LIBRARY OF CONGRESS

Office of Business Enterprises
Duplication Services Section

THIS IS TO CERTIFY that the collections of the Library of Congress contain a bound volume entitled **THE DATA COMPRESSION BOOK,** call number QA 76.9.D33 N46, Copy 2. The attached – Cover Page, Title Page, Second Title page, Copyright Page, Table of Contents Pages and Chapter 5 - are a true and complete representation from that work.

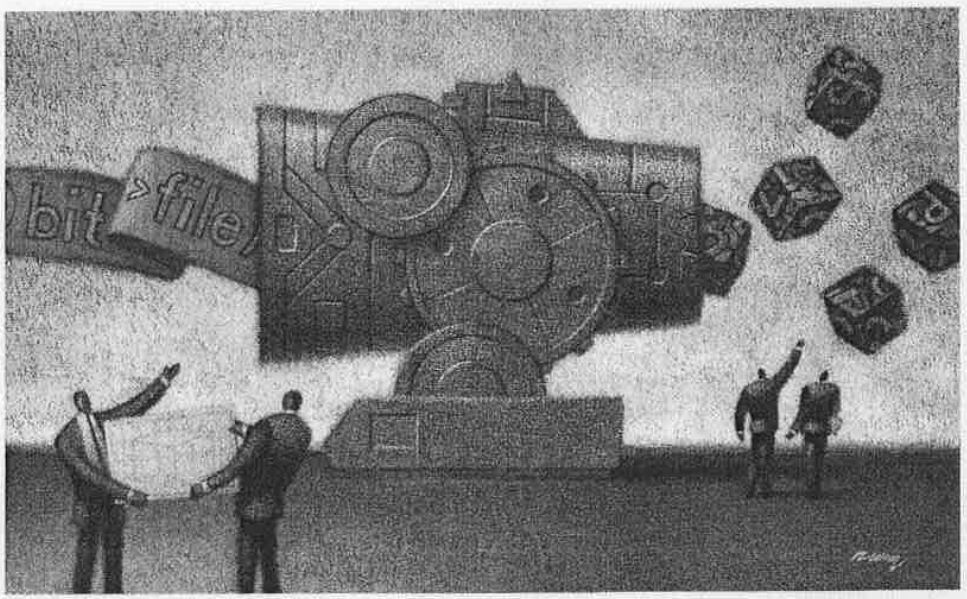
THIS IS TO CERTIFY FURTHER, that work is marked with a Library of Congress Copyright Office stamp dated January 29, 1992.

IN WITNESS WHEREOF, the seal of the Library of Congress is affixed hereto on May 18, 2018.

Deirdre Scott

Deirdre Scott
Business Enterprises Officer
Office of Business Enterprises
Library of Congress

The Data Compression Book



Featuring fast, efficient data compression techniques in C

Mark Nelson

M&T BOOKS

The Data Compression Book

The Data Compression Book

Featuring fast, efficient data compression techniques in C

Mark Nelson

M&T Books
A Division of M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063

Printed in the United States of America

Limits of Liability and Disclaimer of Warranty
The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Library of Congress Cataloging-in-Publication Data

Nelson, Mark, 1958-
The data compression book/ by Mark Nelson.
p. cm.
Includes index.
ISBN 1-55851-214-4: $26.95 --ISBN 1-55851-216-0 (Book disk set): $36.95
1. Data Compression (Computer Science) I. Title.
QA76.9.D33N46 1991
005.74'6--dc20 91-23080
CIP

Editor: Tova F. Fliegel
Technical review: Robert Jung

Cover Design: Lauren Smith Design
Layout: Marni Tapscott

94 93 92 91 4 3 2 1

Contents

CHAPTER 5

Huffman One Better: Arithmetic Coding

The last two chapters show that Huffman coding uses knowledge about information content to efficiently encode symbols. If the probability of a symbol's appearance in a message is known, Huffman techniques can encode that symbol using a minimal number of bits. Huffman coding has been proven the best fixed-length coding method available.

Difficulties

Huffman codes have to be an integral number of bits long, and this can sometimes be a problem. If the probability of a character is 1/3, for example, the optimum number of bits to code that character is around 1.6. Huffman coding has to assign either one or two bits to the code, and either choice leads to a longer compressed message than is theoretically possible.

This nonoptimal coding becomes a noticeable problem when the probability of a character is very high. If a statistical method could assign a 90 percent probability to a given character, the optimal code size would be 0.15 bits. The Huffman coding system would probably assign a 1-bit code to the symbol, which is six times longer than necessary.

This would be a problem when compressing two-color images, like those coming from a fax machine. Since there are only two colors, an ordinary coding method would assign the 1 bit to one color and the 0 bit to the other. Since both codes have only a single bit, Huffman coding is not going to be able to compress this data at all. No matter how high the probability of one of the bits, we are still going to have to encode it using one bit.

The conventional solution to this problem is to group the bits into packets and apply Huffman coding. But this weakness prevents Huffman coding from being a universal compressor.

Arithmetic Coding: A Step Forward

Only in the last ten years has a respectable candidate to replace Huffman coding been successfully demonstrated: arithmetic coding. Arithmetic coding bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point output number. More bits are needed in the output number for longer, complex messages. This concept has been known for some time, but only recently were practical methods found to implement arithmetic coding on computers with fixed-sized registers.

The output from an arithmetic coding process is a single number less than 1 and greater than or equal to 0. This single number can be uniquely decoded to create the exact stream of symbols that went into its construction. To construct the output number, the symbols are assigned a set probabilities. The message "BILL GATES," for example, would have a probability distribution like this:

Character	Probability
SPACE	1/10
A	1/10
B	1/10
E	1/10
G	1/10
I	1/10
L	2/10
S	1/10
T	1/10

Once character probabilities are known, individual symbols need to be assigned a range along a "probability line," nominally 0 to 1. It doesn't matter which characters are assigned which segment of the range, as long as it is done in the same manner by both the encoder and the decoder. The nine-character symbol set used here would look like the following:

Character	Probability	Range
SPACE	1/10	0.00 >= r > 0.10
A	1/10	0.10 >= r > 0.20
B	1/10	0.20 >= r > 0.30
E	1/10	0.30 >= r > 0.40
G	1/10	0.40 >= r > 0.50
I	1/10	0.50 >= r > 0.60
L	2/10	0.60 >= r > 0.80
S	1/10	0.80 >= r > 0.90
T	1/10	0.90 >= r > 1.00

Each character is assigned the portion of the 0 to 1 range that corresponds to its probability of appearance. Note that the character "owns" everything up to, but not including, the higher number. So the letter T in fact has the range .90 to .9999. . .

The most significant portion of an arithmetic-coded message belongs to the first symbol—or B, in the message "BILL GATES." To decode the first character properly, the final coded message has to be a number greater than or equal to .20 and less than .30. To encode this number, track the range it could fall in. After the first character is encoded, the low end for this range is .20 and the high end is .30.

During the rest of the encoding process, each new symbol will further restrict the possible range of the output number. The next character to be encoded, the letter I, owns the range .50 to .60 in the new subrange of .2 to .3. So the new encoded number will fall somewhere in the 50th to 60th percentile of the currently established range. Applying this logic will further restrict our number to .25 to .26. The algorithm to accomplish this for a message of any length is shown here:

```
low = 0.0;
high = 1.0;
```

```
while ( ( c = getc( input ) ) != EOF ) {
      range = high - low;
      high = low + range * high_range( c );
      low = low + range * low_range( c );
}
output( low );
```

Following this process to its natural conclusion with our message results in the following table:

New Character	Low value	High Value
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

So the final low value, .2572167752, will uniquely encode the message "BILL GATES" using our present coding scheme.

Given this encoding scheme, it is relatively easy to see how the decoding process operates. Find the first symbol in the message by seeing which symbol owns the space our encoded message falls in. Since .2572167752 falls between .2 and .3, the first character must be B. Then remove B from the encoded number. Since we know the low and high ranges of B, remove their effects by reversing the process that put them in. First, subtract the low value of B, giving .0572167752. Then divide by the width

of the range of B, or .1. This gives a value of .572167752. Then calculate where that lands, which is in the range of the next letter, I. The algorithm for decoding the incoming number is shown next:

```
number = input_code();
for ( ; ; ) {
     symbol = find_symbol_straddling_this_range( number );
     putc( symbol );
     range = high_range( symbol ) - low_range( symbol );
     number = number - low_range( symbol );
     number = number / range;
}
```

I have conveniently ignored the problem of how to decide when there are no more symbols left to decode. This can be handled either by encoding a special end-of-file symbol or by carrying the stream length with the encoded message. In the example in this chapter, as elsewhere in the book, I carry along a special end-of-stream symbol that tells the decoder when it is out of symbols. The decoding algorithm for the "BILL GATES" message will proceed as shown:

Encoded Number	Output Symbol	Low	High	Range
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1
0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	SPACE	0.0	0.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.2	0.3	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.3	0.4	0.1
0.8	S	0.8	0.9	0.1
0.0				

In summary, the encoding process is simply one of narrowing the range of possible numbers with every new symbol. The new range is proportional to the predefined probability attached to that symbol. Decoding is the inverse procedure, in which the range is expanded in proportion to the probability of each symbol as it is extracted.

Practical Matters. Encoding and decoding a stream of symbols using arithmetic coding is not too complicated. But at first glance it seems completely impractical. Most computers support floating-point numbers of around 80 bits. So do you have to start over every time you encode ten or fifteen symbols? Do you need a floating-point processor? Can machines with different floating-point formats communicate through arithmetic coding?

As it turns out, arithmetic coding is best accomplished using standard 16-bit and 32-bit integer math. Floating-point math is neither required nor helpful. What is required is an incremental transmission scheme in which fixed-size integer state variables receive new bits at the low end and shift them out at the high end, forming a single number that can be as long as the number of bits on the computer.

Earlier, we saw that the algorithm works by keeping track of a high and low number that brackets the range of the possible output number. When the algorithm first starts, the low number is set to .0 and the high number is set to 1. The first simplification made to work with integer math is to change the 1 to .999 . . . , or .111 . . . in binary. Mathematicians agree that .111 . . . binary is exactly the same as 1 binary, and we take their assurance at face value. It simplifies encoding and decoding.

To store these numbers in integer registers, first justify them so the implied decimal point is on the left side of the word. Then load as much of the initial high and low values as will fit into the word size we are working with. My implementation uses 16-bit unsigned math, so the initial value of high is 0xFFFF, and low is 0. We know that the high value continues with Fs forever, and the low continues with zeros forever—so we can shift those extra bits in with impunity when needed.

If you imagine our "BILL GATES" example in a five-decimal digit register (I use decimal digits in this example for clarity), the decimal equivalent of our setup would look like what follows on the next page.

```
HIGH: 99999 implied digits => 999999999...
LOW: 00000 implied digits => 000000000...
```

To find the new range of numbers, apply the encoding algorithm from the previous section. First, calculate the range between the low and high values. The difference between the two registers will be 100000, not 99999. This is because we assume the high register has an infinite number of 9s added to it, so we need to increment the calculated difference. We then compute the new high value using the formula from the previous section:

```
high = low + high_range(symbol).
```

In this case, the high range was .30, which gives a new value for high of 30000. Before storing the new value of high, we need to decrement it, once again because of the implied digits appended to the integer value. So the new value of high is 29999. The calculation of low follows the same procedure, with a resulting new value of 20000. So now high and low look like this:

```
high: 29999 (999...);
low: 20000 (000...).
```

At this point, the most significant digits of high and low match. Due to the nature of our algorithm, high and low can continue to grow closer together without quite ever matching. So once they match in the most significant digit, that digit will never change. We can now output that digit as the first digit of our encoded number. This is done by shifting both high and low left by one digit and shifting in a 9 in the least significant digit of high. The equivalent operations are performed in binary in the C implementation of this algorithm.

As this process continues, high and low continually grow closer together, shifting digits out into the coded word. The process for our "BILL GATES" message is shown next.

	HIGH	LOW	RANGE	CUMULATIVE OUTPUT
Initial state	99999	00000	100000	
Encode B (0.2-0.3)	29999	20000		
Shift out 2	99999	00000	10000	.2
Encode I (0.5-0.6)	59999	50000		.2
Shift out 5	99999	00000	100000	.25
Encode L (0.6-0.8)	79999	60000	20000	.25
Encode L (0.6-0.8)	75999	72000		.25
Shift out 7	59999	20000	40000	.257
Encode SPACE (0.0-0.1)	23999	20000		.257
Shift out 2	39999	00000	40000	.2572
Encode G (0.4-0.5)	19999	16000		.2572
Shift out 1	99999	60000	40000	.25721
Encode A (0.1-0.2)	67999	64000		.25721
Shift out 6	79999	40000	40000	.257216
Encode T (0.9-1.0)	79999	76000		.257216
Shift out 7	99999	60000	40000	.2572167
Encode E (0.3-0.4)	75999	72000		.2572167
Shift out 7	59999	20000	40000	.25721677
Encode S (0.8-0.9)	55999	52000		.25721677
Shift out 5	59999	20000		.257216775
Shift out 2				.2572167752
Shift out 0				.25721677520

After all the letters are accounted for, two extra digits need to be shifted out of either the high or low value to finish the output word. This is so the decoder can properly track the input data. Part of the information about the data stream is still in the high and low registers, and we need to shift that information to the file for the decoder to use later.

A Complication. This scheme works well for incrementally encoding a message. Enough accuracy is retained during the double-precision integer calculations to ensure that the message is accurately encoded. But there is potential for a loss of precision under certain circumstances.

If the encoded word has a string of 0s or 9s in it, the high and low values will slowly converge on a value, but they may not see their most significant digits match immediately. High may be 700004, and low may be 699995. At this point, the calculated range will be only a single digit long, which means the output word will not have enough precision to be accurately encoded. Worse, after a few more iterations, high could be 70000, and low could look be 69999.

At this point, the values are permanently stuck. The range between high and low has become so small that any iteration through another symbol will leave high and low at their same values. But since the most significant digits of both words are not equal, the algorithm can't output the digit and shift. It seems to have reached an impasse.

Defeat this underflow problem by preventing things from ever getting that bad. The original algorithm said something like, "If the most significant digit of high and low match, shift it out." If the two digits don't match, but are now on adjacent numbers, a second test needs to be applied. If high and low are one apart, we test to see if the second most significant digit in high is a 0 and if the second digit in low is a 9. If so, it means we are on the road to underflow and need to take action.

Head off underflow with a slightly different shift operation. Instead of shifting the most significant digit out of the word, delete the second digits from high and low and shift the rest of the digits left to fill the space. The most significant digit stays in place. Then set an underflow counter to remember that we threw away a digit and aren't quite sure whether it was going to be a 0 or a 9. This process is shown here:

	Before	After
High:	40344	43449
Low:	39810	38100
Underflow:	0	1

After every recalculation, check for underflow digits again if the most significant digits don't match. If underflow digits are present, we shift them out and increment the counter.

When the most significant digits do finally converge to a single value, output that value. Then output the underflow digits previously discarded. The underflow digits will all be 9s or 0s, depending on whether high and low converged on the higher or lower value. In the C implementation of this algorithm, the underflow counter keeps track of how many ones or zeros to output.

Decoding. In the "ideal" decoding process, we had the entire input number to work with, and the algorithm had to do things like "divide the encoded number by the symbol probability." In practice, we can't perform an operation like that on a number that could be billions of bytes long. As in the encoding process, however, the decoder can operate using 16- and 32-bit integers for calculations.

Instead of using just two numbers, high and low, the decoder has to use three numbers. The first two, high and low, correspond exactly to the high and low values maintained by the encoder. The third number, code, contains the current bits being read in from the input bit stream. The code value always falls between the high and low values. As they come closer and closer to it, new shift operations will take place, and high and low will move back away from code.

The high and low values in the decoder will be updated after every symbol, just as they were in the encoder, and they should have exactly the same values. By performing the same comparison test on the upper digit of high and low, the decoder knows when it is time to shift a new digit into the incoming code. The same underflow tests are performed as well.

In the ideal algorithm, it was possible to determine what the current encoded symbol was just by finding the symbol whose probabilities enclosed the present value of the code. In the integer math algorithm, things are somewhat more complicated. In this case, the probability scale is determined by the difference between high and low. So instead of the range being between .0 and 1.0, the range will be between two positive 16-bit integer counts. Where the present code value falls along that range determines current probability. Divide (value - low) by (high - low + 1) to get the actual probability for the present symbol.

Where's the Beef? It is not immediately obvious why this encoding process is an improvement over Huffman coding. It becomes clear when we examine a case in which the probabilities are a little different. If we have to encode the stream "AAAAAAA," and the probability of A is known to be .9, there is a 90 percent chance that any incoming character will be the letter A. We set up our probability table so that A occupies the .0 to .9 range, and the end-of-message symbol occupies the .9 to 1 range. The encoding process is shown next:

New Character	Low value	High Value
	0.0	1.0
A	0.0	0.9
A	0.0	0.81
A	0.0	0.729
A	0.0	0.6561
A	0.0	0.59049
A	0.0	0.531441
A	0.0	0.4782969
END	0.43046721	0.4782969

Now that we know the range of high and low values, all that remains is to pick a number to encode this message. The number .45 will make this message uniquely decode to "AAAAAAA." Those two decimal digits take slightly less than seven bits to specify, which means we have encoded eight symbols in less than eight bits! An optimal Huffman message would have taken a minimum of nine bits.

To take this point to an even further extreme, I set up a test that had only two symbols. In it, 0 had a 16382/16383 probability, and an end-of-file symbol had a 1/16383 probability. I then created a file filled with 100,000 0s. After compression using arithmetic coding, the output file was only three bytes long! The minimum size using Huffman coding would have been 12,501 bytes. This is obviously a contrived example, but it shows that arithmetic coding compresses data at rates much better than one bit per byte when the symbol probabilities are right.

The Code

The code supplied with this chapter in ARITH.C is a simple module that performs arithmetic compression and decompression using a simple order 0 model. It works exactly like the nonadaptive Huffman coding program in chapter 3. It first makes a single pass over the data, counting the symbols. The data is then scaled down to make the counts fit into single, unsigned characters. The scaled counts are saved to the output file for the decompressor to get at later, then the arithmetic coding table is built. Finally, the compressor passes through the data, compressing each symbol as it appears. When done, the end-of-stream character is sent out, the arithmetic coder is flushed, and the program exits.

The Compression Program. The compression portion of this program is shown shortly. The main module is called by the utility version of MAIN-E.C, which will have already taken care of opening files, parsing arguments, etc. Once we get to the compression phase of the program, things are ready to go.

The compressor code breaks down neatly into three sections. The first two lines initialize the model and the encoder. The while loop plus an additional line after it performs the compression, and the last three lines shut things down.

```
build_model( input, output->file );
initialize_arithmetic_encoder();

while ( ( c = getc( input ) ) != EOF ) {
   convert_int_to_symbol( c, &s );
   encode_symbol( output, &s );
}
convert_int_to_symbol( END_OF_STREAM, &s );
encode_symbol( output, &s );
flush_arithmetic_encoder( output );
OutputBits( output, 0L, 16 );
```

The build_model() routine has several responsibilities. It makes the first pass over the input data to count all the characters. It scales down the counts to fit in unsigned characters, then it takes those counts and builds the range table used by the coder. Finally, it writes the counts to the output file so the decompressor will have access to them later.

The initialize arithmetic encoder routine is fairly simple. It just sets up the high- and low-integer variables used during the encoding. The encoding loop calls two different routines to encode the symbol. The first, convert_int_to_symbol(), takes the character read in from the file and looks up the range for the given symbol. The range is then stored in the symbol object, which has the structure shown:

```
typedef struct {
   unsigned short int low_count;
   unsigned short int high_count;
   unsigned short int scale;
} SYMBOL;
```

These three values are all that are needed for the symbol to be encoded using the arithmetic encoder. The low-count and high-count values uniquely define where on the 0 to 1 range the symbol lies, and the scale value tells what the total span of the 0 to 1 scale is. If 1,000 characters had been counted in a text file, for example, the low_count and high_count for A might be 178 and 199, and the scale would be 1,000. This would indicate that on the 0 to 1 number scale, A would own the range .178 to .199.

Once the symbol object has been defined, it can be passed to the encoder. The arithmetic encoder needs only those three items to process the symbol and update the output number. It has the high- and low-range values and the underflow count stored internally as static variables, and it doesn't need anything else.

The way we detached the modeling data from the coding data gives us a convenient mechanism for experimenting with new ways of modeling. We just have to come up with the numbers that get plugged into the symbol. The encoder doesn't care how we got those numbers as long as they were derived consistently so we can decode the file later.

When we reach the end of the input file, we encode and send the end-of-stream symbol. The decompression program will use it to determine when it is done. To finish, call a routine to flush the arithmetic encoder, which takes care of any underflow bits. Finally, we have to output an extra sixteen bits. The reason for this is simple. When decoding symbols from the input bit stream, the effective bits are in the most significant bit position of the input registers. To get the bits there, we have to load other bits into the lower positions and shift them over. At least 15 insignificant bits are needed to decode the last symbol. Outputting 16 bits at the end of the program ensures that the decoder won't get a premature end of file while decoding the input file.

The Expansion Program. The main part of the expansion program follows the same pattern. First, the model is set up and the arithmetic coder is initialized. In this program, initializing the model means reading in the counts from the input file where the compressor put them. Initializing the arithmetic decoder means loading the low and high registers with 0000 and FFFF, then reading the first 16 bits from the input file into the current code.

```
input_counts( input->file );
initialize_arithmetic_decoder( input );
for ( ; ; ) {
    get_symbol_scale( &s );
    count = get_current_count( &s );
    c = convert_symbol_to_int( count, &s );
    if ( c == END_OF_STREAM )
        break;
    remove_symbol_from_stream( input, &s );
    putc( (char) c, output );
}
```

The decoding loop is a little more complicated in this routine to keep the modeling and decoding separate. First, get the scale for the current model to pass back to the arithmetic decoder. The decoder then converts its current input code into a count in the routine get_current_count. With the count, we can determine which symbol is the correct one to decode. This is done in the routine convert_symbol_to_int().

Though it seems strange, we don't remove the encoded symbol from the input bit stream till after we actually decode it. The process of removing it involves standard modifications of high and low and, if necessary, shifting in new bits from the input stream. Finally, the decoded character is written to the output file.

Initializing the Model. The model for a program using arithmetic coding needs to have three pieces of information for each symbol: the low end of its range, the high end of its range, and the scale of the entire alphabet's range (this is the same for all symbols in the alphabet). Since the top of a given symbol's range is the bottom of the next, we only need to keep track of N + 1 numbers for N symbols in the alphabet.

An example of how the range information would be created is shown for symbols A, B, C, D, and E. These symbols have the counts 5, 2, 3, 3, and 8 respectively. The numbers can be arranged in any order along the range, if done consistently.

Range	:	21
E	:	13
D	:	10
C	:	7
B	:	5
A	:	0

In this case, the alphabet has five symbols, and the number of counts to keep track of is six. The array is formed by creating the cumulative total at each symbol, starting at zero and working up to the range.

Given a structure like this, it is simple to derive the three items of information that define the symbol for the arithmetic encoder. For symbol x in the array, the low count can be found at totals[x], the high count at totals[x + 1], and the range of scale at totals[N], N being the number of symbols in the alphabet.

The routines that do compression create this array. In this program, the array is named totals[], and it has 258 elements. The number of symbols in the alphabet is 257, the normal 256 plus one for the end-of-stream symbol.

One additional constraint is placed on these counts. Two things determine the highest possible count: the number of bits available in the high and low registers and the number of bits in the code values. Since floating-point calculations are performed in fixed-length registers, we have to minimize the amount of precision needed in our calculations so errors do not occur.

As it happens, there are two restrictions on the number of bits used in arithmetic coding: (1) the bits in the frequency values must be at least two less than the number of bits in the high and low registers; and (2) the bits in the frequency values plus the bits used in either the high and low registers must not exceed the bits used in arithmetic calculations during the coding process.

During calculations on the arithmetic registers, use unsigned long values, which give 32 bits to work with. Since our high and low registers and our frequency counts are limited to 16 bits in unsigned short ints, we meet restriction 2 implicitly. Restriction 1, however, requires more modifications. Since we are only using 16-bit registers for our high and low values, we have to restrict the highest cumulative counts in the totals[] array to no more than 14 bits, or 16,384.

The code to make sure that our total count of symbols is less than 16,384 is in the build_model routine called on initialization of the model. It takes the process a step further, scaling the counts down so they all fit in a single byte. This is done so that the count data stored in the output file takes a minimal amount of space.

During the compression phase of the program, the build_model() routine is called to perform all the necessary chores associated with the order-0 modeling used in this program. The four lines of code from build_model() are shown here:

```
count_bytes( input, counts );
```

```
scale_counts( counts, scaled_counts );
output_counts( output, scaled_counts );
build_totals( scaled_counts );
```

build_model does no work on its own. It calls a series of four worker routines to handle the data for it. The first routine is count_bytes(). It does just that, counting all the occurrences of each symbol in the file and maintaining the total in an array, like so:

```
input_marker = ftell( input );
while ( ( c = getc( input )) != EOF )
  counts[ c ]++;
fseek( input, input_marker, SEEK_SET );
```

The code for count_bytes scans the entire input file, then returns the input pointer to where it was when called. We assume that the number of counts of a given symbol in the file cannot exceed the span of an unsigned long. If this is not true, other measures will need to be taken to avoid overflow of the counts[] array elements.

After the array has been counted, the counts have to be scaled down. This is done in the scale_counts() routine. The first step here is to scale down the unsigned long counts[] array to fit it in an array of unsigned characters. This stores the counts in less space in the output file. The code for this is listed here.

```
max_count = 0;
for ( i = 0 ; i < 256 ; i++ )
  if ( counts[ i ] > max_count )
    max_count = counts[ i ];
scale = max_count / 256;
scale = scale + 1;
for ( i = 0 ; i < 256 ; i++ ) {
    scaled_counts[ i ] = (unsigned char ) ( counts[ i ] /scale );
    if ( scaled_counts[ i ] == 0 && counts[ i ] != 0 )
      scaled_counts[ i ] = 1;
}
```

After this is complete, one more scaling may need to be done. As part of the limitations on performing arithmetic coding using fixed-length registers, we have to restrict the total of our counts to less than 16,384, or fourteen bits. The second part of scale_counts does this with brute force, checking the total, then dividing by either two or four to get the correct results. An additional count has to be added because of the single count used by the end-of-stream character.

```
total = 1;
for ( i = 0 ; i < 256 ; i++ )
  total += scaled_counts[ i ];
if ( total > ( 32767 - 256 ) )
  scale = 4;
else if ( total > 16383 )
  scale = 2;
else
  return;
for ( i = 0 ; i < 256 ; i++ )
  scaled_counts[ i ] /= scale;
```

There is certainly room in the scale_counts() routine for more sophistication. Every time we lose some of the accuracy in our counts by scaling, we also lose a certain amount of compression. An ideal scaling routine would scale down the counts to fit in the correct range while doing as little scaling as possible. The routines listed here don't work very hard at doing that.

Once the counts have been scaled down to fit in the unsigned character array, the output_counts() routine is called to save them to the output file. This program employs the same method to store the counts as used in chapter 3 for the Huffman coding example. Instead of storing the entire array of counts, we only store runs on nonzero values. For details on this, refer to chapter 3.

The last step in building the model is to set up the cumulative totals array in totals[]. This is the array used when actually performing the arithmetic coding. The code shown below builds that array. Remember that after the totals[] array has been built, the range for symbol x is found in totals[x] and totals[x + 1]. The range used for the entire alphabet is found in totals[END_OF_STREAM + 1].

```
totals[ 0 ] = 0;
for ( i = 0 ; i < END_OF_STREAM ; i++ )
  totals[ i + 1 ] = totals[ i ] + scaled_counts[ i ];
totals[ END_OF_STREAM + 1 ] = totals[ END_OF_STREAM ] + 1;
```

Reading the Model. For expansion, the code needs to build the same model array in totals[] that was used in the compression routine. Since the original file is not available to scan for counts, the program reads in the scaled_counts[] array stored in the compressed file. The code that accomplishes this is identical to the Huffman expansion code in chapter 3. Refer to chapter 3 for details on how this code works.

After the scaled_counts[] array has been read in, the same routine used by the compression code can be invoked to build the totals[] array. Calling build_totals() in both the compression and expansion routines helps ensure that we are working with the same array.

Initializing the Encoder. Before compression can begin, we have to initialize the variables that constitute the arithmetic encoder. Three 16-bit variables define the arithmetic encoder: low, high, and underflow_bits. When the encoder first starts to run, the range of the output floating-point number is anywhere between 0 and 1. The low variable is initialized to 0 and the high to 0xFFFF. These two variables have an implied decimal point just ahead of the most significant bit and an infinite trail of ones or zeros. The ones will be shifted into the high variable and the zeros into the low.

```
low = 0;
high = 0xffff;
underflow_bits = 0;
```

The Encoding Process. At this point, the program is ready to begin the actual encoding process. This consists of looping through the entire file, reading in a character, determining its range variables, then encoding it. After the file has been scanned, the final step is to encode the end-of-stream symbol.

```
while ( ( c = getc( input ) ) != EOF ) {
   convert_int_to_symbol( c, &s );
   encode_symbol( output, &s );
}
convert_int_to_symbol( END_OF_STREAM, &s );
encode_symbol( output, &s );
```

Two routines encode a symbol. convert_int_to_symbol() looks up the modeling information for the symbol and retrieves the numbers needed to perform the arithmetic coding. This consists of stuffing numbers into the three elements of the symbol's structure, as shown here:

```
s->scale = totals[ END_OF_STREAM + 1 ];
s->low_count = totals[ c ];
s->high_count = totals[ c + 1 ];
```

After the symbol information is present, we are ready to encode the symbol using the arithmetic encoding routine. The C code to do this, listed in encode_symbol(), has two distinct steps. The first is to adjust the high and low variables based on the symbol data passed to the encoder.

```
range = (long) ( high-low ) + 1;
high = low + (unsigned short int)
       (( range * s->high_count ) / s->scale - 1 );
low = low + (unsigned short int)
       (( range * s->low_count ) / s->scale );
```

The code shown below restricts the range of high and low by the amount indicated in the symbol information. The range of the symbol is defined by s->low_count and s->high_count. These two counts are measured relative to the s->scale variable. After

the adjustments are made, low and high will both be greater than or equal to their previous values. The range, or the distance between the two, will be less than or equal to its previous value.

```
for ( ; ; ) {
   if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
      OutputBit( stream, high & 0x8000 );
      while ( underflow_bits > 0 ) {
         OutputBit( stream, ~high & 0x8000 );
         underflow_bits-;
      }
   } else if ( ( low & 0x4000 ) && !( high & 0x4000 )) {
      underflow_bits += 1;
      low &= 0x3fff;
      high |= 0x4000;
   } else
      return ;
   low <<= 1;
   high <<= 1;
   high |= 1;
}
```

After high and low have been adjusted, the routine needs to shift out any bits available for shifting. After a given arithmetic adjustment, it is never certain how many bits will need to be shifted out. If the encoded symbol has a very high probability, the number of bits will be low. If the encoded symbol has a low probability, the number of bits may be high.

Since the number isn't known in advance, the encoding routine sits in a loop shifting bits until there are no more shifts possible. The routine tests for two conditions to see if shifting is necessary. The first occurs when the most significant bits of the low and high word are the same. Because of the math being used, once the two bits are identical, they will never change. When this occurs, the bit is sent to the output file, and the high and low values are shifted.

Before shifting out the bit found when the most MSBs match, however, we have to transmit any underflow bits previously saved up. The underflow bits will be a sequence of bits set to the opposite value of the MSB. When we have an underflow, we have a binary sequence that ends up looking like that shown above. The number of underflow bits is found in the underflow_bits variable.

```
high = .100000...
low  = .011111...
```

Which leads to the second condition under which high and low variables require shifting: underflow. This occurs when the high and low words are growing dangerously close together but have not yet had their most significant bits match, a situation similar to that shown above.

When words begin growing close together like this, the dynamic range becomes dangerously low. Test for this condition after determining that the two most significant bits don't match. If they don't, check to see if the next bit is 0 in the high word and 1 in the low word. If they are, perform an underflow shift.

The underflow shift operation consists of throwing away the underflow bit (the one next to the most significant digit), shifting the remaining bits over one by one to fill the gap, and incrementing the underflow counter. The code to do this is somewhat opaque, but it performs this operation.

The underflow code takes advantage of the fact that when in danger of underflow, we know two things. First, we know that the most significant bit in the high word is 1 and in the low word 0. Second, the bit we throw away from the high word is 0 and from the low word 1.

Since we know the value of the highest two bits, we can simplify the shift operation. The code used in this chapter toggles the second most significant bit in both the high and low registers, then performs the normal shift operation. It looks as though the lower 14 bits were shifted left and the MSB was left alone.

If we check for both possible shift conditions and don't flag either one, we are done shifting bits out and can end the encoding operation. If either of the tests passed, the actual shift operation can take place. Both the high and low words are shifted left one bit position. The high word has a 1 shifted in to the LSB, and the low word has a 0 shifted in. The loop then repeats, outputting and shifting additional bits as necessary.

Flushing the Encoder. After encoding, it is necessary to flush the arithmetic encoder. The code for this is in the flush_arithmetic_encoder() routine. It outputs two bits and any additional underflow bits added along the way.

The Decoding Process. Before arithmetic decoding can start, we need to initialize the arithmetic decoder variables. While encoding, we had just a high and low variable. Both are maintained by the decoder with a code variable, which contains the current bit stream read in from the input file.

During arithmetic decoding, the high and low variables should track exactly the same values they had during the encoding process, and the code variable should reflect the bit stream as it is read in from the input file. The only exception to this is that the code variable has underflow bits taken from it and thrown away, as with the high and low variables.

```
code = 0;
for ( i = 0 ; i < 16 ; i++ ) {
  code <<= 1;
  code += InputBit( stream );
}
low = 0;
high = 0xffff;
```

This implementation of the arithmetic decoding process requires four separate steps to decode each character. The first is to get the current scale for the symbol. This is simply a matter of looking in the current model data. In this implementation, the scale is found at totals[END_OF_STREAM + 1]. The reason for breaking this out as a

separate routine rather than coding it in-line is that future expansions to the basic compression program may use different modeling techniques. If a different model is used, determining the scale of the model could end up being more complex. This happens in the program used in the next chapter.

Once the current scale for the model is known, a second call is made to get the count for the current arithmetic code. This involves translating the decoders range, expressed by the high and low variables, into the range used by the model, which is in the scale variable.

```
range = (long) ( high - low ) + 1;
count = (short int)
    ((((long) ( code - low ) + 1 ) * s->scale-1 ) / range );
return( count );
```

The count returned to the expansion program is in essence a simple translation of the code variable from one scale to another. Once the count has been determined, we can determine what symbol has been encoded. Since we know the low and high range of the count for every symbol in the alphabet, determining which symbol goes with which count is just a matter of looking through the counts listed in the totals[] array.

```
for ( c = END_OF_STREAM ; count < totals[ c ] ; c-- )
  ;
s->high_count = totals[ c + 1 ];
s->low_count = totals[ c ];
return( c );
```

The implementation of the convert_symbol_to_int() used here determines the correct symbol with brute force. It simply starts looking at the top of the totals[] array for a count that fits with the current symbol, and it works down till if finds one that does. This is not optimal, since it could take 257 comparisons to locate the correct symbol.

An improved method of decoding would keep the symbols sorted so that the most frequently used symbols would be checked first. This would reduce the average time required to locate a symbol, though with random data we would not see much improvement. For simplicity, this was not the method used here.

Once convert_symbol_to_int() locates the correct symbol in the totals[] array, it takes the high and low counts and stores them in the symbol variable. They will be used in the next step of the decoding process. The correct value of the symbol is then returned to the calling program as an int.

After the correct symbol value is set up in the symbol structure, remove_symbol_from_stream() is called. Arithmetic coding is unusual in that it determines what the symbol is *before* it removes the bits associated with it. Then it calls the routine that actually removes those bits from the code and sets up the input code for the next symbol.

```
range = (long)( high - low ) + 1;
high = low + (unsigned short int)
       (( range * s->high_count ) / s->scale - 1 );
low = low + (unsigned short int)
       (( range * s->low_count ) / s->scale );
for ( ; ; ) {
   if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
   } else if ((low & 0x4000) == 0x4000 && (high & 0x4000) == 0 ) {
      code ^= 0x4000;
      low &= 0x3fff;
      high |= 0x4000;
   } else
      return;
   low <<= 1;
   high <<= 1;
   high |= 1;
   code <<= 1;
   code += InputBit( stream );
}
```

The code that removes the symbol from the stream is listed above. It operates almost as a mirror image of the encoding routine. It first rescales the high and low variables to their new ranges as dictated by the range of the symbol being removed. Then the shifting process begins.

As before, there are two possible reasons to shift in new bits. First, if high and low have the same most significant bit, they will be discarded and a new bit will be shifted in as a replacement. Second, if high and low don't have the same MSB, and the second most significant bits are threatening underflow, we will discard the second most significant bits and shift in new bits.

If neither of the possible shift criteria are met, we can return, since the effects of the symbol have been entirely removed from the input stream. Otherwise, we shift high, low, and code. The lowest bit of high gets a 1, the lowest bit of low gets a 0, and the lowest bit of code gets a new bit from the input stream. This process continues indefinitely until all shifting is complete, at which point we return to the calling routine.

Summary

Arithmetic coding seems more complicated than Huffman coding, but the size of the program required to implement it is not significantly different. Runtime performance *is* significantly slower than Huffman coding, however, due to the computational burden imposed on the encoder and decoder. If squeezing the last bit of performance out of the coder is important, arithmetic coding will always provide as good or better performance than Huffman coding. But careful optimization is needed to get performance up to acceptable levels.

Code

```
/************************** Start of ARITH.C ************************/
#include <stdio.h>
#include <stdlib.h>
#include "errhand.h"
#include "bitio.h"

/*
* The SYMBOL structure is what is used to define a symbol in
* arithmetic coding terms. A symbol is defined as a range between
* 0 and 1. Since we are using integer math, instead of using 0 and 1
* as our end points, we have an integer scale. The low_count and
* high_count define where the symbol falls in the range.
```

```
*/

typedef struct {
   unsigned short int low_count;
   unsigned short int high_count;
   unsigned short int scale;
} SYMBOL;

/*
* Internal function prototypes, with or without ANSI prototypes.
*/

#ifdef __STDC__

void build_model( FILE *input, FILE *output );
void scale_counts( unsigned long counts[],
                   unsigned char scaled_counts[] );
void build_totals( unsigned char scaled_counts[] );
void count_bytes( FILE *input, unsigned long counts[] );
void output_counts( FILE *output, unsigned char scaled_counts[] );
void input_counts( FILE *stream );
void convert_int_to_symbol( int symbol, SYMBOL *s );
void get_symbol_scale( SYMBOL *s );
int convert_symbol_to_int( int count, SYMBOL *s );
void initialize_arithmetic_encoder( void );
void encode_symbol( BIT_FILE *stream, SYMBOL *s );
void flush_arithmetic_encoder( BIT_FILE *stream );
short int get_current_count( SYMBOL *s );
void initialize_arithmetic_decoder( BIT_FILE *stream );
void remove_symbol_from_stream( BIT_FILE *stream, SYMBOL  *s);

#else

void build_model();
void scale_counts();
void build_totals();
void count_bytes();
```

```
void output_counts();
void input_counts();
void convert_int_to_symbol();
void get_symbol_scale();
int convert_symbol_to_int();
void initialize_arithmetic_encoder();
void encode_symbol();
void flush_arithmetic_encoder();
short int get_current_count();
void initialize_arithmetic_decoder();
void remove_symbol_from_stream();

#endif

#define END_OF_STREAM 256
short int totals[ 258 ];        /* The cumulative totals */

char *CompressionName   = "Adaptive order 0 model with arithmetic coding";
char *Usage             = "in-file out-file\n\n";

/*
* This compress file routine is a fairly orthodox compress routine.
* It first gathers statistics, and initializes the arithmetic
* encoder. It then encodes all the characters in the file, followed
* by the EOF character. The output stream is then flushed, and we
* exit. Note that an extra two bytes are output. When decoding an arith-
* metic stream, we have to read in extra bits. The decoding process
* takes place in the msb of the low and high range ints, so when we are
* decoding our last bit we will still have to have at least 15 junk
* bits loaded into the registers. The extra two bytes account for
* that.
*/
void CompressFile( input, output, argc, argv )
FILE   *input;
BIT_FILE   *output;
int argc;
char   *argv[];
```

```
{
    int c;
    SYMBOL s;

    build_model( input, output->file );
    initialize_arithmetic_encoder();

    while ( ( c = getc( input ) ) != EOF ) {
        convert_int_to_symbol( c, &s );
        encode_symbol( output, &s );
    }
    convert_int_to_symbol( END_OF_STREAM, &s );
    encode_symbol( output, &s );
    flush_arithmetic_encoder( output );
    OutputBits( output, 0L, 16 );
    while ( argc-- > 0 ) {
        printf( "Unused argument: %s\n", *argv );
        argv++;
    }
}

/*
* This expand routine is also very conventional. It reads in the
* model, initializes the decoder, then sits in a loop reading in
* characters. When we decode an END_OF_STREAM, it means we can close
* up the files and exit. Note decoding a single character is a three
* step process: first we determine what the scale is for the current
* symbol by looking at the difference between the high an low values.
* We then see where the current input values fall in that range.
* Finally, we look in our totals array to find out what symbol is
* a match. After that is done, we still have to remove that symbol
* from the decoder. Lots of work.
*/

void ExpandFile( input, output, argc, argv )
BIT_FILE *input;
FILE *output;
```

```
int argc;
char *argv[];
{
   SYMBOL s;
   int c;
   int count;

   input_counts( input->file );
   initialize_arithmetic_decoder( input );
   for ( ; ; ) {
      get_symbol_scale( &s );
      count = get_current_count( &s );
      c = convert_symbol_to_int( count, &s );
      if ( c == END_OF_STREAM )
         break;
      remove_symbol_from_stream( input, &s );
      putc( (char) c, output );
   }
   while ( argc-- > 0 ) {
      printf( "Unused argument: %s\n", *argv );
      argv++;
   }
}

/*
* This is the routine that is called to scan the input file, scale
* the counts, build the totals array, the output the scaled counts
* to the output file.
*/

void build_model( input, output )
FILE *input;
FILE *output;
{
   unsigned long counts[ 256 ];
   unsigned char scaled_counts[ 256 ];
```

```
    count_bytes( input, counts );
    scale_counts( counts, scaled_counts );
    output_counts( output, scaled_counts );
    build_totals( scaled_counts );
}

/*
* This routine runs through the file and counts the appearances of
* each character.
*/
#ifndef SEEK_SET
#define SEEK_SET 0
#endif

void count_bytes( input, counts )
FILE *input;
unsigned long counts[];
{
    long input_marker;
    int i;
    int c;

    for ( i = 0 ; i < 256 ; i++ )
        counts[ i ] = 0;
    input_marker = ftell( input );
    while ( ( c = getc( input )) != EOF )
        counts[ c ]++;
    fseek( input, input_marker, SEEK_SET );
}

/*
* This routine is called to scale the counts down. There are two
* types of scaling that must be done. First, the counts need to be
* scaled down so that the individual counts fit into a single unsigned
* char. Then, the counts need to be rescaled so that the total of all
* counts is less than 16384.
*/
```

```
void scale_counts( counts, scaled_counts )
unsigned long counts[];
unsigned char scaled_counts[];
{
   int i;
   unsigned long max_count;
   unsigned int total;
   unsigned long scale;

/*
* The first section of code makes sure each count fits into a single
* byte.
*/
   max_count = 0;
   for ( i = 0 ; i < 256 ; i++ )
      if ( counts[ i ] > max_count )
         max_count = counts[ i ];
   scale = max_count / 256;
   scale = scale + 1;
   for ( i = 0 ; i < 256 ; i++ ) {
      scaled_counts[ i ] = (unsigned char ) ( counts[ i ] / scale );
      if ( scaled_counts[ i ] == 0 && counts[ i ] != 0 )
         scaled_counts[ i ] = 1;
   }
/*
* This next section makes sure the total is less than 16384.
* I initialize the total to 1 instead of 0 because there will be an
* additional 1 added in for the END_OF_STREAM symbol;
*/
   total = 1;
   for ( i = 0 ; i < 256 ; i++ )
      total += scaled_counts[ i ];
   if ( total > ( 32767 - 256 ) )
      scale = 4;
   else if ( total > 16383 )
      scale = 2;
   else
```

```
    return;
  for ( i = 0 ; i < 256 ; i++ )
    scaled_counts[ i ] /= scale;
}

/*
* This routine is used by both the encoder and decoder to build the
* table of cumulative totals. The counts for the characters in the
* file are in the counts array, and we know that there will be a
* single instance of the EOF symbol.
*/
void build_totals( scaled_counts )
unsigned char scaled_counts[];
{
  int i;

  totals[ 0 ] = 0;
  for ( i = 0 ; i < END_OF_STREAM ; i++ )
    totals[ i + 1 ] = totals[ i ] + scaled_counts[ i ];
  totals[ END_OF_STREAM + 1 ] = totals[ END_OF_STREAM ] + 1;
}

/*
* In order for the compressor to build the same model, I have to
* store the symbol counts in the compressed file so the expander can
* read them in. In order to save space, I don't save all 256 symbols
* unconditionally. The format used to store counts looks like this:
*
* start, stop, counts, start, stop, counts, ... 0
*
* This means that I store runs of counts, until all the non-zero
* counts have been stored. At this time the list is terminated by
* storing a start value of 0. Note that at least 1 run of counts has
* to be stored, so even if the first start value is 0, I read it in.
* It also means that even in an empty file that has no counts, I have
* to pass at least one count.
*
```

```
* In order to efficiently use this format, I have to identify runs of
* non-zero counts. Because of the format used, I don't want to stop a
* run because of just one or two zeros in the count stream. So I have
* to sit in a loop looking for strings of three or more zero values
* in a row.
*
* This is simple in concept, but it ends up being one of the most
* complicated routines in the whole program. A routine that just
* writes out 256 values without attempting to optimize would be much
* simpler, but would hurt compression quite a bit on small files.
*
*/
void output_counts( output, scaled_counts )
FILE *output;
unsigned char scaled_counts[];
{
   int first;
   int last;
   int next;
   int i;

   first = 0;
   while ( first < 255 && scaled_counts[ first ] == 0 )
            first++;
/*
* Each time I hit the start of the loop, I assume that first is the
* number for a run of non-zero values. The rest of the loop is
* concerned with finding the value for last, which is the end of the
* run, and the value of next, which is the start of the next run.
* At the end of the loop, I assign next to first, so it starts in on
* the next run.
*/
   for ( ; first < 256 ; first = next ) {
      last = first + 1;
      for ( ; ; ) {
         for ( ; last < 256 ; last++ )
            if ( scaled_counts[ last ] == 0 )
```

```
            break;
        last--;
        for ( next = last + 1; next < 256 ; next++ )
          if ( scaled_counts[ next ] != 0 )
            break;
        if ( next > 255 )
          break;
        if ( ( next - last ) > 3 )
          break;
        last = next;
      };
/*
* Here is where I output first, last, and all the counts in between.
*/
      if ( putc( first, output ) != first )
        fatal_error( "Error writing byte counts\n" );
      if ( putc( last, output ) != last )
        fatal_error( "Error writing byte counts\n" );
      for ( i = first ; i <= last ; i++ ) {
        if ( putc( scaled_counts[ i ], output ) !=
          (int) scaled_counts[ i ] )
          fatal_error( "Error writing byte counts\n" );
      }
    }
    if ( putc( 0, output ) != 0 )
        fatal_error( "Error writing byte counts\n" );
}

/*
* When expanding, I have to read in the same set of counts. This is
* quite a bit easier that the process of writing them out, since no
* decision making needs to be done. All I do is read in first, check
* to see if I am all done, and if not, read in last and a string of
* counts.
*/

void input_counts( input )
```

```
FILE *input;
{
    int first;
    int last;
    int i;
    int c;
    unsigned char scaled_counts[ 256 ];

    for ( i = 0 ; i < 256 ; i++ )
        scaled_counts[ i ] = 0;
    if ( ( first = getc( input ) ) == EOF )
        fatal_error( "Error reading byte counts\n" );
    if ( ( last = getc( input ) ) == EOF )
        fatal_error( "Error reading byte counts\n" );
    for ( ; ; ) {
        for ( i = first ; i <= last ; i++ )
            if ( ( c = getc( input ) ) == EOF )
                fatal_error( "Error reading byte counts\n" );
            else
                scaled_counts[ i ] = (unsigned int) c;
        if ( ( first = getc( input ) ) == EOF )
            fatal_error( "Error reading byte counts\n" );
        if ( first == 0 )
            break;
        if ( ( last = getc( input ) ) == EOF )
            fatal_error( "Error reading byte counts\n" );
    }
    build_totals( scaled_counts );
}

/*
* Everything from here down define the arithmetic coder section
* of the program.
*/

/*
* These four variables define the current state of the arithmetic
```

```
* coder/decoder. They are assumed to be 16 bits long. Note that
* by declaring them as short ints, they will actually be 16 bits
* on most 80X86 and 680X0 machines, as well as VAXen.
*/
static unsigned short int code;/* The present input code value       */
static unsigned short int low; /* Start of the current code range    */
static unsigned short int high;/* End of the current code range      */
long underflow_bits;           /* Number of underflow bits pending   */

/*
* This routine must be called to initialize the encoding process.
* The high register is initialized to all 1s, and it is assumed that
* it has an infinite string of 1s to be shifted into the lower bit
* positions when needed.
*/
void initialize_arithmetic_encoder() {
   low = 0;
   high = 0xffff;
   underflow_bits = 0;
}

/*
* At the end of the encoding process, there are still significant
* bits left in the high and low registers. We output two bits,
* plus as many underflow bits as are necessary.
*/
void flush_arithmetic_encoder( stream )
BIT_FILE *stream;
{
   OutputBit( stream, low & 0x4000 );
   underflow_bits++;
   while ( underflow_bits-- > 0 )
      OutputBit( stream, ~low & 0x4000 );
}
```

```
/*
* Finding the low count, high count, and scale for a symbol
* is really easy, because of the way the totals are stored.
* This is the one redeeming feature of the data structure used
* in this implementation.
*/
void convert_int_to_symbol( c, s )
int c;
SYMBOL *s;
{
   s->scale = totals[ END_OF_STREAM + 1 ];
   s->low_count = totals[ c ];
   s->high_count = totals[ c + 1 ];
}

/*
* Getting the scale for the current context is easy.
*/
void get_symbol_scale( s )
SYMBOL *s;
{
   s->scale = totals[ END_OF_STREAM + 1 ];
}

/*
* During decompression, we have to search through the table until
* we find the symbol that straddles the "count" parameter. When
* it is found, it is returned. The reason for also setting the
* high count and low count is so that symbol can be properly removed
* from the arithmetic coded input.
*/
int convert_symbol_to_int( count, s)
int count;
SYMBOL *s;
{
   int c;
```

```
    for ( c = END_OF_STREAM ; count < totals[ c ] ; c-- )
      ;
    s->high_count = totals[ c + 1 ];
    s->low_count = totals[ c ];
    return( c );
}

/*
* This routine is called to encode a symbol. The symbol is passed
* in the SYMBOL structure as a low count, a high count, and a range,
* instead of the more conventional probability ranges. The encoding
* process takes two steps. First, the values of high and low are
* updated to take into account the range restriction created by the
* new symbol. Then, as many bits as possible are shifted out to
* the output stream. Finally, high and low are stable again and
* the routine returns.
*/
void encode_symbol( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
    long range;
/*
* These three lines rescale high and low for the new symbol.
*/
    range = (long) ( high-low ) + 1;
    high = low + (unsigned short int)
                 (( range * s->high_count ) / s->scale - 1 );
    low = low + (unsigned short int)
                 (( range * s->low_count ) / s->scale );
/*
* This loop turns out new bits until high and low are far enough
* apart to have stabilized.
*/
    for ( ; ; ) {
```

```
/*
* If this test passes, it means that the MSDigits match, and can
* be sent to the output stream.
*/
     if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
       OutputBit( stream, high & 0x8000 );
       while ( underflow_bits > 0 ) {
         OutputBit( stream, ~high & 0x8000 );
         underflow_bits-;
       }
     }
/*
* If this test passes, the numbers are in danger of underflow, because
* the MSDigits don't match, and the 2nd digits are just one apart.
*/
     else if ( ( low & 0x4000 ) && !( high & 0x4000 )) {
       underflow_bits += 1;
       low &= 0x3fff;
       high |= 0x4000;
     } else
       return ;
     low <<= 1;
     high <<= 1;
     high |= 1;
   }
}

/*
* When decoding, this routine is called to figure out which symbol
* is presently waiting to be decoded. This routine expects to get
* the current model scale in the s->scale parameter, and it returns
* a count that corresponds to the present floating point code:
*
* code = count / s->scale
*/
short int get_current_count( s )
```

```
SYMBOL *s;
{
   long range;
   short int count;

   range = (long) ( high - low ) + 1;
   count = (short int)
           ((((long) ( code - low ) + 1 ) * s->scale-1 ) / range );
   return( count );
}

/*
* This routine is called to initialize the state of the arithmetic
* decoder. This involves initializing the high and low registers
* to their conventional starting values, plus reading the first
* 16 bits from the input stream into the code value.
*/
void initialize_arithmetic_decoder( stream )
BIT_FILE *stream;
{
   int i;

   code = 0;
   for ( i = 0 ; i < 16 ; i++ ) {
      code <<= 1;
      code += InputBit( stream );
   }
   low = 0;
   high = 0xffff;
}

/*
* Just figuring out what the present symbol is doesn't remove
* it from the input bit stream. After the character has been
* decoded, this routine has to be called to remove it from the
* input stream.
*/
```

```
void remove_symbol_from_stream( stream, s )
BIT_FILE *stream;
SYMBOL *s;
{
  long range;

/*
* First, the range is expanded to account for the symbol removal.
*/
  range = (long)( high - low ) + 1;
  high = low + (unsigned short int)
         (( range * s->high_count ) / s->scale - 1 );
  low = low + (unsigned short int)
         (( range * s->low_count ) / s->scale );
/*
* Next, any possible bits are shipped out.
*/
  for ( ; ; ) {
/*
* If the MSDigits match, the bits will be shifted out.
*/
    if ( ( high & 0x8000 ) == ( low & 0x8000 ) ) {
    }
/*
* Else, if underflow is threatening, shift out the 2nd MSDigit.
*/
    else if ((low & 0x4000) == 0x4000  && (high & 0x4000) == 0 ) {
      code ^= 0x4000;
      low &= 0x3fff;
      high |= 0x4000;
    } else
/*
* Otherwise, nothing can be shifted out, so I return.
*/
```

```
            return;
        low <<= 1;
        high <<= 1;
        high |= 1;
        code <<= 1;
        code += InputBit( stream );
    }
}

/*************************** End of ARITH.C ***************************/
```