CHAPTER 3

Universal Codes

PETER FENWICK

3.1 COMPACT INTEGER REPRESENTATIONS

Many data compressors produce as intermediate output one or more sets of integers, with smaller values being most probable and larger values increasingly less probable. The final, coding stage must convert these integers into a bit stream, such that each value is self-delimiting and the total number of bits in the whole stream is minimized.

When only a finite alphabet of numbers must be handled and where the number probabilities are known *a priori* it is appropriate to use the well-known Huffman or Shannon–Fano codes, possibly in an adaptive version which adjusts the code as the probabilities are discovered.

This chapter describes ways of encoding an arbitrary integer, often unbounded in size, in as few bits as possible. A simple algorithm should be able to encode an integer into the output bit stream or recover an integer from an input bit stream, even if that particular value has never been seen before. These codes, self-delimiting and variable in length for different values, are usually known as *Universal Codes* or *Variable-Length Codes.*

Many of these codes are difficult to find in the literature, often being secondary aspects or side issues in papers on other topics. The book by Bell *et al.* [1] has a (very) few pages on the subject. Salomon [2] discusses several codes in passing, but refers only obliquely to the Fibonacci codes.

3.2 CHARACTERISTICS OF UNIVERSAL CODES

Universal codes may have to satisfy several requirements, whose relevance depends on the context. For example, robustness may be quite unimportant in many computer applications with error-free transmission, but may be crucial in noisy radio communications. Transparency is unimportant in most computer work, except that it may facilitate conversion to and from the internal representation of the computer.

For all of these codes, an integer N occurring with probability P_N is encoded into or represented by a *codeword* of L binary digits. A fundamental result of information theory is that for optimal coding all codewords have a length $L = \log_2(1/P_N)$ bits, leading to the important general rule that frequent values should have short codewords and less frequent values should have longer codewords.

Efficiency: An *efficient code*, or *compact code*, is assumed to be one which, for each value, approximates the minimum codeword length in comparison with other codes. In many codes the codeword length is nearly a constant multiple of the binary representation. For these codes the expansion relative to binary is a useful indication of efficiency.

Length indication: Every universal representation must somehow indicate its length or signal the end of the bit stream for the codeword. Some representations are terminated by a special bit pattern or *comma*. Others have an explicit length as part of the representation, the length itself often using one of the terminated representations.

Transparency: It is pleasant for human understanding, but certainly not essential, if the value is easily extracted once a codeword is recognized. Some codes, for example, have a portion of the codeword as the natural binary representation of the value.

Robustness: The increasingly subtle coding techniques used to increase code efficiency often reduce the redundancy of the code. But the sensitivity to transmission errors usually increases with the decreasing redundancy. In extreme cases a single bit error can lead to catastrophic error propagation and loss of codeword demarcation. Codes with a terminating pattern or comma tend to be more robust and those with an encoded length less robust.

Instantaneousness: This describes a code that is self-contained, in that a codeword can be decoded without reference to adjoining codewords. A non-instantaneous code is fine in a long sequence of codewords from the same code, but may be inconvenient if the compact codes are interspersed with other, instantaneous, codes.

As a final introductory note, within this chapter the term *bit* will always mean *binary digit* rather than an information measure. A codeword (the encoded collection of bits to represent the value) is usually longer than the binary representation of the value (also in bits).

3.3 POLYNOMIAL REPRESENTATIONS

Many representations for an integer N combine a visible *digit vector* $\mathbf{d}$ with an implicit *weight vector* $\mathbf{w}$, such that $N = \mathbf{d} \cdot \mathbf{w}$, the scalar product of the two vectors. A polynomial in some base b results if successive terms of the weight vector are given by $w_{i+1} \approx bw_i$. While an obvious example of a polynomial representation is the familiar decimal representation, it is more instructive to consider some other forms.

Binary: ($b = 2$) The weight vector is powers of 2, least to the right, or $\mathbf{w} = \{\ldots, 128, 64, 32, 16, 8, 4, 2, 1\}$ and the visible digits $d_i \in \{0, 1\}$. The (decimal) value 23 is represented by 10111

$$23 = 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1.$$

Ternary: ($b = 3$) The weight vector is now $\mathbf{w} = \{\ldots, 729, 243, 81, 27, 9, 3, 1\}$ and $d_i \in \{0, 1, 2\}$. 209_{10} is represented by 21202

$$209 = 2 \times 81 + 1 \times 27 + 2 \times 9 + 0 \times 32 + 2 \times 1.$$

Table 3.1 Effective Bases of Some Codes

Code	Base (b)		Expansion (x)
Unary (α)	1	1	—
Binary (β)	2	2	1
Elias γ	$\sqrt{2}$	1.414	2
Fibonacci-2	$\frac{\sqrt{5}+1}{2}$	1.618	1.440
Ternary	$\sqrt{3}$	1.732	1.262
Fibonacci-3	See text	1.839	1.137

Fibonacci: This is a less obvious example with the weight vector the Fibonacci numbers $\mathbf{w} = \{\ldots, 34, 21, 13, 8, 5, 3, 2, 1\}$ and the digit vector $\mathbf{d}$ again binary. The (decimal) value 20 is represented by 101010

$$20 = 1 \times 13 + 0 \times 8 + 1 \times 5 + 0 \times 3 + 1 \times 2 + 0 \times 1.$$

As successive Fibonacci numbers tend to the ratio $(\sqrt{5} + 1)/2 \approx 1.618$, this may be regarded as another polynomial representation.

Many universal codes (including the Fibonacci example above) are approximated by polynomial codes with a non-integral b, $1 < b < 2$. The base b representation of N has $\log_b N$ digits d_i (that are necessarily binary). As the binary representation has $\log_2 N$ digits and $b < 2$, the base b representation expands relative to binary. The base b and expansion x are both descriptive of the representation and are related by

$$x = \log 2 / \log b, \quad \text{or} \quad b = 2^{1/x}$$

To anticipate later developments, the bases and expansions of several codes are given in Table 3.1. The simplest of these is the Elias γ code, which, generating two bits per binary digit, is equivalent to a number with a base of $\sqrt{2} = 1.414$. At the other extreme, the Fibonacci-3 (or Tribonacci) numbers have a ratio between successive digit weights of

$$\frac{1 + \sqrt[3]{19 - 3\sqrt{33}} + \sqrt[3]{19 + 3\sqrt{33}}}{3} \approx 1.83928675521,$$

giving an effective number base of about 1.84 and an expansion relative to binary of $\log 2 / \log 1.839 = 1.137$.

Some codes add pairs of bits as the value grows; on average these have a small additive constant when calculating the length. Although their growth may be slower than that of codes that grow 1 bit at a time, the additive term may make them less attractive for small values. Similarly, codes with smaller expansions often need a more complex length indication, which may cancel any benefit from the smaller expansion.

3.4 UNARY CODES

The unary code of an integer N is simply a sequence of N 1's followed by a 0 (sometimes N 0's with a terminating 1). A variant code for N may have $(N - 1)$ leading digits but cannot encode a 0. Despite their apparent triviality, unary codes are important and often components of more complex codes. (The unary code $11\ldots10$ is a polynomial code, with base $b = 1$ and terminated by the first 0.)

Unary codes are optimal for highly skewed symbol distributions, where the symbol probabilities are related as $P_{k+1} \leq P_k/2$.

3.5 LEVENSTEIN AND ELIAS GAMMA CODES

These codes were first described by Levenstein [3], but the later description by Elias [4] is generally used in the English language literature. Elias describes a whole series of codes:

alpha code: The $\alpha(N)$ code is one of the unary representations above, with $(N - 1)$ 0's followed by a 1. ($\alpha(N)$ has a length of N digits.) Thus $\alpha(7)$ is 0000001. The α code is an example of a *comma* code, where the comma is the terminating 1.

beta code: The $\beta(N)$ code is the natural binary representation of N, starting from the most significant 1. $\beta(6)$ is 110, and $\beta(19)$ is 10011. Sometimes the β code may be modified by omitting the leading 1 bit. The β code is of little use by itself because it has no length indication.

gamma code: The γ code is an intermingling of the bits of the β code and an α code describing its length, omitting the first bit of the β code (which is always 1). Each numeric bit (from the β code) is preceded by a 0 *flag* bit (from the α code), with the whole terminating in the final 1 from the α code. These codes are shown in the first part of Table 3.2, with the flag bits marked by an overline.

gamma′ code: The γ' code is a permutation of the γ code, with the flag bits (now an α code) preceding the data bits (a β code) and the terminating 1 of the α prefix acting also as the leading 1 of the β suffix.

For most of this chapter the term Elias γ code will be used interchangeably for the two variants, often meaning the γ' code. Examples of the four codes (α, β, γ, and γ') are shown in Table 3.3.

Ignoring the terminating condition, the Levenstein and Elias γ codes represent each binary digit by 2 bits; the expansion is $x = 2$ and the effective base is $b = \sqrt{2} \approx 1.414$, as shown in Table 3.1.

The γ code can be extended to higher number bases where such granularity is appropriate. For example, numbers can be held in byte units, with each 8-bit byte containing 1 flag bit (last-byte/more-to-come) and 7 data bits, to give a base-128 code.

Some variants of the γ codes are much older than the systematic descriptions by Levenstein and Elias (1968 and 1975, respectively). For example, the IBM 1620 computer (ca. 1960) used

Table 3.2 Elias' Gamma and Gamma' Codes

γ-Code	γ' Code
$\gamma(1) = \overline{1}$	$\gamma'(1) = \overline{1}$
$\gamma(2) = \overline{0}\,\overline{0}1$	$\gamma'(2) = \overline{0}10$
$\gamma(3) = \overline{0}1\overline{1}$	$\gamma'(3) = \overline{0}11$
$\gamma(4) = \overline{0}\,\overline{0}\,\overline{0}\,\overline{0}1$	$\gamma'(4) = \overline{0}\,\overline{0}100$
$\gamma(5) = \overline{0}1\overline{0}\,\overline{0}1$	$\gamma'(5) = \overline{0}\,\overline{0}101$
$\gamma(6) = \overline{0}\,\overline{0}1\overline{1}$	$\gamma'(6) = \overline{0}\,\overline{0}110$
$\gamma(13) = \overline{0}1\overline{0}\,\overline{0}\,\overline{0}1\overline{1}$	$\gamma'(13) = \overline{0}\,\overline{0}\,\overline{0}1101$
$\gamma(23) = \overline{0}1\overline{0}1\overline{0}1\overline{0}01$	$\gamma'(23) = \overline{0}\,\overline{0}\,\overline{0}\,\overline{0}10111$
$\gamma(44) = \overline{0}\,\overline{0}\,\overline{0}\,\overline{0}1\overline{0}1\overline{0}\,\overline{0}1$	$\gamma'(44) = \overline{0}\,\overline{0}\,\overline{0}\,\overline{0}\,\overline{0}101100$

Table 3.3 Examples of Elias α, β, and γ Codes

N	$\alpha(N)$	$\beta(N)$	$\gamma(N)$	$\gamma'(N)$
1	1	1	1	1
2	01	10	001	010
3	001	11	011	011
4	0001	100	00001	00100
5	00001	101	01001	00101
6	000001	110	00011	00110
7	0000001	111	01011	00111
8	00000001	1000	0000001	0001000
9	000000001	1001	0100001	0001001
10	0000000001	1010	0001001	0001010
50	...	110010	00010000011	00000110010

BCD coding with a flag bit on each decimal digit (memory addressed by individual decimal digit). Numbers were addressed at the least-significant digit and proceeded to lower addresses until terminated by a flagged digit—a precise implementation (or anticipation) of a BCD variant of the γ code. (A flag on the addressed digit denoted a negative number.)

As every *numeric* bit requires two *codeword* bits the γ codes require 2 bits to double the range of values and are equivalent to a polynomial representation with base $b = \sqrt{2}$. The γ codes are very efficient for very small values (they are one of the few codes that represent the smallest value with a single bit). For larger values the large number of flag bits becomes an ever-increasing overhead, as shown by their relatively large expansion.

3.6 ELIAS OMEGA AND EVEN–RODEH CODES

With the γ code length given as a unary code, a natural progression is to encode the length itself as γ code. Elias does this with his δ code, using a γ code for the length, but quickly proceeds to ω codes. Some very similar codes were described by Even and Rodeh [5] and it is convenient to treat the two together. Each of the codes has the value (as a β code) preceded by its length in binary. If the length is not representable in 2 bits (ω code; 3 bits for Even–Rodeh), encode a prefix giving the "length of the length," recursing until a 2-bit value is obtained.

Elias ω code: Some Elias ω codes are shown in Table 3.4, with groups of bits separated by blanks. Each length is followed by the most-significant 1 of the next length or value; the final value is followed by a 0. The codes are most easily described by giving the decoding process, unwinding the recursive generation of the code value.

The first group of bits is always either 10 or 11, specifying a value of 2 or 3, or a following group of 2 or 3 bits. If a group in an ω code is followed by a 0, its value is the value to be delivered. If a group is followed by a 1, its value is the number of bits to be read and placed after that following 1 to give the value of the next group. Thus 15 is read as the sequence 3, 15 and 16 is read as 2, 4, 16. (Observe that the initial bits of each group form an α code controlling the codeword length.) As all other codewords start with a 1, a single 0 bit is used to denote a value of 1.

Even–Rodeh code: The Even–Rodeh code is similar to the Elias ω code, but each group now gives the total number of bits in the following group, *including* the most significant 1. A different starting procedure is used, with values of 0–3 written as 3-bit integers and the

Table 3.4 Examples of Elias' ω and Even–Rodeh Codes

Value	Elias ω code	Even–Rodeh Code
0	—	000
1	0	001
2	10 0	010
3	11 0	011
4	10 100 0	100 0
7	10 111 0	111 0
8	11 1000 0	100 1000 0
15	11 1111 0	100 1111 0
16	10 100 10000 0	101 10000 0
32	10 101 100000 0	110 100000 0
100	10 110 1100100 0	111 1100100 0
1000	11 1010 1111101000 0	1010 1111101000 0

Table 3.5 Lengths of Elias' ω and Even–Rodeh Codes

Values	Elias	Even–Rodeh
1	1	3
2–3	3	3
4–7	6	4
8–15	7	8
16–31	11	9
32–63	12	10
64–127	13	11
128–255	14	17
256–512	21	18

initial group now 3 bits. Values of 4–7 are in natural binary, with a following 0 as terminator. The Even–Rodeh code *does* represent a 0 value, as the bits 000; the smallest value for the Elias ω code is 1.

Both codes are especially efficient just before a new length element is phased in and inefficient just after it is introduced, as for 15 and 16 in the Elias ω code. The codes alternate in relative efficiency as their extra length components phase in at different values, as shown in Table 3.5.

Bentley and Yao [6] develop a very similar code as a consequence of an optimal strategy for an unbounded search, recognizing a correspondence between the tests of the search and the coding of index of the search target, but do not develop the code to the detail of either Elias or Even and Rodeh.

3.7 RICE CODES

Rice codes [7] have an integer parameter k. The representation is the concatenation of $(1 + N/2^k)$ as a unary code and (N mod 2^k) in binary. (Using $1 + N/2^k$ allows for $N < 2^k$.) An integer N is represented by about $N/2^k + k + 1$ bits. Representative Rice codes are shown in Table 3.6.

Table 3.6 Rice Codes for the First Few Integers and Parameter k

k	0	2	3	4
0	0	000	0000	00000
1	10	001	0001	00001
2	110	010	0010	00010
3	1110	011	0011	00011
4	11110	1000	0100	00100
5	111110	1001	0101	00101
6	1111110	1010	0110	00110
7	11111110	1011	0111	00111
8	111111110	11000	10000	01000
9	1111111110	11001	10001	01001
10	11111111110	11010	10010	01010
11	111111111110	11011	10011	01011
12	1111111111110	111000	10100	01100
13	11111111111110	111001	10101	01101
14	111111111111110	111010	10110	01110
15	1111111111111110	111011	10111	01111

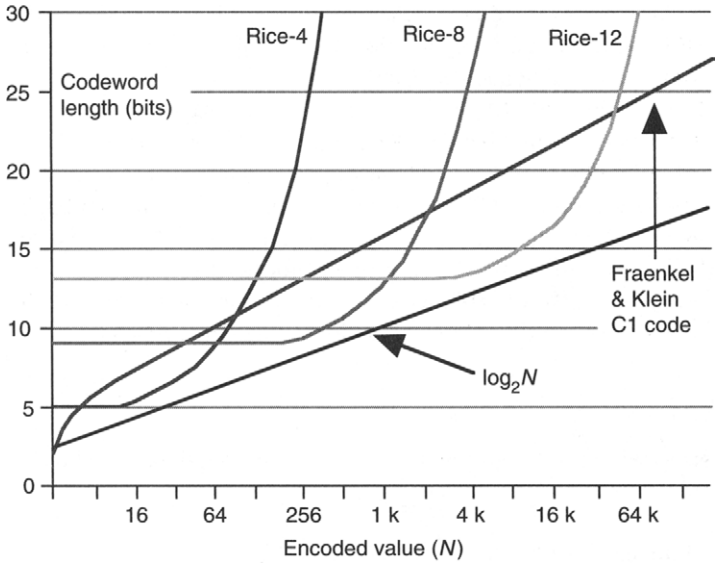

FIGURE 3.1
Lengths of Rice codes (smoothed).

The Rice codes are extremely efficient for values of N near 2^k as shown in Fig. 3.1, comparing Rice codes for $k = 4$, 8, and 12 with the Fraenkel and Klein $C1$ code. The $C1$ code is generally one of the shorter codes; within the appropriate ranges the Rice codes may be several bits shorter. (The codeword lengths should show discontinuities as each bit is introduced but these steps have been smoothed here to make the graphs easier to understand.) The graph also shows how closely the Rice codes approach $\log_2 N$, the binary code length or least possible length.

As smaller values are always represented by $k + 1$ bits, the code becomes progressively less efficient for small values, with many leading 0's. Larger values quickly become less efficient because the α code prefix grows linearly with N, appearing as an exponential increase with the logarithmic axis of Fig. 3.1.

The Rice codes were first developed for deep-space telemetry and especially video transmission. Much of this video data is characterized by a relatively small range of numerical values and Rice shows that by appropriate selection of the parameter k it is possible to code with very small redundancy. The use of the Rice codes in the lossless data compression standard for space applications is described in Chapter 16.

3.8 GOLOMB CODES

The Golomb codes [8] are designed to encode a sequence of asymmetric binary events, where a probable event with probability p is interspersed with unlikely events of probability q ($q = 1 - p$ and $p \gg q$).[1]

The sequence is represented by the lengths of successive runs of the probable event between occurrences of the improbable event. The Golomb codes have a parameter m, related to the probability p by $p^m = 0.5$, or $p = \sqrt[m]{0.5}$. A run of length $N + m$ is half as likely as a run of length N, indicating that the codeword for a run of length $N + m$ should be 1 bit longer than that for a run of length N.

Golomb codes may be regarded as a generalization of the Rice codes, with a Rice(k) code identical to the Golomb (2^k) code. They are the most complex of the codes of this chapter, with quite different rules for encoding and decoding.

The dictionary for a Golomb (m) code consists of a sequence of codeword groups, numbered from 0. All groups except the first have m codewords all of the same length, starting with an α prefix. Each codeword within a group is obtained by adding 1 to its predecessor, but this addition may overflow into the α prefix and modify it for later members of the group. The first member of a group has the last value from the previous group incremented by 1 and extended by a single trailing 0. The dictionary starts with a small group of shorter words.

This example assumes that all arithmetic is in binary or, rather, that the final codeword is represented in binary. To encode N with the Golomb (m) code, first find the least k, such that $2^k \geq 2m$:

$$k = \lceil \log_2 m \rceil + 1. \tag{3.1}$$

The dictionary starts with a small group of j codewords of length $(k - 1)$ bits

$$j = 2^{k-1} - m. \tag{3.2}$$

Calculate g, the number of the group which encodes N:

$$g = (N - j)/m + 1. \tag{3.3}$$

Now generate a raw codeword p with $(g - 1)$ leading 1's and k following 0's:

$$p = (2^{g-1} - 1)2^k. \tag{3.4}$$

The first or *base* codeword b of group g is

$$b = p + 2j. \tag{3.5}$$

[1] Golomb's original paper should be consulted for a full description of how the code developed from activities of Secret Agent 0 0 111 at the casino!

Find the first value s which will encode into the group

$$s = m(g-1) + j \tag{3.6}$$

and add the difference $(N - s)$ into the base codeword, obtaining the final codeword w

$$w = b + (N - s). \tag{3.7}$$

Ignoring the special case of the first group, the first codeword of a group represents the value $(m(g-1)+j)$ and starts with an α code prefix $\alpha(g-1)$ followed by the value $2j$ to $(k-1)$ bits. Later values within the group of m codewords are obtained by successively incrementing the predecessor code. Because the counting eventually overflows into the end of the α code prefix, the group always ends with several words that *seem* to belong to the next group. These anomalous codewords complicate decoding.

The $m = 6$ code is shown in Table 3.7, with a colon separating the prefix and suffix of each codeword. Groups are separated by a horizontal line; there is a group boundary between 7 and 8, but not at other column boundaries. The overflow into the α prefix is seen in the codewords for 6, 12, 18, 24, and 30.

The Golomb coder produces groups of codewords with m words of each length. Most codewords also occur in groups with 1, 2, 3, ... leading 1's, with m words to a group. Unfortunately the encoding and decoding groups do not coincide unless $m = 2^k$.

If m is a power of 2 (corresponding to a Rice code), the codeword for N is a simple concatenation of $\alpha(1 + N/m)$ as a prefix, followed by the binary representation of $N \bmod m$ to $\log m$ bits, i.e.,

$$\alpha(1 + N/m) : \beta(N \bmod m).$$

Table 3.8 shows a few Golomb codes for this case, with the α and β components separated by a colon for the special case where $2^k = 2m$ and $j = 2^{k-1} - m = 0$. Some general examples of Golomb codes are shown in Table 3.9.

Table 3.7 Structure of the Golomb(6) Code

0	:000	8	10:100	16	110:110	24	1111:000
1	:001	9	10:101	17	110:111	25	1111:001
2	0:100	10	10:110	18	111:000	26	11110:100
3	0:101	11	10:111	19	111:001	27	11110:101
4	0:110	12	11:000	20	1110:100	28	11110:110
5	0:111	13	11:001	21	1110:101	29	11110:111
6	1:000	14	110:100	22	1110:110	30	11111:000
7	1:001	15	110:101	23	1110:111	31	11111:001

Table 3.8 Some Golomb Codes for $m = 2^i$

$m \rightarrow$ $\downarrow N$	2	4	8
3	10:1	0:11	0:011
9	11110:1	110:01	10:001
14	11111110:0	1110:10	10:110

Table 3.9 Golomb Codes for Small Integers, with Parameter m

$m \rightarrow$ $\downarrow N$	1	2	3	4	5
0	0	00	00	000	000
1	10	01	010	001	001
2	110	100	011	010	010
3	1110	101	100	011	0110
4	11110	1100	1010	1000	0111
5	111110	1101	1011	1001	1000
6	1111110	11100	1100	1010	1001
7	11111110	11101	11010	1011	1010
8	111111110	111100	11011	11000	10110
9	1111111110	111101	11100	11001	10111
10	11111111110	1111100	111010	11010	11000
11	111111111110	1111101	111011	11011	11001
12	1111111111110	11111100	111100	111000	11010

To decode the Golomb (m) code, the parameter m immediately gives the values of k and j from Eqs. (1) and (2) above. If the first digit is 0, then either $g = 0$ (digits 00...) or $g = 1$ (digits 01...). After the a leading 1's, read the next $(k - 1)$ bits (starting with the first 0) as a value x. If $x \geq j$, then read one more input digit to the low-order end of x. If $x < j$, reduce the value a by 1 because this codeword really belongs to the preceding coding group.

Using the symbols from the code generation, reverse the encoding process, to give

$$N = [m(a - 1) + j] + [x - 2j].$$

The comments on the Rice code efficiency apply mostly to Golomb codes, remembering the equivalence of a Golomb (2^k) and a Rice (k) code. A Golomb code shows the same run-away codeword length for large values, and high efficiency for, say, $m/2 < N < 2m$. However, if $m \neq 2^k$, a Golomb code is shorter than a Rice code for very small values, giving an advantage over the Rice codes.

3.9 START–STEP–STOP CODES

These codes [9] are defined by three parameters, i, j, and k. The representation may be less clearly related to the value than for Elias γ and Rice codes. If the last parameter k is finite, these codes handle only a finite alphabet; an infinite alphabet requires that $k = \infty$.

The codewords contain both a prefix and a suffix. The code defines a series of blocks of codewords of increasing length, the first block with a suffix of i bits (β code), the second with $i + j$ bits, then $i + 2j$ bits, and so on, up to a final suffix length of k bits. A unary prefix gives the number of the suffix group. Thus a 3, 2, 9 code has codewords with suffixes of 3, 5, 7, and 9 bits and prefixes of 0, 10, 110, and 111 (omitting the final 0 from the last prefix) as shown in Table 3.10.

The start–step–codes can generate many of the other codes, or codes equivalent to them, as shown in Table 3.11.

Table 3.10 Code Values for a 3, 2, 9 Start–Step–Stop Code

Codeword	Range
0xxx	0–7
10xxxxx	8–39
110xxxxxxx	40–167
111xxxxxxxxx	168–679

Table 3.11 Special Cases of Start–Step–Stop Codes

Parameters			
i	j	k	Generated Code
N	1	N	A simple binary coding of the integers to $2^N - 1$
0	1	∞	The Elias γ' code
N	N	∞	The base 2^N Elias γ' code
N	0	∞	A code equivalent to the Rice(N) code

Table 3.12 The First Few Fibonacci Numbers

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}
1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1,597	2,584

3.10 FIBONACCI CODES

Universal codes based on Fibonacci numbers were first described by Apostolico and Fraenkel [11] and later by Fraenkel and Klein [12]. Both emphasize the robustness of the codes or their ability to recover from bit errors. Here the emphasis is on their use as universal or variable-length codes, assuming error-free transmission. The presentation here follows the development of the second (Fraenkel and Klein) paper as being rather easier to understand. The earlier paper by Apostolico and Fraenkel treats more general cases and misses some simpler aspects; its results will be given later.

The Fibonacci numbers are a famous integer sequence where each number is the sum of its two immediate predecessors. The first few values are shown in Table 3.12.

$$F_i = F_{i-2} + F_{i-1}, \quad \text{where } F_1 = F_2 = 1. \tag{3.8}$$

The ratio of successive Fibonacci numbers tends to the *golden ratio* φ.

$$\frac{F_{k+1}}{F_k} = \varphi = \frac{1+\sqrt{5}}{2} \approx 1.618034. \tag{3.9}$$

3.10.1 Zeckendorf Representation

A standard result of Fibonacci theory is Zeckendorf's theorem [10, p. 108], which states that any integer can be formed as the sum of Fibonacci numbers. This *Zeckendorf representation* is coded by writing a binary vector with a 1 wherever that Fibonacci number is included, but omitting F_1 as redundant; see Vajda [10] or Knuth [13, p. 85, Ex. 34].

Thus $19 = 13 + 5 + 1$; its Zeckendorf representation is $Z(19) = 101001$ because $19 = 1 \times 13 + 0 \times 8 + 1 \times 5 + 0 \times 3 + 0 \times 2 + 1 \times 1$. The standard Zeckendorf representation assumes that the least-significant bit is on the right, in line with normal numerical representations. When coding it is often better to write the bits in the reverse order, least-significant first, which is represented as $F(N)$. Thus $F(19) = 100101$.

The crucial property of the Zeckendorf representation is that it never has two adjacent 1's. By the definition of the Fibonacci numbers (Eq. (3.8)), such a pair is always equivalent to a single more-significant 1.

As the ratio of two successive Fibonacci numbers tends quickly to $\varphi \approx 1.618034$ (Eq. (3.9)), codes based on the Fibonacci numbers are polynomial representations with a base $b \approx 1.618$ and an expansion $x = 1.440$, as shown earlier in Table 3.1.

3.10.2 Fraenkel and Klein Codes

The Fraenkel and Klein C^1 code (which will be called the FK_1 code) follows immediately from the property that no legitimate Zeckendorf representation has two consecutive 1's. It is simply $F(N)$ immediately followed by another 1 as a terminating comma. Thus, $FK_1(19) = 1001011$.

Fraenkel and Klein develop two other related codes. For uniformity, assume that the smallest value for all codes is 1.

1. The FK_2 code (their C^2 code) includes only $F(N)$ representations with a least-significant bit of 1. This results in codewords of the form $10\ldots1$. Alternatively, the codeword for N can be obtained by adding a prefix 10 to the representation of $F(N-1)$.
   As a special case, the value 1 is represented by a single 1 bit. In contrast to the FK_1 codes, the FK_2 codes are not instantaneous, as each codeword overlaps its neighbors. However, the FK_2 code does allow a more compact representation for the smallest value.
2. The C^3 code (FK_3) is obtained by taking all values of $F(N)$ of some length, say, r bits, and writing down the block twice, first with a prefix of 10 and then with a prefix of 11. Every codeword of FK_3 has an initial 1 bit, no codeword has more than 3 consecutive 1's (and any consecutive 1's appear only as a prefix), and every codeword except $FK_3(2)$ terminates in 01.

Some examples of these codes are shown in Table 3.13, together with the codeword lengths. There is relatively little difference between them; all are better for some values and worse for others. Measurements by Fraenkel and Klein on recoding simple English text show that FK_1 and FK_3 give very similar performances and are superior to FK_2. But FK_2 has a shorter representation for 1 and may be better for more highly skewed distributions.

3.10.3 Higher-Order Fibonacci Representations

Traditional Fibonacci numbers involve the sum of *two* predecessors. In an order-m Fibonacci sequence each number is the sum of its m predecessors. (The order-3 numbers are often known as the *Tribonacci* numbers.) The first few Fibonacci numbers of orders 2 and 3 are shown in

Table 3.13 Fraenkel and Klein's Codes

	Codewords			Lengths		
N	FK_1	FK_2	FK_3	FK_1	FK_2	FK_3
1	11	1	101	2	1	3
2	011	101	111	3	3	3
3	0011	1001	1001	4	4	4
4	1011	10001	1101	4	5	4
5	00011	10101	10001	5	5	5
6	10011	100001	10101	5	6	5
7	01011	101001	11001	5	6	5
8	000011	100101	11101	6	6	5
9	100011	1000001	100001	6	7	6
10	010011	1010001	101001	6	7	6

Table 3.14 Fibonacci Numbers of Orders 2 and 3

$F_1^{(2)}$	$F_2^{(2)}$	$F_3^{(2)}$	$F_4^{(2)}$	$F_5^{(2)}$	$F_6^{(2)}$	$F_7^{(2)}$	$F_8^{(2)}$	$F_1^{(3)}$	$F_2^{(3)}$	$F_3^{(3)}$	$F_4^{(3)}$	$F_5^{(3)}$	$F_6^{(3)}$	$F_7^{(3)}$	$F_8^{(3)}$
1	1	2	3	5	8	13	21	1	1	2	4	7	13	24	44

Table 3.14. The left side is identical to part of Table 3.12, but with the order included in the Fibonacci notation.

These higher-order Fibonacci numbers can be used to generate higher-order analogs of the Zeckendorf representation, with the property that they have no runs of k consecutive 1's if $k \geq m$. Thus an order-3 representation has no runs of 3 or more 1's. Assume throughout this section that the Fibonacci numbers F_k are of order-3, unless otherwise stated.

A simple order-m code for N is simply $Z(N)$ (bits in either order), followed by a 0 and then m 1's. This code is not that efficient, but is useful as an introduction to the better codes of the following sections.

3.10.4 Apostolico and Fraenkel Codes

Apostolico and Fraenkel [11] develop several codes using the higher-order Fibonacci numbers. Their emphasis is not so much on universal codes per se but rather on codes that are robust under occasional data corruption. The discussion here is restricted to order-3 representations, whereas they consider the general case of order-m codes.

They describe two codes, a C_1 code (here AF_1) and then a C_2 code (here AF_2), which is simpler than the AF_1 code. The description here is slightly different from theirs; the original paper should be referred to for much of the underlying theory and justification.

Apostolico and Fraenkel assume that $F_1 = 2$ for all $m \geq 2$. As the *Fibonacci Association* convention is that $F_1 = F_2 = 1$ and $F_3 = 2$ for $m = 2$, the discussion here assumes that all Fibonacci sequences start with $\{F_1 = 1, F_2 = 1, F_3 = 2, \ldots\}$, preceded by an appropriate number of 0's for $m > 2$.

Table 3.15 shows examples of the Apostolico and Fraenkel codes (their C_2 and C_1 codes, here AF_2 and AF_1 codes, for order-3) and some new Fibonacci codes from Section 3.10.5.

Table 3.15 Apostolico and Fraenkel's Codes, with New Fibonacci Codes

	Apostolico and Fraenkel		New Fibonacci Codes	
N	AF_2 Order-3	AF_1 Order-3	Order-2	NF_3 Order-3
1	11	111	11	111
2	1011	0111	011	01111
3	10011	00111	0011	11110
4	11011	10111	1011	001110
5	100011	000111	00011	101110
6	101011	010111	10011	01111
7	110011	100111	01011	0001110
8	1000011	110111	000011	1001110
9	1001011	0000111	100011	0101110
10	1010011	0010111	010011	1101110
11	1011011	0100111	001011	001111
12	1100011	0110111	101011	101111
13	1101011	1000111	0000011	00001110
14	10000011	1010111	1000011	10001110
15	10001011	1100111	0100011	01001110
16	10010011	00000111	0010011	11001110

Table 3.16 Development of the Apostolico–Fraenkel AF_1 Codes

k	1	2	3	4	5	6	7	8
F_k	1	1	2	4	7	13	24	44
$S = \sum F_k$	1	2	4	8	15	28	52	96
Range	—	—	3–4	5–8	9–15	16–28	29–52	53–96

3.10.4.1 The Apostolico–Fraenkel AF_2 Codes

These codes represent the value 1 with $(m-1)$ consecutive 1's. Larger values are encoded as the Zeckendorf representation $Z(N-1)$, most-significant bit leading, followed by a suffix of 0 and then $(m-1)$ 1's. (The termination comes from these 1's and the first 1 of the next codeword; the code is not instantaneous.)

3.10.4.2 The Apostolico–Fraenkel AF_1 Codes

These codes involve a transformation or mapping to remove many awkward codewords. To generate the order-3 codes start with the order-3 Fibonacci numbers $F_k^{(3)}$ from Table 3.14 and calculate the cumulative sums $S = \sum F(k)$ of those numbers, as shown in Table 3.16.

To encode a value N:

1. Find k such that $S_{k-1} < N \leq S_k$.
2. Find $Q = N - S_{k-1} - 1$.
3. Encode $F_{k+1} + Q$ in an order-3 Zeckendorf representation, most-significant bit first.
4. Delete the leading 10 from this codeword and attach the suffix 0111 as terminator. (The codeword *always* has a prefix of 10, by virtue of step 2.)

The values 1 and 2 have the special codewords $1 \rightarrow 111$ and $2 \rightarrow 0111$.

Table 3.17 A Range of Order-3 Zeckendorf Representations

	Digit Weights						Digit Weights				
N	1	2	4	7	13	N	1	2	4	7	13
13	0	0	0	0	1	19	0	1	1	0	1
14	1	0	0	0	1	20	0	0	0	1	1
15	0	1	0	0	1	21	1	0	0	1	1
16	1	1	0	0	1	22	0	1	0	1	1
17	0	0	1	0	1	23	1	1	0	1	1
18	1	0	1	0	1						

To encode the value 11 (their example):

1. Find k such that $S_{k-1} < 11 \leq Sk$; $k = 5$, $S_{k-1} = 8$.
2. $Q = 11 - 8 - 1 = 2$.
3. Encode $F_{k+1} + Q = 13 + 2 = 15$ in an order-3 Zeckendorf representation, giving 10 010.
4. Delete the leading 10 and add the suffix 0111 to give 010 0111 as the final codeword.

Again, to encode 40, first find $N = 7$ and $Q = 11$, and then encode $44 + 11 = 55$ to give first 1001100 and thence 01100 0111 as the final codeword.

The second step ($Q = N - S_k - 1$) needs explanation. Consider the order-3 representations of $F_k \leq N < F_{k+1}$ as shown in Table 3.17 for the range $13 \leq N < 24$ (i.e., $F_6 \leq N < F_7$), with the digit weights in the first row and greater weights to the right. By the Fibonacci definitions, there are $(F_{k-1} + F_{k-2})$ values in this range; the F_{k-1} smaller ones end with ...01 and the F_{k-2} larger with ...11. The adjustment $Q = N - S_k - 1$ eliminates all representations with most-significant bits ...11. Thus a received bit sequence ...0111 *always* corresponds to the numeric bits ...01.

The AF_1 code is then just 2 bits longer than the Zeckendorf representation, whereas the simpler AF_2 code is 3 bits longer than the Zeckendorf representation after absorbing the most-significant 1 bit into its terminator 1110. By discarding some of the possible codewords the AF_1 code is slightly longer for larger values. Against this it is inherently 1 bit shorter than the simpler code; the two effects largely cancel.

From Table 3.1, the Fibonacci-3 (Tribonacci) codes have an effective base $b = 1.839$ and an expansion $x = 1.137$. But they follow the general trend that a more compact code (smaller x) tends to have a more complex length indication that may offset (or even overwhelm) the smaller expansion.

The first order-3 Zeckendorf code above presented the bits in increasing significance, followed by the suffix 0111. In the AF_1 code the bits are presented in *decreasing* significance, again with a suffix 0111. Now consider the AF_1 code with its bits in *increasing* significance, so that the code for 11 is 010 0111 and for 40 is 11010 0111. But these codes are, respectively, 01001 11 and 1101001 11, shifting the break to give a different emphasis to the two components. As both are the representations of $F_{k+1} + Q$ with a suffix of 11, an alternative way of generating a code equivalent to the AF_1 code is to generate $Z(F_{k+1} + Q)$, least-significant bit first, and append a suffix 11.

3.10.5 A New Order-3 Fibonacci Code

In contrast to the rest of the chapter, this section introduces a new, unpublished code as an alternative to existing codes.

Table 3.18 Terminators for Order-5 Fibonacci Code

Numerical Bits	Final Code
...01	...01.1111.0
...011	...011.111.10
...0111	...0111.11.110
...01111	...01111.1.111

As the order-3 Fibonacci codes may end with either one or two consecutive 1's, the terminator must allow the two cases to be distinguished. Apostolico and Fraenkel solve the problem by eliminating those codes whose Zeckendorf representations end in ...11.

For the new code (the NF_3 code), transmit the order-3 Zeckendorf representation least-significant bit first (using $F(N)$) and follow its most significant 1 bit with a suitable comma or terminator as described later. With the order-2 code, the most-significant bit pattern (LSB first) is always ...01 and a single 1 bit acts as an unambiguous terminator. With the order-3 code the most significant bit pattern may be either ...011 or ...01. The terminator must be a run of 111 but it is also necessary to decide how many of those 1's have numeric significance.

The FN_3 code uses the following rules for the terminator:

- If $F(N)$ ends with ...01, add the terminator 110, so that the codeword ends with ...01 110.
- If $F(N)$ ends with ...011, add the terminator 11, so that the codeword ends with ...011 11.

The bit immediately after the terminating sequence 111 indicates how many numeric 1's to retain. But the isolated terminator 111 can be retained as a unique representation for the minimum value. It always follows immediately from another terminator, occurs at the start of the codeword, and can be decoded without ambiguity. While the codeword lengths are similar to those of the Apostolico and Fraenkel C_2 code (a few are 1 bit shorter), the NF_3 code is much simpler to generate. Examples of the NF_3 code were shown earlier in Table 3.15.

The principle can be extended to higher-order Fibonacci codes, but with increasingly expensive terminators. An order-m code must be built out to have m terminating 1's, but then needs a code to say how many of those 1's are numerically significant. The result is that an order-5 code needs on average a 5-bit terminator, as shown in Table 3.18, with dots inserted to separate the components of the terminator. The terminator lengths in Table 3.18 are Huffman coded according to their probabilities. While the order-5 code is inherently quite efficient, its costly terminator means that it is shorter than the order-3 Fibonacci code only for values over about 1 million. Fibonacci codes of order greater than 3 are expected to be useful only in special circumstances.

3.11 TERNARY COMMA CODES

Fenwick [14] shows that bit-pairs can represent the values {0, 1, 2, comma}, leading to a base-3 or ternary code with the digit 3 reserved as a *comma* or terminating code. Table 3.19 shows the ternary comma code representation for the first few integers and some larger ones, with "c" representing the comma. (Although the comma "c" is encoded as the digit 3 or bits 11, it is represented by c to emphasize its non-numerical nature.)

The comma principle can be extended to larger number bases, by sacrificing one of the digits to use as the comma, but becomes increasingly inefficient for small values because the comma consumes a large amount of code space but conveys little information.

Table 3.19 Ternary Codes for the Various Integers

Value	Code	Bits	Value	Code	Bits
1	1c	4	11	102c	8
2	2c	4	12	110c	8
3	10c	6	13	111c	8
4	11c	6	14	112c	8
5	12c	6	15	120c	8
6	20c	6	16	121c	8
7	21c	6	17	122c	8
8	22c	6	18	200c	8
9	100c	8	19	201c	8
10	101c	8	20	202c	8
64	2101c	10	1000	1101001c	16
128	11202c	12	3000	11010010c	18
256	100111c	14	10000	111201101c	20
512	200222c	14	65536	10022220021c	24

Each *ternary* digit of the input value requires 2 bits, giving an effective base $b = \sqrt{3}$ and an expansion relative to natural binary of 1.262, as shown in Table 3.1. To this must be added the constant overhead of 2 bits for the terminating comma.

If the ternary digits are transmitted most-significant first, the first digit must be a 1 or 2 and never 0 or c. Thus a single 0 (bits 00) may be used as a 2-bit representation for a 0 value. An initial c (bits 11) may be used as an escape code or to represent the value 1 (and encoding $N - 1$ for larger values).

3.12 SUMMATION CODES

This class of codes starts with a set of integers and represents values as the sum of a fixed number of members of this set; it is fully described by Fenwick [16].

Just as the binary representation is based on powers of 2 and the Zeckendorf representation on Fibonacci numbers, the summation codes are, initially at least, based on the prime numbers, using the Goldbach conjecture that all even numbers are the sum of two primes.

The Elias α codes have the termination condition *stop at the first 1 bit*. The summation codes may be regarded as a generalization of the α codes and stop at the *second* 1 bit.

The first code is a very simple one, but one that is surprisingly efficient for small values. For later reference it will be known as the G_0 code. As the sum of the primes is always even, encode twice the represented value, for example, 5 as $10 = 3 + 7$. We use the slightly modified table of primes shown in Table 3.20 which forces all sums to be even by omitting the usual value 2. Including a 1 in the set of primes does allow 2 and 3 to be encoded, but adds an extra bit to *all* other codeword lengths and is, overall, not justified.

Then 5 is represented as 101 and 14 ($28 = 11 + 17$) as 000101. The first few codewords are shown in Table 3.21. As very small values are not representable at all, it is usually necessary to encode with an offset.

For small values (essentially those shown), the G_0 code is one of the shortest codes, although the α coding of the first prime index makes it less efficient for values much beyond 15.

Table 3.20 The First Few Prime Numbers, and Indices

Value	3	5	7	11	13	17	19	23
Index	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Value	29	31	37	41	43	47	53	59
Index	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}

Table 3.21 Simple Goldbach Summation (G_0) Codes

N	Sum	Code	N	Sum	Code
1	—	—	9	7 + 11	0011
2	—	—	10	7 + 13	00101
3	—	—	11	5 + 17	010001
4	3 + 5	11	12	11 + 13	00011
5	3 + 7	101	13	7 + 19	0010001
6	5 + 7	011	14	11 + 17	000101
7	3 + 11	1001	15	13 + 17	000011
8	5 + 11	0101	16	13 + 19	0000101

3.12.1 Goldbach G_1 Codes

The simple summation code used a direct encoding of the bit positions, but suffered because of the lengths of the 0 runs. Larger values are handled by encoding the lengths of the two 0 runs as γ codes (which are the shortest for small values or short runs). For example, as $20 = 7 + 13$, represent 20 as the two indices 4 and 6, or 4 and $(6 - 4)$. Encoding these two components as a γ' code gives 0010 010 as the representation of 20.

But this is not the only representation, as shown in Table 3.22. For many encoded values it is necessary to choose the combination of primes to give the shortest representation.

This G_1 code is limited to even integers (by the Goldbach conjecture itself), although like the G_0 code it is possible to encode twice the value. Fenwick then describes a modification that encodes any integer. Assuming that p and q are the indices of the primes which add to the integer N, there are several cases to consider. The values p and $q - p$ are both incremented by 1 and then encoded as a γ' code.

Small values: Encode 1 as 110 and 2 as 111.

Even integer: This the sum of the two primes, P_p and P_q. The value is encoded as the concatenation of $p + 1$ and $q - p + 1$.

Prime integer (odd): Here the integer $N = P_i$; emit i (as a γ code) and then a single 1 bit.

Odd integer (non-prime): Emit an initial 1 bit and the i, where $N - 1 = P_i$.

The resultant Goldbach codes are competitive for a range of values, approximately $4 < N < 100$.

3.12.2 Additive Codes

After describing the Goldbach codes, Fenwick develops a family of related codes where all integers within a range can be represented as the sum of two members of a set of carefully selected *basis integers*. The integer is then encoded as the concatenation of the γ' codes of the two components,

Table 3.22 Some Goldbach G_1 Representations, as Sums of Primes

Value	Sums	Indices	Encode As	Coding	Length
18	1 + 17	1,7	1,7	1 00111	6
...	5 + 13	3,6	3,4	011 00100	8
...	7 + 11	4,5	4,2	00100 010	8
20	1 + 19	1,8	1,8	1 0001000	8
...	3 + 17	2,7	2,6	010 00110	8
...	7 + 13	4,6	4,3	00100 011	8
36	5 + 31	3,11	3,9	011 0001001	10
...	7 + 29	4,10	4,7	00100 00111	10
...	13 + 23	6,9	6,4	00110 00100	10
...	17 + 19	7,8	7,2	00111 010	8
42	1 + 41	1,13	1,13	1 0001011	8
...	5 + 37	3,12	3,10	011 0001010	10
...	11 + 31	5,11	5,7	00101 00111	10
...	13 + 29	6,10	6,5	00110 00101	10
...	19 + 23	8,9	8,2	0001000 010	10

Table 3.23 Basis Integers for Additive Code Add_{256}

0	1	3	5	7	9	10	21
23	25	27	28	39	41	43	45
47	50	58	76	87	98	102	106
118	120	122	124	135	154	166	183
202	214	231	250				

36 values (54 primes) Seeds = 10, 28, 50

as for the first Goldbach code. A typical set of basis integers to encode integers up to 256 is shown in Table 3.23.

The Additive code Add_m is designed to encode integers $N \leq m$ (and might not encode $N > m$). The set of basis integers is generated by a sieve, recording the values that can be generated by adding members of the set-so-far. A value which cannot be formed is included as the next member of the basis set. Suitable seeds are introduced to facilitate code generation; seed values are found by a computer search to minimize, for example, the average length of the generated code.

3.13 WHEELER 1/2 CODE AND RUN-LENGTHS

The codes of this section are embedded in a character stream and encode the length of run-length compressed strings.

A simple method, used in some Burrows–Wheeler compressors, triggers run-length encoding by a succession of say four identical characters and then encodes the length in a base-64 γ code. For example, a run of 50 A's would have the hexadecimal coding "`41 41 41 41 72`" and 1000 X's "`58 58 58 58 28 4F`."

The other codes to be described are based on Wheeler's 1/2 code described by Wheeler [15] and used in his early `bred` implementation of the Burrows–Wheeler compressor. Wheeler states that he has used this code for some years, but its provenance is otherwise unknown. The author notes that he has received many comments as "I just don't understand Wheeler's 1/2 code—however does it work?"

3.13.1 The Wheeler 1/2 Code

In a standard binary code all 0's have a weight of 0, and 1's have weights 2^0, 2^1, 2^2, 2^3, ..., respectively. In the Wheeler code successive 0s have weights 2^0, 2^1, 2^2, 2^3, ... and successive 1's have weights 2^1, 2^2, 2^3, 2^4, ... The code is terminated by any character other than 0 or 1. The first few encodings are shown in Table 3.24. The name arises because the first bit has a weight of 1 or 2, which Wheeler writes as 1/2.

Still using "/" to mean "or," the respective digit weights can be written as $(0 + 1)/(1 + 1)$, $(0+2)/(2+2)$, $(0+4)/(4+4)$, etc. Thus each weight is a standard binary weight, *plus* a constant 2^k weight for digit k. In a representation of n bits these constant weights add to $2^n - 1$; adding a further constant 1 turns these into 2^n or a single more-significant 1. Thus the Wheeler code for N is the same as the binary code for $N + 1$ with the most-significant 1 deleted, as may be seen from the last column of Table 3.24. The Wheeler 1/2 code is little more than a reinterpretation of a binary representation.

A simpler way of generating the Wheeler code for N is to encode $N + 1$ in binary and delete the most-significant or final bit. When decoding, the number is handled as binary, another 1 is implied by whatever terminates the sequence, and a 1 is subtracted.

3.13.2 Using the Wheeler 1/2 Code

Wheeler gives two ways of incorporating his code into a stream of symbols. The first is appropriate to normal text, while the second is more useful when encoding runs of 0's in the compressed output.

Table 3.24 Examples of Wheeler 1/2 Code

Bit Weights				Value	Reverse Binary
1/2	2/4	4/8	Sums	N	$N + 1$
0			1	1	01
1			2	2	11
0	0		1+2	3	001
1	0		2+2	4	101
0	1		1+4	5	011
1	1		2+4	6	111
0	0	0	1+2+4	7	0001
1	0	0	2+2+4	8	1001
0	1	0	1+4+4	9	0101
1	1	0	2+4+4	10	1101
0	0	1	1+2+8	11	0011
1	0	1	2+2+8	12	1011
0	1	1	1+4+8	13	0111
1	1	1	2+4+8	14	1111

1. The binary representation of the run symbol, say x, is used to encode 0 bits in the run-length and the value $x \oplus 1$ is used for the 1 bits, emitting least-significant bits first. ($\oplus$ represents an Exclusive-Or). Any character not in the set $\{x, x \oplus 1, x \oplus 2\}$ terminates the number (and implies the most-significant 1 bit with the simple binary decode). An $x \oplus 2$ is used as count terminator if the run is terminated by $x \oplus 1$ or $x \oplus 2$, and it is followed immediately by the terminating character.
   When using this code for run compression it seems desirable to trigger it from a run of at least three identical characters. The character must occur once to establish the coding and is then confused by sequences such as "ba". Similarly digram triggering is confused by "eed" or "oom"; three or four characters seem to be necessary.
   With a trigger of 4 symbols, encode (length-3). The character sequence "...xbbbbbbbb bbby..." with a run of 11 b's might be encoded as "xbbbb*cbby*...", and the sequence "...xttttttu...", with a run of 7 t's, becomes "...xttt*uu*vu...". (Experience is that this run-encoding is justified only if it gives a considerable decrease in file size.)
2. The Wheeler 1/2 code is also used to encode runs of 0's in the final coder, where the output is dominated by 0's, with many runs of 0's and negligible runs of other symbols. Encode the symbol N as $(N + 1)$ for all $N > 0$. Then *any* occurrence of a 0 or 1 indicates a run of 0's, with 0 and 1 symbols always representing the run-length in a Wheeler 1/2 code. The count is terminated by any symbol $N > 1$. While there is a slight increase in average codeword length from encoding N as $N + 1$, this encoding *never* increases the string length.

3.14 COMPARISON OF REPRESENTATIONS

There is no best variable-length coding of the integers, the choice depending on the probability distribution of the integers to be represented. A fundamental result of coding theory is that a symbol with probability P should be represented by $\log(1/P)$ bits. Equivalently a symbol represented by j bits should occur with a probability of 2^{-j}. The two simplest functions that produce a low value for large arguments n are a power function (n^{-x}) and an exponential function (x^{-n}). In both cases normalizing factors are needed to provide true probabilities, but these do not affect the basic argument.

From Table 3.1 a polynomial code represents a value n with about $x \log n$ bits. The probability P_n of an integer n should therefore be $P_n \approx n^{-x}$, showing that a γ code is well suited to power law distributions and especially where the exponent is about -2, and the order-2 Fibonacci codes to a distribution $P_n \approx n^{-1.44}$.

A Rice code represents a value n with about $((n/2^k) + k + 1)$ bits, so that $-\log P_n \approx (n/2^k) + k + 1$. For a given k, $P_n \propto 2^{-n}$, showing that the Rice code is more suited to symbols following an exponential distribution.

But a Rice code (and by extension a Golomb code) is very well suited to peaked distributions with few small values or large values. As noted earlier, the Rice(k) code is extremely efficient for values in the general range $2^{k-1} < N < 2^{k+2}$.

Most of the other codes are true polynomial codes (γ and Fibonacci) or approximate a polynomial behavior (ω code) and are appropriate for monotonically decreasing symbol probabilities. The results for the polynomial codes are summarized in Table 3.25, giving their code lengths as functions of the corresponding binary representation. The table is based on the same range of values as Table 3.26, but with values of $\lfloor 1.1^n \rfloor$ for a good spread of values. The measured lengths are least-square fitted to the binary lengths. The theoretical expansion factor is taken from the earlier Table 3.1. Two points follow from Table 3.25:

Table 3.25 Measured Parameters of Polynomial Codes

	Expansion (x)		Bias
	Theory	Measured	d
Elias γ	2.000	2.000 ± 0	-1 ± 0
Punctured P1	1.500	$1.478 \pm .032$	$1.53 \pm .37$
Fraenkel C1	1.440	$1.436 \pm .010$	$0.50 \pm .12$
Elias ω	—	$1.386 \pm .024$	$2.61 \pm .27$
Ternary	1.262	$1.255 \pm .015$	$2.43 \pm .17$
Apostolico C1	1.137	$1.142 \pm .009$	$2.36 \pm .10$
Apostolico C2	1.137	$1.150 \pm .010$	$2.53 \pm .11$
New NF_3	1.137	$1.131 \pm .016$	$2.50 \pm .18$

If an integer N has a binary length of ℓ_2 bits, its predicted code length is $\ell_P = x\ell_2 + d$ bits.

Table 3.26 Comparison of Representations—Codeword Lengths Over 1:2 Value Ranges

	Elias				Fibonacci	Fibonacci, Order-3			Goldbach	Additive	Best
Range of Values	β	γ	ω	Ternary	FK_2	AF_1	AF_2	NF_3	G_1	Add_{256}	Length
1	0	1	1	4	2	3	3	3	3	2	1
2	1	3	3	4	3	3	4	4	3	4	3
3	1	3	3	6	4	5	5	5	4	4	3
4–7	2	5	6	6	4.8	5.8	5.8	6	5.5	5	4.8
8–15	3	7	7	7.7	6.4	6.9	7.3	7.1	7.3	6.8	6.4
16–31	4	9	11	8.6	7.7	8.1	8.4	8.1	9.1	7.6	7.6
32–63	5	11	12	10	9.1	9.3	9.6	9.3	10.9	9.9	9.1
64–127	6	13	13	11.1	10.4	10.4	10.6	10.3	12.7	11.4	10.3
128–255	7	15	14	12.3	11.9	11.5	11.8	11.4	14.8	13.9	11.4
256–511	8	17	16	14	13.4	12.7	13	12.7			12.7
512–1,023	9	19	17	14.9	14.7	13.7	14.1	13.9			13.7
1,024–2,047	10	21	18	16	16.3	14.9	15.1	14.9			14.9
2,048–4,095	11	23	19	17.8	17.6	16.1	16.4	16.1			16.1
4,096–8,191	12	25	20	18.6	19.3	17.3	17.6	17.4			17.3
8,192–16,383	13	27	21	20	20.6	18.3	18.6	18.3			18.3
16,384–32,767	14	29	22	21.5	22.1	19.5	19.8	19.4			19.4
32,768–65,535	15	31	23	22.3	23.6	20.7	20.9	20.4			20.4
65,536–131,071	16	33	28	24	25	21.7	22.1	21.9			21.7

- The observed polynomial expansions (and therefore the bases) are very close to those predicted by theory. (Even though the ω code is not strictly a polynomial it behaves as one in these measurements. The correlation between expansions is about 98% for the ω code, compared with 99% or better for the others.) In this respect the codes are just as expected.
- It was noted earlier that the more complex and efficient codes (those with an expansion closer to 1) need more costly length specifications. This is clear for the last column, which is largely the constant overhead of the length or terminator.
  The bias is especially small for the order-2 Fibonacci code, at only half a bit. When coupled with the moderately high base, this gives good performance for small values; this code is

the best for values up to 200. Beyond 200 the order-3 Fibonacci code is shortest; its low expansion is enough to counter the overhead of the termination.

The codeword lengths of the codes are shown in Table 3.26, using the same data as underlies Table 3.25. The β code is included as a lower bound or ideal against which other codes may be compared; the γ code is, for larger values, much like an upper bound. For most codes, and except for the first three rows, each row is the average of a range of values increasing by a factor of 1.1 (but limited to different integers) within ranges 2^k to 2^{k-1} (about 7 values per "octave"). This choice of ranges exactly matches the Elias codes with a binary base, but not the ternary or Fibonacci codes whose averaged lengths thereby have a form of sampling error or uncertainty.

The Goldbach and Additive codes are handled differently. Most obviously the largest values are $N = 256$, this being the limit of the Additive code using the basis values of Table 3.23; at this value the Goldbach $G0$ codeword length is starting to "run away." Less obviously, the lengths are averaged over all values in each range. The lengths vary widely and averaging only a few leads to a form of sampling error or aliasing.

We can, however, make several observations:

1. The γ and ω codes are best for very small values, certainly to 4 and possibly to 8.
2. The order-2 Fibonacci codes are the best for larger values, to 128 and to 256 with little inefficiency. The ternary codes are nearly as good within this range.
3. For values beyond 128 the order-3 Fibonacci codes are the best codes once their inherent coding efficiency overcomes the relatively expensive termination cost. The best code alternates between the older Apostolico–Fraenkel AF_1 and the new Fibonacci code; any differences are small and probably due to the sampling error mentioned earlier. The newer code with its more complex terminator is probably preferable to the older Apostolico–Fraenkel code with its more complex generation.
4. Over much its given range the Goldbach G_1 code is similar to the Elias γ code, but distinctly inferior to the other good codes. The Additive(256) code, tuned to the range shown, is very competitive over much of its range (and is marginally best in one case) but is again starting to run away for large values.

The best general codes in these tests are the order-2 Fibonacci codes. They have a simple structure, have minimal overheads for length indication, are easy to understand, and over a wide range of values give the most compact representation or are close to the best.

3.15 FINAL REMARKS

"Good" codes are simple, with a minimum of non-numeric bits such as length indication and termination. The complexity, as well as quantity of non-numeric bits, counts against the Elias ω and higher-order Fibonacci codes. For general use the Fraenkel and Klein C^1 Fibonacci code (the FK_1 code) is a good choice, as shown by its performance for both versions of the Burrows–Wheeler compression. The Elias γ code is good for *very* skewed distributions. Rice and Golomb codes are an excellent choice where values are confined to a range of perhaps 1:10. The additive codes are useful where only small values (say $N < 250$) occur.

ACKNOWLEDGMENTS

The author thanks Dr. Robert Rice for his assistance in obtaining original documents relating to his codes.

3.16 REFERENCES

1. Bell, T. C., J. G. Cleary, and I. H. Witten, 1990. *Text Compression*, Prentice-Hall, Englewood Cliffs, NJ.
2. Salomon, D., 2000. *Data Compression: The Complete Reference*, 2nd Ed., Springer-Verlag, Berlin/ New York.
3. Levenstein, V. E., 1968. On the redundancy and delay of separable codes for the natural numbers. *Problems of Cybernetics*, Vol. 20, pp. 173–179.
4. Elias, P., 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, Vol. IT-21, No. 2, pp. 194–203, March 1975.
5. Even, S., and M. Rodeh, 1978. Economical encoding of commas between strings. *Communications of the ACM*, Vol. 21, No. 4, pp. 315–317, April 1978.
6. Bentley, J. L., and A. C. Yao, 1976. An almost optimal solution for unbounded searching. *Information Processing Letters*, Vol. 5, No. 3, pp. 82–87, August 1976.
7. Rice, R. F., 1979. Some practical universal noiseless coding techniques. Jet Propulsion Laboratory, JPL Publication 79-22, Pasadena, CA, March 1979.
8. Golomb, S. W., 1966. Run-length encodings. *IEEE Transactions on Information Theory*, Vol. 12, pp. 399–401.
9. Fiala, E. R., and D. H. Greene, 1989. Data compression with finite windows. *Communications of the ACM*, Vol. 32, No. 4, pp. 490–505, April 1989.
10. Vajda, S., 1989. *Fibonacci and Lucas Numbers, and the Golden Section Theory and Applications*, Ellis Horwood, Chichester.
11. Apostolico, A., and A. S. Fraenkel, 1987. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Transactions on Information Theory*. Vol. IT–33, pp. 238–245.
12. Fraenkel, A. S., and S. T. Klein, 1996. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, Vol. 64, pp. 31–55.
13. Knuth, D. E., 1997. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, 3rd ed, Addison-Wesley, Reading, MA.
14. Fenwick, P. M., 1993. Ziv–Lempel encoding with multi-bit flags. In *Proceedings of the Data Compression Conference, DCC-93*, Snowbird, UT, pp. 138–147, March 1993.
15. Wheeler, D. J., 1995. Private communication. Also "An Implementation of Block Coding," October 1995, privately circulated.
16. Fenwick, P. M., 2002. Variable length integer codes based on the Goldbach conjecture and other additive codes. *IEEE Transactions on Information Theory*, Vol. 48, No. 8, August 2002.
17. Burrows, M., and D. J. Wheeler, 1994. A block-sorting lossless data compression algorithm, SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, May 1994. Available at, ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z.