Merits of Probabilistic Models

N-th order Markov models and PPM models cleanly separate the *model* for symbol probabilities from the *coding* based on those probabilities. Such models have several advantages:

- Coding can be nearly optimal (eg, using arithmetic coding).
- It's easy to try out various modeling ideas.
- You can get very good compression, if you use a good model.

The big disadvantage:

- The coding and decoding involves operations for every symbol and every bit, plus possibly expensive model updates, which limits how fast these methods can be.

Merits of Dictionary Methods

Compression using adaptive dictionaries may be less elegant, but has it's own advantages:

- Dictionary methods can be quite fast (especially at decoding), since whole sequences of symbols are specified at once.
- The idea that the data contain many repeated strings fits many sources quite well — eg, English text, machine-language programs, files of names and addresses.

The main disadvantage is that compression may not be as good as a model based method:

- Dictionaries are inappropriate for some sources — eg, noisy images.
- Even when dictionaries work well, a good model-based method may do better — and can't do worse, if it uses the same modeling ideas as the dictionary method.

The LZ77 Scheme

This scheme was devised by Ziv and Lempel in 1977. There are many variants, including the method used by `gzip`.

The idea of LZ77 is to use the past text as the dictionary — avoiding the need to transmit a dictionary separately. We need a buffer of size W that contains the previous S characters plus the following $W-S$ characters.

We encode up to $W-S$ characters at once by sending the following:

- A pointer to a past character in the buffer (an integer from 1 to S).
- The number of characters to take from the buffer (an integer from 0 to $W-S-1$, or maybe more).
- The single character that follows the string taken from the buffer.

An Example of LZ77 Coding

Suppose we look at the past 16 characters, and look ahead at the next 8 characters.

After encoding the first 16 characters of the following string, we would proceed as follows:

```
Way_over_there_is_where_it_is
No match with string in window.
Transmit (-,0,s)

Way_over_there_is_where_it_is
Match 3 back with _
Transmit (3,1,w)

Way_over_there_is_where_it_is
Match with 9 back with here_i
Transmit (9,6,t)
```

Encoding the Pointers

If we look back S characters, we can encode a pointer back in $\lceil \log_2(S) \rceil$ bits.

If we look forward $W - S$ characters, we can encode the length of the match in $\lceil \log_2(W - S) \rceil$ bits.

The character after the match can be encoded in $\lceil \log_2(q) \rceil$ bits, if we have q symbols.

If these lengths are multiples of 8, we can quickly output these codes as one or more bytes.

An alternative: Use Huffman or arithmetic coding. This will give better compression, but won't be as fast.

LZ77 Encoding and Decoding Speed

Even if writing the codes for the match is fast, *finding* the longest match may be slow.

Techniques such as hashing can speed this up, however. The `gzip` program builds a hash table for all strings of length three, then searches within the hash bucket for the next three characters to find the longest match.

Decoding can be very fast. Reading the codes is very quick if they are take up fixed numbers of bytes. Even if we use Huffman codes, table look up on the next few bits (as in `gzip`) can be pretty fast. Once we have the codes, we just copy text from the buffer.

The LZ78 Scheme

Ziv and Lempel introduced another scheme in 1978, in which the dictionary is kept explicitly, and contains phrases from the entire past text.

In the LZW variant, due to Welch, we start with a dictionary containing just the alphabet. We then proceed as follows:

- Find the longest match of following characters with a dictionary item.
- Transmit the index of that dictionary item.
- Add the matched phrase plus the character following it to the dictionary.
- Continue coding with the character following the matched phrase.

Codes for dictionary indexes will have to get longer as we go, but at a fairly slow rate.