Data Compression With Arithmetic Coding

Oct 19, 2014

Arithmetic coding is a common algorithm used in both lossless and lossy data compression algorithms.

It is an *entropy encoding* technique, in which the frequently seen symbols are encoded with fewer bits than rarely seen symbols. It has some advantages over well-known techniques such as Huffman coding. This article will describe the CACM87 implementation of arithmetic coding in detail, giving you a good understanding of all the details needed to implement it.

On a historical note, this post is an update of anData Compression With Arithmetic Coding article I wrote over twenty years ago. That article was published in the print edition of Dr. Dobb's Journal, which meant that a lot of editing was done in order to avoid excessive page count. In particular, that Dr. Dobb's piece combined two topics: a description of arithmetic coding along with a discussion of compression using PPM (Prediction by Partial Matching).

Because this new piece will be published on the web, space considerations are no longer a big factor, which I hope will allow me to do justice to the details of arithmetic coding. PPM, a worthy topic of its own, will be discussed in a later article. I hope that this new effort will be, while annoyingly long, the thorough explanation of the subject I wanted to do in 1991.

I think the best way to understand arithmetic coding is to break it into two parts, and I'll use that idea in this article. First I will give a description of how arithmetic coding works, using regular floating point arithmetic implemented using standard C++ data types. This allows for a completely understandable, but slightly impractical implementation. In other words, it works, but it can only be used to encode very short messages.

The second section of the article will describe an implementation in which we switch to doing a special type of math on unbounded binary numbers. This is a somewhat mind-boggling topic in itself, so it helps if you already understand arithmetic coding - you don't have get hung up trying to learn two things at once.

To wrap up I will present working sample code written in modern C++. It won't necessarily be the most optimized code in the world, but it is portable and easy to add to your existing projects. It should be perfect for learning and experimenting with this coding technique.

Fundamentals

The first thing to understand about arithmetic coding is what it produces. Arithmetic coding takes a *message* (often a file) composed of *symbols* (nearly always eight-bit characters), and converts it to a floating point number greater than or equal to zero and less than one. This floating point number can be quite long - effectively your entire output file is one long number - which means it is not a normal data type that you are used to using in conventional programming languages. My implementation of the algorithm will have to create this floating point number from scratch, bit by bit, and likewise read it in and decode it bit by bit.

This encoding process is done incrementally. As each character in a file is encoded, a few bits will be added to the encoded message, so it is built up over time as the algorithm proceeds.

The second thing to understand about arithmetic coding is that it relies on a*model* to characterize the symbols it is processing. The job of the model is to tell the encoder what the probability of a character is in a given message. If the model gives an accurate probability of the characters in the message, they will be encoded very close to optimally. If the model misrepresents the probabilities of symbols, your encoder may actually expand a message instead of compressing it!

Encoding With Floating Point Math

The term *arithmetic coding* has to cover two separate processes: encoding messages and decoding them. I'll start by looking at the encoding process with sample C++ code that implements the algorithm in a very limited form using C++ `double` data. The code in this first section is only useful for exposition - don't try to do any real compression with it.

To perform arithmetic encoding, we first need to define a proper model. Remember that the function of the model is to provide

probabilities of a given character in a message. The conceptual idea of an arithmetic coding model is that each symbol will own its own unique segment of the number line of real numbers between 0 and 1. It's important to note that there are many different ways to model character probabilities. Some models are static, never changing. Others are updated after every character is processed. The only two things that matter to us are that 1) the model attempts to accurately predict the probability a character will appear, and 2) the encoder and decoder have identical models at all times.

As an example, we can start with an encoder that can only encode an alphabet of 100 different characters. In a simple static model we will start with capital letters, then the lower case letters. This means that the first symbol, 'A', will own the number line from 0 to .01, 'B' will own .01 to .02, and so on. (In all cases, this is strictly a half-closed interval, so the probability range for 'A' is actually >= 0 and < .01.)

With this model my encoder can represent the single letter 'B' by outputting a floating point number that is less than .02 and greater than or equal to .01. So for example, an arithmetic encoder that wanted to create that single letter could output .15 and be done.

An encoder that just outputs single characters is not much use though. To encode a string of symbols involves a slightly more complicated process. In this process, the first character defines a range of the number line that corresponds to the section assigned to it by the model. For the character 'B', that means the message is between .01 and .02.

The next character in the message then further divides that existing range proportionate to its current ownership of the number line. So some other letter that owns the very end of the number line, from .99 to 1.0 would change the range from [.01,.02) to [.0199, .020). This progressive subdividing of the range is just simple multiplication and addition, and is best understand with a simple code sample. My first pass in C++, which is far from a working encoder, might look like this:

```
double high = 1.0;
double low = 0.0;
char c;
while ( input >> c ) {
  std::pair<double, double> p = model.getProbability(c);
  double range = high - low;
  high = low + range * p.second;
  low = low + range * p.first;
}
output << low + (high-low)/2;
```

After the entire message has been processed, we have a final range, [low, high). The encoder outputs a floating point number right in the center of that range.

Examining the Floating Point Prototype

The first pass encoder is demonstrated in the attached project as `fp_proto.cpp`. To get it working I also needed to define a simple model. In this case I've created a model that can encode 100 characters, with each having a fixed probability of .01, starting with 'A' in the first position. To keep things simple I've only fleshed the class out enough to encode the capital letters from the ASCII character set:

```
struct {
  static std::pair<double,double> getProbability( char c )
  {
    if (c >= 'A' && c <= 'Z')
      return std::make_pair( (c - 'A') * .01, (c - 'A') * .01 + .01);
    else
      throw "character out of range";
  }
} model;
```

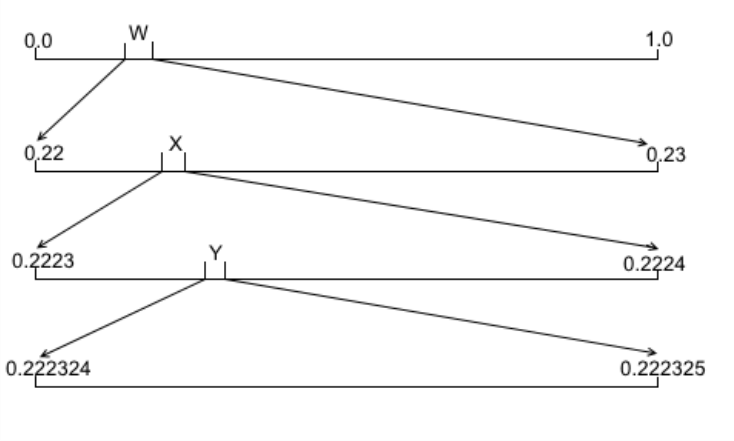
So in this probability model, 'A' owns the range from 0.0 to 0.01, 'B' from .01 to .02, 'C' from .02 to .03, and so on. (Note that this is not an accurate or effective model, but its simplicity is useful at this point.) For a representative example, I called this encoder with the string "WXYZ". Let's walk through what happens in the encoder.

We start with `high` and `low` set to 1.0 and 0.0. The encoder calls the model to get the probabilities for letter 'W', which returns

the interval [0.22, 0.23) - the range along the probability line that 'W' owns in this model. If you step over the next two lines, you'll see that `low` is now set to 0.22, and `high` is set to 0.23.

If you examine how this works, you'll see that as each character is encoded, the range between `high` and `low` becomes narrower and narrower, but `high` will always be greater than `low`. Additionally, the value of `low` is always increasing, and value of `high` is always decreasing. These invariants are important in getting the algorithm to work properly.

So after the first character is encoded, we know that the no matter what other values are encoded, the final number in the message will be less than .23 and greater than or equal to .22. Both `low` and `high` will be greater than equal to 0.22 and less than .23, and `low` will be strictly less than `high`. This means that when decoding, we are going to be able to determine that the first character is 'W' no matter what happens after this, because the final encoded number will fall into the range owned by 'W'. The narrowing process is roughly shown in the figure below:



Successive narrowing of the encoder range

Let's see how this narrowing works when we process the second character, 'X'. The model returns a range of [.23, .24) for this character, and the subsequent recalculation of `high` and `low` results in and interval of [.2223, .2224). So `high` and `low` are still inside the original range of [.22, .23), but the interval has narrowed.

After the final two characters are included, the output looks like:

```
Encoded message: 0.2223242550
```

I'll talk more about how the exact value we want to output needs to be chosen, but in theory at least, for this particular message, any floating point number in the interval [0.22232425,0.22232426) should properly decode to the desired values.

Decoding With Floating Point Math

I find the encoding algorithm to be very intuitive. The decoder reverses the process, and is no more complicated, but the steps might not seem quite as obvious. A first pass algorithm at decoding this message would look something like this:

```
void decode(double message)
{
  double high = 1.0;
  double low = 0.0;
  for ( ; ; )
  {
    double range = high - low;
    char c = model.getSymbol((message - low)/range);
    std::cout << c;
    if ( c == 'Z' )
      return;
    std::pair<double,double> p = model.getProbability(c);
    high = low + range * p.second;
    low = low + range * p.first;
  }
}
```

The math in the decoder basically reverses the math from the encode side. To decode a character, the probability model just has to find the character whose range covers the current value of the message. When the decoder first starts up with the sample value of 0.22232425, the model sees that the value falls between the interval owned by 'W': [0.22,0.23), so the model returns W. In `fp_proto.cpp`, the decoder portion of the simple model looks like this:

```
static char getSymbol( double d)
{
  if ( d >= 0.0 && d < 0.26)
    return 'A' + static_cast<int>(d*100);
  else
    throw "message out of range";
}
```

In the encoder, we continually narrow the range of the output value as each character is processed. In the decoder we do the same narrowing of the portion of the message we are inspecting for the next character. After the 'W' is decoded, `high` and `low` will now define an interval of [0.22,0.23), with a range of .01. So the formula that calculates the next probability value to be decoded, `(message - low)/range`, will be .2324255, which lands right in the middle of of the range covered by 'X'.

This narrowing continues as the characters are decoded, until the hardcoded end of message, letter 'Z' is reached. Success!

Notes and Summary So Far

The algorithm demonstrated to this point can be tested with `fp_proto.cpp` in the attached project. You need to be very aware that we have seen so far is not really a valid or useful encoder. It does, however, demonstrate with some accuracy the general flow of the algorithm. The key observations so far are:

- Characters are encoded according to their position in the probability range [0, 1).
- We keep track of the current state using variables `high` and `low`. A valid output result will be any number in the range [`low`, `high`).
- As each character is encoded, it compresses the range [`low`, `high`) in a way that corresponds exactly to its position in the probability range.
- We have a few invariants that result from the math used in the algorithm. The value of `low` never decreases, the value of `high` never increases, and `low` is always less than `high`.

The big problem with this demonstration algorithm is that it depends on C++ `double` values to hold the message state. Floating point variables have limited precision, which means that eventually, as the range between `high` and `low` continues to narrow, the distance between them becomes too small to represent with floating point variables. With the model used here, you can encode 10 characters or so, but after that the algorithm won't work.

The rest of this article will present a fully functional algorithm. Conceptually it will look very much like the code already presented. The big difference is that the variables used in the encoder and decoder, `high`, `low`, and `message`, are no longer going to be of the C++ type `double`. Instead they will be arbitrarily long binary variables.

In order for our program to handle binary numbers of arbitrary length, they will be processed a bit at a time, with bits being read in, calculations being performed, and then bits being output and then discarded as they are no longer needed. The details of how this is done are where all of the work comes into play in this algorithm.

In addition to those modifications, the rest of this article will cover a few other points that have not been dealt with yet, including:

- How to deal with the end of the stream.
- How to choose the actual output number from the range [low, high).
- Why arithmetic coding is superior to Huffman coding as an entropy coder.

Unbounded Precision Math With Integers

In this, the second part of the exposition, you are going to have to wrap your head around some unconventional mathematics. The algorithm that was described in the first part of this article is still going to be faithfully executed, but it will no longer be implemented using variables of type `double`. It will instead be implemented with integer variables and integer math, albeit in interesting ways.

The basic concept that we implement is this: the values of `high`, `low`, and the actual encoded message, are going to be binary numbers of unbounded length. In other words, by the time we finish encoding the Moby Dick on Project Gutenberg, `high` and `low` will be millions of bits long, and the output value itself will be millions of bits long. All three will still represent a number greater than or equal to 0, and less than 1.

An even more interesting facet of this is that even though the three numbers in play are millions of bits long, each time we process a character we will only do a few operations of simple integer math - 32 bits, or perhaps 64 if you like.

Number Representation

Recall that in the reference version of the algorithm, low and high were initialized like this:

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

In the integer version of the algorithm, we switch to a representation like this:

```
unsigned int high = 0xFFFFFFFFU;
unsigned int low = 0;
```

Both numbers have an implied decimal point leading their values, which would mean that `high` is actually (in hex) 0.FFFFFFFF, or in binary, 0.1111…1111, and `low` is 0.0. The number that we output will likewise have an implied decimal point before the first bit.

But this is not quite right - in the first implementation `high` was 1.0. The value 0.FFFFFFF is close, but it is just a bit less than 1.0. How is this dealt with?

This is where a bit of mind-bending math comes into play. Although `high` has 32 bits in memory, we consider it to have an infinitely long trail of binary 1's trailing off the right end. So it isn't just 0.FFFFFFFF, that string of F's (or 1's) continues off into infinity - they are out there, but haven't been shifted into memory yet.

And while it may not be obvious, Google can help convince you that an infinite string of of 1's starting with 0.1 is actually equal to 1. (The short story: 2x - x = 1.0, therefore x = 1.0.) Likewise, `low` is considered to be an infinitely long binary string, with 0's hanging off the end out to the last binary place.

Resulting Changes to the Model

Remember that the final implementation is going to be entirely implemented using integer math. Previously, the model returned probabilities as a pair of floating point numbers representing the range that a particular symbol owns.

In the updated version of the algorithm, a symbol still owns a specific range of on the number line greater than equal to 0 and less than 1. But we are now going to represent these using a pair of fractions: `upper` / `denom` and `lower` / `denom`. This doesn't actually affect our model code too much. The sample model we used in the previous section returned, for example, .22 and .23 for character 'W'. Now, instead, it will return {22, 23, 100} in a structure called `prob`.

The Real Encoder - First Pass

Before getting into final, working code, I'm presenting some code that implements the basic algorithm, but takes a couple of shortcuts around real problems. This not-quite-done code looks like this:

```
unsigned int high = 0xFFFFFFFFU;
unsigned int low = 0;
char c;
while ( input >> c ) {
  int range = high - low + 1;
  prob p = model.getProbability(c);
  high = low + (range * p.upper)/p.denominator;
  low = low + (range * p.lower)/p.denominator;
  for ( ; ; ) {
    if ( high < 0x80000000U )
      output_bit( 0 );
    else if ( low >= 0x80000000U )
      output_bit( 1 );
    else
      break;
    low <<= 1;
    high << = 1;
    high |= 1;
  }
}
```

The first part of this loop is conceptually and functionally identical to the floating point code given in part 1. We take the range - that is, the difference between and `low` and `high`, and we allocate some subset of that range to the character being encoded, depending on the probability returned from the model. The result is that `low` gets a little larger, and `high` gets a little smaller.

The second part of the code is a little more interesting. The `while` loop is new, and what it does is new as well - the simplified floating point algorithm didn't have anything like this. It performs range checks on `low` and `high`, looking for situations in which the values have the same most significant bit.

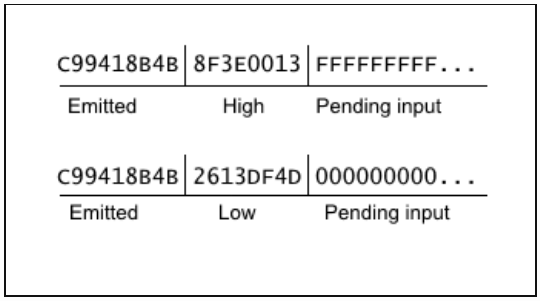
The first check looks to see if `high` has dropped below 0x80000000, in which case its MSB is 0. Because we know that `low` is always less than `high`, its MSB will also be 0. And because the two values only get closer to one another, the MSB of both values will forever be 0.

The other range check looks to see if `low` has increased above 0x7FFFFFFF, in which case both it and `high` will have MSB values of 1, and will always have an MSB of 1.

In either of these cases, remember that we have three invariants to work with `high` only decreases, `low` only increases, and `high` is always greater than `low`. So once `high` has a leading bit of 0, it will never change. Once `low` has a leading bit of 1, it will never change. If this is the case, we can output that bit to the output stream - we *know* what it is with 100% certainty, so let's shift it out to the output stream and get rid of it.

And after we have output the bit, we discard it. Shifting `low` and `high` one bit to the left discards that MSB. We shift in a 1 into the least significant bit of `high`, and a 0 into the least significant of `low`. Thus, we keep working on 32 bits of precision by expelling the bits that no longer contribute anything to the precision of our calculations. In this particular implementation, we just keep 32 bits in working registers, with some additional number already sent to the output, and some other number pending for input.

The figure below shows how the math system now works while in the middle of some arbitrary compression run. Even though we are using 32-bit math, the algorithm is now dealing with arbitrarily long versions of `high` and `low`:



The low and high values, including bits not in memory

As `low` and `high` converge, their matching digits are shifted out on the left side, presumably to a file. These digits are never

going to change, and are no longer needed as part of the calculation. Likewise, both numbers have an infinite number of binary digits that are being shifted in from the right - 1's for `high` and 0's for `low`.

A Fatal Flaw and the Workaround

In the code just presented we have a pretty reasonable way of managing the calculation that creates an arbitrarily long number. This implementation is good, but it isn't perfect.

We run into problems with this algorithm when the values of `low` and `high` start converging on a value of 0.5, but don't quite cross over the line that would cause bits to be shifted out. A sequence of calculations that runs into this problem might produce values like these:

```
low=7C99418B high=81A60145
low=7FF8F3E1 high=8003DFFA
low=7FFFFC6F high=80000DF4
low=7FFFFFF6 high=80000001
low=7FFFFFFF high=80000000
```

Our algorithm isn't quite doing what we want here. The numbers are getting closer and closer together, but because neither has crossed over the 0.5 divider, no bits are getting shifted out. This process eventually leaves the values of the two numbers in a disastrous position.

Why is it disastrous? The initial calculation of `range` is done by subtracting `low` from `high`. In this algorithm, `range` stands in as a proxy for the number line. The subsequent calculation is intended to find the proper subsection of that value for a given character. For example, in the earlier example, we might have wanted to encode a character that has a range of [0.22,0.23). If the value of `range` is 100, then we can clearly see that the character will subdivide that with values of 22 and 23.

But what happens if `low` and `high` are so close that `range` just has value of 1? Our algorithm breaks down - it doesn't matter what the probability of the next character is if we are going to try to subdivide the range [0,1) - we get the same result. And if we do that identically for every character, it means the decoder is not going to have any way to distinguish one character from another.

Ultimately this can decay to the point where `low == high`, which breaks one of our key invariants. It's easy to work out what kind of disaster results at that point.

Later on in this article I'll go over some finer points in the algorithm that show why you run into trouble long before the two values are only 1 apart. We'll come up with specific requirements for the accuracy needed by the value of `range`. Even without those detailed requirements, clearly something needs to be done here.

The fix to the algorithm is shown in the code below. This version of the algorithm still does the normal output of a single bit when `low` goes above 0.5 or `high` drops below it. But when the two values haven't converged, it adds a check of the next most significant bits to see if we are headed towards the problem of *near-convergence*. This will be the case when the two most significant bits of `high` are 10 and the two most significant bits of `low` are 01. When this is the case, we know that the two values are converging, but we don't yet know what the eventual output bit is going to be.

```
unsigned int high = 0xFFFFFFFFU;
unsigned int low = 0;
int pending_bits = 0;
char c;
while ( input >> c ) {
  int range = high - low + 1;
  prob p = model.getProbability(c);
  high = low + (range * p.upper)/p.denominator;
  low = low + (range * p.lower)/p.denominator;
  for ( ; ; ) {
    if ( high < 0x80000000U ) {
      output_bit_plus_pending( 0 );
      low <<= 1;
      high << = 1;
      high |= 1;
    } else if ( low >= 0x80000000U ) {
      output_bit_plus_pending( 1 );
      low <<= 1;
      high << = 1;
      high |= 1;
    } else if ( low >= 0x40000000 && high < 0xC0000000U )
      pending_bits++;
      low << = 1;
      low &= 0x7FFFFFFF;
      high << = 1;
      high |= 0x80000001;
    } else
      break;
  }
}

void output_bit_plus_pending(bool bit, int &pending_bits)
{
  output_bit( bit );
  while ( pending_bits-- )
    output_bit( !bit );
}
```

So what do we do when we are in this near-converged state? We know that sooner or later, either `high` is going to go below 0.5 or `low` will go above it. In the first case, both values will have leading bits of 01, and in the second, they will both have leading bits of 10. As this convergence increases, the leading bits will extend to either 01111… or 10000…, with some finite number of digits extended.

Given this, we know that once we figure out what the first binary digit in the string is going to be, the subsequent bits will all be the opposite. So in this new version of the algorithm, when we are in the near-convergence state, we simply discard the second most significant bit of `high` and `low`, shifting the remaining bits left, while retaining the MSB. Doing that means also incrementing the `pending_bits` counter to acknowledge that we need to deal with it when convergence finally happens. An example of how this looks when squeezing out the converging bit in `low` is shown here below. Removing the bit from `high` is basically identical, but of course a 1 is shifted into the LSB during the process.

```
0101 1010 0011 1100 0100 0100 0000 1111
Erase 2nd most significant bit:
0 01 1010 0011 1100 0100 0100 0000 1111
Shift in new LSB
0011 0100 0111 1000 1000 1000 0001 1110
Increment pending bit count
```

The two steps of removing a bit to prevent overlow

The bit twiddling that makes this happen can be a bit difficult to follow, but the important thing is that the process has to adhere to the following rules:

1. The low 30 bits of both `low` and `high` are shifted left one position.
2. The least significant bit of `low` gets a 0 shifted in.
3. The least significant bit of `high` gets a 1 shifted in.
4. The MSB of both words is unchanged - after the operation it will still be set to 1 fo`high` and 0 for `low`.

The final change that ties all this together is the introduction of the new function `output_bit_plus_pending()`. Each time that we manage this near-convergence process, we know that another bit has been stored away - and we won't know whether it is a one or a zero. We keep a count of all these consecutive bits in `pending_bits`. When we finally reach a situation where an actual MSB can be output, we do it, plus all the pending bits that have been stored up. And of course, the pending bits will be the opposite of the bit being output.

This fixed up version of the code does everything we need to properly encode. The final working C++ code will have a few differences, but they are mostly just tweaks to help with flexibility. The code shown above is more or less the final product.

The Real Decoder

I've talked about some invariants that exist in this algorithm, but one I have skipped over is this: the values of `high` and `low` that are produced as each character is processed in the encoder will be duplicated in the decoder. These values operate in lockstep, right down to the least significant bit.

A consequence of this is that the code in the decoder ends up looking a lot like the code in the encoder. The manipulation of `high` and `low` is effectively duplicated. And in both the encoder and the decoder, the values of these two variables are manipulated using a calculated `range`, along with the probability for the given character.

The difference between the two comes from how we get the probability. In the encoder, the character is known because we are reading it directly from the file being processed. In the decoder, have to determine the character by looking at value of the message we are decoding - where it falls on the [0,1) number line. It is the job of the model to figure this out in function `getChar()`.

The compressed input is read into a variable named `value`. This variable is another one of our pseudo-infinite variables, like `high` and `low`, with the primary difference being what is shifted into it in the LSB position. Recall that `high` gets an infinite string of 1's shifted in, and `low` gets an infinite string of 0's. `value` gets something completely different - it has the bits from the encoded message shifted into. So at any given time, `value` contains 32 of the long string of bits that represent the number that the encoder created. On the MSB side, bits that are no longer are used in the calculation are shifted out of `value`, and on the LSB side, as those bits are shifted out, new bits of the message are shifted in.

The resulting code looks like this:

```
unsigned int high = 0xFFFFFFFFU;
unsigned int low = 0;
unsigned int value = 0;
for ( int i = 0 ; i < 32 ; i++ ) {
  value <<= 1;
  value += m_input.get_bit() ? 1 : 0;
}
for ( ; ; ) {
  unsigned int range = high - low + 1;
  unsigned int count =  ((value - low + 1) * m_model.getCount() - 1) / range;
  int c;
  prob p = m_model.getChar( count, c );
  if ( c == 256 )
    break;
  m_output.putByte(c);
  high = low + (range*p.high)/p.count - 1;
  low = low + (range*p.low)/p.count;
  for( ; ; ) {
    if ( low >= 0x80000000U || high < 0x80000000U ) {
      low <<= 1;
      high <<= 1;
      high |= 1;
      value <<= 1;
      value += m_input.get_bit() ? 1 : 0;
    } else if ( low >= 0x40000000 && high < 0xC0000000U ) {
      low << = 1;
      low &= 0x7FFFFFFF;
      high <<= 1;
      high |= 0x80000001;
      value <<= 1;
      value += m_input.get_bit() ? 1 : 0;
    } else
      break;
  }
}
```

So this code looks very similar to the final encoder. The updating of the values is nearly identical - it adds the update of `value` that updates in tandem with `high` and `low`. The nature of the algorithm introduces another invariant: `value` will always be greater than or equal to `low` and less than `high`.

A Sample Implementation

All of this is put together in the production code included in ari.zip. (Links are at the end of this article.) The use of templates makes this code very flexible, and should make it easy to plug it into your own applications. All the needed code is in header files, so project inclusion is simple.

In this section I'll discuss the various components that need to be put together in order to actually do some compression. The code package has four programs:

- `fp_proto.cpp`, the floating point prototype program. Useful for experimentation, but not for real work.
- `compress.cpp`, which compresses a file using command line arguments.
- `decompress.cpp`, which decompresses a file using command line arguments.
- `tester.cpp`, which puts a file through a compress/decompress cycle, tests for validity, and outputs the compression ratio.

The compression code is implemented entirely as a set of template classes in header files. These are:

- `compressor.h`, which completely implements the arithmetic compressor. The compressor class is parameterized on input stream type, output stream type, and model type.
- `decompressor.h`, the corresponding arithmetic decompressor with the same type parameters.
- `modelA.h`, a simple order-0 model that does an acceptable job of demonstrating compression.

- `model_metrics.h`, a utility class that helps a model class set up some types used by the compressor and decompressor.
- `bitio.h` and `byteio.h`, the streaming classes that implement bit-oriented I/O.

This article is going to skip over the details pertaining to the bit-oriented I/O classes implemented in `bitio.h` and `byteio.h`. The classes provided will allow you to read or write from `std::iostream` and `FILE *` sources and destinations, and can be pretty easily modified for other types. Details on the implementation of these classes are in my article C++ Generic Programming Meets OOP. which includes a bit-oriented I/O class as an illustrative example.

All of the code I use requires a C++11 compiler, but could be modified to work with earlier versions without much trouble. The makefile will build the code with g++ 4.6.3 or clang 3.4. The solution file included will build the projects with Visual C++ 2012.

model_metrics.h

In the code I've shown you so far, I blithely assume that your architecture uses 32-bit math and efficiently supports unsigned integer math. This is a bit of an unwanted and unneeded restriction, so my production code will define the math parameters using templates. The way it works is that the compressor and decompressor classes both take a `MODEL` type parameter, and they rely on that type to provide a few typedefs and constants:

`CODE_VALUE`	The integer type used to perform math. In the sample code shown so far we assumed this was `unsigned int`, but signed and longer types are perfectly valid.
`MAX_CODE`	The maximum value of a code - in other words the highest value that `high` can reach. If we are using all 32 bits of an unsigned in, this would be 0xFFFFFFF, but as we will see shortly, this will normally be quite a bit smaller than the max that can fit into an int or unsigned int.

`ONE_FOURTH`
`ONE_HALF`
`THREE_FOURTHS`
	The three values used when testing for convergence. In the sample code these were set to full 32-bit values of 0x40000000, 0x80000000, and 0xC0000000, but they will generally be smaller than this. The values could be calculated from `MAX_CODE`, but we let the model define them for convenience.

`struct prob`	This is the structure used to return probability values from the model. It is defined in the model because the model will know what types it wants the three values in this structure to be. The three values are `low`, `high`, and `count`, which define the range of a given character being encoded or decoded.

The header file `model_metrics.h` contains a utility class that helps the model class figure out what these types are. In general, the choice of `CODE_VALUE` is going to be defined using the natural int or unsigned int type for your architecture. Calculating the value of `MAX_CODE` requires a little bit more thinking though.

Digression - Overflow Avoidance

It's a given that we are doing the math in the encoder and decoder using type `CODE_VALUE`. We need to make sure that these three lines of code don't generate an intermediate result that won't fit in that type:

```
CODE_VALUE range = high - low + 1;
high = low + (range * p.high / p.count) - 1;
low = low + (range * p.low / p.count);
```

The values of `high` and `low` will have some number of bits, which in `model_metrics.h` we designate by `CODE_VALUE_BITS`. The maximum number of bits that can be in the counts `low` and `high` returned from the model in `struct prob` are defined in the header file by `FREQUENCY_BITS`. The same header defines `PRECISION` as the maximum number of bits that can be contained in type `CODE_VALUE`. Looking at the calculation above shows you that this expression must always be true:

```
PRECISION >= CODE_VALUE_BITS + FREQUENCY_BITS
```

On machines commonly in use today, `PRECISION` will be 31 if we use signed integers, 32 if we use unsigned. If we arbitrarily split the difference, we might decide that `CODE_VALUE_BITS` and `FREQUENCY_BITS` could both be 16. This will avoid overflow, but in fact it doesn't address a second constraint, discussed next.

Digression - Underflow Avoidance

In the decoder, we have the following code that is executed each time decode a new character:

```
CODE_VALUE range = high - low + 1;
CODE_VALUE scaled_value =  ((value - low + 1) * m_model.getCount() - 1 ) / range;
int c;
prob p = m_model.getChar( scaled_value, c );
```

The value returned by `m_model.getCount()` will be a frequency count, so it will be represented by `FREQUENCY_BITS`. We need to make sure that there are enough possible values of `scaled_value` so that it can be used to look up the smallest values in the model.

Because of the invariants described earlier, we know that `high` and `low` have to be at least `ONE_FOURTH` apart. The MSB of `high` has to be 1, making it greater than `ONE_HALF`, and the MSB of `low` has to be 0, making it less than `ONE_HALF`. But in addition, the special processing for near-convergence insures that if `high` is less than `THREE_FOURTHS`, then `low` must be less than `ONE_FOURTH`. Likewise, if `low` is greater than `ONE_FOURTH`, then `high` must be greater than `THREE_FOURTHS`. So the worst case for convergence between these values is represented, for example, when `low` is `ONE_HALF-1` and `high` is `THREE_FOURTHS`. In this case, `range` is going to be just a bit over `ONE_FOURTH`.

Now let's consider what happens when we are using 16 bits for our frequency counts. The largest value returned by `m_model.getCount()` will be 65,535. Let's look at what might happen if we were using just eight bits in `CODE_VALUE`, making `MAX_CODE` 256. In the worst case, `high` might be 128 and `low` might be 63, giving a range of 65. Because `value` has to lie between `low` and `high` (back to invariants), this means there are only going to 65 possible values of `scaled_value`. Because the range occupied by a given symbol can be as small as 1, we need to be able to call `m_model.getChar()` with any value in the range [0,65536) to be able to properly decode a rarely appearing character. Thus, `MAX_CODE` of 255 won't cut it.

In the worst case we are dividing the range occupied by `CODE_VALUE` calculations by 4, so we are in effect chopping two bits off the range. In order to get that scaled up so it can generate all the needed values of `scaled_value`, we end up with this prerequisite:

```
CODE_VALUE_BITS >= (FREQUENCY_BITS + 2)
```

If we have a 32 bit `CODE_VALUE` type to work with, this means that a comfortable value of `CODE_VALUE` will be 17, making `FREQUENCY_BITS` 15. In general, we want `FREQUENCY_BITS` to be as large as possible, because this provides us with the most accurate model of character probabilities. Values of 17 and 15 maximize `FREQUENCY_BITS` for 32-bit unsigned integer math.

modelA.h

I haven't talked too much about modeling in this article. The focus is strictly on the mechanics of arithmetic coding. We know that this depends on having an accurate model of character probabilities, but getting deep into that topic requires another article.

For this article, the sample code uses a simple adaptive model, which I call `modelA`. This model has character probabilities for 257 symbols - all possible symbols in an eight-bit alphabet, plus one additional EOF symbol. All of the characters in the model start out with a count of 1, meaning each has an equal chance of being seen, and that each will take the same number of bits to encode.

After each character is encoded or decoded, its count is updated in the model, making it more likely to be seen, and thus reducing the number of bits that will be required to encode it in the future.

The model is implemented by using a `CODE_VALUE` array of size 258. The range for a given character i is defined by the count at i and the count at i+1, with the total count being found at location 257. Thus, the routine to get the probability looks like this:

```
prob getProbability(int c)
{
  prob p = { cumulative_frequency[c],
             cumulative_frequency[c+1],
             cumulative_frequency[257] };
  if ( !m_frozen )
    update(c);
  pacify();
  return p;
}
```

Because of the way arithmetic coding works, updating the count for character i means the counts for all characters greater than i have to be incremented as well, because adjusting the range for one position on the number line is going to squeeze everyone to the right - the counts in `cumulative_frequency` are in fact *cumulative*, representing the sum of all characters less than i. This means the update function has to work its way through the whole array. This is a lot of work, and there are various things we could do to try to make the model more efficient. But `ModelA` is just for demonstration, and the fact that the array probably fits neatly in cache means this update is adequate for now.

One additional factor we have to watch out for is that the we can't exceed the maximum number of frequency bits in our count. This is managed by setting a flag to freeze the model once the maximum frequency is reached. Again, there are a number of things we could do to make this more efficient, but for demonstration purposes this is adequate:

```
  void inline update(int c)
  {
    for ( int i = c + 1 ; i < 258 ; i++ )
      cumulative_frequency[i]++;
    if ( cumulative_frequency[257] >= MAX_FREQ )
      m_frozen = true;
  }
```

The decoder has to deal with the same issues when looking up a character based on the `scaled_value`. The code does a linear search of the array, whose only saving grace is that it ought to be sitting in the cache:

```
prob getChar(CODE_VALUE scaled_value, int &c)
{
  for ( int i = 0 ; i < 257 ; i++ )
    if ( scaled_value < cumulative_frequency[i+1] ) {
      c = i;
      prob p = {cumulative_frequency[i],
                cumulative_frequency[i+1],
                cumulative_frequency[257]};
      if ( !m_frozen)
        update(c);
      return p;
    }
  throw std::logic_error("error");
}
```

In a future article, the notion of how to develop efficient models will be a good topic to cover. You won't want to use `modelA` as part of a production compressor, but it does a great job of letting you dig into the basics of arithmetic compression.

When you instantiate `modelA` you have three optional template parameters, which you can use to tinker with the math used by the compressor and decompressor. (Of course, the compressor and decompressor have to use the same parameters).

```
template<typename CODE_VALUE_ = unsigned int,
        int CODE_VALUE_BITS_ = (std::numeric_limits<CODE_VALUE_>::digits + 3) / 2,
        int FREQUENCY_BITS_ = std::numeric_limits<CODE_VALUE_>::digits - CODE_VALUE_BITS_>
struct modelA : public model_metrics<CODE_VALUE_, CODE_VALUE_BITS_, FREQUENCY_BITS_>
{
```

On 32 bit compiler, opting for all defaults will result in you using 17 bits for `CODE_VALUE` calculations, and 15 bits for frequency counts. Static assertions in the `model_metrics` class check to insure that your selections will result in correct compression.

compressor.h

This header file contains all the code that implements the arithmetic compressor. The class that does the compression is a template class, parameterized on the input and output classes, as well as the model. This allows you to use the same compression code with different types of I/O in as efficient a manner as possible.

The compressor engine is a simple C++ object that has an overloaded `operator()()`, so the normal usage is to instantiate the engine then call it with input, output, and model parameters. Because of the way that C++ manages template instantiation, the easiest way to actually use the engine is via a convenience function, `compress()`, that takes care of those details, and can be called without template parameters, as in:

```
std::ifstream input(argv[1], std::ifstream::binary);
std::ofstream output(argv[2], std::ofstream::binary);
modelA<int, 16, 14> cmodel;
compress(input, output, cmodel);
```

You can see the (simple) implementation of the convenience function in the attached source.

The `operator()()` is where all the work happens, and it should look very similar to the trial code shown earlier in this article. The difference is that what is shown here is fully fleshed out, with all the details taken care of:

```
int operator()()
{
  int pending_bits = 0;
  CODE_VALUE low = 0;
  CODE_VALUE high = MODEL::MAX_CODE;
  for ( ; ; ) {
    int c = m_input.getByte();
    if ( c == -1 )
      c = 256;
    prob p = m_model.getProbability( c );
    CODE_VALUE range = high - low + 1;
    high = low + (range * p.high / p.count) - 1;
    low = low + (range * p.low / p.count);
    //
    // On each pass there are six possible configurations of high/low,
    // each of which has its own set of actions. When high or low
    // is converging, we output their MSB and upshift high and low.
    // When they are in a near-convergent state, we upshift over the
    // next-to-MSB, increment the pending count, leave the MSB intact,
    // and don't output anything. If we are not converging, we do
    // no shifting and no output.
    // high: 0xxx, low anything : converging (output 0)
    // low: 1xxx, high anything : converging (output 1)
    // high: 10xxx, low: 01xxx : near converging
    // high: 11xxx, low: 01xxx : not converging
    // high: 11xxx, low: 00xxx : not converging
    // high: 10xxx, low: 00xxx : not converging
    //
    for ( ; ; ) {
      if ( high < MODEL::ONE_HALF )
```

```
      put_bit_plus_pending(0, pending_bits);
    else if ( low >= MODEL::ONE_HALF )
      put_bit_plus_pending(1, pending_bits);
    else if ( low >= MODEL::ONE_FOURTH && high < MODEL::THREE_FOURTHS ) {
      pending_bits++;
      low -= MODEL::ONE_FOURTH;
      high -= MODEL::ONE_FOURTH;
    } else
      break;
    high <<= 1;
    high++;
    low <<= 1;
    high &= MODEL::MAX_CODE;
    low &= MODEL::MAX_CODE;
  }
  if ( c == 256 ) //256 is the special EOF code
    break;
 }
 pending_bits++;
 if ( low < MODEL::ONE_FOURTH )
  put_bit_plus_pending(0, pending_bits);
 else
  put_bit_plus_pending(1, pending_bits);
 return 0;
}
inline void put_bit_plus_pending(bool bit, int &pending_bits)
{
 m_output.put_bit(bit);
 for ( int i = 0 ; i < pending_bits ; i++ )
  m_output.put_bit(!bit);
 pending_bits = 0;
}
```

Digression - Internal End of File

The output stream created by this encoder depends on a special EOF character to indicate that it is complete. That character, 256, is the last symbol encoded and output to the stream. When the decoder reads a symbol of 256, it stops decoding and storing symbols and knows it is done.

This method of terminating a stream works pretty well in most scenarios. There are some interesting alternatives you can take if you are compressing individual files or other data types (such as network packets) that have a system-imposed EOF. Since a file already has an EOF in place, we are wasting some information by encoding our own EOF symbol.

David Scott has done a lot of work in this area, and some of his results are here. His general technique for eliminating internal EOF signaling in any compression algorithm is to first make the algorithm bijective. When a compression algorithm is bijective, it means that *any file* will decompress properly to some other file. This is generally not the case for most compressors - there are usually illegal files that will break the decompressor, and the code I have presented here definitely has that problem.

The next step is to ensure that when the input file reaches an EOF, we can terminate the compressed file at that point with a valid sequence of bytes that ends on a byte boundary. This means that when the decompressor sees an EOF on input, it knows two things. First, it knows that it has properly decoded and output all symbols up to this point. Second, that it is done.

Depending on an external EOF to terminate a compressed stream won't always work - if you are reading data from some sort of stream that doesn't have external termination, you will need an internal EOF marker, as used here, to delineate the end of the compressed stream.

Digression - Ending the Encoded Value
At some point when `c==256`, `high` and `low` will stop converging, and we will break out of the main encoding loop - we are more or less done.

The question at this point is: how many more bits do we need to output to ensure that the decoder can decode the last symbol(s) properly? The easiest way to do this might be to just output `CODE_VALUE_BITS` of `low`, ensuring that all the precision we have left is used. But we don't actually need this much precision. All we need to guarantee is that when the decoder is reading in `value`, the final bits ensure that `low <= value < high`. Because of the way the encoding loop works, we know that when we break out of it, `high` and `low` are in just one of three states:

high	low
11xxx	01yyy
11xxx	00yyy
10xxx	00yyy

Examining this, we can see that if `low` starts with `00`, we can write the bits `01` and be satisfied that any values for the remaining bits will result in `value` being greater than `low` and less than `high`. If `low` starts with `01`, we can write the bits `10` and again be ensured that the desired condition holds. Because there may be additional pending bits left over from the encoding loop, we write `01` or `10` by calling `put_bit_plus_pending()` with a `0` or `1`, after incrementing `pending_bits`. That incremented value of `pending_bits` ensures that if a `0` is written, it will be followed by a `1`, and vice versa. So just two bits (plus any pending) are all that are needed to properly terminate the bit stream.

decompressor.h

The decompressor code mirrors the compressor code. The engine is a template class that is parameterized on the input and output stream types, and the model type. A convenience function called `decompress()` makes it easy to instantiate the engine and then decompress without needing to declare template parameters:

```
std::ifstream input(argv[1], std::ifstream::binary);
std::ofstream output(argv[2], std::ofstream::binary);
modelA<int, 16, 14> cmodel;
decompress(input, output, cmodel);
```

The convenience function is in the attached source package.

The actual operator code is shown here. This is very much like the sample code shown earlier, with all the additional loose ends tied up so that this is production-ready:

```
int operator()()
{
  CODE_VALUE high = MODEL::MAX_CODE;
  CODE_VALUE low = 0;
  CODE_VALUE value = 0;
  for ( int i = 0 ; i < MODEL::CODE_VALUE_BITS ; i++ ) {
    value <<= 1;
    value += m_input.get_bit() ? 1 : 0;
  }
  for ( ; ; ) {
    CODE_VALUE range = high - low + 1;
    CODE_VALUE scaled_value =  ((value - low + 1) * m_model.getCount() - 1 ) / range;
    int c;
    prob p = m_model.getChar( scaled_value, c );
    if ( c == 256 )
      break;
    m_output.putByte(c);
    high = low + (range*p.high)/p.count -1;
    low = low + (range*p.low)/p.count;
    for( ; ; ) {
      if ( high < MODEL::ONE_HALF ) {
        //do nothing, bit is a zero
      } else if ( low >= MODEL::ONE_HALF ) {
        value -= MODEL::ONE_HALF;  //subtract one half from all three code values
        low -= MODEL::ONE_HALF;
        high -= MODEL::ONE_HALF;
      } else if ( low >= MODEL::ONE_FOURTH && high < MODEL::THREE_FOURTHS ) {
        value -= MODEL::ONE_FOURTH;
        low -= MODEL::ONE_FOURTH;
        high -= MODEL::ONE_FOURTH;
      } else
        break;
      low <<= 1;
      high <<= 1;
      high++;
      value <<= 1;
      value += m_input.get_bit() ? 1 : 0;
    }
  }
  return 0;
}
```

Digression - Priming the Input Pump

One of the implementation problems when performing arithmetic coding is the management of the end of the stream. An example of this problem can be imagined if we have a best-case scenario in which we encode an entire file using just two or three bits. The actual compressed file will of course need to have one byte, but that's it.

We run into a problem in the decode, because we need to be able to fill the initial value of `value` with the appropriate number of bits when decoding. A typical number would be 17, which would require that we read three bits into `value` before we start encoding. Something like this is executed when the encoder starts:

```
for ( int i = 0 ; i < MODEL::CODE_VALUE_BITS ; i++ ) {
  value <<= 1;
  value += m_input.get_bit() ? 1 : 0;
}
```

For every eight bits read via `get_bit()`, there will be one call to read a byte from the underlying file or other input stream.

We could just ignore EOF conditions and return 0xFF or 0x00 whenever reading a byte. This would work pretty well as

long as we properly detect the end of stream in our encoded stream. But in the case of encoder error, or file corruption, it could result in the decoder running forever, reading in bogus values, decoding them to other bogus values, and writing them to output.

Things would be much better if this error condition was detected. What we would really like is for the attempt to read past the end of file to generate an error.

To accomplish this, the bit-oriented I/O class I use for input has a specialized constructor that asks for the number of bits that will be used in a `CODE_VALUE`. When constructing it in the convenience function, I pass this along:

```
input_bits<INPUT_CLASS> in(source, MODEL::CODE_VALUE_BITS);
```

This value tells the input class that I may need to read that many bits past the EOF, but no more. When constructed, I store that number in the input class member variable `m_CodeValueBits`. The code that reads in bytes when they are needed now has processing that looks like this:

```
m_CurrentByte = m_Input.getByte();
if ( m_CurrentByte < 0 ) {
  if ( m_CodeValueBits <= 0 )
    throw std::logic_error("EOF on input");
  else
    m_CodeValueBits -= 8;
}
```

This gives the error checking we need while still allowing a few reads past the end of file.

Implementation Notes - Using Exceptions for Fatal Errors

Using C++ exceptions properly can be a really difficult task. Ensuring that all possible paths through stack unwinding leave your program in a correct state is not for the faint of heart. This is less of a problem when you are using exceptions strictly as a fatal error mechanism, as I discuss here.

The demo code I use here throws exceptions in response to the following error conditions:

- EOF on the input stream.
- A request for a character from the model with a `scaled_value` larger than the maximum value currently defined - this is a logic error that should never happen.

Tightening up this code would give some other places that this would be useful - for example failures when writing output.

Propagating errors this way is efficient, and helps keep the code clean - no checking for failures from the model or I/O objects, and it provides a very consistent way to implement error handling when you write new classes for modeling or I/O.

To catch these errors, the normal template for compressing or decompressing is going to look like this:

```
try {
  //
  // set up I/O and model
  // then compress or decompress
    return 0; //the success path
  }
  catch (std::exception &ex)
  {
    std::cerr << "Failed with exception: " << ex.what() << "\n";
  }
  return 255; //failure path
```

Implementation Notes - Building the Demo Programs

There are four executables that can be built from the attached source package:

fp_proto:	The floating point prototype code. This code is useful to demonstrate the basics of how arithmetic coding works, but isn't much good for real work, with reasons outlined in the article.
compress:	A compressor that accepts an input and output file name on the command line, then compresses using a simple order-0 model. You should see modestly good compression with this file. To get superior compression requires some work on more advanced modeling.
decompress:	A decompressor that accepts an input and output file name on the command line. This will work properly with files created by the correspond compress program.
tester:	A test program that accepts a single file name on the command line. It executes a compress to a temporary file, then a decompress, then compares the result with the original file. The app exits with 0 if the comparison succeeds, 255 in the event of a failure

If you are working on a Windows box, you can build all four applications from the enclosed `ari.sln` solution file. This should work with Visual C++ 12 or later.

If you are working on Linux, you can build all four files with the attached makefile, using:

```
make all
```

By default this will use g++ to compile, but if you change one option in the file you can easily switch to clang. The code has been tested with g++ version 4.6.3, and clang++ version 3.4.

The code uses a few C++11 features, so if you try to build the projects with earlier versions of any of these compilers you may run into some problems - mostly with the I/O code. It should be reasonably easy to strip out some of the functionality and create classes that will support either `iostreams` or `FILE *` objects.

Implementation Notes - Testing

The `tester` program provides a good way to exercise the code. If you are on a Linux system, you create a directory called `corpus`, populate it with a tree of as many files as you like, then execute `make test`, which will run the tester program against all files. If a failure occurs, the process should abort with message, allowing you to see which file failed:

```
mrn@mrn-ubuntu-12:$ make test
find corpus -type f -print0 | xargs -0r -n 1 ./tester
compressing corpus/about/services/trackback/index.html... 5.41378
compressing corpus/about/services/feed/index.html... 5.71502
compressing corpus/about/services/index.html... 5.41378
compressing corpus/about/trackback/index.html... 5.39995
compressing corpus/about/feed/index.html... 5.25144
compressing corpus/about/serial1/trackback/index.html... 5.43231
compressing corpus/about/serial1/feed/index.html... 5.32922
compressing corpus/about/serial1/index.html... 5.43231
compressing corpus/about/tdcb/trackback/index.html... 5.37245
```

The `tester` app also prints out the number of bits/byte that was achieved by the compressor, that output can be collected to provide stats.

It’s a little harder to do this with a one-liner under Windows, so I am afraid I don’t have an easy bulk test option.

Implementation Notes - Logging

When an arithmetic compressor goes bad, it can be exceedingly difficult to determine what went wrong.

There are two common sources of error. The encoder itself can go awry, some error with bit twiddling, or failure to maintain the invariants that have been discussed ad nauseum here.

The second possible source of error is in the model itself. There are a few things that can go wrong here, but the most common error is a loss of synch between the encoder and decoder models. For things to work properly, both models have to operate in perfect lockstep.

Some low level logging can be turned on in the encoder by building with the `LOG` constant defined. You can do this by editing the makefile and adding `-DLOG` to the compiler command line, or defining it in the *C++|Preprocessor|Preprocessor Definitions* area of the project properties for Windows builds.

Once you have the projects reconfigured, you can do a clean and rebuild of the projects. From that point on, the compressor will create a log file called `compressor.log`, and the decompressor will create a file called `decompressor.log`. These log files will output the state of the compressor or decompressor as each character is encoded or decoded, with output like this:

```
0x2f(/) 0x0 0x1ffff => 0x5da2 0x5f9f
0x2f(/) 0xd100 0x1cfff => 0xff74 0x1016d
0x0d 0xba00 0x1b6ff => 0xc6b2 0xc7ab
0x0a 0xb200 0x1abff => 0xbb9d 0xbc92
0x2f(/) 0x9d00 0x192ff => 0xcb2f 0xce01
0x2f(/) 0xcbc0 0x1807f => 0xed8d 0xf04f
0x20 0x6340 0x113ff => 0x7a19 0x7ac4
0x20 0x3200 0x189ff => 0x5e4d 0x60e7
0x42(B) 0x2680 0x173ff => 0x83a0 0x84e2
0x57(W) 0xa000 0x1e2ff => 0x11492 0x115c8
0x54(T) 0x9200 0x1c8ff => 0xfe53 0xff7c
0x2e(.) 0x5300 0x17cff => 0x8a98 0x8bb4
0x43(C) 0x9800 0x1b4ff => 0xe994 0xeaa2
0x50(P) 0x9400 0x1a2ff => 0xef56 0xf056
0x50(P) 0x5600 0x156ff => 0xac4c 0xae31
```

The data you see there is the character being encoded, both in hex and text representation (if printable), followed by the values of `low` and `high` before and after the character is processed. If you diff the two log files, the only difference between the two should be in the last character position, because the decompressor doesn't update its state when it sees an EOF:

```
mrn@mrn-ubuntu-12:$ diff compressor.log decompressor.log
10587c10587
< 0x100 0x27a0 0x10cff => 0x10cfa 0x10cff
---
> 0x100 0x27a0 0x10cff
mrn@mrn-ubuntu-12:$
```

Any other differences indicate an error. You can use the line number of the position to determine exactly where things went awry.

EOF

By itself, an entropy encoder is not a magic bullet. A good compressor needs sophisticated modeling to back it up. With that good model, an arithmetic encoder will give you excellent results, and should outperform Huffman or other older and more revered coders. There is an entire class of coders referred to as Range encoders that use the ideas here, tailored for greater efficiency.

The computational requirements of an arithmetic encoder are definitely going to be higher than with something like Huffman, but modern CPU architectures should be able to minimize the impact this has on your program. Arithmetic encoders are particularly well suited for adaptive models when compared to Huffman coding.

Source is attached here.

Your comments and corrections are welcome, and will help improve this post as time goes on.

Thanks for reading.

- Mark

Mark Nelson

Books, articles, and posts from 1989 to today.