There is a tricky detail in decoding LZW:

- In order to insert the phrase $Z_{\sigma+j} = T_j t_{i+|T_j|}$ into the dictionary, the decoder needs to know the symbol $t_{i+|T_j|}$. For the encoder this is not a problem, but the decoder only knows that $t_{i+|T_j|}$ is the first symbol of the next phrase T_{j+1}. Thus the insertion is delayed until the next phrase has been decoded.

- The encoder inserts $Z_{\sigma+j}$ to the dictionary without a delay and has the option of choosing $T_{j+1} = Z_{\sigma+j}$. If this happens, the decoder is faced with a reference to a phrase that is not yet in the dictionary! However, in this case the decoder knows that the unknown symbol $t_{i+|T_j|}$ is the first symbol of $Z_{\sigma+j}$, which is the same as the first symbol of T_j. Given $t_{i+|T_j|}$ the decoder can set $Z_{\sigma+j} = T_j t_{i+|T_j|}$, insert it into the dictionary, and decode T_{j+1} normally.

The phrase $T_{j+1} = Z_{\sigma+j}$ is problematic because it is *self-referencial*.

Example 2.9: In Example 2.8, the phrase $T_5 = Z_8 = \texttt{ada}$ is self-referential.

LZ77

The original LZ77 algorithm works as follows:

- A phrase T_j starting at a position i is encoded as a triple of the form $\langle$distance, length, symbol$\rangle$. A triple $\langle d, l, s\rangle$ means that:
$$T_j = T[i...i+l] = T[i-d...i-d+l)s$$
In other words, the string $T[i..i+l)$ of length l has another occurrence d positions earlier in the text. A decoder can simply copy the string from the already decoded part of the text.
- The values d and l should satisfy $d \in [1..d_{\max}]$ and $l \in [0..l_{\max}]$. In other words, the earlier occurrence should be no longer than $l_{\max}$ and should start within a *window* $T[i-d_{\max}..i-1]$. If $l > d$, the strings $T[i..i+l)$ and $T[i-d...i-d+l)$ overlap, which makes the phrase *self-referential*.
- The algorithm searches the window for the longest possible match under the above contraints, i.e., it tries to maximize l.
- The triple is encoded with fixed length codes using $\lceil \log d_{\max} \rceil + \lceil \log(l_{\max}+1) \rceil + \lceil \log \sigma \rceil$ bits.

Example 2.10: Let $T = \texttt{badadadabaab}$ and assume $d_{\max}$ and $l_{\max}$ are large.

phrase	b	a	d	adadab	aa	b
encoding	$\langle 0,0,\texttt{b}\rangle$	$\langle 0,0,\texttt{a}\rangle$	$\langle 0,0,\texttt{d}\rangle$	$\langle 2,5,\texttt{b}\rangle$	$\langle 2,1,\texttt{a}\rangle$	$\langle 0,0,\texttt{b}\rangle$

Later variants have improved the encoding of the phrases:

- As with LZ78, having the extra symbol at the end of each phrase is wasteful, but there must still be a way to encode symbols that do not occur in the window. A common solution is to have two types of codes: $\langle$length, distance$\rangle$ and $\langle$symbol$\rangle$. The second type of code may also be the more efficient code for a phrase of length one even if it occurs in the window.

- Use variable length codes instead of fixed length codes. For example, we could use a semiadaptive Huffman code with some codewords representing symbols and others representing lengths. A length is always followed by a distance encoded using Golomb–Rice code. The most advanced compressors can further have a complex model to estimate the probability of each bit in the encoding and then use arithmetic coding to encode them.

Many popular compression programs, such as `zip`, `gzip` and `7zip`, use this types of LZ77 variants.

Another way to optimize LZ77 is to replace the exhaustive search of the window with an efficient data structure.

- Many different data structures including binary trees, hash tables and suffix trees have been used for the purpose.

- Fast searching enables larger window sizes or even unbounded distances and lengths. Increasing the window size can lead to longer phrases and thus better compression.

- On the other hand, the compression can suffer from longer codes needed for larger values. With the fixed length codes of the original LZ77, this is another reason to use a small window. With variable length codes, a small upper bound is not important, but a smaller distance has a shorter code. Thus the longest possible phrase is not always the optimal choice.

- A recent algorithm by Ferragina, Nitto and Venturini (SODA 2009) solves this optimization problem efficiently for many encoding schemes, i.e., it finds the parsing that minimizes the total length of the final encoding.

LZFG

As the final LZ-type compression method let us briefly look at LZFG that is a kind of hybrid of LZ77 and LZ78 algorithms:

- LZFG is like LZ77 but with the restriction that the earlier occurrence of each phrase has to begin at a previous *phrase boundary*. There is no restriction on the end of the phrase.
- Each phrase is encoded as a ⟨length, distance⟩ pair, but the distance is now measured in phrases not symbols. The positions of all phrase boundaries are recorded in a separate array and the distance values point to this array rather than directly to the text. In this sense, LZFG is an LZ78 type method.
- Using a large or unbounded window size is easier with LZFG than with LZ77, because the distance values are smaller and the data structures for finding phrases are simpler.

Example 2.11: Let $T = \texttt{badadadabaab}$. Assume two types of codes, ⟨length, distance⟩ and ⟨symbol⟩, and no length or distance limits.

phrase	b	a	d	adada	ba	a	b
encoding	⟨b⟩	⟨a⟩	⟨d⟩	⟨5, 2⟩	⟨2, 4⟩	⟨a⟩	⟨b⟩

An important attribute of Lempel–Ziv compression methods is the size of their effective dictionary, i.e, the number of possible distinct phrases. For a text of length n with a parsing of size z, the effective dictionary sizes are bounded by:

LZ78	LZW	LZ77 (original)	LZ77 (variant)	LZFG
$(z+1)\sigma$	$\sigma + z$	$d_{\mathsf{max}}(l_{\mathsf{max}}+1)\sigma$	$n^2+\sigma$	$zn+\sigma$

In general, the effective dictionary size of LZ78 type algorithms grows slowly with n, while LZ77 type algoritms can have a much faster growth rate.

- A larger dictionary usually leads to longer and thus fewer phrases. This does not necessarily mean better compression, because the code sizes increase too, but fewer phrases can speed up decoding.
- A faster dictionary growth rate can improve compression significantly on some highly compressible texts (exercise).

Grammar compression

A special type of *semiadaptive* dictionary compression is grammar compression that represents a text as a *context-free grammar*.

Example 2.12: $T = \texttt{a␣rose␣is␣a␣rose␣is␣a␣rose}$.

$$S \to ABB$$
$$A \to \texttt{a␣rose}$$
$$B \to \texttt{␣is␣}A$$

The grammar should generate exactly one string. Such a grammar is called a *straight-line* grammar because of the following properties:

- The are *no branches*, i.e., each non-terminal is the left-hand size of only one rule. Otherwise, multiple strings could be generated.
- There are no loops, i.e., no cyclic dependencies between non-terminals. Otherwise, infinite strings could be generated.

A straight-line grammar in Chomsky normal form is called a straight-line program.

The size of a grammar is the total length of the right-hand sides. The *smallest grammar problem* of computing the smallest straight-line grammar that generates a given string is NP hard. Nevertheless, there are many algorithms for constructing small grammars for a text, for example:

- LZ78 parsing is easily transformed into a grammar with one rule for each phrase.
- The best approximation ratio $\mathcal{O}(\log(n/g))$, where g is the size of the smallest grammar, has been achieved by algorithms that transform an LZ77 parsing into a grammar.
- Greedy algorithms add one rule at a time as long as they can find a rule that reduces the size of the grammar. The right hand side of each new rule is chosen greedily by some criterion, for example:
  - the longest substring with at least two occurrences
  - the most frequent substring of length at least two
  - the string that produces the biggest reduction in size.

Re-Pair

Re-Pair is a greedy grammar compression algorithm that operates as follows:

1. Find the pair of symbols XY that is the most frequent in the text T. If no pair occurs twice in T, stop.
2. Create a new non-terminal Z and add the rule $Z \to XY$ to the grammar. Replace all occurrences of XY in T by Z. Go to step 1.

The whole process can be performed in *linear time* using suitable data structures. The details are omitted, but the key observation is that, if n_{XY} is the number of occurrences of the most frequent pair XY in a given step, then the replacement reduces the size of the grammar by $n_{XY} - 2$. Thus we can spend $\mathcal{O}(n_{XY})$ time to perform the step.

Here is a simple encoding of the final result:

- The number r of rules and the length z of the compressed text in γ code.
- The right-hand sides of rules using $\lceil \log(\sigma + i - 1) \rceil$-bit fixed length codes to encode the ith rule.
- The compressed text using $\lceil \log(\sigma + r) \rceil$-bit fixed length codes.

Better compression can be achieved with a more sophisticated encoding.

Example 2.13: $T = \texttt{singing\textvisiblespace do\textvisiblespace wah\textvisiblespace diddy\textvisiblespace diddy\textvisiblespace dum\textvisiblespace diddy\textvisiblespace do}$.

rule added	text after replacement
$A \to \texttt{\textvisiblespace d}$	$\texttt{singing}A\texttt{o\textvisiblespace wah}A\texttt{iddy}A\texttt{iddy}A\texttt{um}A\texttt{iddy}A\texttt{o}$
$B \to \texttt{dd}$	$\texttt{singing}A\texttt{o\textvisiblespace wah}A\texttt{i}B\texttt{y}A\texttt{i}B\texttt{y}A\texttt{um}A\texttt{i}B\texttt{y}A\texttt{o}$
$C \to A\texttt{i}$	$\texttt{singing}A\texttt{o\textvisiblespace wah}CB\texttt{y}CB\texttt{y}A\texttt{um}CB\texttt{y}A\texttt{o}$
$D \to B\texttt{y}$	$\texttt{singing}A\texttt{o\textvisiblespace wah}CDCDA\texttt{um}CDA\texttt{o}$
$E \to CD$	$\texttt{singing}A\texttt{o\textvisiblespace wah}EEA\texttt{um}EA\texttt{o}$
$F \to \texttt{in}$	$\texttt{s}F\texttt{g}F\texttt{g}A\texttt{o\textvisiblespace wah}EEA\texttt{um}EA\texttt{o}$
$G \to A\texttt{o}$	$\texttt{s}F\texttt{g}F\texttt{g}G\texttt{\textvisiblespace wah}EEA\texttt{um}EG$
$H \to F\texttt{g}$	$\texttt{s}HHG\texttt{\textvisiblespace wah}EEA\texttt{um}EG$

A common feature of most dictionary compression algorithms is asymmetry of compression and decompression:

- The compressor needs to do a lot of work in choosing the phrases or rules.
- The decompressor only needs to replace each phrase.

Thus the decompressor is often simple and fast.

- LZ77-type methods are particularly simple and fast as they have no dictionary other than the already decoded part of the text.
- LZ78-type and grammar-based methods need some extra effort in constructing and accessing the dictionary.

The best possible compression may require a complex model and arithmetic coding to encode the phrase, which can make the decoder dramatically slower. Thus many implementations are optimized more for speed than maximum compression.