Even if you're on the right track, you'll get run over if you just sit there.
Will Rogers

Pegasus Imaging Corporation (PIC) is proud to present ImpacStar ^{TM} (IM∗), our trade name for a wonderful new image format incorporating the patent-pending ELS (*E*ntropy *L*ogarithmic-*S*cale) coding algorithm developed by one of our research team. I think you will agree with me that this is a stunning, eloquent algorithm which has applications in lossless compression of low-color (2–256) images or as a replacement for Huffman coding with JPEG or other compression schemes. The ImpacStar ^{TM} name is in itself a play on the DOS wildcard ∗ to denote the broad applicability of this image format.

Pegasus is best known for its licensing of advanced JPEG compression algorithms to software developers. However, as I hope the ELS-coder paper points out, we have dedicated much of our resources to developing technology which better answers the problems of image or data compression. Our own plans for IM∗ include providing developer toolkits along with our own suite of end-user products which will take advantage of ImpacStar ^{TM} coding. For one- to eight-bit image types the savings over G4 (fax) or LZW (GIF) should be valuable, especially in low-bandwidth environments, such as the Internet. For this very reason, our first release of a commercial ImpacStar ^{TM} decoder will be progressive in nature (comparable to interlaced GIF or progressive JPEG) and allow for embedded comments, such as text overlays and alpha channels for transparencies. Our release of a DCT (JPEG) compressor using the ELS-coder algorithm will offer an additional lossless improvement to any compression ratio. Further, even though the resultant compressed file will no longer be JFIF compliant, converting to/from is a lossless translation.

We often describe Pegasus as a "company with great technology in brown paper bags," referring to our lack of "packaged" applications utilizing our own research. Hopefully, this will become an untruth over the coming months. Please enjoy this peek into our new brown-wrapped package. None of us that have been involved with the microcomputer revolution should be heard to say: "this *is* the way we do things, because this *is* the way we have always done things!" This goes for GIF, JPEG or whatever. We invite your comments, ideas and inquiries because we can and will do more.

Jack Berlin
President
Pegasus Imaging Corporation

For licensing information, please contact:

Steve Armstrong, Sales Manager
Pegasus Imaging Corporation
4350 West Cypress Street
Tampa, FL 33607

e-mail: sarmstr@jpg.com
(813) 875-7705 Fax

http://www.jpg.com/

(813) 875-7575 Phone

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

i

A RAPID ENTROPY-CODING ALGORITHM

WM. DOUGLAS WITHERS

Department of Mathematics
United States Naval Academy
and
Pegasus Imaging Corporation

ABSTRACT. We present a new algorithm for entropy coding, competitive in both speed and compression efficiency with the Q-coder and QM-coder.

1. ENTROPY CODING

Truth is the most valuable thing we have. Let us economize it.

Mark Twain

The problem of compressing digital data of a certain type (text, images, audio data, whatever) can be decoupled into two subproblems: *modeling* and *entropy coding*. Whatever the given data may represent in the real world, in digital form it exists as a sequence of symbols, such as bits. The modeling problem is to choose a suitable symbolic representation for the data and to predict for each symbol of the representation the probability that it takes each of the allowable values for that symbol. The entropy-coding problem is to code each symbol as compactly as possible, given this knowledge of probabilities. (In the realm of lossy compression, there is a third subproblem: evaluating the relative importance of various kinds of errors.)

For example, suppose we want to transmit messages composed of the four letters **a**, **b**, **c**, and **d**. A straightforward scheme for coding these messages in bits would be to represent **a** by "00", **b** by "01", **c** by "10" and **d** by "11". However, suppose we know that for any letter of the message (independent of all other letters), **a** occurs with probability .5, **b** occurs with probability .25, and **c** or **d** occur with probability .125 each. Then we might choose a shorter representation for **a**, at the necessary cost of accepting longer representations for the other letters. We could represent **a** by "0", **b** by "10", **c** by "110", and **d** by "111". This representation is more compact on average than the first one; indeed, it is the most compact representation possible (though not uniquely so). In this simple example, the modeling part of the problem is determining the probabilities for each symbol; the entropy-coding part of the problem is determining the representations in bits from those

probabilities; let us emphasize that *the probabilities associated with the symbols play a fundamental role in entropy coding.*

In general, entropy coding is an abstract problem weakly related to the type of data being compressed, while the modeling aspect of data compression depends intimately on the type of data being compressed. Entropy coding is well understood theoretically—known algorithms provide the greatest compression possible *for a given modeling method*—while for many real-world types of data, the modeling issue is as yet mysterious and only somewhat tractable. This article treats only entropy coding and the most rudimentary aspect of modeling; thus it provides a partial solution to almost all data compression problems but a complete solution to none.

One well-known method of entropy coding is *Huffman coding*, which yields an optimal coding provided all symbol probabilities are integer powers of .5. Another method, yielding optimal compression performance for any set of probabilities, is *arithmetic coding*. In spite of the superior compression given by arithmetic coding, so far it has not been a dominant presence in real data-compression applications. This is most likely due to concerns over speed and complexity, as well as patent issues; a rapid, simple algorithm for arithmetic coding is therefore potentially very useful.

An algorithm is known which allows rapid encoding and decoding in a fashion akin to arithmetic coding. Developed by G. G. Langdon, J. L. Mitchell, W. B. Pennebaker, and J. J. Rissanen [4], [6], [7], it is known as the *Q-coder.* The *QM-coder* is a subsequent variant. However, these algorithms being protected by patents, new algorithms with competitive performance continue to be of interest. The algorithm described here is one such; we call it the ELS-coder (for *E*ntropy *L*ogarithmic-*S*cale). Though the ELS-coder is likewise protected by a pending patent, Pegasus Imaging Corporation intends to license it on a royalty-free basis.

Due to space limitations, we do not provide a general introduction to data compression theory, nor a description of other algorithms for entropy coding. Witten and Cleary [5] provides an introduction to the "standard" arithmetic compression algorithm. Pennebaker and Mitchell [3] provides a good introduction to the operation of the QM-coder. Nelson [1] and Cover and Thomas [2] are general practical and theoretical introductions to data compression.

2. THE DECODING ALGORITHM

I were better to be eaten to death with rust than to be scoured to nothing with perpetual motion.

Shakespeare

The ELS-coder works only with an alphabet of two symbols (which we call **0** and **1**). One can certainly encode symbols from larger alphabets; but they must be converted to a two-symbol format first; the example programs **compress** and **decompress** illustrate how this is done. The necessity for this conversion is a disadvantage, but the restriction to a two-symbol alphabet facilitates rapid coding and rapid probability estimation.

Our algorithm was developed with an eye toward a particular application which would generally involve compressing a dataset once and subsequently decompressing it many

times. Thus decoding speed was of somewhat greater concern to us than encoding speed. We therefore designed the decoder first, and then designed the encoder to fit. We likewise describe the decoder first.

```
int bitCount;
int threshold[10] = {0,1,2,4,8,16,32,64,128,256};
int allowable[] = threshold+1; /* For expository purposes only */
FILE *externalIn;
unsigned char internalBuffer;

int decodeBit(){
    if (internalBuffer>=threshold[bitCount]){
        internalBuffer -= threshold[bitCount];
        if ((--bitCount)<=0)
            decodeImport();
        return 1;
    }else{
        if ((--bitCount)<=0)
            decodeImport();
        return 0;
    }
}

void decodeImport(){
    internalBuffer = fgetc(externalIn);
    bitCount += 8;
}
```

EXAMPLE 1.   A pair of procedures which extract bits from a file with no compression involved.

Let us consider first the procedures given in Example 1. This decoder does not as yet provide any compression; it simply extracts bits from a file (and not in the most efficient manner, either). The most-significant bits of each byte in the file are extracted first. We give it here purely as an illustration of our paradigm for the operation of a decoder. (Note: throughout this article, we assume that a `char` has eight bits of precision, a `short` has sixteen bits of precision, and a `long` has thirty-two bits of precision.)

In order for the decoder to hold data, it must be furnished with several states. It is convenient to represent the state of the decoder by two components: one indicating the quantity of data held by the decoder (`bitCount`) and another indicating the content of that data (`internalBuffer`). When called upon to decode a symbol (the procedure `decodeBit()`), the decoder is in one of several possible states. One subset of these states represents the symbol **0**, and another represents the symbol **1**. The decoder must determine to which subset its state belongs (by comparing `internalBuffer` to `threshold[bitCount]`). It then

transits to a new state to reflect the depletion of its store of data (by decrementing `bitCount` and, if necessary, decreasing `internalBuffer`). At times (when `bitCount` reaches the value 0) the decoder must replenish its store of data by reading from an external file (the procedure `decodeImport()`). This *import* operation modifies both `internalBuffer` and `bitCount`.

A few enhancements to this simple example will produce the ELS decoding algorithm. We first call attention to the table `allowable[]`, not directly used by the decoder but a convenient reference for discussion. It is the counterpart of a mathematical function we call S, relating the quantity of data in the encoder to the number of allowable states. A number with B bits of precision can take 2^B distinct values; thus

$$S(B) = 2^B.$$

In Example 1, `allowable[bitCount]` takes the value $S(B)$ when `bitCount` takes the value B. Thus there are `allowable[bitCount]` allowable values for `internalBuffer` at any time. The insight underpinning arithmetic coding is that this relationship between the number of bits in the decoder and the number of allowable states remains valid even when B is not an integer and $S(B)$ is not a power of two.

Note also: we apportioned the allowable values corresponding to **0** and **1**, respectively, so that all of the former values were less than all of the latter. Thus we can determine to which subset of values the value of `internalBuffer` belongs by comparing it to `threshold[bitCount]`, which is the minimum value of `internalBuffer` corresponding to **1**. When the symbol **0** is decoded, the number of allowable values for `internalBuffer` necessarily becomes the value of `threshold[bitCount]` before decoding the symbol.

Example 2 describes a full-fledged ELS-decoder, although at a crude level of accuracy. (The procedures for initializing and terminating the decoding process are not shown; straightforward, they can be found in the accompanying sample listings.) Let us note the changes from Example 1 to Example 2:

First: to compress symbols from a binary alphabet, we must work with quantities of data smaller than a bit. We therefore define a *jot* to be a unit of data equal to $1/F$ of a byte, F being an integer larger than 8. In Example 2, F is given the value 15, and is represented by the macro `JOTS_PER_BYTE`. In practice F normally takes much larger values. The decoder in Example 2 measures data in jots rather than bits.

For example, `allowable[jotCount]` now gives the number of allowable values for `internalBuffer` for a given number of jots. This is determined by the same relation as previously: when `jotCount` takes the value J, `allowable[jotCount]` takes the value $S(8J/F)$ with appropriate rounding ("appropriate" to be discussed fully later), S still measuring data in bits. For example, 23 jots is equal to $23/15$ bytes or $8 \cdot 23/15$ bits. The number of corresponding allowable values is $S(8 \cdot 23/15) \approx 4927.59$. We round this value to give a value of 4928 for `allowable[23]`.

Second: we must now consider the probability associated with the given symbol. Thus `decodeBit()` now has a parameter `rung` (whose relation to the probability will later be made explicit). This is used as an index into a table `ladder[]` of structures with three elements: `codeLength0` and `codeLength1`, indicating the number of jots required to code **0** and **1**, respectively; and `threshold`, which (as before) is the lower bound for allowable

```
#define JOTS_PER_BYTE 15

unsigned short allowable[2*JOTS_PER_BYTE] = {1,2,3,4,5,7,
10,14,20,28,41,59,85,123,177,256,371,536,776,1123,1625,
2353,3405,4928,7132,10321,14938,21619,31288,45283};

struct{
    unsigned short *threshold;
    short codeLength0;
    short codeLength1;
    } ladder[3] = {{allowable+14,1,4},
{allowable+13,2,2},{allowable+11,4,1}};

unsigned short internalBuffer;
int jotCount;
FILE *externalIn;

int decodeBit(unsigned char rung){
    if (internalBuffer >= ladder[rung].threshold[jotCount]){
        internalBuffer -= ladder[rung].threshold[jotCount];
        if ((jotCount -= ladder[rung].codeLength1) <= 0)
            decodeImport();
        return 1;
    }else{
        if ((jotCount -= ladder[rung].codeLength0) <= 0)
            decodeImport();
        return 0;
    }
}

void decodeImport(){
    jotCount += JOTS_PER_BYTE;
    internalBuffer <<= 8;
    internalBuffer |= fgetc(externalIn);
}
```

EXAMPLE 2. The ELS-decoder.

values of internalBuffer corresponding to the symbol **1**. (The aptness of our termi-
nology ladder and rung will become apparent in Section 4, when we discuss probability
estimation.)

Third: unlike bitCount in Example 1, jotCount is not decremented by a single pre-
dictable amount for each symbol decoded. Thus we can not depend on jotCount hitting
the value 0 exactly as it is decremented. Therefore we have expanded internalBuffer

from one byte to two; we call the higher-order byte the *working* byte and the lower-order byte the *reserve* byte. We try to keep the reserve byte completely filled with data and the working byte at least partially filled with data—this is equivalent to saying that we maintain at least 256 allowable values for `internalBuffer`. We let `jotCount` indicate the amount of data in the working byte of `internalBuffer`, negative values of `jotCount` indicating that the reserve byte is at least partially depleted. Meaningful values for `jot-Count` range from `-JOTS_PER_BYTE` when the decode is completely empty of data (though in actual operation the minimum value attained by `jotCount` is `1-JOTS_PER_BYTE`), to `JOTS_PER_BYTE` when the decoder is completely full. The decoder calls `decodeImport()` when the value of `jotCount` dips to zero or lower. Moreover, `jotCount` is never decremented by more than `JOTS_PER_BYTE` in a single call to `decodeBit()`, lest we exhaust the reserve byte.

Fourth: the operation of importing a byte is complicated somewhat by the expansion of `internalBuffer` to two bytes. It involves a shift and OR rather than a simple assignment as in Example 1.

Consider as an example a call to `decodeBit()` with a value of 1 for `rung`, the value of `jotCount` being 3. This indicates that `internalBuffer` contains $F + 3 = 18$ meaningful jots. Note that the value of `allowable[18]` is 776; thus the value of `internalBuffer` must be one of $0, 1, \ldots, 775$. If the decoded symbol is **0**, then `jotCount` will be decreased to 2 by subtracting `ladder[1].codeLength0`, and `internalBuffer` will then contain $F + 2 = 17$ meaningful jots. If the decoded symbol is **1**, then `jotCount` will be decreased to -1 by subtracting `ladder[1].codeLength1`, and `internalBuffer` will then contain $F - 1 = 14$ meaningful jots. Note `allowable[17]` is 536 and `allowable[14]` is 177; therefore, out of the 776 allowable values for `internalBuffer`, the 536 values $0, 1, \ldots, 535$ represent **0** and the 177 values $536, 537, \ldots, 712$ represent **1**. Suppose the value of `internalBuffer` is 600. The first step in decoding the symbol is to compare 600 to 536 (the value of `ladder[1].threshold[3]`) and see that the symbol is **1**. We then add -4 to `jotCount`, giving it the value -1. The allowable values for `internalBuffer` are now $0, 1, \ldots 177$; we subtract 536 from `internalBuffer` (making it 64) to bring it within this range.

Since $-1 \leq 0$, we have exhausted the data in the working byte of `internalBuffer`; we call `decodeImport()` to import a byte from the external file. Suppose the next byte in the external file has the value 137. We update the value of `internalBuffer` to $(64 \cdot 256) + 137 = 16521$ (actually accomplished by shifting and ORing) and update the value of `jotCount` to $-1 + 15 = 14$. Note that the number of allowable values for `internalBuffer` is now `allowable[14+15]`, or 45283.

This is an opportune moment to remark on some fundamental principles for the design of a decoder according to our paradigm. For example, note that of the 776 allowable values for `internalBuffer`, 536 represent **0** and 177 represent **1**. The other $776 - 536 - 177 = 63$ values are wasted. This is a defect of the decoder; ideally every allowable value should represent either **0** or **1**, but the restriction of `jotCount` and `allowable` to integer values makes such waste unavoidable at least some of the time.

Our definition of "allowable" states does not consider the future development of a state. Thus some allowable states may not lead to allowable states in the future; such states are unusable in practice. We call the situation where such states exist a *data leak*. In the

presence of a data leak, there are possible states for the decoder which are not allowable. Another characteristic of a data leak is that some possible coded data streams are illegal or redundant.

Data leaks form one of two main sources of coding inefficency in the ELS-coder, the other being inaccuracy in the proportion of values corresponding to **0** and **1**, which will be discussed in Section 4. However, for larger values of `JOTS_PER_BYTE`, the inefficiency is quite modest. As discussed in Section 5, a good working value for `JOTS_PER_BYTE` is 754; in this case the data leak described causes a coding inefficiency of less than .008 bits per symbol.

A data leak exists when some allowable states of the decoder do not lead to allowable states of the decoder at a later time. Since these allowable states embody the data present in the decoder, a data leak implies that data is somehow dissipating into nothing. The situation is analogous to an engine failing to convert all the energy input (as fuel) into usable output energy. In designing an engine, we of course strive to maximize the proportion of input energy converted to output energy. On the other hand, if we designed an engine and found that the output energy was expected to exceed the input energy, we would know that we had made some sort of fundamental error. The analogous principle for the ELS-coder (or for any decoder viewed in the light of our paradigm) is that all allowable states of the decoder *must* be derivable from allowable states of the decoder at any *earlier* time. Our name for a violation of this principle is *perpetual motion*. Perpetual motion implies the existence of allowable states which are not possible. Data leaks and perpetual motion are dual evils; while the former is regrettable, the latter must be avoided at all costs.

For example, the components `codeLength0` and `codeLength1` for each entry of the array `ladder[]` in Example 2 must satisfy the constraint that

```
allowable[jotCount-codeLength0] + allowable[jotCount-codeLength1] <=
                        allowable[jotCount]
```

for all values of `jotCount` between 1 and F, inclusive; i.e., of A allowable values for `internalBuffer` at any time, those representing **0** and those representing **1** can total to no more than A.

Our example decoding process illustrates a second data leak in the decoder, which may be less immediately apparent. This occurs while importing a byte from the external file. Immediately before importing the byte, `internalBuffer` has 177 allowable values. Importing a byte makes the number of values available $256 \cdot 177 = 45312$. However, the number of allowable values after importing is specified as 45283; thus 29 allowable values have been lost. In constructing the table `allowable[]`, we must take care to avoid perpetual motion while importing a byte. Recall that when `jotCount` takes the value J, `allowable[jotCount]` takes the value $S(8J/F)$ with "appropriate" rounding: "appropriate" means that `allowable[jotCount]` takes the value $S(8J/F)$ rounded to the nearest integer when $F \leq J < 2F$, but for $0 \leq J < F$, to avoid perpetual motion, we calculate `allowable[jotCount]` as `(allowable[jotCount+JOTS_PER_BYTE]+255)/256` (i.e. rounded *up*).

The choice of the value of `rung` to be used in a call to `decodeBit()` is dictated by the probabilities that the symbol to be decoded will be **0** or **1**. Suppose that `ladder[rung]` has values L_0 and L_1 for `codeLength0` and `codeLength1`, respectively. Let p denote the

probability that the symbol is **1**. Then the expected number of jots used to code the symbol is $L_0(1-p) + L_1 p$. For example, with a value of 0 for `rung`, the expected number of jots is $(1-p) + 4p = 1 + 3p$; with a value of 1 for `rung`, the expected number of jots is $2(1-p) + 2p = 2$. For purposes of data compression we would of course prefer the smaller of these two values; we can solve the inequality $1 + 3p \leq 2$ for p to find that the value 0 is to be preferred to 1 for `rung` if $p \leq 1/3$. With similar calculations we find that 1 is the preferred value if $1/3 \leq p \leq 2/3$ and 2 is the preferred value if $2/3 \leq p$. (At the boundary points between these intervals we may equally well choose the preferred `rung` value for either side.)

Incidentally, the theoretical optimum compression can be calculated by setting

$$(2.1) \qquad\qquad L_0 = F \log_{256}(1-p), \ L_1 = F \log_{256} p;$$

the corresponding expected number of jots $-F((1-p)\log_{256}(1-p) + p \log_{256} p)$ is known in data compression theory as the *entropy* of the symbol (usually measured in bits rather than jots).

All entries of the table `ladder[]` are calculated so that the values of `codeLength0` and `codeLength1` satisfy the following constraints, termed the *ladder* constraints::

(1) `allowable[J-codeLength0] + allowable[J-codeLength1] <= allowable[J]` for any value of J between 0 and $F - 1$, inclusive (to avoid perpetual motion);
(2) the values of `codeLength0` and `codeLength1` must be positive (so that each symbol **0** or **1** corresponds to at least some allowable values);
(3) the values of `codeLength0` and `codeLength1` must be no greater than `JOTS_PER_BYTE` (to avoid running beyond the end of the reserve byte while decoding a symbol);
(4) subject to the above criteria, `allowable[J] - (allowable[J-codeLength0] + allowable[J-codeLength1])` should be made as small as possible (to minimize data leaks).

Then `ladder[i].threshold` takes the value `allowable - ladder[i].codeLength0 + JOTS_PER_BYTE`. With $F = 15$ there are but three combinations of `codeLength0` and `codeLength1` satisfying all the above criteria; these yield the three entries of the table `ladder` in Example 2.

3. THE ENCODING ALGORITHM

Now for an instant their eyes met in the mirror; and the woman's face he saw there, or seemed to see there, yearned toward him, and was unutterably loving, and compassionate, and yet was resolute in its denial. For it denied him, no matter with what wistful tenderness, or with what wonder at his folly. Just for a moment he seemed to see that; and then he doubted, for Kathleen's lips lifted complaisantly to his, and Kathleen's matter-of-fact face was just as he was used to seeing it.

And thus, with no word uttered, Felix Kennaston understood that his wife must disclaim any knowledge of the sigil of Scoteia, should he be bold enough to speak of it.

James Branch Cabell, *The Cream of the Jest*

Such a moment must be considered a formidable accomplishment from the standpoint of data compression. The reader can most likely cite other occasions when a glance or a handful of words sufficed to carry volumes of meaning. A prerequisite for communication of such high efficiency is a prior high level of empathy between sender and receiver. This is no less true for machines than people; indeed, for machines such a high level of understanding is far more easily attained. One computer program can contain a complete copy of another, if necessary.

The ELS-coder decoding algorithm has already been described. The encoder must use its knowledge of the decoder's inner workings to create a data stream which will manipulate the decoder into producing the desired sequence of decoded symbols; the encoder plays Rasputin to the decoder's Alexandra.

```
unsigned long min;
int jotCount;
int backlog;
FILE *externalOut;

void encodeBit(unsigned char rung, int bit){
    if (bit){
        /* Encode a 1 */
        min += ladder[rung].threshold[jotCount];
        jotCount -= ladder[rung].codeLength1;
    }else{
        /* Encode a 0.  */
        jotCount -= ladder[rung].codeLength0;
    }
    if (jotCount<=0)
        encodeExport();
}
```

EXAMPLE 3. ELS-encoder: basic procedure.

Examples 3 and 4 show procedures for encoding compatible with the decoding procedures of Example 2. The tables `allowable` and `ladder` and the macro `JOTS_PER_BYTE` are identical to those used by the decoder.

In rough terms, the ELS-encoder operates by considering *all* possible coded data streams and gradually eliminating those inconsistent with the current state of the decoder ("current" and other adverbs of time used similarly in this discussion should be understood to refer to position in the data stream rather than actual physical time). For the decoder the "allowable" values of the internal buffer form a convenient reference point for discussion; for the encoder we are concerned with the set of values for the coded data stream *consistent* with the current set of allowable values in the decoder.

As a practical matter, the encoder need not actually consider the entire coded data stream at one time. We can partition the coded data stream at any time into three portions; from end to beginning of the data stream they are: *preactive* bytes, which as yet exert no influence over the current state of the decoder; *active* bytes, which affect the current state of the decoder and have more than one consistent value; and *postactive* bytes, which affect the current state of the decoder and have converged to a single consistent value. Each byte of the coded data stream goes from preactive to active to postactive; the earlier a byte's position in the stream, the earlier these transitions occur. A byte is not actually moved to the external file until it becomes postactive. Only the active portion of the data stream need be considered at any time.

Since the internal buffer of the decoder contains two bytes, there are always at least two active bytes. The variable `backlog` counts the number of active bytes in excess of two. In theory `backlog` can take arbitrarily high values, but higher values become exponentially less likely. When `backlog` takes the value N, there are $N + 2$ active bytes; we number them $0, 1, \ldots, N + 1$ from latest to earliest. The encoder has a variable `jotCount` matching in value the decoder's variable `jotCount` at the same point in the data stream.

Regarding the arithmetic operations used for encoding we can think of the $N + 2$ active bytes as forming a single unsigned number with $N + 2$ bytes of precision, byte 0 being least significant and byte $N + 1$ being most significant. The set of allowable values in the decoder at any time form a continuous range; it can be shown that the consistent values of the active bytes in the encoder at any time likewise form a continuous range. Thus we can describe this range simply by its minimum and maximum values, which we denote mathematically by m and M, respectively. Each of these is a nonnegative integer with N bytes of precision; when we wish to refer specifically to byte k of m or M, we will write $m[k]$ or $M[k]$, respectively. Moreover, since the number of elements of the set is given by `allowable[JOTS_PER_BYTE+jotCount]` only the minimum value need be specified.

Suppose the value N of `backlog` is positive. Consider the most-significant active byte, byte $(N + 1)$. The minimum consistent value for this byte is $m[N + 1]$ and the maximum consistent value is $M[N + 1]$. Since this byte is active and not yet postactive, it has more than one consistent value; thus $m[N + 1] < M[N + 1]$. Let A be the current number of allowable values for the decoder's internal buffer; then $A \leq 2^{16}$ and $M = m + (A - 1)$. Note A can be represented as a two-byte value. If we consider the operation of adding $(A - 1)$ to m byte-by-byte to obtain M, we see that a carry must occur from byte 1 to byte 2 and on upward to byte $(N + 1)$ in order for byte $(N + 1)$ to take differing values for

m and M. We see also that we must have $m[N+1] + 1 = M[N+1]$. And furthermore, if $N > 1$, then $m[2], m[3], \dots, m[N]$ must all take the maximum possible value 255 in order for the carry to propagate from byte 1 to byte $(N+1)$. (And hence $M[2], M[3], \dots, M[N]$ all take the minimum possible value 0.)

This is convenient; for we need not separately store all $N + 1$ bytes of m, but can make the four-byte variable `min` serve. Bytes 0, 1, and 2 of `min` represent $m[0]$, $m[1]$, and (when `backlog` is positive) $m[2]$, respectively. When `backlog` takes a value $N > 1$, then byte 3 of `min` represents $m[N+1]$. Moreover, most of the arithmetic operations to be performed on m can be performed on `min` in a completely ordinary way. The exceptions can be recognized because they require the manipulation of `backlog`.

Having concluded these rather lengthy preliminaries, let us now consider the arithmetic operations entailed in encoding a symbol. Recall the sequence of events in the decoder attending the decoding of the symbol **0**. Initially, `internalBuffer` has one of `allowable[JOTS_PER_BYTE+jotCount]` values. The decoder compares `internalBuffer` to `ladder[rung].threshold[jotCount]`; a **0** is decoded if `internalBuffer` holds the lesser value. Then `ladder[rung].codeLength0` is subtracted from `jotCount`, reflecting the newly decreased number of allowable values for `internalBuffer`. Note that the allowable values eliminated are all taken from the top of the range (those greater than `ladder[rung].threshold[jotCount]`). Thus m does not change; and the encoder need not modify `min` or `backlog` but only `jotCount`.

To encode the symbol **1**: The chief difference in the sequence of events in the decoder as compared to the case for **0** is that those values representing **0**, numbering `ladder[rung].threshold[jotCount]`, are eliminated from the *bottom* of the range of consistent values for the coded data stream. Some additional values may be eliminated from the top of the range as well, if there is a data leak. Thus the encoder, as well as changing the value of `jotCount` to track its value in the decoder, must add `ladder[rung].threshold[jotCount]` to `min` to raise m. The encoder not representing M directly, the operation of eliminating consistent values from the top of the range takes care of itself. The unusual format by which `min` represents m never becomes an issue in `encodeBit()`.

When the value of `jotCount` dips to zero or lower, the encoder calls the procedure `encodeExport()`. In contrast to `decodeImport()`, which invariably reads a single byte from the external file, `encodeExport()` writes from zero to several bytes in a single call. One of its functions is to determine whether the most-significant active bytes have yet converged to a unique consistent value, thus becoming postactive and hence ready to be exported to the encoder's external file. These most-significant bytes may actually converge to a unique consistent value well before `encodeExport()` is called, but there is no harm in waiting until then to check. On each call, `encodeExport()` moves a single byte from the preactive to the active portion of the coded data stream. The format by which `min` represents m does become an issue in `encodeExport()`; we must manipulate `backlog` and `min` together.

```
void encodeExport(){
    unsigned long diffBits;

    /* First check for bytes becoming postactive;
    diffBits marks differences between max and min consistent values */
    jotCount += JOTS_PER_BYTE;
    diffBits = (min + allowable[jotCount] - 1) ∧ min;
    switch(backlog){
    default:  /* 2 or greater */
        if (diffBits & 0xFF000000)
            break;
        fputc(min>>24,externalOut);
        while(--backlog > 1)
            fputc((min>>16) & 0xFF,externalOut);
    case 1:
        if (diffBits & 0x00FF0000)
            break;
        fputc((min>>16) & 0xFF,externalOut);
        backlog--;
    case 0:
        if (diffBits & 0x0000FF00)
            break;
        fputc((min>>8) & 0xFF,externalOut);
        backlog--;
        }
    }
    /* Move a byte from preactive to active.  */
    if (++backlog>2)
        min = (min&0xFF000000)|((min&0x0000FFFF)<< 8);
    else
        min <<= 8;
    }
```

EXAMPLE 4.   Encoding procedure to export data.

4. Probability Estimation

> *As for a future life, every man must judge for himself between conflicting vague*
> *probabilities.*

Charles Darwin

The foregoing sections describe an encoder and decoder for entropy coding; however, powerful operations of data modeling can be incorporated in a natural way into the encoder and decoder using the developer's own sophisticated model for the particular type of data involved.

Earlier we commented that the two main sources of compression inefficiency in the ELS-coder are data leaks and misestimates of probabilities. We now provide some rules of thumb for estimating the compression inefficiency due to the latter in any entropy coder.

Data compression theory teaches that the shortest possible expected code length for a binary symbol taking the value **1** or **0** with probability p or $(1 - p)$, respectively is achieved by using a code of length $\log_{.5} p$ bits for **1** and a code of length $\log_{.5}(1 - p)$ bits for **0**. The resulting optimum expected code length (the entropy of the symbol) is $[(1 - p)\log_{.5}(1 - p) + p\log_{.5} p]$ bits. Suppose we estimate p by $p' = p + \epsilon$ with an error of ϵ; then we would use code lengths for **0** and **1** of $\log_{.5}(1 - p - \epsilon)$ and $\log_{.5}(p + \epsilon)$; the resulting expected code length is $[(1 - p)\log_{.5}(1 - p - \epsilon) + p\log_{.5}(p + \epsilon)]$. Using a power series representation for the logarithm, it can be shown that for small values of ϵ the expected code length is approximately

$$(4.1) \qquad \frac{\epsilon^2}{p(1 - p)\log 2}$$

bits per symbol greater than optimum.

This depends on p; at times it is more convenient to work with the *probability angle* α such that $p = \sin^2 \alpha$ and $1 - p = \cos^2 \alpha$. Then an error of η radians in the probability angle entails a compression inefficiency of approximately

$$(4.2) \qquad \frac{2\eta^2}{\log 2}$$

bits per symbol—independent of α.

In a real-world application, it can be quite difficult to estimate the probabilities for a particular symbol. One approach is to examine the history of symbols which appeared in similar contexts. (This determination of "context" is part of the modeling problem; the example programs COMPRESS and EXPAND provide an illustration.) On the $(t + 1)$st occurrence of a particular context, we create an estimate p_{t+1} for the probability p that the symbol takes the value **1** by averaging its values over previous occurrences:

$$(4.3) \qquad p_{t+1} = \frac{1}{t}(\sigma_1 + \sigma_2 + \cdots + \sigma_t),$$

where σ_u represents the value of the symbol on the uth occurrence of this context.

Such an approach may be improved upon if the value of p shifts over time. For example, in compressing the Bible, one would find that "Moses" occurs more frequently than "Peter" in the Old Testament, while the opposite is true in the New Testament. Therefore we might choose to weight recent occurrences of a given context more heavily. For example, we could use geometrically decreasing weights to create an estimate P_t for p:

$$(4.4) \qquad P_{t+1} = s(\sigma_t + (1 - s)\sigma_{t-1} + (1 - s)^2\sigma_{t-2} + \cdots),$$

s being a parameter between 0 and 1. We have simplified matters by assuming in (4.4) that the context has occurred infinitely many times already; the weights given to occurrences

in the distant past are very small anyway. The greater the value of s, the more heavily more recent occurrences of a context are weighted, and thus the more rapidly P_t reflects changes in p. We therefore call s the *speed* of adaption.

Higher speeds of adaptation better recognize shifting probabilities, but there is an attendant disadvantage. Small speeds of adaptation are akin to calculating an average over a large sample, while large values of speed calculate an average over a small sample. Aside from shifts in the value of p, we would expect our estimate to be more accurate as the sample is larger. Indeed, assuming the symbols σ_u are independent (a reasonable assumption, since dependence indicates an unexploited opportunity for the modeling scheme), and ignoring shifts in the value of p, statistics theory indicates that the mean-squared error in our estimate P_{t+1} for p is given by

$$\epsilon^2 = \frac{s}{2-s} p(1-p),$$

increasing as s increases. In light of formula (4.1), the corresponding coding inefficiency is

$$(4.5) \qquad\qquad\qquad \frac{s}{(2-s)\log 2}$$

bits per symbol (conveniently independent of p).

This dichotomy is of fundamental importance in real-world data compression problems. In estimating probabilities associated with a particular type of data, often one's only recourse is to judge empirically based on the data previously seen. One must decide how much additional importance should be accorded to more recent data. If recent data is weighted heavily, then the accuracy of the estimate suffers because one is estimating from a small sample. If recent data is not weighted heavily, then the accuracy of the estimate may suffer because one's estimate will be slow to reflect shifts in the true probability. The state of the art in modeling many real-world classes of data is such that the probability estimate presented to the entropy coder should often be regarded as rather haphazard.

This gloomy state of affairs is not completely without its consolations. Suppose the probability estimate presented to an entropy coder (of whatever variety) has some significant error. If, due to its own limitations, the entropy coder cannot reproduce the requested probability value exactly, this is as likely to improve compression as to worsen it. Other types of coding inaccuracies may also pale in severity in the face of this fundamental difficulty of probability estimation. Thus the real world can be surprisingly forgiving of minor inefficiencies in an entropy coder, particularly of those related to probability values.

Though more sophisticated than (4.3), formula (4.4) can be more rapidly computed in practice. Note that from (4.4) we can derive:

$$P_{t+1} = s\sigma_t + (1-s)P_t;$$

i.e.,

$$(4.6) \qquad\qquad P_{t+1} = \begin{cases} (1-s)P_t & \text{if } \sigma_t = \mathbf{0}, \\ (1-s)P_t + s & \text{if } \sigma_t = \mathbf{1}. \end{cases}$$

In software, this can be implemented by a pair of lookup tables giving P_{t+1} as a function of P_t, depending on whether the last symbol for the given context was **0** or **1**. We cannot treat P_t as a floating-point value but must restrict it to a discrete set. However, a relatively modest collection of values for P_t can provide excellent coding efficiency. For example, consider 90 values, distributed between 0 and 1 so that the corresponding probability angles are evenly spaced between 0 and 90°. Then the error in probability angle need never exceed .5°. From (4.2), we find the corresponding coding inefficiency to be 0.0002197 bits per symbol. This is the error due to the discretization of p; as noted above, this is likely to be insignificant compared to coding inefficiencies from other effects.

Example 5 illustrates the incorporation of these lookup tables into the ELS-coder. We have added two elements `next0` and `next1` to the structures making up the table `ladder[]`, indicating the next value of the first argument to the procedures `decodeBit()` and `encodeBit()`. if the current symbol is **0** or **1**, respectively. Here can be seen the origin of our nomenclature `ladder[]`; we think of the elements of this table as rungs of a ladder, which we ascend and descend as the probability associated with the context rises and falls. The procedures `decodeBit()` and `encodeBit()` now must modify the value of their first argument, so it is no longer appropriate to pass it by value. The procedures `decodeBit()` and `encodeBit()` now modify the value of the index into `ladder[]`; thus the parameter `rung` is now of type `unsigned char*` rather than `unsigned char` and is a pointer to this index rather than representing the index directly.

The foregoing assumes that probabilities are updated once for every symbol coded. A slight modification of the described technique is to update the probabilities only when the decoder imports data or the encoder exports data (i.e., in the procedures `decodeImport()` and `encodeExport()` of our examples). This should provide faster coding, since the update operation is performed less often, but could decrease coding efficiency, since the probabilities are estimated from a more restricted sample. This approach is appealing from the standpoint that probabilities are modified more often when compression is poor, since in that case bytes are imported more frequently. Our name for this variant is *inalacritous* updating, the other method being of course *alacritous*. The sample listing accompanying this article embodies both approaches, distinguished by compile switches.

Inalacritous updating requires a different set of look-up tables for the probability ladder, since bytes are more likely to be imported during the coding of certain symbols than others. The updating formula (4.6) indicates that the probability estimate increases by $\Delta p_1 = s(1 - P_t)$ when the coded symbol is **1** and decreases by $\Delta p_0 = -sP_t$ when the coded symbol is **0**—note $\Delta p_1 - \Delta p_0 = s$. Thus the ratio of these two step sizes is $\Delta p_1 / \Delta p_0 = -(1 - P_t)/P_t$. If a given element of the table `ladder[]` has values L_0 for `codeLength0` and L_1 for `codeLength1`, then the probability of importing a byte when **0** is coded is L_0/F and the probability of importing a byte when **1** is coded is L_1/F. To compensate, we therefore adjust the ratio of the two step sizes to

$$\frac{\Delta p_1'}{\Delta p_0'} = \left(\frac{\Delta p_1}{\Delta p_0}\right)\left(\frac{L_0/F}{L_1/F}\right) = -\frac{(1 - P_t)L_0}{P_t L_1}.$$

If we define the speed s to be $\Delta p_1' - \Delta p_0'$ as previously, then we obtain

$$\Delta p_0' = \frac{-sP_t L_1}{(1 - P_t)L_0 + P_t L_1},$$

```
struct{
    unsigned short *threshold;
    short codeLength0;
    short codeLength1;
    unsigned char next0;
    unsigned char next1;
    } ladder[...] = {...};
...
int decodeBit(unsigned char *rung){
    if (internalBuffer >= ladder[*rung].threshold[jotCount]){
        internalBuffer -= ladder[*rung].threshold[jotCount];
        if ((jotCount -= ladder[*rung].codeLength1) <= 0)
            decodeImport();
        *rung = ladder[*rung].next1;
        return 1;
    }else{
        if ((jotCount -= ladder[*rung].codeLength0) <= 0)
            decodeImport();
        *rung = ladder[*rung].next0;
        return 0;
    }
}

void encodeBit(unsigned char *rung, int bit){
    if (bit){
        min += ladder[*rung].threshold[jotCount];
        jotCount -= ladder[*rung].codeLength1;
        *rung = ladder[*rung].next1;
    }else{
        jotCount -= ladder[*rung].codeLength0;
        *rung = ladder[*rung].next0;
    }
    if (jotCount<=0)
        encodeExport();
}
```

EXAMPLE 5. Entropy decoding and encoding procedures incorporating probability estimation.

$$\Delta p_1' = \frac{s(1 - P_t)L_0}{(1 - P_t)L_0 + P_t L_1}.$$

For P_t near 0 or 1, we may need to restrict s to small values lest $P_t + \Delta p_0' < 0$ or $P_t + \Delta p_1' > 1$. The accompanying sample listings include complete examples of both types

of probability ladder.

5. The Code Samples

Listings for four files **elscoder.h**, **elscoder.c**, **compress.c** and **expand.c** accompany this article. The files **elscoder.h** and **elscoder.c** contain declarations and definitions for an implementation of the ELS-coder, respectively. The files **compress.c** and **expand.c** are sample programs which use the ELS-coder to compress and expand a file using a one-byte model.

The code fragments given as examples in the previous sections were designed more for pedagogical value than utility. The sample programs represent a compromise between these two goals with more emphasis on utility. The reader will note the following changes:

(1) The variables describing the states of the decoder and encoder have been collected into structures named d and e, respectively.

(2) Globally visible names in the coder have been prefixed by els, Els, or ELS, to help prevent namespace collisions.

(3) The procedures decodeImport() and encodeExport() have been eliminated and the corresponding operations incorporated directly into elsDecodeBit() and elsEncodeBit().

(4) The value of JOTS_PER_BYTE has been increased from 15 to 754. The value 754 represents a sort of local optimum. Although higher values for JOTS_PER_BYTE generally yield better coding efficiency, the vagaries of the ladder constraints listed in Section 2 dictate that higher compression ratios are attainable for the value 754 than for 755 and subsequent higher values up to around 1508.

The file **elscoder.c** includes straightforward procedures elsDecodeStart() and elsEncodeStart() for initializing and elsDecodeEnd() and elsEncodeEnd() for terminating encoding and decoding.

One can choose alacritous or inalacritous probability updating in the coder by defining or not defining ALACRITOUS in compiling **elscoder.c**

The sample files also incorporate some enhancements to the basic ELS-coder described in the previous sections:

First: Although the entropy-coding algorithm works with units of data smaller than a bit, most likely the source and destination files must have lengths that are multiples of a byte. Thus a file coded with the ELS-coder is likely to end with a fraction of a byte not containing meaningful data. The sample ELS-coder attempts to make a virtue of necessity by using this empty portion to encode something akin to a checksum. When encoding is concluded, a certain number of consistent values remain in the encoder; these are used to store the value of jotCount to whatever precision is possible. This is done automatically in the procedure elsEncodeEnd(). If desired, the user can call the function elsDecodeOk() when finished decoding but before calling elsDecodeEnd(); this will verify that the value of jotCount matches that sent by the encoder. The ELS-coder is such that any corruption of the compressed data will most likely profoundly alter the course of execution of the decoder; the probability of ending with the same value of jotCount is quite small. Of course, this value is only encoded to the precision possible with the fraction of a byte

remaining at the end of coding; thus the probability of hitting the correct value by chance ranges from 1 in 255 to 1, depending on what that fraction is.

Second: Recall that our design of the probability state ladder (in both the alacritous and non-alacritous cases) assumes that any given context has already occurred infinitely many times. This assumption is least tenable near the beginning of the coding process. It seems appropriate to use smaller samples to estimate probabilities (i.e., to use greater speeds of adaptation) early in the coding process and larger samples (lesser speeds of adaptation) later in the process. This has been achieved by incorporating *transient* states into the probability ladder; these are states which can be visited at most once early in the coding process and never again thereafter.

Consider for example the alacritous probability ladder. The initial rung is at index 0 and has probability .5 (for convenience, the index and probability value corresponding to each rung of the probability ladder are listed in a comment preceding the initializer for that rung). State 0 is followed by rung 1 or 2, with probabilities of .33333 or .66667, respectively. The speed of rung 0 is equal to the difference between these values, or .33333. Rungs 1 and 2 are followed by rungs 3,4,5,6; the speed for rungs 1 and 2 is given by, for example, the difference between the probability values for rungs 3 and 4, or .29. Similarly, rungs 3,4,5,6, are followed by rungs 7,... , 14, which are in turn followed by rungs 15,... , 30, which are in turn followed by rungs 31,... , 62, which are in turn followed by rungs 63,... , 126. The speed decreases at each step, from .33333 to .29 to .24667 to .20333 to .16 to .11667 to .07333. From one of rungs 63,... , 126 the coder transits to one of rungs 127,... ,254; these form the permanent part of the ladder; these rungs can be visited repeatedly. The speed for the permanent rungs is .03.

The inalacritous probability ladder likewise consists of 127 transient rungs and 128 permanent rungs.

The values of speed used here are by no means sacred (although some theoretical argument can be made for an initial value of $1/3$); the best value of speed for the permanent rungs depends on the nature of the data being compressed. Nor is it required to have 127 transient rungs and 128 permanent rungs; we simply chose values totaling to almost 256 in order to get the most mileage out of the `unsigned char` serving as an index into the ladder.

Third: Rung 255 (of either ladder) does not represent a probability state, but is available for use as an "inactive" state. The only really important components of this rung are `codeLength0` and `codeLength1`, both set to `2*JOTS_PER_BYTE+1`. This value is guaranteed to trigger an import operation; moreover the value of `jotCount` remains negative even after the addition of `JOTS_PER_BYTE`. Both encoder and decoder are equipped to recognize this as an error condition.

The example shell programs **compress.c** and **expand.c** illustrate the use of various contexts for modeling. In this case the model is a one-byte model. For example, if compressing English text, the model exploits the fact that 'e' occurs more frequently than 'u', but not the fact that 'u' occurs more frequently than 'e' following 'q'; it represents frequencies for each byte but not relationships between different bytes. This model is described by a state machine, each state holding a probability rung. Each state of the machine corresponds to a particular context. For example, the first bit of a byte is one

of these contexts. The last bit of a byte corresponds to 128 different contexts, depending on the values of the preceding seven bits. Any model to be used with the ELS-coder or similar coders handling only binary symbols must be expressed as such a state machine.

This one-byte model requires 255 states (stored in the table `context[]`), organized in a tree fashion similar to the transient rungs of the probability ladder—we number these from 1 to 255. State 1 is used for the first bit of any byte. State 2 is used for the second bit of a byte if the first bit was **0**; state 3 is used for the second bit of a byte if the first bit was **1**. In general, state $2n$ is used for the next bit of a byte if the preceding bit corresponding to state n was **0**, while state $2n + 1$ is used for the next bit of a byte if the preceding bit corresponding to state n was **1**. This arrangement of states also makes for straightforward calculation of state transitions, so storing the values of successor states in unnecessary. Such a tree of probabilities is equivalent to storing the probabilities for all 256 byte values.

The entry `context[0]` has not been used in any of the above; we use it to identify a 257th symbol serving as an end-of-file marker. We represent the byte value 255 by the sequence of *nine* bits **111111110**; the end-of-file marker is the nine-bit sequence **111111111**. The state `context[0]` is used for the final bit of this sequence. Both **compress.c** and **expand.c** must take special action in the case of a byte with value 255.

One can compress a file *raw* to one *compressed* by the command

compress *raw compressed.*

Subsequently one can expand the file *compressed* to one *expanded* by the command

expand *compressed expanded.*

The files *raw* and *expanded* should then be identical.

6. Comparison with the Q-coder

We now point out similarities and differences between the ELS-coder and the Q-coder. These may prove useful for those wishing to make a later study of the Q-coder.

The Q-coder is easily interpreted in terms of our paradigm for a decoder as given in Section 2. (Actually we would claim that *any* decoder can be described by this paradigm, but the description is more transparent for some than for others.) Like the ELS-coder, the Q-coder uses two components to describe the state of the decoder at any time: one, the "augend", denoted A, describes the amount of data at any time (as does `jotCount`); the other, denoted X, gives the content of that data (as does `internalBuffer`). While `jotCount` in the ELS-coder directly measures the quantity of data—the corresponding number of allowable states being given by an exponential relation as embodied in the table `allowable[]`—, the augend A in the Q-coder directly measures the number of allowable states in the decoder—the corresponding quantity of data being proportional to the logarithm of A. Most points of difference between the two coders are consequences of this fundamental difference.

Ideally, while decoding a symbol, the proportion of allowable states corresponding to the symbol **0** should match the probability p_0 that the symbol is **0**. Determining the number

of allowable states corresponding to each symbol therefore requires a multiplication or division to be accomplished in some way. In the ELS-coder, the number of allowable states allocated to the symbol **0** is given by an entry of a lookup table `threshold[]` depending on p_0. Since the ELS-coder represents the number of states indirectly, via an exponential table, a simple addition performed on the index into the table is equivalent to a multiplication performed on the number of allowable states. It is interesting to note that in spite of the fundamental role played by probabilities in entropy coding, the ELS-coder operates without representing any probabilities directly. By contrast the Q-coder represents each probability state by a single number (called a *Qe-value*) proportional to the probability. The Q-coder approximates the desired multiplication operation either by subtracting the Qe-value directly from the augend A or by assigning the Qe-value to A.

This approximation provides (perhaps surprising) coding efficiency, but it does impose some constraints on the design of the Q-coder:

(1) Maintaining sufficient accuracy of approximation requires that the number of al-lowable states in the decoder be restricted to a much smaller range than that used in the ELS-coder; in general the number of allowable states is not permitted to vary by more than a factor of two (whereas in the ELS-coder as described here, this number varies by a factor of 256). As a consequence, data must be imported much more frequently in the Q-coder than in the ELS-coder. Moreover, this data import operation is also more complex in the Q-coder: one or several bits may be import at one time, whereas in the ELS-coder, the quantity of data import is always exactly one byte.

(2) The approximation is most accurate for small values of the probability, and is acceptable only for values less than one-half. Thus the Q-coder does not work with symbols **0** and **1** but with the so-called "MPS" (more probable symbol) and "LPS" (less probable symbol). In encoding or decoding a symbol, the Q-coder must determine which of **0** or **1** is the MPS. Such an extra level of indirection might be useful in an implementation of the ELS-coder, but can be left to the discretion of the implementer.

Unlike the ELS-coder, the Q-coder has no data leaks. On the other hand, under the best of conditions, the approximation used in the Q-coder is less accurate than that used in the ELS-coder even with data leaks taken into consideration. However, in the light of our remarks in Section 4 on the practical difficulties of accurate probability estimation and the effect of such on coding efficiency, not too much should be made of this distinction.

The Q-coder uses a probability ladder as does the ELS-coder, but derived on different principles. This ladder relies strictly on inalacritous probability updating; an alacritous version would certainly be possible but does not seem to have been the subject of much study.

The question of which of the Q-coder and ELS-coder codes more efficiently and which operates more rapidly depends on many issues of architecture and nature of the data.

The Q-coder performs best in the realm of profound compression: that is, when the probability of the LPS is very close to 0 and each symbol can be coded very compactly. The approximation used in the Q-coder provides its best coding efficiency in such a case, and a high compression ratio entails less frequent importing of data, so that operation is

faster. Indeed the highest compression level attainable by the Q-coder ($2^{15} : 1$) exceeds that of the ELS-coder ($754 : 1$ if `JOTS_PER_BYTE` takes the value 754). In relative terms the difference is large (a ratio of 43.5) but in absolute terms it is small (a difference of 0.00130 bits per symbol). Our design philosophy is that the absolute measure of coding efficiency is more important. A version of the ELS-coder using sixteen-bit words rather than eight-bit bytes as its basic unit of data would provide profound compression comparable to that of the Q-coder with about twice the memory usage of the version given here.

In the realm of mild compression, the superior accuracy and less frequent data import of the ELS-coder probably give it an edge. It is worth noting that data compression applications can often be designed so that profound compression is unnecessary; such a design is likely to improve both speed and compression efficiency. For example, in a simple binary model, if **0** is overwhelmingly more probable than **1**, one is likely to find long runs of the symbol **0** in the data. Run-length-encoding the symbols **0** would reduce the number of calls to the entropy coder while reducing the dependence of compression ratio on achievability of profound compression.

Another example of such a design decision is given in the sample programs **compress.c** and **expand.c** with respect to the coding of the end-of-file marker. The bit indicating end-of-file could have been inserted at the base of the state tree. However, this would require every byte of the compressed file to carry an extra **0** bit indicating that it was not end-of-file; profound compression would be needed to compress these bits collectively to a tiny portion of the file. By attaching this bit rather to a node of the tree, we insured that such bits would usually appear much less often; thus these bits usually represent a small portion of the compressed file even without profound compression.

We have compared the QM-coder and the ELS-coder empirically in the context of an image-compression application. True to expectations, the alacritous version of the ELS-coder performed better than the QM-coder on not-easily-compressible images, while the QM-coder performed better on easily-compressible images. The non-alacritous version of the ELS-coder entailed an overall loss in compression efficiency of about 15% (though the relative performance could be manipulated by choosing the images to be compressed); however the the non-alacritous ELS-coder might be the method of choice where speed is an overriding concern.

Acknowledgments

I would like to thank Stephen Mann for reading an early draft of this article and proposing many improvements in both form and content.

References

1. T. M. Cover and J. A. Thomas, *Elements of Information Theory*, John Wiley and Sons, 1991.
2. M. Nelson, *The Data Compression Book*, M&T Books, Redwood City, CA, 1991.
3. W. B. Pennebaker and J. L. Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993.
4. W. B. Pennebaker, J.L. Mitchell, G.G. Langdon, and R.B. Arps, *An overview of the basic principles of the Q-coder adaptive binary arithmetic coder*, IBM Journal of Research and Development **32** (1988), 771–726.

5. Ian H. Witten, Radford M. Neal, and John H. Cleary, *Arithmetic coding for data compression*, Communications of the Association for Computing Machinery **30, No. 6** (1987), 520–540.

6. , *Arithmetic coding encoder and decoder system*, United States Patent Number 4,905,297 (1990).

7. , *Probability Adaptation for Arithmetic Coders*, United States Patent Number 4,935,882 (1990).

LISTING ONE: elscoder.h

```
/*  "elscoder.h":   include file for ELS-coder. */

#ifndef ELSCODER_H
#define ELSCODER_H
#include <stdio.h>

typedef unsigned char ElsRung;

/* Initialize decoding. */
void elsDecodeStart(FILE *file);
/* Decode a single bit. */
int elsDecodeBit(ElsRung *rung);
/* Perform consistency check prior to ending decoding. */
int elsDecodeOk(void);
/* Conclude decoding. */
void elsDecodeEnd(void);
/* Initialize encoding. */
void elsEncodeStart(FILE *file);
/* Encode a single bit. */
void elsEncodeBit(ElsRung *rung, int bit);
/* Conclude encoding. */
void elsEncodeEnd(void);
#endif /* ndef ELSCODER_H */
```

LISTING TWO: elscoder.c

```c
/* "elscoder.c":  data and procedures for ELS-coder. */

#define ALACRITOUS
#include "elscoder.h"
#include <stdio.h>
#include <stdlib.h>

typedef unsigned char uchar;
typedef unsigned short ushort;
typedef unsigned long ulong;

#define JOTS_PER_BYTE 754

static ushort allowable[2*JOTS_PER_BYTE] = {
      1,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
      2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
      2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
      2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
      2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
      2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
      2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
      2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     2,     3,
      3,     3,     3,     3,     3,     3,     3,     3,     3,     3,     3,     3,
      3,     3,     3,     3,     3,     3,     3,     3,     3,     3,     3,     3,
      3,     3,     3,     3,     3,     3,     3,     3,     3,     3,     3,     3,
      3,     3,     3,     3,     3,     3,     3,     3,     3,     3,     3,     3,
      3,     3,     3,     3,     3,     3,     4,     4,     4,     4,     4,     4,
      4,     4,     4,     4,     4,     4,     4,     4,     4,     4,     4,     4,
      4,     4,     4,     4,     4,     4,     4,     4,     4,     4,     4,     4,
      4,     4,     4,     4,     4,     4,     4,     4,     4,     5,     5,     5,
      5,     5,     5,     5,     5,     5,     5,     5,     5,     5,     5,     5,
      5,     5,     5,     5,     5,     5,     5,     5,     5,     5,     5,     5,
      5,     5,     5,     6,     6,     6,     6,     6,     6,     6,     6,     6,
      6,     6,     6,     6,     6,     6,     6,     6,     6,     6,     6,     6,
      6,     6,     6,     6,     7,     7,     7,     7,     7,     7,     7,     7,
      7,     7,     7,     7,     7,     7,     7,     7,     7,     7,     7,     7,
      7,     8,     8,     8,     8,     8,     8,     8,     8,     8,     8,     8,
      8,     8,     8,     8,     8,     8,     8,     9,     9,     9,     9,     9,
      9,     9,     9,     9,     9,     9,     9,     9,     9,     9,     9,    10,
     10,    10,    10,    10,    10,    10,    10,    10,    10,    10,    10,    10,
     10,    10,    11,    11,    11,    11,    11,    11,    11,    11,    11,    11,
     11,    11,    11,    12,    12,    12,    12,    12,    12,    12,    12,    12,
```

```
 12,   12,   13,   13,   13,   13,   13,   13,   13,   13,   13,   13,
 13,   14,   14,   14,   14,   14,   14,   14,   14,   14,   14,   15,
 15,   15,   15,   15,   15,   15,   15,   15,   15,   16,   16,   16,
 16,   16,   16,   16,   16,   16,   17,   17,   17,   17,   17,   17,
 17,   17,   18,   18,   18,   18,   18,   18,   18,   18,   19,   19,
 19,   19,   19,   19,   19,   20,   20,   20,   20,   20,   20,   20,
 21,   21,   21,   21,   21,   21,   22,   22,   22,   22,   22,   22,
 22,   23,   23,   23,   23,   23,   23,   24,   24,   24,   24,   24,
 24,   25,   25,   25,   25,   25,   26,   26,   26,   26,   26,   26,
 27,   27,   27,   27,   27,   28,   28,   28,   28,   28,   29,   29,
 29,   29,   30,   30,   30,   30,   30,   31,   31,   31,   31,   32,
 32,   32,   32,   32,   33,   33,   33,   33,   34,   34,   34,   34,
 35,   35,   35,   35,   36,   36,   36,   36,   37,   37,   37,   38,
 38,   38,   38,   39,   39,   39,   39,   40,   40,   40,   41,   41,
 41,   42,   42,   42,   42,   43,   43,   43,   44,   44,   44,   45,
 45,   45,   46,   46,   46,   47,   47,   47,   48,   48,   48,   49,
 49,   49,   50,   50,   51,   51,   51,   52,   52,   52,   53,   53,
 54,   54,   54,   55,   55,   56,   56,   56,   57,   57,   58,   58,
 58,   59,   59,   60,   60,   61,   61,   62,   62,   62,   63,   63,
 64,   64,   65,   65,   66,   66,   67,   67,   68,   68,   69,   69,
 70,   70,   71,   71,   72,   72,   73,   73,   74,   74,   75,   75,
 76,   77,   77,   78,   78,   79,   79,   80,   81,   81,   82,   82,
 83,   84,   84,   85,   85,   86,   87,   87,   88,   89,   89,   90,
 91,   91,   92,   93,   93,   94,   95,   95,   96,   97,   97,   98,
 99,  100,  100,  101,  102,  103,  103,  104,  105,  106,  106,  107,
108,  109,  110,  110,  111,  112,  113,  114,  114,  115,  116,  117,
118,  119,  120,  121,  121,  122,  123,  124,  125,  126,  127,  128,
129,  130,  131,  132,  133,  134,  135,  136,  137,  138,  139,  140,
141,  142,  143,  144,  145,  146,  147,  148,  149,  150,  151,  152,
153,  155,  156,  157,  158,  159,  160,  162,  163,  164,  165,  166,
168,  169,  170,  171,  173,  174,  175,  176,  178,  179,  180,  182,
183,  184,  186,  187,  188,  190,  191,  193,  194,  196,  197,  198,
200,  201,  203,  204,  206,  207,  209,  210,  212,  214,  215,  217,
218,  220,  221,  223,  225,  226,  228,  230,  231,  233,  235,  237,
238,  240,  242,  244,  245,  247,  249,  251,  253,  255,  256,  258,
260,  262,  264,  266,  268,  270,  272,  274,  276,  278,  280,  282,
284,  286,  288,  290,  292,  294,  297,  299,  301,  303,  305,  308,
310,  312,  315,  317,  319,  322,  324,  326,  329,  331,  334,  336,
339,  341,  344,  346,  349,  351,  354,  356,  359,  362,  364,  367,
370,  373,  375,  378,  381,  384,  386,  389,  392,  395,  398,  401,
404,  407,  410,  413,  416,  419,  422,  425,  428,  432,  435,  438,
441,  444,  448,  451,  454,  458,  461,  464,  468,  471,  475,  478,
482,  485,  489,  493,  496,  500,  504,  507,  511,  515,  519,  522,
526,  530,  534,  538,  542,  546,  550,  554,  558,  562,  566,  571,
```

```
 575,  579,  583,  588,  592,  596,  601,  605,  610,  614,  619,  623,
 628,  633,  637,  642,  647,  651,  656,  661,  666,  671,  676,  681,
 686,  691,  696,  701,  706,  712,  717,  722,  727,  733,  738,  744,
 749,  755,  760,  766,  771,  777,  783,  789,  795,  800,  806,  812,
 818,  824,  830,  837,  843,  849,  855,  861,  868,  874,  881,  887,
 894,  900,  907,  914,  920,  927,  934,  941,  948,  955,  962,  969,
 976,  983,  991,  998, 1005, 1013, 1020, 1028, 1035, 1043, 1051, 1058,
1066, 1074, 1082, 1090, 1098, 1106, 1114, 1123, 1131, 1139, 1148, 1156,
1165, 1173, 1182, 1191, 1199, 1208, 1217, 1226, 1235, 1244, 1254, 1263,
1272, 1281, 1291, 1300, 1310, 1320, 1329, 1339, 1349, 1359, 1369, 1379,
1389, 1400, 1410, 1420, 1431, 1442, 1452, 1463, 1474, 1485, 1496, 1507,
1518, 1529, 1540, 1552, 1563, 1575, 1586, 1598, 1610, 1622, 1633, 1646,
1658, 1670, 1682, 1695, 1707, 1720, 1732, 1745, 1758, 1771, 1784, 1797,
1811, 1824, 1837, 1851, 1865, 1878, 1892, 1906, 1920, 1935, 1949, 1963,
1978, 1992, 2007, 2022, 2037, 2052, 2067, 2082, 2098, 2113, 2129, 2144,
2160, 2176, 2192, 2208, 2225, 2241, 2258, 2274, 2291, 2308, 2325, 2342,
2359, 2377, 2394, 2412, 2430, 2448, 2466, 2484, 2502, 2521, 2540, 2558,
2577, 2596, 2615, 2635, 2654, 2674, 2693, 2713, 2733, 2754, 2774, 2794,
2815, 2836, 2857, 2878, 2899, 2920, 2942, 2964, 2986, 3008, 3030, 3052,
3075, 3097, 3120, 3143, 3166, 3190, 3213, 3237, 3261, 3285, 3309, 3334,
3358, 3383, 3408, 3433, 3459, 3484, 3510, 3536, 3562, 3588, 3615, 3641,
3668, 3695, 3723, 3750, 3778, 3806, 3834, 3862, 3890, 3919, 3948, 3977,
4007, 4036, 4066, 4096, 4126, 4157, 4187, 4218, 4249, 4281, 4312, 4344,
4376, 4409, 4441, 4474, 4507, 4540, 4574, 4607, 4641, 4676, 4710, 4745,
4780, 4815, 4851, 4887, 4923, 4959, 4996, 5033, 5070, 5107, 5145, 5183,
5221, 5260, 5298, 5338, 5377, 5417, 5457, 5497, 5537, 5578, 5620, 5661,
5703, 5745, 5787, 5830, 5873, 5916, 5960, 6004, 6048, 6093, 6138, 6183,
6229, 6275, 6321, 6368, 6415, 6462, 6510, 6558, 6606, 6655, 6704, 6754,
6804, 6854, 6904, 6955, 7007, 7058, 7111, 7163, 7216, 7269, 7323, 7377,
7431, 7486, 7542, 7597, 7653, 7710, 7767, 7824, 7882, 7940, 7999, 8058,
8117, 8177, 8237, 8298, 8359, 8421, 8483, 8546, 8609, 8672, 8736, 8801,
8866, 8931, 8997, 9064, 9131, 9198, 9266, 9334, 9403, 9473, 9543, 9613,
9684, 9755, 9827, 9900, 9973,10047,10121,10196,10271,10347,10423,10500,
10577,10655,10734,10813,10893,10974,11055,11136,11218,11301,11385,11469,
11553,11639,11725,11811,11898,11986,12075,12164,12253,12344,12435,12527,
12619,12712,12806,12901,12996,13092,13189,13286,13384,13483,13582,13683,
13784,13885,13988,14091,14195,14300,14405,14512,14619,14727,14835,14945,
15055,15166,15278,15391,15505,15619,15735,15851,15968,16086,16204,16324,
16444,16566,16688,16811,16935,17060,17186,17313,17441,17570,17699,17830,
17962,18094,18228,18362,18498,18634,18772,18910,19050,19191,19332,19475,
19619,19764,19909,20056,20204,20354,20504,20655,20808,20961,21116,21272,
21429,21587,21746,21907,22069,22232,22396,22561,22727,22895,23064,23234,
23406,23579,23753,23928,24105,24283,24462,24642,24824,25008,25192,25378,
25565,25754,25944,26136,26329,26523,26719,26916,27115,27315,27517,27720,
```

```
27924,28130,28338,28547,28758,28970,29184,29399,29616,29835,30055,30277,
30501,30726,30953,31181,31411,31643,31877,32112,32349,32588,32828,33071,
33315,33561,33808,34058,34309,34563,34818,35075,35334,35594,35857,36122,
36388,36657,36928,37200,37475,37751,38030,38311,38594,38878,39165,39455,
39746,40039,40335,40632,40932,41235,41539,41846,42154,42466,42779,43095,
43413,43733,44056,44381,44709,45039,45371,45706,46044,46384,46726,47071,
47418,47768,48121,48476,48834,49194,49558,49923,50292,50663,51037,51414,
51793,52176,52561,52949,53340,53733,54130,54529,54932,55337,55746,56157,
56572,56989,57410,57834,58261,58691,59124,59561,60000,60443,60889,61339,
61791,62248,62707,63170,63636,64106,64579,65056 };

static struct
    {
    ushort *threshold;
    short codeLength0;
    short codeLength1;
    uchar next0;
    uchar next1;
    } ladder[256] = {
#ifdef ALACRITOUS
 /*    0:0.50000 */ {allowable+ 665,  89, 101,    1,    2},
 /*    1:0.33333 */ {allowable+ 692,  62, 138,    3,    4},
 /*    2:0.66667 */ {allowable+ 616, 138,  62,    5,    6},
 /*    3:0.23667 */ {allowable+ 716,  38, 194,    7,    8},
 /*    4:0.52667 */ {allowable+ 653, 101,  89,    9,   10},
 /*    5:0.47333 */ {allowable+ 665,  89, 101,   11,   12},
 /*    6:0.76333 */ {allowable+ 560, 194,  38,   13,   14},
 /*    7:0.17829 */ {allowable+ 727,  27, 236,   15,   16},
 /*    8:0.42496 */ {allowable+ 679,  75, 118,   17,   18},
 /*    9:0.39676 */ {allowable+ 689,  65, 133,   19,   20},
 /*   10:0.64342 */ {allowable+ 616, 138,  62,   21,   22},
 /*   11:0.35658 */ {allowable+ 692,  62, 138,   23,   24},
 /*   12:0.60324 */ {allowable+ 621, 133,  65,   25,   26},
 /*   13:0.57504 */ {allowable+ 636, 118,  75,   27,   28},
 /*   14:0.82171 */ {allowable+ 518, 236,  27,   29,   30},
 /*   15:0.14204 */ {allowable+ 732,  22, 262,   31,   32},
 /*   16:0.34537 */ {allowable+ 692,  62, 138,   33,   34},
 /*   17:0.33855 */ {allowable+ 692,  62, 138,   35,   36},
 /*   18:0.54188 */ {allowable+ 646, 108,  83,   37,   38},
 /*   19:0.31608 */ {allowable+ 704,  50, 162,   39,   40},
 /*   20:0.51942 */ {allowable+ 653, 101,  89,   41,   42},
 /*   21:0.51259 */ {allowable+ 653, 101,  89,   43,   44},
 /*   22:0.71593 */ {allowable+ 580, 174,  45,   45,   46},
 /*   23:0.28407 */ {allowable+ 709,  45, 174,   47,   48},
```

```
/*   24:0.48741 */ {allowable+ 665,  89, 101,  49,  50},
/*   25:0.48058 */ {allowable+ 665,  89, 101,  51,  52},
/*   26:0.68392 */ {allowable+ 592, 162,  50,  53,  54},
/*   27:0.45812 */ {allowable+ 671,  83, 108,  55,  56},
/*   28:0.66145 */ {allowable+ 616, 138,  62,  57,  58},
/*   29:0.65463 */ {allowable+ 616, 138,  62,  59,  60},
/*   30:0.85796 */ {allowable+ 492, 262,  22,  61,  62},
/*   31:0.11931 */ {allowable+ 737,  17, 295,  63,  64},
/*   32:0.27931 */ {allowable+ 709,  45, 174,  65,  66},
/*   33:0.29011 */ {allowable+ 709,  45, 174,  67,  68},
/*   34:0.45011 */ {allowable+ 671,  83, 108,  69,  70},
/*   35:0.28438 */ {allowable+ 709,  45, 174,  71,  72},
/*   36:0.44438 */ {allowable+ 679,  75, 118,  73,  74},
/*   37:0.45518 */ {allowable+ 671,  83, 108,  75,  76},
/*   38:0.61518 */ {allowable+ 621, 133,  65,  77,  78},
/*   39:0.26551 */ {allowable+ 713,  41, 185,  79,  80},
/*   40:0.42551 */ {allowable+ 679,  75, 118,  81,  82},
/*   41:0.43631 */ {allowable+ 679,  75, 118,  83,  84},
/*   42:0.59631 */ {allowable+ 636, 118,  75,  85,  86},
/*   43:0.43058 */ {allowable+ 679,  75, 118,  87,  88},
/*   44:0.59058 */ {allowable+ 636, 118,  75,  89,  90},
/*   45:0.60138 */ {allowable+ 621, 133,  65,  91,  92},
/*   46:0.76138 */ {allowable+ 560, 194,  38,  93,  94},
/*   47:0.23862 */ {allowable+ 716,  38, 194,  95,  96},
/*   48:0.39862 */ {allowable+ 689,  65, 133,  97,  98},
/*   49:0.40942 */ {allowable+ 679,  75, 118,  99, 100},
/*   50:0.56942 */ {allowable+ 636, 118,  75, 101, 102},
/*   51:0.40369 */ {allowable+ 679,  75, 118, 103, 104},
/*   52:0.56369 */ {allowable+ 636, 118,  75, 105, 106},
/*   53:0.57449 */ {allowable+ 636, 118,  75, 107, 108},
/*   54:0.73449 */ {allowable+ 569, 185,  41, 109, 110},
/*   55:0.38482 */ {allowable+ 689,  65, 133, 111, 112},
/*   56:0.54482 */ {allowable+ 646, 108,  83, 113, 114},
/*   57:0.55562 */ {allowable+ 636, 118,  75, 115, 116},
/*   58:0.71562 */ {allowable+ 580, 174,  45, 117, 118},
/*   59:0.54989 */ {allowable+ 646, 108,  83, 119, 120},
/*   60:0.70989 */ {allowable+ 580, 174,  45, 121, 122},
/*   61:0.72069 */ {allowable+ 580, 174,  45, 123, 124},
/*   62:0.88069 */ {allowable+ 459, 295,  17, 125, 126},
/*   63:0.10539 */ {allowable+ 738,  16, 303, 152, 161},
/*   64:0.22206 */ {allowable+ 721,  33, 211, 165, 172},
/*   65:0.24672 */ {allowable+ 716,  38, 194, 167, 174},
/*   66:0.36339 */ {allowable+ 692,  62, 138, 177, 183},
/*   67:0.25626 */ {allowable+ 713,  41, 185, 168, 175},
```

```
/*  68:0.37293 */ {allowable+ 692,  62, 138, 178, 184},
/*  69:0.39760 */ {allowable+ 689,  65, 133, 180, 186},
/*  70:0.51426 */ {allowable+ 653, 101,  89, 189, 195},
/*  71:0.25120 */ {allowable+ 713,  41, 185, 168, 174},
/*  72:0.36787 */ {allowable+ 692,  62, 138, 177, 183},
/*  73:0.39254 */ {allowable+ 689,  65, 133, 179, 185},
/*  74:0.50920 */ {allowable+ 653, 101,  89, 188, 194},
/*  75:0.40208 */ {allowable+ 679,  75, 118, 180, 186},
/*  76:0.51874 */ {allowable+ 653, 101,  89, 189, 195},
/*  77:0.54341 */ {allowable+ 646, 108,  83, 191, 197},
/*  78:0.66008 */ {allowable+ 616, 138,  62, 200, 206},
/*  79:0.23453 */ {allowable+ 716,  38, 194, 166, 173},
/*  80:0.35120 */ {allowable+ 692,  62, 138, 176, 182},
/*  81:0.37587 */ {allowable+ 689,  65, 133, 178, 184},
/*  82:0.49253 */ {allowable+ 665,  89, 101, 187, 193},
/*  83:0.38541 */ {allowable+ 689,  65, 133, 179, 185},
/*  84:0.50207 */ {allowable+ 653, 101,  89, 188, 194},
/*  85:0.52674 */ {allowable+ 653, 101,  89, 190, 196},
/*  86:0.64341 */ {allowable+ 616, 138,  62, 198, 205},
/*  87:0.38034 */ {allowable+ 689,  65, 133, 178, 184},
/*  88:0.49701 */ {allowable+ 665,  89, 101, 187, 193},
/*  89:0.52168 */ {allowable+ 653, 101,  89, 189, 195},
/*  90:0.63834 */ {allowable+ 616, 138,  62, 198, 204},
/*  91:0.53122 */ {allowable+ 653, 101,  89, 190, 196},
/*  92:0.64788 */ {allowable+ 616, 138,  62, 199, 205},
/*  93:0.67255 */ {allowable+ 592, 162,  50, 201, 207},
/*  94:0.78922 */ {allowable+ 543, 211,  33, 210, 217},
/*  95:0.21078 */ {allowable+ 721,  33, 211, 164, 171},
/*  96:0.32745 */ {allowable+ 704,  50, 162, 174, 180},
/*  97:0.35212 */ {allowable+ 692,  62, 138, 176, 182},
/*  98:0.46878 */ {allowable+ 665,  89, 101, 185, 191},
/*  99:0.36166 */ {allowable+ 692,  62, 138, 177, 183},
/* 100:0.47832 */ {allowable+ 665,  89, 101, 186, 192},
/* 101:0.50299 */ {allowable+ 653, 101,  89, 188, 194},
/* 102:0.61966 */ {allowable+ 621, 133,  65, 197, 203},
/* 103:0.35659 */ {allowable+ 692,  62, 138, 176, 183},
/* 104:0.47326 */ {allowable+ 665,  89, 101, 185, 191},
/* 105:0.49793 */ {allowable+ 665,  89, 101, 187, 193},
/* 106:0.61459 */ {allowable+ 621, 133,  65, 196, 202},
/* 107:0.50747 */ {allowable+ 653, 101,  89, 188, 194},
/* 108:0.62413 */ {allowable+ 621, 133,  65, 197, 203},
/* 109:0.64880 */ {allowable+ 616, 138,  62, 199, 205},
/* 110:0.76547 */ {allowable+ 560, 194,  38, 208, 215},
/* 111:0.33992 */ {allowable+ 692,  62, 138, 175, 181},
```

```
/* 112:0.45659 */ {allowable+ 671,  83, 108, 184, 190},
/* 113:0.48126 */ {allowable+ 665,  89, 101, 186, 192},
/* 114:0.59792 */ {allowable+ 636, 118,  75, 195, 201},
/* 115:0.49080 */ {allowable+ 665,  89, 101, 187, 193},
/* 116:0.60746 */ {allowable+ 621, 133,  65, 196, 202},
/* 117:0.63213 */ {allowable+ 616, 138,  62, 198, 204},
/* 118:0.74880 */ {allowable+ 569, 185,  41, 207, 213},
/* 119:0.48574 */ {allowable+ 665,  89, 101, 186, 192},
/* 120:0.60240 */ {allowable+ 621, 133,  65, 195, 201},
/* 121:0.62707 */ {allowable+ 616, 138,  62, 197, 203},
/* 122:0.74374 */ {allowable+ 569, 185,  41, 206, 213},
/* 123:0.63661 */ {allowable+ 616, 138,  62, 198, 204},
/* 124:0.75328 */ {allowable+ 560, 194,  38, 207, 214},
/* 125:0.77794 */ {allowable+ 543, 211,  33, 209, 216},
/* 126:0.89461 */ {allowable+ 451, 303,  16, 220, 229},
/* 127:0.00004 */ {allowable+ 753,   1, 716, 127, 141},
/* 128:0.00034 */ {allowable+ 753,   1, 716, 127, 141},
/* 129:0.00094 */ {allowable+ 753,   1, 716, 128, 141},
/* 130:0.00184 */ {allowable+ 753,   1, 716, 129, 141},
/* 131:0.00305 */ {allowable+ 753,   1, 716, 130, 141},
/* 132:0.00455 */ {allowable+ 753,   1, 716, 131, 142},
/* 133:0.00635 */ {allowable+ 753,   1, 716, 132, 142},
/* 134:0.00845 */ {allowable+ 753,   1, 716, 133, 143},
/* 135:0.01084 */ {allowable+ 752,   2, 606, 134, 143},
/* 136:0.01353 */ {allowable+ 752,   2, 606, 135, 144},
/* 137:0.01651 */ {allowable+ 751,   3, 538, 136, 144},
/* 138:0.01978 */ {allowable+ 751,   3, 538, 137, 145},
/* 139:0.02335 */ {allowable+ 750,   4, 496, 138, 145},
/* 140:0.02720 */ {allowable+ 750,   4, 496, 139, 146},
/* 141:0.03133 */ {allowable+ 749,   5, 462, 140, 147},
/* 142:0.03575 */ {allowable+ 749,   5, 462, 141, 147},
/* 143:0.04044 */ {allowable+ 748,   6, 436, 142, 148},
/* 144:0.04542 */ {allowable+ 748,   6, 436, 143, 149},
/* 145:0.05066 */ {allowable+ 747,   7, 415, 144, 150},
/* 146:0.05618 */ {allowable+ 745,   9, 380, 145, 151},
/* 147:0.06196 */ {allowable+ 745,   9, 380, 146, 151},
/* 148:0.06801 */ {allowable+ 744,  10, 366, 147, 152},
/* 149:0.07432 */ {allowable+ 743,  11, 353, 148, 153},
/* 150:0.08089 */ {allowable+ 741,  13, 330, 149, 154},
/* 151:0.08771 */ {allowable+ 741,  13, 330, 150, 155},
/* 152:0.09477 */ {allowable+ 741,  13, 330, 151, 156},
/* 153:0.10208 */ {allowable+ 738,  16, 303, 152, 156},
/* 154:0.10963 */ {allowable+ 738,  16, 303, 153, 157},
/* 155:0.11742 */ {allowable+ 737,  17, 295, 154, 158},
```

```
/* 156:0.12543 */ {allowable+ 736,  18, 288, 155, 159},
/* 157:0.13367 */ {allowable+ 732,  22, 262, 156, 160},
/* 158:0.14213 */ {allowable+ 732,  22, 262, 157, 161},
/* 159:0.15081 */ {allowable+ 731,  23, 256, 158, 162},
/* 160:0.15970 */ {allowable+ 731,  23, 256, 159, 163},
/* 161:0.16879 */ {allowable+ 727,  27, 236, 160, 164},
/* 162:0.17808 */ {allowable+ 727,  27, 236, 161, 165},
/* 163:0.18757 */ {allowable+ 727,  27, 236, 162, 165},
/* 164:0.19724 */ {allowable+ 722,  32, 215, 163, 166},
/* 165:0.20710 */ {allowable+ 721,  33, 211, 164, 167},
/* 166:0.21713 */ {allowable+ 721,  33, 211, 165, 168},
/* 167:0.22734 */ {allowable+ 716,  38, 194, 166, 169},
/* 168:0.23771 */ {allowable+ 716,  38, 194, 167, 170},
/* 169:0.24823 */ {allowable+ 716,  38, 194, 168, 171},
/* 170:0.25891 */ {allowable+ 713,  41, 185, 169, 172},
/* 171:0.26973 */ {allowable+ 709,  45, 174, 170, 173},
/* 172:0.28069 */ {allowable+ 709,  45, 174, 171, 174},
/* 173:0.29179 */ {allowable+ 709,  45, 174, 172, 175},
/* 174:0.30300 */ {allowable+ 704,  50, 162, 173, 176},
/* 175:0.31434 */ {allowable+ 704,  50, 162, 174, 177},
/* 176:0.32579 */ {allowable+ 704,  50, 162, 175, 178},
/* 177:0.33734 */ {allowable+ 692,  62, 138, 176, 179},
/* 178:0.34900 */ {allowable+ 692,  62, 138, 177, 180},
/* 179:0.36074 */ {allowable+ 692,  62, 138, 178, 181},
/* 180:0.37257 */ {allowable+ 692,  62, 138, 179, 182},
/* 181:0.38447 */ {allowable+ 689,  65, 133, 180, 183},
/* 182:0.39644 */ {allowable+ 689,  65, 133, 181, 184},
/* 183:0.40848 */ {allowable+ 679,  75, 118, 182, 184},
/* 184:0.42057 */ {allowable+ 679,  75, 118, 183, 185},
/* 185:0.43271 */ {allowable+ 679,  75, 118, 184, 186},
/* 186:0.44489 */ {allowable+ 671,  83, 108, 185, 187},
/* 187:0.45710 */ {allowable+ 671,  83, 108, 186, 188},
/* 188:0.46934 */ {allowable+ 665,  89, 101, 187, 189},
/* 189:0.48160 */ {allowable+ 665,  89, 101, 188, 190},
/* 190:0.49386 */ {allowable+ 665,  89, 101, 189, 191},
/* 191:0.50614 */ {allowable+ 653, 101,  89, 190, 192},
/* 192:0.51840 */ {allowable+ 653, 101,  89, 191, 193},
/* 193:0.53066 */ {allowable+ 653, 101,  89, 192, 194},
/* 194:0.54290 */ {allowable+ 646, 108,  83, 193, 195},
/* 195:0.55511 */ {allowable+ 646, 108,  83, 194, 196},
/* 196:0.56729 */ {allowable+ 636, 118,  75, 195, 197},
/* 197:0.57943 */ {allowable+ 636, 118,  75, 196, 198},
/* 198:0.59152 */ {allowable+ 636, 118,  75, 197, 199},
/* 199:0.60356 */ {allowable+ 621, 133,  65, 197, 200},
```

```
/* 200:0.61553 */ {allowable+ 621, 133,  65, 198, 201},
/* 201:0.62743 */ {allowable+ 616, 138,  62, 199, 202},
/* 202:0.63926 */ {allowable+ 616, 138,  62, 200, 203},
/* 203:0.65100 */ {allowable+ 616, 138,  62, 201, 204},
/* 204:0.66266 */ {allowable+ 616, 138,  62, 202, 205},
/* 205:0.67421 */ {allowable+ 592, 162,  50, 203, 206},
/* 206:0.68566 */ {allowable+ 592, 162,  50, 204, 207},
/* 207:0.69700 */ {allowable+ 592, 162,  50, 205, 208},
/* 208:0.70821 */ {allowable+ 580, 174,  45, 206, 209},
/* 209:0.71931 */ {allowable+ 580, 174,  45, 207, 210},
/* 210:0.73027 */ {allowable+ 580, 174,  45, 208, 211},
/* 211:0.74109 */ {allowable+ 569, 185,  41, 209, 212},
/* 212:0.75177 */ {allowable+ 560, 194,  38, 210, 213},
/* 213:0.76229 */ {allowable+ 560, 194,  38, 211, 214},
/* 214:0.77266 */ {allowable+ 560, 194,  38, 212, 215},
/* 215:0.78287 */ {allowable+ 543, 211,  33, 213, 216},
/* 216:0.79290 */ {allowable+ 543, 211,  33, 214, 217},
/* 217:0.80276 */ {allowable+ 539, 215,  32, 215, 218},
/* 218:0.81243 */ {allowable+ 518, 236,  27, 216, 219},
/* 219:0.82192 */ {allowable+ 518, 236,  27, 216, 220},
/* 220:0.83121 */ {allowable+ 518, 236,  27, 217, 221},
/* 221:0.84030 */ {allowable+ 498, 256,  23, 218, 222},
/* 222:0.84919 */ {allowable+ 498, 256,  23, 219, 223},
/* 223:0.85787 */ {allowable+ 492, 262,  22, 220, 224},
/* 224:0.86633 */ {allowable+ 492, 262,  22, 221, 225},
/* 225:0.87457 */ {allowable+ 466, 288,  18, 222, 226},
/* 226:0.88258 */ {allowable+ 459, 295,  17, 223, 227},
/* 227:0.89037 */ {allowable+ 451, 303,  16, 224, 228},
/* 228:0.89792 */ {allowable+ 451, 303,  16, 225, 229},
/* 229:0.90523 */ {allowable+ 424, 330,  13, 225, 230},
/* 230:0.91229 */ {allowable+ 424, 330,  13, 226, 231},
/* 231:0.91911 */ {allowable+ 424, 330,  13, 227, 232},
/* 232:0.92568 */ {allowable+ 401, 353,  11, 228, 233},
/* 233:0.93199 */ {allowable+ 388, 366,  10, 229, 234},
/* 234:0.93804 */ {allowable+ 374, 380,   9, 230, 235},
/* 235:0.94382 */ {allowable+ 374, 380,   9, 230, 236},
/* 236:0.94934 */ {allowable+ 339, 415,   7, 231, 237},
/* 237:0.95458 */ {allowable+ 318, 436,   6, 232, 238},
/* 238:0.95956 */ {allowable+ 318, 436,   6, 233, 239},
/* 239:0.96425 */ {allowable+ 292, 462,   5, 234, 240},
/* 240:0.96867 */ {allowable+ 292, 462,   5, 234, 241},
/* 241:0.97280 */ {allowable+ 258, 496,   4, 235, 242},
/* 242:0.97665 */ {allowable+ 258, 496,   4, 236, 243},
/* 243:0.98022 */ {allowable+ 216, 538,   3, 236, 244},
```

```
/* 244:0.98349 */ {allowable+ 216, 538,   3, 237, 245},
/* 245:0.98647 */ {allowable+ 148, 606,   2, 237, 246},
/* 246:0.98916 */ {allowable+ 148, 606,   2, 238, 247},
/* 247:0.99155 */ {allowable+  38, 716,   1, 238, 248},
/* 248:0.99365 */ {allowable+  38, 716,   1, 239, 249},
/* 249:0.99545 */ {allowable+  38, 716,   1, 239, 250},
/* 250:0.99695 */ {allowable+  38, 716,   1, 240, 251},
/* 251:0.99816 */ {allowable+  38, 716,   1, 240, 252},
/* 252:0.99906 */ {allowable+  38, 716,   1, 240, 253},
/* 253:0.99966 */ {allowable+  38, 716,   1, 240, 254},
/* 254:0.99996 */ {allowable+  38, 716,   1, 240, 254},
#else
/*   0:0.50000 */ {allowable+ 665,  89, 101,   1,   2},
/*   1:0.32281 */ {allowable+ 704,  50, 162,   3,   4},
/*   2:0.65614 */ {allowable+ 616, 138,  62,   5,   6},
/*   3:0.14678 */ {allowable+ 731,  23, 256,   7,   8},
/*   4:0.43678 */ {allowable+ 679,  75, 118,   9,  10},
/*   5:0.52228 */ {allowable+ 653, 101,  89,  11,  12},
/*   6:0.81228 */ {allowable+ 518, 236,  27,  13,  14},
/*   7:0.03670 */ {allowable+ 749,   5, 462,  15,  16},
/*   8:0.20427 */ {allowable+ 721,  33, 211,  17,  18},
/*   9:0.30122 */ {allowable+ 704,  50, 162,  19,  20},
/*  10:0.54788 */ {allowable+ 646, 108,  83,  21,  22},
/*  11:0.40125 */ {allowable+ 679,  75, 118,  23,  24},
/*  12:0.64791 */ {allowable+ 616, 138,  62,  25,  26},
/*  13:0.73060 */ {allowable+ 580, 174,  45,  27,  28},
/*  14:0.97727 */ {allowable+ 216, 538,   3,  29,  30},
/*  15:0.00917 */ {allowable+ 752,   2, 606,  31,  32},
/*  16:0.04451 */ {allowable+ 748,   6, 436,  33,  34},
/*  17:0.07792 */ {allowable+ 743,  11, 353,  35,  36},
/*  18:0.28125 */ {allowable+ 709,  45, 174,  37,  38},
/*  19:0.18273 */ {allowable+ 727,  27, 236,  39,  40},
/*  20:0.38606 */ {allowable+ 689,  65, 133,  41,  42},
/*  21:0.44983 */ {allowable+ 671,  83, 108,  43,  44},
/*  22:0.65317 */ {allowable+ 616, 138,  62,  45,  46},
/*  23:0.29689 */ {allowable+ 704,  50, 162,  47,  48},
/*  24:0.50022 */ {allowable+ 653, 101,  89,  49,  50},
/*  25:0.55589 */ {allowable+ 636, 118,  75,  51,  52},
/*  26:0.75922 */ {allowable+ 560, 194,  38,  53,  54},
/*  27:0.64678 */ {allowable+ 616, 138,  62,  55,  56},
/*  28:0.85011 */ {allowable+ 498, 256,  23,  57,  58},
/*  29:0.97318 */ {allowable+ 258, 496,   4,  59,  60},
/*  30:0.99432 */ {allowable+  38, 716,   1,  61,  62},
/*  31:0.00229 */ {allowable+ 753,   1, 716,  63,  64},
```

```
/*   32:0.01163 */  {allowable+ 752,    2, 606,  65,   66},
/*   33:0.01113 */  {allowable+ 752,    2, 606,  67,   68},
/*   34:0.05438 */  {allowable+ 746,    8, 397,  69,   70},
/*   35:0.01948 */  {allowable+ 751,    3, 538,  71,   72},
/*   36:0.09947 */  {allowable+ 741,   13, 330,  73,   74},
/*   37:0.18492 */  {allowable+ 727,   27, 236,  75,   76},
/*   38:0.34492 */  {allowable+ 692,   62, 138,  77,   78},
/*   39:0.07689 */  {allowable+ 743,   11, 353,  79,   80},
/*   40:0.23689 */  {allowable+ 716,   38, 194,  81,   82},
/*   41:0.29603 */  {allowable+ 704,   50, 162,  83,   84},
/*   42:0.45603 */  {allowable+ 671,   83, 108,  85,   86},
/*   43:0.36736 */  {allowable+ 692,   62, 138,  87,   88},
/*   44:0.52736 */  {allowable+ 653,  101,  89,  89,   90},
/*   45:0.57984 */  {allowable+ 636,  118,  75,  91,   92},
/*   46:0.73984 */  {allowable+ 569,  185,  41,  93,   94},
/*   47:0.20446 */  {allowable+ 721,   33, 211,  95,   96},
/*   48:0.36446 */  {allowable+ 692,   62, 138,  97,   98},
/*   49:0.42524 */  {allowable+ 679,   75, 118,  99,  100},
/*   50:0.58524 */  {allowable+ 636,  118,  75, 101,  102},
/*   51:0.48500 */  {allowable+ 665,   89, 101, 103,  104},
/*   52:0.64500 */  {allowable+ 616,  138,  62, 105,  106},
/*   53:0.69813 */  {allowable+ 592,  162,  50, 107,  108},
/*   54:0.85813 */  {allowable+ 492,  262,  22, 109,  110},
/*   55:0.57456 */  {allowable+ 636,  118,  75, 111,  112},
/*   56:0.73456 */  {allowable+ 569,  185,  41, 113,  114},
/*   57:0.79610 */  {allowable+ 543,  211,  33, 115,  116},
/*   58:0.95610 */  {allowable+ 318,  436,   6, 117,  118},
/*   59:0.96730 */  {allowable+ 292,  462,   5, 119,  120},
/*   60:0.99330 */  {allowable+  38,  716,   1, 121,  122},
/*   61:0.99328 */  {allowable+  38,  716,   1, 123,  124},
/*   62:0.99858 */  {allowable+  38,  716,   1, 125,  126},
/*   63:0.00057 */  {allowable+ 753,    1, 716, 127,  130},
/*   64:0.00334 */  {allowable+ 753,    1, 716, 129,  132},
/*   65:0.00291 */  {allowable+ 753,    1, 716, 129,  132},
/*   66:0.01407 */  {allowable+ 752,    2, 606, 131,  137},
/*   67:0.00278 */  {allowable+ 753,    1, 716, 129,  132},
/*   68:0.01358 */  {allowable+ 752,    2, 606, 131,  137},
/*   69:0.01359 */  {allowable+ 752,    2, 606, 131,  137},
/*   70:0.06867 */  {allowable+ 744,   10, 366, 137,  151},
/*   71:0.00487 */  {allowable+ 753,    1, 716, 129,  133},
/*   72:0.02358 */  {allowable+ 750,    4, 496, 133,  141},
/*   73:0.01347 */  {allowable+ 752,    2, 606, 131,  137},
/*   74:0.13014 */  {allowable+ 736,   18, 288, 150,  159},
/*   75:0.10736 */  {allowable+ 738,   16, 303, 146,  157},
```

```
/*  76:0.22403 */ {allowable+ 721,  33, 211, 162, 169},
/*  77:0.28197 */ {allowable+ 709,  45, 174, 168, 175},
/*  78:0.39863 */ {allowable+ 689,  65, 133, 179, 185},
/*  79:0.01922 */ {allowable+ 751,   3, 538, 132, 139},
/*  80:0.09846 */ {allowable+ 741,  13, 330, 144, 155},
/*  81:0.16535 */ {allowable+ 731,  23, 256, 155, 163},
/*  82:0.28202 */ {allowable+ 709,  45, 174, 168, 175},
/*  83:0.22875 */ {allowable+ 716,  38, 194, 163, 170},
/*  84:0.34541 */ {allowable+ 692,  62, 138, 174, 181},
/*  85:0.39516 */ {allowable+ 689,  65, 133, 178, 184},
/*  86:0.51183 */ {allowable+ 653, 101,  89, 189, 195},
/*  87:0.30158 */ {allowable+ 704,  50, 162, 170, 177},
/*  88:0.41825 */ {allowable+ 679,  75, 118, 181, 187},
/*  89:0.46952 */ {allowable+ 665,  89, 101, 185, 191},
/*  90:0.58618 */ {allowable+ 636, 118,  75, 195, 201},
/*  91:0.52532 */ {allowable+ 653, 101,  89, 190, 196},
/*  92:0.64199 */ {allowable+ 616, 138,  62, 199, 206},
/*  93:0.69473 */ {allowable+ 592, 162,  50, 204, 211},
/*  94:0.81140 */ {allowable+ 518, 236,  27, 215, 223},
/*  95:0.13193 */ {allowable+ 736,  18, 288, 150, 159},
/*  96:0.24859 */ {allowable+ 716,  38, 194, 165, 172},
/*  97:0.29904 */ {allowable+ 704,  50, 162, 170, 176},
/*  98:0.41571 */ {allowable+ 679,  75, 118, 180, 186},
/*  99:0.36249 */ {allowable+ 692,  62, 138, 176, 182},
/* 100:0.47915 */ {allowable+ 665,  89, 101, 186, 192},
/* 101:0.53008 */ {allowable+ 653, 101,  89, 190, 196},
/* 102:0.64675 */ {allowable+ 616, 138,  62, 200, 206},
/* 103:0.42473 */ {allowable+ 679,  75, 118, 181, 187},
/* 104:0.54139 */ {allowable+ 646, 108,  83, 191, 197},
/* 105:0.59256 */ {allowable+ 636, 118,  75, 195, 201},
/* 106:0.70923 */ {allowable+ 580, 174,  45, 206, 212},
/* 107:0.64954 */ {allowable+ 616, 138,  62, 200, 206},
/* 108:0.76621 */ {allowable+ 560, 194,  38, 211, 218},
/* 109:0.81883 */ {allowable+ 518, 236,  27, 216, 224},
/* 110:0.93550 */ {allowable+ 374, 380,   9, 231, 246},
/* 111:0.52068 */ {allowable+ 653, 101,  89, 189, 195},
/* 112:0.63734 */ {allowable+ 616, 138,  62, 199, 205},
/* 113:0.69021 */ {allowable+ 592, 162,  50, 204, 210},
/* 114:0.80688 */ {allowable+ 539, 215,  32, 215, 222},
/* 115:0.75187 */ {allowable+ 560, 194,  38, 209, 217},
/* 116:0.86854 */ {allowable+ 466, 288,  18, 222, 231},
/* 117:0.94624 */ {allowable+ 357, 397,   8, 233, 245},
/* 118:0.98903 */ {allowable+ 148, 606,   2, 245, 250},
/* 119:0.95945 */ {allowable+ 318, 436,   6, 236, 246},
```

```
/* 120:0.99182 */ {allowable+  38, 716,    1, 247, 251},
/* 121:0.99226 */ {allowable+  38, 716,    1, 247, 251},
/* 122:0.99832 */ {allowable+  38, 716,    1, 250, 253},
/* 123:0.99224 */ {allowable+  38, 716,    1, 247, 251},
/* 124:0.99832 */ {allowable+  38, 716,    1, 250, 253},
/* 125:0.99753 */ {allowable+  38, 716,    1, 250, 252},
/* 126:0.99964 */ {allowable+  38, 716,    1, 251, 254},
/* 127:0.00004 */ {allowable+ 753,   1, 716, 127, 129},
/* 128:0.00034 */ {allowable+ 753,   1, 716, 127, 130},
/* 129:0.00094 */ {allowable+ 753,   1, 716, 128, 130},
/* 130:0.00184 */ {allowable+ 753,   1, 716, 128, 131},
/* 131:0.00305 */ {allowable+ 753,   1, 716, 129, 132},
/* 132:0.00455 */ {allowable+ 753,   1, 716, 129, 133},
/* 133:0.00635 */ {allowable+ 753,   1, 716, 130, 134},
/* 134:0.00845 */ {allowable+ 753,   1, 716, 130, 135},
/* 135:0.01084 */ {allowable+ 752,   2, 606, 131, 136},
/* 136:0.01353 */ {allowable+ 752,   2, 606, 131, 137},
/* 137:0.01651 */ {allowable+ 751,   3, 538, 132, 138},
/* 138:0.01978 */ {allowable+ 751,   3, 538, 132, 139},
/* 139:0.02335 */ {allowable+ 750,   4, 496, 129, 141},
/* 140:0.02720 */ {allowable+ 750,   4, 496, 132, 142},
/* 141:0.03133 */ {allowable+ 749,   5, 462, 134, 143},
/* 142:0.03575 */ {allowable+ 749,   5, 462, 136, 143},
/* 143:0.04044 */ {allowable+ 748,   6, 436, 137, 144},
/* 144:0.04542 */ {allowable+ 748,   6, 436, 139, 145},
/* 145:0.05066 */ {allowable+ 747,   7, 415, 140, 146},
/* 146:0.05618 */ {allowable+ 745,   9, 380, 142, 147},
/* 147:0.06196 */ {allowable+ 745,   9, 380, 143, 148},
/* 148:0.06801 */ {allowable+ 744,  10, 366, 144, 149},
/* 149:0.07432 */ {allowable+ 743,  11, 353, 145, 150},
/* 150:0.08089 */ {allowable+ 741,  13, 330, 147, 151},
/* 151:0.08771 */ {allowable+ 741,  13, 330, 148, 152},
/* 152:0.09477 */ {allowable+ 741,  13, 330, 149, 153},
/* 153:0.10208 */ {allowable+ 738,  16, 303, 150, 154},
/* 154:0.10963 */ {allowable+ 738,  16, 303, 151, 155},
/* 155:0.11742 */ {allowable+ 737,  17, 295, 152, 156},
/* 156:0.12543 */ {allowable+ 736,  18, 288, 153, 157},
/* 157:0.13367 */ {allowable+ 732,  22, 262, 155, 158},
/* 158:0.14213 */ {allowable+ 732,  22, 262, 156, 159},
/* 159:0.15081 */ {allowable+ 731,  23, 256, 157, 160},
/* 160:0.15970 */ {allowable+ 731,  23, 256, 158, 161},
/* 161:0.16879 */ {allowable+ 727,  27, 236, 159, 162},
/* 162:0.17808 */ {allowable+ 727,  27, 236, 160, 163},
/* 163:0.18757 */ {allowable+ 727,  27, 236, 161, 164},
```

```
/* 164:0.19724 */ {allowable+ 722,  32, 215, 162, 165},
/* 165:0.20710 */ {allowable+ 721,  33, 211, 163, 166},
/* 166:0.21713 */ {allowable+ 721,  33, 211, 164, 167},
/* 167:0.22734 */ {allowable+ 716,  38, 194, 165, 168},
/* 168:0.23771 */ {allowable+ 716,  38, 194, 166, 169},
/* 169:0.24823 */ {allowable+ 716,  38, 194, 167, 170},
/* 170:0.25891 */ {allowable+ 713,  41, 185, 168, 171},
/* 171:0.26973 */ {allowable+ 709,  45, 174, 169, 172},
/* 172:0.28069 */ {allowable+ 709,  45, 174, 170, 173},
/* 173:0.29179 */ {allowable+ 709,  45, 174, 171, 174},
/* 174:0.30300 */ {allowable+ 704,  50, 162, 172, 175},
/* 175:0.31434 */ {allowable+ 704,  50, 162, 173, 176},
/* 176:0.32579 */ {allowable+ 704,  50, 162, 174, 177},
/* 177:0.33734 */ {allowable+ 692,  62, 138, 176, 178},
/* 178:0.34900 */ {allowable+ 692,  62, 138, 177, 179},
/* 179:0.36074 */ {allowable+ 692,  62, 138, 178, 180},
/* 180:0.37257 */ {allowable+ 692,  62, 138, 179, 181},
/* 181:0.38447 */ {allowable+ 689,  65, 133, 180, 182},
/* 182:0.39644 */ {allowable+ 689,  65, 133, 181, 183},
/* 183:0.40848 */ {allowable+ 679,  75, 118, 182, 184},
/* 184:0.42057 */ {allowable+ 679,  75, 118, 183, 185},
/* 185:0.43271 */ {allowable+ 679,  75, 118, 184, 186},
/* 186:0.44489 */ {allowable+ 671,  83, 108, 185, 187},
/* 187:0.45710 */ {allowable+ 671,  83, 108, 186, 188},
/* 188:0.46934 */ {allowable+ 665,  89, 101, 187, 189},
/* 189:0.48160 */ {allowable+ 665,  89, 101, 188, 190},
/* 190:0.49386 */ {allowable+ 665,  89, 101, 189, 191},
/* 191:0.50614 */ {allowable+ 653, 101,  89, 190, 192},
/* 192:0.51840 */ {allowable+ 653, 101,  89, 191, 193},
/* 193:0.53066 */ {allowable+ 653, 101,  89, 192, 194},
/* 194:0.54290 */ {allowable+ 646, 108,  83, 193, 195},
/* 195:0.55511 */ {allowable+ 646, 108,  83, 194, 196},
/* 196:0.56729 */ {allowable+ 636, 118,  75, 195, 197},
/* 197:0.57943 */ {allowable+ 636, 118,  75, 196, 198},
/* 198:0.59152 */ {allowable+ 636, 118,  75, 197, 199},
/* 199:0.60356 */ {allowable+ 621, 133,  65, 198, 200},
/* 200:0.61553 */ {allowable+ 621, 133,  65, 199, 201},
/* 201:0.62743 */ {allowable+ 616, 138,  62, 200, 202},
/* 202:0.63926 */ {allowable+ 616, 138,  62, 201, 203},
/* 203:0.65100 */ {allowable+ 616, 138,  62, 202, 204},
/* 204:0.66266 */ {allowable+ 616, 138,  62, 203, 205},
/* 205:0.67421 */ {allowable+ 592, 162,  50, 204, 207},
/* 206:0.68566 */ {allowable+ 592, 162,  50, 205, 208},
/* 207:0.69700 */ {allowable+ 592, 162,  50, 206, 209},
```

```
/* 208:0.70821 */ {allowable+ 580, 174,  45, 207, 210},
/* 209:0.71931 */ {allowable+ 580, 174,  45, 208, 211},
/* 210:0.73027 */ {allowable+ 580, 174,  45, 209, 212},
/* 211:0.74109 */ {allowable+ 569, 185,  41, 210, 213},
/* 212:0.75177 */ {allowable+ 560, 194,  38, 211, 214},
/* 213:0.76229 */ {allowable+ 560, 194,  38, 212, 215},
/* 214:0.77266 */ {allowable+ 560, 194,  38, 213, 216},
/* 215:0.78287 */ {allowable+ 543, 211,  33, 214, 217},
/* 216:0.79290 */ {allowable+ 543, 211,  33, 215, 218},
/* 217:0.80276 */ {allowable+ 539, 215,  32, 216, 219},
/* 218:0.81243 */ {allowable+ 518, 236,  27, 217, 220},
/* 219:0.82192 */ {allowable+ 518, 236,  27, 218, 221},
/* 220:0.83121 */ {allowable+ 518, 236,  27, 219, 222},
/* 221:0.84030 */ {allowable+ 498, 256,  23, 220, 223},
/* 222:0.84919 */ {allowable+ 498, 256,  23, 221, 224},
/* 223:0.85787 */ {allowable+ 492, 262,  22, 222, 225},
/* 224:0.86633 */ {allowable+ 492, 262,  22, 223, 226},
/* 225:0.87457 */ {allowable+ 466, 288,  18, 224, 228},
/* 226:0.88258 */ {allowable+ 459, 295,  17, 225, 229},
/* 227:0.89037 */ {allowable+ 451, 303,  16, 226, 230},
/* 228:0.89792 */ {allowable+ 451, 303,  16, 227, 231},
/* 229:0.90523 */ {allowable+ 424, 330,  13, 228, 232},
/* 230:0.91229 */ {allowable+ 424, 330,  13, 229, 233},
/* 231:0.91911 */ {allowable+ 424, 330,  13, 230, 234},
/* 232:0.92568 */ {allowable+ 401, 353,  11, 231, 236},
/* 233:0.93199 */ {allowable+ 388, 366,  10, 232, 237},
/* 234:0.93804 */ {allowable+ 374, 380,   9, 233, 238},
/* 235:0.94382 */ {allowable+ 374, 380,   9, 234, 239},
/* 236:0.94934 */ {allowable+ 339, 415,   7, 235, 241},
/* 237:0.95458 */ {allowable+ 318, 436,   6, 236, 242},
/* 238:0.95956 */ {allowable+ 318, 436,   6, 237, 244},
/* 239:0.96425 */ {allowable+ 292, 462,   5, 238, 245},
/* 240:0.96867 */ {allowable+ 292, 462,   5, 238, 247},
/* 241:0.97280 */ {allowable+ 258, 496,   4, 239, 249},
/* 242:0.97665 */ {allowable+ 258, 496,   4, 240, 252},
/* 243:0.98022 */ {allowable+ 216, 538,   3, 242, 249},
/* 244:0.98349 */ {allowable+ 216, 538,   3, 243, 249},
/* 245:0.98647 */ {allowable+ 148, 606,   2, 244, 250},
/* 246:0.98916 */ {allowable+ 148, 606,   2, 245, 250},
/* 247:0.99155 */ {allowable+  38, 716,   1, 246, 251},
/* 248:0.99365 */ {allowable+  38, 716,   1, 247, 251},
/* 249:0.99545 */ {allowable+  38, 716,   1, 248, 252},
/* 250:0.99695 */ {allowable+  38, 716,   1, 249, 252},
/* 251:0.99816 */ {allowable+  38, 716,   1, 250, 253},
```

```
 /* 252:0.99906 */ {allowable+  38, 716,   1, 251, 253},
 /* 253:0.99966 */ {allowable+  38, 716,   1, 251, 254},
 /* 254:0.99996 */ {allowable+  38, 716,   1, 252, 254},
#endif
 /* 255:inactive*/ {allowable+  38,1509,1509, 255, 255} };

static struct
    {
    ushort internalBuffer;
    int jotCount;
    FILE *externalIn;
    } d;

/* Initialize decoding. */
void elsDecodeStart(FILE *file)
    {
    d.externalIn = file;
    d.jotCount = JOTS_PER_BYTE;
    /* Import first two bytes from external file. */
    d.internalBuffer = fgetc(file);
    d.internalBuffer = (d.internalBuffer<<8) | fgetc(file);
    }

/* Decode a single bit. */
int elsDecodeBit(ElsRung *rung)
    {
    /* Determine whether 0 or 1. */
    if (d.internalBuffer >= ladder[*rung].threshold[d.jotCount])
        {
        /* 1: */
        d.internalBuffer -= ladder[*rung].threshold[d.jotCount];
        d.jotCount -= ladder[*rung].codeLength1;
        #ifdef ALACRITOUS
        /* Update probability. */
        *rung = ladder[*rung].next1;
        #endif
        if (d.jotCount <= 0)
            {
            /* Import a byte. */
            if ((d.jotCount += JOTS_PER_BYTE)<=0)
                {
                fprintf(stderr,"elsDecodeBit:  called with inactive rung.\n");
                exit(1);
                }
```

```
            d.internalBuffer <<= 8;
            d.internalBuffer |= fgetc(d.externalIn);
            #ifndef ALACRITOUS
            /* Update probability. */
            *rung = ladder[*rung].next1;
            #endif
            }
        return 1;
        }
    else
        {
        /* 0: Buffer value need not change. */
        d.jotCount -= ladder[*rung].codeLength0;
        #ifdef ALACRITOUS
        /* Update probability */
        *rung = ladder[*rung].next0;
        #endif
        if (d.jotCount<=0)
            {
            /* Import a byte. */
            if ((d.jotCount += JOTS_PER_BYTE)<=0)
                {
                fprintf(stderr,"elsDecodeBit:  called with inactive rung.\n");
                exit(1);
                }
            d.internalBuffer <<= 8;
            d.internalBuffer |= fgetc(d.externalIn);
            #ifndef ALACRITOUS
            /* Update probability */
            *rung = ladder[*rung].next0;
            #endif
            }
        return 0;
        }
    }


/* Perform consistency check prior to ending decoding. */
int elsDecodeOk()
    {
    long x;

    if (d.jotCount==JOTS_PER_BYTE)
        x = 65536l;
    else
```

```
        x = allowable[d.jotCount+JOTS_PER_BYTE];
    x = (x*(d.jotCount-1))/JOTS_PER_BYTE;
    return d.internalBuffer == x;
    }

/* Conclude decoding. */
void elsDecodeEnd()
    {
    /* No action necessary. */
    }

static struct
    {
    ulong min;
    int jotCount;
    int backlog;
    FILE *externalOut;
    } e;

/* Initialize encoding. */
void elsEncodeStart(FILE *file)
    {
    e.backlog = 0;
    e.min = 0;
    e.jotCount = JOTS_PER_BYTE;
    e.externalOut = file;
    }

/* Encode a single bit. */
void elsEncodeBit(ElsRung *rung, int bit)
    {
    if (bit)
        {
        /* Encode a 1 */
        e.min += ladder[*rung].threshold[e.jotCount];
        e.jotCount -= ladder[*rung].codeLength1;
        #ifdef ALACRITOUS
        *rung = ladder[*rung].next1;
        #endif
        }
    else
        {
        /* Encode a 0. */
        e.jotCount -= ladder[*rung].codeLength0;
```

```
    #ifdef ALACRITOUS
    *rung = ladder[*rung].next0;
    #endif
    }
if (e.jotCount<=0) /* Will the decoder import a byte? */
    {
    ulong diffBits;

    /* First check for bytes ready to export. */
    if ((e.jotCount += JOTS_PER_BYTE)<=0)
        {
        fprintf(stderr,"elsEncodeBit:  called with inactive rung.\n");
        exit(1);
        }
    diffBits = (e.min + allowable[e.jotCount] - 1) ∧ e.min;
    switch(e.backlog)
        {
    default: /* 2 or greater */
        if (diffBits&0xFF000000)
            break;
        fputc(e.min>>24,e.externalOut);
        while(--e.backlog > 1)
            fputc((e.min>>16)&0xFF,e.externalOut);
    case 1:
        if (diffBits&0x00FF0000)
            break;
        fputc((e.min>>16)&0xFF,e.externalOut);
        e.backlog--;
    case 0:
        if (diffBits&0x0000FF00)
            break;
        fputc((e.min>>8)&0xFF,e.externalOut);
        e.backlog--;
        }
    e.backlog++;
    /* Now move one byte from preactive to active. */
    if (e.backlog>2)
        e.min = (e.min&0xFF000000)|((e.min&0x0000FFFF)<< 8);
    else
        e.min <<= 8;
    #ifndef ALACRITOUS
    if (bit)
        *rung = ladder[*rung].next1;
    else
```

```
            *rung = ladder[*rung].next0;
        #endif
        }
    }

/* Conclude encoding. */
void elsEncodeEnd()
    {
    long x;

    /*
     @ Consistency check.
     */
    if (e.jotCount==JOTS_PER_BYTE)
        x = 65536l;
    else
        x = allowable[e.jotCount+JOTS_PER_BYTE];
    x = (x*(e.jotCount-1))/JOTS_PER_BYTE;
    e.min += x;

    switch(e.backlog)
        {
    default:
        fputc((e.min>>24)&0xFF,e.externalOut);
        while(--e.backlog > 1)
            fputc((e.min>>16)&0xFF,e.externalOut);
    case 1:
        fputc((e.min>>16)&0xFF,e.externalOut);
    case 0:
        fputc((e.min>>8)&0xFF,e.externalOut);
        }
    fputc((e.min)&0xFF,e.externalOut);
    }
```

LISTING THREE: compress.c

```
/*
 @  "compress.c":  Test program for entropy coding: compresses a file
 @  using a one-byte data model.
 */
#include <stdio.h>
#include "elscoder.h"

ElsRung context[256];

int main(int argc,char *argv[])
    {
    FILE *inFile,*outFile;
    int buffer,cIndex;

    if (argc!=3)
        {
        fprintf(stderr,"Usage:\n\n compress <inFile> <outFile>\n\n"
         "Tests entropy coder with 1-byte-context file encoding.\n");
        return 1;
        }

    if (!(inFile = fopen(argv[1],"rb")))
        {
        printf("Unable to open input file.\n");
        return 1;
        }
    if (!(outFile = fopen(argv[2],"wb")))
        {
        printf("Unable to open output file.\n");
        return 1;
        }

    /*
     @  Initialize probabilities of state machine.
     */
    for (cIndex=0; cIndex<256; cIndex++)
        context[cIndex] = 0;

    elsEncodeStart(outFile);
    while((buffer = fgetc(inFile)) != EOF)
        {
        cIndex = 1;
```

```
    elsEncodeBit(context+cIndex,buffer&0x80);
    cIndex = (cIndex<<1) | !!(buffer&0x80);
    elsEncodeBit(context+cIndex,buffer&0x40);
    cIndex = (cIndex<<1) | !!(buffer&0x40);
    elsEncodeBit(context+cIndex,buffer&0x20);
    cIndex = (cIndex<<1) | !!(buffer&0x20);
    elsEncodeBit(context+cIndex,buffer&0x10);
    cIndex = (cIndex<<1) | !!(buffer&0x10);
    elsEncodeBit(context+cIndex,buffer&0x08);
    cIndex = (cIndex<<1) | !!(buffer&0x08);
    elsEncodeBit(context+cIndex,buffer&0x04);
    cIndex = (cIndex<<1) | !!(buffer&0x04);
    elsEncodeBit(context+cIndex,buffer&0x02);
    cIndex = (cIndex<<1) | !!(buffer&0x02);
    elsEncodeBit(context+cIndex,buffer&0x01);
    cIndex = (cIndex<<1) | !!(buffer&0x01);
    if (buffer==255)
        {
        /* Send 0 to distinguish from EOF marker. */
        elsEncodeBit(context,0);
        }
    }
/* Send 255 followed by 1 for EOF marker. */
elsEncodeBit(context+1,1);
elsEncodeBit(context+3,1);
elsEncodeBit(context+7,1);
elsEncodeBit(context+15,1);
elsEncodeBit(context+31,1);
elsEncodeBit(context+63,1);
elsEncodeBit(context+127,1);
elsEncodeBit(context+255,1);
elsEncodeBit(context,1);

elsEncodeEnd();
fclose(outFile);
fclose(inFile);

return 0;
}
```

LISTING FOUR: expand.c

```
/*
 @  "expand.c":  Test program for entropy coding:  decodes a file
 @  using a one-byte model.
 */
#include <stdio.h>
#include "elscoder.h"

ElsRung context[256];

int main(int argc,char *argv[])
    {

    FILE *inFile,*outFile;
    int buffer;

    if (argc!=3)
    {
        fprintf(stderr,"Usage:\n\nexpand <inFile> <outFile>\n\n"
         "Tests entropy encoder with 1-byte-context file decoding.\n");
        return 1;
    }

    if (!(inFile = fopen(argv[1],"rb")))
        {
        fprintf(stderr,"Unable to open input file.\n");
        return 1;
        }
    if (!(outFile = fopen(argv[2],"wb")))
        {
        fprintf(stderr,"Unable to open output file.\n");
        return 1;
        }

    /*
     @  Initialize probabilities of state machine.
     */
    for (buffer=0; buffer<256; buffer++)
        context[buffer] = 0;

    elsDecodeStart(inFile);

    while(1)
```

```
    {
    buffer=1;
    buffer = (buffer<<1) | elsDecodeBit(context+buffer);
    buffer = (buffer<<1) | elsDecodeBit(context+buffer);
    buffer = (buffer<<1) | elsDecodeBit(context+buffer);
    buffer = (buffer<<1) | elsDecodeBit(context+buffer);
    buffer = (buffer<<1) | elsDecodeBit(context+buffer);
    buffer = (buffer<<1) | elsDecodeBit(context+buffer);
    buffer = (buffer<<1) | elsDecodeBit(context+buffer);
    buffer = (buffer<<1) | elsDecodeBit(context+buffer);
    buffer &= 0xFF;
    if ((buffer==255)&& elsDecodeBit(context))
        {
        if (elsDecodeOk())
            printf("ELS Decoder consistency check succeeded.\n");
        else
            {
            fprintf(stderr,"ELS Decoder consistency check failed.\n");
            return 1;
            }
        elsDecodeEnd();
        fclose(outFile);
        fclose(inFile);
        return 0;
        }
    fputc(buffer,outFile);
    }
}
```

DEPARTMENT OF MATHEMATICS, UNITED STATES NAVAL ACADEMY, ANNAPOLIS, MARYLAND 21402