Example 2.6: Let $T = \mathtt{badadabada}$ and $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The frequencies of symbols in first order contexts are:

	a	b	c	d
a	0/5	2/5	0/5	3/5
b	2/2	0/2	0/2	0/2
c	0/0	0/0	0/0	0/0
d	3/3	0/3	0/3	0/3

The first order empirical entropy is

$$H_1(T) = -\frac{5}{10}\left(\frac{2}{5}\log\frac{2}{5} + \frac{3}{5}\log\frac{3}{5}\right) \approx 0.485$$

while $H_0(T) \approx 1.49$.

(When computing entropies, terms involving zero and one probabilities can be ignored since $0 \log 0 = 1 \log 1 = 0$.)

The semiadaptive and adaptive (count+1) models assign the following frequencies to the symbols:

	b	a	d	a	d	a	b	a	d	a
semiadaptive	1/4	2/2	3/5	3/3	3/5	3/3	2/5	2/2	3/5	3/3
adaptive	1/4	1/4	1/4	1/4	2/5	2/5	1/6	2/5	3/7	3/6

A problem with kth order models is the choice of the context length k. If k is too small, the compression rate is not as good as it could be. If k is too large, the overhead components start to dominate:

- the storage for the counts in semiadaptive compression
- the poor compression in the beginning for adaptive compression.

Furthermore, there may not exist a single right context length and a different k should be used at different places.

One approach to address this problem is an adaptive method called *prediction by partial matching (PPM)*:

- Use multiple context lengths $k \in \{0, \ldots, k_{\max}\}$. To encode a symbol t_i, find its longest context $T[i-k..i-1]$ that has occurred before.
- If t_i has not occurred in that context, encode the escape symbol similarly to what we saw with zeroth order compression. Then reduce the context length by one, and try again. Keep reducing the context length until t_i can be encoded in that context. Each reduction encodes a new escape symbol.

There are further details and variations.

There are other complex modelling techniques such as *dynamic Markov compression* that dynamically builds an automaton, where each state represents a certain set of contexts.

The most advanced technique is *context mixing*. The idea is to use multiple models and combine their predictions. The variations are endless with respect to what models are used and how the predictions are combined. Context mixing compressors are at the top in many compression benchmarks.

Many of the most complex techniques operate on a binary source alphabet, which simplifies both modelling and encoding using arithmetic coding. Texts with a larger alphabet are often transformed into binary using a fixed-length encoding. An alternative is to use a simple static or semiadaptive Huffman coding.

Dictionary compression

The compression techniques we have seen so far replace individual symbols with a variable length codewords. In dictionary compression, variable length substrings are replaced by short, possibly even fixed length codewords. Compression is achieved by replacing long strings with shorter codewords.

The general scheme is as follows:

- The *dictionary* $\mathcal{D}$ is a collection of strings, often called *phrases*. For completeness, the dictionary includes all single symbols.
- The text T is *parsed* into a sequence of phrases:

  $$T = T_1 T_2 \ldots T_z,\ T_i \in \mathcal{D}.$$

  The sequence is called a *parsing* or a *factorization* of T with respect to $\mathcal{D}$.
- The text is encoded by replacing each phrase T_i with a code that acts as a pointer to the dictionary.

Here is a simple static dictionary encoding for English text:

- The dictionary consists of some set of English words plus individual symbols.
- Compute the frequencies of the words in some corpus of English texts. Compute the frequencies of symbols in the corpus from which the dictionary words have been removed.
- Number the words and symbols in descending order of their frequencies.
- To encode a text, replace each dictionary word and each symbol that does not belong to a word with its corresponding number. Encode the sequence of number using γ coding.

Lempel-Ziv compression

In 1977 and 1978, Abraham Lempel and Jacob Ziv published two adaptive dictionary compression algorithms that soon became to dominate practical text compression. Numerous variants have been published and implemented, and they are still the most commonly used algorithms in general purpose compression tools.

The common feature of the two algorithms and all their variants is that the dictionary consists of substrings of the already processed part of the text. This means that the dictionary adapts to the text.

The two algorithms are known as LZ77 and LZ78, and most related methods can be categorized as a variant of one or the other. The primary difference is the encoding of the phrases:

- LZ77 uses direct pointers to the preceding text.
- LZ78 uses pointers to a separate dictionary.

LF78

The original LZ78 encoding works as follows:

- The dictionary consists of phrases numbered from 0 upwards: $\mathcal{D} = \{Z_0, Z_1, Z_2, \ldots\}$. Each new phrase inserted to the dictionary gets the next free number. Initially, the only phrase is the empty string $Z_0 = \epsilon$.
- Suppose we have so far computed the parsing $T_1 \ldots T_{j-1}$ for $T[0 \ldots i)$ and the next phrase T_j starts at position i. Let $Z_k \in \mathcal{D}$ be the longest phrase in the dictionary that is a prefix of $T[i \ldots n-1)$. The next phrase is $T_j = Z_j = T[i \ldots i + |Z_k|] = Z_k t_{i+|Z_k|}$, and it is inserted to the dictionary.
- The phrase T_j is encoded as the pair $\langle k, t_{i+|Z_k|} \rangle$. Using fixed length codes, the pair needs $\lceil \log(j+1) \rceil + \lceil \log \sigma \rceil$ bits.

Example 2.7: Let $T =$ `badadadabaab`.

index	0	1	2	3	4	5	6	7
phrase	ϵ	b	a	d	ad	ada	ba	ab
encoding		$\langle 0, \mathtt{b} \rangle$	$\langle 0, \mathtt{a} \rangle$	$\langle 0, \mathtt{d} \rangle$	$\langle 2, \mathtt{d} \rangle$	$\langle 4, \mathtt{a} \rangle$	$\langle 1, \mathtt{a} \rangle$	$\langle 2, \mathtt{b} \rangle$
code length		0+2	1+2	2+2	2+2	3+2	3+2	3+2

LZW

LZW is simple optimization of LZ78 used, e.g., in the unix tool `compress`.

- Initially, the dictionary $\mathcal{D}$ contains all individual symbols: $\mathcal{D} = \{Z_1, \ldots, Z_\sigma\}$.
- Suppose the next phrase T_j starts at position i. Let $Z_k \in \mathcal{D}$ be the longest phrase in the dictionary that is a prefix of $T[i..n)$. Now the next text phrase is $T_j = Z_k$ and the phrase added to the dictionary is $Z_{\sigma+j} = T_j t_{i+|T_j|}$.
- The phrase T_j is encoded with the index k requiring $\lceil \log(\sigma + j - 1) \rceil$ bits.

Omitting the symbol codes saves space in practice, even though the index codes can be longer and the phrases can be shorter.

Example 2.8: Let $T =$ `badadadabaab`.

text phrase					b	a	d	ad	ada	ba	a	b
encoding					2	1	4	6	8	5	1	2
code length					2	3	3	3	3	4	4	4
dictionary phrase	a	b	c	d	ba	ad	da	ada	adab	baa	ab	
index	1	2	3	4	5	6	7	8	9	10	11	

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.