Decoding arithmetic code performs much the same process, but the process is now controlled by the code string and it produces the source string as output. Here is an outline of the decoding procedure in our example:

- We start with the code interval $[0, 64)$. Each bit read from the code string halves the code interval.

- In the beginning, the source interval is $[0, 64)$ too, and it is divided into three candidate intervals: $[0, 13)$, $[13, 45)$ and $[45, 64)$. Whenever the code interval falls within one of the candidate intervals, that interval becomes the new source interval and we output the corresponding symbol. If the new source interval is within one of the three halves of the unit interval, we zoom in scaling both the source and the code interval. Then the new source interval is divided into three candidate intervals and the process continues.

- The process ends, when n symbols have been output.

The full details are left as an exercise.

Summary

We have seen two encoding algorithms, Huffman coding and arithmetic coding, that perform the same task:

- Turn a sequence of symbols with a given probability distribution into a uniquely decodable binary code sequence.
- Try to minimize the average length of the code sequence.

This task is often known as *entropy encoding* reflecting the fact that entropy is the ideal average code length and the algorithms try to get as close to that as possible.

Huffman coding and arithmetic coding cannot be easily used when the alphabet is large or even infinite. Fixed codes such as γ or Golomb–Rice codes address this problem in the case of integer alphabet.

The Huffman algorithm computes the optimal prefix code. But there are still other algorithms for constructing prefix codes that can sometimes be useful. For example:

- The Shannon-Fano algorithm (Exercise 1.3) is simpler to implement than the Huffman algorithm, and usually produces an optimal or nearly optimal prefix code. See also Exercise 2.1.

- The package-merge algorithm constructs the optimal prefix code under the constraint that the codeword is longer than L. Such a code is sometimes known as a length-limited Huffman code.

- The Hu-Tucker algorithm constructs an optimal alphabetic prefix code, which has the property that the lexicographical order of codewords is the same as the order of the symbols they encode.

2. Text Compression

We will now look at techniques for text compression.

These techniques are particularly intended for compressing natural language text and other data with a similar sequential structure such as program source code. However, these techniques can achieve some compression on almost any kind of (uncompressed) data.

These are the techniques used by general purpose compressors such as `zip`, `gzip`, `bzip2`, `7zip`, etc..

Compression Models

Entropy encoding techniques such as Huffman coding and arithmetic coding (only) need a probability distribution as an input. The method for determining the probabilities is called a *model*. Finding the best possible model is the real art of data compression.

There are three types of models:

- static
- semiadaptive or semistatic
- adaptive.

A *static* model is a fixed model that is known by both the compressor and the decompressor and does not depend on the data that is being compressed. For example, the frequencies of symbols in English language computed from a large corpus of English texts could be used as the model.

A *semiadaptive* or *semistatic* model is a fixed model that is constructed from the data to be compressed. For example, the symbol frequencies computed from the text to be compressed can be used as the model. The model has to be included as a part of the compressed data.

- A semiadaptive model adapts to the data to be compressed. For example, a static model for English language would be inoptimal to compressing other languages or something completely different such as DNA sequencies. A semiadaptive model can always be the optimal one.
- The drawback of a semiadaptive model is the need to include the model as a part of the compressed data. For a short data, this could completely negate any compression achieved. On the other hand, for a very long data, the space needed for the model would be negligable. Sometimes it may possible to adapt the complexity of the model to the size of the data in order to minimize the total size. (According to the Minimum Descrition Length (MDL) principle, such a model is the best model in the machine learning sense too.)
- Every semiadaptive model can be seen as a static model with adaptable parameters. An extreme case would be to have two static models and a one bit parameter to choose between them.

An *adaptive* model changes during the compression. At a given point in compression, the model is a function of the previously compressed part of the data. Since that part of the data is available to the decompressor at the corresponding point in decompression, there is no need to store the model. For example, we could start compressing using a uniform distibution of symbols but then adjust that distribution towards the symbol frequencies in the already processed part of the text.

- Not having to store a model saves space, but this saving can be lost to a poor compression rate in the beginning of the compression. There is no clear advantage either way. As the compression progresses, the adaptive model improves and approaches optimal. In this way, the model automatically adapts to the size of the data.
- The data is not always uniform. An optimal model for one part may be a poor model for another part. An adaptive model can adapt to such local differences by forgetting the far past.
- The disadvantage of adaptive computation is the time needed to maintain the model. Decompression, in particular, can be slow compared with semiadaptive compression.

Zeroth order text compression

Let $T = t_0 t_2 \ldots t_{n-1}$ be a text of length n over an alphabet Σ of size σ. For any symbol $s \in \Sigma$, let n_s be the number of occurrencies of s in T. Let $f_s = n_s/n$ denote the frequency of s in T.

Definition 2.1: The *zeroth order empirical entropy* of the text T is

$$H_0(T) = -\sum_{s\in\Sigma} f_s \log f_s .$$

The quantity $nH_0(T)$ represents a type of lower bound for the compressed size of the text T.

- If we encode the text with arithmetic coding using some probability distribution P on Σ, the length of the encoding is about
$$-\sum_{s\in\Sigma} n_s \log P(s) = -n \sum_{s\in\Sigma} f_s \log P(s).$$
- The distribution P that minimizes the length of the encoding is $P(s) = f_s$ for all $s \in \Sigma$. Then the size of the encoding is $\geq nH_0(T)$ bits.

A simple *semiadaptive* encoding of T consists of:

- the symbol counts n_s encoded with γ-coding, for example,
- the text symbols encoded with Huffman or arithmetic coding using the symbol frequencies f_s as probabilities.

The size of the first part can be reduced slightly by encoding codeword lengths (Huffman coding) or rounded frequencies (low precision arithmetic coding) instead of the symbol counts.

The size of the second part is between $nH_0(T)$ and $nH_0(T) + n$ bits.

Note that $nH_0(T)$ bits is not a lower bound for the second part but for the whole encoding. The size of the second part could be reduced as follows:

- When encoding and, more importantly, when decoding t_i, the frequencies in $T[0..i)$ are known, and therefore the frequencies in $T[i..n)$ can be computed.

- Encoding the symbol t_i using the frequencies from $T[i..n)$ improves compression. For example, the last symbol t_{n-1} has a frequency of one and does not need to encoded at all.

However, the saving is usually too small to make it worth while.

As just noted, the frequencies in $T[0..i)$ are known when t_i is being encoded or decoded. By using those frequencies to encode t_i, we obtain an *adaptive* encoder. Then there is no need to store the symbol counts at all.

However, we have to deal with the *zero frequency problem.* If $t_i = s$ is the first occurrence of the symbol s in T, its frequency in $T[0..i)$ is zero. There are two ways to solve this problem:

- Add one to the counts of all symbols. Then a first occurrence t_i will have a frequency $1/(i + \sigma)$.

- Add a special *escape symbol* before each first occurrence. The symbol after an escape is assigned a uniform probability over the symbols that have not occurred (or simply a fixed length code such ASCII code). The escape symbol itself can can have a permanent count of one, for example.

Example 2.2: Let $T = \mathtt{badada}$ and $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The frequencies assigned to the symbols are:

	b	a	d	a	d	a
semi-adaptive	1/6	3/6	2/6	3/6	2/6	3/6
optimized semi-adaptive	1/6	3/5	2/4	2/3	1/2	1/1
adaptive (count+1)	1/4	1/5	1/6	2/7	2/8	3/9
adaptive (escape)	$1/1 \cdot 1/4$	$1/2 \cdot 1/3$	$1/3 \cdot 1/2$	1/4	1/5	2/6

When the symbol counts grow large, they can be rescaled by dividing by some constant. This has the effect of partially *forgetting* the past, since the symbols encountered after the rescaling have a bigger effect on the counts than those preceding the rescaling. This can make the model adapt better to local differences.

Adaptive entropy coders

The constantly changing frequencies of adaptive models require some modifications to the entropy coders. Assume that the model maintains symbol counts n_s, $s \in \Sigma$ as well as their sum $n = \sum_s n_s$.

For arithmetic coding, the changing probabilities are not a problem as such, but there are a couple of details to take care of:

- Arithmetic coding needs *cumulative* probabilities. Since a single increment of a count can change many cumulative counts, maintaining the cumulative counts can become expensive for a large alphabet. A simple optimization is to update the counts only every few steps.

- When using integer arithmetic, the update of the interval boundaries is changed into

  $$l \leftarrow l + \left\lfloor \frac{p \cdot l'}{n} \right\rfloor$$
  $$r \leftarrow l + \left\lfloor \frac{p \cdot r'}{n} \right\rfloor$$

  where the $[l'/n, r'/n)$ is the interval for the next symbol. In other words, the divider is now n instead of the 64 or another fixed value.

Making Huffman coding adaptive requires a more drastic change. It is based on the fact that a small change to the frequencies is likely to have only a small effect on the code. There exists an algorithm for efficiently updating the code after a small change.

The algorithm maintains the Huffman tree:

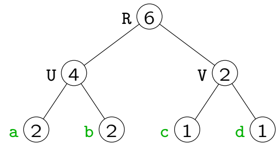
- The leaves store the symbol counts.
- Each internal node stores the sum of its children's counts.

All nodes are kept on a list in the order of their counts. Ties are broken so that siblings (children of the same parent) are next to each other. Whenever an increment causes a node u to be in a wrong place on this list, it is swapped with another node v:

- v is chosen so that after swapping u and v on the list, the list is again in the correct order.
- The subtrees rooted at u and v are swapped in the tree.

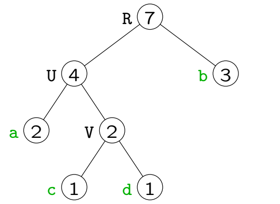
The increments start at a leaf and are propagated upwards only after a possible swap.

Example 2.3: Let $T = \mathtt{abb}\ldots$ and $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$.



Starting with all counts being one and a balanced Huffman tree, processing the first two symbols `ab` updates the counts but does not change the tree.

R	U	V	a	b	c	d
6	4	2	2	2	1	1



The third symbol `b` causes the swap of the nodes `b` and V.

R	U	b	a	V	c	d
6	4	3	2	2	1	1

It can be shown that this algorithm keeps the tree a Huffman tree. The swaps can be implemented in constant time. The total update time is proportional to the codeword length.

Higher order models

In natural language texts, there is a often a strong correlation between a letter and the neighboring letters. Zeroth order models take no advantage of this correlation. Higher order models do and thus can achieve much better compression.

Let $T = t_0 t_2 \ldots t_{n-1}$ be a text of length n over an alphabet Σ of size σ. Consider the text to be cyclic, i.e., $t_i = t_{i \bmod n}$ for any integer i.

The *kth order context* of a symbol t_i in T is the string $T[i-k..i-1]$. For any string $w \in \Sigma^*$, let n_w the number of occurrence of w in T (including those that cyclically overlap the boundary).

Example 2.4: If $T = \mathtt{badada}$, then $n_{\mathtt{da}} = 2$ and $n_{\mathtt{ab}} = 1$.

Definition 2.5: The *kth order empirical entropy* of the text T is

$$H_k(T) = -\sum_{w \in \Sigma^k} \frac{n_w}{n} \sum_{s \in \sigma} \frac{n_{ws}}{n_w} \log \frac{n_{ws}}{n_w}.$$

The value n_{ws}/n_w is the frequency of s in the context w.

A simple *semiadaptive* kth order encoding of T consists of:

- the symbol counts n_w for all $w \in \Sigma^{k+1}$ encoded with γ-coding, for example,
- the first k symbols of T using some simple encoding
- the rest of text symbols encoded with Huffman or arithmetic coding using as probabilities the symbol frequencies in their kth order context.

The size of the last part of the encoding is between $nH_K(T)$ and $nH_k(T) + n$ bits.

An *adaptive* kth order model maintains σ^k separate zeroth order models, one for each $w \in \Sigma^k$. Each symbol t_i is encoded using the model for the context $T[i-k..i-1]$ and then the model is updated just as in zeroth order compression.

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.