Up to now, we have looked at how some symbols in an alphabet occur more frequently than others and how we can save bits by using a code such that the codewords for more frequently occuring symbols are shorter than the codewords for less frequently occuring symbols. The methods we have seen still ignore several common types of probabilistic structure that is found in real sequences, however, namely interactions between the occurences of different symbols in an alphabet. In English, a `q` is almost always followed by a `u` for example. (The word `Iraq` is one exception, since in that case `q` is followed by a blank.) The next few methods we examine will address these interactions.

Lempel-Ziv methods

Lempel-Ziv methods are named after two researchers, Lempel and Ziv, who invented them and who provided the first proofs on their performance.

Suppose we have a sequence of n symbols drawn from some alphabet $\{A_1, A_2, \ldots, A_N\}$. Let X_j stand for the j^{th} symbol in the sequence. (Later we will think of X_j as a random variable. For now, just treat it as a variable which has a particular value, namely some symbol in the alphabet.)

LZ version 1 (LZ1)

Here's the basic Lempel-Ziv idea which I'll call LZ1. Suppose we have already encoded the first $j-1$ symbols in the sequence $(X_1, X_2, \ldots, X_{j-1})$ for some j, using some as yet unspecified scheme. Now we want to encode the rest of the sequence, $(X_j, \ldots, X_n)$. We do so by finding the largest subsequence of $(X_j, \ldots, X_n)$ that begins at index j and that matches a subsequence that begins *prior to* position j. As long as this largest matching subsequence has length greater than or equal to one, then we don't need to code it, since we have seen it already! Instead we can just write down how many symbols long is the match, and where the matching subsequence began.

There are two cases:

1. The symbol X_j did not occur in $(X_1, X_2, \ldots, X_{j-1})$. In this case, we encode that the length is 0, and we send a code of that symbol. (We need to have a code for each of the symbols of the alphabet.)

2. The symbol X_j did occur in $(X_1, X_2, \ldots, X_{j-1})$. In this case, we encode the *length* of the matching subsequence (> 0) and the *offset*. The offset is the number of symbols back in the sequence where this matching subsequence begins i.e. If the subsequence begins at symbol, X_{j-m}, then the offset is m. The length is just the length of the longest matching subsequence.

Note that we could have done it slightly differently, coding the offset first (with 0 offset meaning that there was no previous matching substring) and then encoding either the length (offset > 0) or the new symbol (offset $= 0$).

A matching subsequence is called a *phrase*. The idea of Lempel-Ziv methods is to partition the sequence into phrases and to encode each phrase. In case 1 above, the phrase has only one symbol, namely the new symbol that has not occured before. In case 2, the phrase has 1 or more symbols.

To implement the above coding method, we need to have three codes:

- C_1: a code for the length of best match, defined on $\{0, 1, 2, 3, \ldots\}$
- C_2: a code for the individual symbols, defined on $\{A_1, A_2, \ldots\}$

- C_3: a code for the offsets, defined on $\{1, 2, 3, \dots\}$

Notice that the code for the lengths is defined on 0 as well as the positive integers, whereas the code for offsets is defined only on the positive integers.

Example 1: "meet me at the theatre"

length	*symbol*	*offset*
0	`m`	
0	`e`	
1		1
0	`t`	
0	blank	
2		5
1		3
0	`a`	
2		6
1		2
0	`h`	
2		7
3		4
2		10
0	`r`	
1		4

The sequence is encoded:

$$C_1(0)C_2(\texttt{m})C_1(0)C_2(\texttt{e})C_1(1)C_3(1)C_1(0)C_2(\texttt{t})C_1(0)C_2(\mathit{blank})C_1(2)C_3(5)C_1(1)C_3(3)C_1(0)C_2(\texttt{a})\dots$$

The decoder can decode this sequence of bits, provided it knows the codes C_1, C_2, C_3, and provided it knows that each phrase will be encoded as a pair: either (length,symbol) or (length,offset), depending on whether length is zero or greater than zero.

Example 2: "`baaaaaaab...`"

The second example is more subtle:

length	*symbol*	*offset*
0	`b`	
0	`a`	
6		1

The subtlety here is that the best matching subsequence (starting at X_j) must itself start in $X_1 \dots X_{j-1}$, but it needn't be restricted to $X_1 \dots X_{j-1}$.

(This might seem like a contrived example. However, in cases where you have lots of blank characters in a file, this phenomenon comes up quite alot.)

Let's make some general observations about this algorithm.

- Each symbol in the alphabet is coded at most once (using code C_2) so we don't need to worry much about efficiency for this code.
- The lengths tend to be small numbers. While it is common to have words or sequences of words repeated in an English text (e.g. "in the morning"), it is rare that these repeated sequences are more than say 50 characters long. This would be considered bad style.
- The offsets tend to be large. For English text, words might be repeated throughout the text, and they may even come in clusters, but phrases are typically separated by hundreds or even thousands of positions. Thus, *we don't want to use the same code for offsets and we do for lengths of match!*

Lempel-Ziv version 2 (LZ2): sliding window

One of the issues with LZ1 above is that it might use too many bits to encode the offset. Since the best matching phrase can begin anywhere in $(X_1, X_2, \ldots, X_{j-1})$, the offset can be any number from 1 to j.

One simple method to avoid the problem, which I'll call LZ2, is to restrict the search to a finite size window. The window is of size n_w symbols, where n_w is a power of 2, typically 2^{15} or so. The algorithm searches backwards only over the window $(X_{j-n_w}, \ldots, X_{j-1})$ rather than all the way back to the beginning of the sequence. Of course, if $j < n_w$ then it only searches back to the beginning of the sequence.

This version of the algorithm is called "sliding window" because the window of n_w symbols is always the n_w symbols that come before the next symbol(s) to be coded.

Again need to encode offsets, lengths, and symbols. Suppose that n_w and N are both powers of 2. We could use:

- length of match, L: use a Golomb or Elias code defined in an earlier lecture to encode $L + 1$. (It is possible that $L = 0$.) This uses shorter codewords for shorter length matches, since shorter match lengths tend to be more likely in practice.
- offset: use $\log n_w$ bits (fixed length code)
- symbols: use $\log N$ bits (fixed length code)

Lempel-Ziv method 3 (LZ3)

The next LZ method we consider, which I will refer to as LZ3, takes a different approach. Rather than searching for the longest match to any subsequence starting in $X_1, \ldots, X_{j-1}$, LZ3 instead searches for the longest match to any *phrase* in $X_1, \ldots, X_{j-1}$. Once it finds this match, the next phrase (starting at X_j) is the longest matching phrase just found, concatenated with the next (unmatched) symbol. LZ3 does not encode the offset; instead it encodes the phrase number. LZ3 builds a tree of phrases (see below).

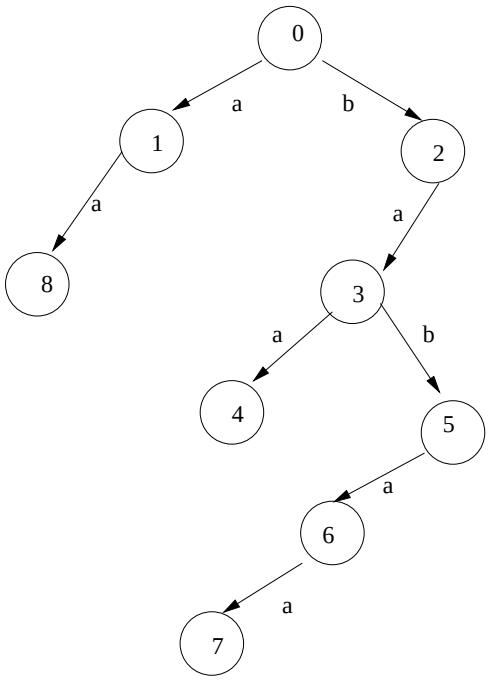
Note that the number of phrases in $X_1, \ldots, X_{j-1}$ tends to be much less than the number $j - 1$ of symbols that have appeared. Thus, to encode the best matching phrase typically requires far fewer bits than to encode the arbitrary offset as in LZ1.

Example

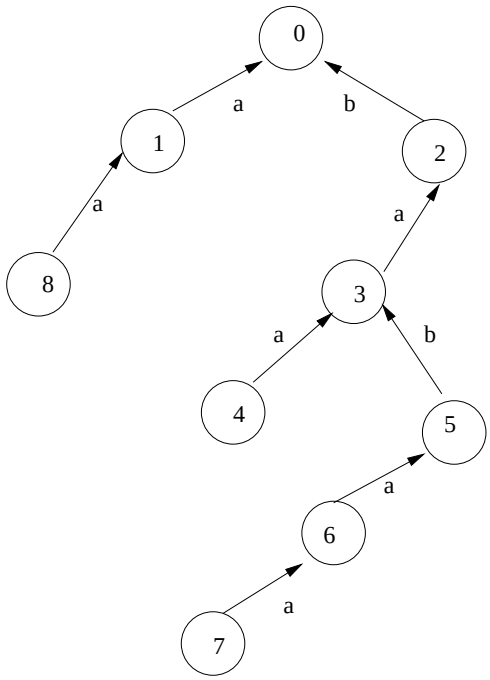
Let's consider an example. Take the sequence `abbabaababbababababaaaab`. We parse it as follows: `a,b,ba,baa,bab,baba,babaa,aa,b`.

As the encoder parses the sequence, it builds the tree shown on the left below.

Note that the final phrase terminates at an interior node in the tree. (It could be a root node, or a non-root internal node.) How do we deal with this case? One simple way would be to encode the current number of phrases parsed plus 1. (Call this phrase_ct + 1.) The decoder then knows that the encoder has reached the final phrase. Once this fact has been encoded, the encoder could give the internal node at which this final phrase terminates. This could be anywhere from node 0 to node phrase_ct.



ENCODER TREE



DECODER TREE

For the above example, the encoder would send the following codewords. Let C be the code for numbers and let C^* be the code for symbols.

$C(0)\, C^*(\texttt{a})\, C(0)\, C^*(\texttt{b})\, C(2)\, C^*(\texttt{a})\, C(3)\, C^*(\texttt{a})\, C(3)\, C^*(\texttt{b})\, C(5)\, C^*(\texttt{a})\, C(6)\, C^*(\texttt{a})\, C(1)\, C^*(\texttt{a})\, C(9)\, C(2)$

Encoding algorithm (LZ3)

```
phrase_ct = 0;
root = empty;
while not endoffile
   phrase_ct = phrase_ct + 1;
   traverse tree to find the longest matching prefix phrase;
```

```
   if end of file
      encode (phrase_ct, index of matching prefix phrase)
      // if no match, then longest matching phrase is root i.e.  phrase 0)
   else
      encode (index of longest matching prefix phrase, next non-matching symbol);
      insert a child (phrase_ct, next symbol) below leaf of longest matching prefix;
   endif
end while
```

LZ1 and LZ2 vs. LZ3: number of phrases

Let's briefly compare the parsings of LZ1 and LZ2 vs. LZ3 in terms of the number of phrases.

Example: `aaaaaaaaaaaaaaaaaaaaaaa`...

LZ1/2 would parse the sequence as `a,aaaaaaaaaaaaaaaaaaaaaaa`..., whereas LZ3 would parse it as `a,aa,aaa,aaaa,aaaaa,aaaaaa,aaa`.... Thus, LZ2 has fewer phrases for this example.

You might suspect that LZ2 always gives fewer phrases than LZ3 since LZ2 can match to any previous sequence. However, this intuition turns out to be false, as the following example shows.

Example: ``meet_me_at_the_theatre''

LZ2: `m,e,e,t,_,me,_,a,t_,t,h,e_,the,at,r,e` (16 phrases)

LZ3: `m,e,et,_,me,_a,t,_t,h,e_,th,ea,tr,e` (14 phrases)

What's going on here? Although LZ3 can only match to previously seen phrases, when it does match it *adds a symbol* the length of the best match. In this sense, the phrases of LZ3 can be longer than those of LZ2.

LZ3 decoder

Instead of constructing a tree, the LZ3 decoder constructs a table. The table consists of a set of pairs: `(index to longest matching prefix phrase, next symbol)`. These are the pairs that the encoder had encoded.

Note that the entries in the table define a binary tree – similar to the one used by the encoder – except that now the branches point upwards from child to parent, rather than downward from parent to child. (See **DECODER TREE** on previous page.)

Here is what the table would look like for the above example `abbabaababbababababaaaab` where the * indicates that a different coding method is used for the final phrase in the sequence.

index	index of parent/prefix	last symbol
0	null	
1	0	a
2	0	b
3	2	a
4	3	a
5	3	b
6	5	a
7	6	a
8	1	a
9	9^*	2^*

Make sure that you understand how the decoder can use the table to reconstruct the original sequence. In particular, note that the prefixes are reconstructed backwards, since the decoder traverses the tree/table from leaf to root.