + m_0 + \cdots + m_t$ and on average the start/stop code of an integer s is never longer than $\lceil \log_2 s \rceil$ (the length of the binary representation of s). If an optimal set of such codes is constructed by an encoder and is used to compress a data file, then the only side information needed in the compressed file is the value of t and the array of $t + 1$ parameters m_i (which are mostly small integers). This is considerably less than the size of a Huffman tree or the side information required by many other compression methods.

The start/stop codes can also encode an indefinite number of arbitrarily large integers. Simply set all the parameters to the same value and set t to infinity.

Steven Pigeon, the developer of these codes, shows that the parameters of the start/stop codes can be selected such that the resulting set of codes will have an average length shorter than what can be achieved with the start-step-stop codes for the same probability distribution. He also shows how the probability distribution can be employed to determine the best set of parameters for the code. In addition, the number of codes in the set can be selected as needed, in contrast to the start-step-stop codes that often result in more codes than needed.

> The Human Brain starts working the moment you are born and never stops until you stand up to speak in public!
>
> —George Jessel.

23.12 Elias Codes

In his pioneering work [Elias 75], Peter Elias described three useful prefix codes. The main idea of these codes is to prefix the integer being encoded with an encoded representation of its order of magnitude. For example, for any positive integer n there is an integer M such that $2^M \le n < 2^{M+1}$. We can therefore write $n = 2^M + L$ where L is at most M bits long, and generate a code that consists of M and L. The problem is to determine the length of M and this is solved in different ways by the various Elias codes. Elias denoted the unary code of n by $\alpha(n)$ and the standard binary representation of n, from its most significant 1, by $\beta(n)$. His first code was therefore designated γ (gamma).

The Elias gamma code $\gamma(n)$ for positive integers n is simple to encode and decode and is also universal.

Encoding. Given a positive integer n, perform the following steps:

1. Denote by M the length of the binary representation $\beta(n)$ of n.
2. Prepend $M-1$ zeros to it (i.e., the $\alpha(n)$ code without its terminating 1).

Step 2 amounts to prepending the length of the code to the code, in order to ensure unique decodability.

The length M of the integer n is

$$1 + \lfloor \log_2 n \rfloor \tag{23.4},$$

so the length of $\gamma(n)$ is

$$2M - 1 = 2\lfloor \log_2 n \rfloor + 1. \tag{23.5}$$

We later show that this code is ideal for applications where the probability of n is $1/(2n^2)$.

An alternative construction of the gamma code is as follows:

1. Find the largest integer N such that $2^N \le n < 2^{N+1}$ and write $n = 2^N + L$. Notice that L is at most an N-bit integer.
2. Encode N in unary either as N zeros followed by a 1 or N 1's followed by a 0.
3. Append L as an N-bit number to this representation of N.

Peter Elias

$1 = 2^0 + 0 = 1$	$10 = 2^3 + 2 = 0001010$
$2 = 2^1 + 0 = 010$	$11 = 2^3 + 3 = 0001011$
$3 = 2^1 + 1 = 011$	$12 = 2^3 + 4 = 0001100$
$4 = 2^2 + 0 = 00100$	$13 = 2^3 + 5 = 0001101$
$5 = 2^2 + 1 = 00101$	$14 = 2^3 + 6 = 0001110$
$6 = 2^2 + 2 = 00110$	$15 = 2^3 + 7 = 0001111$
$7 = 2^2 + 3 = 00111$	$16 = 2^4 + 0 = 000010000$
$8 = 2^3 + 0 = 0001000$	$17 = 2^4 + 1 = 000010001$
$9 = 2^3 + 1 = 0001001$	$18 = 2^4 + 2 = 000010010$

Table 23.29: 18 Elias Gamma Codes.

Table 23.29 lists the first 18 gamma codes, where the L part is in italics.

In his 1975 paper, Elias describes two versions of the gamma code. The first version (titled γ) is encoded as follows:

1. Generate the binary representation $\beta(n)$ of n.
2. Denote the length $|\beta(n)|$ of $\beta(n)$ by M.
3. Generate the unary $u(M)$ representation of M as $M-1$ zeros followed by a 1.
4. Follow each bit of $\beta(n)$ by a bit of $u(M)$.
5. Drop the leftmost bit (the leftmost bit of $\beta(n)$ is always 1).

Thus, for $n=13$ we prepare $\beta(13)=\bar{1}\bar{1}\bar{0}\bar{1}$, so $M=4$ and $u(4)=0001$, resulting in $\bar{1}0\bar{1}0\bar{0}0\bar{1}1$. The final code is $\gamma(13)=0\bar{1}0\bar{0}0\bar{1}1$. In general, the length of the gamma code for the integer i is $1+2\lfloor\log_2 i\rfloor$.

The second version, dubbed γ', moves the bits of $u(M)$ to the left. Thus $\gamma'(13)=0001|\bar{1}\bar{0}\bar{1}$. The gamma codes of Table 23.29 are Elias's γ' codes. Both gamma versions are universal.

Decoding is also simple and is done in two steps:

1. Read zeros from the code until a 1 is encountered. Denote the number of zeros by N.
2. Read the next N bits as an integer L. Compute $n=2^N+L$.

It is easy to see that this code can be used to encode positive integers even in cases where the largest integer is not known in advance. Also, this code grows slowly (see Figure 23.31), so it is a good candidate for compressing integer data where small integers are common and large integers are rare.

It is easy to understand why the gamma code (and in general, all variable-length codes) should be used only when it is known or estimated that the distribution of the integers to be encoded is close to the ideal distribution for the given code. The simple computation here assumes that the first n integers are given and are distributed uniformly (i.e., each appears with the same probability). We compute the average gamma code length and show that it is much larger than the length of the fixed-size binary codes of the same integers.

The length of the Elias gamma code for the integer i is $1+2\lfloor\log_2 i\rfloor$. Thus, the average of the gamma codes of the first n integers is

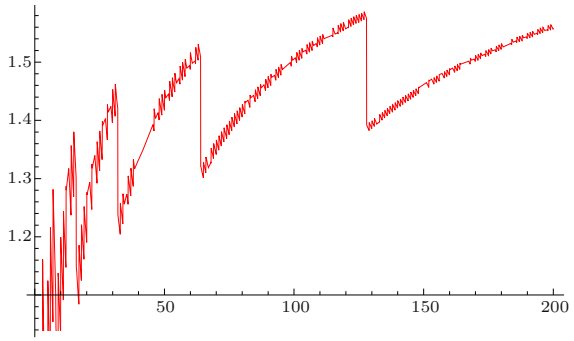
$$a=\frac{n+2\sum_{i=1}^{n}\lfloor\log_2 i\rfloor}{n}.$$

The length of the fixed-size codes of the same integers is $b=\lceil\log_2 n\rceil$. Figure 23.30 is a plot of the ratio a/b and it is easy to see that this value is greater than 1, indicating that for flat distributions of the integers the fixed-length (beta) code is better than the gamma code. The curve in the figure generally goes up, but features sudden drops (where it remains greater than 1) at points where $n=2^k+1$ for integer k.

The Elias delta code. In his gamma code, Elias prepends the length of the code in unary (α). In his next code, δ (delta), he prepends the length in binary (β). Thus, the Elias delta code, also for the positive integers, is slightly more complex to construct.

Encoding a positive integer n, is done in the following steps:

1. Write n in binary. The leftmost (most significant) bit will be a 1.




```
gamma[i_] := 1. + 2 Floor[Log[2, i]];
Plot[Sum[gamma[j], {j,1,n}]/(n Ceiling[Log[2,n]]), {n,1,200}]
```

Figure 23.30: Gamma Code Versus Binary Code.

2. Count the bits, remove the leftmost bit of n, and prepend the count, in binary, to what is left of n after its leftmost bit has been removed.

3. Subtract 1 from the count of step 2 and prepend that number of zeros to the code.

When these steps are applied to the integer 17, the results are: $17 = 10001_2$ (five bits). Remove the leftmost 1 and prepend $5 = 101_2$ yields 101|0001. Three bits were added, so we prepend two zeros to obtain the delta code 00|101|0001.

To compute the length of the delta code of n, we notice that step 1 generates (from Equation (23.4)) $M = 1 + \lfloor \log_2 n \rfloor$ bits. For simplicity, we omit the $\lfloor$ and $\rfloor$ and observe that

$$M = 1 + \log_2 n = \log_2 2 + \log_2 n = \log_2(2n).$$

The count of step 2 is M, whose length C is therefore $C = 1+\log_2 M = 1+\log_2(\log_2(2n))$ bits. Step 2 therefore prepends C bits and removes the leftmost bit of n. Step 3 prepends $C - 1 = \log_2 M = \log_2(\log_2(2n))$ zeros. The total length of the delta code is therefore the three-part sum

$$\underbrace{\log_2(2n)}_{\text{step 1}} + \underbrace{[1 + \log_2 \log_2(2n)] - 1}_{\text{step 2}} + \underbrace{\log_2 \log_2(2n)}_{\text{step 3}} = 1 + \lfloor \log_2 n \rfloor + 2\lfloor \log_2 \log_2(2n) \rfloor. \tag{23.6}$$

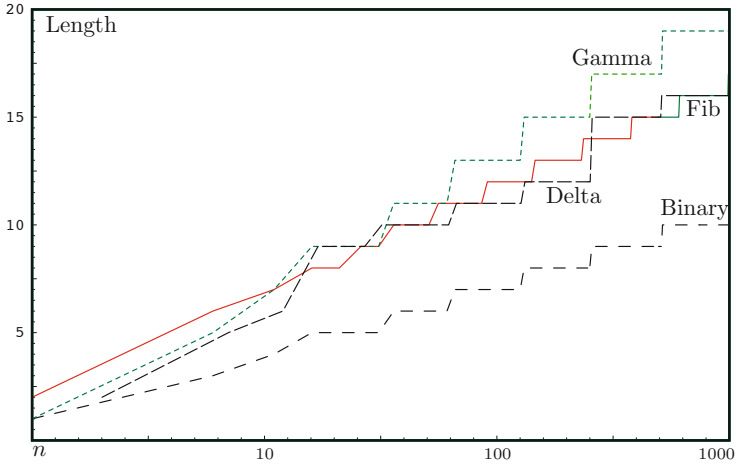
Figure 23.31 illustrates the length graphically (the Fibonacci code is described in [Salomon 07]). We show below that the delta code is ideal for data where the integer n occurs with probability $1/[2n(\log_2(2n))^2]$.

An equivalent way to construct the delta code employs the gamma code:

1. Find the largest integer N such that $2^N \geq n < 2^{N+1}$ and write $n = 2^N + L$. Notice that L is at most an N-bit integer.

2. Encode $N + 1$ with the Elias gamma code.

3. Append the binary value of L, as an N-bit integer, to the result of step 2.




```
(* Plot the lengths of four codes
    1. staircase plots of binary representation *)
bin[i_] := 1 + Floor[Log[2, i]];
Table[{Log[10, n], bin[n]}, {n, 1, 1000, 5}];
g1 = ListPlot[%, AxesOrigin -> {0, 0},  PlotJoined -> True,
     PlotStyle -> { AbsoluteDashing[{5, 5}]}]
(* 2. staircase plot of Fibonacci code length *)
fib[i_] := 1 + Floor[ Log[1.618, Sqrt[5] i]];
Table[{Log[10, n], fib[n]}, {n, 1, 1000, 5}];
g2 = ListPlot[%, AxesOrigin -> {0, 0},  PlotJoined -> True]
(* 3. staircase plot of gamma code length*)
gam[i_] := 1 + 2Floor[Log[2, i]];
Table[{Log[10, n], gam[n]}, {n, 1, 1000, 5}];
g3 = ListPlot[%, AxesOrigin -> {0, 0},  PlotJoined -> True,
     PlotStyle -> { AbsoluteDashing[{2, 2}]}]
(* 4. staircase plot of delta code length*)
del[i_] := 1 + Floor[Log[2, i]] + 2Floor[Log[2, Log[2, i]]];
Table[{Log[10, n], del[n]}, {n, 2, 1000, 5}];
g4 = ListPlot[%, AxesOrigin -> {0, 0},  PlotJoined -> True,
     PlotStyle -> { AbsoluteDashing[{6, 2}]}]
Show[g1, g2, g3, g4, PlotRange -> {{0, 3}, {0, 20}}]
```

Figure 23.31: Lengths of Binary, Fibonacci, and Two Elias Codes.

When these steps are applied to $n = 17$, the results are: $17 = 2^N + L = 2^4 + 1$. The gamma code of $N + 1 = 5$ is 00101, and appending $L = 0001$ to this yields 00101|0001.

Table 23.32 lists the first 18 delta codes, where the L part is in italics.

Decoding is done in the following steps:

1. Read bits from the code until you can decode an Elias gamma code. Call the decoded result $M + 1$. This is done in the following substeps:

1.1 Count the leading zeros of the code and denote the count by C.

1.2 Examine the leftmost $2C + 1$ bits (C zeros, followed by a single 1, followed by C more bits). This is the decoded gamma code $M + 1$.

2. Read the next M bits. Call this number L.

3. The decoded integer is $2^M + L$.

In the case of $n = 17$, the delta code is 001010001. We skip two zeros, so $C = 2$.

$1 = 2^0 + 0 \to \lvert L\rvert = 0 \to 1$	$10 = 2^3 + 2 \to \lvert L\rvert = 3 \to 00100\mathit{010}$
$2 = 2^1 + 0 \to \lvert L\rvert = 1 \to 010\mathit{0}$	$11 = 2^3 + 3 \to \lvert L\rvert = 3 \to 00100\mathit{011}$
$3 = 2^1 + 1 \to \lvert L\rvert = 1 \to 010\mathit{1}$	$12 = 2^3 + 4 \to \lvert L\rvert = 3 \to 00100\mathit{100}$
$4 = 2^2 + 0 \to \lvert L\rvert = 2 \to 011\mathit{00}$	$13 = 2^3 + 5 \to \lvert L\rvert = 3 \to 00100\mathit{101}$
$5 = 2^2 + 1 \to \lvert L\rvert = 2 \to 011\mathit{01}$	$14 = 2^3 + 6 \to \lvert L\rvert = 3 \to 00100\mathit{110}$
$6 = 2^2 + 2 \to \lvert L\rvert = 2 \to 011\mathit{10}$	$15 = 2^3 + 7 \to \lvert L\rvert = 3 \to 00100\mathit{111}$
$7 = 2^2 + 3 \to \lvert L\rvert = 2 \to 011\mathit{11}$	$16 = 2^4 + 0 \to \lvert L\rvert = 4 \to 00101\mathit{0000}$
$8 = 2^3 + 0 \to \lvert L\rvert = 3 \to 00100\mathit{000}$	$17 = 2^4 + 1 \to \lvert L\rvert = 4 \to 00101\mathit{0001}$
$9 = 2^3 + 1 \to \lvert L\rvert = 3 \to 00100\mathit{001}$	$18 = 2^4 + 2 \to \lvert L\rvert = 4 \to 00101\mathit{0010}$

Table 23.32: 18 Elias Delta Codes.

The value of the leftmost $2C + 1 = 5$ bits is $00101 = 5$, so $M + 1 = 5$. We read the next $M = 4$ bits 0001, and end up with the decoded value $2^M + L = 2^4 + 1 = 17$.

To better understand the application and performance of these codes, we need to identify the type of data it compresses best. Given a set of symbols a_i, where each symbol occurs in the data with probability P_i and the length of its code is l_i bits, the average code length is the sum $\sum P_i l_i$ and the entropy (the smallest number of bits required to represent the symbols) is $\sum[-P_i \log_2 P_i]$. The redundancy is the difference $\sum_i P_i l_i - \sum_i[-P_i \log_2 P_i]$ and we are looking for probabilites P_i that will minimize this difference.

In the case of the gamma code, $l_i = 1 + 2\lfloor \log_2 i \rfloor$. If we select symbol probabilities $P_i = 1/(2i^2)$ (a power law distribution of probabilities, where the first 10 values are 0.5, 0.125, 0.0556, 0.03125, 0.02, 0.01389, 0.0102, 0.0078, 0.00617, and 0.005), both the average code length and the entropy become the identical sums

$$\sum_i \frac{1 + 2\log i}{2i^2},$$

indicating that the gamma code is asymptotically optimal for this type of data. A power law distribution of values is dominated by just a few symbols and especially by the first. Such a distribution is very skewed and is therefore handled very well by the gamma code which starts very short. In an exponential distribution, in contrast, the small values have similar probabilities, which is why data with this type of statistical distribution is compressed better by a Rice code [Salomon 07].

In the case of the delta code, $l_i = 1 + \log i + 2\log\log(2i)$. If we select symbol probabilities $P_i = 1/[2i(\log(2i))^2]$ (where the first five values are 0.5, 0.0625, 0.025, 0.0139, and 0.009), both the average code length and the entropy become the identical sums

$$\sum_i \frac{\log 2 + \log i + 2\log\log(2i)}{2i(\log(2i))^2},$$

indicating that the redundancy is zero and the delta code is therefore asymptotically optimal for this type of data.

> The phrase "working mother" is redundant.
> —Jane Sellman.

The Elias omega code. Unlike the previous Elias codes, the omega code uses itself recursively to encode the prefix M, which is why it is sometimes referred to as a recursive Elias code. The main idea is to prepend the length of n to n as a group of bits that starts with a 1, then prepend the length of the length, as another group, to the result, and continue prepending lengths until the last length is 2 or 3 (and therefore fits in two bits). In order to distinguish between a length group and the last, rightmost group (of n itself), the latter is followed by a delimiter of 0, while each length group starts with a 1.

Encoding a positive integer n is done recursively in the following steps:

1. Initialize the code-so-far to 0.
2. If the number to be encoded is 1, stop; otherwise, prepend the binary representation of n to the code-so-far. Assume that we have prepended L bits.
3. Repeat step 2, with the binary representation of $L-1$ instead of n.

The integer 17 is therefore encoded by (1) a single 0, (2) prepended by the 5-bit binary value 10001, (3) prepended by the 3-bit value of $5-1=100_2$, and (4) prepended by the 2-bit value of $3-1=10_2$. The result is 10|100|10001|0.

Table 23.33 lists the first 18 omega codes. Note that $n=1$ is handled as a special case.

1	0	10	11 1010 0
2	10 0	11	11 1011 0
3	11 0	12	11 1100 0
4	10 100 0	13	11 1101 0
5	10 101 0	14	11 1110 0
6	10 110 0	15	11 1111 0
7	10 111 0	16	10 100 10000 0
8	11 1000 0	17	10 100 10001 0
9	11 1001 0	18	10 100 10010 0

Table 23.33: 18 Elias Omega Codes.

Decoding is done in several nonrecursive steps where each step reads a group of bits from the code. A group that starts with a zero signals the end of decoding.

1. Initialize n to 1.
2. Read the next bit. If it is 0, stop. Otherwise read n more bits, assign the group of $n+1$ bits to n, and repeat this step.

Some readers may find it easier to understand these steps rephrased as follows.

1. Read the first group, which will either be a single 0, or a 1 followed by n more digits. If the group is a 0, the value of the integer is 1; if the group starts with a 1, then n becomes the value of the group interpreted as a binary number.
2. Read each successive group; it will either be a single 0, or a 1 followed by n more digits. If the group is a 0, the value of the integer is n; if it starts with a 1, then n becomes the value of the group interpreted as a binary number.

Example: Decode 10|100|10001|0. The decoder initializes $n = 1$ and reads the first bit. It is a 1, so it reads $n = 1$ more bit (0) and assigns $n = 10_2 = 2$. It reads the next bit. It is a 1, so it reads $n = 2$ more bits (00) and assigns the group 100 to n. It reads the next bit. It is a 1, so it reads four more bits (0001) and assigns the group 10001 to n. The next bit read is 0, indicating the end of decoding.

The omega code is constructed recursively, which is why its length $|\omega(n)|$ can also be computed recursively. We define the quantity $l^k(n)$ recursively by $l^1(n) = \lfloor \log_2 n \rfloor$ and $l^{i+1}(n) = l^1(l^i(n))$. We already know that $|\beta(n)| = l^1(n) + 1$ (where β is the standard binary representation), and this implies that the length of the omega code is given by the sum

$$|\omega(n)| = \sum_{i=1}^{k} \beta(l^{k-i}(n)) + 1 = 1 + \sum_{i=1}^{k} (l^i(n) + 1),$$

where the sum stops at the k that satisfies $l^k(n) = 1$. From this, Elias concludes that the length satisfies $|\omega(n)| \le 1 + \frac{5}{2}\lfloor \log_2 n \rfloor$.

A quick glance at any table of these codes shows that their lengths fluctuate. In general, the length increases slowly as n increases, but when a new length group is added, which happens when $n = 2^{2^k}$ for any positive integer k, the length of the code increases suddenly by several bits. For k values of 1, 2, 3, and 4, this happens when n reaches 4, 16, 256, and 65,536. Because the groups of lengths are of the form "length," "log(length)," "log(log(length))," and so on, the omega code is sometimes referred to as a logarithmic-ramp code.

Table 23.34 compares the length of the gamma, delta, and omega codes. It shows that the delta code is asymptotically best, but if the data consists mostly of small numbers (less than 8) and there are only a few large integers, then the gamma code performs better.

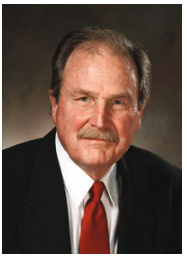
Values	Gamma	Delta	Omega
1	1	1	2
2	3	4	3
3	3	4	4
4	5	5	4
5–7	5	5	5
8–15	7	8	6–7
16–31	9	9	7–8
32–63	11	10	8–10
64–88	13	11	10
100	13	11	11
1000	19	16	16
10^4	27	20	20
10^5	33	25	25
10^5	39	28	30

Table 23.34: Lengths of Three Elias Codes.

> Beware of bugs in the above code; I have only proved it correct, not tried it.
> —Donald Knuth.

23.13 Huffman Coding

David Huffman (1925–1999)



Being originally from Ohio, it is no wonder that Huffman went to Ohio State University for his BS (in electrical engineering). What is unusual was his age (18) when he earned it in 1944. After serving in the United States Navy, he went back to Ohio State for an MS degree (1949) and then to MIT, for a PhD (1953, electrical engineering).

That same year, Huffman joined the faculty at MIT. In 1967, he made his only career move when he went to the University of California, Santa Cruz as the founding faculty member of the Computer Science Department. During his long tenure at UCSC, Huffman played a major role in the development of the department (he served as chair from 1970 to 1973) and he is known for his motto "my products are my students." Even after his retirement, in 1994, he remained active in the department, teaching information theory and signal analysis courses.

Huffman made significant contributions in several areas, mostly information theory and coding, signal designs for radar and communications, and design procedures for asynchronous logical circuits. Of special interest is the well-known Huffman algorithm for constructing a set of optimal prefix codes for data with known frequencies of occurrence. At a certain point he became interested in the mathematical properties of "zero curvature" surfaces, and developed this interest into techniques for folding paper into unusual sculptured shapes (the so-called computational origami).

Huffman coding is a popular method for data compression. It serves as the basis for several popular programs run on various platforms. Some programs use just the Huffman method, while others use it as one step in a multistep compression process. The Huffman method [Huffman 52] is somewhat similar to the Shannon–Fano method. It generally produces better codes, and like the Shannon–Fano method, it produces the 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).

Since its development in 1952 by D. Huffman, this method has been the subject of intensive research in data compression. The long discussion in [Gilbert and Moore 59] proves that the Huffman code is a minimum-length code in the sense that no other encoding has a shorter average length. An algebraic approach to constructing the Huffman code is introduced in [Karp 61]. In [Gallager 78], Robert Gallager shows that the redundancy of Huffman coding is at most $p_1 + 0.086$ where p_1 is the probability of the most

common symbol in the alphabet. The redundancy is the difference between the average Huffman codeword length and the entropy. Given a large alphabet, such as the set of letters, digits, and punctuation marks used by a natural language, the largest symbol probability is typically around 15–20%, bringing the value of the quantity $p_1 + 0.086$ to around 0.1. This means that Huffman codes are at most 0.1 bit longer (per symbol) than an ideal entropy encoder, such as arithmetic coding.

The Huffman algorithm starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up. This is done in steps, where at each step the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codes of the symbols.

> Truth is stranger than fiction, but this is because fiction is obliged to stick to probability; truth is not.
>
> —Anonymous.

This process is best illustrated by an example. Given five symbols with probabilities as shown in Figure 23.35a, they are paired in the following order:

1. a_4 is combined with a_5 and both are replaced by the combined symbol a_{45}, whose probability is 0.2.
2. There are now four symbols left, a_1, with probability 0.4, and a_2, a_3, and a_{45}, with probabilities 0.2 each. We arbitrarily select a_3 and a_{45}, combine them, and replace them with the auxiliary symbol a_{345}, whose probability is 0.4.
3. Three symbols are now left, a_1, a_2, and a_{345}, with probabilities 0.4, 0.2, and 0.4, respectively. We arbitrarily select a_2 and a_{345}, combine them, and replace them with the auxiliary symbol a_{2345}, whose probability is 0.6.
4. Finally, we combine the two remaining symbols, a_1 and a_{2345}, and replace them with a_{12345} with probability 1.

The tree is now complete. It is shown in Figure 23.35a "lying on its side" with its root on the right and its five leaves on the left. To assign the codes, we arbitrarily assign a bit of 1 to the top edge, and a bit of 0 to the bottom edge, of every pair of edges. This results in the codes 0, 10, 111, 1101, and 1100. The assignments of bits to the edges is arbitrary.

The average size of this code is $0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2$ bits/symbol, but even more importantly, the Huffman code is not unique. Some of the steps above were chosen arbitrarily, since there were more than two symbols with smallest probabilities. Figure 23.35b shows how the same five symbols can be combined differently to obtain a different Huffman code (11, 01, 00, 101, and 100). The average size of this code is $0.4 \times 2 + 0.2 \times 2 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 2.2$ bits/symbol, the same as the previous code.

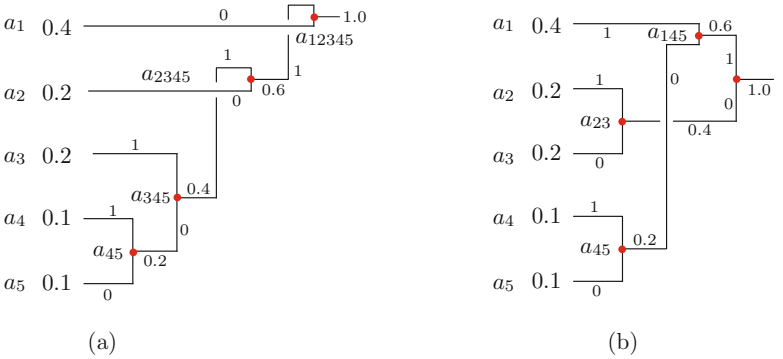


Figure 23.35: Huffman Codes.

◇ **Exercise 23.8:** Given the eight symbols A, B, C, D, E, F, G, and H with probabilities 1/30, 1/30, 1/30, 2/30, 3/30, 5/30, 5/30, and 12/30, draw three different Huffman trees with heights 5 and 6 for these symbols and calculate the average code size for each tree.

◇ **Exercise 23.9:** Figure Ans.64d shows another Huffman tree, with height 4, for the eight symbols introduced in Exercise 23.8. Explain why this tree is wrong.

It turns out that the arbitrary decisions made in constructing the Huffman tree affect the individual codes but not the average size of the code. Still, we have to answer the obvious question, which of the different Huffman codes for a given set of symbols is best? The answer, while not obvious, is simple: The best code is the one with the smallest variance. The variance of a code measures how much the sizes of the individual codes deviate from the average size. The variance of code 23.35a is

$$0.4(1-2.2)^2+0.2(2-2.2)^2+0.2(3-2.2)^2+0.1(4-2.2)^2+0.1(4-2.2)^2=1.36,$$

while the variance of code 23.35b is

$$0.4(2-2.2)^2+0.2(2-2.2)^2+0.2(2-2.2)^2+0.1(3-2.2)^2+0.1(3-2.2)^2=0.16.$$

Code 23.35b is therefore preferable (see below). A careful look at the two trees shows how to select the one we want. In the tree of Figure 23.35a, symbol a_{45} is combined with a_3, whereas in the tree of 23.35b it is combined with a_1. The rule is: When there are more than two smallest-probability nodes, select the ones that are lowest and highest in the tree and combine them. This will combine symbols of low probability with ones of high probability, thereby reducing the total variance of the code.

If the encoder simply writes the compressed stream on a file, the variance of the code makes no difference. A small-variance Huffman code is preferable only in cases where the encoder *transmits* the compressed stream, as it is being generated, over a communications line. In such a case, a code with large variance causes the encoder to

generate bits at a rate that varies all the time. Since the bits have to be transmitted at a constant rate, the encoder has to use a buffer. Bits of the compressed stream are entered into the buffer as they are being generated and are moved out of it at a constant rate, to be transmitted. It is easy to see intuitively that a Huffman code with zero variance will enter bits into the buffer at a constant rate, so only a short buffer will be needed. The larger the code variance, the more variable is the rate at which bits enter the buffer, requiring the encoder to use a larger buffer.

The following claim is sometimes found in the literature:

> It can be shown that the size of the Huffman code of a symbol a_i with probability P_i is always less than or equal to $\lceil -\log_2 P_i \rceil$.

Even though it is correct in many cases, this claim is not true in general. It seems to be a wrong corollary drawn by some authors from the Kraft-MacMillan inequality, [Salomon 09]. I am indebted to Guy Blelloch for pointing this out and also for the example of Table 23.36.

◇ **Exercise 23.10:** Find an example where the size of the Huffman code of a symbol a_i is greater than $\lceil -\log_2 P_i \rceil$.

P_i	Code	$-\log_2 P_i$	$\lceil -\log_2 P_i \rceil$
.01	000	6.644	7
*.30	001	1.737	2
.34	01	1.556	2
.35	1	1.515	2

Table 23.36: A Huffman Code Example.

◇ **Exercise 23.11:** It seems that the size of a code must also depend on the number n of symbols (the size of the alphabet). A small alphabet requires just a few codes, so they can all be short; a large alphabet requires many codes, so some must be long. This being so, how can we say that the size of the code of symbol a_i depends just on its probability P_i?

Figure 23.37 shows a Huffman code for the 26 letters. As a self-exercise, the reader may calculate the average size, entropy, and variance of this code.

◇ **Exercise 23.12:** Discuss the Huffman codes for equal probabilities.

Exercise 23.12 shows that symbols with equal probabilities don't compress under the Huffman method. This is understandable, since strings of such symbols normally make random text, and random text does not compress. There may be special cases where strings of symbols with equal probabilities are not random and can be compressed. A good example is the string $a_1a_1 \ldots a_1a_2a_2 \ldots a_2a_3a_3 \ldots$ in which each symbol appears in a long run. This string can be compressed with run-length encoding (RLE) but not with Huffman codes.

Notice that the Huffman method cannot be applied to a two-symbol alphabet. In such an alphabet, one symbol can be assigned the code 0 and the other code 1. The Huffman method cannot assign to any symbol a code shorter than one bit, so it cannot

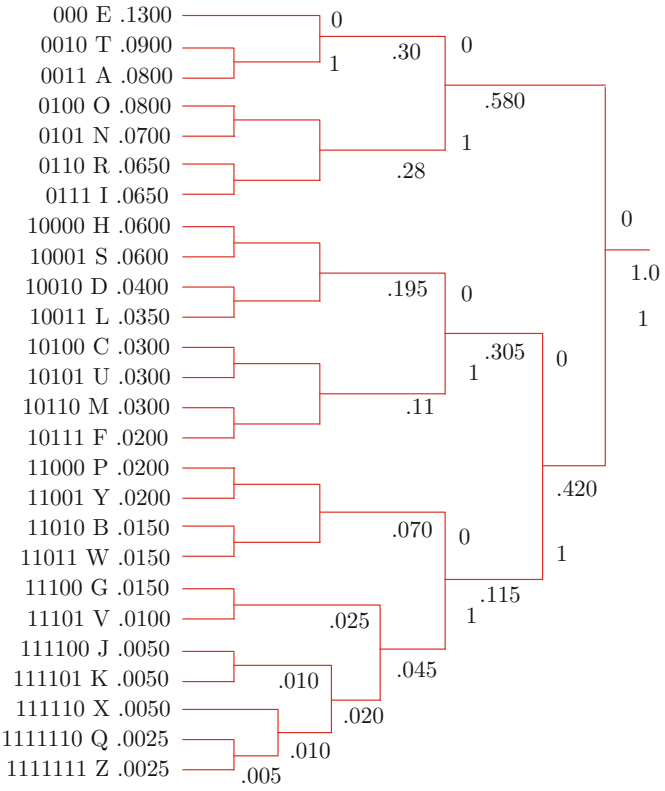


Figure 23.37: A Huffman Code for the 26-Letter Alphabet.

improve on this simple code. If the original data (the source) consists of individual bits, such as in the case of a bi-level (monochromatic) image, it is possible to combine several bits (perhaps four or eight) into a new symbol and pretend that the alphabet consists of these (16 or 256) symbols. The problem with this approach is that the original binary data may have certain statistical correlations between the bits, and some of these correlations would be lost when the bits are combined into symbols. When a typical bi-level image (a painting or a diagram) is digitized by scan lines, a pixel is more likely to be followed by an identical pixel than by the opposite one. We therefore have a file that can start with either a 0 or a 1 (each has 0.5 probability of being the first bit). A zero is more likely to be followed by another 0 and a 1 by another 1. Figure 23.38 is a finite-state machine illustrating this situation. If these bits are combined into, say, groups of eight, the bits inside a group will still be correlated, but the groups themselves will not be correlated by the original pixel probabilities. If the input stream contains, e.g., the two adjacent groups 00011100 and 00001110, they will be encoded independently, ignoring

the correlation between the last 0 of the first group and the first 0 of the next group. Selecting larger groups improves this situation but increases the number of groups, which implies more storage for the code table and longer time to calculate the table.

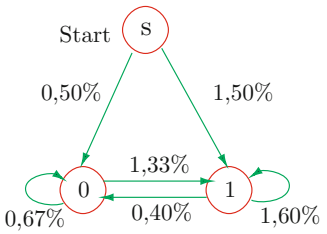


Figure 23.38: A Finite-State Machine.

◇ **Exercise 23.13:** How does the number of groups increase when the group size increases from s bits to $s + n$ bits?

A more complex approach to image compression by Huffman coding is to create several complete sets of Huffman codes. If the group size is, e.g., eight bits, then several sets of 256 codes are generated. When a symbol S is to be encoded, one of the sets is selected, and S is encoded using its code in that set. The choice of set depends on the symbol preceding S.

◇ **Exercise 23.14:** Imagine an image with 8-bit pixels where half the pixels have values 127 and the other half have values 128. Analyze the performance of run-length encoding (RLE) on the individual bitplanes of such an image, and compare it with what can be achieved with Huffman coding.

23.13.1 Dual Tree Coding

Dual tree coding, an idea due to G. H. Freeman ([Freeman 91] and [Freeman 93]), combines Tunstall and Huffman coding in an attempt to improve the latter's performance for a two-symbol alphabet. The idea is to use the Tunstall algorithm [Salomon 09] to extend such an alphabet from two symbols to 2^k strings of symbols, and select k such that the probabilities of the strings will be close to negative powers of 2. Once this is achieved, the strings are assigned Huffman codes and the input stream is compressed by replacing the strings with these codes. This approach is illustrated here by a simple example.

Given a binary source that emits two symbols a and b with probabilities 0.15 and 0.85, respectively, we try to compress it in four different ways as follows:

1. We apply the Huffman algorithm directly to the two symbols. This simply assigns the two 1-bit codes 0 and 1 to a and b, so there is no compression.

2. We combine the two symbols to obtain the four two-symbol strings aa, ab, ba, and bb, with probabilities 0.0225, 0.1275, 0.1275, and 0.7225, respectively. The four strings are assigned Huffman codes as shown in Figure 23.39a, and it is obvious that the

average code length is $0.0225 \times 3 + 0.1275 \times 3 + 0.1275 \times 2 + 0.7225 \times 1 = 1.4275$ bits. On average, each two-symbol string is compressed to 1.4275 bits, yielding a compression ratio of $1.4275/2 \approx 0.714$.

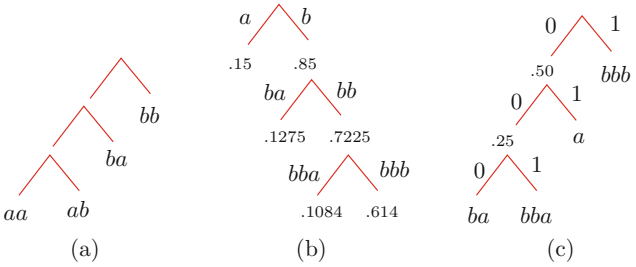


Figure 23.39: Dual Tree Coding.

3. We apply Tunstall's algorithm [Salomon 09] to obtain the four strings *bbb*, *bba*, *ba*, and *a* with probabilities 0.614, 0.1084, 0.1275, and 0.15, respectively. The resulting parse tree is shown in Figure 23.39b. Tunstall's method compresses these strings by replacing each with a two-bit code. Given a string of 257 bits with these probabilities, we expect the strings *bbb*, *bba*, *ba*, and *a* to occur 61, 11, 13, and 15 times, respectively, for a total of 100 strings. Thus, Tunstall's method compresses the 257 input bits to $2 \times 100 = 200$ bits, for a compression ratio of $200/257 \approx 0.778$.

4. We now change the probabilities of the four strings above to negative powers of 2, because these are the best values for the Huffman method. Strings *bbb*, *bba*, *ba*, and *a* are thus assigned the probabilities 0.5, 0.125, 0.125, and 0.25, respectively. The resulting Huffman code tree is shown in Figure 23.39c and it is easy to see that the 61, 11, 13, and 15 occurrences of these strings will be compressed to a total of $61 \times 1 + 11 \times 3 + 13 \times 3 + 15 \times 2 = 163$ bits, resulting in a compression ratio of $163/257 \approx 0.634$, much better.

To summarize, applying the Huffman method to a two-symbol alphabet produces no compression. Combining the individual symbols in strings as in 2 above or applying the Tunstall method as in 3, produce moderate compression. In contrast, combining the strings produced by Tunstall with the codes generated by the Huffman method, results in much better performance. The dual tree method starts by constructing the Tunstall parse tree and then using its leaf nodes to construct a Huffman code tree. The only (still unsolved) problem is determining the best value of k. In our example, we iterated the Tunstall algorithm until we had $2^2 = 4$ strings, but iterating more times may have resulted in strings whose probabilities are closer to negative powers of 2.

23.13.2 Huffman Decoding

Before starting the compression of a data stream, the compressor (encoder) has to determine the codes. It does that based on the probabilities (or frequencies of occurrence) of the symbols. The probabilities or frequencies have to be written, as side information, on the compressed stream, so that any Huffman decompressor (decoder) will be able to

decompress the stream. This is easy, since the frequencies are integers and the probabilities can be written as scaled integers. It normally adds just a few hundred bytes to the compressed stream. It is also possible to write the variable-length codes themselves on the stream, but this may be awkward, because the codes have different sizes. It is also possible to write the Huffman tree on the stream, but this may require more space than just the frequencies.

In any case, the decoder must know what is at the start of the stream, read it, and construct the Huffman tree for the alphabet. Only then can it read and decode the rest of the stream. The algorithm for decoding is simple. Start at the root and read the first bit off the compressed stream. If it is zero, follow the bottom edge of the tree; if it is one, follow the top edge. Read the next bit and move another edge toward the leaves of the tree. When the decoder gets to a leaf, it finds the original, uncompressed code of the symbol (normally its ASCII code), and that code is emitted by the decoder. The process starts again at the root with the next bit.

This process is illustrated for the five-symbol alphabet of Figure 23.40. The four-symbol input string $a_4a_2a_5a_1$ is encoded into 1001100111. The decoder starts at the root, reads the first bit 1, and goes up. The second bit 0 sends it down, as does the third bit. This brings the decoder to leaf a_4, which it emits. It again returns to the root, reads 110, moves up, up, and down, to reach leaf a_2, and so on.

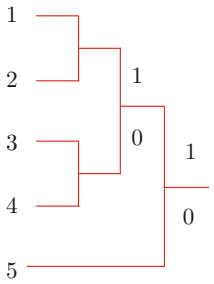


Figure 23.40: Huffman Codes for Equal Probabilities.

23.13.3 Fast Huffman Decoding

Decoding a Huffman-compressed file by sliding down the code tree for each symbol is conceptually simple, but slow. The compressed file has to be read bit by bit and the decoder has to advance a node in the code tree for each bit. The method of this section, originally conceived by [Choueka et al. 85] but later reinvented by others, uses preset partial-decoding tables. These tables depend on the particular Huffman code used, but not on the data to be decoded. The compressed file is read in chunks of k bits each (where k is normally 8 or 16 but can have other values) and the current chunk is used as a pointer to a table. The table entry that is selected in this way can decode several symbols and it also points the decoder to the table to be used for the next chunk.

As an example, consider the Huffman code of Figure 23.35a, where the five codewords are 0, 10, 111, 1101, and 1100. The string of symbols $a_1a_1a_2a_4a_3a_1a_5\ldots$ is compressed by this code to the string 0|0|10|1101|111|0|1100.... We select $k = 3$ and

read this string in 3-bit chunks 001|011|011|110|110|0.... Examining the first chunk, it is easy to see that it should be decoded into a_1a_1 followed by the single bit 1 which is the prefix of another codeword. The first chunk is $001 = 1_{10}$, so we set entry 1 of the first table (table 0) to the pair $(a_1a_1, 1)$. When chunk 001 is used as a pointer to table 0, it points to entry 1, which immediately provides the decoder with the two decoded symbols a_1a_1 and also directs it to use table 1 for the next chunk. Table 1 is used when a partially-decoded chunk ends with the single-bit prefix 1. The next chunk is $011 = 3_{10}$, so entry 3 of table 1 corresponds to the encoded bits 1|011. Again, it is easy to see that these should be decoded to a_2 and there is the prefix 11 left over. Thus, entry 3 of table 1 should be $(a_2, 2)$. It provides the decoder with the single symbol a_2 and also directs it to use table 2 next (the table that corresponds to prefix 11). The next chunk is again $011 = 3_{10}$, so entry 3 of table 2 corresponds to the encoded bits 11|011. It is again obvious that these should be decoded to a_4 with a prefix of 1 left over. This process continues until the end of the encoded input. Figure 23.41 is the simple decoding algorithm in pseudocode.

```
i←0; output←null;
repeat
  j←input next chunk;
  (s,i)←Table_i[j];
  append s to output;
until end-of-input
```

Figure 23.41: Fast Huffman Decoding.

Table 23.42 lists the four tables required to decode this code. It is easy to see that they correspond to the prefixes Λ (null), 1, 11, and 110. A quick glance at Figure 23.35a shows that these correspond to the root and the four interior nodes of the Huffman code tree. Thus, each partial-decoding table corresponds to one of the four prefixes of this code. The number m of partial-decoding tables therefore equals the number of interior nodes (plus the root) which is one less than the number N of symbols of the alphabet.

$T_0 = \Lambda$			$T_1 = 1$			$T_2 = 11$			$T_3 = 110$		
000	$a_1a_1a_1$	0	1|000	$a_2a_1a_1$	0	11|000	a_5a_1	0	110|000	$a_5a_1a_1$	0
001	a_1a_1	1	1|001	a_2a_1	1	11|001	a_5	1	110|001	a_5a_1	1
010	a_1a_2	0	1|010	a_2a_2	0	11|010	a_4a_1	0	110|010	a_5a_2	0
011	a_1	2	1|011	a_2	2	11|011	a_4	1	110|011	a_5	2
100	a_2a_1	0	1|100	a_5	0	11|100	$a_3a_1a_1$	0	110|100	$a_4a_1a_1$	0
101	a_2	1	1|101	a_4	0	11|101	a_3a_1	1	110|101	a_4a_1	1
110	–	3	1|110	a_3a_1	0	11|110	a_3a_2	0	110|110	a_4a_2	0
111	a_3	0	1|111	a_3	1	11|111	a_3	2	110|111	a_4	2

Table 23.42: Partial-Decoding Tables for a Huffman Code.

Notice that some chunks (such as entry 110 of table 0) simply send the decoder to another table and do not provide any decoded symbols. Also, there is a tradeoff

between chunk size (and thus table size) and decoding speed. Large chunks speed up decoding, but require large tables. A large alphabet (such as the 128 ASCII characters or the 256 8-bit bytes) also requires a large set of tables. The problem with large tables is that the decoder has to set up the tables after it has read the Huffman codes from the compressed stream and before decoding can start, and this process may preempt any gains in decoding speed provided by the tables.

To set up the first table (table 0, which corresponds to the null prefix Λ), the decoder generates the 2^k bit patterns 0 through $2^k - 1$ (the first column of Table 23.42) and employs the decoding method of Section 23.13.2 to decode each pattern. This yields the second column of Table 23.42. Any remainders left are prefixes and are converted by the decoder to table numbers. They become the third column of the table. If no remainder is left, the third column is set to 0 (use table 0 for the next chunk). Each of the other partial-decoding tables is set in a similar way. Once the decoder decides that table 1 corresponds to prefix p, it generates the 2^k patterns $p|00\ldots0$ through $p|11\ldots1$ that become the first column of that table. It then decodes that column to generate the remaining two columns.

This method was conceived in 1985, when storage costs were considerably higher than today (late 2010). This prompted the developers of the method to find ways to cut down the number of partial-decoding tables, but these techniques are less important today and are not described here.

He can compress the most words into
the smallest idea of any man I know.
—Abraham Lincoln