Lempel-Ziv Notes
Prof. Peter Shor

We now explain the algorithm that Lempel and Ziv gave in a 1978 paper, and generally called LZ78. This is opposed to LZ77, an earlier algorithm which differed significantly in the implementation details but is based on the same general idea. This idea is that if some text is not random, a substring that you see once is more likely to appear again than substrings you haven't seen.

The LZ78 algorithm works by constructing a dictionary of substrings, which we will call"phrases," that have appeared in the text. The LZ78 algorithm constructs its dictionary on the fly, only going through the data once. This is a great advantage in that you don't have to receive the entire document before starting to encode it. This might be a problem if, for example, the first half of some document is in English and the second half is in Chinese. In this case, the dictionary constructed for the first half will be suboptimal when used on the second half.

There are many variations of Lempel Ziv around, but they all follow the same basic idea. We'll just concentrate on LZ78 because it is one of the simplest to explain and analyze, although other variants may work somewhat better in practice. The idea is to parse the sequence into distinct phrases. The version we analyze does this greedily. Suppose, for example, we have the string

$$AABABBBABAABABBBABBABB$$

We start with the shortest phrase on the left that we haven't seen before. This will always be a single letter, in this case A:

$$A|ABABBBABAABABBBABBABB$$

We now take the next phrase we haven't seen. We've already seen A, so we take AB:

$$A|AB|ABBBABAABABBBABBABB$$

The next phrase we haven't seen is ABB, as we've already seen AB. Continuing, we get B after that:

$$A|AB|ABB|B|ABAABABBBABBABB$$

and you can check that the rest of the string parses into

$$A|AB|ABB|B|ABA|ABAB|BB|ABBA|BB$$

Because we've run out of letters, the last phrase on the end is a repeated one. That's O.K.

Now, how do we encode this? For each phrase we see, we stick it in a dictionary. The next time we want to send it, we don't send the entire phrase, but just the number of this phrase. Consider the following table

1	2	3	4	5	6	7	8	9
A	AB	ABB	B	ABA	$ABAB$	BB	$ABBA$	BB
$\emptyset A$	$1B$	$2B$	$\emptyset B$	$2A$	$5B$	$4B$	$3A$	7

The first row gives the numbers of the phrase, the second row gives the phrases, and the third row their encodings. That is, when we're encoding the ABAB (the sixth phrase), we encode it as 5B. This maps to ABAB since the fifth phrase was ABA, and we add B to it. Here, the empty set $\emptyset$ should be considered as the 0'th phrase and encoded by 0. The last piece is encoding this string into binary. This gives

$$001110100101001011100101100111$$

To see how this works, I've now inserted dividers and commas, to make it more comprehensible)

$$0, 0|1, 1|10, 1|00, 1|010, 0|101, 1|100, 1|011, 0|0111$$

We have taken the third row of the previous array, expressed all the numbers in binary (before the comma) and the letters in binary (after the comma) Note that I've mapped A to 0 and B to 1. If you had a larger alphabet, you would encode the letters by more than one bit. (In fact, you could even use a Huffman code to encode the letters if you know the frequencies of your letters.) Note also that as soon as a reference to a phrase might conceivably involve k bits (starting with the $2^k + 1$ dictionary element), I've actually used k bits, so the number of bits used before the comma keeps increasing. This ensures that the decoding algorithm knows where to put the commas and dividers.

To decode, the decoder needs to construct the same dictionary. To do this, he first takes the binary string he receives, and inserts dividers and commas. This is straightforward. The first two dividers each come after 2 bits. The next two each come after 3 bits. We then get 2^2 of length 4 bits, 2^3 of length 5 bits, 2^4 of length 6 bits, and in general 2^k of length $k + 2$ bits. This is because when we encode our phrases, if we have r phrases in our dictionary, we use $\lceil \log_2 r \rceil$ bits to encode the number of the phrase to ensure that the encodings of all phrases use the same number of bits. You can see that in the example above, the first time we encode AB (phrase 2) we encode it as 10, and the second time we encode it as 010. This is because the first time, we had three phrases in our dictionary, and the second time we had five.

The decoder then uses the same algorithm to construct the dictionary as the encoder did. He knows phrases 1 through $r-1$ when he is trying to figure out what the rth phrase is, and this is exactly the information he might need to reconstruct the dictionary.

You might notice that in this case, the compression algorithm actually makes the sequence longer. This is the case for one of two reasons. Either this original sequence was too random to be compressed much, or it was too short for the asymptotic efficiency of Lempel-Ziv to start being noticeable.

How well have we encoded the string? Suppose we have broken it up into $c(n)$ phrases, where n is the length of the string. Each phrase is broken up into a reference to a previous phrase and to a letter of our alphabet. The previous phrase can be represented by at most $\lceil \log_2 c(n) \rceil$ bits, since there are $c(n)$ phrases, and the letter can be represented by at most $\lceil \log_2 \alpha \rceil$ bits, where α is the size of the alphabet (in the above example, it is 2). We have thus used at most

$$c(n)(\log_2 c(n) + \log_2 \alpha)$$

bits total in our encoding.

In practice, you don't want to use too much memory for your dictionary. Thus, most implementations of Lempel-Ziv type algorithms have some maximum size for the dictionary. When it gets full, they drop a little-used word from the dictionary and replace it by the current word. This also helps the algorithm adapt to messages with changing characteristics. You need to use some deterministic algorithm for which word to drop, so that both the sender and the receiver will drop the same word.

So how well does the Lempel-Ziv algorithm work? In this lecture, we will calculate how well it works in the worst case, It actually also works well both in the random case where each letter of the message is chosen uniformly and independently from a probability distribution, as well as in the case where the letters of the message are chosen with a more sophisticated random process called a Markov chain (don't worry if you don't know what this is). It also works well for a lot of data that occur in practice, although it is not guaranteed to be asymptotically optimal for all types of sources. In all three of these cases, the compression is asymptotically optimal. That is, in the worst case, the length of the encoded string of bits is $n + o(n)$. Since there is no way to compress all length-n strings to fewer than n bits, this can be counted as asymptotically optimal. In the second case, the source is compressed to length

$$H(p_1, p_2, \dots, p_\alpha)n + o(n) = n\sum_{i=1}^{\alpha}(-p_i \log_2 p_i) + o(n),$$

which is to first order the Shannon bound.

Let's do the worst case analysis first. Suppose we are compressing a binary alphabet. We ask the question: what is the maximum number of distinct phrases that a string of length n can be parsed into. There are some strings which are clearly worst case strings. These are the ones in which the phrases are all possible strings of length at most k. For example, for $k = 1$, one of these strings is

$$0|1$$

with length 2. For $k = 2$, one of them is

$$0|1|00|01|10|11$$

with length 10; and for $k = 3$, one of them is

$$0|1|00|01|10|11|000|001|010|011|100|101|110|111$$

with length 34. In general, the length of such a string is

$$n_k = \sum_{j=1}^{k} j2^j$$

since it contains 2^j phrases of length j. It is easy to check that

$$n_k = (k-1)2^{k+1} + 2$$

by induction [This is an exercise.] If we let $c(n_k)$ be the number of distinct phrases in this string of length n_k, we get that

$$c(n_k) = \sum_{i=1}^{k} 2^i = 2^{k+1} - 2$$

For n_k, we thus have

$$c(n_k) = 2^{k+1} - 2 \le \frac{(k-1)2^{k+1}}{k-1} \le \frac{n_k}{k-1}$$

Now, for an arbitrary length n, we can write $n = n_k + \Delta$. To get the case where $c(n)$ is largest, the first n_k bits can be parsed into $c(n_k)$ distinct phrases, containing all phrases of length at most k, and the remaining Δ bits can be parsed into into phrases of length $k+1$. This is clearly the most distinct phrases a string of length n can be parsed into, so we have that for a general string of length n, the number of phrases is at most total is

$$c(n) \leq \frac{n_k}{k-1} + \frac{\Delta}{k+1} \leq \frac{n_k + \Delta}{k-1} = \frac{n}{k-1} \leq \frac{n}{\log_2 c(n) - 3}$$

Now, we have that a general bit string is compressed to around $c(n) \log_2 c(n) + c(n)$ bits, and if we subsitute

$$c(n) \leq \frac{n}{\log_2 c(n) - 3}$$

we get

$$c(n) \log_2 c(n) + c(n) \leq n + 4c(n) = n + O(\frac{n}{\log_2 n})$$

So asymptotically, we don't use much more than n bits for compressing any string of length n. This is good: it means that the Lempel-Ziv algorithm doesn't expand any string by very much. We can't hope for anything more from a general compression algorithm, as it is impossible to compress all strings of length n into fewer than n bits. So if a lossless compression algorithm compresses some strings to fewer than n bits, it will have to expand other strings to more than n bits. [Lossless here means the uncompressed string is exactly the original message.]

The Lempel-Ziv algorithm also works well for messages that are generated by the random process where each letter α_i is generated independently with some probability p_i. We don't have enough probability theory or time to give the proof in this lecture, but maybe we will get to it later. A more complicated version of the proof shows that the Lempel-Ziv algorithm also works for messages that are generated by probabilistic processes with limited memory. This means that the probability of seeing a given letter may depend on the previous letters generated, but this probability mainly depends on letters that were generated recently. This kind of process seems to reflect real-world sequences pretty well, at least in that the Lempel-Ziv family of algorithms works very well on a lot of real-world sequences.

All the compression algorithms we've talked about so far are lossless compression algorithms. This means that the reconstructed message is exactly the same as the original message. For many real-world processes, lossy compression algorithms are adequate, and these can often achieve much better compression ratios than lossless algorithms. For example, if you want to send video or music, it's generally not worth it to retain distinctions which are invisible or inaudible to our senses. Lossy compression thus gets much more complicated, because in addition to the mathematics, you have to figure out what kinds of differences can be distinguished by human eyes or ears, and then find an algorithm which ignores those kinds of differences, but doesn't lose significant differences. (And it may even be useful to have compression algorithms that reduce the quality of the signal.)